Freescale Semiconductor
Application Note

Document Number: AN3441
Rev. 1, 12/2007

Coherency and Synchronization Requirements for PowerQUICC™ III

by *Power Architecture Applications Engineering*
Networking and Multimedia Group
Freescale Semiconductor, Inc.
Austin, TX

Contents

This application note describes aspects of memory synchronization and cache coherency requirements for Freescale's PowerQUICC™ III processor family. Coherency and synchronization must be considered, both for data and instructions, when initializing memory, moving memory contents from one location to another, or changing ownership of memory.

Issues can arise when:

- Moving flash contents to DDR
- Changing local access windows
- Performing memory tests
- Releasing locks on data

Coherency and synchronization must be addressed for such applications.

1 Introduction

The PowerQUICC III processor is built on the Power Architecture™ technology and conforms with the Power ISA™ in its architecture as an out-of-order execution machine supporting weak memory ordering. To fully understand the implications, it is helpful to first define out of order execution and weak memory ordering.

Memory is considered to be well-behaved when the full address range is populated (for example, there are no memory holes) and speculative instruction or data pre-fetches will cause no undesired side effects. For example, DRAM memory mapped from address 0x0000_0000 to 0x01FF_FFFF is considered well-behaved memory. The full region from 0x0000_0000 to 0x01FF_FFFF is populated with memory with no memory holes. Non-sequential accesses to DRAM are allowed and cause no undesired side effects.

Conversely, non-well-behaved memory may contain memory holes, and speculative data accesses to non-well-behaved memory may produce undesired side effects. Examples of this would be an ASIC or I/O device using indirect addressing, or a load from an I/O device (UART with FIFO) that auto-increments the address. In both cases, speculative data accesses must be explicitly prohibited through TLB attributes.

Similarly, a speculative instruction access to non-well-behaved memory may cause a Machine Check exception or other undesired side effects. If a memory hole is unintentionally defined at the end of memory, the core may prefetch to that space. If code space is followed by non-execute space (possibly an I/O device), but mistakenly marked as execute space, the core may prefetch to that space and cause undesired side effects.

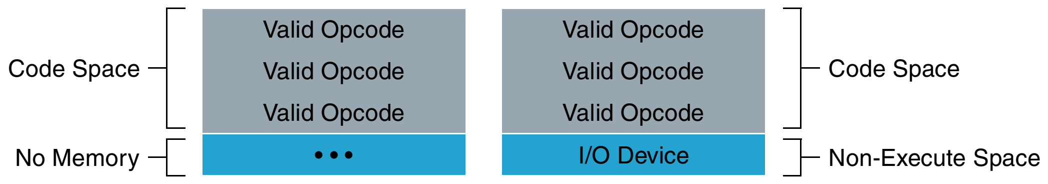

Figure 1. Non-Well-Behaved Memory

2 Cache Coherency

Cache coherency refers to managing all copies of data to ensure they are true reflections of data in memory. Unfortunately, disabling the caches does not always avoid cache coherency issues.

2.1 Data Cache Coherency

Data cache content may be coherent with physical memory, or not, depending on how the physical memory contents are changed. Data cache is coherent with memory when load, store, and cache operations are executed to pages marked as coherence required. If TLB pages are marked as cacheable, the core may keep internal copies of instructions and data, even if the caches are disabled. Cache management instructions (for example, **dcbf** and **icbi**) must be used, as specified by the architecture, to ensure the core does not use or keep internal copies of instructions and data.

Examples of methods that effectively change the contents of the physical memory the core sees are as follows:

- Executing store or **dcbz** instructions in the core
- DMA Transfers
- Changing local access windows

For example, executing store or **dcbz** instructions to pages that are marked as coherence required results in the data cache being coherent with memory; if DMA transfer is performed with coherency enabled, the cache contents will be coherent with memory. However, changing a local access window to point to a new region of memory can leave non-coherent data in the data cache.

2.2 Instruction Cache Coherency

Instruction cache coherency must be handled separately from data cache coherency. Whereas there is hardware support for data cache coherency, instruction cache coherency must be maintained in software. Even if stores are performed to pages marked as coherence required, **icbi** instructions are required to ensure the instruction cache is coherent.

When changing the contents of physical memory, the user must ensure the following:

- The instruction and data caches do not contain stale instructions or data that were cached before changing the memory contents.
- The new contents reach the primary storage device (for example, DDR) before attempting to branch to addresses in the new region.

The methods for ensuring these conditions depends on whether or not the changes are being made coherently.

