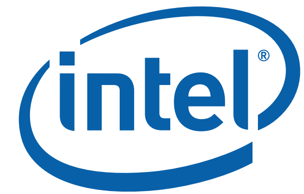
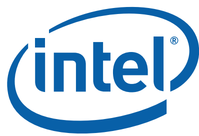


3rd Generation Intel XScale® Microarchitecture

Software Design Guide

July 2007

Order Number: 315058-001US



INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Intel Corporation may have patents or pending patent applications, trademarks, copyrights, or other intellectual property rights that relate to the presented subject matter. The furnishing of documents and other materials and information does not provide any license, express or implied, by estoppel or otherwise, to any such patents, trademarks, copyrights, or other intellectual property rights.

Designers must not rely on the absence or characteristics of any features or instructions marked “reserved” or “undefined.” Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See http://www.intel.com/products/processor_number for details.

The 3rd Generation Microarchitecture may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

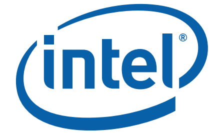
Copies of documents which have an order number and are referenced in this document, or other Intel literature may be obtained by calling 1-800-548-4725 or by visiting Intel's website at http://www.intel.com.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino logo, Core Inside, Dialogic, FlashFile, i960, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, IPLink, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skoool, Sound Mark, The Journey Inside, VTune, Xeon, and Xeon Inside are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

The ARM* and ARM Powered logo marks (the ARM marks) are trademarks of ARM, Ltd., and Intel uses these marks under license from ARM, Ltd.

*Other names and brands may be claimed as the property of others.

Copyright © 2007, Intel Corporation. All Rights Reserved.



Contents

1.0 Introduction 5

1.1 Quick Porting Guide 5

1.2 Use of Coprocessors 8

2.0 System Initialization and Reset 9

2.1 Configuration 9

2.2 Soft Reset 13

3.0 Memory 16

3.1 Virtual Memory (MMU) 16

3.2 Cache Management 25

3.3 Aborts 51

3.4 Memory Barriers 55

4.0 Performance Monitoring (PMU) 59

5.0 Interrupts 62

Figures

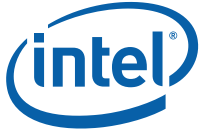
1 Global Observation vs. Program Execution 55

2 Data Memory Barrier Operation 56

3 Data Write Barrier Operation 57

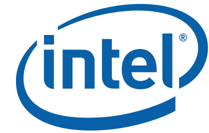
Tables

1 S Bit Scenarios 6



Revision History

Date	Revision	Description
July 2007	001	Initial release.



1.0 Introduction

This design guide contains routines for OS, monitor and device driver writers. It is generally not of interest for user-level code. Also, it is not intended to be an optimization how-to guide. What it does contain are routines and code sequences that are reference examples and templates necessary for system software to function correctly on 3rd generation Intel XScale® microarchitecture[1] (3rd generation microarchitecture or microarchitecture).

1.1 Quick Porting Guide

This section summarizes some significant changes in the 3rd generation microarchitecture. Subsequent sections give more details and code suggestions.

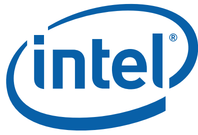
1.1.1 L2 Cache

The 3rd generation microarchitecture introduces a physically tagged, physically indexed, L2 cache (Section 3.2.6). Availability of the L2 cache varies by product. This typically has minimal impact on the OS with the following exceptions.

- System initialization code, at the appropriate time, enables the L2.
- Pages containing page table descriptors are marked as L2 cacheable. In addition, the table walk outer cache attribute, in the translation table base register, must be set to 0b11 (Outer Write Back) to allow the 3rd generation microarchitecture to hit the cached page descriptors.

Page table descriptor L2 cacheability and the table walk outer cache attribute must be enabled together, or disabled together, to ensure consistency between software and the processor microarchitecture.

1. ARM* architecture compliant.



1.1.2 Shared Bit

The shared bit is a new page attribute enabling cache coherency for memory shared by multiple agents. The shared bit marks pages that are shared between the 3rd generation microarchitecture and external devices. Not all products support I/O coherency (refer to the appropriate ASSP specification for more details.) Allowing the processor to manage I/O coherency achieves higher performance than using uncacheable memory regions.

Shared memory does not imply any specific memory ordering. For regions of memory that require more explicit access and timing controls see Memory Barriers below.

Do not use the S bit in Operating Systems without consideration. Performance is lower in some circumstances when the S bit is used and specific products that do not enable hardware coherency prohibit its use. See Table 1 for notes on use of the S bit.

Table 1. S Bit Scenarios

S bit in page table entry	Product supports hardware coherency	L2 cache present	Effect
0	X	X	Normal operation. Coherence enforced by software.
1	X	False	Performance suffers because the machine degrades memory accesses to “L1 uncacheable”.
1	False	X	Unpredictable — do not use S bit when hardware does not support coherency.
1	True	True	Performance is slightly lower because of coherency overhead. Coherency is supported by hardware.

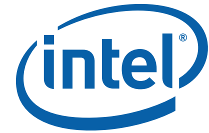
1.1.3 Memory Barriers

The 3rd generation microarchitecture optimizes memory accesses; this optimization causes external viewers of the microarchitecture to see memory operations in a different order than sequential execution implies. This reordering aids performance and does not alter the correctness of single-processor application programs. However, system software occasionally needs to control memory access order. Three new instructions were added to handle these situations. These instructions are:

- data memory barrier (DMB)
- data write barrier (DWB)
- pipeline flush (PF)

Refer to Section 3.2.8 for an example use of these instructions.

There are other methods to control memory ordering in addition to the three explicit instructions. This includes specific page table attributes and certain instructions which have a side-effect of synchronizing memory operations.



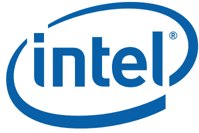
1.1.4 Miscellaneous Changes

The 3rd generation microarchitecture introduces new cache management functions useful for system-level software. The L1 data cache is now cleaned by set and way. Also, a cache line is now cleaned and invalidated in one instruction (by either MVA or set and way, refer to Section 3.2.4).

The 3rd generation microarchitecture requires new L1 cache cleaning routines using the new instructions mentioned above. Cache cleaning routines from previous microarchitectures produces unreliable results. See Example 27 for a routine to clean the L1 D cache.

The 3rd generation microarchitecture no longer has a mini data cache. Instead, system software uses Low Locality of Reference (LLR) memory attributes. LLR has the same page table entry encoding as formerly used for the mini data cache. However, LLR is not always a direct functional replacement for the mini data cache. Evaluate each situation to ensure the correct behavior is achieved.

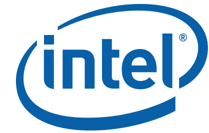
A specific routine to lock data into the L1 data cache has been defined in this document (Example 22). When data is locked in the L1, be sure to use this routine; variations from it does not operate correctly.



1.2 Use of Coprocessors

Throughout this guide, ARM coprocessor access instructions are used to perform many operations. Many of these instructions, especially for control of the caches, do not use the value in the specified ARM register operand — a simple write to the co-processor causes the operation to occur.

According to the *ARM Architecture Reference Manual*, the contents of the ARM registers needs to be zero when the value is not used by the function. However, in most of the code sequences the specified ARM register contains some non-zero value, this value is ignored and the contents of the ARM register are not modified. This behavior is valid and is used for the 3rd generation microarchitecture, and the code sequences are able to avoid zeroing and using a scratch ARM register by depending on this behavior.



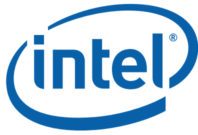
2.0 System Initialization and Reset

Certain tasks must be performed in order to initialize the 3rd generation microarchitecture after a system reset, and to perform a soft-reboot during a reset sequence. This section describes some routines that are utilized as part of these tasks.

2.1 Configuration

This section describes operations relating to access of configuration and co-processor registers in the 3rd generation microarchitecture. Co-processor registers are accessed to control certain functions of the microarchitecture (such as cache enabling and disabling) as well as to perform specialized operations using other implementation specific co-processors.

Throughout this Guide, individual bits are turned on and off with a read-modify-write sequence on the CP15 ARM control register. Although in most examples only one bit is changed at a time (for example, turning BTB on/off, MMU on/off), it is possible to simultaneously enable multiple features in the ARM control register, by setting multiple bits at once. It is useful to do this in order to have a shorter configuration code sequence when multiple features of the 3rd generation microarchitecture need to be turned on or off at once.



2.1.1 Co-processor Access

Product specific co-processors are accessed using the co-processor access instructions in the 3rd generation microarchitecture. An example of this is accumulator functionality of co-processor 0. Within a running system, multiple applications want to use a single co-processor with an exclusive access view to it. In this case, it is desirable to have the co-processor state and its registers appear to be consistent and exclusively used by each process. When a process switches out with another it simply saves all the registers out to memory, so another process restores its old state and continue running.

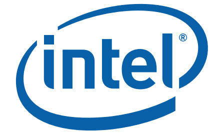
The 3rd generation microarchitecture provides a mechanism for co-processor context switching to occur only when needed, by providing a co-processor access register, which allows system software to receive an undefined exception whenever an application accesses a specific co-processor. This is accomplished by clearing a bit in the Co-processor Access Register (CPAR). Each bit in the CPAR corresponds to a co-processor. When the bit is cleared, then access to the corresponding co-processor causes an undefined exception, which system software handles and uses to determine what action to take to ensure each process sees the co-processor in the state it last left it.

Example 1. Setting Access Control Bits in the Co-processor Access Register

```
@ This macro enables or disables access to Co-processor 0.
@ The register Rd contains a nonzero value when access is to be enabled,
@ and zero when it is to be disabled.
.macro SET_CP0_ACCESS, Rd
    cmp \Rd, #0                  @ check whether to enable or disable
    mrc p15, 0, \Rd, c15, c1, 0 @ read original CPAR value
    biceq \Rd, \Rd, #0x1        @ disable when zero
    orrne \Rd, \Rd, #0x1        @ enable when nonzero
    mcr p15, 0, \Rd, c15, c1, 0 @ move back to CPAR
    mcr p15, 0, \Rd, c7, c5, 4  @ Prefetch Flush instruction so next instruction
                                @ re-fetched and see the effect
.endm
```

Example 1 sets Co-processor Access Register (CPAR) bit 0 based on the value of Rd. When Rd is zero, bit CPAR[0] is cleared, disabling access to co-processor 0 for any software that runs afterward. When software attempts to either read or write a co-processor 0 register, the 3rd generation microarchitecture produces an undefined exception. Since this is a precise exception, the exact state of the microarchitecture and which instruction caused the exception is determined. With this information system software is determined whether to allow the software to access the co-processor (and perhaps save and restore some state information) or to return an error to the process which caused the exception. Note, that at power-on-reset, all bits in the CPAR register are zero, so access to all co-processors is initially disabled.

An example of how system software utilizes the CPAR, is to allow multiple applications to have an exclusive view of a shared co-processor. Initially, access to the co-processor is disabled at boot time. When a process attempts to access the co-processor, an undefined exception occurs. The exception is handled and the co-processor being accessed is determined. At this point the exception handler decides when it needs to save the state of the co-processor, and also when it needs to load a saved state into the co-processor. When the co-processor is ready to be used by the current process, the handler code sets the corresponding bit in CPAR and exits the exception handler. The process that was accessing the co-processor continues as though the co-processor was always available and its state was consistent with when it was last context-switched.



2.1.2 Synchronization (CPWAIT)

At times, it is necessary to be able to ensure a point, after which a write to the ARM control register has taken effect. For example, when enabling memory address translation (turning on the MMU), it is vital to know when the MMU is actually ensured to be in operation. When the configuration routine returned and other software resumed before the MMU was enabled, memory accesses initially is affected in an unpredictable way.

The CPWAIT macro is a sequence of instructions which utilizes a read-after-write dependency, followed by a pipeline flush, to ensure changes to the ARM control register have taken effect before the next instruction executes.

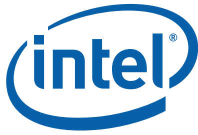
Since any configuration operation on the ARM control register occurs with a write to a register in CP15, this macro works by issuing a read to a register in CP15. This read does not occur until after the previous writes have completed.

Additionally, reading this register causes a stall until the effect of the original configuration operation takes effect. In order to ensure execution stops before continuing any further, the next instruction after the read creates a load-use-dependency on the register used. This causes a stall until the previous read completes, along with any prior write operations.

The final instruction causes the pipeline to be flushed, which ensures that the next instruction executed must be re-fetched either out of cache when it exists there, or from main memory.

Example 2. CPWAIT Macro

```
@ Use the following macro when software needs to be
@ assured that a CP15 update has taken effect.
@ It is only used while in a privileged mode, because it
@ accesses CP15. Rs is a scratch register.
.macro CPWAIT, Rs
    mrc p15, 0, \Rs, c2, c0, 0 @ arbitrary read of CP15
    mov \Rs, \Rs               @ wait for it
    sub pc, pc, #4             @ flush the pipeline
    @ At this point, any previous CP15 writes are
    @ ensured to have taken effect.
.endm
```



It is important to understand when it is necessary to utilize the CPWAIT macro. Because the macro ensures that all previous writes to CP15 have completed, it is not necessary to issue a CPWAIT after each write in a sequence of CP15 register writes, unless it was desired to have some part of the sequence take effect before the entire sequence completed.

Additionally it is not necessary to issue a CPWAIT immediately after a CP15 write, it is delayed and issued at a later time, and still ensures that the operation takes effect by the time the macro had completed.

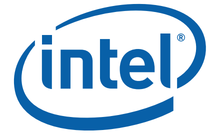
Also, since CPWAIT does not explicitly change anything with respect to the configuration of the 3rd generation microarchitecture, it is important to remember that the effect of the original writes to CP15 are observed during or even before the issue of the CPWAIT macro. That is to say, although software cannot know exactly when the effect of a configuration operation is observed, it is ensured to be no later than the end of the CPWAIT sequence.

It is also useful to note that the entire code sequence above is not necessarily required for the CPWAIT operation to occur.

The Example 2 on page 11 works as a self-contained macro that is invoked from anywhere in the code. The macro creates its own dependency stall to ensure the 3rd generation microarchitecture stops executing until it has completed, but it does so by executing a dummy instruction after the read, to create the stall. When there was an instruction that generates a dependency on ‘r0’ that executes, after the read of the co-processor register, which is then used in its place.

For example, when desiring to invalidate the TLB immediately after disabling the MMU, a write to co-processor 15, using ‘r0’ (the contents are ignored) to invalidate the TLB just after the read, still ensures the disabling of the MMU takes effect, but result in a more compact code. An example of this ‘short CPWAIT’ is shown in “Soft Reset Sequence (Code Mapped 1:1)” on page 14.

Note: This method does not also flush the pipeline. It is only suitable in a situation where the operation does not affect the code currently being executed (and any instructions already in the pipeline).



2.2 Soft Reset

Performing a soft reset in the 3rd generation microarchitecture, essentially involves returning the system to a power-on like-state and then branching to the reset vector to allow the system to boot again. It is important that certain components be returned to the power-on state in a specific order, or a situation occurs where the reset sequence causes a lockup, and the system is unable to reset or continue operation.

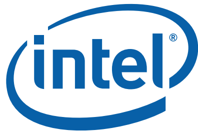
For example, when the L1 caches are disabled, but still contain valid lines, these produce cache hits when those memory locations are accessed, thaty cause errors when the system is restarting.

A similar problem occurs when the data or instruction TLB, the L2 cache, or the BTB have stale data, when a soft reset occurs.

