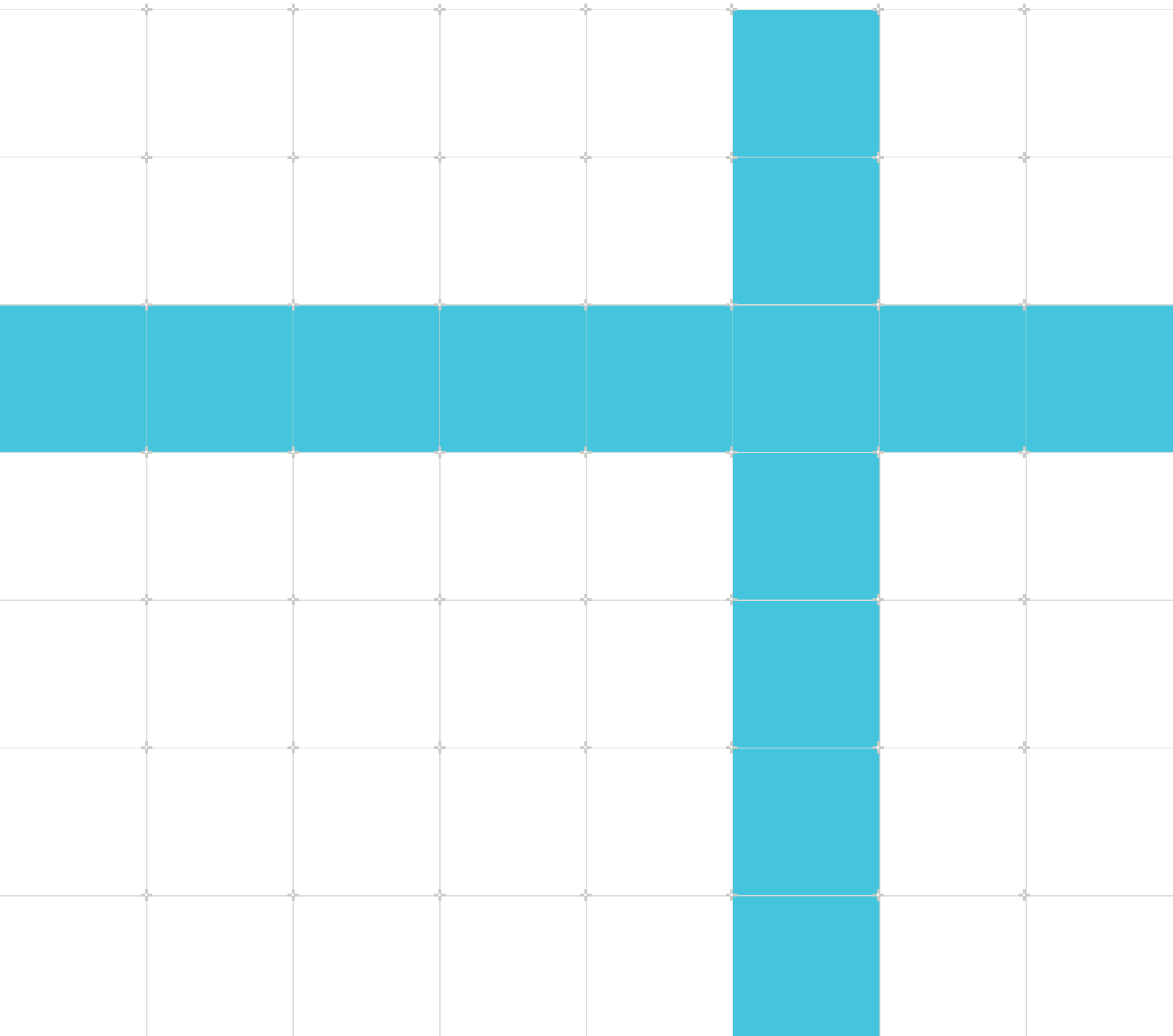
Learn the architecture - Memory Systems, Ordering, and Barriers

Version 1.0

Issue 01
102336_0100_01_en



Learn the architecture - Memory Systems, Ordering, and Barriers

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	10 June 2022	Non-Confidential	Initial release

Proprietary Notice

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. Overview

This guide introduces the memory ordering model that is defined by the Armv8-A architecture, and introduces the different memory barriers that are provided. This guide also identifies some common cases in which explicit ordering is required, and how to use memory barriers to ensure that correct operation is achieved.

This guide is suitable for developers of low-level code, such as boot code or drivers. It is particularly relevant to anyone writing code for multi-threaded applications or shared memory systems.

At the end of this guide you can Check your knowledge.

Before you begin

This guide assumes that you are familiar with the Arm memory types. If you are not, read about Device memory and Normal memory in the Armv8-A memory model guide.