Approaches for avoiding coherency issues include the following:

- Marking pages as caching-inhibited
- Preventing speculative accesses to memory before the contents are established (this may also require synchronization)
- Using cache management instructions to avoid coherency problems on cacheable pages
- Using implementation-dependent processor features to maintain coherency

CAUTION

Using implementation-dependent processor features may result in non-portable code.

The most difficult case is when memory contents are being changed non-coherently. The example below changes the local access window; the architecturally recommended procedure for doing so is as follows:

1. Mark the pages for the new area as guarded and non-executable, then execute an **msync** and **isync**. This will ensure that no future speculative accesses will occur to the new region before everything is ready.
2. Use **dcbf** to invalidate all addresses in the physical address space that is being remapped, then perform an **msync**. This will ensure the core does not contain any cached copies of the data from the old address space. Note that this is necessary even if the data cache is disabled, because the

core may keep copies of the data in other hardware structures. This step can be omitted if the user can guarantee that there has been no cacheable TLB mapping for this address space since power on reset.

3. Use **icbi** to invalidate all addresses in the physical address space that is being remapped, then perform an **msync** and **isync**. This will ensure the core does not contain cached copies of instructions from the old address space. Note that this is necessary even if the instruction cache is disabled because the core may keep copies of the instructions in other hardware structures. This step can be omitted if the user can guarantee that there has been no executable TLB mapping for this address space since power on reset.
4. Change the local access window mapping by reading the local access window contents back.
5. Mark the pages for the new area as needed for their intended use (for example, not guarded and/or executable).

The procedure for altering memory coherently only differs slightly. Step 2, above, can be omitted because the data cache coherency is handled by the hardware. However, instruction cache coherency must still be handled in software, therefore the following steps should be used for establishing a new memory region coherently:

1. Mark the pages for the new area as non-executable, then execute an **isync**. This will ensure that no future speculative instruction fetches will occur to the new region before everything is ready.
2. Perform store or **dcbz** instructions to establish instructions and data in the new region.
3. If the new region contains instructions, execute **dcbf** and **msync** to ensure the instructions are pushed out of the data cache. Use **icbi** to invalidate all addresses in the new region; follow this with an **msync** and **isync**. Note that this is necessary even if the instruction cache is disabled, because the core may keep copies of the instructions in other hardware structures. This step can be omitted if the user can guarantee that there has been no executable TLB mapping for this address space since power on reset.
4. Mark the pages for the new area as needed for their intended use (for example, executable).

If the user can ensure that the new physical address space has never before been used for cacheable accesses (instruction or data), the invalidation steps above may be skipped. This could be the case, for example, if during the boot process, the user is decompressing the contents of a FLASH memory into DDR. In this case, no TLB entry has previously been created for the new addresses, so there is no chance that speculative accesses have occurred to this region. The initial mappings used for decompression should be marked as non-executable.

A lower performance approach is to mark pages as caching-inhibited. By doing so, the user can avoid problems associated with cached copies. The core does not keep copies of instructions or data on pages that are marked as caching-inhibited. Additionally, simply executing an **msync** will guarantee that any modified data is pushed out to the core complex bus. No **dcbf** instructions are necessary.

A non-portable alternative is to clear the caches and core with implementation-dependent features. Code that uses these methods may not continue to work on future processors or, possibly, future revisions of the e500 processor. Setting L1CSR0[DCFI] to one will ensure that the contents of the L1 data cache are invalidated, and executing an **msync** will ensure that any other copies of data that exist in the core are purged. Setting L1CSR1[ICFI] to one will ensure that the contents of the L1 instruction cache are invalidated. This should be followed by modification of the TLB entry (presumably to enable execution privileges), which will also invalidate any speculative copies of instructions that may exist in the core.

3 Synchronization

EREF: A Programmer's Reference Manual for Freescale Book E Processors describes both context and execution synchronizing instructions, which are both necessary for proper memory synchronization.

Any instruction that alters the context in which data addresses or instruction addresses are interpreted, or in which instructions are executed, is called a context-altering instruction. A Context Synchronizing Instruction ensures that all changes due to context altering instructions have taken effect. The term context includes privilege level, address space and memory protection. Context Synchronizing Instructions include **isync**, **sc**, **rfi**, **rfci**, and **rfmci**.

An execution synchronizing instruction ensures that an operation is not initiated, or does not complete, until all instructions already in execution have completed to a point at which they have reported all of the exceptions that they cause. These instructions have nothing to do with the context before, or following, the execution synchronization instruction. Execution synchronizing instructions include **msync**, **mtmsr**, **wrtee**, and **wrteei**.

