many examples of how you have to explicitly flush the prefetch and branch target buffer after changing coprocessor 15 state.

Chapter B2
Memory Order Model

This chapter provides a high-level overview of the memory order model. It contains the following sections:

B2.1 About the memory order model

The architecture prior to ARMv6 did not attempt to define the acceptable memory ordering of explicit memory transactions, describing the regions of memory according to the hardware approaches that had previously been used to implement such memory systems. Thus regions of memory had been termed as being one of *Write-Through Cacheable*, *Write-Back Cacheable*, *Non-Cacheable Bufferable* or *Non-Cacheable, Non-Bufferable*. These terms are based on the previous hardware implementations of cores and the exact properties of the memory transactions could not be rigorously inferred from the memory names. Implementations have chosen to interpret these names in different ways, leading to potentially incompatible uses.

In a similar manner, the order in which memory accesses could be presented to memory was not defined, and in particular there was no definition of what order could be relied upon by an observer of the memory transactions generated by a processor. As implementations and systems become more complicated, these undefined areas of the architecture move from being simply based on a standard default to having the potential of presenting significant incompatibilities between different implementations; at processor core and system level.

ARMv6 introduces a set of memory types - Normal, Device, and Strongly Ordered - with memory access properties defined to fit in a largely backwards compatible manner to the defacto meanings of the original memory regions. A potential incompatibility has been introduced with the need for a software polling policy when it is necessary for the program to be aware that memory accesses to I/O space have completed, and all side effects are visible across the whole system. This reflects the increasing difficulty of ensuring linkage between the completion of memory accesses and the execution of instructions within a complex high-performance system.

A *shared* memory attribute to indicate whether a region of memory is shared between multiple processors (and therefore requires an appearance of cache transparency in an ordering model) is also introduced. Implementations remain free to choose the mechanisms to implement this functionality.

The key issues with the memory order model are slightly different depending on the target audience:

- for software programmers, the key factor is that side effects are only architecturally visible after software polling of a location that indicates that it is safe to proceed
- for silicon Implementors, the *Strongly Ordered and Device* memory attributes defined in this chapter place certain restrictions on the system designer in terms of what they are allowed to build, and when to indicate completion of a transaction.

Additional attributes and behaviors relate to the memory system architecture. These features are defined in other areas of this manual:

- Virtual memory systems based on an MMU described in Chapter B4 *Virtual Memory System Architecture*.
- Protected memory systems based on an MPU described in Chapter B5 *Protected Memory System Architecture*.
- Caches and write buffers described in Chapter B6 *Caches and Write Buffers*.
- Tightly Coupled Memory (TCM) described in Chapter B7 *Tightly Coupled Memory*

Some attributes are described in relation to an MMU for ARMv6. In general, these can also be applied to an MPU based system.

B2.2 Read and write definitions

Memory accesses can be either reads or writes.

B2.2.1 Reads

Reads are defined as memory operations that have the semantics of a load.

In the ARM® instruction set, these are:

- `LDM`, `LDRH`, `LDRSH`, `LDRB`, `LDRSB`
- `LDM`, `LDRD`, `LDRT`, `LDRBT`,
- `LDC`, `RFE`, `SWP`, `SWPB`, `LDREX`, `STREX`.

In the Thumb® instruction set, they are:

- `LDR`, `LDRH`, `LDRSH`, `LDRB`, `LDRSB`
- `LDM`, `POP`.

Jazelle® opcodes that are accelerated by hardware can cause a number of reads to occur, according to the state of the operand stack and the implementation of the Jazelle hardware acceleration.

B2.2.2 Writes

Writes are defined as operations that have the semantics of a store.

In the ARM instruction set, these are:

- `STR`, `STRH`, `STRB`
- `STM`, `STRD`, `STRT`, `STRBT`
- `STC`, `SRS`, `SWP`, `SWPB`, `STREX`

In the Thumb instruction set, they are:

- `STR`, `STRH`, `STRB`
- `STM`, `PUSH`

Jazelle opcodes that are accelerated by hardware can cause a number of writes to occur, according to the state of the operand stack and the implementation of the Jazelle hardware acceleration.

B2.2.3 Memory synchronization primitives

Synchronization primitives are required to ensure correct operation of system semaphores within the memory order model. The memory synchronization primitive instructions are defined as those instructions that are used to ensure memory synchronization:

- `LDREX`, `STREX`
- `SWP`, `SWPB` (deprecated in ARMv6).

Prior to ARMv6, support consisted of the `SWP` and `SWPB` instructions. ARMv6 has introduced new `LDREX` and `STREX` (Load and Store Exclusive) instructions. See *Memory barriers* on page B2-18 for the architecture details.

LDREX and STREX are supported to shared and non-shared memory. Non-shared memory can be used when the processes to be synchronized are running on the same processor. When the processes to be synchronized are running on different processors, shared memory must be used.

B2.2.4 Observability and completion

The concept of observability applies to all memory, however, the concept of global observability only applies to shared memory. Normal, Device and Strongly Ordered memory are defined in *ARMv6 memory attributes - introduction* on page B2-8.

