Barrier Litmus Tests and Cookbook

Business Unit

Group

Document number:	**PRD03-GENC-007826** 1.0
Date of Issue:	26 November 2009
Author:	Richard Grisenthwaite
Authorised by:	

Abstract

Keywords

Distribution list

Name	Function	Name	Function

Contents

1 ABOUT THIS DOCUMENT

1.1 Change control

1.1.1 Current status and anticipated changes

Final

1.1.2 Change history

Issue	Date	By	Change

1.2 References

This document refers to the following documents.

Ref	Doc No	Author(s)	Title
1	ARM DDI 0406 B	ARM Limited	ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition
2	PRD03-GENC-007853	Richard Grisenthwaite	ARMv7 MP Extensions
3		Kourosh Gharachorloo	Memory Consistency Models for Shared Memory-Multiprocessors. Published by Stanford University (Technical Report CSL-TR-95-685)

1.3 Terms and abbreviations

This document uses the following terms and abbreviations.

Term	Meaning

2 INTRODUCTION

The exact rules for the insertion of barriers into code sequences is a very complicated subject, and this document describes many of the corner cases and behaviours that are called out in the ARM architecture with the advent of the ARMv7 MP Extensions.

The document is designed to help programmers, hardware design engineers and validation engineers to better understand the need for the different kinds of barriers.

3 OVERVIEW OF MEMORY CONSISTENCY

Early generations of microprocessors were relatively simple processing engines, which executed each instruction in program order. In such processors, the effective behaviour was that each instruction was executed in its entirety before a subsequent instruction started to be executed. This behaviour is sometimes referred to as the *Sequential Execution Model (SEM)*.

Over the increasing processor generations, the needs to increase processor performance, both in terms of the frequency of operation and in terms of the number of instructions executed each cycle, have meant that such a simple from of execution has been abandoned. Many techniques, such as pipelining, write buffering, caching, speculation and out-of-order execution, have been introduced to provide improved performance.

For general purpose processors, such as ARM, these micro-architectural innovations have largely been hidden from the programmer by a number of micro-architectural techniques. These techniques have ensured that within an individual processor, the behaviour of the processor largely remains the same as the Sequential Execution Model; there have been some exceptions to this, where explicit synchronisation has been required; in the case of the ARM architecture, these have been limited to cases such as:

- Synchronization of changes to the instruction stream
- Synchronization of changes to system control registers

In both these cases, the ARM *ISB* instruction is used to provide the necessary synchronization.

While the effect of ordering has been largely hidden from the programmer within a single processor, the micro-architectural innovations have had (and will continue to have) a profound impact in the order in which accesses are made to memory. Write buffering, speculation and cache coherency protocols, in particular, all tend to mean that the order in which memory accesses occur, as seen by an external observer, may differ significantly from the order of accesses that would appear in the SEM. And while this is basically invisible within the uniprocessor environment, the effect becomes much more significant when multiple processors are trying to communicate in memory. In reality, these effects are often only actually significant at particular synchronization boundaries between the different threads of execution.

This problems that arise from memory ordering considerations is sometimes described as being the problem of *memory consistency*. Processor architectures have adopted one or more *memory consistency models* (or more generally, *memory models*) that describe the permitted limits of the memory re-ordering that can be performed by implementations of that architecture. The comparison and categorization of these has been the subject of significant research and comment in academic circles, and in particular, Gharachorloo's paper [3] is recommended as an excellent detailed treatment of this subject.

This document does not seek to reproduce such a work, but instead concentrates on some cases which demonstrate the features of the weakly-ordered memory model that has been adopted by ARM as its memory model from ARMv6. In particular, the cases show how the use of the ARM memory barrier instructions *DMB* and *DSB* can be used to provide the necessary safeguards to limits memory ordering effects at the required synchronization points.

4 DEFINITION OF THE BARRIER OPERATIONS

The definition of the barriers in [1, section A3.8.3] are copied here for ease of reference

DMB

The DMB instruction is a data memory barrier. The processor that executes the DMB instruction is referred to as the executing processor, Pe. The DMB instruction takes the *required shareability domain* and *required access types* as arguments. If the required shareability is *Full system* then the operation applies to all observers within the system.

A DMB creates two groups of memory accesses, Group A and Group B:

Group A Contains:

- All explicit memory accesses of the required access types from observers in the same required shareability domain as Pe that are observed by Pe before the DMB instruction. These accesses include any accesses of the required access types and required shareability domain performed by Pe.
- All loads of required access types from observers in the same required shareability domain as Pe that have been observed by any given observer, Py, in the same required shareability domain as Pe before Py has performed a memory access that is a member of Group A.

Group B Contains:

- All explicit memory accesses of the required access types by Pe that occur in program order after the DMB instruction.
- All explicit memory accesses of the required access types by any given observer Px in the same required shareability domain as Pe that can only occur after Px has observed a store that is a member of Group B.

Any observer with the same required shareability domain as Pe observes all members of Group A before it observes any member of Group B to the extent that those group members are required to be observed, as determined by the shareability and cacheability of the memory locations accessed by the group members.

Where members of Group A and Group B access the same memory-mapped peripheral, all members of Group A will be visible at the memory-mapped peripheral before any members of Group B are visible at that peripheral.

DSB

The DSB instruction is a special memory barrier, that synchronizes the execution stream with memory accesses. The DSB instruction takes the *required shareability domain* and *required access types* as arguments.

If the required shareability is *Full system* then the operation applies to all observers within the system. A DSB behaves as a DMB with the same arguments, and also has the additional properties defined here.