The soft-reset sequence handles the case where the soft-reset code executes in a memory region, with the virtual and physical addresses mapped 1:1. That is to say, that the address the code resides at from the view of the software, is the same with the MMU enabled and disabled. In this case, the code is marked ‘outer’ cacheable in the page table entries describing the code, and there is no requirement to change this during the soft boot sequence.

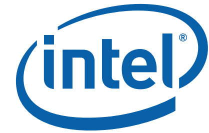
In either case, it is essential that interrupts are disabled before this code is executed. This is because any execution of an interrupt handler causes the 3rd generation microarchitecture to branch from this code at an unknown point, possibly causing lines to be fetched in to the cache, or causing an abort loop when the handler code is no longer accessible (after the MMU is off).

Example 3 on page 14 illustrates the sequence of operations required to facilitate a soft-reset.



Example 3. Soft Reset Sequence (Code Mapped 1:1)

```
@ This code must have a 1:1 virtual to physical memory mapping
@ This code is or is not marked 'outer cacheable' in its page table entry
@ This first instruction has the effect of disabling interrupts.
    msr cpsr, #0xD3                 @ Reset cpsr, F,I bits, SVC mode
    GET_L2_PRESENT r0                @ macro call sets Z flag when L2 is not present
    mov r0, #0

@ First unlock all caches and the TLBs
    mcr p15, 0, r0, c9, c5, 1       @ unlock all lines in the L1 I-cache
    mcr p15, 0, r0, c9, c6, 1       @ unlock all lines in the L1 D-cache
    mcrne p15, 1, r0, c9, c5, 1     @ unlock all L2 cache lines, when L2 present
    mcr p15, 0, r0, c10, c4, 1      @ unlock the I-tlb
    mcr p15, 0, r0, c10, c8, 1      @ unlock the D-tlb

@ Disable the L1 data cache
    mrc p15, 0, r0, c1, c0, 0       @ Get Control Register
    bic r0, r0, #0x0004             @ Clear C Bit (bit 2)
    mcr p15, 0, r0, c1, c0, 0       @ Update control register

@ When desired, branch to the L1 D-cache clean function here
@   bl l1_dcache_clean
    bic r0, r0, #0x1800             @ Clear I Bit (bit 12), Z Bit (bit 11)
    bic r0, r0, #0x0001             @ Clear M Bit (bit 0)
    mcr p15, 0, r0, c1, c0, 0       @ Disable MMU, L1 I cache, and BTB
    mrc p15, 0, r0, c1, c0, 0       @ short CPWAIT
    mcr p15, 0, r0, c8, c7, 0       @ Invalidate I, D-TLB (stall on CPWAIT read)
    beq CLEANDONE                   @ Jump over L2 clean when L2 not present

@ This code cleans and invalidates all lines in the L2 cache
    mcr p15, 0, r0, c7, c10, 5      @ DMB Operation to impose memory fence.
    GET_L2_SIZE r0                  @ low four bits returned in r0 are the L2
                                    @ cache size, starting with 64 KB (0b0000)
                                    @ increasing by powers of 2 (Example 30)
    mov r1, #0xffffffe0
    mov r2, #19
    sub r2, r2, r0                  @ determine the number of sets
    mov r0, r1, lsl r2              @ clear out the extra bits when configuring
    mov r0, r0, lsr r2              @ the number of sets

CLEANLOOP:
    mcr p15, 1, r0, c7, c15, 2 @ clean and invalidate set/way in r0
    adds r0, r0, #0x20000000   @ Increment shifted way index
    bcc CLEANLOOP              @ Clean the next way when not done with this set
    subs r0, r0, #0x00000020   @ Decrement shifted set index
    bpl CLEANLOOP              @ Go to next set when not at last one
    mcr p15, 0, r0, c7, c10, 5 @ DMB Operation to impose memory fence.

CLEANDONE:
    mcr p15, 0, r0, c7, c6, 0       @ globally invalidate the L1 D-cache
    mcr p15, 0, r1, c7, c5, 0       @ Invalidate I-cache, BTB
@ Reset microarchitecture registers to power-on-reset state.
    mov r0, #0x0
    mcr p15, 0, r0, c1, c0, 1       @ Reset Aux Control Register
    mcr p15, 0, r0, c13, c0, 0      @ Process ID Register
    mcr p15, 0, r0, c15, c1, 0      @ Co-processor Access Register
    mcr p14, 0, r0, c0, c1, 0       @ PMU Control register
    mcr p14, 0, r0, c4, c1, 0       @ PMU Interrupt enable register
    mov r0, #0x1f
    mcr p14, 0, r0, c5, c1, 0       @ PMU Overflow Flag Register
@ When applicable reset CP14 c7 and c6 to reset values.
@ Now restore the ARM control register to its power-on-reset state.
@ Note the L2 cannot be disabled once it has been enabled.
    mrc p15, 0, r0, c1, c0, 0       @ Read Control Register
    and r0, r0, #0x04000000         @ Clear all bits but L2 Enable
    mcr p15, 0, r0, c1, c0, 0       @ write back value, reset to power on state
@ Branch to zero (or reset handler)
    mov pc, #0
```



As shown in Example 3 on page 14, in order to shut down the system, the components of the 3rd generation microarchitecture must be disabled in a particular order, to ensure that the reset code still executes during this time. Additionally, it is important that the system appear to be in a power-on-reset state when the final branch to the reset vector occurs.

In order to ensure the processor is in a power-on reset state, the CPSR register is reset, to have interrupts disabled (F and I bits) and to be in supervisor mode. Then the next step is to unlock all entries in the L1 cache, L2 cache, and the TLB. This ensures the subsequent clean/invalidate operations work as expected, and not leave any locked valid entries behind.

After that the L1 data cache is disabled, then its contents are cleaned when desired, so any dirty cached data is cleaned and written out. The L1 instruction cache and BTB are also disabled. Cleaning the cache is or is not desired, depending on the specific application of the system software. In a case where dirty written data exists that needs to be cleaned out to some non-volatile storage, it needs to be cleaned before the system soft boots. In other cases, the system has already cleaned out all important data and any valid dirty lines in cache need not be written out.

It is important that the cache is disabled first and then emptied. This ensures that no new entries are created in the cache during or after the time it is cleaned and invalidated. This procedure is followed to ensure the L1 instruction cache and L2 cache are also emptied.

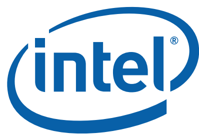
Now that the L1 Data and Instruction caches are disabled, the MMU is disabled. This effectively disables the L2 cache. After the MMU is off and no new TLB entries are fetched, the TLB is invalidated. Then, the L2 cache must be unlocked, cleaned when desired, and invalidated so that it is completely empty, since it is in a power on state.

Finally the L1 instruction cache, L1 data cache, and BTB are both invalidated so these are empty as well. The power-on-reset values of many co-processor registers are written out before the soft reboot occurs. This ensures that the processor behaves as though it had just powered on when the boot up code executes.

Note: The L2 cache cannot be disabled once it has been enabled, so it must be left enabled even during a soft reboot.

Depending on the product configuration, Co-processor 14 registers 6 and 7 need to be reset to restore the power and clock modes. At this point, branching to the reset vector produce a reboot with the 3rd generation microarchitecture in the same state as though it had just been power cycled.

It is not possible to perform a soft reset of the 3rd generation microarchitecture with code that is not mapped 1:1. When using code that is not mapped 1:1 then the 1:1 code sequence must be copied to a page that is mapped 1:1 where the code is then executed. Treat this code as though it were self-modifying code as described in Chapter 3.2.8



3.0 Memory

This section describes code sequences and examples which explain how system software utilizes memory related functions of the 3rd generation microarchitecture. Memory access in the microarchitecture is affected by:

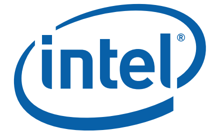
- memory translation
- L1 caching
- L2 caching
- memory barriers

3.1 Virtual Memory (MMU)

The MMU is a collection of hardware functions that implement virtual-to-physical memory address translation. The MMU contains:

- control registers
- the TLB
- translation table walk hardware

Access to the data and L2 caches is also controlled by enabling the MMU, because the page tables provide the cacheability information about the specific memory regions. Enabling the MMU is performed by code that is mapped 1:1 (virtual addresses match the translated physical addresses). Example 6 on page 19 shows a code fragment that enables the MMU.



Example 4. Creating a First Level Page Table

```
@ This code calls a function that sets up a first-level page table
@ of section descriptors. After it returns, it calls a function to create
@ a coarse 2nd level page table, and modifies the attributes of
@ the pages containing both page tables to have the correct cache attributes
@ for the microarchitecture.

@ The address where the table(s) need to reside is in r3.

    mov r0, r3                      @ Address where table needs to reside
    bl create_1stlevel_table        @ create table of sections, @ addr r0 1:1 mapped
    add r4, r3, #16384              @ set r4 to the address just after the 1st
                                    @ level table
    mov r0, r4
    mov r1, r3, lsr #20
    mov r1, r1, lsl #20             @ put 1Meg aligned address of 1st table in r1
    bl create_coarse_table          @ create table of 2nd level coarse descriptors
                                    @ table @ addr r0, mapping for 1meg @ addr r1

@ Now set the correct 1st level descriptor to point to the 2nd level table.
    mov r2, r3, lsr #20             @ put index of 1st level descriptor that
                                    @ contains table into r2
    ldr r2, [r3, r2]                @ load section descriptor
    mov r2, r2, lsl #22
    mov r2, r2, lsr #22             @ convert section descriptor to coarse pg tbl
    orr r2, r2, r4                  @ set coarse page table base address
    bic r2, r2, #0xF                @ Clear section type, c,b bits
    orr r2, r2, #0x1                @ Set coarse pg table type
    str r2, [r3, r2]                @ Write coarse page table pointer back

    mov r2, r3, lsl #12
    mov r2, r2, lsr #24             @ Get index into 2nd level table
    add r2, r2, r4                  @ Get addr of coarse descriptors to modify
    mov r0, #5                      @ number of extended small pages to modify
ATTR_LOOP:
    ldr r1, [r2]                    @ Load descriptor
    bic r1, r1, #0xFF
    orr r1, r1, #0x3                @ Set Extended small page type, CB=00
    orr r1, r1, #0x170              @ Set Tex=101, AP=11
    subs r0, r0, #1
    str r1, [r2], #4                @ write back descriptor
    bne ATTR_LOOP

    mov r1, r3                      @ address to clean, invalidate from DCache
    mov r0, #544                    @ Clean 544 lines = 16k + 1k of tables
    bl clean_inv_unlock             @ Call function
    mcr p15, 0, r0, c7, c10, 4      @ Data Write Barrier

@ At this point, the first and second level tables written out have
@ the correct attributes in place for being updated while used, and
@ these are ready to be used by the MMU.
```

The code in Example 4 calls a function that creates a page table of section descriptors, which configure all virtual memory to have a 1-to-1 mapping with physical memory. Depending on particular system configuration, regions of address space are or are not marked cacheable in the L1 and L2 caches.

After creating the first level table, another function is called to create a second level table, located just after the first level table. The second level table contains pages which define attributes for 4 K of address space each. The second level descriptors, which map to the address range the table exists in, are modified as extended small page descriptors with L2 cacheability and no L1 cacheability. This ensures any page table writes, once it is being used by the MMU, do not need a cache clean/invalidate sequence, since these go directly to the L2. Any pages containing page tables must be marked L2 cacheable on the processor.

After the code has executed and the page table has been written, the L1 data cache lines that contain the modified data are cleaned and invalidated, and a data write barrier operation is executed. After this operation, it is possible to set the page table base register to the address of this table.

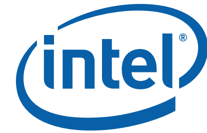
Example 5. Creating a Supersection

```
@ Create a supersection in the first level page table.
@ It assumes the first level page table is already created.
@ r1 = address of first level page table
@ r2 = 32-bit virtual address of start of supersection
@ r3 = bits 35:24 of supersection physical address in bits 11:0

    mov r4, r2, lsr #20          @ Get the index into page table
    add r4, r1, r4, lsl #2       @ multiply index by 4 (word offset into table)
                                 @ and add to page table base address
    mov r5, r3, lsl #24          @ put bits 7:0 of r3 into bits 31:24 of r5
    mov r6, r3
    bic r6, r6, #0xFF
    add r5, r5, r6, lsl #12      @ put bits 11:8 of r3 into bits 23:20 of r5
    orr r5, r5, #0xe             @ Set C,B, SectionType bits
    orr r5, r5, #0x00040000      @ Set bit 18, supersection bit
@ At this point, r5 contains the supersection descriptor, which is repeated
@ for 16 entries (so it occupies 16 megs)
    mov r0, #16                  @ put count into r0
1:
    subs r0, r0, #1              @ decrement count
    str r5, [r4], #4             @ store descriptor and increment
    bne 1b                       @ loop through all 16 entries

@ At this point VA r2 maps to 16-megabyte address space specified in bits
@ 11:0 of r3.
@ It is necessary to invalidate the instruction and data TLBs
@ at this point.
```

The code sequence in Example 5 creates a supersection, which maps the specified virtual address to the specified 36-bit physical address. It is assumed that the first level page table has already been created, and so only the supersection descriptors need to be written to.

The format of a supersection is similar to a section descriptor except that the upper 4 bits of the 36-bit physical address are stored in bits 23:20 of the descriptor, and bit 18 of the descriptor is set to designate that this is a supersection descriptor. The same supersection descriptor must be repeated for 16 consecutive page table entries, otherwise unpredictable behavior occurs.

After any change to the page table, it is necessary to invalidate the instruction and data TLBs, to clear out any stale entries. Also note, that page table updates must be managed carefully, when table walks are configured as L2 cacheable in the Translation Table Base register. Old page table entries were cached in the L2 cache and stores to the page table (when the MMU is disabled or the MMU enabled and the page table mapped to a L2 non-cacheable memory region) bypass the L2 cache.



To avoid issues when table walks are L2 cacheable, it is recommended that the memory region containing the page table be marked as L2 cacheable in the page table and the MMU be enabled to allow the stores to be L2 cacheable.

Once the supersection is created, any access by the 3rd generation microarchitecture to the specified virtual address, causes the memory operation to access the specified 36-bit physical address. Access to 36-bit addresses only occurs when the MMU is enabled, because the actual address specified in code is still only 32-bits.

Example 6 shows a method of enabling the MMU on the 3rd generation microarchitecture. In order for the MMU to operate, a valid page table must exist in memory and its base address must be set in the Translation Table Base register (TTBASE).

Additionally the domains which are represented in the page table must be set up with the appropriate access in the domain access control register (DACR). Code executing memory operations from this point forward has the addresses translated by the page table entries, and has accesses checked against the access permission bits in the page section or page descriptor and the corresponding domain access bits in the DACR.

It is important that a CPWAIT instruction be used after the enabling of the MMU and before any code attempts to perform a memory operation. Using a CPWAIT allows the system software to ensure that the effect of the MMU being enabled is seen by any code that executes after the CPWAIT.

Example 6. Enable the MMU

```
@ Enable the MMU. Before enabling, make sure the page table base (TTBASE) is set,
@ and that the domain access control register (DACR) is configured as desired.
    mrc p15, 0, r0, c1, c0, 0   @ Read the ARM control register
    orr r0, r0, #0x1
    mcr p15, 0, r0, c1, c0, 0   @ Enable the MMU
    CPWAIT r0
```

Example 7 disables the operation of the MMU by turning off the MMU enable bit in the ARM control register. Since the L1 data cache cannot operate without the MMU, it is important that it be cleaned and disabled before the MMU is disabled.

