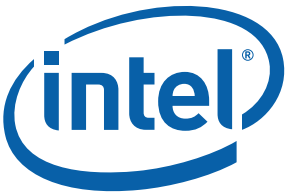


# 3rd Generation Intel XScale® Microarchitecture

## Developer's Manual

*May 2007*

Order Number: 316283-002US

# Contents

## Figures

# Tables

# Revision History

| Date | Revision | Description |
|---|---|---|
| May 2007 | 002 | Table of Content corrections.<br>Miscellaneous typographical errors.<br>Reformatting of code examples. |
| April 2007 | 001 | Initial release. |



# 1.0 Introduction

## 1.1 About This Document

This document is the authoritative and definitive reference for the external architecture of the 3rd generation Intel XScale® microarchitecture (3rd generation microarchitecture or 3rd generation), which is ARM* architecture compliant.

Intel Corporation assumes no responsibility for any errors which appears in this document nor does it make a commitment to update the information contained herein.

Intel retains the right to make changes to these specifications at any time, without notice. In particular, descriptions of features, timings, and pin-outs does not imply a commitment to implement.

### 1.1.1 How to Read This Document

It is necessary to be familiar with the *ARM Architecture Version 5TE Specification* (ARMv5TE) in order to understand some aspects of this document.

Refer to Section 1.3.2, "Terminology and Acronyms" on page 25 for a description of some of the terms used throughout this document.

### 1.1.2 Other Relevant Documents

- *ARM Architecture Version 5TE Specification* Document Number: ARM DDI 0100E
  This document describes Version 5TE of the ARM Architecture, which includes the *Thumb ISA* and ARM *Enhanced DSP Extension*. (ISBN 0 201 737191)
- *3rd Generation Intel XScale® Microarchitecture Software Design Guide*
  This document describes recommended code sequences useful for low level software developers. These sequences ensure proper behavior of the microarchitecture when using various 3rd generation features.
- *3rd Generation Intel XScale® Microarchitecture Software Debug Guide*
  This document supplements the Software Debug Chapter of the *3rd Generation Intel XScale® Microarchitecture Developer's Manual*.

## 1.2 High-Level Overview of 3rd Generation Microarchitecture

3rd generation microarchitecture is an ARMv5TE compliant microarchitecture. It has been designed for high-performance and low-power. The microarchitecture is not intended to be delivered as a stand alone product, but as a building block for an ASSP (Application Specific Standard Product) with embedded markets such as wireless, networking, storage, remote access servers, etc.

3rd generation microarchitecture is an evolutionary enhancement to the previous generations. Application code targeting previous generations runs without any changes on 3rd generation microarchitecture. System code requires minimal changes (to deal with a different cache organization for example). For information on the differences between 3rd generation microarchitecture and previous generations refer to Appendix B, "Microarchitecture Compatibility Guide".

### 1.2.1 ARM Compatibility

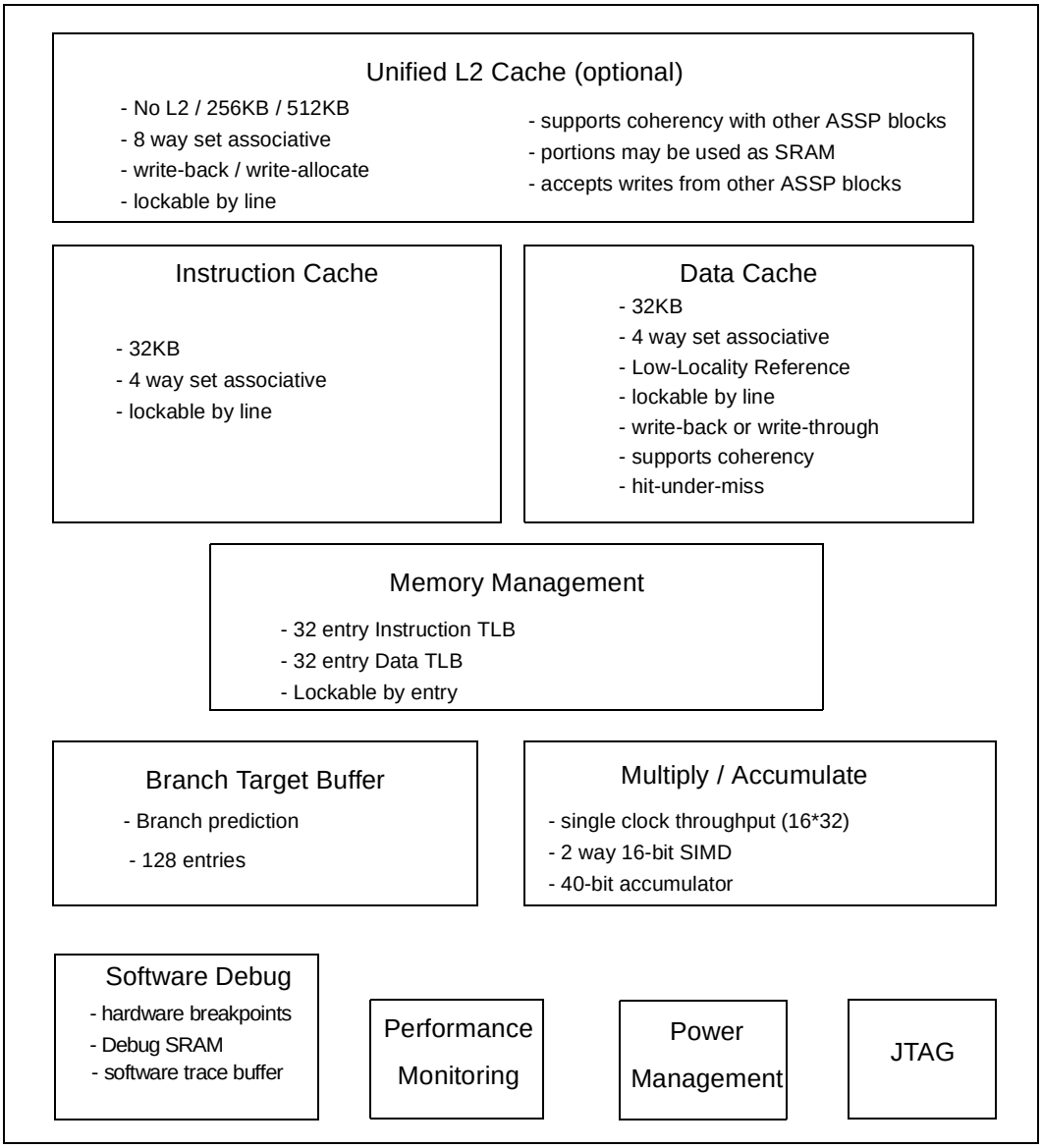
3rd generation microarchitecture implements the integer instruction set architecture of ARMv5, but does not provide hardware support of the floating point instructions (VFP).

3rd generation microarchitecture implements the Thumb instruction set (ARMv5T) and the Enhanced DSP Extension (ARMv5E).

### 1.2.2 Features

Figure 1 shows the major functional blocks of 3rd generation microarchitecture. The following sections give a brief, high-level overview of these blocks and other features.

**Figure 1. 3rd Generation Microarchitecture Features**



#### 1.2.2.1 Level-2 Cache

The optional L2 cache in 3rd generation microarchitecture provides the next level of memory hierarchy for the L1 instruction and data caches. ASSPs opts for no L2 cache, a 256KB L2 cache or a 512KB L2 cache.

The L2 cache, when present, is 8-way set associative, write-back, write-allocate. It allows software to create regions which act as SRAM -- these are not subject to the normal replacement of other cache regions. Also, the L2 allows other bus agents to write directly into the cache; this is a "push" capability.

See Chapter 8.0, "Level 2 Unified Cache (L2)" for more details.

#### 1.2.2.2 Memory Coherency

Software opts to have hardware coherency support on shared memory regions. This allows hardware to maintain coherency between data in the
3rd generation microarchitecture caches and main memory, ensuring that multiple agents in the system see the proper data values.

This facility is explained in detail in Chapter 9.0, "Cache Coherence".

#### 1.2.2.3 Multiply/Accumulate (MAC)

The MAC unit supports early termination of multiplies/accumulates and sustains a throughput of a MAC operation every cycle. Several architectural enhancements were made to the MAC to support media processing algorithms, including a 40-bit accumulator and support for 16-bit packed data.

See Section 2.3, "Extensions to ARM Architecture" for more details.

#### 1.2.2.4 Memory Management

3rd generation microarchitecture implements an enhanced version of the Memory Management Unit (MMU) Architecture specified in the *ARM Architecture Version 5TE Specification*. The MMU on 3rd generation microarchitecture implements two new page types to provide additional functionality, including support for a 36-bit physical address space.

In addition to address translation and memory protection, the MMU Architecture specifies shared memory and caching policies for the various caches. These policies are specified as page attributes and include:

- identifying a memory region as L1 and/or L2 cacheable / non-cacheable
- identifying a data region as low-locality reference (LLR)
- write-back or write-through L1 data caching
- enabling the write buffer to coalesce stores to external memory
- identifying a memory region as shared (enabling hardware coherency for that region).

Chapter 3.0, "Memory Management" discusses this in more detail.