2. Memory ordering

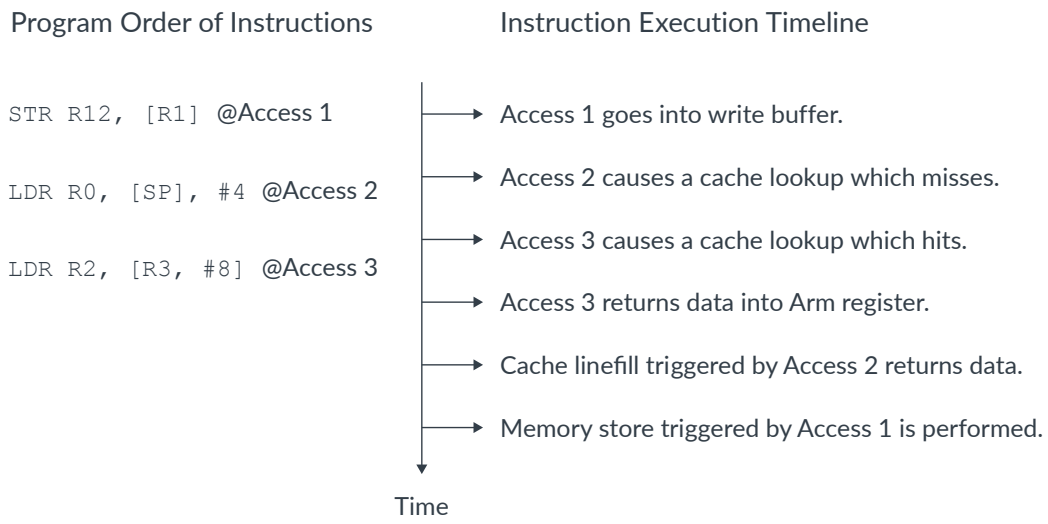
Armv8-A implements a weakly-ordered memory architecture. This architecture permits memory accesses which impose no dependencies to be issued or observed, and to complete in a different order from the order that is specified by the program order.

These weakly-ordered memory behaviors are only permitted if:

- The defined memory is Normal, Device-nGRE, or DeviceGRE, or
- Device-nR accesses span a peripheral, as described in the Device memory and Normal memory topics in the Armv8-A memory model guide.

Memory reordering allows advanced processors to operate more efficiently, as you can see in the following diagram:

Figure 2-1: Example of memory reordering



In this diagram, three instructions are listed in program order:

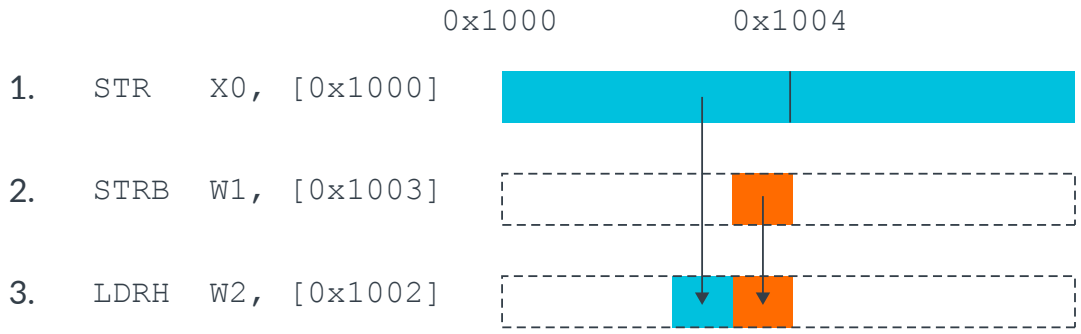
1. The first instruction, Access 1, performs a write to external memory that goes to the write buffer. This instruction is followed in program order by two reads.
2. The first read, Access 2, misses in the cache.
3. The next read, Access 3, hits in the cache.

Both read accesses could complete before the write buffer completes the write that is associated with Access 1. Caches which support Hit-Under-Miss mean that a load that hits in the cache, like Access 3, can complete before a load earlier in the program that missed in the cache, like Access 2.

Limits on reordering

It is possible to reorder accesses to locations that are marked Normal, Device-nGRE, or DeviceGRE. Consider the following example code sequence:

Figure 2-2: Example code sequence showing limits on memory reordering



If the processor reordered these accesses, we might end up with the wrong value in memory, which is not allowed.

For accesses to the same bytes, ordering must be maintained. The processor needs to detect the read-after-write hazard and ensure that the accesses are ordered correctly for the intended outcome.

This does not mean that there is no possibility of optimization with this example. The processor could merge the two stores together, presenting a single combined store to the memory system. The processor could also detect that the load operation is from the bytes that are written by the store instructions. This means that the processor could return the new value without rereading it from memory.

The sequence that is given in the example is deliberately contrived to make the point. In practice, these kinds of hazards are more subtle.

In other cases, for example Address Dependencies, ordering is enforced. An Address Dependency occurs when a load or store uses the result of a previous load as an address. Here is an example:

```
LDR X0, [X1]
STR X2, [X0]   ; Result of previous load is the address in this store.
```

Here is another example:

```
LDR X0, [X1]
STR X2, [X5, X0] ; Result of previous load is used to calculate the address.
```

If there is an Address Dependency between two memory accesses, the processor must maintain the order.

This rule does not apply to control dependencies, which occur when the value from a previous load is used to make a decision. For example:

```
LDR X0, [X1]
CBZ X0, somewhere_else
LDR X2, [X5]      ; The control dependency on X0 does not guarantee ordering.
```

Sometimes, ordering needs to be enforced between accesses to Normal memory, or accesses to Normal and Device memory. Ordering can be achieved using Barrier instructions.

3. Memory barriers

Memory barrier is the general term for an instruction which explicitly forces some form of ordering, synchronization, or restriction to memory accesses.

The Armv8 architecture defines memory barriers that provide a range of functionality, including:

- Ordering of load and store instructions
- Completion of load and store instructions
- Context synchronization
- Restrictions on speculation

There are some situations in which the effects of a weakly-ordered memory architecture are not desirable. The effects of these situations can later cause incorrect operation. This guide introduces the different memory barriers that are described by the architecture, and then identifies some of the common cases in which explicit ordering is required. This guide also describes how to use memory barriers to ensure correct operation.

4. Who is an Observer?

The Armv8-A Architecture Reference Manual uses the term Observer to describe the effects of memory barriers.

An Observer refers to either a Processor Element (PE) or some other mechanism in the system, such as a peripheral device, that can generate reads from, or writes to, memory. Observers can observe memory accesses. The memory barriers in this guide specify when and which Observers observe these memory accesses.