A DSB completes when both:

- all explicit memory accesses that are observed by Pe before the DSB is executed, are of the required access types, and are from observers in the same required shareability domain as Pe, are complete for the set of observers in the required shareability domain
- all cache, branch predictor, and TLB maintenance operations issued by Pe before the DSB are complete for the required shareability domain.

In addition, no instruction that appears in program order after the DSB instruction can execute until the DSB completes.

Auxilliary Definitions

The definition of *observation* and *completion* are defined in [1] section A3.8.1.

Program order is defined to the order of instructions as they appear in an assembly language program. This document does not attempt to describe or define the legal transformations from a program written in a higher level programming language, such as C or C++, into the machine language which can then be disassembled to give an equivalent assembly language program. Such transformations are a function of the semantics of the higher level language and the capabilities and options on the compiler.

5 CONVENTIONS

Many of the examples are written in a stylised extension to ARM assembler, in order to avoid confusing the examples with unnecessary code sequences. In particular, the construct

```
WAIT([Rx]==1)
```

Is used to describe the following sequence

```
loop

    LDR R12, [Rx]

    CMP R12, #1

    BNE loop
```

R12 is chosen as an arbitrary temporary register that has not been used; it is named to allow the generation of a false dependency to ensure ordering.

For each example, a code sequence will by preceded by an identifier of the observer running it.:

- P0, P1…Px refer to cached coherent processors implementing the ARMv7 architecture with MP Extensions all situated within the same Shareability domain
- E0, E1…Ex refer to uncached observers, not participating in the coherency protocol, but which, for documentation purposes execute ARM instructions and have a weakly-ordered memory model. This does not preclude such objects being quite different (eg DMA engines or other system masters).

These observers are unsynchronised other than is required by the documented code sequence.

Results are expressed in terms of <agent>:<register>, such as P0:R5.

Some results are described as being “permissible”. This does not imply that the results expressed are required or are the only possible results; in most cases they are results that would not have been possible under a sequentially consistent running of the code sequences on the agents involved. Loosely, this means that these results might be unexpected to those unfamiliar with memory consistency issues.

Results that are described as being “not permissible” are those which the architecture expressly forbids.

Results that are described as being “required” are those which the architecture expressly requires.

The required shareability domain arguments of the DMB/DSB have been omitted, and are assumed to be selected appropriate to the shareability domains of the observers.

Where the barrier function in the litmus test can be achieved by a DMB ST (which acts as a barriers to stores only), this is shown by the use of DMB [ST] which indicates that the ST qualifier can be omitted without affecting the result of the test. In some implementations DMB ST will be faster than DMB.

Other conventions are used unless otherwise stated.

- All Memory is assumed to be initialised to 0
- R0 is assumed to always contain the value 1

- R1 – R4 contain arbitrary independent addresses which are initialised to the same value on all processors; R1, R2's addresses are Write-Back Cacheable Normal memory ; R3 is Write-Through Cacheable Normal Memory. R4's address is a Non-cacheable Normal memory location. All locations are Shareable.
- R5 – R8 contain 0x55, 0x66, 0x77, 0x88 when used for STR unless they have been initialised by other values. Similarly they contain the value 0 before being loaded into by an LDR, unless they have been initialised to other values.
- R11 contains a new instruction or new translation table entry as appropriate; R10 contains the virtual address (and ASID) being changed by a change of translation table entry.
- Memory locations are assumed to be Normal memory locations unless otherwise stated.

The cache maintenance operations are abbreviated with the following mnemonics:

- `ICIALLU -Instruction cache invalidate all to PoU`
- `ICIALLUIS -Instruction cache invalidate all to PoU Inner Shareable`
- `ICIMVAU Rx - Instruction cache invalidate by MVA to PoU`
- `BPIALL - Branch Predictor invalidate all`
- `BPIALLIS - Branch Predictor invalidate all Inner Shareable`
- `BPIMVA Rx - Branch Predictor invalidate by MVA`
- `DCIMVAC Rx - Data Cache Invalidate by MVA to PoC`
- `DCISW Rx - Data Cache Invalidate by set/way`
- `DCCMVAC Rx - Data Cache Clean by MVA to PoC`
- `DCCSW Rx - Data Cache Clean by set/way`
- `DCCMVAU Rx - Data Cache Clean by MVA to PoU`
- `DCCIMVAC Rx - Data Cache Clean & Invalidation by MVA to PoC`
- `DCCISW Rx - Data Cache Clean & Invalidation by Set/Way`

The TLB maintenance operations are abbreviated with the following mnemonics:

- `ITLBIALL - Instruction TLB Invalidate All`
- `ITLBIMVA - Instruction TLB Invalidate by MVA`
- `ITLBIASID - Instruction TLB Invalidate by ASID`
- `DTLBIALL - Data TLB Invalidate All`
- `DTLBIMVA - Data TLB Invalidate by MVA`
- `DTLBIASID - Data TLB Invalidate by ASID`
- `TLBIALL - Unified TLB Invalidate All`
- `TLBIMVA - Unified TLB Invalidate by MVA`
- `TLBIMVAAA - Unified TLB Invalidate by MVA All ASID`
- `TLBIASID - Unified TLB Invalidate by ASID`
- `TLBIALLIS - Unified TLB Invalidate All Inner Shareable`
- `TLBIMVAIS - Unified TLB Invalidate by MVA Inner Shareable`
- `TLBIASIDIS - Unified TLB Invalidate by ASID Inner Shareable`
- `TLBIMVAAAIS- Unified TLB Invalidate by MVA All ASID Inner Shareable`

5.1 Notes on Timing Effects

Many of the examples are supplied for the purpose of demonstrating the effect of memory ordering on transactions. In general, the architecture makes no statement about *when* an instruction will occur, and in particular, stores may take an unbounded time to be observed by other observers (and so the WAIT loop waiting of the result of that store may take an unbounded time to move forward).