### 1.2.2.5 Instruction Cache

3rd generation microarchitecture provides a 4-way set associative, 32 KB instruction cache with a cache line size of 32 bytes. All requests that "miss" the instruction cache generate a 32-byte read request to the next level of memory. A mechanism to lock critical code into the cache is also provided. The instruction cache uses the pseudo-LRU replacement algorithm (assuming all four ways of target set are unlocked).

Chapter 4.0, "Instruction Cache" discusses this in more detail.

### 1.2.2.6 Branch Target Buffer

3rd generation microarchitecture provides a Branch Target Buffer (BTB) to predict the outcome of branch type instructions. It provides storage for the target address of branch type instructions and predicts the next address to present to the instruction cache when the current instruction address is that of a branch.

The BTB holds 128 entries. See Chapter 5.0, "Branch Target Buffer" for more details.

### 1.2.2.7 Data Cache

3rd generation microarchitecture provides a 4-way set associative, 32 KB data cache, with a line size of 32 bytes. The data cache supports write-through or write-back caching and is controlled by page attributes defined in the MMU Architecture and by coprocessor 15.

3rd generation microarchitecture allows a portion of the data cache to be used for low-locality references (LLR). This features allows data from specified regions of memory to be isolated in one way of the data cache, eliminating replacement of critical data in the other ways of the same set.

A portion of the data cache is also used by applications as data RAM. Software places data structures or frequently used variables in this RAM.

Chapter 6.0, "Data Cache" discusses all this in more detail.

### 1.2.2.8 Performance Monitoring

3rd generation microarchitecture provides four performance monitoring counters that are configured to monitor various events. These events allow a software developer to measure cache efficiency, detect system bottlenecks and reduce the overall latency of programs.

Chapter 11.0, "Performance Monitoring" discusses this in more detail.

### 1.2.2.9 Power Management

3rd generation microarchitecture incorporates a power and clock management unit which allows software to use various low power modes implemented by ASSPs.

These features are described in Section 7.3, "CP14 Registers".

### 1.2.2.10 Software Debug

3rd generation microarchitecture supports software debugging through two instruction address breakpoint registers, one data-address breakpoint register, one data-address/ mask breakpoint register, and a trace buffer.

Chapter 12.0, "Software Debug" discusses this in more detail.

### 1.2.2.11 JTAG

Testability is supported on 3rd generation microarchitecture through the Test Access Port (TAP) Controller implementation, which is based on IEEE 1149.1 (JTAG) Standard Test Access Port and Boundary-Scan Architecture. The purpose of the TAP controller is to support test logic internal and external to 3rd generation microarchitecture such as built-in self-test and boundary-scan.

Appendix D discusses this in more detail.



### 1.2.3 ASSP Options

3rd generation microarchitecture provides a number of features which provide the ASSP with various implementation options. For example, an ASSP chooses whether to provide an L2 cache or not. Or the ASSP chooses to implement additional external co-processors, in addition to 3rd generation microarchitecture's internal co-processors.

A complete list of these ASSP options for 3rd generation microarchitecture features is in Appendix C, "ASSP Options".

For details on how an ASSP implements these options, refer to the relevant product documentation.

## 1.3 Terminology and Conventions

### 1.3.1 Number Representation

All numbers in this document are assumed to be base 10 unless designated otherwise. In text and pseudo code descriptions, hexadecimal numbers have a prefix of 0x and binary numbers have a prefix of 0b. For example, 107 is represented as 0x6B in hexadecimal and 0b1101011 in binary.

### 1.3.2 Terminology and Acronyms

**Table 1. Terminology and Acronyms**

| Term | Description |
|---|---|
| ASSP | Application Specific Standard Product: a product incorporating 3rd generation microarchitecture, often a single chip. |
| Clean | A clean operation writes the contents of a specified cache line out to backing memory when that line is valid and dirty. |
| Coalescing | Coalescing means bringing together a new store operation with an existing store operation already resident in the memory buffer. The new store is placed in the same memory buffer entry as an existing store when the address of the new store falls in the eight word aligned address of the existing entry. |
| Reserved | *A reserved* field is a field that is used by an implementation. When the initial value of a reserved field is supplied by software, this value must be zero. Software must not modify reserved fields or depend on any values in reserved fields. |
| Unpredictable | When a behavior is documented as *unpredictable*, it means that software cannot rely on any specific outcome from the behavior. 3rd generation microarchitecture ensures that such behavior does not cause a hardware lockup or a security hole. Software must avoid using *unpredictable* aspects of 3rd generation microarchitecture. |



# 2.0 Programming Model

This chapter describes the programming model of the 3rd generation Intel XScale® microarchitecture (3rd generation microarchitecture or 3rd generation)[1], namely the implementation options and extensions to the *ARM Architecture Version 5TE Specification* (ARMv5TE).

## 2.1 ARM Architecture Compatibility

3rd generation microarchitecture implements the integer instruction set architecture specified in *ARM Architecture Version 5TE Specification*. 'T' refers to the Thumb instruction set and E refers to the Enhanced DSP Extension.

## 2.2 ARM Architecture Implementation Options

### 2.2.1 Big Endian versus Little Endian

3rd generation microarchitecture supports both Big and Little Endian data representations. The B-bit of the Control Register (Co-processor 15, register 1, bit 7) selects Big and Little Endian mode. To run in Big Endian mode, the B bit must be set before attempting any sub-word accesses to memory, or the results are unpredictable. This bit takes effect even when the MMU is disabled.

*Note:* ASSP chooses to implement only one endian mode. Refer to the 3rd generation microarchitecture implementation options section of the relevant product documentation for more information about which endian modes are supported.

### 2.2.2 Thumb

3rd generation microarchitecture supports the Thumb instruction set.

1. ARM* architecture compliant.

### 2.2.3 ARM Enhanced DSP Extension

3rd generation microarchitecture implements the ARM Enhanced DSP Extension, which includes a set of instructions that boost the performance of signal processing applications. The following are some implementation notes in regard to these instructions:

- **PLD** is interpreted as a read operation by the MMU and is ignored by the data breakpoint unit.
- **PLD** to a non-cacheable page performs no action. Also, when the targeted cache line is already resident, this instruction has no effect.
- **PLD** to a memory region whose virtual-to-physical address translation is not cached in the Translation Lookaside Buffer (TLB) results in a hardware page table walk. However, any MMU aborts resulting from the table walk are ignored.
- Both **LDRD** and **STRD** generates an alignment aborts when address bits [2:0] is not 0b000 and alignment checking is enabled.

**MCRR** and **MRRC** are supported by internal 3rd generation microarchitecture co-processors only when directed to co-processor 0 to access the internal accumulator. See Section 2.3.1.2 for more information on accessing the internal accumulator with these instructions. Using these instructions to access any other internal 3rd generation microarchitecture co-processor (co-processors 14 and 15) results in an undefined instruction exception. Refer to the 3rd generation microarchitecture implementation options section of the relevant product documentation for the behavior when accessing all other co-processors.

### 2.2.4 Base Register Update

When a precise data abort is signalled on a memory instruction that specifies writeback, the contents of the base register is not updated. This holds for all load and store instructions. This is referred to in the ARMv5TE architecture as the *Base Restored Abort Model*.

### 2.2.5 Multiply Operand Restriction

3rd generation microarchitecture supports specifying the same ARM register as Rd and Rm for **MUL** and **MLA**. For **SMULL**, **SMLAL**, **UMULL**, and **UMLAL**, the same ARM register is specified for RdHi and Rm or RdLo and Rm. The results are no longer unpredictable as defined in ARMv5TE.

## 2.3 Extensions to ARM Architecture

3rd generation microarchitecture extends the ARMv5TE architecture to meet the needs of various markets and design requirements. The following is a list of the extensions which are discussed in subsequent sections.

- A Media Processing Co-processor (CP0) has been added that contains a 40-bit internal accumulator. Five new instructions have been added which access the 40-bit accumulator.
- Page Attributes were added to the page table descriptors and the description of existing attributes in ARMv5TE were enhanced. Note that compatibility is maintained with software developed using page table attributes for previous microarchitectures.
- Co-processor 7 and Co-processor 14 registers are added to 3rd generation microarchitecture.
- Co-processor 15 functionality is extended and new registers are added.
- Enhancements were made to the Exception Architecture, which include instruction cache and data cache parity error exceptions, debug exceptions, and imprecise external data aborts.

### 2.3.1 Media Processing Co-processor (CP0)

3rd generation microarchitecture adds a Media Processing co-processor to the architecture for the purpose of increasing the performance and the precision of audio processing algorithms. This co-processor contains a 40-bit internal accumulator and eight new instructions.

*Note:* Products using 3rd generation microarchitecture extend the definition of CP0; for example, products implement 64-bit accumulators or additional instructions are defined. Refer to the 3rd generation microarchitecture implementation options section of the relevant product documentation for more information on any extensions. The remainder of this section applies only when the ASSP has not extended the definition of CP0.

The 40-bit accumulator is referenced by several new instructions that were added to the architecture; **MIA**, **MIAPH** and **MIAxy** are multiply/accumulate instructions that reference the 40-bit accumulator instead of a register specified accumulator. **MAR** and **MRA** provide the ability to read and write the 40-bit accumulator.