A write to memory is observed when it reaches a point in the memory system in which it becomes visible. When it is visible, it is coherent to all the Observers in the specified Shareability domain, as specified in the memory barrier instruction. If a PE writes to a memory location, the write is observable if another PE would see the updated value if it read the same location. For example, if the memory is Normal cacheable, the write is observable when it reaches the coherent data caches of that Shareability domain.

The Armv8-A memory model is described as Other-multi-copy atomic. In an Other-multi-copy atomic system, a write from an Observer, if observed by a different Observer, must be observed by all other Observers that access the location coherently. However, an Observer can observe its own writes before making them visible to other observers in the system.

In practice, a memory model that is described as Other-multi-copy atomic allows PEs to implement local store buffers that are not coherent with other observers in the system, but that are locally hazard-checked for dependencies. Store Buffers (STBs) are micro-architectural mechanisms that are used to decouple the execution pipeline of a PE from the Load/Store Unit (LSU).

5. Data Memory Barrier

The Data Memory Barrier (DMB) prevents the reordering of specified explicit data accesses across the barrier instruction. All explicit data load or store instructions, which are executed by the PE in program order before the DMB, are observed by all Observers within a specified Shareability domain before the data accesses after the DMB in program order.

The `DMB` instruction takes an argument that denotes which types of explicit accesses are ordered, and to which Observers in the specified Shareability domain the ordering is enforced. This is discussed further in Limiting the scope of memory barriers.

The following code demonstrates the possibility of reordering as permitted by a weakly-ordered memory model. The memory locations at `X1` and `X3` are initialized to `0x0`:

```
STR #1, [X1]
STR #1, [X3]  ; Might be observed before the previous STR.
```

In this example, the memory location at `X3` is permitted to be updated and observed before the memory location `X1` is updated. Now imagine that another Observer is going to read the same two memory locations in the same order. The following truth table shows the permitted combinations of observed values that the memory system might return:

X1	X3
0x0	0x0
0x0	0x1
0x1	0x0
0x1	0x1

The following example shows how a `DMB` instruction is used to enforce ordering observation. The memory locations at `X1` and `X3` are initialized to `0x0`:

```
STR #1, [X1]
DMB
STR #1, [X3]  ; Cannot observe this STR without first observing the previous STR.
```

In this example, if the memory location at `X3` is observed updated, then `X1` must also be observed updated. Now imagine that another Observer is going to read the same two memory locations, in the same order. The following truth table shows the permitted combinations of observed values that the memory system might return:

X1	X3
0x0	0x0
0x1	0x0
0x1	0x1

Here is some more information about the DMB:

- The use of a DMB creates order between accesses. The Armv8-A Architecture Reference Manual describes this order as Barrier-ordered-before.
- For the DMB, data cache maintenance operations are considered explicit data access instructions, and are subject to the ordering restrictions that are imposed by the DMB.


Note

A `DMB` instruction that is intended to enforce the ordering of cache maintenance instructions must specify an argument that includes both loads and stores.

- A DMB does not guarantee when the accesses occur. A DMB guarantees that when the accesses do occur, the ordering restrictions that are defined by the barrier and its arguments apply. A DMB will still permit the PE to continue execution while the explicit data are waiting to complete.
- A DMB does not prevent future explicit data reads from being speculatively executed. If a read is speculatively executed, the core must discard the speculative data from the register. The core must also re-execute the load after all previous explicit data accesses are observed.

6. Data Synchronization Barrier

A DSB is a memory barrier that ensures that those memory accesses that occur before the DSB have completed before the completion of the DSB instruction. In doing this, it acts as a stronger barrier than a DMB. All the ordering that a DMB creates with specific arguments is also generated by a DSB with the same arguments.

A DSB that is executed by a PE completes when:

- All explicit memory accesses of the required access types appear in program order before the DSB are complete for the set of observers in the required Shareability domain.
- If the argument specified in the DSB is reads and writes, then all cache maintenance instructions and all TLB maintenance instructions that are issued by the PE before the DSB are complete for the required Shareability domain.

Also, no instruction that appears in program order after the DSB instruction can alter any state of the system, or perform any part of its functionality, until the DSB completes. A DSB does not prevent fetching and decoding instructions.

The following example demonstrates the rules that are associated with a DSB:

```
STR X0, [X1]     ; Must complete before the DSB can retire.
DSB
ADD X1, X2, X3   ; Must NOT be executed before the first STR completes.
STR X4, [X5]     ; Must NOT be executed until the first STR completes.
```

In this sequence, the ordering restrictions that are imposed by the `DSB` guarantee that the second `STR` and `ADD` instructions are not executed until the previous `STR` completes and the `DSB` retires.

7. When is an access considered complete?

As described in the previous example, a DSB enforces the completion of previous memory accesses as specified by the DSB arguments. When is a memory access considered complete?

The completion of a read is easier to explain than the completion of a write. This is because the completion of a read is the point at which read data is returned to the architectural general-purpose registers of the PE.

The completion of a write is more complicated. For a write to Device memory, the point at which the write is complete depends on the Early-write acknowledgment attribute that is specified in the Device memory type, as described in Device memory in the Armv8-A memory model guide. If the memory system supports Early-write acknowledgment, then the DSB can retire before the write has reached the end peripheral. A write to memory that is defined as Device-nGnRnE can only complete when the write response comes from the end peripheral.

In the following example, the `DSB` stalls execution until the `STR` to Device-nGnRnE memory receives a write response from the end peripheral at the corresponding memory location:

```
STR X0, [Device-nGnRnE]  ; Must receive a write response from the end-peripheral
DSB SY
```

8. Limiting the scope of memory barriers

Both the DMB and DSB memory barrier instructions take an argument that defines which type of memory accesses are ordered by the memory barrier and the Shareability domain over which the instruction must operate. This scope effectively defines which Observers the ordering imposed by the barriers extend to.