Where it is necessary to guarantee the completion of a store, a DSB can be used to force this drain (as would be the case with WFI or WFE – see section 6.6

In general, the examples in this document associated with ordering assume that stores will become observable over time and so a final DSB barrier to ensure the completion of stores is omitted.

6 SIMPLE ORDERING AND BARRIER CASES

ARM implements a weakly consistent memory model for Normal memory. Loosely, this means that the order of memory accesses observed by other observers may not be the order that appears in the program, for either loads or stores.

A number of simple examples of this exist.

6.1 Simple Weakly Consistent Ordering Example

```
P1:

    STR R5, [R1]

    LDR R6, [R2]

P2:

    STR R6, [R2]

    LDR R5, [R1]
```

In the absence of barriers, the result of P1: R6 = 0, P2: R5 = 0 is permissible

6.2 Message Passing

6.2.1 Weakly-Ordered Message Passing problem

```
P1:

    STR R5, [R1]          ; set new data

    STR R0, [R2]          ; send flag indicating data ready

P2:

    WAIT([R2]==1)         ; wait on flag

    LDR R5, [R1]          ; read new data
```

In the absence of barriers, an end result of P2: R5=0 is permissible.

6.2.1.1 Resolving by the addition of barriers

The addition of barriers to ensure the observed order of both the reads and the writes would allow transfer of data such that the result P2:R5==0x55 is guaranteed, as follows:

```
P1:

    STR R5, [R1]          ; set new data

    DMB [ST]              ; ensure all observers observe data before the flag

    STR R0, [R2]          ; send flag indicating data ready


P2:
```

```
    WAIT([R2]==1)          ; wait on flag
    DMB                    ; ensure that the load of data is after the flag has been observed
    LDR R5, [R1]
```

6.2.1.2 Resolving by the use of barriers and address dependency

There is a rule within the ARM architecture that:

> Where the value returned by a read is used to compute the virtual address of a subsequent read or write (this is known as an *address dependency)*, then these two memory accesses will be observed in program order. An address dependency exists even if the value read by the first read has no effect in changing the virtual address (as might be the case if the value returned is masked off before it is used, or if it had no effect on changing a predicted address value).
>
> This restriction applies only when the data value returned from one read is used as a data value to calculate the address of a subsequent read or write. This does not apply if the data value returned from one read is used to determine the condition code flags, and the values of the flags are used for condition code evaluation to determine the address of a subsequent reads, either through conditional execution or the evaluation of a branch. This is known as a *control dependency*.
>
> Where both a control and address dependency exist, the ordering behaviour is consistent with the address dependency.

In the table below, examples (a) and (b) exhibit address dependencies; examples (c) and (d) exhibit control dependencies; example (e) exhibits both address and control dependencies, the address dependency taking priority.

(a)	(b)	(c)	(d)	(e)

`LDR r1, [r0]`
`LDR r2, [r1]`
	`LDR r1, [r0]`
	`AND r1, r1, #0`
	`LDR r2, [r3, r1]`
		`LDR r1, [r0]`
		`CMP r1, #55`
		`LDRNE r2, [r3]`
			`LDR r1, [r0]`
			`CMP r1, #55`
			`MOVNE r4, #22`
			`LDR r2, [r3, r4]`
				`LDR r1, [r0]`
				`CMP r1, #0`
				`LDRNE r2, [r1]`

This means that the data transfer example can also be satisfied in the following case.

```
P1:
    STR R5, [R1]           ; set new data
    DMB [ST]               ; ensure all observers observe data before the flag
    STR R0, [R2]           ; send flag indicating data ready


P2:
    WAIT([R2]==1)
    AND R12, R12, #0       ; R12 is destination of LDR in WAIT macro
    LDR R5, [R1, R12]      ; Load is dependent and so is ordered after the flag has been seen
```

In this case, the load of R5 by P2 is ordered with respect to the load from [R2] as there is an address dependency using R12. The use of a DMB by P1 is still required to ensure that the write of [R2] is not observed before the write of [R1].

6.2.2 Message passing with multiple observers

Where the ordering of normal memory accesses has not been resolved by the use of barriers or dependencies, then different observers may observe the accesses in a different order. In the following case

```
P1:

    STR R5, [R1]          ; set new data

    STR R0, [R2]          ; send flag indicating data ready



P2:

    WAIT([R2]==1)

    AND R12, R12, #0      ; R12 is destination of LDR in WAIT macro

    LDR R5, [R1, R12]     ; Load is dependent and so is ordered after the flag has been seen


P3:

    WAIT([R2]==1)

    AND R12, R12, #0      ; R12 is destination of LDR in WAIT macro

    LDR R5, [R1, R12]     ; Load is dependent and so is ordered after the flag has been seen
```

In this case, it is permissible that P2:R5 and P3:R5 contain different values as there is no order guaranteed between the two stores performed by P1.