For all memory:

- A write to a location in memory is said to be observed by a memory system agent when a subsequent read of the location by the same memory system agent returns the value written by the write.
- A write to a location in memory is said to be globally observed when a subsequent read of the location by any memory system agent returns the value written by the write.
- A read to a location in memory is said to be observed by a memory system agent when a subsequent write of the location by the same memory system agent has no effect on the value returned by the read.
- A read to a location in memory is said to be globally observed when a subsequent write of the location by any memory system agent has no effect on the value returned by the read.

Additionally, for Strongly Ordered memory:

- A read or write to a memory mapped location in a peripheral which exhibits side-effects is said to be observed, and globally observed, only when the read or write meets the general conditions listed, can begin to affect the state of the memory-mapped peripheral, and can trigger any side effects that affect other peripheral devices, cores and/or memory.

For all memory, the completion rules are:

- A read or write is defined to be complete when it is globally observed and any page table walks associated with the read or write are complete.
- A page table walk is defined to be complete when the memory transactions associated with the page table walk are globally observed, and the TLB is updated.
- A cache, branch predictor or TLB maintenance operation is defined to be complete when the effects of operation are globally observed and any page table walks which arise are complete.

Note

For all memory-mapped peripherals, where the side-effects of a peripheral are required to be visible to the entire system, the peripheral must provide an IMPLEMENTATION DEFINED location which can be read to determine when all side effects are complete.

Side effect completion in Strongly Ordered and Device memory

To determine when any side effects have completed, it is necessary to poll a location associated with the device, for example, a status register. This is a key element of the architected memory order model.

B2.3 Memory attributes prior to ARMv6

Prior to ARMv6, all memory has been tagged with a combination of two control bits in the ARM virtual and protected memory management models, VMSA and PMSA respectively. The bits are:

- a bufferable (B) bit (allow write buffering between the core and memory)
- a cacheable (C) bit.

These are traditionally interpreted to define the memory behavior of a given location as shown in Table B2-1.

Table B2-1 Interpretation of cacheable and bufferable bits

C	B	Write-through cache	Write-back only cache	Write-back/write-through cache
0	0	Uncached/unbuffered	Uncached/unbuffered	Uncached/unbuffered
0	1	Uncached/buffered	Uncached/buffered	Uncached/buffered
1	0	IMPLEMENTATION DEFINED	UNPREDICTABLE	Write-through cached/buffered
1	1	Cached/buffered	Cached/buffered	Write-back cached/buffered

B2.4 ARMv6 memory attributes - introduction

ARMv6 defines a set of memory attributes with the characteristics required to support all memory and devices in the system memory map. The ordering of accesses for regions of memory is also defined by the memory attributes.

There are three mutually exclusive main memory type attributes to describe the memory regions:

- Normal
- Device
- Strongly Ordered.

Normal memory is idempotent, exhibiting the following properties:

- write transactions can be repeated with no side effects
- repeated read transactions return the last value written to the resource being read
- transactions can be restarted if interrupted
- multibyte accesses need not be atomic, and can be restarted or replayed
- unaligned accesses can be supported
- transactions can be merged prior to accessing the target memory system
- read transactions can prefetch additional memory locations with no side effects.

System peripherals (I/O) generally conform to different access rules; defined in ARMv6 as Strongly Ordered or Device memory. Examples of I/O accesses are:

- FIFOs where consecutive accesses add (write) or remove (read) queued values
- interrupt controller registers where an access can be used as an interrupt acknowledge changing the state of the controller itself
- memory controller configuration registers that are used to set up the timing (and correctness) of areas of normal memory
- memory-mapped peripherals where the accessing of memory locations causes side effects within the system.

To ensure system correctness, access rules are more restrictive than those to normal memory:

- accesses (reads and writes) can have side effects
- transactions must not be repeated, for example, on return from an exception
- transaction number, size and order must be maintained.

In addition, the Shared attribute indicates whether the memory is private to a single processor, or accessible from multiple processors or other bus master resources, for example, an intelligent peripheral with DMA capability.

Table B2-2 on page B2-9 shows a summary of the memory attributes.

Table B2-2 Memory attribute summary

Memory type attribute	Shared attribute	Other attributes	Description
Strongly Ordered	-		All memory accesses to Strongly Ordered memory occur in program order. All Strongly Ordered accesses are assumed to be Shared.
Device	Shared		Designed to handle memory mapped peripherals that are shared by several processors.
	Non-Shared		Designed to handle memory mapped peripherals that are used only by a single processor.
Normal	Shared	Non-cacheable/ Write-Through cacheable/ Write-Back cacheable	Designed to handle normal memory which is shared between several processors.
	Non-Shared	Non-cacheable/ Write-Through cacheable/ Write-Back cacheable	Designed to handle normal memory which is used only by a single processor.

B2.4.1 Normal memory attribute

This attribute is defined for each page in an MMU, can be further defined as being Shared or Non-Shared, and describes most memory used in a system. It is designed to provide memory access orderings that are suitable for Normal memory. Such memory stores information without side effects. Normal memory may be read/write or read-only.

For writable Normal memory unless there is a change to the physical address mapping:

- A load from a specific location will return the most recently stored data at that location for the same processor.
- Two loads from a specific location, without a store in between, will return the same data for each load.