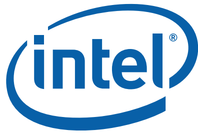
Also note, that when the MMU is off, the L2 cache is effectively disabled, since any memory accesses are no longer able to see the attribute bits that enable L2 cacheability.

After the L2 is disabled it is necessary to globally clean it to ensure any data written that is stored in the L2 cache is cleaned out.

Finally, it is important that this code is mapped 1:1 with physical memory, because during the execution of this code, the MMU turns off and when the virtual address location, the code was executing from is not the same as the physical address, unpredictable results occur when the instructions are fetched. The CPWAIT macro at the end of the sequence ensures that any code after the macro, sees the effects of the MMU being disabled.

Example 7. Disable the MMU

```
@ This code disables the MMu. It is important that the L1 data cache
@ be cleaned and disabled before the MMU is disabled.
    mrc p15, 0, r0, c1, c0, 0   @ Read the ARM control register
    bic r0, r0, #0x1
    mcr p15, 0, r0, c1, c0, 0   @ Disable the MMU
    CPWAIT r0
```



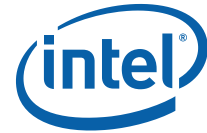
3.1.1 Translation Look Aside Buffers (TLB)

In order to increase the speed of memory address translations, the 3rd generation microarchitecture implements two fully associative 32-entry translation look aside buffers (TLBs).

- One is used to cache the lookup of instruction address translations
- The other is for data address translations

When an address translation occurs, the appropriate TLB (instruction or data) is checked prior to performing a ‘table walk’ to the page table residing in the backing levels of memory. When a cached page table entry is found in the TLB, the address translation, as well as permission and cacheability checks, occur based on the information cached in the TLB. Because the TLB is used to cache page table entries for use by the MMU, the TLB only operates when the MMU is enabled. Some operations, such as TLB entry locking, require that the MMU be enabled or unpredictable results occur.

When an entry is not found in the TLB, a ‘table-walk’ occurs to load the page table entry from the backing levels of memory. Page table entries are cached in the L2 cache, depending on configuration of certain control bits in the Translation Table Base Register. When table walks are configured to be cacheable in the L2, the L2 cache is checked next. When the table walk does not hit in the L2 cache, then a memory access occurs to the external memory.



Example 8 sets the base address of the page table in main memory. When the MMU does not find a page table entry in the TLB, it uses the Translation Table Base Register (TTBASE) to calculate the physical address where it reads the page table entry. Changing page table base from one page table to another likely affects the mappings or cacheability of certain areas of memory, so it is important that the L1 caches and BTB are empty when the mappings are changed. When a line was valid in the cache and its virtual address changed or became no longer cacheable, unpredictable results occur.

Example 8. Changing the Page Table Base

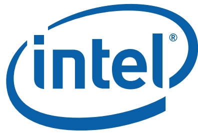
```
@ This code sets the page table base register. The address in this register
@ defines the base address that are accessed when a 'table walk' is
@ performed.

.set    TTBR_OC, 0x18              @ Page table entries are outer cacheable

@       NOTE: ICache, DCache, and the BTB need to all be cleaned/invalidated before
@       this point, to ensure any data which resides in memory
@       locations that are affected by new mappings is not cached in any way.
@       When any updates to the page table about to be used have just been made,
@       It is necessary to also invoke a DWB operation.
    ldr r1, =page_table_base
    orr r1, r1, #TTBR_OC          @ Allow table walks to be L2 cacheable
    mcr p15, 0, r1, c2, c0, 0     @ Write the page table base
    mcr p15, 0, r0, c8, c7, 0     @ Globally invalidate I, D TLB
    CPWAIT r1
```

After the page table base has changed, both the instruction and data TLBs are invalidated, because these contain cached page table entries from the old page table, and any accesses to those entries produce a 'hit' without correctly checking the values in memory. Finally, a CPWAIT macro is used to ensure that any code executing after this code sequence sees the new mappings.

When page table entries are being modified by software, it is also important that similar steps be taken to ensure stale data is not cached anywhere within the 3rd generation microarchitecture. It is necessary to invalidate both the instruction and data TLBs, in case any of the entries being modified had previously been cached in the TLBs.

It is also necessary to clean and invalidate lines in the L1 data cache that exist due to the memory accesses to the page table entries. In particular stores to the page table must be properly forced out to memory. Once the TLB has been invalidated and the L1 data cache has been cleaned and invalidated, the next memory access to the virtual address, whose translation was modified, results in a table walk which accesses the updated information.

The entries for the address space containing the code executing the change in the page table base must not have its mappings changed while it is executing. When the mapping changes, memory accesses (such as those fetching future instructions into the pipeline) cause entries to be created in the instruction TLB after it has been invalidated, but before the effect of the page table change was observed. When the virtual to physical address translation of those entries was changing between the two sets of page tables, then the old ones become cached after this code had completed.



Example 9. Incrementally Locking Instruction TLB Entries

```
    .align 6
@ r0, r1 and r2 contain the virtual addresses to translate and lock into
@ the instruction TLB. The TLB entries specified by r0, r1, and r2 must
@ not already be locked in the ITLB.
@ Hardware ensures that accesses to CP15 occur in program order

    mcr p15,0,r0,c8,c5,1     @ Invalidate ITLB Entry specified in r0.
    mcr p15,0,r1,c8,c5,1     @ Invalidate ITLB Entry specified in r1.
    mcr p15,0,r2,c8,c5,1     @ Invalidate ITLB Entry specified in r2.
    mcr p15,0,r0,c10,c4,0    @ Translate virtual address (r0) and lock into
                             @ instruction TLB
    mcr p15,0,r1,c10,c4,0    @ Translate virtual address (r1) and lock
                             @ into instruction TLB
    mcr p15,0,r2,c10,c4,0    @ Translate virtual address (r2) and lock into
                             @ instruction TLB
    CPWAIT r0

@ The MMU is ensured to be updated at this point; the next instruction
@ sees the locked instruction TLB entries.
```

At times it is desirable to lock entries in the TLB to improve performance. When an entry is locked in the TLB it is never evicted and always produces a 'hit' when the virtual address is translated. When a translation is cached in the TLB it is not necessary for the MMU to execute a 'table walk', which creates a delay while it retrieves the information from the L2 cache or even main memory.

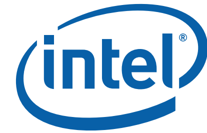
Note: When locking an entry into the TLB, when the virtual address being translated were already in the TLB and valid, then issuing a lock operation on that address, produces unpredictable results.

For this reason, it is necessary to ensure that the address is not valid in the TLB first, either by globally invalidating the TLB or by specifically invalidating the virtual addresses which are about to be locked. Also due to this restriction, take care that the code executing the lock is not in the same page (referenced through the same TLB entry) as the address being locked. There is the possibility that the very end of this code sequence occurs on a page boundary, and the instructions after the code sequence then occurs on a new page (and therefore a new TLB entry). When the TLB entry being locked was the next entry after the code, it is already in the TLB when the lock occurs, due to fetching and speculative execution of the instructions after the CPWAIT.

Note: An alignment directive is used to ensure the code sequence does not cross a page boundary to prevent a table walk from the above scenario.

Since this is an example of 'incremental' TLB locking, where system software repeatedly calls this function to lock different entries each time, the specific entry invalidate operation is used. This ensures that the entry being locked is not present in the TLB when the lock operation is performed. It is important that the entries being invalidated are not locked in the TLB, were these then invalidated by MVA operation have unpredictable results. Because there is no method to query the TLB for which entries it has locked, it is necessary to store this information elsewhere in order to check when an entry was already locked in the TLB. For this reason, it is recommended that the system software maintain a table of locked entries.

Only 31 entries in the TLB are lockable. Attempting to lock the 32nd way fails and the request to lock that entry is ignored.



Example 10. Invalidating and Locking Data TLB Entries

```
@ r1, and r2 contain the virtual addresses to translate
@ and lock into the data TLB
    mcr p15,0,r1,c8,c6,1      @ Invalidate the data TLB entry specified by the
                              @ virtual address in r1
    mcr p15,0,r1,c10,c8,0     @ Translate virtual address (r1) and lock into
                              @ data TLB

@ Repeat sequence for virtual address in r2
    mcr p15,0,r2,c8,c6,1      @ Invalidate the data TLB entry specified by the
                              @ virtual address in r2
    mcr p15,0,r2,c10,c8,0     @ Translate virtual address (r2) and lock into
                              @ data TLB
    CPWAIT r2                 @ wait for locks to complete

@ The MMU is ensured to be updated at this point; the next instruction
@ sees the locked data TLB entries.
```

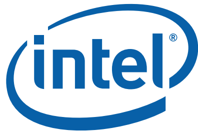
This code sequence illustrates how to lock page table entries in the data TLB. The instruction is virtually identical to the instruction TLB locking sequence, except that CRm=8 instead of 4. For this example, the specific addresses being locked are first invalidated in the data TLB. This is a way of avoiding the previously mentioned problem where locking an address in the TLB that is already valid in the TLB causes unpredictable results. When the lines being invalidated were not in the TLB, then the invalidation instruction acts like a no-op and has no results. It is still necessary to ensure that any of the entries being locked have not already been locked into the TLB. For this reason it is still necessary to maintain a separate table of locked TLB entries.

In the following examples, both the instruction and data TLB instructions are shown together in the same code sequences. Both operations are nearly identical, and both have the same effect on respective TLBs. The only difference between these, is the CRm field of the co-processor access instruction.

Example 11. Invalidating TLB entries

```
@ This code invalidates a specifc entry in the TLB. The entry invalidated
@ corresponds to the virtual address specified in r0.
@ To invalidate in the iTLB, CRm=5. To invalide in the dTLB, CRm=6

    mcr p15, 0, r0, c8, c5, 1     @ Invalidate addr in r0 in instruction TLB
    mcr p15, 0, r0, c8, c6, 1     @ Invalidate addr in r0 in data TLB
```

The instructions above invalidate specific lines in either the instruction or data TLBs depending on which co-processor access instruction is issued. This operation was used in the previous examples just before locking, in order to ensure that the entry is not valid and unlocked in the TLB. Note, that when the entries being invalidated are in the TLB and are locked, the result is unpredictable. It is important for this reason to first globally unlock the TLB or use a table of locked entries to check when it has already been locked.



Example 12. Globally Invalidating the TLBs

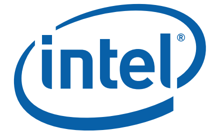
```
    mov r0, #0                    @ The data in R0 is to be zero
@ This code globally invalidates both the instruction and data TLBs
@ The data in r0 is ignored.
    mcr p15, 0, r0, c8, c7, 0    @ Globally invalidate I, D TLB
```

Both the instruction and data TLBs are globally invalidated by writing to co-processor 15, register 8. The operation simultaneously ensures that no valid lines are left in either the data or instruction TLBs. However, when any TLB entries locked before this operation, are not invalidated, remain locked with contents unchanged. In order to ensure that the global invalidate operation leaves no valid entries in either TLB, it is necessary to issue a global unlock on each TLB before invalidating.

Example 13. Globally unlocking the TLB

```
@ This code globally unlocks the instruction and data TLBs. When any entries
@ are valid and locked in the targeted TLB when this code executes, these
@ become unlocked and remain unchanged.
@ To globally unlock the iTLB, CRm=4. To globally unlock the dTLB, CRm=8.

    mov r0, #0                     @ The data in R0 is to be zero
    mcr p15, 0, r0, c10, c4, 1     @ Globally unlock instruction TLB
    mcr p15, 0, r0, c10, c8, 1     @ Globally unlock data TLB
```

This example shows how to globally unlock all entries in both the instruction or data TLB. This is performed by writing to the co-processor 15 register with CRm=4 for the iTLB and CRm=8 for the dTLB. Since it is required that entries in the TLB are unlocked before these are invalidated, it is useful to globally unlock the TLB to be certain that no locked entries exist. Then the TLB is invalidated (either globally or via specific addresses) without the possibility of a locked entry having an invalidate instruction issued against it.



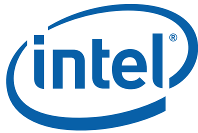
3.2 Cache Management

The 3rd generation microarchitecture contains 32 KB of L1 data cache, 32 KB of L1 instruction cache, and 0 KB, 256 KB, or 512 KB of L2 cache. All microarchitecture cache management operations are performed by utilizing co-processor 15, register 7 (cache functions), register 9 (cache lock down), and register 1 (control). Operations are typically "global", meaning one instruction has an effect on the entire cache, or are "per-line", meaning each cache control instruction has an effect on a single cache line. In the case of "per-line" instructions, the line is specified either using a modified virtual address or by a set and way index.

Some of the examples in this section contain references to a CPWAIT macro. This macro is used as a way to delay execution of subsequent code until the previous co-processor 15 operation has taken effect, refer to Section 2, "CPWAIT Macro" on page 11 for more information.

The caches on the 3rd generation microarchitecture help to isolate its execution from the latency of accessed memory and buses. The microarchitecture attempts to access the L1 or L2 cache, in lieu of sending a memory operation all the way out to memory, and this reduces the number of cycles a memory operation instruction must wait before it is retired. In order for some memory and device access routines to function correctly however, certain memory regions and certain explicit memory operation ordering must occur some of the time. The code sequences in this section are reference examples that explain how system software controls the cache, allowing some control over how and when memory operations are observed outside the microarchitecture.

Because cache sizes are different depending on the ASSP, it is necessary to check the size at run-time in order to allow the software to be portable. The co-processor 15 ARM cache type registers contain information about both the L1 and L2 caches.



3.2.1 L1 Cache

The 3rd generation microarchitecture L1 cache consists of separate 32 KB 4-way set associative instruction and data caches. When a region of memory is marked cacheable in the L1, a fetch of either instruction or data memory cause a cache line allocation and fill. When a subsequent instruction fetch occurs, when the data is in the L1 cache and the line is valid, the instruction from the cache line is used and no external memory operation occurs. The operations described in this section allow system software to explicitly cache, clean, invalidate, lock, and unlock lines in the L1 instruction and data caches.

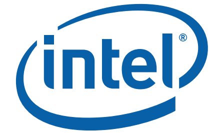
It is important to note that the operation of the L1 and L2 caches in the 3rd generation microarchitecture is closely linked to the operation of the MMU, the TLB, and the BTB. More information on these components are found in the following sections: "Virtual Memory (MMU)" on page 16, "Translation Look Aside Buffers (TLB)" on page 20, and "Branch Target Buffer (BTB)" on page 47.

3.2.1.1 L1 Instruction Cache Enable/Disable

The 3rd generation microarchitecture L1 instruction cache is enabled independently from the MMU or BTB. When the L1 instruction cache is enabled, on every instruction fetch that is found to be cacheable by the MMU (or every fetch when the MMU is disabled), the instruction cache are checked and when the instructions are found there these are used. When the address is not found in the L1 instruction cache, a fill request is issued and a new cache line is allocated to contain the instruction data from memory. Although these operations refer to 'enabling' and 'disabling' the L1 instruction cache, it is better to view these operations as merely enabling and disabling the ability of the L1 instruction cache to fill a cache line.

In other words, with the exception of a new line being allocated and filled in the L1 instruction cache, every aspect of the L1 instruction cache remains active all of the time. This means that when a cacheable instruction fetch occurs, the L1 instruction cache in the 3rd generation microarchitecture is checked prior to any access to external memory. When the system software elects to ensure that the L1 instruction cache is never 'hit', it is necessary to both disable its ability to allocate and fill new lines, and to invalidate all the lines which already exist in the L1 cache. When completely invalidated and unable to fill new lines, the L1 instruction cache is effectively disabled.