Access to CP0 is always allowed in all processor modes when bit 0 of the Co-processor Access Register is set. Any access to CP0 when this bit is clear causes an undefined instruction exception. (See Section 7.2.15, "Register 15: Co-processor Access Register" for more details). Note that only privileged software sets this bit in the Co-processor Access Register.

**LDC** and **STC** instructions that target co-processor 0 generates an undefined instruction exception.

Software must save the 40-bit accumulator on a context switch when multiple processes are using it.

Two new instruction formats were added for co-processor 0: Multiply with Internal Accumulate Format and Internal Accumulator Access Format. The formats and instructions are described next.

### 2.3.1.1 Multiply With Internal Accumulate Format

A multiply format has been created to define operations on 40-bit accumulators. Table 2, "Multiply with Internal Accumulate Format" on page 29 shows the layout of the new format. The opcode for this format lies within the co-processor register transfer instruction type, however a new syntax has been created for these instructions to simplify usage.

**Table 2. Multiply with Internal Accumulate Format**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | | | | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | opcode_3 | | | | Rs | | | | 0 | 0 | 0 | 0 | acc | | | 1 | Rm | | | |

| Bits | Field | Description |
|---|---|---|
| 31:28 | cond | Condition under which the instruction is executed[a] |
| 19:16 | opcode_3 | Type of multiply with the internal accumulate.<br>3rd generation microarchitecture defines the following:<br>0b0000 = **MIA**<br>0b1000 = **MIAPH**<br>0b1100 = **MIABB**<br>0b1101 = **MIABT**<br>0b1110 = **MIATB**<br>0b1111 = **MIATT**<br>The effect of all other encodings are unpredictable. |
| 15:12 | Rs | ARM Register containing Multiplier |
| 7:5 | acc | Specifies 1 of 8 accumulators.<br>3rd generation microarchitecture only implements acc0; access to any other acc has unpredictable results. |
| 3:0 | Rm | ARM register containing Multiplicand |

a. Specifying 0b1111 in the cond field results in an undefined instruction exception when the instruction executes.

Two new fields were created for this format, *acc* and *opcode_3*. The *acc* field specifies 1-of-8 internal accumulators to operate on and *opcode_3* defines the operation for this format. 3rd generation microarchitecture defines a single 40-bit accumulator referred to as *acc0;* future implementations define multiple internal accumulators.
3rd generation microarchitecture uses opcode_3 to define six instructions, **MIA**, **MIAPH**, **MIABB, MIABT, MIATB and MIATT.**

**Table 3. MIA{<cond>} acc0, Rm, Rs**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | | | | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | Rs | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | Rm | | | |

```
Operation:  if ConditionPassed(<cond>) then

                acc0 = Rm[31:0] * Rs[31:0] + acc0[39:0]

Exceptions: none

Qualifiers  Condition Code

            no condition code flags are updated

Notes:      Instruction timings are found
            in Section 13.4.4, "Multiply Instruction Timings" on page 225.

            Specifying R15 for register Rs or Rm has unpredictable results.

            acc0 is defined to be 0b000 on 3rd generation microarchitecture
```

The **MIA** instruction operates similarly to **MLA** except that the 40-bit accumulator is used. **MIA** multiplies the signed value in register Rs (multiplier) by the signed value in register Rm (multiplicand) and then adds the result to the 40-bit accumulator (acc0).

**MIA** does not support unsigned multiplication; all values in Rs and Rm are interpreted as signed data values. **MIA** is useful for operating on signed 16-bit data that was loaded into a general purpose register by **LDRSH**.

The instruction is only executed when the condition specified in the instruction matches the condition code status.

**Table 4. MIAPH{<cond>} acc0, Rm, Rs**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | | | | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | Rs | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | Rm | | | |

```
Operation:  if ConditionPassed(<cond>) then

                acc0 = sign_extend(Rm[31:16] * Rs[31:16]) +

                       sign_extend(Rm[15:0] * Rs[15:0]) +

                       acc0[39:0]

Exceptions: none

Qualifiers  Condition Code

            no condition code flags are updated

Notes:      Instruction timings are found
            in Section 13.4.4, "Multiply Instruction Timings" on page 225.

            Specifying R15 for register Rs or Rm has unpredictable results.

            acc0 is defined to be 0b000 on 3rd generation microarchitecture
```

The **MIAPH** instruction performs two16-bit signed multiplies on packed half-word data and accumulates these to a single 40-bit accumulator. The first signed multiplication is performed on the lower 16 bits of the value in register Rs with the lower 16 bits of the value in register Rm. The second signed multiplication is performed on the upper 16 bits of the value in register Rs with the upper 16 bits of the value in register Rm. Both signed 32-bit products are sign extended to 40 bits and then added to the value in the 40-bit accumulator (acc0).

The instruction is only executed when condition specified in the instruction matches the condition code status.

**Table 5. MIA<T,B><T,B>{<cond>} acc0, Rm, Rs**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | | | | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | x | y | Rs | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | Rm | | | |

```
Operation:  if ConditionPassed(<cond>) then

                if (bit[17] == 0)

                    <operand1> = Rm[15:0]

                else

                    <operand1> = Rm[31:16]

                if (bit[16] == 0)

                    <operand2> = Rs[15:0]

                else

                    <operand2> = Rs[31:16]

                acc0[39:0] = sign_extend(<operand1> * <operand2>) + acc0[39:0]

Exceptions: none

Qualifiers  Condition Code

            no condition code flags are updated

Notes:      Instruction timings are found
            in Section 13.4.4, "Multiply Instruction Timings" on page 225.

            Specifying R15 for register Rs or Rm has unpredictable results.

            acc0 is defined to be 0b000 on 3rd generation microarchitecture.
```

The **MIAxy** instruction performs one 16-bit signed multiply and accumulates this to a single 40-bit accumulator. **x** refers to either the upper half or lower half of Rm (multiplicand) and **y** refers to the upper or lower half of Rs (multiplier). The upper or lower 16-bits of each source register half is selected by specifying either the B (bottom) or T (top) qualifier in each of the xy positions of the mnemonic.

**Table 6. MIAxy Subfield Encoding**

| Qualifier | Field | Value |
|---|---|---|
| T | x | 1 |
| B | x | 0 |
| T | y | 1 |
| B | y | 0 |

**MIAxy** does not support unsigned multiplication; all values in Rs and Rm are interpreted as signed data values. The instruction is only executed when the condition specified in the instruction matches the condition code status.

### 2.3.1.2 Internal Accumulator Access Format

3rd generation microarchitecture defines an instruction format for accessing internal accumulators in CP0. Table 7, "Internal Accumulator Access Format" on page 33 shows that the opcode falls into the co-processor register transfer space.

The *RdHi* and *RdLo* fields allow up to 64 bits of data transfer between 3rd generation microarchitecture registers and an internal accumulator. The *acc* field specifies 1 of 8 internal accumulators to transfer data to/from.
3rd generation microarchitecture defines a single 40-bit internal accumulator referred to as acc0; future implementations provide multiple internal accumulators of varying size, up to 64-bits.

3rd generation microarchitecture implements two instructions **MAR** and **MRA** that move two ARM registers to acc0 and move acc0 to two ARM registers, respectively.

**Table 7. Internal Accumulator Access Format**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | | | | 1 | 1 | 0 | 0 | 0 | 1 | 0 | L | RdHi | | | | RdLo | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | acc | | |

| Bits | Field | Description |
|---|---|---|
| 31:28 | cond | Condition under which instruction is executed[a] |
| 20 | L | Move to / from internal accumulator<br>0: move to internal accumulator (MAR)<br>1: move from internal accumulator (MRA) |
| 19:16 | RdHi | ARM register for high order 8 bits of the internal accumulator (acc[39:32]).<br>On a read from the acc, acc[39:32] are sign extended to 32-bits and placed in this register.<br>On a write to the acc, the lower 8 bits of this register are written to acc[39:32] |
| 15:12 | RdLo | ARM register for low order 32 bits of the internal accumulator.<br>On a read from the acc, acc[31:0] are placed in this register.<br>On a write to the acc, this register is written to acc[31:0] |
| 2:0 | acc | Specifies 1 of 8 internal accumulators.<br>3rd generation microarchitecture only implements acc0; access to any other acc is unpredictable |

a. Specifying 0b1111 in the cond field results in an undefined instruction exception when the instruction executes.

*Note:* **MAR** has the same encoding as **MCRR** (to co-processor 0) and **MRA** has the same encoding as **MRRC** (to co-processor 0). These instructions move 64-bits of data to/from ARM registers from/to co-processor registers. **MCRR** and **MRRC** are defined in ARM Enhanced DSP instruction set.