For read-only Normal memory:

- Two loads from a specific location will return the same data for each load.

Accesses to Normal Memory conform to the weakly-ordered model of memory ordering. A description of the weakly-ordered model can be found in standard texts describing memory ordering issues. A recommended text is chapter 2 of *Memory Consistency Models for Shared Memory-Multiprocessors*, Kourosh Gharachorloo, Stanford University Technical Report CSL-TR-95-685.

All explicit accesses must correspond to the ordering requirements of accesses described in *Ordering requirements for memory accesses* on page B2-16.

Non-shared Normal memory

The Non-Shared Normal memory attribute is designed to describe normal memory that can be accessed only by a single processor.

actually, should we mark as shared in order to get stronger guarantees?

A region of memory marked as Non-Shared Normal does not have any requirement to make the effect of a cache transparent. For regions of memory marked as Non-shared Non-cacheable, a DMB memory barrier must be used in situations where the forwarding of data from the internal buffering of previous accesses within the single processor is required.

Shared Normal memory

The Shared Normal memory attribute is designed to describe normal memory that can be accessed by multiple processors or other system masters.

A region of memory marked as Shared Normal is one in which the effect of interposing a cache (or caches) on the memory system is entirely transparent to data accesses. Explicit software management is still required to ensure coherency of instruction caches. Implementations can use a variety of mechanisms to support this, from very simply not caching accesses in shared regions to more complex hardware schemes for cache coherency for those regions.

Writes to Shared Normal Memory may not be atomic, that is, all observers might not see the writes occurring at the same time. To preserve coherence where two writes are made to the same location, it is required that the order of those writes is seen to be the same by all observers. Reads to Shared Normal Memory that are aligned in memory to the size of the access must be atomic.

Cacheable write-through, cacheable write-back and non-cacheable memory

In addition to marking a region of normal memory as being Shared or Non-Shared, each page of memory marked in an MMU as Normal can also be marked as being one of:

- cacheable write-through
- cacheable write-back
- non-cacheable.

This marking is independent of the marking of a region of memory as being Shared or Non-Shared. It indicates the required handling of the data region for reasons other than those to handle the requirements of shared data. As a result, it is acceptable for a region of memory that is marked as being cacheable and shared not to be held in the cache in an implementation which handles shared regions as not caching the data.

If the same memory locations are marked as having different cacheable attributes, for example by the use of synonyms in a virtual to physical address mapping, UNPREDICTABLE behavior results.

B2.4.2 Device memory attribute

The Device memory attribute is defined for memory locations where an access to the location can cause side effects, or where the value returned for a load can vary depending on the number of loads performed. Memory mapped peripherals and I/O locations are typical examples of areas of memory that should be marked as being Device. The Device attribute is defined for each page in an MMU.

Explicit accesses from the processor to regions of memory marked as Device occur at the size and order defined by the instruction. The number of accesses that occur to such locations is the number that is specified by the program. Implementations must not repeat accesses to such locations when there is only one access in the program, that is, the accesses are not *restartable*. An example where an implementation might want to repeat an access is before and after an interrupt, in order to allow the interrupt to cause a slow access to be abandoned. Such implementation optimizations must not be performed for regions of memory marked as Device.

how to mark
as device?

In addition, address locations marked as Device are non-cacheable. While writes to device memory may be buffered, writes shall only be merged where the correct number of accesses, order, and their size is maintained. Multiple accesses to the same address cannot change the number of accesses to that address. Coalescing of accesses is not permitted in this case.

Accesses to memory mapped locations that have side effects that apply to Normal memory locations require Memory Barriers to ensure correct execution. An example is the programming of the configuration registers of a memory controller with respect to the memory accesses it controls.

All explicit accesses to memory marked as Device must correspond to the ordering requirements of accesses described in *Ordering requirements for memory accesses* on page B2-16.

Shared attribute

The Shared attribute is defined for each page in an MMU. These regions can be referred to as:

- memory marked as Shared Device
- memory marked as Non-Shared Device.

Memory marked as Non-Shared Device is defined as only accessible by a single processor. An example of a system supporting Shared and Non-shared Device memory is an implementation that supports a local bus for its private peripherals, whereas system peripherals are situated on the main (Shared) system bus. Such a system might have more predictable access times for local peripherals such as watchdog timers or interrupt controllers.

B2.4.3 Strongly Ordered memory attribute

The Strongly Ordered memory attribute is defined for each page in the MMU. Accesses to memory marked as Strongly Ordered have a strong memory-ordering model for all explicit memory accesses from that processor. An access to memory marked as Strongly Ordered is required to act as if a DMB memory barrier were inserted before and after the access from that processor. See *DataMemoryBarrier (DMB) CP15 register 7* on page B2-18.

To maintain backwards compatibility with ARMv5, any ARMv5 instructions that implicitly or explicitly change the interrupt masks in the CSPR and appear in program order after a Strongly Ordered access must wait for the Strongly Ordered memory access to complete. These instructions are `MSR`, with the control field mask bit set, and the flag-setting variants of arithmetic and logical instructions with R15 as the destination register (these copy the SPSR to CSPR). This requirement exists only for backwards compatibility with previous versions of the ARM architecture; the behavior is deprecated in ARMv6. ARMv6 compliant programs must not rely on this behavior, but instead include an explicit Memory Barrier between the memory access and the following instruction, see *DataSynchronizationBarrier (DSB) CP15 register 7* on page B2-18 when synchronization is required.