Note: This behavior (hitting the cache when it is disabled) is deprecated on the 3rd generation microarchitecture. It is possible this behavior changes on future generations so do not let software depend it its existence.



Example 14. Enable the L1 Instruction Cache

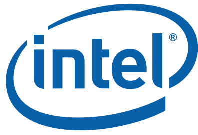
```
@ Enable I-cache
    mrc p15, 0, r3, c1, c0, 0   @ Get control register
    orr r3, r3, #0x1000         @ Set I bit (bit 12)
    mcr p15, 0, r3, c1, c0, 0   @ Update control register
```

The code above enables the L1 instruction cache on the 3rd generation microarchitecture. This is done by reading Register 1 of the Control Register and modifying it so bit 12 (L1 I-Cache Enable/Disable) is turned on. Unlike the L1 data cache, the L1 instruction cache on the microarchitecture is enabled at any time, independently of the MMU or BTB. However, it is important to note that the behavior of the L1 instruction cache is very different with respect to the MMU being enabled or disabled. When the MMU is disabled and the L1 I-cache is enabled, every instruction fetch, regardless of address, is cached. When the MMU is enabled and the L1 I-cache is enabled, whether or not instructions are cached depends on the cacheable bit being set on the page corresponding to the instruction memory address being fetched. Also when the MMU is off, no address translation takes place. Therefore the addresses in the L1 instruction cache when the MMU is off are physical addresses.

Example 15. Disable the L1 Instruction Cache

```
@ Disable I-cache
    mrc p15, 0, r3, c1, c0, 0     @ Get control register
    bic r3, r3, #0x1000           @ Clear I bit (bit 12)
    mcr p15, 0, r3, c1, c0, 0     @ Update control register
@ Now that I-cache is disabled, it is necessary to invalidate
@ the L1 I-Cache and the BTB as described in the text.
```

The L1 instruction cache on the 3rd generation microarchitecture is disabled by reading Register 1 of the Control Register and modifying it so bit 12 (L1 I-Cache Enable/Disable) is cleared. After this operation, even though the instruction cache is disabled, any lines that are not invalid cause a hit when memory address is fetched. When this is not the desired behavior, it is the responsibility of system software to invalidate the L1 instruction cache as well as the BTB, in order to ensure future instruction fetches do not hit the L1 cache. Operations on the L1 cache, such as invalidation and unlocking, work with the cache enabled or disabled. The reason the L1 instruction cache is invalidated after it is disabled, rather than before, is to avoid caching the code that does the disabling (and any instructions also in the lines it occupies), which results in the code being 'trapped' in the L1 instruction cache after the cache becomes disabled.



3.2.2 L1 Instruction Cache Operations

This section describes various operations that have an effect on L1 instruction cache lines, including locking, unlocking and invalidating on a per-line and global basis.

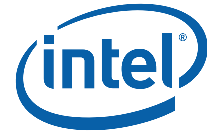
It is desirable to lock certain L1 instruction cache code routines. This favors achieving higher performance on code often executed, such as interrupt handler routines. When instructions are locked in the L1 instruction cache, these are always available and always produce a cache 'hit' every time the memory location is executed.

Example 16. Lock Lines In the L1 Instruction Cache

```
codeLock:                @ This code locks the "lockMe" routine
    ldr r0, =(lockMe)    @ ptr to first cache line to lock
    bic r0, r0, #0x1F
    ldr r1, =(lockMeEnd) @ ptr to last cache line to lock
    sub r1, r1, #1       @ up to last instruction of function
    bic r1, r1, #0x1F
lockLoop:
    mcr p15, 0, r0, c7, c5, 1  @ invalidate IC Line first
    mcr p15, 0, r0, c9, c5, 0  @ lock next line into Icache
    cmp r0, r1                 @ check when done
    add r0, r0, #32            @ increment by cache line size
    bne lockLoop               @ when not done, do next line

    b finished
    nop
    nop
    nop
    nop
@ ------------------------------------------------
    .p2align 5                 @ Align the function on a cache-line boundary
lockMe:                         @ This is the code to lock into I-cache
    mov r1, #5
    add r1, r1, #8
    mov pc, lr
    @...
lockMeEnd:
```

The previous code sequence locks an L1 instruction cach function. It accomplishes this by calculating the cache line - aligned start and end address of a function, then looping through each cache line the function occupies, locking the line. The line lock operation is performed by writing the modified virtual address to co-processor 15, register 9. It is important that no part of the code being locked is already present in the L1 instruction cache before this operation is executed.

When an address being locked is already in the instruction cache, the operation fails and the line is not locked. For this reason, each line in the instruction cache is invalidated before it is locked, to avoid this possibility. Due to the fact that way0 of each set cannot be locked, a maximum of 24 KB of code is locked in the L1 instruction cache. However, in practice the maximum is less, due to the limitation that only 3 ways of each set are locked at a time.



In general, it is useful for system software to maintain a table of locked lines. Since there is no method for determining how 'full' the instruction cache is with locked lines, it is necessary to check a separate table, in order to determine when a line lock operation is successful. Because fetches for speculative execution cause instructions after the end of the lock loop to be fetched into the instruction cache, it is possible in the case where the code being locked is part of the next cache line, that the last line lock fails because the line is already in the cache.

For this reason, enough NOPs to fill the pipeline are inserted after the loop, so these are fetched instead of the code after the end of the loop. The function occurring next is aligned on a cache line boundary, so it is ensured it does not occupy the same cache lines as the function performing the lock.

Example 17. Globally Unlock the L1 Instruction Cache

```
    mov r1, #0                  @ The data in R1 is to be zero
    mcr p15, 0, r1, c9, c5, 1   @ unlock all lines in the L1 I-cache
                                @ data in R1 is ignored
```

The code above globally unlocks the L1 instruction cache. This operation does not modify or invalidate any contents of the valid lines in the instruction cache. After this operation is complete, no lines are locked, and these are potentially evicted when an instruction fetch occurred. The L1 instruction cache is unlocked by performing a write to co-processor 15, register 9. The contents of the data written are ignored. This operation is performed and has the same effect with the L1 instruction cache enabled or disabled.

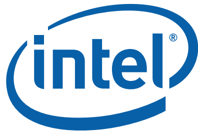
It is necessary to invalidate instructions in the L1 cache when these have been modified externally and the modification needs to have taken effect the next time the 3rd generation microarchitecture executes those instructions. When L1 instruction cache lines are invalidated, it ensures that the next time an instruction fetch to that memory address occurs, the data is retrieved externally from the L1 cache.

Example 18. Unlock and Invalidate L1 Instruction Cache Lines

```
@ This code unlocks & validates a memory region in the l1 I-cache.
@ r0 contains the MVA of the address to start the unlocking
@ r1 contains the number of 32-byte lines to invalidate & unlock.
@ In this example, 16 lines are unlocked and invalidated.

    mov r1, #16

unlockLoop:
    mcr p15, 0, r0, c7, c5, 1   @ invalidate and unlock address R0 in I-cache
    add r0, r0, #32             @ increment by cache line size
    subs r1, r1, #1             @ decrement loop counter, set flags
    bne unlockLoop              @ when not done, do next line
```

This example invalidates cache lines in the L1 instruction cache. The number of lines specified in register r1 are invalidated sequentially, starting at the line that corresponds to the modified virtual address stored in register r0. This is done by incrementing a pointer by the size of a line and writing the address to co-processor 15, register 7. When the cache does not contain a valid entry corresponding to the modified virtual address, no action is taken.

Before the invalidation takes place, this operation also unlocks the line when it is locked. For this reason, the invalidate operation is used as a way to unlock only certain routines in the L1 instruction cache, without having to unlock all of thesewith a global unlock.



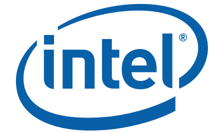
It is important to be aware that the BTB also contains information relating to instructions. When routines are invalidated in the L1 instruction cache, any corresponding branch entries in the BTB are not invalidated. When the purpose of the operation is to allow the 3rd generation microarchitecture to execute code that has been modified in memory, then it is also necessary to invalidate the BTB, to ensure the new code when fetched has the correct entries in the BTB. When the BTB and the L1 instruction cache are not in sync, then the results are unpredictable.

Example 19. Globally Invalidate the L1 Instruction Cache

```
mov r1, #0                   @ The data in R0 is to be zero
mcr p15, 0, r1, c7, c5, 0    @ Invalidate the I-cache and BTB
                             @ Data in r1 is ignored
```

The code above performs a global invalidation of the L1 instruction cache. It does this by performing a write to co-processor 15, register 7, the contents of the register written are ignored. Note, that when any lines in the L1 instruction cache are locked, these are not touched and remain valid, and locked after this operation. When ensured global invalidation is the desired behavior, and when there is the possibility of locked lines existing in the cache, then the cache lines must be unlocked either through a global unlock operation, or by virtual address before the global invalidate.

This operation is used as part of the sequence to disable operation of the 3rd generation microarchitecture L1 instruction cache, by first disabling the cache and second ensuring all lines are invalidated. Also note, that since the BTB also contains branch prediction data, it is implicitly invalidated as well to prevent stale branch data from causing unpredictable behavior.



3.2.3 L1 Data Cache Enable/Disable

The 3rd generation microarchitecture L1 data cache is dependent on the MMU. In order for it to be enabled, the MMU must have been enabled first. The MMU performs the address translation lookup, and determines when the data being accessed is cacheable in the L1 data cache. When the address being accessed is determined to be cacheable, then the L1 data cache allocates a line and fills it with data from the address being accessed on a read.

For writes, when a valid cache line already exists, the 3rd generation microarchitecture L1 Data cache is configured to write only to the cache line (write-back), or write to the cache line as well as out to external memory (write-through). This behavior is determined by the page table entry bits corresponding to the memory address being accessed.

Example 20. Enable the L1 Data Cache

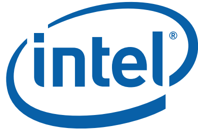
```
@ MMU must be enabled
@ Enable dcache
    mrc p15, 0, r3, c1, c0, 0  @ Get Control Register
    orr r3, r3, #0x0004        @ Set C Bit (bit 2)
    mcr p15, 0, r3, c1, c0, 0  @ Update control register
```

This routine enables the L1 data cache in the 3rd generation microarchitecture. This is accomplished by reading, modifying, and writing bit 2 (L1 D-Cache enable/disable) in co-processor 15 register 1. Note, that the MMU must be enabled prior to or at the same time as the enabling of the L1 data cache. This requirement is different than the L1 instruction cache, which does not require the MMU to be enabled in order to operate. When this routine is executed without the MMU being enabled, the results are unpredictable. When the L1 data cache is enabled, memory pages that are marked as cachable allow a cache 'hit' when the data these contain is already in the cache, and memory pages marked as write-back is cached on write hits.

Example 21. Disable the L1 Data Cache

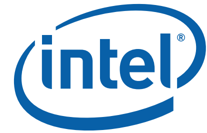
```
@ Disable dcache
    mrc p15, 0, r3, c1, c0, 0  @ Get Control Register
    bic r3, r3, #0x0004        @ Clear C Bit (bit 2)
    mcr p15, 0, r3, c1, c0, 0  @ Update control register

@ Now unlock, clean, and invalidate L1 D-cache
```

This code disables the L1 data cache in the 3rd generation microarchitecture. This is accomplished by reading, modifying, and writing bit 2 (L1 D-Cache enable/disable) in co-processor 15 register 1. Note, that even though the L1 data cache is disabled after this operation completes, any valid lines in the cache results in both cache hits on reads for regions marked cacheable, as well as writes for memory regions marked write-back. This behavior is unlikely to be desired because any changes to memory in the regions that are in the cache are not observable to the processor, and any writes to regions in the cache are written into the cache, but never cleaned out. For this reason, it is necessary to unlock, clean, and invalidate all L1 data cache lines. Because invalidate, clean, and unlock operations still affect the cache, even though it is disabled, this is done after the L1 data cache is disabled, to ensure no new lines are allocated during this operation or afterwards.



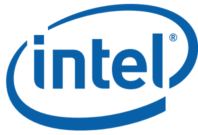
3.2.4 L1 Data Cache Operations

This section describes various operations that are performed on the L1 data cache. These operations change the behavior of regions of the data cache, as well as assist in turning it on or off. Lines in the data cache are locked, cleaned (written out when dirty), and invalidated, either globally or on a per line basis. Regions of the L1 data cache are used to permanently hold and cache a small part of memory, or these are configured to act as a small amount of high-speed “scratch memory”, that is not present in allocated system memory. Specific regions of locked data memory are cleaned, but still remain locked in cache, in order to have the contents written out to main memory in a cache-line burst operation, even though these are written in small amounts multiple times.



Example 22. Lock Lines in the L1 Data Cache (Sheet 1 of 2)

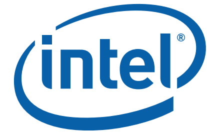
```
@
@ Restrictions:
@
@  Prefetch abort handler MUST turn off lock mode bit
@  as soon as possible.
@
@  Data abort handler MUST turn off lock mode bit as
@  soon as possible.
@
@  When there are any imprecise aborts during locking
@  the result of locking is unpredictable.
@
@  NOTE: This routine always locks at least one line.
@        End address needs to be greater than start address.

@ PSR bit defines
.setPSR_I, 0x80
.setPSR_F, 0x40

@ LockDCache flags
.setL_LEN, 0x01
.setL_INV, 0x02

@ Data Cache Lock Mode
.setM_LOCKED, 0x01
.setM_NOT_LOCKED, 0x00

@ Cache Line Size
.setLINE_SIZE, 0x20
@
@ _LockDCache(Start,End_or_Length,Flags)
@
@ Inputs
@   R0  Start Address
@   R1  End Address (inclusive) or Length
@   R2  Flags
@
@ Outputs
@   R0  Success (1) / Failure (0)
@
@ Registers used
@   R1 - R3, IP
@
@ Usage:
@
@  When a region length is specified in R1 then the
@  L_LEN flag must be set. All data cache
@  unlocked and invalidated when the L_INV flag is set.
@  When several things are to be locked into the cache
@  the L_INV flag needs to be set for the first item
@  to be locked and clear for subsequent items.
@
```



Example 22. Lock Lines in the L1 Data Cache (Sheet 2 of 2)

```
    .p2align 2
    .global _LockDCache
    .type_LockDCache,function
_LockDCache:
    mrs ip, cpsr@ Save CPSR
    orr r3, ip, #PSR_I | PSR_F
    msr cpsr_c, r3@ Disable Interrupts

    mrc p15, 0, r3, 1, 0, 0
    mcr p15, 0, r3, 1, 0, 0@ Force a DCU drain

    tst r2, #L_INV
    mcrnep15, 0, r0, c9, c6, 1@ Unlock D-Cache
    mcrnep15, 0, r0, c7, c6, 0@ Invalidate D-Cache

    mov r3, #M_LOCKED
    mcr p15, 0, r3, c9, c6, 0@ Set Lock Mode

    tst r2, #L_LEN
    addner1, r1, r0
    subner1, r1, #1

CacheFill:
    ldr r2, [r0], #LINE_SIZE
    mrc p15, 0, r3, 1, 0, 0
    mcr p15, 0, r3, 1, 0, 0@ Force a DCU drain
    cmp r0, r1
    ldrlsr3, [r0], #LINE_SIZE
    mrclsp15, 0, r3, 1, 0, 0
    mcrlsp15, 0, r3, 1, 0, 0@ Force a DCU drain
    cmplsr0, r1
    bls CacheFill

    orr r2, r2, r3@ Create a dependency stall

    mrc p15, 0, r0, c9, c6, 0@ Read Lock Mode
    mov r2, #M_NOT_LOCKED
    mcr p15, 0, r2, c9, c6, 0@ Clear Lock Mode

    msr cpsr_f, ip@ Reset Interrupt flags

    and r0, r0, #M_LOCKED@ Test lock flag
    mov pc, lr

.Lfe1:
    .size_LockDCache,.Lfe1-_LockDCache
```

The above routine in Example 22 locks lines of data from memory into the L1 data cache. This is useful as a high-performance method for accessing periodically used symbols or lookup tables in close proximity in memory. Because of the fact that way0 of the data cache cannot be locked, a maximum of 24 KB of L1 data cache is locked. In practice, the maximum is less than that, depending on the addresses of the locked lines and due to the fact that at most 3 ways of each set are locked. The locking is accomplished by putting the L1 data cache in 'lock mode' by writing a '1' to co-processor 15 register 9. Once the data cache is in 'lock mode', any newly allocated cache lines have the lock bit set, which prevents these from being evicted. The routine puts the data cache in lock mode, then proceeds to load the first word of each cache line using a 'ldr' instruction. The data being loaded into register 'r2' or 'r3' is thrown away, since the 'ldr' instruction is used just to cause the line to be fetched and locked. Loading the first word causes the 3rd generation microarchitecture to fetch the full 32-bytes to fill each cache line.