6.2.2.1 Resolving by the addition of barriers

The addition of barrier by P1 ensures the observed order of the writes would allow transfer of data such that P2:R5 and P3:R5 contain the same value of 0x55.

```
P1:

    STR R5, [R1]          ; set new data

    DMB [ST]              ; ensure all observers observe data before the flag

    STR R0, [R2]          ; send flag indicating data ready


P2:

    WAIT([R2]==1)

    AND R12, R12, #0      ; R12 is destination of LDR in WAIT macro
```

```
    LDR R5, [R1, R12]      ; Load is dependent and so is ordered after the flag has been seen


P3:
    WAIT([R2]==1)
    AND R12, R12, #0       ; R12 is destination of LDR in WAIT macro
    LDR R5, [R1, R12]      ; Load is dependent and so is ordered after the flag has been seen
```

6.3 Address Dependency with object construction

The address dependency rule allows the avoidance of barriers in relatively common cases of accessing an object oriented data structure, even in the face of that object being initialised.

```
P1:
     STR R5, [R1, #offset] ; set new data in a field
     DMB  [ST]             ; ensure all observers observe data before base address is updated
     STR R1, [R2]          ; update base address
P2:
     LDR R1, [R2]          ; read for base address
     CMP R1, #0            ; check if it is valid
     BEQ null_trap
     LDR R5, [R1, #offset] ; use base address to read field
```

It is required that P2:R5==0x55 if the null_trap is not taken. This avoids P2 seeing a partially constructed object from P1. Significantly, P2 does not need a barrier to ensure this behaviour.

It should be appreciated that the barrier is required with P1 to ensure the observed order of the writes by P1. In general, the impact of requiring a barrier in the construction case is a lot less than the impact of requiring a barrier for every read access.

6.4 Causal consistency issues with Multiple observers

The fact that different observers can observe memory accesses in different orders extends, in the absence of barriers, to behaviours that do not fit naturally expected causal properties.

```
P1:
    STR R0, [R2]           ; set new data
P2:
    WAIT([R2]==1)          ; wait to see new data from P1
    STR R0, [R3]           ; send flag, must be after the new data has been by P2 as stores
                           ; must not be speculative
P3:
```

```
    WAIT([R3]==1)          ; wait for P2’s flag
    AND R12, R12, #0       ; dependency to ensure order
    LDR R0, [R2, R12]      ; read P1’s data
```

In this case, P3:R0 == 0 is permissible. P3 is not guaranteed to see the store from P1 and the store from P2 in any particular order. This applies despite the fact that the store from P2 can only happen after P2 has observed the store from P1.

This result means that the ARM memory ordering model for Normal memory does not conform to *causal consistency*. It therefore means that the apparently transitive causal relationship between two variables is not guaranteed to be transitive.

The insertion of a barrier in P2 as shown below, creates causal consistency in this case:

```
P1:
    STR R0, [R2]           ; set new data
P2:
    WAIT([R2]==1)           ; wait to see new data from P1
    DMB                     ; ensure P1’s data is observed by all observers before any
                            ; following store
    STR R0, [R3]            ; send flag
P3:
    WAIT([R3]==1)          ; wait for P2’s flag
    AND R12, R12, #0       ; dependency to ensure order
    LDR R0, [R2, R12]      ; read P1’s data
```

This is because DMB is defined to order all accesses which the executing processor has observed before the DMB, (not just those is has issued), before any of the accesses which follow the DMB.

6.5 Multiple observers of writes to multiple locations

The ARM weakly consistent memory model means that different observers can observe writes to different locations in different orders.

```
P1:
    STR R0, [R1]           ; set new data


P2:
    STR R0, [R2]           ; set new data


P3:
```

```
    LDR R10, [R2]          ; read P2's data before P1s
    LDR R9, [R1]           ;
    BIC R9, R10, R9        ; R9  <- R10 & ~R9
                           ; R9 contains 1 iff read from [R2] is observed to be 1 and
                           ; read from [R1] is observed to be 0.
P4  :
    LDR R9, [R1]
    LDR R10, [R2]
    BIC  R9, R9, R10       ; R9 <- R9 & ~R10
                           ; R9 contains 1 iff read from [R2] is observed to be 0 and
                           ; read from [R1] is observed to be 1.
```

In this case, the result P3:R9==1 and P4:R9==1 is permissible, which means that P3 and P4 have observed the stores from P1 and P2 in different orders.

This breaking of sequential consistency can be resolved by the use of DMB barriers, as shown:

```
P1:
    STR R0, [R1]           ; set new data

P2:
    STR R0, [R2]           ; set new data

P3:
    LDR R10, [R2]          ; read P2's data before P1s
    DMB
    LDR R9, [R1]
    BIC R9, R10, R9        ; R9 <- R10 & ~R9
                           ; R9 contains 1 iff read from [R2] is observed to be 1 and
                           ; read from [R1] is observed to be 0.

P4:
    LDR R9, [R1]           ; read P1's data before P2s
    DMB
    LDR R10, [R2]
    BIC R9, R9, R10        ; R9 <- R9 & ~R10
```

```
                        ; R9 contains 1 iff read from [R2] is observed to be 0 and
                        ; read from [R1] is observed to be 1.
```

In this case:

- the DMB executed by P3 ensures that, if P3's load from [R2] observes P2's store to [R2], then all observers will observe it before they observe P3's load from [R1].
- the DMB executed by P4 ensures that, if P4's load from [R1] observes P1's store to [R1], then all observers will observe it before they observe P4's load from [R2].