Explicit accesses from the processor to memory marked as Strongly Ordered occur at their program size, and the number of accesses that occur to such locations is the number that are specified by the program. Implementations must not repeat accesses to such locations when there is only one access in the program, that is, the accesses are not *restartable*.

Address locations marked as Strongly Ordered are not held in a cache, and are always treated as Shared memory locations.

All explicit accesses to memory marked as Strongly Ordered must correspond to the ordering requirements of accesses described in *Ordering requirements for memory accesses* on page B2-16.

B2.4.4 Memory access restrictions

The following restrictions apply to memory accesses:

- For any access X, the bytes accessed by X must all have the same memory type attribute, otherwise, the behavior of the access is UNPREDICTABLE. That is, unaligned accesses that span a boundary between different memory types are UNPREDICTABLE.
- For any two memory accesses X and Y, such that X and Y are generated by the same instruction, X and Y must all have the same memory type attribute, otherwise, the results are UNPREDICTABLE. For example, an `LDC`, `LDM`, `LDRD`, `STC`, `STM`, or `STRD` that spans a boundary between Normal and Device memory is UNPREDICTABLE.
- Instructions that generate unaligned memory accesses to Device or Strongly Ordered memory are UNPREDICTABLE.
- Memory operations which cause multiple transactions to Device or Strongly Ordered memory should not crosses a 4KB address boundary to ensure access rules are maintained. For this reason, it is important that accesses to volatile memory devices are not made using single instructions that cross a 4KB address boundary. This restriction is expected to cause restrictions to the placing of such devices in the memory map of a system, rather than to cause a compiler to be aware of the alignment of memory accesses.
- For instructions that generate accesses to Device or Strongly Ordered memory, implementations do not change the sequence of accesses specified by the pseudo-code of the instruction. This includes not changing how many accesses there are, nor their time order, nor the data sizes and other properties of each individual access. Furthermore, processor core implementations expect any attached memory system to be able to identify accesses by memory type, and to obey similar restrictions with regard to the number, time order, data sizes and other properties of the accesses.

  Exceptions to this rule are:

  — An implementation of a processor core can break this rule, provided that the information it does supply to the memory system enables the original number, time order, and other details of the accesses to be reconstructed. In addition, the implementation must place a requirement on attached memory systems to do this reconstruction when the accesses are to Device or Strongly Ordered memory.

  For example, the word loads generated by an `LDM` might be paired into 64-bit accesses by an implementation with a 64-bit bus. This is because the instruction semantics ensure that the 64-bit access is always a word load from the lower address followed by a word load from the higher address, provided a requirement is placed on memory systems to unpack the two word loads where the access is to Device or Strongly Ordered memory.

  — Any implementation technique that produces results that cannot be observed to be different from those described above is legitimate.
- Multi-access instructions that load or store R15 must only access normal memory. If they access Device or Strongly Ordered memory the results are UNPREDICTABLE.

- Instruction fetches must only access normal memory. If they access Device or Strongly Ordered memory, the results are UNPREDICTABLE. By example, instruction fetches must not be performed to areas of memory containing read-sensitive devices, because there is no ordering requirement between instruction fetches and explicit accesses.
- If the same memory location is marked as Shared Normal and Non-Shared Normal in a MMU, for example by the use of synonyms in a virtual to physical address mapping, UNPREDICTABLE behavior results.
- If the same memory locations are marked as having different memory types (Normal, Device, or Strongly Ordered), for example by the use of synonyms in a virtual to physical address mapping, UNPREDICTABLE behavior results.
- If the same memory locations are marked as having different cacheable attributes, for example by the use of synonyms in a virtual to physical address mapping, UNPREDICTABLE behavior results.
- If the same memory location is marked as being Shared Device and Non-Shared Device in an MMU, for example by the use of synonyms in a virtual to physical address mapping, UNPREDICTABLE behavior results.

Note

Implementations must also ensure that prefetching down non-sequential paths, for example, as a result of a branch predictor, cannot cause unwanted accesses to read-sensitive devices. Implementations may prefetch by an IMPLEMENTATION DEFINED amount down a sequential path from the instruction currently being executed.

crazy

Prior to ARMv6, it is IMPLEMENTATION DEFINED whether a low interrupt latency mode is supported. From ARMv6, low interrupt latency support is controlled from the System Control coprocessor (FI-bit). It is IMPLEMENTATION DEFINED whether multi-access instructions behave correctly in low interrupt latency configurations.

B2.4.5 Backwards compatibility

ARMv6 memory attributes are significantly different from those in previous versions of the architecture. Table B2-3 shows the interpretation of the earlier memory types in the light of this definition.