Disassemblers not aware of **MAR** and **MRA** produces the following syntax:

```
MCRR{<cond>} p0, 0x0, RdLo, RdHi, c0

MRRC{<cond>} p0, 0x0, RdLo, RdHi, c0
```

### Table 8. MAR{<cond>} acc0, RdLo, RdHi

| 31 30 29 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 | 2 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | RdHi | RdLo | 0 0 0 0 | 0 0 0 0 | 0 | 0 0 0 |

```
Operation:  if ConditionPassed(<cond>) then

                acc0[39:32] = RdHi[7:0]

                acc0[31:0] = RdLo[31:0]

Exceptions: none

Qualifiers  Condition Code

            No condition code flags are updated

Notes:      Instruction timings are found in

            Section 13.4.4, "Multiply Instruction Timings" on page 225

            Specifying R15 as either RdHi or RdLo has unpredictable results.
```

The **MAR** instruction moves the value in register RdLo to bits[31:0] of the 40-bit accumulator (acc0) and moves bits[7:0] of the value in register RdHi into bits[39:32] of acc0. The instruction is only executed when the condition specified in the instruction matches the condition code status.

### Table 9. MRA{<cond>} RdLo, RdHi, acc0

| 31 30 29 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 | 2 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | RdHi | RdLo | 0 0 0 0 | 0 0 0 0 | 0 | 0 0 0 |

```
Operation:  if ConditionPassed(<cond>) then

                RdHi[31:0] = sign_extend(acc0[39:32])

                RdLo[31:0] = acc0[31:0]

Exceptions: none

Qualifiers  Condition Code

            No condition code flags are updated

Notes:      Instruction timings are found in

            Section 13.4.4, "Multiply Instruction Timings" on page 225

            Specifying the same register for RdHi and RdLo has unpredictable
            results.

            Specifying R15 as either RdHi or RdLo has unpredictable results.
```

The **MRA** instruction moves the 40-bit accumulator value (acc0) into two registers. Bits[31:0] of the value in acc0 are moved into the register RdLo. Bits[39:32] of the value in acc0 are sign extended to 32 bits and moved into the register RdHi.

The instruction is only executed when the condition specified in the instruction matches the condition code status.

## 2.3.2 Page Attributes

3rd generation microarchitecture implements the MMU architecture defined by ARMv5TE, with the following extensions:

- new first-level page table descriptor format added (supersection descriptor)
- new second-level page table descriptor format added (extended small page descriptor)
- 3-bit field in page table descriptor to define memory attributes (TEX field)
- One bit in page table descriptor to define shared memory (S bit)
- One bit in page table descriptor for ASSP defined attribute (P bit)
- Auxiliary Control Register (Low-Locality Reference attribute control)
- Translation Table Base Register extensions (P bit and outer cacheability field for table walk)

3rd generation microarchitecture adds the super section descriptor within the first-level descriptor format and the extended small page descriptor within the coarse second-level descriptor format. The super section descriptor allows a 36-bit physical address space to be supported. The extended small page descriptor allows additional memory attributes (vs. small page) to be programmed for a 4 K page. These new formats are described in Section 3.2.2, "Page Table Descriptor Formats".

3rd generation microarchitecture also extends page attributes defined in ARMv5TE. These extensions allow more attributes to be defined, including support for shared memory and L2 caching.

The descriptor TEX field extends page attributes defined by C and B bits for additional L1 and L2 cache attributes. When TEX is 0b000, 3rd generation microarchitecture retains the ARMv5TE definitions of the C and B encodings for L1 cache behavior (with some extensions to control L2 cache behavior). When the TEX field is not 0b000, these bits provide additional control over L1 and L2 cache behavior.

One of the particular options for the L1 data cache, using the TEX field, is to define a region of memory as having Low-Locality Reference (LLR). This features allows 3rd generation microarchitecture to provide similar functionary to the mini-data cache found on previous microarchitectures (Section 6.1.2, "Low-Locality of Reference (LLR)" for more information). 3rd generation microarchitecture adds the Auxiliary Control Register (co-processor 15, register 1, opcode2=1) to control the LLR attributes. Refer to Section 7.2.2, "Register 1: Control and Auxiliary Control Registers" for more details.

The S bit in the descriptor enables a memory region to be defined as shared. Setting this bit to 1 allows a region of memory to be defined for multi-agent access and allows cache coherence to be performed on accesses to that memory.

A full list of memory attribute encodings of the TEX, C, B and S bits is found in Section 3.2.2, "Page Table Descriptor Formats". The location of the new bits with the various descriptor types are found in Section 3.2.2, "Page Table Descriptor Formats".

3rd generation microarchitecture adds a P bit in the first-level descriptors to allow an ASSP to identify a new memory attribute. Refer to the 3rd generation microarchitecture implementation options section of the relevant product documentation to find out how the P bit has been defined. More details on the P-bit are found in Section 3.2.3.4, "ASSP Specific Attribute (P-bit)".

3rd generation microarchitecture also allows software to program the P bit and outer cacheability attributes for memory accesses made during a page table walk. This is done using the corresponding P bit and OC field in the Translation Table Base Register. See Section 7.2.3, "Register 2: Translation Table Base Register" for more details.

### 2.3.3 CP7 Functionality

3rd generation microarchitecture uses a portion of CP7 to provide error logging registers for reporting information on external bus errors and L2 cache parity errors.

The remaining portion of CP7 not used by 3rd generation microarchitecture is used by ASSP specific co-processors.

The error logging registers are described in Section 7.4, "CP7 Registers".

### 2.3.4 CP14 Functionality

3rd generation microarchitecture uses CP14 to provide additional co-processor functionality related to the following areas:

- Software Debug
- Performance Monitoring
- Clock and Power Management

For more specific details on these co-processor registers refer to Chapter 7.0, "Configuration". Additional information on these software debug and performance monitoring features are found in Chapter 12.0, "Software Debug" and Chapter 11.0, "Performance Monitoring", respectively.

### 2.3.5 CP15 Functionality

To accommodate the functionality in 3rd generation microarchitecture, the following CP15 registers have been added to or changed from ARMv5TE.:

- L2 System ID and L2 Cache Type Registers
- Auxiliary Control Register
- Co-processor Access Register
- Hardware Breakpoint Resources
- Instruction Cache and Data Cache Lockdown
- Instruction TLB and Data TLB Lockdown
- Fault Status Register
- Translation Table Base
- Functions to control an L2 cache
- Expanded definition of DC Line Allocate

Refer to Chapter 7.0, "Configuration" for more specific information on these registers.

## 2.3.6 Exception Architecture

### 2.3.6.1 Exception Summary

Table 10 shows all the exceptions that 3rd generation microarchitecture generates, and the attributes of each. Subsequent sections give details on each exception.

**Table 10. Exception Summary**

| Exception Description | Exception Type[a] | Precise | Updates FSR | Updates FAR |
|---|---|---|---|---|
| Reset | Reset | N | N | N |
| FIQ | FIQ | N | N | N |
| IRQ | IRQ | N | N | N |
| External Instruction[b] | Prefetch | Y | Y | N |
| Instruction MMU | Prefetch | Y | Y | N |
| Instruction Cache Parity | Prefetch | Y | Y | N |
| Lock Abort | Data | Y | Y | N |
| Data MMU | Data | Y | Y | Y |
| External Data[c] | Data | N | Y | N |
| Data Cache Parity | Data | N | Y | N |
| Software Interrupt | Software Interrupt | Y | N | N |
| Undefined Instruction | Undefined Instruction | Y[d] | N | N |
| Debug Exceptions[e] | varies | varies | varies | N |

a. Exception types are those described in the *ARM Architecture Version 5TE Specification*, Section 2.6.
b. External Instruction includes bus errors and L2 cache parity errors on instruction fetches
c. External Data includes bus errors and L2 cache parity errors on data accesses
d. An ASSP uses an undefined instruction exception to report imprecise co-processor exceptions. Refer to the implementation options section of the relevant product documentation for more information on any co-processors that are defined.
e. Refer to Chapter 12.0, "Software Debug" for more details

### 2.3.6.2 Exception Priority

3rd generation microarchitecture follows the exception priority specified in the *ARM Architecture Version 5TE Specification*. The processor has additional exceptions that are generated while debugging. For information on these debug exceptions, see Chapter 12.0, "Software Debug".

**Table 11. Exception Priority**

| Exception | Priority |
|---|---|
| Reset | 1 (Highest) |
| Data Abort (Precise & Imprecise) | 2 |
| FIQ | 3 |
| IRQ | 4 |
| Prefetch Abort | 5 |
| Undefined Instruction, SWI | 6 (Lowest) |

### 2.3.6.3 Prefetch Aborts

3rd generation microarchitecture detects three types of prefetch aborts: Instruction MMU abort, external instruction error, and an instruction cache parity error. These aborts are described in Table 12.

When a prefetch abort occurs, hardware reports it in the extended Status field of the Fault Status Register. The value placed in R14_ABORT (the link register in abort mode) is the address of the aborted instruction + 4.

The external instruction error includes external bus errors and L2 cache parity errors which are reported during an instruction fetch.

**Table 12. Encoding of Fault Status for Prefetch Aborts**

| Priority | Sources | FS[10,3:0][a] | Domain | FAR |
|---|---|---|---|---|
| Highest | Instruction MMU Exception<br>Several exceptions generate this encoding:<br>- translation faults<br>- external abort on translation<br>- domain faults, and<br>- permission faults<br>It is up to software to figure out which one occurred. | 0b10000 | invalid | invalid |
| | External Instruction Error Exception | 0b10110 | invalid | invalid |
| Lowest | Instruction Cache Parity Error Exception | 0b11000 | invalid | invalid |

a. All other encodings not listed in the table are reserved.