If P3's load from [R1] returns 0, then it has not observed P1's store to [R1]. If additionally P3's load of [R2] returns 1, then all observers must have observed P2's store to [R2] before they observe P1's store to [R1]. Thus P4 cannot observe P1's store to [R1] without also observing P2's store to [R2].

Alternatively, if P4's load from [R2] returns 0, then it has not observed P2's store to [R2]. If additionally P4's load of [R1] returns 1, then all observers must have observed P1's store to [R1] before they observe P2's store to [R2]. Thus P3 cannot observe P2's store to [R2] without also observing P1's store to [R1].

As a result, of the 4 possible values for {P3:R9, P4:R9}, the value {1,1} is impossible with the insertion of these barriers.

6.6 Posting a Store before polling for acknowledgement

In the case where an observer stores to a location, and then polls for an acknowledge from a different observer, the weak ordering of the memory model can lead to a deadlock:

```
P1:

     STR R0, [R2]

     WAIT ([R3)==1)

P2:

     WAIT ([R2]==1)

     STR R0, [R3]
```

This can effectively deadlock as the P1's store might not be observed by P2 for an indefinite period of time.

The addition of a DMB barrier:

```
P1:

     STR R0, [R2]

     DMB

     WAIT ([R3)==1)

P2:

     WAIT ([R2]==1)

     STR R0, [R3]
```

The DMB executed by P1 ensures that P1's store is observed by P2 before P1's load is observed by P2, so ensuring a timely completion.

One interesting variant of this example is where the same memory location is being polled, as in the following example:

```
P1:

     STR R0, [R2]

     WAIT ([R2)==2)

P2:

     WAIT ([R2]==1)

     LDR R0, [R2]

     ADD R0, R0, #1

     STR R0, [R2]
```

In this case, the same effective deadlock can occur, as it permissible that P1 can read the result of its own store to P1 early (and continue to do so for an indefinite amount of time). The insertion of a barrier again solves this effective deadlock:

```
P1:

     STR R0, [R2]

     DMB

     WAIT ([R2)==2)

P2:

     WAIT ([R2]==1)

     LDR R0, [R2]

     ADD R0, R0, #1

     STR R0, [R2]
```

6.7 WFE and WFI and Barriers

The Wait For Event and Wait For Interrupt instructions provide the capability to suspend execution and enter a low-power state. An explicit DSB barrier is required if it is necessary to ensure memory accesses that have been made before the WFI or WFE are visible to other observers and no other mechanism (such as the DMB that discussed in section 6.6, or a dependency on a load) has ensured this.

For example, in the following example, the DSB is required ensure that the store is visible:

```
P1:

     STR R0, [R2]
```

```
      DSB
Loop
      WFI
      B Loop
```

However, if the example in section 6.6 is enhanced to include a WFE as shown, no risk of a deadlock arises:

```
P1:
      STR R0, [R2]
      DMB
loop
      LDR R12, [R3]
      CMP R12, #1
      WFENE
      BNE loop


P2:
     WAIT ([R2]==1)
     STR R0, [R3]
     DSB
     SEV
```

In this case, the P1's DMB ensures that P1's store is observed by P2 before P1's load is observed by P2, and the dependency of the WFE on the result of the P1's load means that P1's load must be complete before the WFE is executed. More discussion of the SEV case is covered in section 7.3

7 LOAD EXCLUSIVE/STORE EXCLUSIVE AND BARRIERS

7.1 Introduction

The Load Exclusive and Store Exclusive instructions (LDREX* and STREX*) are predictable only with Normal memory. The instructions do not have any implicit barrier functionality defined as part of their operations, and so any use of Load Exclusive and Store Exclusive to implement locks of any type require the explicit addition of barriers.

7.2 Acquiring and Releasing a Lock

7.2.1 Acquiring a Lock

A common use of Load Exclusive and Store Exclusive is to claim a lock to allow entry into a critical region. This is typically performed by testing a lock variable which indicates 0 for a free lock and some other value (commonly 1 or an identifier of the process holding the lock) for a taken lock.

The lack of implicit barriers as part of the Load Exclusive and Store Exclusive instructions means that it is common to require a DMB barrier after acquiring a lock (and before any loads or stores in the critical region) to ensure the successful claim of the lock is observed by all observers before they observe any subsequent loads or stores.

```
Px:

Loop

    LDREX R5, [R1]              ; read lock

    CMP R5, #0                  ; check if 0

    STREXEQ R5, R0, [R1]        ; attempt to store new value

    CMPEQ R5, #0                ; test if store suceeded

    BNE Loop                    ; retry if not

    DMB                         ; ensures that all subsequent accesses are observed after the

                                ; gaining of the lock is observed

    ; loads and stores in the critical region can now be performed
```

7.2.2 Releasing a Lock

The converse operation of releasing a lock does not typically require the use of Load Exclusive and Store Exclusive as there should only be a single observer which is able to write to the lock. However, it is typically necessary for any memory updates that have been made or any memory values that have been loaded into memory to have been observed by any observer before the release of the lock is observed. As a result, the lock release is usually preceded by a DMB.

```
Px:

    ; loads and stores in the critical region

    MOV R0, #0

    DMB                         ; ensure all previous accesses are observed before the lock is

                                ; cleared

    STR R0, [R1]                ; clear the lock.
```

7.3 Use of Wait For Event (WFE) and Send Event (SEV) with Locks

The ARMv7 architecture introduced Wait For Event and Send Event to provide a mechanism to help reduce the number of iterations round a lock acquire loop (a spinlock), and so reduce power. The basic mechanism involves an observer that is in a spinlock executing a Wait For Event instruction which suspends execution on that observer until an asynchronous exception or an explicit event (sent by some observer using the SEV instruction) is

seen by that observer. The observer that holds the lock uses the SEV instruction to send an event after a lock has been released.