The ability to limit the scope of a memory barrier is useful when trying to optimize the effects of memory barriers. In some cases, the full ordering restrictions that are imposed by a barrier can be too restrictive. Limiting the scope of a barrier to a subset of memory accesses and Observers can allow micro-architecture optimizations that could reduce some of the performance impacts of the memory barrier.

The Armv8-A AArch64 Architecture requires the explicit definition of the argument when programming a DSB or DMB. This requirement differs from previous versions, which would default the option to SY when an explicit argument was neglected.

The following table shows the valid DSB and DMB arguments:

Argument	Ordered Accesses (Before-After)	Shareability Domain
NSHLD	Load-Load, Load-Store	Non-shareable
NSHST	Store-Store	Non-shareable
NSH	Any-Any	Non-shareable
ISHLD	Load-Load, Load-Store	Inner Shareable
ISHST	Store-Store	Inner Shareable
ISH	Any-Any	Inner Shareable
OSHLD	Load-Load, Load-Store	Outer Shareable
OSHST	Store-Store	Outer Shareable
OSH	Any-Any	Outer Shareable
LD	Load-Load, Load-Store	Full System
ST	Store-Store	Full System
SY	Any-Any	Full System

For example, a DMB ISHST only affects the ordering of explicit store instructions. The ordering of loads around the barrier is unaffected. The DMB also only enforces ordering observation to the Observers which extend to the same Inner Shareable domain as the PE executing the DMB.

Consider the following example:

```
PE0

LDR x0, [X4]  ; Can be observed out-of-order
STR #1, [X1]
DMB ISHST
STR #1, [X3]
```

If PE0 and PE1 are NOT in the same Inner Shareable domain, then it would be architecturally permitted for PE1 to observe the memory location at `x3` updated before the memory location `x1` is updated. In addition, the load from `x4` is observed out of order regarding the reads from `x1` and `x3` on all PEs, including PE0.

This relaxation could reduce some of the overhead that is required when forcing ordering observation in a system.

9. Different Observers

The architecture considers the following as separate Observers:

- The instruction interface of the core, typically called the Instruction Fetch Unit (IFU)
- The data interface, typically called the Load Store Unit (LSU)
- The MMU table walk unit

As described in Who is an Observer?, an Observer is something that can make memory accesses. For example, the MMU generates reads to walk translation tables.

AArch64 does not require ordering between accesses that are made by a different Observer, even if an addresses dependency exists. For example, the following sequence could execute and complete in any order, even though a dependency is present:

```
DC CVAU, X0 ; Operations are executed in any order
IC IVAU, X0 ; despite address dependency.
```

If these instructions were reordered, the instruction cache could potentially refill with stale data from the data cache. To resolve this issue, the architecture requires a memory barrier. An example can be seen in the following code:

```
DC CVAU, X0 ; Operations are executed in any order
DSB ISH
IC IVAU, X0 ; despite address dependency.
```

In this example, the data cache clean, `DC CVAU`, is required to complete before the instruction cache invalidate is executed, `IC IVAU`. The `DC CVAU` ensures that the new data is always visible to the instruction cache before the invalidate is performed.


Note

A `DSB` is required because the `DMB` only affects data accesses. This includes the data cache maintenance operation, but not the cache invalidate instruction.

10. Load-Acquire and Store-Release instructions

Armv8-A AArch64 provides a set of instructions with Acquire semantics for loads, and Release semantics for stores. These instructions support the Release Consistency sequentially consistent (RCsc) model.

These new load and store instructions include implicit barrier semantics, which are like one-way barriers. This is because these instructions impose weaker ordering requirements than a `DMB` or a `DSB`, because they affect the ordering of specified explicit memory accesses that are on each side of the memory barrier instruction. Weaker ordering requirements that are imposed by Load-Acquire and Store-Release instructions allow for micro-architectural optimizations, which could reduce some of the performance impacts that are otherwise imposed by an explicit memory barrier. If the ordering requirement is satisfied using either a Load-Acquire or Store-Release, then it would be preferable to use these instructions instead of a `DMB`.

The Shareability domain defines which Observers the ordering imposed by these instructions extend to. The Shareability domain for a Load-Acquire and Store-Release is defined by the Shareability domain of the address that is being accessed by the instruction. For example, if PE0 and PE1 are not in the same Inner Shareable domain, then it would be architecturally permitted for PE1 to observe the memory location at `X3` updated before the memory location `X1` is updated if `X3` was defined as Inner Shareable. This is shown in the following code:

```
PE0

STR  #1, [X1]
STLR #1, [X3]
```

Here is some more information about Load-Acquire and Store-Release instructions:

- For Load-Acquire, Load-AcquirePC, and Store-Release instructions, the address of the data object that is supplied must be aligned to the size of the data element that is being accessed. Otherwise, the access generates an Alignment fault.
- For a Load-Acquire Exclusive Pair and Store-Release Exclusive Pair , the address that is supplied to the instructions must be aligned to twice the size of the element that is being loaded. Otherwise the access generates an Alignment fault. This is shown in the following code:

  ```
  LDAXP x0, x1, [0x08]  ; Alignment fault
  LDAXP x0, x1  [0x10]
  ```

- Exclusive variants of the Load-Acquire and Store-Release instructions are available.