### 2.3.6.4 Data Aborts

Two classes of data aborts exist in 3rd generation microarchitecture: precise and imprecise.

On a precise data abort, execution does not proceed beyond the aborting instruction before the microarchitecture redirects execution to the data abort handler. For precise data aborts, R14_ABORT is the address of the aborted instruction + 8.

On an imprecise data abort, execution proceeds beyond the instruction that caused the abort before the microarchitecture enters the data abort handler, or the reported data abort is not associated with a specific instruction. For imprecise data aborts, R14_ABORT is the address of the next instruction to execute in the program flow + 4.

On 3rd generation microarchitecture precise data aborts are recoverable and imprecise data aborts are not recoverable.

**Table 13. Encoding of Fault Status for Data Aborts**

| Priority | Sources | | FS[10,3:0][a] | Domain | FAR |
|---|---|---|---|---|---|
| Highest | Alignment | | 0b000x1 | invalid | valid |
| | External Abort on Translation | First level<br>Second level | 0b01100<br>0b01110 | invalid<br>valid | valid<br>valid |
| | Translation | Section<br>Page | 0b00101<br>0b00111 | invalid<br>valid | valid<br>valid |
| | Domain | Section<br>Page | 0b01001<br>0b01011 | valid<br>valid | valid<br>valid |
| | Permission | Section<br>Page | 0b01101<br>0b01111 | valid<br>valid | valid<br>valid |
| | Lock Abort | | 0b10100 | invalid | invalid |
| | Co-processor Data Abort | | 0b11010 | invalid | invalid |
| | Imprecise External Data Abort | | 0b10110 | invalid | invalid |
| Lowest | Data Cache Parity Error Exception | | 0b11000 | invalid | invalid |

a. All other encodings not listed in the table are reserved.

#### 2.3.6.4.1 Precise Data Aborts

- Lock aborts are precise. These abort occurs when a TLB lock operation (instruction or data TLB) or an instruction cache lock operation causes an exception due to either a translation fault, access permission fault or external bus fault.

  When a lock abort occurs, the Extended Status field of the Fault Status Register (FSR) is set to 0b10100.

  The Fault Address Register is invalid for lock aborts.

- Data MMU aborts are precise. These are due to an alignment fault, translation fault, domain fault, permission fault or external data abort on an MMU translation.

  For MMU aborts, the status field is set to values defined by ARMv5TE. These values is shown in Table 13, "Encoding of Fault Status for Data Aborts" on page 39.

  The Fault Address Register is set to the effective address of the aborting data access.

- Co-processor Data aborts are precise. These data aborts are definable by the ASSP; these allow a co-processor attached to the microarchitecture the ability to generate a data abort and have it reflected in the Fault Status Register. Refer to the 3rd generation microarchitecture implementation options section of the relevant product documentation to see whether this feature is used.

  When a co-processor data abort is generated, the Extended Status field of the FSR is set to 0b11010.

  The Fault Address Register is invalid for co-processor data aborts.

#### 2.3.6.4.2 Imprecise Data Aborts

- Data cache parity errors are imprecise; the extended Status field of the Fault Status Register is set to 0xb11000.
- All external data aborts except for those generated on a data MMU translation are imprecise. External data abort includes external bus errors and L2 parity errors on data accesses. The ASSP also reports other types of external errors as external data aborts. Refer to the implementation options section of the relevant product documentation for additional types of errors that are reported.

The Fault Address Register for all imprecise data aborts is invalid.

Although 3rd generation microarchitecture guarantees the *Base Restored Abort Model* (see Section 2.2.4, "Base Register Update" on page 27) for precise aborts, it cannot do so in the case of imprecise aborts. Thus a memory access that uses an addressing mode which updates the base register and generates an imprecise data abort still updates the base register.

Imprecise data aborts create scenarios that are difficult for an abort handler to recover. Both external data aborts and data cache parity errors result in corrupted data in the targeted registers. Because these faults are imprecise, it is possible that the corrupted data was used before the Data Abort handler is invoked. Thus, software treats imprecise data aborts as unrecoverable.

#### 2.3.6.4.3 Multiple Data Aborts

Multiple data aborts are detected by hardware but only the highest priority one is reported. Refer to Table 13 on page 39 for the priorities of each type of data abort. When the reported data abort is precise, software corrects the cause of the abort and re-execute the aborted instruction. When the lower priority abort still exists, it is then reported. Software handles each abort separately until the instruction successfully executes.

### 2.3.6.5 Exceptions from Preload Instructions

Even through the **PLD** instruction goes through the normal MMU address translation and loads data from memory, it does not generate any precise data aborts. When the **PLD** encounters a condition which causes a data abort, the **PLD** is effectively canceled, without affecting any of the caches, and the abort is not reported. This allows software to issue **PLD**s speculatively without affecting the state of the processor when an abort is encountered.

For example, Example 1 places a **PLD** instruction early in the loop. This **PLD** is used to fetch data for the next loop iteration. In this example, the list is terminated with a node that has a null pointer. When execution reaches the end of the list, the **PLD** on address 0x0 does not cause a fault. Rather, it is ignored and the loop terminates normally.

**Example 1. Speculatively issuing PLD**

```
; R0 points to a node in a linked list. A node has the following layout:
; Offset    Contents
;-----------------------------------
;       0   data
;       4   pointer to next node
; This code computes the sum of all nodes in a list. The sum is placed into R9.
;
        MOV R9, #0      ; Clear accumulator
sumList:
        LDR R1, [R0, #4]    ; R1 gets pointer to next node
        LDR R3, [R0]        ; R3 gets data from current node
        PLD [R1]            ; Speculatively start load of next node
        ADD R9, R9, R3      ; Add into accumulator
        MOVS R0, R1         ; Advance to next node. At end of list?
        BNE sumList         ; If not then loop
;
; Note that the end of the list is marked with a NULL pointer (0x0).
; The descriptor for this page of memory is valid, but
; disallow access. If an invalid descriptor is used, then it
; will not be cached in the TLB and will require a table walk
; each time it is PLDed.
;
```

### 2.3.6.6 Debug Exceptions

Debug exceptions are covered in Section 12.3.3, "Debug Exceptions".



# 3.0 Memory Management

Together with the *ARM Architecture Version 5TE Specification,* this chapter describes the memory management unit implemented by the 3rd generation Intel XScale® microarchitecture (3rd generation microarchitecture or 3rd generation).

## 3.1 Overview

3rd generation microarchitecture implements the ARM Memory Management Unit (MMU) defined by the *ARM Architecture Version 5TE Specification* with some extensions, including support for shared memory, an L2 cache, and 36-bit physical addressing. This chapter describes the 3rd generation microarchitecture specific MMU features and assumes the reader has prior knowledge of the ARM MMU Architecture.

3rd generation microarchitecture supports the multi-level page table structure and page table entries defined by the ARM MMU Architecture. The page table allows various size regions of memory to be defined with similar attributes. The individual entries in the table (known as descriptors) specify the virtual to physical address translation, memory protection, and memory attribute information for a specific region of memory.

3rd generation microarchitecture extends the ARM MMU Architecture with two new descriptor types: supersection and extended small page. A supersection allows a 32-bit virtual address to be mapped to a 36-bit physical address space (see Section 3.2.2.1); the extended small page is similar to a small page, except it allows additional memory attributes to be specified for 4KB pages of memory (see Section 3.2.2.2).

The memory protection used by 3rd generation microarchitecture is the same as that defined by the ARM MMU Architecture (see *ARM Architecture Version 5TE Specification*).

3rd generation microarchitecture extends the memory attributes defined by the ARM MMU Architecture to support additional capabilities such as an L2 cache and shared memory. The 3rd generation microarchitecture page tables allow system software to associate the following attributes with regions of memory:

- cacheable in Level 1 (L1) instruction cache and data cache (see Section 3.2.5)
- cacheable in L2 cache (see Section 3.2.6)
- shared memory (hardware supported coherency) (see Section 3.2.3.2)
- write-back vs. write-through L1 data cache write policy (see Section 3.2.4)
- coalescing (see Section 3.2.4)
- an ASSP definable attribute (see Section 3.2.3.4)
- low locality of reference (LLR) for L1 data cache (see Section 3.2.3.3)

To accelerate virtual to physical address translation, 3rd generation microarchitecture uses both an instruction Translation Lookaside Buffer (TLB) and a data TLB to cache the latest translations. In addition to the address translation, the TLBs contain memory access permissions and memory region attributes.

On an instruction or data TLB miss, the microarchitecture invokes a hardware mechanism, known as a table walk. The table walk reads the page table in backing memory to get the virtual to physical address mapping, as well as memory attributes for the region of memory being accessed.

When a Level 2 (L2) cache is present and enabled, 3rd generation microarchitecture is configured to cache all table walks in the L2 cache to help improve performance when fetching descriptors on a TLB miss.

Following a table walk, the address translation and memory attribute information is placed in the TLB.

For additional details on the address translation process refer to Section 3.2.1.

The MMU reports prefetch aborts (for instruction fetches) or data aborts (for data accesses) during the address translation process. The types of aborts which are generated are described in Section 3.2.7.

Software controls and manage the MMU using registers and functions in Co-processor 15 (CP15). More information on the control and management functions are found in Section 3.3.