The *EREF: A Programmer's Reference Manual for Freescale Book E Processors* manual describes these synchronizing instructions in great detail. Several of the more common instructions are summarized in the sections below.

3.1 isync (Instruction Synchronize)

isync causes any prefetched instructions to be discarded by the core, and it ensures that any subsequent instructions are fetched and then executed in the context established in the instructions preceding isync. Note that **isync** does not affect data accesses and does not wait for all stores to be performed.

In Figure 2, **tlbwe** instruction is modifying a TLB entry. The **isync** in this example ensures the TLB is written to and the instructions following the **isync** are executed in the new context established by the TLB write.

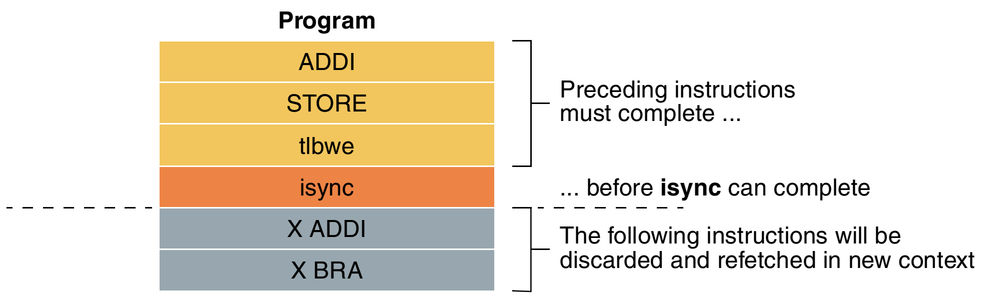
Program
ADDI
STORE
tlbwe
isync
X ADDI
X BRA
Preceding instructions must complete ...
... before isync can complete
The following instructions will be discarded and refetched in new context

Figure 2. isync Example

3.2 msync (Memory Synchronize)

msync performs two important functions:

1. **msync** provides an ordering function for data memory accesses across all storage classes.
2. Executing **msync** ensures that all instructions preceding the **msync** have completed before **msync** completes and that no subsequent instructions are initiated until after the **msync** completes.

Figure 3 is an example of loads and stores to two separate classes. Class 1 consists of a cache-inhibited region, while Class 2 is a cacheable region of memory. To enforce order between the cache-inhibited loads and stores and the cacheable load, it is necessary to insert an **msync** in between. Note that the store and load to cacheable space may still occur out of order (for example, the load will most likely bypass the store). In this example, the **msync** only is enforcing order between classes and guarantees that both the store and load to cache-inhibited space complete before the load to cacheable space initiates.

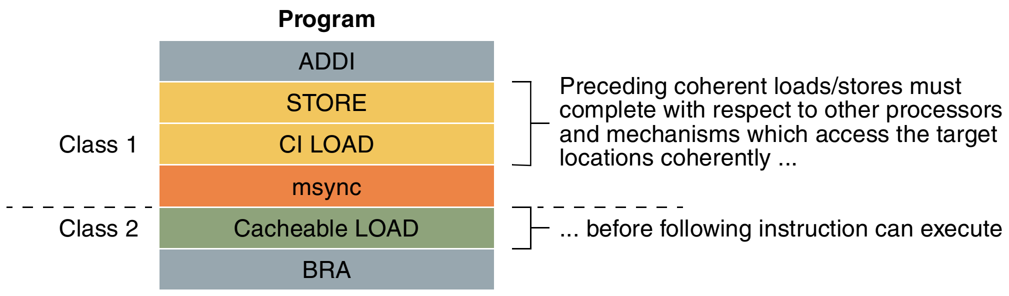
Program
ADDI
STORE
CI LOAD
msync
Cacheable LOAD
BRA
Class 1
Class 2
Preceding coherent loads/stores must complete with respect to other processors and mechanisms which access the target locations coherently ...
... before following instruction can execute

Figure 3. msync Example

3.3 mbar (Memory Barrier)

There are two flavors of the **mbar** instruction, depending on the MO field (bits 6–10).

1. MO = 0, **mbar** behaves similarly to msync
2. MO = 1, **mbar** is a weaker, faster memory barrier designed to order the following:
   — All stores
   — Caching-inhibited and guarded loads to caching-inhibited and guarded stores

With MO = 1, **mbar** does not wait for its address tenure to be performed successfully on the bus before allowing subsequent instructions to complete. This provides a faster way to order data accesses in a limited subset of storage classes.

Refer to Section 4, “mbar Erratum” for errata relating to **mbar** MO = 1 for PowerQUICC III.