Load-Acquire

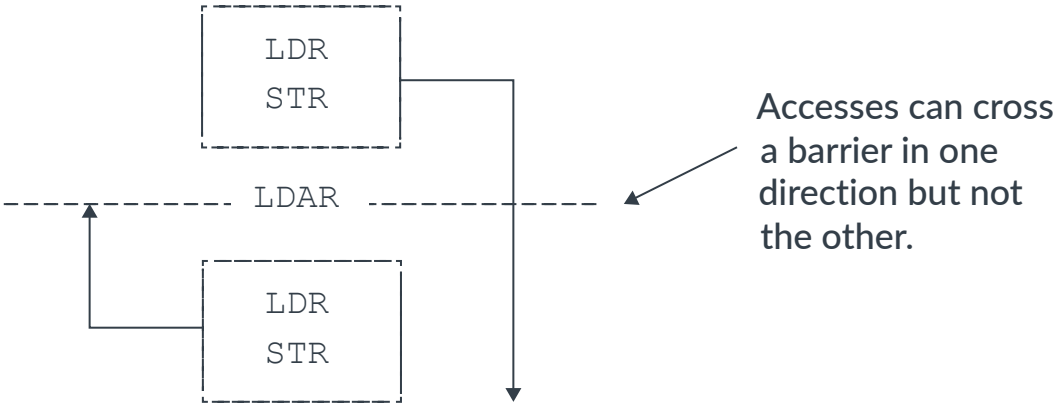
The ordering requirements that are imposed by the Load-Acquire `LDAR` instruction are as follows:

- All explicit memory accesses after the `LDAR` are observed after the `LDAR`.

- All explicit memory accesses before the `LDAR` are not affected, and are reordered regarding the `LDAR`.

You can see these ordering requirements in the following diagram:

Figure 10-1: LDAR ordering requirements



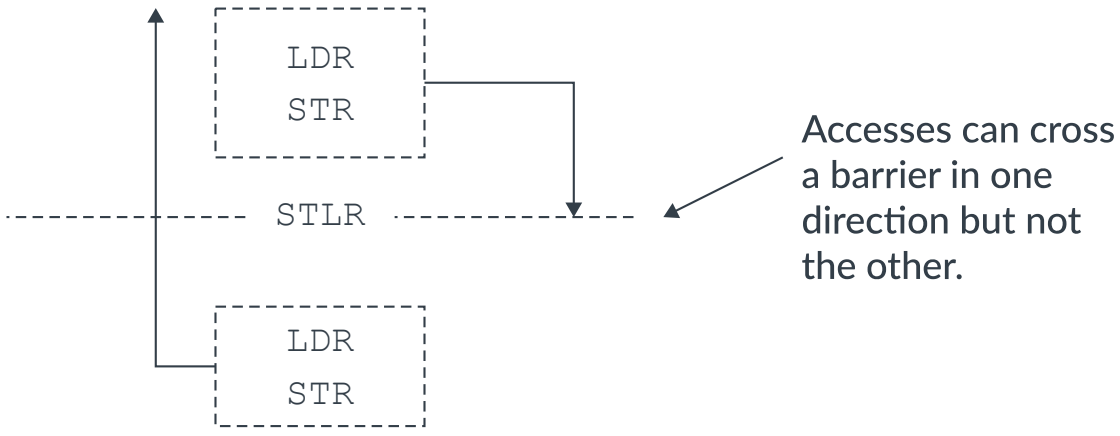
Store-Release

The ordering requirements that are imposed by Store-Release `STLR` instruction are as follows:

- All explicit memory accesses before the `STLR` are observed before the `STLR`.
- All explicit memory accesses after the `STLR` are not affected, and are reordered regarding the `STLR`.

You can see these ordering requirements in the following diagram:

Figure 10-2: STLR ordering requirements



The following example shows how to use an `STLR` to enforce ordering observation. The memory locations at `x1` and `x3` are initialized to `0x0`:

```
STR  #1, [X1]
STLR #1, [X3]  ; Cannot observe this STLR without observing the previous STR.
```

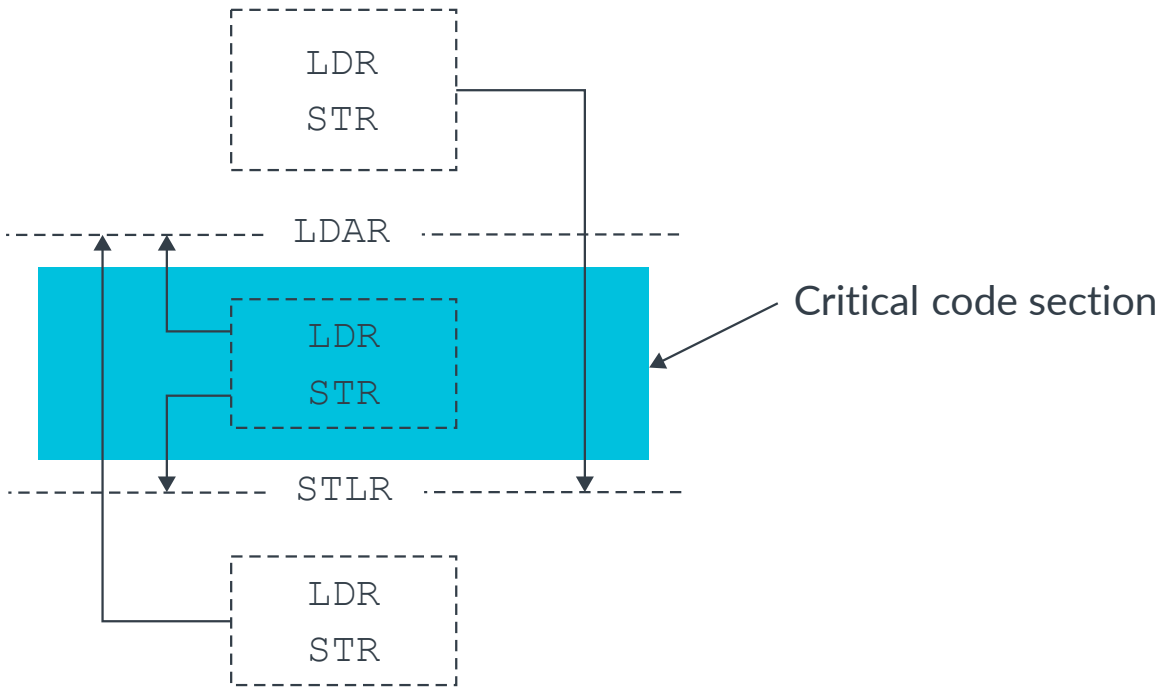
In this example, if the memory location at `x3` is observed updated, then `x1` must also be observed updated. If another observer reads the same two memory locations, in the same order, the following truth table shows the combination of values that the memory system might return:

X1	X3
0x0	0x0
0x1	0x0
0x1	0x1

Load-Acquire and Store-Release pairs

Load-Acquire and Store-Release instructions can be combined in a pair to protect a critical section of code. Combining these instructions ensures that accesses that are made within the critical code section are not reordered outside of the critical section. Accesses inside the critical code section are not affected, and can be reordered, as you can see in the following diagram:

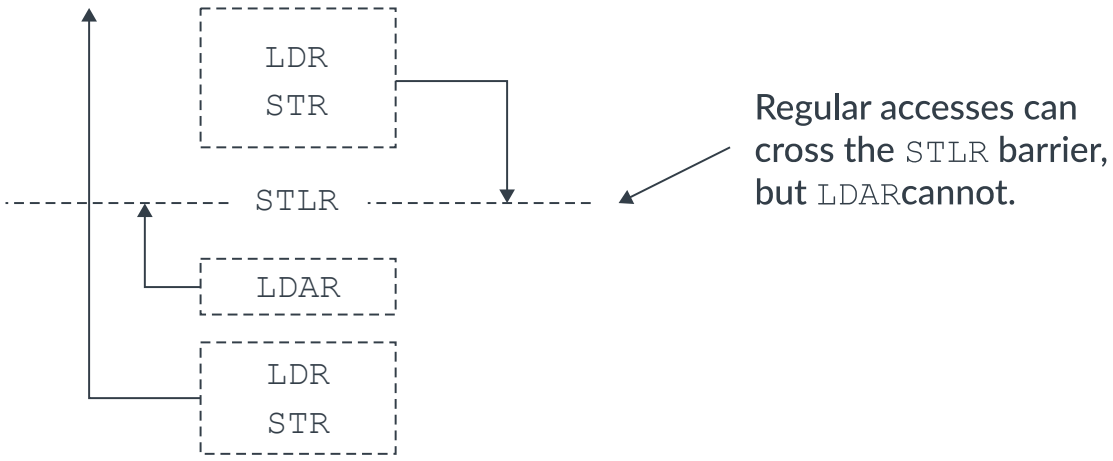
Figure 10-3: Protecting a critical code section with an LDAR-STLR pair



Sequentially consistent

Acquire/release operations use a sequentially consistent model. This means that, when a Load-Acquire appears in program order after a Store-Release, the memory access that is generated by the Store-Release instruction is observed before the memory access that is generated by the Load-Acquire instruction. You can see these ordering requirements in the following diagram:

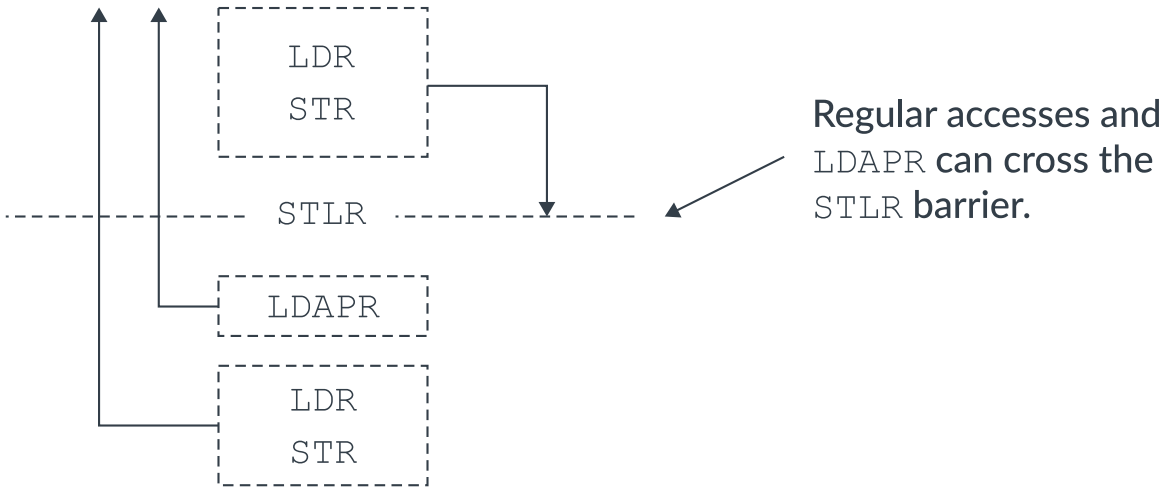
Figure 10-4: Sequentially consistent ordering requirements



Load-AcquirePC

Armv8.3-A also provides Load-AcquirePC instructions. The combination of Load-AcquirePC and Store-Release is used to support the weaker Release Consistency processor consistent (RCpc) model, as you can see in the following diagram:

Figure 10-5: Load-AcquirePC instructions



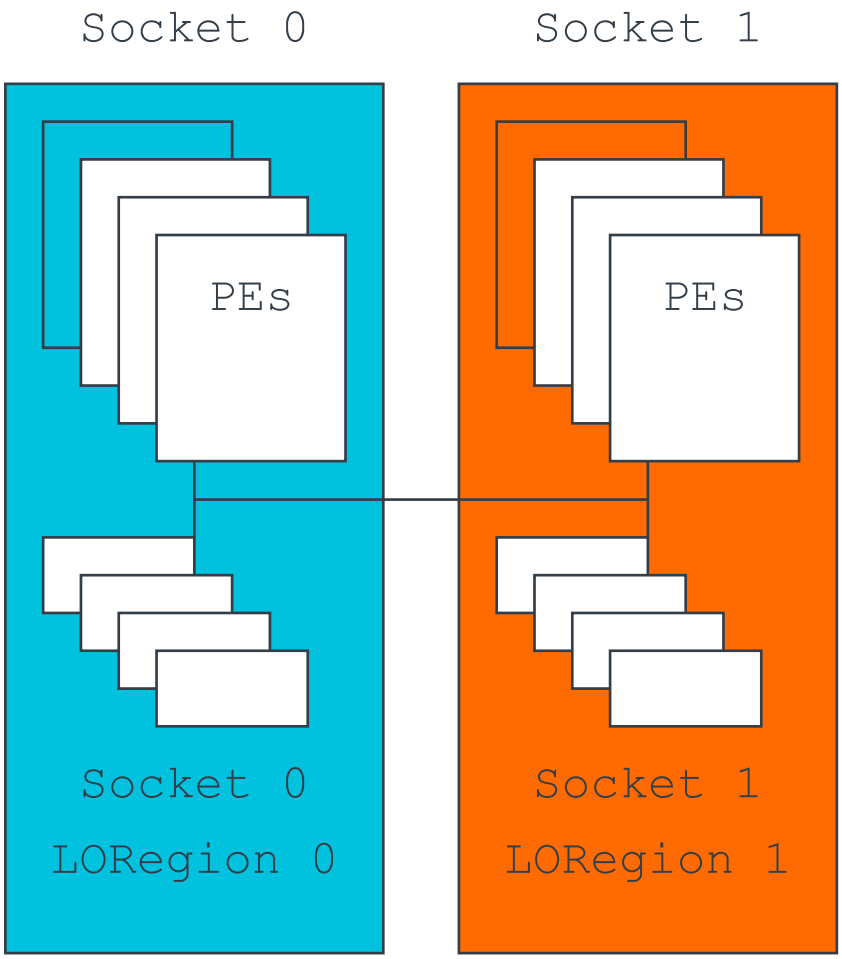
The requirement that Load-Acquires are observed in order with preceding store-releases is dropped for these new Load-AcquirePC instructions.

Limited Ordering Regions

Armv8.1-A adds support for Limited Ordering Regions (LORegions). LORegions allow large systems to perform special Load-Acquire (LDLAR) and Store-Release (STLLR) instructions that provide order between the memory accesses only to a specified region of the Physical Address (PA) map.

LORegions avoid the need to take large performance hits when waiting for a memory access, to any location in the memory map, to be observed by all Observers in the same Shareability domain as the initiating PE. This situation would also occur with the existing Load-Acquire and Store-Release instructions. This functionality is used only by software that is aware of a limited set of observers wanting to share a memory location. This software is usually software that is aware of the topology of the system. For example, the following diagram shows a multi-socket system in which ordered accesses that spanned the sockets could impose a large delay:

Figure 10-6: Multi-socket system using LORegions



It is possible to design a system in which limiting ordering to a local region of physical memory that is used by a socket gives an overall performance gain.

LORegions only apply to Non-secure physical memory accesses. An LORegion is defined by an LORegion descriptor. The number of LORegion descriptors is **IMPLEMENTATION DEFINED** and is discovered by reading the LORID_EL1 register.

Each LORegion descriptor includes the following elements, which are programmed using System register accesses:

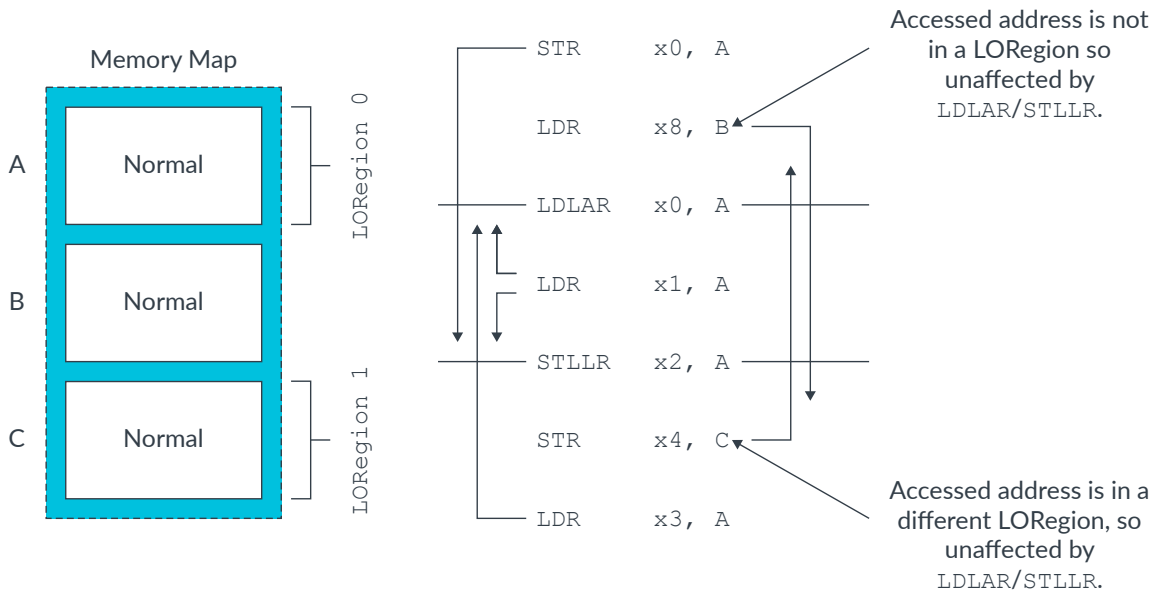
- Start Address (LORSA_EL1)
- End Address (LOREA_EL1)
- LORegion Number (LORN_EL1)
- Valid bit which indicates whether that LORegion descriptor is valid (LORC_EL1)