Note: This routine takes advantage of a special 3rd generation microarchitecture feature — writing the Control Register has the side effect of forcing a Data Cache Unit (DCU) drain. Do not depend on this behavior in other microarchitectures!

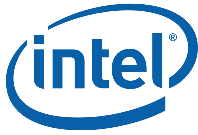
It is important to note that when any part of the data being fetched already existed in the L1 data cache, the data is not fetched into the cache, and the lock operation for that cache line does not occur. To avoid this, the code above has a flag that unlocks and invalidate all data in the cache before locking any new data.

Note: Any existing data is lost so the cache needs to be cleaned before the lock routine is called in this case.

Additionally, care must be taken to ensure the processor is not interrupted during this operation. When the lock mode is on, any memory operation resulting in a data cache fill causes that line to be locked, and were an interrupt handler to execute during this time, undesired results occur. It is recommended that interrupts be disabled when performing this operation. For more information on interrupts in the 3rd generation microarchitecture, see "Disable Interrupts" on page 62, and "Enable Interrupts" on page 62.

Example 23. Create L1 Data Cache RAM

```
@ R1 contains the virtual address of a region of memory to configure as data RAM,
@ which is aligned on a 32-byte boundary.
@ MMU is configured so that the memory region is cacheable.
@ R0 is the number of 32-byte lines to designate as data RAM. In this example 16
@ lines of the data cache are re-configured as data RAM.
@ MMU and data cache are enabled prior to this code.
@ Care must be taken to ensure no interrupts occur during the time 'lock mode'
@ is enabled. It is recommended interrupts be disabled during this operation.
.macro ALLOCATE, Rx
    mcr p15, 0, \Rx, c7, c2, 5
.endm
.macro BARRIER, Rd
    mcr p15, 0, \Rd, c7, c10, 5  @ DMB Operation to provide memory barrier
                                 @ contents of \Rd ignored
.endm

    BARRIER r0
    mov r2, #0x1
    mcr p15, 0, r2, c9, c6, 0   @ Put the data cache in lock mode
    CPWAIT r3
    mov r0, #16
LOOP1:
    ALLOCATE r1                 @ Allocate and lock a tag into the data cache at
                                @ address [R1].

    @ Note that newly allocated line contains unpredictable data.
    @ The caller to this function ensures it writes the line with known
    @ data to avoid using the unpredictable data.
    add r1, r1, #32             @ Increment to the next line
    subs r0, r0, #1
    bne LOOP1

    @ Turn off data cache locking
    mov r2, #0x0
    mcr p15, 0, r2, c9, c6, 0   @ Take the data cache out of lock mode.
    CPWAIT r3
```



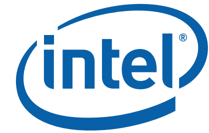
This example creates “data cache RAM” by locking lines that correspond to an unallocated memory region in L1 data cache. This is useful when a high-performance scratch pad region is required, either to use as a place to store values that do not fit in the available registers, or as a place to consolidate and quickly access symbols which are widely dispersed in physical memory.

Because way0 of the L1 data cache cannot be locked, a maximum of 24 KB of L1 data cache is locked. The actual number of lines is less than that, due to the fact that only 3 ways of each set are locked.

The creation of data RAM is accomplished by putting the L1 data cache into “lock mode”, by writing to co-processor 15 register 9. Once the data cache is in “lock mode”, any newly allocated cache lines have the lock bit set, which prevents these from being evicted. The data RAM is created by executing a cache line allocate operation for each line that needs to be locked. The allocate operation causes the L1 data cache to allocate a cache line for the specified memory region, but does not issue a fetch for the data. This allows the allocation of cache lines for memory regions that do not exist in the system.

Note, that the memory region used for data cache RAM must have valid page table descriptors, which are set to be cachable and writeback. This is because the page table entry for the address is checked when a memory operation references the data RAM, and it must be set so reads and writes both hit the cache and do not go out to memory. After this operation is complete, the lines locked are accessible for reads and writes by simply loading or storing to the memory region. See “L1 Data Cache Line Allocation” on page 40 for another example of the uses of the L1 data cache line allocate operation.

As in the previous example, it is important that the code be allowed to run without handling any interrupts while lock mode is enabled, or data becomes locked in cache that was used as part of the interrupt handler.



Example 24. Globally Unlock the L1 Data Cache

```
mov r1, #0                      @ The data in R1 is to be zero
mcr p15, 0, r1, c9, c6, 1       @ globally unlock all lines in the d-cache
                                @ data in r1 is ignored
```

This code unlocks all lines in the L1 data cache by writing to co-processor 15, register 9. The contents of the L1 data cache lines (both locked and unlocked) are not altered. After execution of this code, any data in the L1 data cache is potentially evicted when a new line is fetched and there is no invalid line in the set to store the incoming data in.

Note, that some L1 cache locking functions were moved on the 3rd generation microarchitecture, and that this code uses the new function. The old functions are deprecated. This operation is issued in a situation where system software wanted to ensure that all lines in the L1 data cache were unlocked; for example, just before a global invalidation during the disable sequence of the L1 data cache.

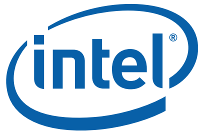
Example 25. Globally Invalidate the L1 Data Cache

```
mov r1, #0                     @ The data in R1 is to be zero
mcr p15, 0, r1, c7, c6, 0      @ globally invalidate all lines in the d-cache
                               @ data in r1 is ignored
```

This code globally invalidates all lines in the L1 data cache by writing to co-processor 15, register 7. Any lines in the data cache are marked as invalid, and when the contents had been modified, the data was not written out to memory.

Note, that when any of the data cache lines were locked at the time this operation was executed, these are not invalidated or unlocked; the contents remain locked in the L1 data cache unaltered. When ensuring invalidation of the entire data cache is desired, and the possibility of locked lines exists, software must globally unlock the data cache before performing the global invalidation function.

This operation is useful when the system software wanted to have a known state for the contents of the L1 data cache. For example, when the system is beginning its boot sequence after having been started by a platform boot-loader or firmware, it is better to invalidate the L1 data cache than clean it, because it is undesirable to have any of the unknown contents of the cache written out to memory.



Example 26. Clean, Invalidate, and Unlock Lines in the L1 Data Cache

```
@ r1 contains the virtual address of a region of memory to clean & unlock
@ r0 is the number of 32-byte lines to clean, invalidate and unlock in
@ the data cache.
@ MMU and data cache are enabled prior to this code.

LOOP1:
    mcr p15, 0, r1, c7, c14, 1 @ Clean and invalidate in one operation
    add r1, r1, #32            @ increment addr in r1 to the next cache line

    subs r0, r0, #1            @ Decrement loop count
    bne LOOP1
@ The data at the addresses in r1 has been written out when dirty, unlocked,
@ and the cache lines are now invalid.
```

This code cleans, invalidates, and unlocks cache lines corresponding to a region of memory in the L1 data cache. This is done by writing the cache line aligned address to co-processor 15, register 7. When data exists in the L1 Data Cache corresponding to the modified virtual address being invalidated, it is written out when modified (inother words, the dirty bit is set), unlocked when it was locked, and invalidated.

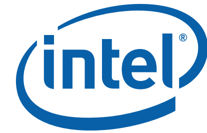
This operation is used during the shutdown or exit process of a device driver that had allocated locked lines in the L1 data cache in order to have higher performance. By using the clean, invalidate, and unlock operation it simultaneously writes out any changes to memory, then free up the cache lines for use by other code by invalidating and unlocking these.

Example 27. Clean L1 Data Cache Lines

```
@ r1 contains the virtual address of a region of memory in
@ the L1 D-cache to clean.
@ r0 is the number of 32-byte lines to clean in the data cache.
@ In this example 16 lines of data are cleaned.
@ MMU and data cache are enabled prior to this code.

    mov r0, #16
LOOP1:
    mcr p15, 0, r1, c7, c10, 1 @ Write out the line when its dirty in the cache
    add r1, r1, #32            @ increment addr in r1 to the next cache line

    subs r0, r0, #1            @ Decrement loop count
    bne LOOP1

@ At this point the data at address r1 has been written out
@ when it had been modified.
```

This operation is used to clean both locked and unlocked lines in the L1 data cache, by writing the modified virtual address of the locked data to co-processor 15, register 7. When the lines are locked, these remain locked and unaltered after this operation completes, and when these are modified the contents was written out. Any modified virtual address that has valid page table descriptors are cleaned, and when the contents of the line are valid and have been modified it is written out to memory.

This operation is typically used as both a way to ensure that writes to a memory region have been cleaned out of the cache (in the case of lines that are not locked) and as a way to exercise control over when writes are cleaned out of the cache (in the case of locked lines).



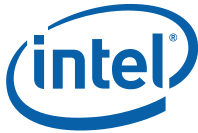
Example 28. Globally Clean (and Invalidate) L1 Data Cache by Set & Way

```
@ Clean (and invalidate) by set/way
@ Increment the way index in the inner loop while decrementing the set index
@ in the outer loop. Check flags to decide when to branch.
@ Number of ways per set: 4
@ Number of Sets:         256
@ Defines for number of sets, and bit position of set and way indexes:
.setNUMSETS, 256
.setWAYINDEX, 0x40000000
.setSETINDEX, 0x20

    mov r0, #NUMSETS
    sub r0, r0, #1
    mov r0, r0, lsl #5          @ Put number of sets-1 into bits 12:5 of r0
CLEANLOOP:
    mcr p15, 0, r0, c7, c10, 2  @ clean set/way specified in r0
@ The operator also uses a clean and invalidate here instead of just a clean
@   mcr p15, 0, r0, c7, c14, 2 @ clean/invalidate set/way specified in r0

    adds r0, r0, #WAYINDEX      @ Increment shifted way index
    bcc CLEANLOOP               @ Clean the next way when not done with this set
    subs r0, r0, #SETINDEX      @ Decrement shifted set index
    bpl CLEANLOOP               @ Go to next set when not at last one
```

Unlike the invalidate and unlock operations, there is no single global clean operation for the L1 data cache. This example performs the equivalent task by iterating through all sets and ways in the L1 data cache and issuing a set/way clean operation by writing to co-processor 15, register 7.

Depending on CRm, this operation is also invalidate the line after it has been cleaned in one operation (CRm=c10 just cleans, CRm=c14 cleans and invalidates). Both forms of the operation are shown in the example above, with the invalidate operation commented out. The set and way is stored in two registers and concatenated into r0.

After this code, all modified lines in the L1 data cache are written out when the dirty bit was set, and all lines are invalid (when the invalidate operation was used). It is important to note that when any lines in the L1 data cache are locked when this operation is executed, the contents are *not* written out to main memory, and these remain locked. When it is desired to clean the contents of locked lines in the L1 data cache, it is necessary to either unlock those lines with a global unlock operation or clean that region by modified virtual address as shown in previous examples.



3.2.5 L1 Data Cache Line Allocation Operation

The 3rd generation microarchitecture L1 data cache allows a user-mode application to access the line allocation function. This function allows software to allocate a line in the L1 data cache, corresponding to a specified virtual address. When the cache line is allocated, no external memory access is performed and the operation completes quickly.

After the line has been allocated, writes to that address produce a cache 'hit' the first time. Because of this, it is a useful method of preparing the cache for a series of writes to a memory location. Because the line is not fetched, its contents are unpredictable following allocation.

Software is required to initialize the entire cache line prior to doing any reads or clean operations to that line. When this is not done, part or all of the data read back (or cleaned) is unpredictable. Since it is a user-mode accessible function, it is possible for user-mode applications to take advantage of the performance improvement line allocating provides.

Example 29. L1 Data Cache Line Allocation

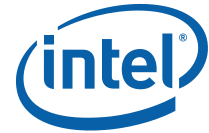
```
@ Allocates cache lines for the specified memory address
@ The virtual address (which is PIDified) to allocate lines for is in r1,
@ The number of bytes (a multiple of 32 which is rounded up to a cache line)
@ is in r2.

    add r2, r2, #31             @ round up to nearest cache line
    mov r0, r2, lsr #5          @ get number of lines, put in r0

LOOP1:
    mcr p15, 0, r1, c7, c2, 5  @ allocate a line at the address in r1
    subs r0, r0, #1             @ Decrement loop count
    add r1, r1, #32             @ Increment the address by 1 line
    bne LOOP1
```

In Example 29 allocating an L1 data cache line is done by writing the cache-aligned virtual address to co-processor 15, register 7. The instruction is issued in user mode, allowing user applications takes full advantage of the performance benefits of this function.

It is useful for an application to allocate a cache line any time it is known that the memory location is written initially (with all lines involved entirely written), and not read first or modified. It ensures that the first write produces a write cache 'hit', and that the data is not written out until the cache lines containing the data are cleaned. This allows the software to ensure a burst memory transaction is generated regardless of the number and size of the individual writes to the cache lines.

Note, that issuing an L1 data cache line allocate instruction when the L1 data cache is disabled results in a no-op, with no action being taken. It is important to note that the line allocation operation takes a VA (virtual address) as opposed to an MVA (modified virtual address). This means that the address being allocated is modified by the Process ID register before being translated by the MMU to a physical address.

Note: The L1 data cache line allocation function is deprecated.



3.2.6 L2 Cache

When implemented, the L2 cache on the 3rd generation microarchitecture is several times the size of the L1 data and instruction caches, and is organized such that instruction and data memory are not distinguished from each other. Instead of 4 ways, the L2 cache contains 8 ways and either 1024 or 2048 sets, depending on whether the L2 cache is 256 KB bytes or 512 KB. Some specific applications do not have an L2 cache present.