## 3.2 Architecture Model

### 3.2.1 Address Translation Process

3rd generation microarchitecture uses separate TLBs for instruction and data accesses to speed up the address translation process. Both the instruction TLB and data TLB contain 32 entries and are fully associative. These are managed using the TLB functions available in CP15, register 8 and register 10 (see Section 3.3).

The MMU accesses the TLB and does table walks based on the modified virtual address (MVA). This means that all operations which operate on a virtual address (instruction fetches, data accesses, DC line allocate) are first remapped by the Process ID register, as described in Section 7.2.13, "Register 13: Process ID". It is this remapped address, the MVA, that is used when searching the TLB or, in the case of a TLB miss, for reading the page table in memory.

When an MVA does not hit in the TLB, a table walk is required. During a page table walk, bits in the Translation Table Base Register are used to specify certain memory attributes to use during the table walk. In particular, the ASSP specific attribute and L2 cacheability - are applied to the table walk. For more information on programming these attributes, refer to Section 7.2.3, "Register 2: Translation Table Base Register".

The L2 cacheability of page table walks is used to control whether page table walks are cached in the L2 or not. When the L2 cacheability is programmed for non-L2-cacheable, table walks do not get cached in the L2 cache. When the L2 cacheability for table walks is programmed to be L2 cacheable, then all table walks when the L2 cache is present and enabled, gets cached into the L2 cache.

Thus, when page table walks are configured to be L2 cacheable, on a TLB miss (with the L2 cache present and enabled), page table walks first check the L2 cache before going to main memory. When the table walk hits the L2 cache, the descriptor is read from the cache. When it misses the L2 cache, then the descriptor is loaded from external memory, caching it in the L2 cache in the process. When a descriptor is cached in the L2 cache, an entire cache line is written, so additional page table entries are also cached starting at the previous cache line boundary. Note that caching descriptors in the L2 cache on a table walk leads to an existing line in the cache being replaced or evicted.

## 3.2.2 Page Table Descriptor Formats

3rd generation microarchitecture extends the descriptors defined in ARM MMU Architecture with the supersection and extended small page descriptors. Table 14 through Table 16 show the page table descriptor formats supported by 3rd generation microarchitecture.

### Table 14. First-level Descriptors

<table>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="30">SBZ</td><td>0</td><td>0</td></tr>
<tr><td colspan="21">Coarse page table base address</td><td>P</td><td colspan="4">Domain</td><td colspan="3">SBZ</td><td>0</td><td>1</td></tr>
<tr><td colspan="12">Section base Address</td><td>SBZ</td><td>0</td><td>SBZ</td><td>S</td><td>SBZ</td><td>TEX</td><td>AP</td><td>P</td><td colspan="4">Domain</td><td>0</td><td>C</td><td>B</td><td>1</td><td>0</td></tr>
<tr><td colspan="4">Supersection base address</td><td colspan="8">Base address [35:32]</td><td>SBZ</td><td>1</td><td>SBZ</td><td>S</td><td>SBZ</td><td>TEX</td><td>AP</td><td>P</td><td colspan="4">SBZ</td><td>0</td><td>C</td><td>B</td><td>1</td><td>0</td></tr>
<tr><td colspan="20">Fine page table base address</td><td>SBZ</td><td>P</td><td colspan="4">Domain</td><td colspan="3">SBZ</td><td>1</td><td>1</td></tr>
</table>

### Table 15. Second-level Descriptors for Coarse Page Table

<table>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="30">SBZ</td><td>0</td><td>0</td></tr>
<tr><td colspan="16">Large page base address</td><td>S</td><td colspan="2">TEX</td><td>AP3</td><td>AP2</td><td>AP1</td><td>AP0</td><td>C</td><td>B</td><td>0</td><td>1</td></tr>
<tr><td colspan="20">Small page base address</td><td>AP3</td><td>AP2</td><td>AP1</td><td>AP0</td><td>C</td><td>B</td><td>1</td><td>0</td></tr>
<tr><td colspan="20">Extended small page base address</td><td>SBZ</td><td>S</td><td colspan="2">TEX</td><td>AP</td><td>C</td><td>B</td><td>1</td><td>1</td></tr>
</table>

### Table 16. Second-level Descriptors for Fine Page Table

<table>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="30">SBZ</td><td>0</td><td>0</td></tr>
<tr><td colspan="16">Large page base address</td><td>S</td><td colspan="2">TEX</td><td>AP3</td><td>AP2</td><td>AP1</td><td>AP0</td><td>C</td><td>B</td><td>0</td><td>1</td></tr>
<tr><td colspan="20">Small page base address</td><td>AP3</td><td>AP2</td><td>AP1</td><td>AP0</td><td>C</td><td>B</td><td>1</td><td>0</td></tr>
<tr><td colspan="20">Tiny Page Base Address</td><td>SBZ</td><td colspan="3">TEX</td><td>AP</td><td>C</td><td>B</td><td>1</td><td>1</td></tr>
</table>

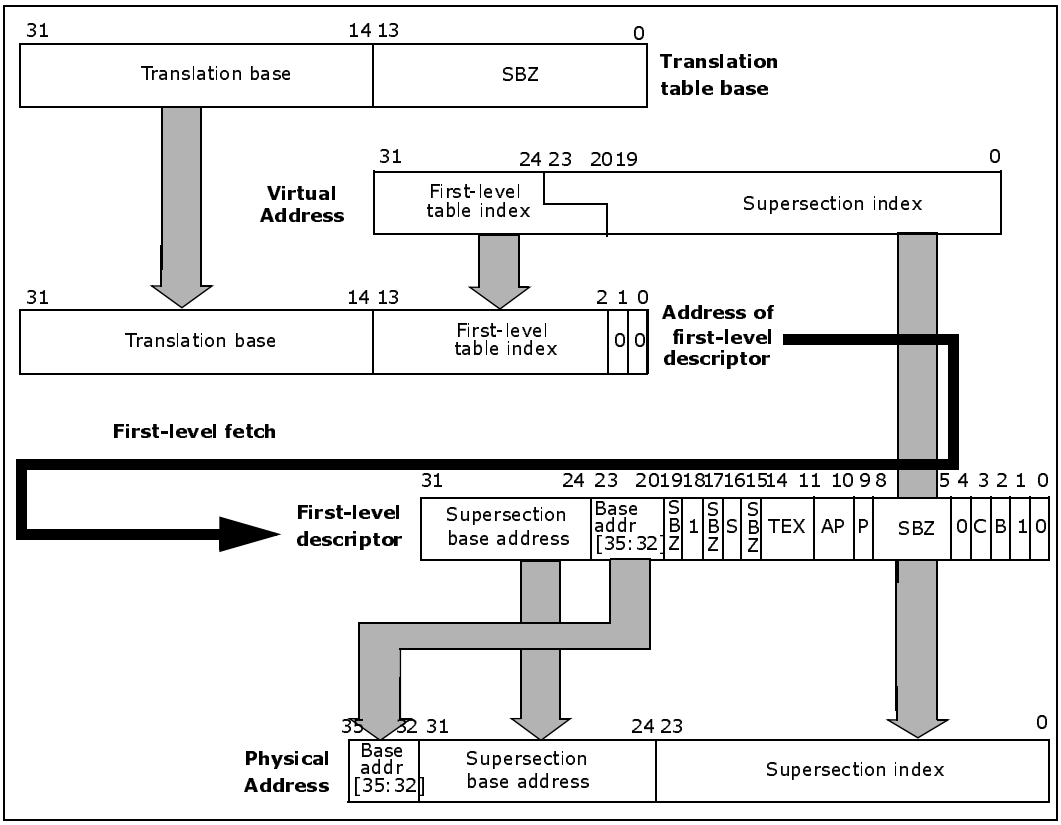
### 3.2.2.1 Supersection Descriptor

3rd generation microarchitecture defines a first-level descriptor, known as a supersection, to support 36-bit physical addressing. The supersection descriptor, shown in Table 14 is based on the section descriptor format, with bit 18 of the descriptor used to differentiate between the two.

Figure 2, "Address Translation for Supersection" on page 46 shows the process for translating a 32-bit virtual address into a 36-bit physical address using a supersection descriptor. A supersection defines a 16 MB region of memory and must start on a 16 MB boundary. Supersections always use Domain 0.

Note in Figure 2, the virtual address shows the lower 4 bits of the first-level table index overlapping with the upper four bits of the supersection index. Since a supersection covers 16 MB of memory, it consumes 16 consecutive descriptor entries in the first level page table. All 16 entries must be programmed with the same descriptor value, otherwise the results are unpredictable.

**Figure 2. Address Translation for Supersection**




### 3.2.2.2 Extended Small Page Descriptor

3rd generation microarchitecture defines a second level descriptor, known as an extended small page, to allow memory attributes to be specified on a 4 KB page size. Note that the extended small page is only defined for a coarse second level page table (refer to Table 15).

The address translation for an extended small page is the same as for a small page (refer to the *ARM Architecture Version 5TE Specification*).

## 3.2.3 Memory Attributes

The attributes associated with a region of memory are configured in the page table and control the behavior of accesses to the L1 caches (instruction and data), L2 cache, and write-buffers.