The Event signal is seen to be a non-memory communication, and as such the update to memory releasing the lock must actually be observable by all observers when the SEV instruction is executed and the event is sent. This then requires the use of DSB rather than DMB.

The lock acquire code using WFE is therefore:

```
Px:

Loop
    LDREX R5, [R1]              ; read lock
    CMP R5, #0                  ; check if 0
    WFENE                       ; sleep if the lock is held
    STREXEQ R5, R0, [R1]        ; attempt to store new value
    CMPEQ R5, #0                ; test if store suceeded
    BNE Loop                    ; retry if not
    DMB                         ; ensures that all subsequent accesses are observed after the
                                ; gaining of the lock is observed
    ; loads and stores in the critical region can now be performed
```

And the lock release code using SEV is therefore:

```
Px:
    ; loads and stores in the critical region
    MOV R0, #0
    DMB                         ; ensure all previous accesses are observed before the lock is
                                ; cleared
    STR R0, [R1]                ; clear the lock.
    DSB                         ; ensure completion of the store that cleared the lock before
                                ; sending the event
    SEV
```

8 SENDING INTERRUPTS AND BARRIERS

8.1 Using a Mailbox to send an interrupt

In some message passing systems, it is common for one observer to update memory and then send an interrupt using a mailbox of some sort to a second observer to indicate that memory has been updated and the new contents have been read.

Even though the sending of the interrupt using a mailbox might be initiated using a memory access, a DSB barrier must be used to ensure the completion of previous memory accesses.

Therefore the following sequence is needed to ensure that P2 sees the updated value.

```
P1:
    STR R5, [R1]          ; message stored to shared memory location
    DSB [ST]
    STR R1, [R4]          ; R4 contains the address of a mailbox


 P2:
    ; interrupt service routine
    LDR R5, [R1]
```

Even if R4 is a pointer to Strongly-Ordered memory, the update to R1 might not be visible without the DSB executed by P1.

It should be appreciated that these rules are required in connection to the ARM Generic Interrupt Controller (GIC).

9 CACHE & TLB MAINTENANCE OPERATIONS AND BARRIERS

9.1 Data Cache maintenance operations

9.1.1 Message Passing to Uncached Observers

The ARM ARM requires that data cache maintenance operations and their effects are ordered by DMB. This allows the following message passing approaches to be used to communicate between cached observers and non-cached observers:

```
P1:
    STR R5, [R1]              ; update data (assumed to be in P1's cache)
    DCCMVAC R1                ; clean cache to point of coherency
    DMB                       ; ensure effects of the clean will be observed before the
                              ; flag is set
    STR R0, [R4]              ; send flag to external agent (Non-cacheable location)


E1:
    WAIT ([R4] == 1)          ; wait for the flag
    DMB                       ; ensure that flag has been seen before reading data
    LDR R5, [R1]              ; read the data
```

In this example, it is required that E1:R5== 0x55

9.1.2 Multi-processing Message Passing to Uncached Observers

The broadcast nature of the cache maintenance operations under the MP Extensions, combined with properties of barriers means that the Message Passing principle to Uncached observers extends quite straightforwardly:

```
P1:
    STR R5, [R1]              ; update data (assumed to be in P1's cache)
    DMB [ST]                  ; ensure new data observed before the flag is observed
    STR R0, [R2]              ; send a flag for P2


P2:
    WAIT ([R2] == 1)          ; wait for P1's flag
    DMB                       ; ensure cache clean is observed after P1's flag is observed
    DCCMVAC R1                ; clean cache to point of coherency - will clean P1's cache
    DMB                       ; ensure effects of the clean will be observed before the
                              ; flag to E1`is set
    STR R0, [R4]              ; send flag to E1


E1:
    WAIT ([R4] == 1)          ; wait for P2's flag
    DMB                       ; ensure that flag has been seen before reading data
    LDR R5, [R1]              ; read data
```

In this example, it is required that E1:R5== 0x55; the clean operation executed by P2 has an effect on the data location held within P1's cache. The cast-out of P1's cache is guaranteed to be seen before P2's store the [R4]

9.1.3 Invalidating DMA Buffers - Non-functional example

The basic scheme for communicating with an external observer that is passing in data to a Cacheable region memory must take into account the principle that regions marked as Cacheable can be allocated into a cache at any time, for example as a result of speculation.

```
P1:
    DCIMVAC R1            ; ensure cache clean wrt memory. A clean operation could be used
                          ; but as the DMA will subsequently overwrite this region an
                          ; invalidate operation is sufficient and usually more efficient
    DMB                   ; ensures cache invalidation is observed before the next store
                          ; is observed
    STR R0, [R3]          ; send flag to external agent
    WAIT ([R4]==1)        ; wait for a different flag from an external agent
    DMB                   ; ensure that flag from external agent is observed before reading
```

```
                        ; new data. However [R1] could have been brought into cache earlier
    LDR R5, [R1]


E1:
    WAIT ([R3] == 1)    ; wait for flag
    STR R5, [R1]        ; store new data
    DMB
    STR R0, [R4]        ; send a flag
```

In this example, there is no guarantee that as a result of speculative access, the cache line containing [R1] will not be brought back into the cache after the cache invalidation, but before [R1] has been written by E1. As a result, the result P1:R5=0 is permissible.