Example 30. Obtaining L2 cache information through the cp15 Cache Type Register

```
@ These macros read the L2 Cache Type Register in cp15.

@ This macro sets CPSR.z = 1 when L2 not present, else CPSR.z = 0
.macro GET_L2_PRESENT, Rd
    mrc p15, 1, \Rd, c0, c0, 1 @ Read L2 Cache Type Register for Associativity
    ands \Rd, \Rd, #0xf8        @ See when the L2 Cache is absent
.endm
```

When an instruction or data memory fetch occurs, when the address is not found in the L1 cache and the MMU finds that the memory region is L2 cacheable, the 3rd generation microarchitecture attempts to produce an L2 'hit' by looking for the data in the L2 cache. When the data is present, it is returned to the microarchitecture. This situation requires more microarchitecture clock cycles than when the data was present in the L1 cache, but takes much fewer cycles than a load from main memory.

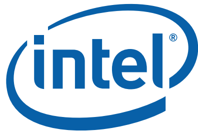
When a write to a region is L2 cacheable, and there is a valid line in the L2 cache corresponding to the address, the write always goes into the L2 cache. When the line is not in the L2 cache and a write occurs, the data is fetched from memory, placed in the cache, and the write 'hits' the line.

This write-back and write-allocate behavior is always the case with the L2 cache on the 3rd generation microarchitecture. The L2 cache is disabled at system reset, and all lines are invalidated and unlocked. Because the MMU is also disabled at system power-on reset, even after the L2 cache is enabled, the L2 cache is effectively disabled until the MMU is enabled.

Example 31. Enable the L2 Cache

```
@ Enable the L2 unified cache on the microarchitecture.
    mrc p15, 0, r3, c1, c0, 0      @ Get Control Register
    orr r3, r3, #0x04000000        @ Set U Bit (bit 26)
    mcr p15, 0, r3, c1, c0, 0      @ Update control register
```

The L2 cache is enabled by modifying the control register (co-processor 15, register 1) so that the U bit (bit 26) is turned on. The L2 cache is disabled at system boot and needs only to be enabled once, and not disabled during the normal operation of the system. Disabling the L2 cache once it has been enabled produces unpredictable results.



Example 32. Invalidate and Unlock L2 Cache Lines

```
@ r1 contains the virtual address of the region of memory to invalidate and unlock
@ r0 is the number of 32-byte lines to invalidate and unlock in the L2 cache.
@ In this example 16 lines of data are invalidated
@ MMU is enabled prior to this code.

    mov r0, #16               @ Number of cache lines to invalidate
LOOP1:
    mcr p15, 1, r1, c7, c7, 1 @ Invalidate any lines corresponding to addr r1
    add r1, r1, #32           @ increment the address in r1 to
                              @ the next cache line

    subs r0, r0, #1           @ Decrement loop count
    bne LOOP1
@ The data at address r1 has been unlocked, and the cache lines are now invalid.
```

Specific L2 cache lines are invalidated and unlocked by accessing co-processor 15, register 7 and writing the modified virtual address of the memory location, which corresponds to the cache line to unlock.

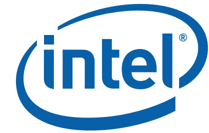
When the cache line corresponding to the address exists and is valid in the L2 cache, it is invalidated, and unlocked when it had been locked. When it had been modified in the L2 cache, its contents were *not* written out to memory.

It is important to note that this operation is weakly ordered along with all other L2 cache access, and it is necessary to issue a bi-directional memory fence operation prior to this code in order to ensure all outstanding L2 memory operations have occurred in program order. For example, when a read had just been issued to one of the addresses being invalidated, it is necessary to wait for it to complete, to ensure the line was still valid at the time of the read, so the contents of the cache line are returned and not the contents of main memory.

Example 33. Clean L2 Cache Lines

```
@ r1 contains the virtual address of a the region of memory to clean
@ r0 is the number of 32-byte lines to clean in the L2 cache.
@ In this example 16 L2 cache lines are cleaned.
@ MMU is enabled prior to this code.

    mov r0, #16
LOOP1:
    mcr p15, 1, r1, c7, c11, 1 @ clean any line corresponding to addr r1
    add r1, r1, #32            @ increment the address in r1 to the next
                               @ cache line

    subs r0, r0, #1            @ Decrement loop count
    bne LOOP1
@ The data at address r1 has been written out of the L2 cache when it was modified.
```

This example performs a clean operation on specific L2 cache lines corresponding to a modified virtual address. When the cache line corresponding to the address exists in the L2 cache and has been modified (in other words, the dirty bit is set) then the modified line is written out of the L2 cache. The contents of the cache line is not modified, and when it was locked it remains locked.



Example 34. Globally Clean and Invalidate L2 Cache by Set and Way

```
    @ Clean and Invalidate L2 by set/way
    @ Increment the way index in the inner loop while
    @ Decrementing the set index in the outer loop. Check flags to decide
    @ when to branch.
    @ Number of ways per set: 8
    @ Number of Sets:         2048

.macro BARRIER, Rd
    mcr p15, 0, \Rd, c7, c10, 5  @ DMB Operation to impose memory fence.
                                 @ contents of \Rd ignored
.endm

    BARRIER r0
    GET_L2_SIZE r0                 @ low four bits returned in r0 are the L2
                                   @ cache size, starting with 64 KB (0b0000)
                                   @ and increasing by powers of 2.
    mov r1, #0xffffffe0
    mov r2, #19
    sub r2, r2, r0                 @ determine the number of sets
    mov r0, r1, lsl r2             @ clear out the extra bits when configuring
    mov r0, r0, lsr r2             @ the number of sets
CLEANLOOP:
    mcr p15, 1, r0, c7, c15, 2 @ clean and invalidate set/way in r0
    adds r0, r0, #0x20000000   @ Increment shifted way index
    bcc CLEANLOOP              @ Clean the next way when not done with this set
    subs r0, r0, #0x00000020   @ Decrement shifted set index
    bpl CLEANLOOP              @ Go to next set when not at last one

    BARRIER r0
```

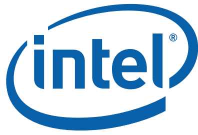
Unlike the unlock and invalidate operations, there is no single operation that globally cleans the L2 cache. Code above performs this task by executing the “clean and invalidate by set/way” operation.

The L2 cache line is specified by writing the set and way to co-processor 15, register 7. The code above issues this operation to all cache lines that are valid in the L2 cache, by iterating the index through all possible sets and ways. The number of sets is determined by calling a macro to read the L2 Cache Type Register (Example 30 on page 41).

Note, when any L2 cache lines are locked, these are not cleaned, invalidated or modified, and the contents remains locked and unaltered. To ensure that there are no valid lines and no modified data in the L2 cache, it is necessary to globally unlock all lines in the L2 cache prior to starting this operation.

Also note, that a memory barrier operation is used before and after the code to ensure that all L2 memory operations in program order are completed before the global clean/invalidate begins, and that none occur until all lines have been cleaned and invalidated.

Although this operation is used to ensure lines in the L2 cache is marked invalid and all modified lines ware cleaned out, when the code in the example is in a page marked outer cacheable, the lines containing the code is in the L2 cache at the completion of this operation.



Example 35. Load and Lock Lines in the L2 Cache

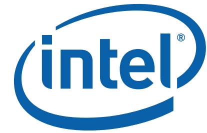
```
@ r1 contains the virtual address of the region of memory to lock in L2 cache
@ r0 is the number of 32-byte lines to lock.
@ In this example 16 cache lines of data are fetched and locked in the L2 cache
@ MMU is enabled prior to this code.

    mov r0, #16
LOOP1:
    mcr p15, 1, r1, c9, c5, 0  @ fetch the data at addr r1, and lock
                               @ it in the l2 cache
    add r1, r1, #32            @ increment the address in r1 to
                               @ the next cache line

    subs r0, r0, #1            @ Decrement loop count
    bne LOOP1
```

This code example performs a fetch and cache line fill of the data in the region specified, and then locks all the lines which are filled with the data.

By writing to co-processor 15, register 9, a fetch of a modified virtual address is performed, and the cache line is locked with the data retrieved. Unlike the operation of the L1 caches, when the data is already present in the L2 cache its lock bit is simply set, due to the fact that any way of the L2 cache sets are locked.

Also note, that unlike the L1 data cache, the lock operation is targeted at the line being loaded, as opposed to putting the cache in a 'lock mode' and having all fills locked while the mode is on. It is because of these differences that a memory barrier is not necessary during the course of this operation.

After this operation is complete, the data from the memory region specified in r1 is present and locked in the L2 cache, and generates a cache 'hit' when accessed.

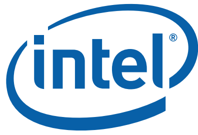


Example 36. Create and Destroy L2 Cache RAM

```
@ These macros are used to create and destroy a section of L2
@ Cache RAM. The address is passed in as register Ra and the number of
@ 32-byte lines is passed in as register Rn for both macros.
@ The L2 Cache RAM is destroyed using the L2 Invalidate by MVA Operation,
@ which also ensures the line is unlocked.

.macro CREATEL2RAM, Ra, Rn
   1:
   mcr p15, 1, \Ra, c9, c5, 2 @ Allocate & Lock L2 Line
   add \Ra, \Ra, #32          @ Move to next cache line
   subs \Rn, \Rn, #1          @ Decrement loop count
   bne 1b
.endm

.macro DESTROYL2RAM, Ra, Rn
   1:
   mcr p15, 1, \Ra, c7, c7, 1   @ Invalidate L2 line
   add \Ra, \Ra, #32            @ Move to next cache line
   subs \Rn, \Rn, #1            @ Decrement loop count
   bne 1b
.endm

@ Note that the contents of the newly created L2 RAM are unpredictable, and
@ The caller ensures it initializes the line with data before
@ attempting to read or utilize the data.
```

This sequence includes macros that create and destroy L2 Cache RAM by locking a series of lines in the L2 cache, and then unlocking and invalidating these.

The lines corresponding to the address specified in Ra are allocated and locked by writing to co-processor 15, register 9. No fetch from main memory is performed, so the data present in the lines after these are allocated is unpredictable. Since the locked lines are not evicted from the L2 cache, the contents of the lines are not written out, unless these are explicitly cleaned or unlocked. Since the lines are locked, these are not evicted from the L2 cache and always produce a cache 'hit' when read from or written to. By using an address space that has valid page table entries, but does not correspond to any physical memory, the region effectively becomes L2 RAM, which is read and written to - much faster than when it required accessing main memory for reads and writes.



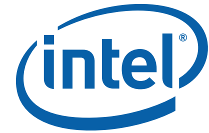
Example 37. Globally Unlock the L2 Cache

```
mov r1, #0                    @ The data in R1 is to be zero
mcr p15, 1, r1, c9, c5, 1     @ unlock all L2 cache lines
                              @ data in R1 is ignored
```

This operation uses co-processor 15, register 9 to globally unlock all lines in the L2 cache. Any lines in the L2 cache with the lock bit set has it unset, but the contents of the L2 cache is not altered. After this operation completes, any new fetch of data potentially evicts any lines in the L2 cache, and the memory contents are written out when these had been modified.

This operation is necessary when it was desired to globally clean and invalidate the L2 cache when there are cache lines locked. When the global cache clean and invalidate sequence (by set and way) encounters a locked line, the line is not cleaned or unlocked, so it is necessary to issue a global unlock instruction prior to cleaning to ensure all modified data in the L2 cache was written out.

Note, that while the global unlock of the L2 cache is a single operation, the global L2 cache clean and invalidate is a loop where each set/way is individually cleaned, as shown in Example 34 on page 43.



3.2.7 Branch Target Buffer (BTB)

The 3rd generation microarchitecture contains a 128-entry direct mapped branch target cache referred to as the branch target buffer or BTB. This cache holds the target address of a direct branch operation that is used as the next pipeline fetch address for the predictable branch. It is very desirable to enable the BTB. When a direct branch is predicted correctly by the BTB, the branch is taken without any penalties relating to flushing the instruction pipeline. With the BTB disabled, all direct predictable branches are assumed to be not taken, and when these are taken, program execution must stall while the pipeline is flushed and instructions from the branch target are fetched.

Example 38 on page 47 shows a simple code fragment that enables the BTB. The BTB is enabled at any time and is independent of the MMU. The addresses saved in the cache are virtual addresses so anytime page table mappings are changed the BTB also needs to be invalidated. In certain situations, such as a change to the Process ID (PID) register, or when invalidating the entire instruction cache, the BTB is also implicitly invalidated.

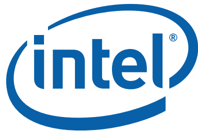
Example 39 on page 48 shows the instruction to invalidate the BTB. Additionally, when writing self-modifying code, be sure to invalidate the BTB before executing any of the newly created code.

Example 38. Enable the BTB

```
mrc, p15, 0, r0, cl, 0          @ Read the ARM control register
orr r0, r0, #0x800          @ Enable the Z (BTB) bit
mcr p15, 0, r0, c1, c0, 0    @ Write the modified value
```

The BTB on the 3rd generation microarchitecture is enabled by modifying the ARM control register to have the Z bit (BTB enable) turned on. When the BTB is enabled, direct branches are cached along with whether or not these are taken is recorded. When the same branch is encountered again it is predicted and when correct, no pipeline flush is necessary when the branch actually occurs.

Although the BTB is mainly transparent in operation, at certain times it is necessary to explicitly perform operations on it to ensure correctness. These occur when data in the L1 instruction cache changes or is invalidated and the data in the BTB is not kept in synchronization. When that situation occured, the processor behaves unpredictably when stale branch addresses were encountered.



Example 39 shows how to specifically invalidate only the BTB using co-processor 15, register 7.

Example 39. Invalidate the BTB

```
mov r0, #0                     @ The data in R0 is to be zero
mcr p15, 0, r0, c7, c5, 6      @ Invalidate the BTB
```

Although the operation above is used any time to invalidate the entire BTB, there are certain times when it is implicitly invalidated as well. Whenever the Process ID (PID) register is changed, it is affect the mapping between virtual addresses and the addresses of physical instructions, so the BTB is invalided. Additionally, whenever the entire instruction cache is invalidated, the BTB is also invalidated, because there is no case where the BTBs contents need to remain intact after the entire instruction cache has been invalidated.

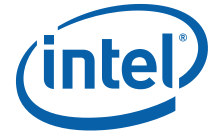
However, for operations like single instruction cache line invalidation by MVA, and for changes being made to the page table mappings of instruction memory addresses, the BTB is not implicitly invalidated, and it is necessary to execute the BTB invalidate operation in order to ensure no stale branches are cached in the BTB.

Example 40. Disable the BTB

```
mrc p15, 0, r0, cl, c0, 0     @ Read the ARM control register
bic r0, r0, #0x800            @ Disable the Z (BTB) bit
mcr p15, 0, r0, c1, c0, 0     @ Write the modified value
mov r0, #0                    @ The data in R0 is to be zero
mcr p15, 0, r0, c7, c5, 6     @ Invalidate the BTB
```

This code disables the BTB by modifying the ARM control register to have the Z bit (BTB enable) turned off.

Note, that the BTB is then invalidated as shown in the previous example after it has been turned off. Turning off the BTB first then invalidating it ensures that no new branch prediction data ise cached during the execution of the invalidation operation. It is also possible to disable the BTB and instruction cache simultaneously and then invalidate both simultaneously by issuing the L1 instruction cache and BTB invalidate operation as shown in Example 19 on page 30.



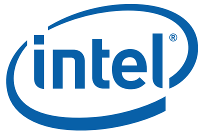
3.2.8 Self-modifying Code

This section describes the operations that system software invokes, in order to correctly execute code that has been modified or created by another running process. This typically involves JIT (Just In Time) compilers which are becoming increasingly prevalent with the advent of Java, .NET, and dynamic object code translation technology. It is important that after any creation of code in a buffer has taken place, the correct operations are performed, so that the 3rd generation microarchitecture view of the instructions is not stale or incorrect.