When the MMU is disabled, memory attributes defined in the page table are ignored. All instruction fetches default to L1 cacheable and L2 uncacheable. Data accesses default to strongly ordered (refer to Section 3.2.4 for a definition of strongly ordered).

Refer to Section 3.2.5, "L1 Instruction Cache, Data Cache Behavior" on page 55 and Section 3.2.6, "L2 Cache Behavior" on page 56 for more information on L1 and L2 cache behavior.

### 3.2.3.1 Inner/Outer Cacheability

3rd generation microarchitecture provides support for multiple layers of cache, referred to as the *inner* and *outer* caches. Inner/Outer refers to the levels of caches that are built in a system. Inner refers to the inner-most caches, including L1. Outer refers to the outer-most caches. Inner cache on 3rd generation microarchitecture is defined to be the L1 instruction and data caches. The outer cache is defined to be the L2 cache, when present.

The inner/outer cacheability attributes are not controlled by any individual bits but rather by a combination of descriptor bits and also by whether the L2 is present or not. For memory regions defined as Low Locality Reference (see Section 3.2.3.3) attribute bits in the Auxiliary Control Register (see Section 7.2.2, "Register 1: Control and Auxiliary Control Registers") also control inner/outer cacheability.

### 3.2.3.2 Coherent Memory Attribute (S-bit)

The coherent memory attribute is used to define a region of memory as being shared by multiple agents. 3rd generation microarchitecture provides hardware cache coherence support for the L1 data cache and the L2 cache based on whether the region is defined to be shared. Hardware based cache coherency is not supported for the L1 instruction cache. For shared memory, 3rd generation microarchitecture employs the MOESI protocol to maintain L2 cache coherence and VI protocol for the L1 data cache.

The shared attribute is supported for all page types, except for small pages and tiny pages. It is represented by the S bit in the descriptors (Table 14, Table 15, and Table 16).

Setting the S bit to 1 does not ensure that a page is both cacheable and coherent. At a minimum, the following conditions must be also be met:

- The MMU must be enabled AND
- The L2 cache must be present and enabled AND
- The page must be defined as L2 cacheable/write-back in the descriptor
- ASSP coherence support must be present and enabled (Refer to the relevant product documentation for more information on whether any additional coherency support is provided). When the S-bit is set for an ASSP that does not support coherency, the results are unpredictable.

Do not lock data from shared memory regions in the L1 data cache. Doing so results in unpredictable behavior of the system.

*Note:* Setting the S-bit in the descriptor is not the only way to define a shared region of memory. Non-cacheable memory regions are coherent since the data is not cached for these memory regions.

### 3.2.3.3 Low Locality of Reference (LLR)

Certain page table encodings define the L1 data cache to be LLR. This feature allows an application to confine data in LLR memory regions to a single way of the L1 data cache, instead of polluting the entire cache.

The L1 and L2 cache write policies for LLR regions are defined in the Auxiliary Control Register. (Section 34, "Auxiliary Control Register").

For more LLR feature details see Section 6.1.2, "Low-Locality of Reference (LLR)".

### 3.2.3.4 ASSP Specific Attribute (P-bit)

3rd generation microarchitecture provides a method for allowing ASSPs to define their own attribute for a region of memory. ASSPs use the P bit in the 1st level descriptors to assign its own page attribute to a memory region.

This bit is only present in the first level descriptors, so the attribute is only used to specify behavior at 1 megabyte and 16 megabyte (supersection) memory granularity.

Refer to the relevant product documentation for usage details.

### 3.2.4 Memory Attribute Encodings

The memory attributes are encoded within the page table descriptors using the C, B, and shared (S) bits, and type extension (TEX) field. Table 14, Table 15, and Table 16 show the location of these bits in the descriptors.

Table 17 through Table 24 are complete listing of the 3rd generation microarchitecture page attributes. These tables use the following terms for non-cacheable memory:

- *Strongly ordered* defines a non-cacheable, non-coalesceable memory region to which memory accesses behave as bi-directional fences. This means that all agents in a system sees explicit memory accesses in program order relative to a strongly ordered memory access. Explicit memory access refers instructions which do loads and/or stores. Also note that strongly ordered memory is implied to be shared (regardless of the S bit value in the descriptor).
- *Device* memory is non-cacheable memory well suited for memory mapped peripherals. The processor does not coalesce writes to device memory. Instruction fetches to non-shared device memory results in unpredictable behavior. However, for compatibility with previous processors, instruction fetches are done to shared device memory.
- *Inner/Outer Uncacheable* is non-cacheable memory which allows writes to coalesce and be re-ordered.

Additional information on the ordering behavior of access to various types of memory regions is described in Chapter 10.0, "Memory Ordering".

The following table usage of 'X' in a bit position (in other words, 1X0) indicates a bit is 1 or 0.

## Table 17. Cache Attributes with L2 present, S=0

| TEX | C | B | L1 I-cache Cacheable | L1 D-cache Cacheable | L1 DC Write Policy | L2 Cacheable | Writes Coalesce | Description |
|---|---|---|---|---|---|---|---|---|
| 000 | 0 | 0 | N | N | - | N | N | Strongly ordered (shared) |
| 000 | 0 | 1 | N | N | - | N | Y | Shared Inner/Outer uncacheable[a] |
| 000 | 1 | 0 | Y | Y | WT | N | Y | Inner write-through, read-allocate; Outer uncacheable |
| 000 | 1 | 1 | Y | Y | WB | Y | Y | Inner write-back, read allocate; Outer write-back, write allocate |
| 001 | 0 | 0 | N | N | - | N | Y | Inner/Outer uncacheable |
| 001 | 0 | 1 | N | N | - | N | N | Shared Device |
| 001 | 1 | 0 | Y | Y | See Description | See Description | Y | Low Locality of Reference (LLR) Memory Auxiliary Control Register specifies L1 D-cache write policy and L2 cacheability. See Table 19 for details |
| 001 | 1 | 1 | Y | Y | WB | Y | Y | Inner write-back, read-allocate; Outer write-back, write-allocate |
| 010 | 0 | 0 | N | N | - | N | N | Non-shared device |
| 010 | 0 | 1 | N/A | N/A | N/A | N/A | N/A | Reserved |
| 010 | 1 | 0 | N/A | N/A | N/A | N/A | N/A | Reserved |
| 010 | 1 | 1 | N/A | N/A | N/A | N/A | N/A | Reserved |
| 011 | X | X | N/A | N/A | N/A | N/A | N/A | Reserved |
| 1X0 | 0 | 0 | N | N | - | N | Y | Inner/Outer uncacheable |
| 1X0 | 0 | 1 | Y | Y | WB | N | Y | Inner write-back, read-allocate; Outer uncacheable |
| 1X0 | 1 | 0 | Y | Y | WT | N | Y | Inner write-through, read allocate; Outer uncacheable |
| 1X0 | 1 | 1 | Y | Y | WB | N | Y | Inner write-back, read allocate; Outer uncacheable |
| 1X1 | 0 | 0 | N | N | - | Y | Y | Inner uncacheable; Outer write-back, write allocate |
| 1X1 | 0 | 1 | Y | Y | WB | Y | Y | Inner write-back, read-allocate; Outer write-back, write-allocate |
| 1X1 | 1 | 0 | Y | Y | WT | Y | Y | Inner write-through, read allocate; Outer write-back, write allocate |
| 1X1 | 1 | 1 | Y | Y | WB | Y | Y | Inner write-back, read allocate; Outer write-back, write allocate |

a. The inner/outer uncacheable behavior for TEX CB encoding '000 01' is deprecated on 3rd generation microarchitecture. Use TEX CB encoding '001 00' instead when inner/outer uncacheable memory is required.

## Table 18. Cache Attributes with L2 present, S=1

| TEX | C | B | L1 I-cache Cacheable | L1 D-cache Cacheable | L1 D-cache Write Policy | L2 Cacheable | Writes Coalesce | Description |
|---|---|---|---|---|---|---|---|---|
| 000 | 0 | 0 | N | N | - | N | N | Strongly ordered (shared) |
| 000 | 0 | 1 | N | N | - | N | Y | Shared Inner/Outer uncacheable[a] |
| 000 | 1 | 0 | Y | N | - | N | Y | Inner IC cacheable, DC uncacheable<br>Outer uncacheable |
| 000 | 1 | 1 | Y | Y | WT | Y | Y | Inner write-through, read-allocate<br>Outer write-back, write-allocate |
| 001 | 0 | 0 | N | N | - | N | Y | Inner/Outer uncacheable |
| 001 | 0 | 1 | N | N | - | N | N | Shared Device |
| 001 | 1 | 0 | Y | See Description | See Description | See Description | Y | Low Locality of Reference (LLR) memory<br>Auxiliary Control Register specifies L1 D-cache cacheability/write policy and L2 cacheability<br>See Table 20 for details. |
| 001 | 1 | 1 | Y | Y | WT | Y | Y | Inner write-through, read-allocate;<br>Outer write-back, write allocate |
| 010 | 0 | 0 | N | N | - | N | N | Non-shared device |
| 010 | 0 | 1 | N/A | N/A | N/A | N/A | N/A | RESERVED |
| 010 | 1 | 0 | N/A | N/A | N/A | N/A | N/A | RESERVED |
| 010 | 1 | 1 | N/A | N/A | N/A | N/A | N/A | RESERVED |
| 011 | X | X | N/A | N/A | N/A | N/A | N/A | RESERVED |
| 1X0 | 0 | 0 | N | N | - | N | Y | Inner/Outer uncacheable |
| 1X0 | 0 | 1 | Y | N | - | N | Y | Inner IC cacheable, DC uncacheable;<br>Outer uncacheable |
| 1X0 | 1 | 0 | Y | N | - | N | Y | Inner IC cacheable, DC uncacheable;<br>Outer uncacheable |
| 1X0 | 1 | 1 | Y | N | - | N | Y | Inner IC cacheable, DC uncacheable;<br>Outer uncacheable |
| 1X1 | 0 | 0 | N | N | - | Y | Y | Inner uncacheable;<br>Outer write-back, write allocate |
| 1X1 | 0 | 1 | Y | Y | WT | Y | Y | Inner write-through, read-allocate;<br>Outer write-back, write allocate |
| 1X1 | 1 | 0 | Y | Y | WT | Y | Y | Inner write-through, read-allocate;<br>Outer write-back, write allocate |
| 1X1 | 1 | 1 | Y | Y | WT | Y | Y | Inner write-through, read-allocate;<br>Outer write-back, write allocate |