Figure 4 shows an **mbar** MO = 0 instruction used as a memory barrier between stores and loads to the same class. The **mbar** enforces ordering between the initial store and load and the load instruction subsequent to the **mbar** instruction. Note that **mbar** MO = 1 will not order loads. For an example usage of **mbar** MO = 1, refer to Figure 7 in Section 5.3, "Example—Synchronization Between Classes."

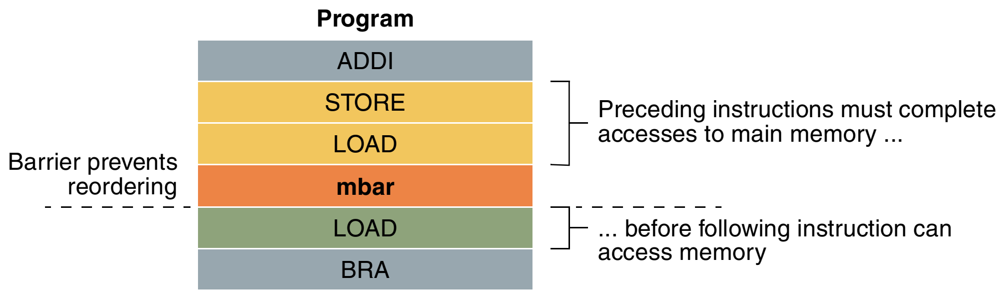
Program
ADDI
STORE
LOAD
mbar
LOAD
BRA
Barrier prevents reordering
Preceding instructions must complete accesses to main memory ...
... before following instruction can access memory

Figure 4. mbar Example

3.4 Core Complex Bus

On PowerQUICC III, synchronization instructions only enforce order on memory accesses to the point where they are performed on the core complex bus. To synchronize memory accesses beyond the core complex bus, the PowerQUICC III requires additional steps. In general, to ensure that store data has reached its final destination, a caching-inhibited store should be followed by a caching-inhibited load from the same address. If cacheable accesses are used, the store should be followed by the following:

1. **dcbf** or **dcbst** to the same address
2. **msync**
3. Load from the same address

NOTE

Order may not be enforced on some peripheral busses as per their specifications (for example, relaxed ordering in PCI-X and PCI-Express). Refer to Section 5.1.1, "Special Consideration for PCI-Express Ordering" for more information.

3.5 CCSR Space

It is important to note that when changing register values in CCSR space, it is sometimes necessary to prevent speculative memory accesses to certain address regions. The core does not initiate speculative memory accesses to any address for which there is no TLB mapping.

Additionally, the core will not initiate speculative instruction fetches from mapped pages that are marked non-executable nor speculative data reads from pages that are marked guarded.

4 mbar Erratum

Specific to e500v1 and e500v2 cores, the **mbar** instruction may fail when the MO field is equal to "1". In particular, **mbar** MO = 1 fails to properly order caching-inhibited guarded loads with respect to caching-inhibited guarded stores. For guaranteed store-load ordering to cache-inhibited guarded memory,

mbar with MO = 0 or **msync** is required. This failure is limited to caching-inhibited loads bypassing the **mbar** MO = 1 barrier. The **mbar** MO = 1 instruction correctly enforces the ordering of caching-inhibited stores to caching-inhibited stores, and the ordering of cacheable stores to cacheable stores.

Based on the impact of this erratum, Table 1 lists the revised memory barrier requirements for maintaining memory access ordering for all storage classes.

Table 1. FLS Power Architecture Synchronization for Memory Access Ordering

Cache/Memory Access Attributes	WIMGE	Speculative Execution	Store-Store Ordered	Load-Load Ordered	Store-Load Ordered	Load-Store Ordered
Caching-inhibited, guarded	01x1x	None	Yes	Yes	Requires **mbar** (MO = 0)	Yes
Caching-inhibited, non-guarded	01x0x	Permitted	Requires **mbar** (MO = 1)	Requires **msync**	Requires **msync**	Yes
Write-through, guarded	10x1x	None	Requires **mbar** (MO = 1)	Requires **msync**	Requires **msync**	Yes
Write-through, non-guarded	10x0x	Permitted	Requires **mbar** (MO = 1)	Requires **msync**	Requires **msync**	Yes
Write-back, coherency-required	001xx	Permitted	Requires **mbar** (MO = 1)	Requires **msync**	Requires **msync**	Yes
Write-back, coherency not required	000xx	Permitted	Requires **mbar** (MO = 1)	Requires **msync**	Requires **msync**	Yes

Note: Power Architecture technology states that combinations where WIMGE = 11xxx are not supported.