Since creating a buffer of translated or compiled instructions likely involves a lot of writes to a localized region of memory, it is desirable to pre-allocate the region before it is written, so the data is written once in a burst to main memory. See "L1 Data Cache Line Allocation" on page 40 for more information on the line allocation operation. After the code has been written to memory, some or all of it is present in the L1 data cache. Since the L1 instruction cache is not coherent with the data cache, simply writing the code and immediately branching to it generates unpredictable results. For this reason, a series of steps must be completed to ensure the code is first cleaned out of the data cache, then correctly fetched back into the instruction cache, and finally that any residual data from stale instructions are cleared out of the 3rd generation microarchitecture. An example code sequence which prepares the microarchitecture to execute a buffer of just-in-time compiled code is shown in Example 41 on page 49.

Example 41. Branching to Runtime Generated Code

```
@ This code sequence branches to a buffer of instructions
@ that has just been written by an application sach as a JIT compiler.

@ r1 contains the address of the code buffer
@ r2 contains the number of bytes in the code buffer

    add r2, r2, #31              @ must be atleast 1 line
    mov r0, r2, lsr #5           @ get number of lines, put in r0

@ First, clean the d-cache and invalidate the i-cache lines with this region
@ (to ensure the instruction cache sees the recent writes)
    mov r3, r0                   @ put # of lines in r3
    mov r4, r1                   @ put address to clean in r4

CLEANLOOP:
    mcr p15, 0, r4, c7, c10, 1   @ Write out the line when dirty in the d-cache
    mcr p15, 0, r4, c7, c5, 1    @ invalidate address r4 in I-cache
    add r4, r4, #32              @ increment addr to clean
    subs r3, r3, #1              @ decrement loop counter
    bne CLEANLOOP                @ continue until done

@ Any I-cache lines at this location have been invalidated. But when the BTB
@ contained branch information it is still there and cause
@ unpredictable execution of the new code. Invalidate the BTB.
    mcr p15, 0, r3, c7, c5, 6    @ Invalidate all entries in the BTB
                                 @ contents of r3 ignored

@ Now before executing the code, ensure that any memory operations
@ which were invoked from the previous clean have completed so the instruction
@ fetch gets the correct data.
    mcr p15, 0, r3, c7, c10, 4   @ DWB, data in r3 is ignored

@ Prefetch flush to clear the pipeline of any fetched instructions

    mcr p15, 0, r3, c7, c5, 4    @ PF, data in r3 is ignored

@ Now branch directly to the address at the start of the code buffer.
    mov pc, r1
```

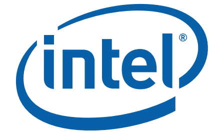


In this example, the 3rd generation microarchitecture is set up to execute code that has just been written to memory. Since the microarchitecture L1 instruction and data cache are not coherent (these are completely separate), it is important to follow certain steps to ensure that the instructions which have been created or modified are executed correctly.

The memory region that contains the code, which was created or modified, must first be cleaned out of the L1 data cache, so the instruction cache fetch loads it when the 3rd generation microarchitecture begins to execute the new code. Additionally, any stale instructions present in the microarchitecture pipeline must be flushed out.

Since instruction information is present in the L1 instruction cache, the BTB, and the pipeline, all of these must be cleaned in order to ensure these do not interfere with the execution of the new code. A memory barrier operation is used to ensure the memory operations invoked by this procedure have completed before the compiled code begins to execute.

The operations used in this example are explained in more detail in the following sections: Section 3.2, "Cache Management", and Section 3.4, "Memory Barriers".



3.3 Aborts

Aborts in the 3rd generation microarchitecture are handled as part of the ARM standard exception architecture. Prefetch aborts are lower in priority than the external interrupts, FIQ and IRQ, while data aborts are higher priority. When an abort occurs, the microarchitecture branches to the appropriate abort exception vector, where it is then read the FSR and FAR registers to determine the type and cause. In the case of multiple aborts, the highest priority abort is visible first. A table of aborts, the attributes, and the priorities are found in the Programming Model chapter *3rd Generation Intel XScale® Microarchitecture Developer's Manual*.

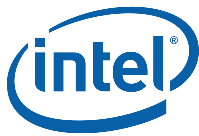
3.3.1 Prefetch Aborts

The memory system detects a memory abort on an instruction fetch and flag the fetched instruction(s) as invalid to the 3rd generation microarchitecture. A prefetch abort occurs when the microarchitecture attempts to execute an instruction that was flagged as invalid. There are several different causes for prefetch aborts. The memory management unit (MMU) reports an abort because permissions set in the page table or in the domain access control register for the memory region do not allow the target memory to be accessed. Another cause is an external error occurring outside the microarchitecture during the memory access. Also, when there was a parity error on the cache line being accessed, a prefetch abort occurs.

Prefetch aborts are precise exceptions, which means that the abort occurs before any subsequent instructions execute. Thus the architectural state of the 3rd generation microarchitecture is consistent and reflects what was happening on the microarchitecture up to the aborting instruction. The R14_ABORT register (which is the link register as viewed from the abort handler) contains the address of the instruction whose fetch caused the abort plus 4 bytes. Because the exact execution address is known, it is possible for the system to recover from a prefetch abort error by either a long-jump back to a known good state, or in the case that the code being executed needed to be paged in, paging it in and returning to the running program. In the second case, the code running that caused the abort is unaware it happened and continue executing as though it was always loaded.

Example 42. Recovering From an L1 Instruction Cache Parity Error

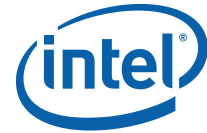
```
@ Prefetch abort handler
@ First Check FSR - for FS[10,3:0] = 0b11000 (IC parity)0x408

    mrc p15, 0, r0, c5, c0, 0   @ Read FSR
    tst r0, #0x0400             @ Check FSR for IC parity error
    tstne r0, #0x08
    bne icache_parity           @ branch to IC parity code
    @ otherwise it was not IC parity error
@ ...

icache_parity:
    sub r0, r14, #4             @ get address to invalidate in IC.
    bic r0, r0, #0x1F           @ Get cache-line that caused abort
    mcr p15, 0, r0, c7, c5, 1   @ Invalidate & unlock line that caused abort
    mcr p15, 0, r0, c7, c5, 6   @ Invalidate entire BTB
    mcr p15, 0, r0, c7, c5, 4   @ Prefetch flush to clear any invalid
                                @ instructions out of the pipline
    subs pc, r14, #4            @ Returns to the instruction that generated the
                                @ parity error
```



Example 42 shows some steps that system software takes in order to recover from an L1 instruction cache parity error. An L1 instruction cache parity error is a precise prefetch abort, and when it occurs, the 3rd generation microarchitecture branches to the prefetch abort handler. The prefetch abort handler is checked the Fault Status register in co-processor 15 to determine what kind of prefetch abort occurred. In the case of an L1 instruction cache parity error, the error is recovered from by invalidating the L1 instruction cache line that caused the abort. The address to invalidate is determined by LR_ABORT-4 - this invalidates the entire cache line, aligned down to a cache line boundary.

While it is also reasonable to use the global IC invalidate operation, locked lines are not unlocked in this case. Invalidating by modified virtual address (MVA) both unlocks and invalidates the target cache line. In either case, the BTB is also invalidated since it contains branch information related to code in the instruction cache. After the L1 instruction cache line invalidation has taken effect, the instruction that encountered the IC parity error is executed by subtracting 4 bytes from the abort link register and placing it in the program counter register. This procedure applies to the instruction cache only, because it is known that the contents of the line that was lost was not modified (dirty), and therefore it is possible to simply invalidate the cache without any loss of dirty data.

Although some prefetch aborts occur due to errors retrieving the instruction memory such as parity errors, an instruction that accesses an invalid co-processor also causes a precise abort. In this case an undefined instruction exception is generated. Using this exception, it is possible to emulate a co-processor that does not exist. In that case, the undefined instruction exception handler reads and parses the instruction that caused the undefined exception; perform whatever emulation is necessary, including update of any destination registers; then return back to the code at the next instruction. The calling application is unaware that the results came from an emulated co-processor and not an actual co-processor in hardware.



3.3.2 Data Aborts

Data aborts occur when the 3rd generation microarchitecture encounters an exception while attempting to access data memory. All data aborts fall into two categories:

Precise: On a precise abort software is determined precisely which instruction caused the abort by examining R14_ABORT. When the data abort exception handler is entered, the 3rd generation microarchitecture places the address of the instruction that caused the abort plus 8 bytes into R14_ABORT.

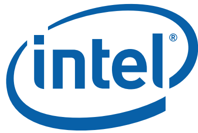
When a precise data abort occurs, it is possible for the system to recover and continue running when desired. This is because the exact address and state of the microarchitecture is known when the abort occurs, and system software is used this to long-jump to a known good execution location or, when the abort is due to an access of paged-out memory, load the page and continue execution.

Imprecise: For imprecise data aborts software cannot always determine the exact instruction that caused the abort and in some cases, there is not a specific instruction associated with the abort. In this case, the 3rd generation microarchitecture places the address of the next instruction to execute in the program flow (at the time the abort is generated) plus 4 bytes in R14_ABORT.

When an imprecise data abort occurs, it is considered unrecoverable. This is because the state of the 3rd generation microarchitecture at the time of the abort is unknown, and the program cannot continue execution because it is unsure which instructions have executed since the memory operation that failed. It is possible that certain instructions, which depended on a data operation, executed with unpredictable data. In this case the imprecise abort serves generally to alert the system to the error and to allow it to halt execution as soon as the integrity of the system is in doubt. It is possible to issue a soft reset or to somehow otherwise reset and then continue operation after an imprecise data abort. The exact action to take depends on what the products specification requires with respect to system integrity.

While certain data aborts occur due to errors such as data cache parity errors or errors on an external memory bus, some precise data aborts are used during the normal course of operation of the system. Since a precise data abort occurs on a virtual address translation when the page table entry permission bits are checked, it is possible for system software to mark pages which are “paged out” with the correct permission bits, then handle the exception when that page is accessed, by making the page available. Because the abort is precise and program execution stops on the instruction accessing the page which was paged out, the program continues when it is available without knowledge that the paging operation took place. This mechanism allows an operating system to support virtual memory.

In the case of certain bus transactions an external data abort is part of the system configuration and initialization procedure. For example, in order to discover devices on a PCI bus that are accessed by the 3rd generation microarchitecture through a bridge, it is necessary to issue a read operation to addresses that never return data, and use the result to determine what devices exist in the system.

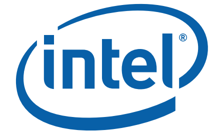


Example 43. Using An External Abort Handler During Device Configuration

```
@ This is the data abort handler, this code executes
@ When the read of a non-existent device occurs.
.dabrt_handler
    mov r6, #1          @ r6 indicates whether or not an error occured.

@ Reset interrupt that is causing external data abort (write to
@ external registers to clear master abort

    subs pc, lr, #4     @ return to instruction after the read


@ This code reads a memory mapped region of a device register.
@ The address of the register to read is in r0.
@ It uses the result (error or not) to determine when a device is present.
read_device:

    mov r6, #0          @ r6 indicates whether or not an error occured.

    mrs r3, cpsr        @ disable interrupts
    orr r3, r3, #0xc0
    msr cpsr_c, r3

    ldr r1, [r0]        @ Read from memory mapped address
    mov r1, r1
    cmp r6, #0          @ Check to see when r6 is 1, meaning a data abort occured
    mvnne r1, #0        @ Set read value to 0xFFFFFFFF when error

    mrs r3, cpsr        @ restore interrupts
    bic r3, r3, #0xc0
    msr cpsr_c, r3

    mov r0, r1          @ Return the read value, or 0xFFFF_FFFF when not present
```

This code sequence illustrates how an external data abort, arising from a read of memory mapped device space, is used during system initialization. This code generates a read of an address space, where a PCI configuration space header register exists, is used to determine when a device is present in the slot. To higher level code calling the PCI access functions, a return value of binary 1s siginifies that nothing exists. However at a lower level, the processor in fact receives an external data abort when the PCI bridge experiences a time-out (master abort), while attempting to read the address.

It is important that interrupts are disabled. When these were not, an interrupt occurs at any time that cause the 3rd generation microarchitecture to start executing the interrupt handler code. When an interrupt occured just after the read of the device and an external data abort came in during that time, it is impossible to determine the state of the system when the data abort occured. With interrupts disabled the pipeline is stalled, waiting for the data to be read into r1. When a data abort occurs, r1 contains unpredictable data. The code checks r6 (which needs to be set by the abort handler) to see when a data abort occured. When it did occur, it sets r1 to contain all 1s, signifying no device was present when the register space was read.



3.4 Memory Barriers

Often system software wants to ensure that a memory operation or a set of memory operations have achieved global observation. Because of the features which help to isolate the 3rd generation microarchitecture from the latency of the external busses, a memory operation is not necessarily globally observed when the instruction that initiated it has retired (See Figure 1 on page 55). For this reason, several instructions have been implemented, which allow the system to wait for a set of memory operations to be globally observed before continuing execution of other memory operations.

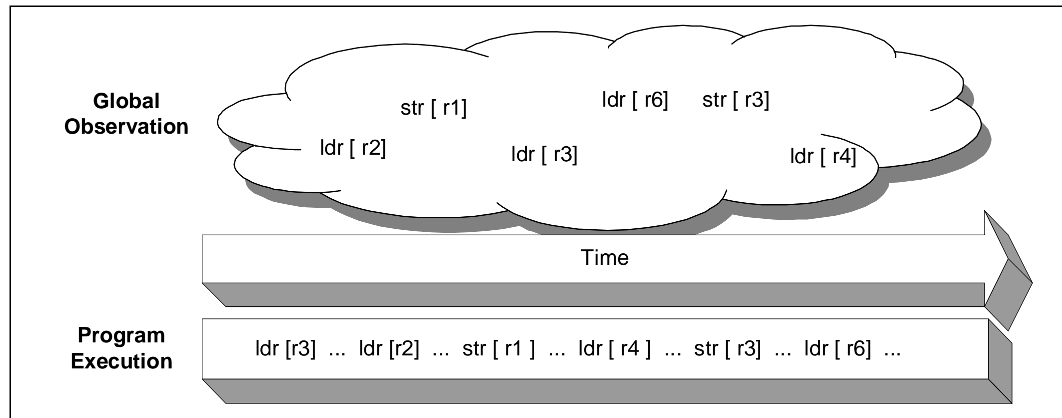
The three instructions are:

- Data Memory Barrier (DMB)
- Data Write Barrier (DWB)
- Prefetch Flush (PF)

These instructions are useful and necessary when synchronization of memory needs to be ensured, either between multiple 3rd generation microarchitectures or within a single processor system.

For example, when an application wants to ensure that a series of writes has been globally observed, it most likely cleans the cache lines that contain the writeback data. Although this ensures that the data is no longer only in the cache, it takes extra cycles before the data is written to the device or to memory as it leaves the memory pipeline. Using a memory barrier, ensures that the 3rd generation microarchitecture waits until the data has been globally observed before continuing program execution.