Table B2-3 Backwards compatibility

Previous architectures	ARMv6 attribute
NCNB (Non-cacheable, Non-Bufferable)	Strongly Ordered [a]
NCB (Non-cacheable, Bufferable)	Shared Device [a]
Write-Through cacheable, Bufferable	Non-Shared Normal (Write-Through cacheable)
Write-Back cacheable, Bufferable	Non-Shared Normal (Write-Back cacheable)

[a] Memory locations contained within the TCMs are treated as being Non-Cacheable, not Strongly Ordered or Shared Device

B2.5 Ordering requirements for memory accesses

ARMv6 defines access restrictions in the memory ordering allowed, depending on the memory attributes of the accesses involved. Figure B2-1 shows the memory ordering between two explicit accesses A1 and A2, where A1 occurs before A2 in program order.

The symbols used in Figure B2-1 are as follows:

< Accesses must be globally observed in program order, that is, A1 must be globally observed strictly before A2.

(blank) Accesses can be globally observed in any order, provided that the requirements of uniprocessor semantics, for example respecting dependencies between instructions within a single processor, are maintained.

A2	Normal Read	Device Read		Strongly Ordered Read	Normal Write	Device Write		Strongly Ordered Write
A1		Non-Shared	Shared			Non-Shared	Shared	
Normal Read				<				<
Device Read (Non-Shared)		<		<		<		<
Device Read (Shared)			<	<			<	<
Strongly Ordered Read	<	<	<	<	<	<	<	<
Normal Write				<				<
Device Write (Non-Shared)		<		<		<		<
Device Write (Shared)			<	<			<	<
Strongly Ordered Write	<	<	<	<	<	<	<	<

Figure B2-1 Memory ordering restrictions

There are no ordering requirements for implicit accesses to any type of memory.

B2.5.1 Program order for instruction execution

Program order of instruction execution is the order of the instructions in the control flow trace.

Explicit memory accesses in an execution can be either:

Strictly Ordered Denoted by <. Must occur strictly in order.

Ordered Denoted by <=. Must occur either in order, or simultaneously.

Multiple load and store instructions, such as LDM, LDRD, STM, and STRD, generate multiple word accesses, each of which is a separate access for the purpose of determining ordering.

The rules for determining program order for two accesses A1 and A2 are:

If A1 and A2 are generated by two different instructions:

- A1 < A2 if the instruction that generates A1 occurs before the instruction that generates A2 in program order
- A2 < A1 if the instruction that generates A2 occurs before the instruction that generates A1 in program order.

If A1 and A2 are generated by the same instruction:

- If A1 and A2 are the load and store generated by a SWP or SWPB instruction:
  - — A1 < A2 if A1 is the load and A2 is the store
  - — A2 < A1 if A2 is the load and A1 is the store.
- If A1 and A2 are two word loads generated by an LDC, LDRD, or LDM instruction, or two word stores generated by an STC, STRD, or STM instruction, excluding LDM or STM instructions whose register list includes the PC:
  - — A1 <= A2 if the address of A1 is less than the address of A2
  - — A2 <= A1 if the address of A2 is less than the address of A1.
- If A1 and A2 are two word loads generated by an LDM instruction whose register list includes the PC or two word stores generated by an STM instruction whose register list includes the PC, the program order of the memory operations is not defined.
- If A1 and A2 are two word loads generated by an LDRD instruction or two word stores generated by an STRD instruction whose register list includes the PC, Rd equals R14 and the instruction is UNPREDICTABLE.

B2.6 Memory barriers

Memory barrier is the general term applied to an instruction, or sequence of instructions, used to force synchronization events by a processor with respect to retiring load/store instructions in a processor core. A memory barrier is used to guarantee completion of preceding load/store instructions to the programmers model, flushing of any prefetched instructions prior to the event, or both. ARMv6 mandates three explicit barrier instructions in the System Control Coprocessor to support the memory order model described in this chapter, and requires these instructions to be available in both privileged and user modes:

- DataMemoryBarrier as described in *DataMemoryBarrier (DMB) CP15 register 7*
- DataSynchronizationBarrier (DataWriteBarrier) as described in *DataSynchronizationBarrier (DSB) CP15 register 7*
- PrefetchFlush as described in *PrefetchFlush CP15 register 7* on page B2-19.

These instructions may be sufficient on their own, or may need to be used in conjunction with cache and memory management maintenance operations; operations which are only available in privileged modes. Support of memory barriers in earlier versions of the architecture is IMPLEMENTATION DEFINED.

Explicit memory barriers affect reads and writes to the memory system generated by load and store instructions being executed in the CPU. Reads and writes generated by L1 DMA transactions, and instruction fetches or accesses caused by a hardware page table access, are not explicit accesses.

B2.6.1 DataMemoryBarrier (DMB) CP15 register 7

DMB acts as a data memory barrier, exhibiting the following behavior:

- All explicit memory accesses by instructions occurring in program order before this instruction are globally observed before any explicit memory accesses due to instructions occurring in program order after this instruction are observed.
- DataMemoryBarrier has no effect on the ordering of other instructions executing on the processor.

As such, DMB ensures the apparent order of the explicit memory operations before and after the instruction, without ensuring their completion.

The encoding for DataMemoryBarrier is described in *Register 7: cache management functions* on page B6-19.