5 Examples

This section contains examples in the following subsections: Section 5.1, "Example—Ordering of Stores and Loads to I/O," Section 5.2, "Example—Buffer Descriptor Ownership," Section 5.3, "Example—Synchronization Between Classes," Section 5.4, "Example—Copying ROM to DDR," and Section 5.5, "Example—Changing a Local Access Window."

5.1 Example—Ordering of Stores and Loads to I/O

As mentioned in Section 1, "Introduction," ordering of stores and loads to non-well-behaved memory, specifically I/O, can be especially problematic. Consider an I/O device (an ASIC) where data needs to be written to one location, and a result is returned in a different location. Because this is I/O, it is assumed that the TLB pages pointing to the ASIC locations are marked as guarded and non-cacheable.

High level C code might look like the following:

```
WRITE_ASIC_REG(rxx_start_addr, rxxStartAddr);

rxxDataReport=READ_ASIC(rxx_data_repoty);
```

The PowerQUICC III is a weakly ordered machine, so loads and stores are executed out of order. In this case, the read bypasses the write, and incorrect data is read out of the ASIC, which causes undesirable results. The guarded TLB entry does not enforce order between loads and stores. Instead, guarded attribute enforces order between loads and loads, stores and stores, and loads and stores to the same address. To properly enforce order in the above example, **msync** or **mbar** MO = 0 must be used in between the store and the load.

```
WRITE_ASIC_REG(rxx_start_addr, rxxStartAddr);
_asm_volatile("mbar");
rxxDataReport=READ_ASIC(rxx_data_repoty);
```

The above C code produces assembly similar to that in Figure 5.

stw	r31,20(r1)
mr	r31,r3
mr	r3,r31
addi	r31,r31,60
cmpwi	r30,0
mbar	
lwz	r3,28(r1)

Figure 5. Example Code—Ordering Loads and Stores to I/O

Note the use of _asm_volatile in the C code. Without this syntax, the **gcc** compiler is allowed to optimize the **mbar** and put it out of order in the sequential code model. Other compilers may use alternate syntax for this function, so the **mbar** must be marked as non-optimizable as per the particular compiler in use.

Because order is being enforced only on a single class, **mbar** MO = 1 should be sufficient to ensure the store completes before the load initiates. However, the **mbar** failure discussed in Section 4, "mbar Erratum" is important to note, and because I/O is typically marked as cache-inhibited and guarded, it is necessary to use **mbar** MO = 0 as specified in Table 1.

5.1.1 Special Consideration for PCI-Express Ordering

Many legacy applications that use custom FPGAs and/or ASICs to extend the microprocessor's functionality are beginning to move to and become standardized on PCI-Express as the system interconnect of choice. It is important for system designers that are moving away from legacy PCI 2.x to understand that even though load/store ordering can be maintained to caching-inhibited and guarded I/O space on the internal processor bus, certain PCI-Express implementations and configurations may still allow those transactions to be reordered to increase performance or avoid deadlock.

The summary of the ordering rules for the PCI-Express controller implemented in the PowerQUICC III processor family is provided in Table 2, which is based on Table 2-23 for ordering rules of the *PCI-Express Base Specification, Rev 1.0a,* but is only specific to Freescale's PCI-Express controller implementation.

In Table 2, columns 1–5 represent a first issued transaction, and rows A–E represent a subsequently issued transaction. The table entry indicates the ordering relationship between the two transactions, as follows:

- Yes—the second transaction (row) must be allowed to pass the first transaction (column).
- No—the second transaction (row) must not be allowed to pass the first transaction (column).
- RO = 0—the relaxed ordering bit in the PCI-Express device control register is cleared.
- RO = 1—the relaxed ordering bit in the PCI-Express device control register is set.

For PowerQUICC III, only one case exists where reordering of memory read or write requests can take place in the PCI-Express controller—a memory write request may bypass a read request to avoid deadlock.

NOTE

It still may be possible for an external PCI-Express switch or ASIC connected to the microprocessor to reorder reads and/or write requests per the *PCI-Express Base Specification, Rev 1.0a*.

The ordering rules defined in Table 2 apply within a single Traffic Class (TC). There is no ordering requirement among transactions with different TC labels.