The following code demonstrates programming for LORegion 2:

```
MOV  x0, #0x2            // LORegion number
MOV  x1, #0x80000000      // LORegion start address
MOV  x2, #0xC0000000      // LORegion end address
MOV  x3, #0x1            // LORegion enable (valid bit)
MSR  LORN_EL1, x0        // Select the LORegion number descriptor
ISB
MSR  LORSA_EL1, x1
MSR  LOREA_EL1, x2
MSR  LORC_EL1, x3
ISB
```

In the following diagram, only accesses that are made to addresses in the same LORegion, as specified by either the `LDLAR` or `STLLR` instruction, are affected. Memory accesses that are made outside of the LORegion are unaffected. For example, the store to c is observed before the `LDLAR`. If software used a `LDAR`, the store to c would be observed after the `LDAR`, as you can see in the following diagram:

Figure 10-7: Example showing LORegion memory accesses



11. Instruction barriers

The Arm architecture defines the context of the PE as the state of the caches, TLBs, and the System registers. Performing cache or TLB maintenance operations or updating System registers is classed as a context-changing operation.

The architecture only guarantees that a context-changing operation is seen after a Context synchronization event.

The requirement for explicit synchronization relaxes the need for processor designers to propagate all changes of context each cycle. Implicit synchronization is unnecessary overhead. The requirement of a Context synchronization event means that software can explicitly define when it wants a new context to apply.

A Context synchronization event does one of the following:

- Performs an ISB operation
- Takes an exception
- Returns from an exception
- Exits from Debug state


Note

Arm processor implementations are permitted to never update their context to subsequent execution if the PE never performs a Context synchronization event.

The execution of a Context synchronization event ensures that:

- All unmasked interrupts that are pending at the time of the Context synchronization event are taken before the first instruction after the Context synchronization event.
- No instructions that appear in program order after an instruction that causes a Context synchronization event can perform any part of their functionality until the Context synchronization event has occurred.
- All writes to System registers that are made before the Context synchronization event affect any instruction that appears in program order after the instruction that causes the Context synchronization event.
- All completed changes to the translation tables for entries that, before the change, were not permitted to be cached in a TLB, affect all instruction fetches that appear in program order after the instruction causing the Context synchronization event.
- All invalidations of TLBs, instruction caches, and, in AArch32 state, branch predictors, that are completed before the Context synchronization event affect all instructions that appear in program order after an instruction causing a Context synchronization event.

Example usage

Software must first ensure that accesses to SVE, Advanced SIMD, and floating-point registers are not trapped. For example, while executing at EL1, trapping of SVE, Advanced SIMD, and floating-point register accesses can be disabled by programming CPACR_EL1.FPEN to `0x3`, as shown in the following code:

```
MRS     X1, CPACR_EL1
ORR     X1, X1, #(0x3 << 20)  ; Write CPACR_EL1.FPEN bits
MSR     CPACR_EL1, X1
ISB
FADD    S0, S1, S2
```

Without the `ISB` instruction, the act of disabling trapping, which is a context-changing operation, would not be guaranteed to be seen by the `FADD` instruction. Lack of an `ISB` ultimately causes the `FADD` instruction to take a Synchronous exception. In this case, the `ISB` is the Context synchronization event that is required to ensure that the new context, which includes the disabling of traps to the SVE, Advanced SIMD, and floating-point registers, is seen by the `FADD` instruction.

In this example, if EL1 returned to EL0 before any SVE, Advanced SIMD, and floating-point register access, the `ISB` would not be required. This is because the act of performing the exception return from EL1 to EL0 is a Context synchronization event.

12. Check your knowledge

Q1. What is an Observer?

A1. An Observer is a processing element or mechanism in the system, such as a peripheral device, that can generate reads from, or writes to, memory.

Q2. Which `DMB` qualifier do I use to ensure that two stores that are separated by a DMB are observed in order by other Observers in the same Inner Shareable domain?

A2. `DMB ISHST`

Q3. Which memory barrier would be required to ensure that previous memory accesses are complete before execution continues?

A3. `DSB`

Q4. What are the architecturally defined Context synchronization events?

A4. The architecturally defined Context synchronization events are as follows:

- Performing an `ISB` operation
- Taking an exception
- Returning from an exception
- Exit from Debug state

13. Related information

Here are some resources that are related to material in this guide:

- Armv8-A Architecture Reference Manual. In particular, the sections:
  - Barrier Litmus Tests
  - The AArch64 Application Level Memory Model
- Arm Community - Ask development questions, and find articles and blogs on specific topics from Arm experts.
- Learn the architecture - AArch64 memory model
- Memory Consistency Models for Shared Memory-Multiprocessors, Gharachorloo, Kourosh, 1995, Stanford University Technical Report CSL-TR-95-685 – This paper provides more information on RSsc and RCpc.

14. Next steps

Refer to Barrier Litmus Tests in the latest Armv8-A Architecture Reference Manual for several detailed memory and instruction barrier examples. This section of the Armv8-A Architecture Reference Manual provides detailed worked examples for many of the common software cases that require barriers.

To keep learning about the Armv8-A architecture, see more in our Learn the architecture series of guides.