B2.6.2 DataSynchronizationBarrier (DSB) CP15 register 7

Note

This operation has historically been referred to as DrainWriteBuffer or DataWriteBarrier (DWB). From ARMv6, these names (and the use of DWB) are deprecated in favor of the new DataSynchronizationBarrier name and DSB. DSB better reflects the functionality provided in ARMv6; it is architecturally defined to include all cache, TLB and branch prediction maintenance operations as well as explicit memory operations.

The DataSynchronizationBarrier operation acts as a special kind of memory barrier. The DSB operation completes when:

- All explicit memory accesses before this instruction complete.
- All Cache, Branch predictor and TLB maintenance operations preceding this instruction complete.

In addition, no instruction subsequent to the DSB may execute until the DSB completes.

The encoding for DataSynchronizationBarrier is described in *Register 7: cache management functions* on page B6-19.

B2.6.3 PrefetchFlush CP15 register 7

The PrefetchFlush instruction flushes the pipeline in the processor, so that all instructions following the pipeline flush are fetched from cache or memory after the instruction has been completed. It ensures that the effects of context altering operations, such as changing the *Application Space IDentifier* (ASID), or completed TLB maintenance operations or branch predictor maintenance operations, as well as all changes to the CP15 registers, executed before the PrefetchFlush are visible to the instructions fetched after the PrefetchFlush.

In addition, the PrefetchFlush operation ensures that any branches which appear in program order after the PrefetchFlush are always written into the branch prediction logic with the context that is visible after the PrefetchFlush. This is required to ensure correct execution of the instruction stream.

Note

Any context altering operations appearing in program order after the PrefetchFlush only take effect after the PrefetchFlush has been executed. This is due to the behavior of the context altering instructions.

Note

ARM implementations are free to choose how far ahead of the current point of execution they prefetch instructions; either a fixed or a dynamically varying number of instructions. As well as being free to choose how many instructions to prefetch, an ARM implementation can choose which possible future execution path to prefetch along. For example, after a branch instruction, it can choose to prefetch either the instruction following the branch or the instruction at the branch target. This is known as branch prediction.

A potential problem with all forms of instruction prefetching is that the instruction in memory might be changed after it was prefetched but before it is executed. If this happens, the modification to the instruction in memory does not normally prevent the already prefetched copy of the instruction from executing to completion. The PrefetchFlush and memory barrier instructions (`DMB` or `DSB` as appropriate) are used to force execution ordering where necessary. See *Ordering of cache maintenance operations in the memory order model* on page B2-21.

The encoding for the PrefetchFlush is described in *Register 7: cache management functions* on page B6-19.

B2.7 Memory coherency and access issues

System designers and programmers need to consider all aspects of a design for overall system correctness. This section outlines some of the problems and pitfalls faced, along with the necessary steps which should be taken to ensure predictable system behavior.

Note

For the definitions in this section, a return from an exception is defined to mean one of:

- Using a data-processing instruction with the S bit set, and the PC as the destination.
- Using the Load Multiple with Restore CPSR instruction. See *LDM (3)* on page A4-40 for details.
- Using an RFE instruction.

B2.7.1 Introduction to cache coherency

When a cache and/or a write buffer is used, the system can hold multiple versions of the value of a memory location. Possible physical locations for these values are main memory, write buffers and caches. If Harvard caches are used, either or both of the instruction cache and the data cache can contain a value for the memory location. In a multi-level cache, a cache line may only be present in some levels, having been overwritten or evicted elsewhere.

Not all of these physical locations necessarily contain the value written to the memory location most recently. The memory coherency problem is to ensure that when a memory location is read (either by a data read or an instruction fetch), the value actually obtained is always the value that was written to the location most recently.

In the ARM memory system architectures, some aspects of memory system coherency are required to be provided automatically by the system. Other aspects are dealt with by memory coherency rules, which are limitations on how programs must behave if memory coherency is to be maintained. The memory attribute distinguishing *shared* and *non-shared* memory, as defined in *ARMv6 memory attributes - introduction* on page B2-8 for ARMv6 is designed to provide information on coherency needs, allowing implementations to maintain overall correctness, for example, allowing an implementation to enforce a non-cacheable policy on a region of memory marked as shared cacheable where snooping is not provided. The behavior of a program that breaks a memory coherency rule is UNPREDICTABLE. Address mapping and caches require careful management to ensure memory coherency at all times. Cache and write buffer management typically requires a sequence containing one or more of the following:

- cleaning the data cache if it is a write-back cache
- invalidating the data cache
- invalidating the instruction cache
- draining the write buffer
- performing a prefetch flush on the instruction pipeline.
- flushing branch prediction logic (branch target buffers).

in this order?

Prior to ARMv6, the operations and sequences are IMPLEMENTATION DEFINED. In ARMv6, the memory order model, cache, TLB and memory barrier operations supported in the System Control Coprocessor (CP15) allow the operating system support to be standardized for level 1 memory.

Note

Implementors are strongly advised to work with ARM where control of additional cache levels is required, to minimize potential impacts of future compatibility.