Table 2. PowerQUICC III PCI Express Transaction Ordering Rules Summary

Row Pass Column?		Posted Request	Non-Posted Request		Request Completion	
		Memory Write or Message Request (Column 1)	Read Request (Column 2)	I/O or Configuration Write Request (Column 3)	Read Completion (Column 4)	I/O or Configuration Write Completion (Column 5)
Posted Request	Memory Write or Message Request (Row A)	No	Yes	Yes	Yes	Yes
Non-Posted Request	Read Request (Row B)	No	No	No	a) No, RO=0 b) Yes, RO=1	a) No, RO=0 b) Yes, RO=1
	I/O or Configuration Write Request (Row C)	No	No	No	a) No, RO=0 b) Yes, RO=1	a) No, RO=0 b) Yes, RO=1
Completion	Read Completion (Row D)	a) No, RO=0 b) Yes, RO=1	Yes	Yes	No	No
	I/O or Configuration Write Completion (Row E)	a) No, RO=0 b) Yes, RO=1	Yes	Yes	No	No

In the example in Figure 5, the order of the write followed by the read is properly ordered by the PowerQUICC III's PCI-Express controller independent of the Relaxed Ordering (RO) bit's setting.

5.2 Example—Buffer Descriptor Ownership

Even though SDRAM is typically defined as “well-behaved,” the user must still be aware of cases when cores and peripherals share data structures and ownership is important. One of the common related issues is synchronization of a lock release.

A problem may occur in advanced micro-architectures supporting out-of-order memory accesses when a store, which releases a lock on a data structure, is performed ahead of stores to a different address(es) that provide context or messaging to the new owner of that data structure. Note that if two stores occur to the same address, they are guaranteed to be ordered by the architecture. Because the lock has been released, the new owner may be working out of the wrong context or with incorrect messaging. The resulting activity by the new owner may cause undesired side effects to the operation of the system.

One solution is to ensure the lock release store occurs after all other stores are complete. As mentioned previously, the Power Architecture technology defines memory synchronization instructions **mbar** (and **eieio**[1]) to properly order stores with respect to other stores under all MMU settings. Changing the attributes of the MMU pages in the e500 core to write-through does not guarantee the desired sequential store ordering to the buffer descriptors (BDs). Caching-inhibited stores must be performed in program order, so even though this may be a potential alternative to using the memory barriers, it may have a negative impact on packet performance.

An example of this problem is the use of shared buffer descriptors by the e500v2 core and eTSEC controllers of the MPC8548E. The eTSECs provide a means of receiving Ethernet frame data into buffer memory and transmitting buffered frames out on the line based on a pointer and certain bits in the header resident in the BD. These buffers and BDs are shared between the eTSECs and software running on the e500v2 core.

The driver software running on the e500v2 core initializes the eTSECs for proper operation and also supplies a means of handing ownership of the buffers and BDs between the eTSECs and the e500v2 core. The Ethernet controller uses a ring of BDs to receive and transmit Ethernet Frames. The beginning is indicated by a register pointing to the physical address of the start of the ring. The end is determined by a “wrap” bit being set in the last descriptor of the ring.

When a packet is received, the empty bit in the receive BD is cleared and the RXF bit in the IEVENT register is set, triggering an interrupt when the corresponding bit in the IMASK register is also set. This interrupt notifies the driver software running on the core that it may take ownership of the descriptor and associated buffered data. Once the buffer is processed and the header and pointer information updated in the BD by the core, it can hand the ownership of the BD and buffer back to the eTSEC for continued Rx of Ethernet frames by clearing the empty bit (the lock release).

When the kernel requests that a packet be transmitted, the driver starts where it last left off, and points the descriptor at the buffer that is ready to be sent. The driver informs the eTSEC that there are packets ready to be transmitted by setting the ready bit (the lock release) in the descriptor header. Once the eTSEC finishes transmitting the packet, the ready bit is cleared by the eTSEC, TXF bit set in the IEVENT register, and an interrupt may be triggered, which allows the driver to clean up the buffer and prepares it for the next transmission of Frame data.

1. **mbar** MO = 1 is functionally the same as **eieio** in the classic PowerPC ISA™, however **mbar** MO = 0 (which is functionally the same as **msync** and **sync)** has the same opcode as **eieio** in the classic PowerPC ISA.

For this example, the buffer descriptors are allocated in system memory (DDR2 SDRAM) where each BD contains the pointer to a packet buffer in system memory. The eTSEC BD is 8 bytes and has status, and length in first 32 bits and the buffer pointer in next 32 bits.

The proper sequence is to first perform a write to the packet pointer and then a second write to the length and status bits to give up the ownership of this BD to eTSEC. To ensure these stores are performed in order and the eTSEC is operating in the correct context, an **mbar** or memory barrier is required, as outlined in Figure 6.