9.1.4 Invalidating DMA Buffers - Functional example with 1 processor

```
P1:
    DCIMVAC R1          ; ensure cache clean wrt memory. A clean operation could be used
                        ; but as the DMA will subsequently overwrite this region an
                        ; invalidate operation is sufficient and usually more efficient
    DMB                 ; ensures cache invalidation is observed before the next store
                        ; is observed
    STR R0, [R3]        ; send flag to external agent
    WAIT ([R4]==1)      ; wait for a different flag from an external agent
    DMB                 ; ensure that cache invalidate is observed after the flag
                        ; from external agent is observed
    DCIMVAC R1          ; ensure cache discards stale copies before use
    LDR R5, [R1]


E1:
    WAIT ([R3] == 1)    ; wait for flag
    STR R5, [R1]        ; store new data
    DMB [ST]
    STR R0, [R4]        ; send a flag
```

In this example, the result P1:R5 == 0x55 is required. The inclusion of an invalidation of the cache after the store by E1 to [R1] has been observed will ensure that the line is fetched from external memory after it has been updated.

9.1.5 Invalidating DMA Buffers - Functional example with multiple coherent processors

The broadcasting of cache maintenance operations and the use of DMB to ensure their observability means that this example extends naturally to a multi-processor system. Typically this example would require a transfer of ownership of the region that the external observer is updating.

```
P0:
    (Use data from [R1], potentially using [R1] as scratch space)
    DMB
    STR R0, [R2]                ; signal release of [R1]
    WAIT ([R2] == 0)            ; wait for new value from DMA
    DMB
    LDR R5, [R1]


P1:
    WAIT ([R2] == 1)            ; wait for release of [R1] by P0
    DCIMVAC R1                  ; ensure caches are clean wrt memory, invalidate is sufficient
    DMB
    STR R0, [R3]                ; request new data for [R1]
    WAIT ([R4] == 1)            ; wait for new data
    DMB
    DCIMVAC R1                  ; ensure caches discard stale copies before use
    DMB
    MOV R0, #0
    STR R0, [R2]                ; signal availability of new [R1]


E1:
    WAIT ([R3] == 1)            ; wait for new data request
    STR R5, [R1]                ; send new [R1]
    DMB [ST]
    STR R0, [R4]                ; indicate new data available to P1
```

In this example, the result P0:R5 == 0x55 is required. The DMB issued by P1 after the first data cache invalidation ensures that effect of the cache invalidation on P0 is seen by E1 before the store by E1 to [R1]. The DMB issued by P1 after the second data cache invalidation ensures that its effects are seen before the store of 0 to semaphore location in [R2].

9.2 Instruction Cache Maintenance operations

9.2.1 Ensuring the visibility of updates to instructions for a uniprocessor

On a single core, the agent that causes instruction fetches (including instruction cache linefills) is treated as a separate observer to the memory system from the agent that causes data accesses. As a result, any operations to invalidate the instruction cache can only rely on seeing updates to memory that have completed. This is ensured by the use of a DSB.

In addition instruction cache maintenance operations are only guaranteed to complete after the execution of a DSB, and an ISB is required to discard any instructions which might have been prefetched before the instruction cache invalidation had been completed. Therefore the following sequence is required on a uniprocessor to ensure the visibility of an update to code and to branch to it.

```
P1:
    STR R11, [R1]              ; R11 contains a new instruction to stored in program memory
    DCCMVAU R1                 ; clean to PoU makes visible to instruction cache
    DSB
    ICIMVAU R1                 ; ensure instruction cache/branch predictor discard stale data
    BPIMVA R1
    DSB                        ; ensure completion of the invalidation
    ISB                        ; ensure instruction fetch path sees new I cache state
    BX R1
```

9.2.2 Ensuring the visibility of updates to instructions for a multiprocessor

The multiprocessing extensions require that a DSB is executed by the processor which issued an instruction cache maintenance instruction to ensure its completion, this also ensures that the maintenance operation is completion on all cores within the Shareable, Not Outer-Shareable domain.

ISB is not broadcast, and so does not have an effect on other cores. This requires that other cores perform their own ISB synchronisation once it is known that the update is visible, if it is necessary to ensure the synchronisation of those other cores.

```
P1:
    STR R11, [R1]              ; R11 contains a new instruction to stored in program memory
    DCCMVAU R1                 ; clean to PoU makes visible to instruction cache
    DSB                        ; ensure completion of the clean on all processors
    ICIMVAU R1                 ; ensure instruction cache/branch predictor discard stale data
    BPIMVA R1
    DSB                        ; ensure completion of the ICache and branch predictor
                               ; invalidation on all processors
    STR R0, [R2]               ; set flag to signal completion
    ISB                        ; synchronize context on this processor
```

```
    BX R1                       ; branch to new code


P2-PX:
    WAIT ([R2] == 1)            ; wait for flag signalling completion
    ISB                         ; synchronize context on this processor
    BX R1; branch to new code
```

9.2.2.1 Non-functional approach

The following sequence does not have the same effect, as DSB is not required to complete the instruction cache maintenance operations that have been issued by other cores:

```
P1:
    STR R11, [R1]               ; R11 contains a new instruction to stored in program memory
    DCCMVAU R1                  ; clean to PoU makes visible to instruction cache
    DSB                         ; ensure completion of the clean on all processors
    ICIMVAU R1                  ; ensure instruction cache/branch predictor discard stale data
    BPIMVA R1
    DMB                         ; ensure ordering of the store after the invalidation
                                ; DOES NOT guarantee completion of instruction cache/branch
                                ; predictor on other processors
    STR R0, [R2]                ; set flag to signal completion
    DSB                         ; ensure completion of the invalidation on all processors
    ISB                         ; synchronize context on this processor
    BX R1                       ; branch to new code



P2-PX:
    WAIT ([R2] == 1)            ; wait for flag signalling completion
    DSB                         ; this DSB does not guarantee completion of P1's
                                ; ICIMVAU/BPIMVA
    ISB
    BX R1
```

In this case, P2…PX might not see the updated region of code at R1.