B2.7.2 Ordering of cache maintenance operations in the memory order model

The following rules apply to cache maintenance operations with respect to the memory order model:

- All Cache and Branch Predictor Maintenance operations are executed in program order relative to each other. Where a cache or branch predictor maintenance operation appears in program order before a change to the page tables, the cache or branch predictor maintenance operation is guaranteed to take place before change to the page tables is visible.
- Where a change of the page tables appears in program order before a cache or branch predictor maintenance operation, the sequence outlined in *TLB maintenance operations and the memory order model* on page B2-22 must be executed before that change can be guaranteed to visible.
- DMB causes the effect of all cache maintenance operations appearing in program order prior to the DMB operation to be visible to all explicit load and store operations appearing in program order after the DMB. It also ensures that the effects of any cache maintenance operations appearing in program order before the DMB are globally observable before any cache maintenance or explicit memory operations appearing in program order after the DMB are observed. Completion of the DMB does not ensure the visibility of all data to other (relevant) observers. (e.g. page table walks).
- DSB causes the completion of all cache maintenance operations appearing in program order prior to the DSB operation, and ensures that all data written back is visible to all (relevant) observers.
- PrefetchFlush or a return from exception causes the effect of all Branch Predictor maintenance operations appearing in program order prior to the PrefetchFlush operation to be visible to all instructions after the PrefetchFlush operation or exception return.
- An exception causes the effect of all Branch Predictor maintenance operations appearing in program order prior to the point in the instruction stream where the exception is taken to be visible to all instructions executed after the exception entry (including the instruction fetch of those instructions).
- A Data (or unified) cache maintenance operation by MVA must be executed in program order relative to any explicit load or store on the same processor to an address covered by the MVA of the cache operation.
- The ordering of a Data (or unified) cache maintenance operation by MVA relative to any explicit load or store on the same processor where the address of the explicit load or store is not covered by the MVA of the cache operation is not restricted. Where the ordering is to be restricted, a DMB operation must be inserted to enforce ordering.

- The ordering of a Data (or unified) cache maintenance operation by Set/Way relative to any explicit load or store on the same processor is not restricted. Where the ordering is to be restricted, a DMB operation must be inserted to enforce ordering.
- The execution of a Data (or unified) cache maintenance operation by Set/Way is not necessarily visible to other observers within the system until a DSB operation has been executed.
- The execution of an Instruction cache maintenance operation is only guaranteed to be complete after the execution of a DSB barrier.
- The completion of an Instruction cache maintenance operation is only guaranteed to be visible to the instruction fetch after the execution of a PrefetchFlush operation or an exception or return from exception.

As a result of the last two points, the sequence of cache cleaning operations for a line of self-modifying code on a uniprocessor system is:

```
STR rx, [Instruction location]
Clean Data cache by MVA to point of unification [instruction location]
DSB           ; ensures visibility of the data cleaned from the D Cache
Invalidate Instruction cache by MVA [instruction location]
Invalidate BTB entry by MVA [instruction location]
DSB           ; ensures completion of the ICache invalidation
PrefetchFlush
```

B2.7.3 TLB maintenance operations and the memory order model

The following rules apply to the TLB maintenance operations with respect to the memory order model:

- The completion of a TLB maintenance operation is only guaranteed to be completed by the execution of a DSB instruction.
- PrefetchFlush, or a return from an exception, causes the effect of all completed TLB maintenance operations appearing in program order prior to the PrefetchFlush or return from exception to be visible to all subsequent instructions (including the instruction fetch for those instructions).
- An exception causes all completed TLB maintenance operations which appear in the instruction stream prior to the point that the exception was taken to be visible to all subsequent instructions (including the instruction fetch for those instructions).

crucial

- All TLB Maintenance operations are executed in program order relative to each other.
- The execution of a data (or unified) TLB maintenance operation is guaranteed by hardware not to affect any explicit memory transaction of any instructions which appear in program order prior to the TLB maintenance operation. As a result, no memory barrier is required.
- The execution of a data (or unified) TLB maintenance operation is only guaranteed to be visible to a subsequent explicit load or store after the execution of a DSB operation to ensure the completion of the TLB operation and a subsequent PrefetchFlush operation, the taking of an exception, or the return from an exception.

- The execution of an instruction (or unified) TLB maintenance operation is only guaranteed to be visible to the instruction fetch after the execution of a DSB operation to ensure the completion of the TLB operation and a subsequent PrefetchFlush operation, the taking of an exception, or the return from an exception.

The following rules apply when writing page table entries to ensure their visibility to subsequent transactions (including cache maintenance operations):

- The TLB page table walk is treated as a separate observer for the purposes of TLB maintenance:
  - — A write to the page tables (once cleaned from the cache if appropriate) is only guaranteed to be seen by a page table walk caused by an explicit load or store after the execution of a DSB operation. However, it is guaranteed that any writes to the page tables will not be seen by an explicit memory transaction occurring in program order before the write to the page tables.
  - — A clean of the page table must be performed between writing to the page tables and their visibility by a hardware page table walk if the page tables are held in WB cacheable memory.
  - — A write to the page tables (once cleaned from the cache if appropriate) is only guaranteed to be seen by a page table walk caused by an instruction fetch of an instruction following the write to the page tables after the execution of a DSB operation and a PrefetchFlush operation.