In the example provided in Figure 6, **mbar** MO = 1 is used based on the EREF recommendation and the expectation of higher packet performance. Note that the opcode for **mbar** M = 0 (not **mbar** MO = 1) is identical to the opcode for the **eieio** instruction on classic PowerPC™ cores; **mbar** MO = 0 is guaranteed to execute on all past cores developed by Freescale. If cross core compatibility is desired, it is recommended that the user replaces the memory barrier with **mbar** MO = 0 in the sequence above. If performance is desired and only the e500 core or future Book E cores are being used, it is recommended that the user employs **mbar** MO = 1 as the memory barrier.

In the Linux 2.6 Gianfar.c Ethernet driver, there are only two memory barriers required to maintain proper operation of the lock release:

- For receive, the **mbar** (or **eieio**) is inserted after the RxBD’s buffer pointer is set and before the RxBD's Empty (E) bit is set, which can be seen in the Gianfar.c function gfar_new_skb( ).
- For transmit, the **mbar** (or **eieio**) is inserted after the TxBD's buffer pointer is set and before the TxBD's Ready (R) bit is set, which can be seen in the Gianfar.c function gfar_private( ).

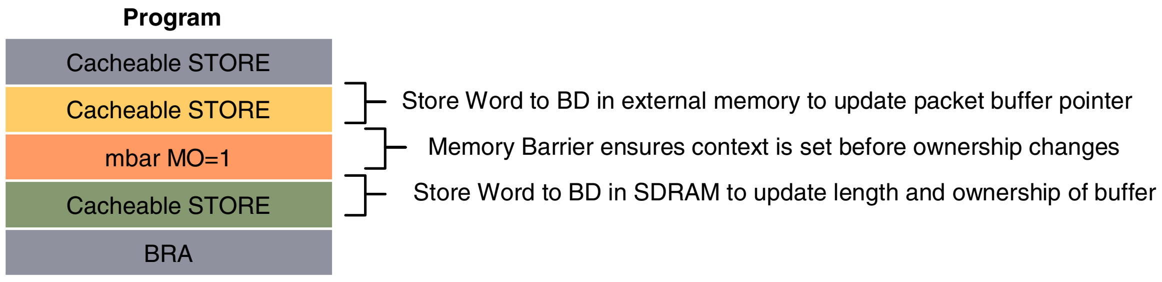
Program
Cacheable STORE
Cacheable STORE
mbar MO=1
Cacheable STORE
BRA
Store Word to BD in external memory to update packet buffer pointer
Memory Barrier ensures context is set before ownership changes
Store Word to BD in SDRAM to update length and ownership of buffer

Figure 6. Lock Release on Packet Buffer

5.3 Example—Synchronization Between Classes

As mentioned in Example 5.1, **mbar** with MO = 1 orders the following:

1. Stores with respect to other stores
2. Cache-inhibited and guarded loads with respect to caching-inhibited and guarded stores

mbar with MO = 1 does not impose order between cacheable and cache-inhibited space.

Refer to Section 4, “mbar Erratum” for errata relating to **mbar** MO = 1.

Figure 7 is an example of a sequential program with stores and loads to two different classes. In this example, Set 1 consists of cache-inhibited, non-guarded space, while Set 2 is cacheable space. With the **mbar** inserted, loads and stores are ordered within their respective sets. In this case, the first load/store to cache-inhibited, non-guarded space (LOAD/STORE) is ordered with respect to the second load/store to cache-inhibited, non-guarded space. Similarly, the stores to cacheable space (STORE) is ordered with respect to one another. However, there is no ordering enforced between the two classes, meaning the first store to cacheable space may occur after the second load/store to cache-inhibited, non-guarded space occurs.

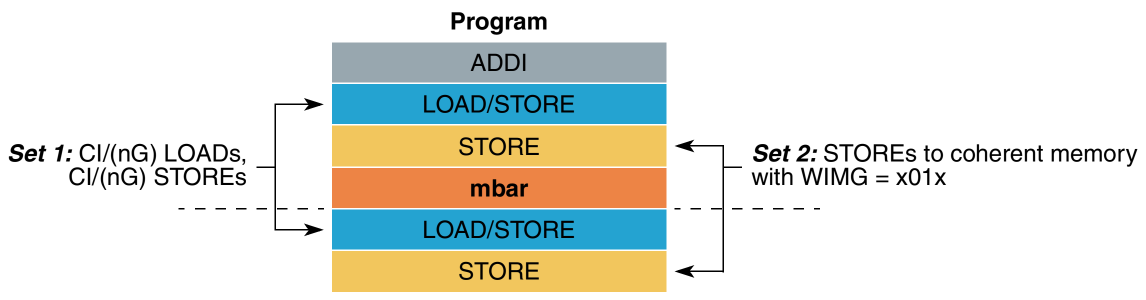
Program
ADDI
LOAD/STORE
STORE
mbar
LOAD/STORE
STORE
Set 1: CI/(nG) LOADs, CI/(nG) STOREs
Set 2: STOREs to coherent memory with WIMG = x01x

Figure 7. Synchronization Between Classes

To enforce order between the two classes, the user needs to insert an **msync** or **mbar** with MO = 0 in place of the **mbar** with MO = 1 in the example above.