9.3 TLB Maintenance operations and Barriers

9.3.1 Ensuring the visibility of updates to translation tables for a uniprocessor.

On a single core, the agent that causes translation table walks is treated as a separate observer to the memory system from the agent that causes data accesses. As a result, any operations to invalidate the TLB can only rely on seeing updates to memory that have completed. This is ensured by the use of a DSB.

The MP Extensions require that the translation table walks look in the data or unified caches at L1, so there is no need for data cache cleaning in such systems.

Once the translation tables have been updated, any old copies of the entries which are cached in the TLBs need to be invalidated. This operation is only guaranteed to have taken effect to all instructions (including instruction fetches and data accesses) after the execution of a DSB and an ISB. Thus the code for updating a translation table entry is:

```
P1:

    STR R11, [R1]               ; update the translation table entry

    DSB                         ; Ensure visibility of the update to translation table walks

    TLBIMVA R10

    BPIALL

    DSB                         ; ensure completion of the BP and TLB invalidation

    ISB                         ; synchronise context on this processor
    ; new translation table entry can be relied upon at this point and all accesses
    ; generated by this observer using
    ; the old mapping have been completed.
```

Importantly, by the end of this sequence, all accesses which used the old translation table mappings will have been observed by all observers.

A particular example of this is where a translation table entry is being marked as being invalid. Such a system must provide a mechanism to ensure that any accesses to a region of memory being marked as invalid have completed before some action is taken based on the region being marked as invalid.

9.3.2 Ensuring the visibility of updates to translation tables for a multiprocessor

The same code sequence can be used within a multiprocessing system. The multiprocessing extensions require that a DSB is executed by the processor which issued the TLB maintenance instruction to ensure its completion, but that the completion is ensured on all cores within the Shareable, Not Outer-Shareable domain.

The completion of a DSB that completes a TLB maintenance operation ensures that all accesses which used the old mapping have been completed.

```
P1:

   STR R11, [R1]               ; update the translation table entry

    DSB                        ; Ensure visibility of the update to translation table walks

    TLBIMVAIS R10

    BPIALLIS
```

```
DSB                          ; ensure completion of the BP and TLB invalidation =

ISB                          ; Note ISB is not broadcast and must be executed locally

                             ; on other processors
;  new translation table entry can be relied upon at this point and all accesses
;  generated by any observers affected by the broadcast TLBIMVAIS operation using
; the old mapping have been completed.
```

It should be appreciated that the completion of the TLB maintenance instruction is only guaranteed by the execution of a DSB by the observer that executed the TLB maintenance instruction. The execution of a DSB by a different observer does not have this effect, even if the DSB is known to be executed after the TLB maintenance instruction has been observed by that different observer.

9.3.3 Paging memory in and out.

A particular example for ensuring the visibility of updates from translation tables in a multiprocessor situation comes from the needs of paging regions of memory into RAM from a backing store. This may, or may not, also involve paging existing locations in memory out of RAM to a backing store. In such situations, the operating system selects a page (or more) of memory that might be in use but is suitable to be discarded (with or without copying to a back store, depending on whether or not the region of memory is writeable). Access to this page by any agents is prohibited by disabling the translation table mappings for that page (and ensuring the visibility of those updates).

For this reason, it is important that the DSB which is performed after the TLB invalidation ensures that no other updates to memory using those mappings are possible.

A typical sequence for paging out of an updated region of memory and the subsequent paging in of memory is as follows:

P1:

```
STR R11, [R1] ; update the translation table for the region being paged out

DSB           ; Ensure visibility of the update to translation table walks

TLBIMVAIS R10  ; invalidate the old entry

DSB           ; ensure completion of the invalidation on all processors

ISB           ; ensure visibility of the invalidation

BL SaveMemoryPageToBackingStore

BL LoadMemoryFromBackingStore

DSB           ; ensure completion of the memory transfer (this could be part of

              ; LoadMemoryFromBackingStore

ICIALLIS      ; Also invalidates the branch predictor

STR R9, [R1] ; create a new translation table entry with a new mapping

DSB          ; ensure completion of the I Cache & Branch Predictor invalidation

             ; AND ensure visibility of the new translation table mapping

ISB          ; ensure synchronisation of this instruction stream
```

The details of how the functions `SaveMemoryPageToBackingStore` and `LoadMemoryFromBackingStore` work are omitted for clarity, but the copies of memory are assumed to be performed by an observer that is coherent with the caches of processor P1 (and may in fact be P1 itself, using a specific paging mapping). `LoadMemoryFromBackingStore` is required to ensure that the memory updates that it makes are visible to instruction fetches.

The use of `ICIALLIS` to invalidate the entire instruction cache is a simplification in this case, but may not be optimal for performance. An alternative approach involves invalidating all of the lines in the caches using `ICIMVAU` operations; however this must be done when the mapping being used for the `ICIMVAU` operations is valid but not executable.