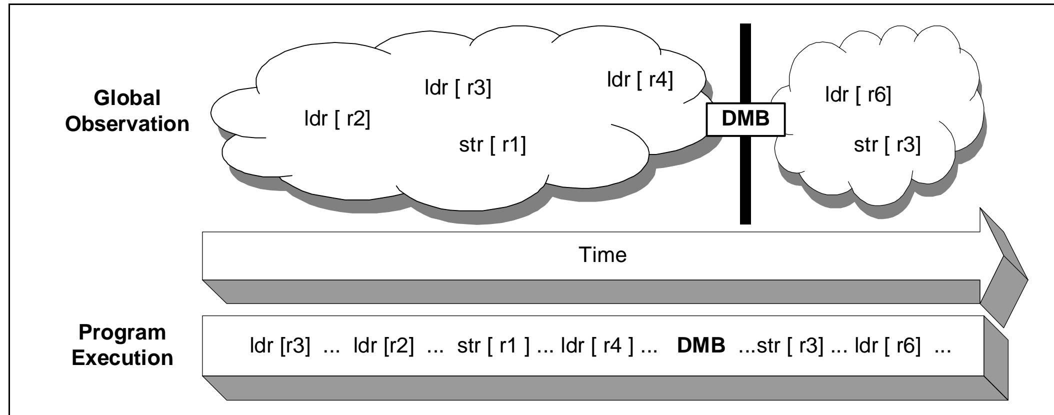
Figure 1. Global Observation vs. Program Execution



The Data Memory Barrier instruction ensures that no memory operations, that occur in program order after it is globally observed, before any memory operations that were executed before it (See Figure 2 on page 56). Other instructions execute during this time when these do not issue a memory operation.

Also, it is important to note that the DMB instruction does not affect memory operations relating to 'non-explicit' accesses, such as instruction fetches and page table walks. Those memory operations occur before, during, and after the invocation of the DMB. Refer to Example 44 on page 56 to see how a DMB operation is invoked.

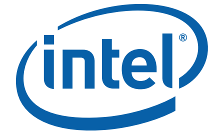
Figure 2. Data Memory Barrier Operation



Example 44. Data Memory Barrier (DMB)

```
mov r0, #0                  @ The data in R0 is to be zero
mcr p15, 0, r0, c7, c10, 5  @ Data Memory Barrier, Data in R0 is ignored
```

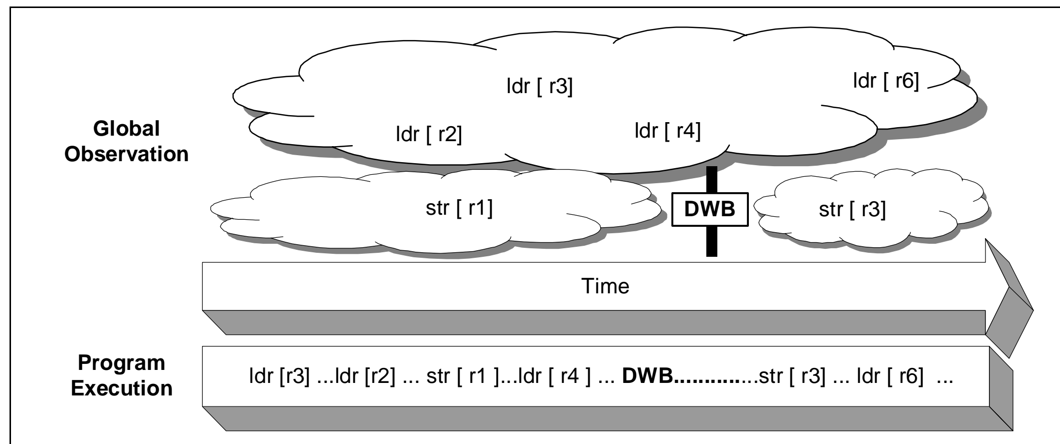
As seen above, the Data Memory Barrier instruction is initiated by performing a write to co-processor 15, register 7. The contents of the register written are ignored.



The Data Write Barrier (DWB) operation creates a memory write barrier. This means that all memory write operations, that were issued before the DWB, achieves global observation before write operations that occur after the DWB (See Figure 3 on page 57).

Similar to the DMB instruction, DWB does not provide any barrier for non-memory access instructions or instructions that relate to non-explicit memory accesses such as instruction fetches and table walks. The DWB operation does however additionally specify that no other instructions after the DWB executes, regardless of whether or not these access memory, until the DWB instruction has retired.

Figure 3. Data Write Barrier Operation

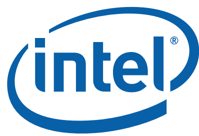


Example 45. Data Write Barrier (DWB)

```
mov r0, #0                    @ The data in R0 is to be zero
mcr p15, 0, r0, c7, c10, 4    @ Data Write Barrier, Data in R0 is ignored
```

The Data Write Barrier instruction is initiated by performing a write to co-processor 15, register 7. The contents of the register written are ignored.

Both the DMB and DWB instructions provide synchronization with respect to data memory operations. However, neither of these have an effect on instruction-related memory operations such as instruction fetches and address translation table walks. In order to complement these instructions, a Prefetch Flush instruction exists, which ensures that all memory operations relating to instruction execution have completed. The Prefetch Flush (PF) instruction ensures that all table walks and instruction fetches to cache are complete before it retires. Additionally it ensures that the next instruction executed is fetched from cache or memory (it is not in the pipeline) because the pipeline was flushed. See Example 46 on page 58 for the instruction which causes a prefetch flush.



Example 46. Prefetch Flush (PF)

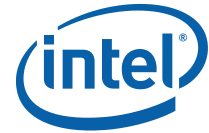
```
mov r0, #0                  @ The data in R0 is to be zero
mcr p15, 0, r0, c7, c5, 4   @ Prefetch Flush, data in R0 is ignored
```

In Example 46, the Prefetch Flush instruction is initiated by performing a write to co-processor 15, register 7. The contents of the register written are ignored. Once this instruction has retired, all outstanding memory operations relating to instruction execution (instruction memory fetches, table walks) are completed. The next instruction executed is fetched from either the instruction cache or external memory, because the pipeline was flushed. The Prefetch Flush instruction is also useful in cases where the system software is modifying instruction data it is about to execute. For more information on self-modifying code, refer to Section 3.2.8, “Self-modifying Code” on page 49.

Although these operations are useful for ensuring synchronization within one 3rd generation microarchitecture or multiple microarchitectures, these do not provide automatic synchronization with all components in a system. This is because, even though a memory operation has left the microarchitecture, it is not possible to automatically know when the operation has been written to an external agent on a bus (possibly several busses away) and when it has taken effect.

One example of this is an interrupt handler running on a system with an external interrupt controller. When the interrupt initially is handled, 3rd generation microarchitecture external interrupts are usually disabled through the cpsr register. After this, the interrupt being serviced is masked at the external interrupt controller, and the CPSR register is restored so the system handles other interrupts while processing the first one. The memory operations invoked to mask the external interrupt most likely occur to an uncached region of memory, and a memory barrier instruction, as described above, is utilized to ensure that the mask write operation had been globally observed. However, this sequence does not ensure that the interrupt had been masked when the next operation, which is to re-enable interrupts to the microarchitecture, was performed.

This is because even though the write had completely left the 3rd generation microarchitecture, it is still many clock cycles before it has its effect on the external interrupt controller. Since the next operation is a write to a register that is inside the microarchitecture, it is conceivable and likely that the external interrupt is still asserted when interrupts were re-enabled; even though it was to be masked externally at that time. Since the microarchitecture is level-sensitive to interrupts, it triggers and this causes a condition where an interrupt was detected for a short while, but none are found when the handler checked for unmasked interrupts. For this reason, system software has to ensure the mask write operation had taken effect by utilizing some sort of external synchronization method, such as stalling on a read of the same memory location just written.



4.0 Performance Monitoring (PMU)

The 3rd generation microarchitecture hardware provides four 32-bit performance counters that allow up to four unique events to be monitored simultaneously. In addition, the microarchitecture implements a 32-bit clock counter that is used in conjunction with the performance counters; its main purpose is to count the number of microarchitecture clock cycles which is useful in measuring total execution time.

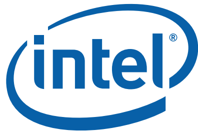
Example 47 on page 59 shows how to enable the PMU and start the clock count register (CCNT) running. Example 48 on page 60 shows how to modify an event counter after the PMU has been enabled.

Example 47. Initialize the PMU and Enable CCNT

```
    mov r0, #0x0              @ Disable all counters
    mcr p14, 0, r0, c0, c1, 0 @ Write the PMNC value
    mvn r0, #0x0
    mcr p14, 0, r0, c8, c1, 0 @ Disable events in EVTSEL register

    mov r0, #0x6              @ Reset performance and clock counters
    mcr p14, 0, r0, c0, c1, 0 @ Write the PMNC value
    mov r0, #0x11              @ disable event counters and enable clock counter
    mcr p14, 0, r0, c0, c1, 0 @ Write the PMNC value
```

Example 47 describes how the PMU is initialized and the clock counter is enabled on the 3rd generation microarchitecture. In this example, no event counters are enabled and the events being tracked are set to disabled in the EVTSEL register. Setting bits 2 and 1 of the PMNC register resets the clock counter and all performance counters to '0x0'. After this code completes, the clock counter is enabled and the CCNT register increments once for every microarchitecture clock cycle.

Used by itself it is a source of highly accurate timing information. When used with the performance event counters it is used to calculate various system performance statistics such as cycles per instruction (CPI).



Example 48. Enable PMU Event Counter 0

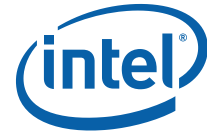
```
    mrc p14, 0, r0, c0, c1, 0   @ Read PMNC register
    and r1, r0, #0x0F           @ Clear unpredictable bits
    orr r0, r1,#0x10            @ Disable event counters
    mcr p14, 0, r0, c0, c1, 0   @ Write PMNC register

    mrc p14, 0, r0, c8, c1, 0   @ Read EVTSEL register
    bic r0, r0, #0xFF           @ Clear event 0
    orr r0, r0,#0x07            @ Insert new event 0
    mcr p14, 0, r0, c8, c1, 0   @ Write EVTSEL register

    mov r0, #0x0
    mcr p14, 0, r0, c0, c2, 0   @ Clear event count 0

    mrc p14, 0, r0, c4, c1, 0   @ Read INTEN register
    orr r0, r0,#0x02            @ Enable event 0 interrupts
    mcr p14, 0, r0, c4, c1, 0   @ Write INTEN register

    orr r0, r1, #0x01           @ Ensure the PMU is enabled
    mcr p15, 0, r0, c0, c1, 0   @ Write the PMNC value
```

This code configures the PMU on the 3rd generation microarchitecture to count certain events as well as generate an interrupt when the counters for event 0 overflow.

Initially all event counters are disabled by setting bit 4 (the M bit) of the PMNC register. Once the counters have been disabled, it is then possible to change the events being counted by writing to the EVTSEL register.

Note, that it is important the counters are stopped, either by the PMNC.M bit or the PMNC.E bit.

Modifying the EVTSEL register while the event counters are running produce unpredictable results. In this case, event0 is changed to count event 0x7, the number of instructions executed. The counter for event0 is then initialized to zero, and the INTEN register is written so that interrupts are generated whenever the event0 counter overflows 32-bits. When an interrupt occurs, it is possible for the handler to record elsewhere the number of overflows (2^{32} events of that type occured) and later use that information to calculate the total number of events that occurred over the sampling period. After the registers are configured to correctly count event0, the PMNC PMU enable bit (bit E) is turned on to ensure that the PMU is activated, and the write back to the PMNC register enables the counters.



Example 49. Handling a PMU Interrupt

```
@ Assume that performance counting interrupts are the only IRQ in the system
external_irq_routine:
    mrc p14, 0, r1, c0,c1,0  @ read the PMNC register
    bic r2, r1,#1            @ clear enable bit, preserve other bits in PMNC
    mcr p14, 0,r2, c0, c1,0  @ disable counting
    mrc p14, 0,r3, c5, c1,0  @ read FLAG register
    mrc p14, 0,r4, c1, c1,0  @ read CCNT register
    mrc p14, 0,r5, c0, c2,0  @ read PMN0 register
    mrc p14, 0,r6, c1, c2,0  @ read PMN1 register

    @ <process the results>

    mrc p14, 0, r1, c0,c1,0  @ read the PMNC register
    orr r2, r1, #1           @ set the enable bit, preserve other bits in PMNC
    mcr p14, 0, r2, c0, c1,0 @ re-enable counting
    subs pc, r14, #4         @ return from interrupt
```

This code is an example of an interrupt handler that is called whenever a counter overflows. In this case, it is assumed that the only interrupt that is triggered and IRQ is the PMU. Depending on the product, an external interrupt status register needs to be read to determine what caused the interrupt before determining it was the PMU.

In this example an interrupt triggered by an overflow of either the CCNT register or one of the event counter registers causes the handler to be called. The handler immediately disables all PMU counters by clearing bit 0 (the E bit) of the PMNC register. Once the PMU has been stopped, the overflow flag register, event counter and clock count registers are read to determine what overflowed and what the total values of these are.

It is also possible to simply record which counter overflowed without recording the values of the registers. Then when the sampling period ended the value of a given event counter is *(*2^{32} ** number_of_overlfows + current_register_value)*. Once any processing has taken place, the event counters are re-enabled by setting the E bit in the PMU and returning from the interrupt handler.

Example 50. Computing PMU Results

```
@ Assume CCNT overflow

CCNT = 0x00000020 @ Overflowed and continued counting
Number of instructions executed = PMN0 = 0x6AAAAAAA
Number of instruction cache miss requests = PMN1 = 0x05555555
Instruction Cache miss-rate = 100 * PMN1/PMN0 = 5%
CPI = (CCNT + 2^32)/Number of instructions executed = 2.4 cycles/instruction
```

5.0 Interrupts

All 3rd generation microarchitecture external interrupts are handled through either IRQ or FIQ vectors. Internal interrupts, such as those generated by the PMU, are steered to either IRQ or FIQ as well. This section shows some microarchitecture examples pertaining to interrupts and interrupt handling.

It is necessary to disable interrupts during certain system operations. In Example 22 on page 33, the cache is placed into 'lock mode' where any data cache fetch also becomes locked in the cache. When an interrupt occured while the cache was in this mode and the handler made some memory references, unexpected lines become locked in the cache. With the interrupt disabled it ensures that any external interrupt that occurs during the lock operation is not taken until interrupts are re-enabled.

Example 51. Disable Interrupts

```
@ disable interrupts
    mrs r3, cpsr            @ copy CPSR register to r3
    orr r3, r3, #0xC0         @ Set F and I bits (to disable FIQ and IRQ)
    msr cpsr_c, r3          @ Copy back to CPSR
```

Interrupts are disabled in the 3rd generation microarchitecture by setting the I bit and the F bit in the CPSR register. In this case both the I and F bit are set, however it is only necessary to mask one of these interrupts, depending on whether or not IRQ and FIQ have the potential to trigger. When these bits are set, and an interrupt occurs (either via IRQ or FIQ), the microarchitecture does not jump to the interrupt service vector. This allows code to continue executing despite possible external interrupts from affecting the execution address. Interrupts are often disabled during critical system initialization and configuration code sequences, to ensure that an interrupt is not serviced at a time when the system state is being modified.

Example 52. Enable Interrupts

```
@ restore interrupts
    mrs r3, cpsr           @ Copy the CPSR to r3
    bic r3, r3, #0xC0           @ Clear the F and I bits (to enable FIQ and IRQ)
    msr cpsr_c, r3         @ Copy r3 back to the CPSR
```

Interrupts are enabled by clearing the I and F bits in the CPSR register on the 3rd generation microarchitecture.

Once the bits are clear, any IRQ or FIQ signal asserted to the 3rd generation microarchitecture, either from the PMU or an external source, causes the microarchitecture to branch to the appropriate interrupt handler vector and begin executing.