5.4 Example—Copying ROM to DDR

A common operation at boot time is to decompress the contents of a FLASH or ROM into DDR, then begin executing from the DDR. Assuming that there has not been a TLB mapping for the DDR physical address space since power on reset, this process can safely be performed with the following steps:

1. Configure the DDR local access windows by writing to the registers in CCSR space. CCSR space should be cache-inhibited and guarded.
2. Ensure the configuration is complete by performing caching-inhibited loads from the modified locations in CCSR space.
3. Establish TLB entries for DDR space. These entries should be marked as non-executable and guarded to prevent speculative instruction and data reads; the entries can be marked as cacheable for performance.
4. Inflate the code into DDR. This can be done with the caches enabled. Either **dcbst** or **dcbf** instructions should be executed to ensure the inflated code is pushed out of the core.
5. Execute an **msync** to ensure all of the **dcbst** or **dcbf** instructions have been performed.
6. Change the TLB entry to allow execution privileges. Perform the required synchronization for changing TLB entries, which includes the required context-synchronizing instruction (for example, **isync**) to ensure that subsequent instruction fetches use the correct translation. See the *e500 Core Family Reference Manual* for details.
7. Branch to the inflated code in DDR.

Setting the TLB entries for DDR space only one time is a common programming error. If these entries are set to executable in step 2 above, then the core is allowed to speculatively fetch instructions to this space.

Depending on when these instructions are speculatively fetched, for example, prior to valid instructions being copied to DDR, the core may have stale instructions consisting of invalid opcodes. From the core’s point of view, this is essentially self-modifying code. As outlined in the steps above, to prevent such a coherency issue the user must first set up the DDR TLBs to both guarded and non-executable to prevent speculative instruction and data accesses to the space. Only once the desired code has been copied over to DDR, the software may reconfigure the TLBs to executable and non-guarded, allowing speculative accesses to memory.

5.5 Example—Changing a Local Access Window

Care must be taken, and coherency issues must be considered, when modifying an existing local access window. This example demonstrates one safe way to alter local access windows. Assume that the core previously had an existing TLB entry for some addresses in the range that is to be mapped, and assume that the entry is marked cacheable and executable. Care must be taken to manage the coherency of the region, even if the caches are disabled.

The following steps are portable and safe:

1. Set up TLB entries for the physical address space to be remapped. Mark the entries as guarded and non-executable.
2. Use **dcbf** and **icbi** to invalidate the entire region. Follow this process with **msync** and **isync**, which ensures that stale copies of instructions and/or data are purged from the caches.
3. Perform a caching-inhibited store to CCSR space to modify the LAW.
4. Perform a caching-inhibited load from the modified CCSR addresses to ensure the LAW update is complete.
5. Begin accessing the newly mapped area.

6 Summary

Synchronization and coherency are two continually important issues in embedded systems programming. Coherency refers to the issue of managing all copies of data to ensure that all the copies are true reflections of data in memory. The e500 core on the PowerQUICC III manages data coherency in hardware, but instruction coherency must be managed in software.

Additionally, the Power ISA defines a Weakly Ordered Memory Access Model (or Relaxed Memory Coherency Model). The benefits of such an architecture include the following:

- Reduced effect of memory latency on instruction throughput
- No time penalty incurred even if the result is not needed later, because execution is typically performed by idle resources

The risk is that out of order operations may violate requirements for memory that is non-well-behaved. Note that it is the software programmer’s responsibility for access ordering.

7 Revision History

Table 3 provides a revision history for this application note.

Table 3. Document Revision History

Rev. Number	Date	Substantive Change(s)
1	12/2007	Added Section 4, “**mbar** Erratum.” Added Section 5.1.1, “Special Consideration for PCI-Express Ordering.” Added Section 5.2, “Example—Buffer Descriptor Ownership.” Modified Sections 3.3, 5.1, and 5.3 to account for the **mbar** erratum in section 4.
0	9/2007	Initial release.

Document Number: AN3441
Rev. 1
12/2007