The typical code for writing a page table entry (covering changes to the instruction or data mappings) in a uniprocessor system is therefore:

```
STR rx, [Page table entry] ;
Clean line [Page table entry]
DSB            ; ensures visibility of the data cleaned from the D Cache
Invalidate TLB entry by MVA [page address]
Invalidate BTB
DSB            ; ensure completion of the Invalidate TLB
PrefetchFlush
```

B2.7.4 Synchronization primitives and the memory order model

The synchronization primitives, SWP/SWPB and LDREX/STREX, follow the memory ordering model of the memory types accessed by those instructions. For this reason:

- Portable code for claiming a spinlock is expected to include a DMB instruction between claiming the spinlock and making accesses that make use of the spinlock.

- Portable code for releasing a spinlock is expected to include a DMB instruction before writing to clear the spinlock.

B2.7.5 Branch predictor maintenance operations and the memory order model

The following rule applies to the Branch Predictor maintenance operations with respect to the memory order model:

when you invalidate BTB you better do a prefetchflush!

- Any invalidation of the branch predictor is only guaranteed to take effect after the execution of a PrefetchFlush operation, the taking of an exception, or a return from an exception.

The branch predictor maintenance operations must be used to invalidate entries in the branch predictor after one of the following events:

- enabling or disabling the MMU
- writing new data to instruction locations
- writing new mappings to the page tables
- changes to the TTBR0, TTBR1, or TTBCR
- changes to the FCSE ProcessID or ContextID.

Failure to invalidate entries might give UNPREDICTABLE results caused by the execution of old branches.

B2.7.6 Changes to CP15 registers and the memory order model

All changes to CP14 and CP15 registers which appear in program order after any explicit memory operations are guaranteed not to affect those preceding memory operations.

any change to cp14 and cp15 need a prefetchflush

All changes to CP14 and CP15 registers are only guaranteed to be visible to subsequent instructions after the execution of a PrefetchFlush operation, or the taking of an exception, or the return from an exception.

However, the following applies to coprocessor register accesses:

- When an MRC operation directly reads a register using the same register number which was used by an MCR operation to write it, it is guaranteed to observe the value written, without requiring a context-synchronization between the MCR and the MRC.
- When an MCR operation directly writes a register using the same register number which was used by a previous MCR operation to write it, the final result will be the value of the second MCR, without requiring a context-synchronization between the two MCR instructions.

Some CP15 registers might, on a case by case basis, require additional operations prior to the PrefetchFlush, exception or return from exception to guarantee their visibility. These cases are specifically identified with the definition of those registers.

Where a change to the CP15 registers which is not yet guaranteed to be visible has an effect on exception processing, the following rule applies:

- Any change of state held in CP15 registers involved in the triggering of an exception is not yet guaranteed to be visible while any change involved with the processing of the exception itself (once it is determined that the exception is being taken) is guaranteed to take effect.

Therefore, in the following example (where A=1, V=0 initially), the `LDR` may or may not take a data abort due to the unaligned transaction, but if an exception occurs, the vector used will be affected by the V bit:

```
    MCR p15, r0, c1, c0, 0   ; clears the A bit and sets the V bit
    LDR r2, [R3]             ; unaligned load.
```

Synchronization of changes of ASID and TTBR

A common usage model of TLB management requires that the ContextID and Translation Table Base Registers are changed together to allow the ContextID to be associated with different page tables. However, the IMPLEMENTATION DEFINED depth of prefetch and the use of branch prediction create problems in ensuring the synchronization of changes of the ContextID and Translation Table Register (for example, TLBs, branch target caches and/or other caching of ASID and translation information might become corrupt with invalid translations). This synchronization is necessary to avoid either:

- the *old* ASID from being associated with page table walks from the *new* page tables
- the *new* ASID from being associated with page table walks from the *old* page tables.

There are a number of possible solutions to this problem, as illustrated by the following example.

wow.
wow.
wow.

Example solution

In this approach, the ASID value of 0 is reserved by the operating system, and is not used except for the synchronization of the ASID and Translation Table Base Register. The following sequence is then followed (executed from memory marked as being Global):

```
Change ASID to 0
PrefetchFlush
Change Translation Table Base Register
PrefetchFlush
Change ASID to new value
```

This approach ensures that any non-global pages accessed (by prefetch) at a time when it is uncertain whether the old or new page tables are being accessed will be associated with the unused ASID value of 0, and so cannot result in corruption of execution.

Another manifestation of this same problem is that if a branch is encountered between the changing of an ASID and its synchronization, then the value in the branch predictor might be associated with the incorrect ASID. This manifestation is addressed by the ASID 0 approach, but might also be addressed by avoiding such branches.

B2.7.7 Changes to CPSR and the memory order model

implication is we do not have to do any sync operations?

All changes to the CPSR via CPS, SETEND, and MSR instructions (that operate on the CPSR without causing or returning from exceptions), that appear in program order after any instruction operations, are guaranteed not to affect those instructions.

All changes to the CPSR via CPS, SETEND, and MSR instructions (that operate on the CPSR without causing or returning from exceptions), are guaranteed to be visible to all instructions that appear in program order after those changes, in all aspects except the effect on instruction permission checking. If the effect on the CPSR is to change the privilege (or security) status of the execution, then this change is only visible for the purposes of instruction permission checking after the execution of a PrefetchFlush operation, or the taking of an exception, or the return from an exception.