a. The inner/outer uncacheable behavior for TEX CB encoding '000 01' is deprecated on 3rd generation microarchitecture. Use TEX CB encoding '001 00' instead when inner/outer uncacheable memory is required.

### Table 19. LLR Page Attributes, L2 Present Case, S=0

| Auxiliary Control Register Setting | L1 I-cache Cacheable | L1 D-cache Cacheable | L1 D-Cache Write Policy | L2 Cacheable | Writes Coalesce | Description |
|---|---|---|---|---|---|---|
| inner write-through<br>outer uncacheable | Y | Y | WT | N | Y | - |
| inner write-through<br>outer write-back, write allocate | Y | Y | WT | Y | Y | - |
| inner write-back<br>outer uncacheable | Y | Y | WB | N | Y | - |
| inner write-back<br>outer write-back, write allocate | Y | Y | WB | Y | Y | - |

### Table 20. LLR Page Attributes, L2 Present Case, S=1

| Auxiliary Control Register Setting | L1 I-cache Cacheable | L1 D-cache Cacheable | L1 D-cache Write Policy | L2 Cacheable | Writes Coalesce | Description |
|---|---|---|---|---|---|---|
| inner write-through<br>outer uncacheable | Y | N | - | N | Y | L1 DC downgrades to uncacheable |
| inner write-through<br>outer write-back, write allocate | Y | Y | WT | Y | Y | - |
| inner write-back<br>outer uncacheable | Y | N | - | N | Y | L1 DC downgrades to uncacheable |
| inner write-back<br>outer write-back, write allocate | Y | Y | WT | Y | Y | L1 DC downgrades to write-through |

## Table 21. Cache Attributes with no L2, S=0

| TEX | C | B | L1 I-cache Cacheable | L1 D-cache Cacheable | L1 D-cache Write Policy | Writes Coalesce | Description |
|---|---|---|---|---|---|---|---|
| 000 | 0 | 0 | N | N | - | N | Strongly Ordered (shared) |
| 000 | 0 | 1 | N | N | - | Y | Shared Inner uncacheable[a] |
| 000 | 1 | 0 | Y | Y | WT | Y | Inner write-through, read-allocate |
| 000 | 1 | 1 | Y | Y | WB | Y | Inner write-back, read-allocate |
| 001 | 0 | 0 | N | N | - | Y | Inner uncacheable |
| 001 | 0 | 1 | N | N | - | N | Shared Device |
| 001 | 1 | 0 | Y | Y | See Description | Y | Low Locality of Reference (LLR) memory<br>Auxiliary Control Register specifies L1 D-cache write policy.<br>See Table 23 for details. |
| 001 | 1 | 1 | Y | Y | WB | Y | Inner write-back, read-allocate |
| 010 | 0 | 0 | N | N | - | N | Non-shared device |
| 010 | 0 | 1 | N/A | N/A | N/A | N/A | RESERVED |
| 010 | 1 | 0 | N/A | N/A | N/A | N/A | RESERVED |
| 010 | 1 | 1 | N/A | N/A | N/A | N/A | RESERVED |
| 011 | X | X | N/A | N/A | N/A | N/A | RESERVED |
| 1XX | 0 | 0 | N | N | - | Y | Inner uncacheable |
| 1XX | 0 | 1 | Y | Y | WB | Y | Inner write-back, read allocate |
| 1XX | 1 | 0 | Y | Y | WT | Y | Inner write-through, read allocate |
| 1XX | 1 | 1 | Y | Y | WB | Y | Inner write-back, read allocate |

a. The inner/outer uncacheable behavior for TEX CB encoding '000 01' is deprecated on 3rd generation microarchitecture. Use TEX CB encoding '001 00' instead when inner/outer uncacheable memory is required.

## Table 22. Cache Attributes with no L2, S=1

| TEX | C | B | L1 I-cache Cacheable | L1 D-cache Cacheable | L1 D-cache Write Policy | Writes Coalesce | Description |
|---|---|---|---|---|---|---|---|
| 000 | 0 | 0 | N | N | - | N | Strongly ordered (shared) |
| 000 | 0 | 1 | N | N | - | Y | Shared Inner uncacheable[a] |
| 000 | 1 | 0 | Y | N | - | Y | Inner IC cacheable, DC uncacheable |
| 000 | 1 | 1 | Y | N | - | Y | Inner IC cacheable, DC uncacheable |
| 001 | 0 | 0 | N | N | - | Y | Inner uncacheable |
| 001 | 0 | 1 | N | N | - | N | Shared Device |
| 001 | 1 | 0 | Y | N | - | Y | Low Locality of Reference (LLR) memory<br>L1 D-cache downgrades to uncacheable.<br>See Table 24 for details. |
| 001 | 1 | 1 | Y | N | - | Y | Inner IC cacheable, DC uncacheable |
| 010 | 0 | 0 | N | N | - | N | Non-shared device |
| 010 | 0 | 1 | N/A | N/A | N/A | N/A | RESERVED |
| 010 | 1 | 0 | N/A | N/A | N/A | N/A | RESERVED |
| 010 | 1 | 1 | N/A | N/A | N/A | N/A | RESERVED |
| 011 | X | X | N/A | N/A | N/A | N/A | RESERVED |
| 1XX | 0 | 0 | N | N | - | Y | Inner uncacheable |
| 1XX | 0 | 1 | Y | N | - | Y | Inner IC cacheable, DC uncacheable |
| 1XX | 1 | 0 | Y | N | - | Y | Inner IC cacheable, DC uncacheable |
| 1XX | 1 | 1 | Y | N | - | Y | Inner IC cacheable, DC uncacheable |

a. The inner/outer uncacheable behavior for TEX CB encoding '000 01' is deprecated on 3rd generation microarchitecture. Use TEX CB encoding '001 00' instead when inner/outer uncacheable memory is required.

### Table 23. LLR Page Attributes, no L2 case, S=0

| Auxiliary Control Register Setting | L1 I-cache Cacheable | L1 D-cache Cacheable | L1 D-cache Write Policy | Writes Coalesce | Description |
|---|---|---|---|---|---|
| inner write-through<br>outer uncacheable | Y | Y | WT | Y | - |
| inner write-through<br>outer write-back, write allocate | Y | Y | WT | Y | - |
| inner write-back<br>outer uncacheable | Y | Y | WB | Y | - |
| inner write-back<br>outer write-back, write allocate | Y | Y | WB | Y | - |

### Table 24. LLR page attributes, no L2 case, S=1

| Auxiliary Control Register Setting | L1 I-cache Cacheable | L1 D-cache Cacheable | L1 D-cache Write Policy | Writes Coalesce | Description |
|---|---|---|---|---|---|
| inner write-through<br>outer uncacheable | Y | N | - | Y | L1 DC downgrades to uncacheable |
| inner write-through<br>outer write-back, write allocate | Y | N | - | Y | L1 DC downgrades to uncacheable |
| inner write-back<br>outer uncacheable | Y | N | - | Y | L1 DC downgrades to uncacheable |
| inner write-back<br>outer write-back, write allocate | Y | N | - | Y | L1 DC downgrades to uncacheable |

### 3.2.5 L1 Instruction Cache, Data Cache Behavior

While the MMU is disabled all page table attributes are ignored. All instruction accesses are considered to be cacheable and are cached in the L1 instruction cache when it is enabled. Data accesses are treated as strongly ordered.

When the MMU is enabled, the following conditions must be met in order to enable L1 instruction caching:

- instruction cache must be enabled (bit 12, register 1 of CP15 Control Register is set)
- specified address must be marked as L1 cacheable in the page table attributes

Similarly, the following conditions must be met in order to enable L1 data caching:

- data cache must be enabled (bit 2, register 1 of CP15 Control Register is set)
- specified address must be marked as L1 cacheable in the page table attributes
- When S bit is set, must be marked as L2 cacheable/write-back in the page table attributes, and the L2 cache must be present and enabled.

When the S bit is set for a memory region that is defined as L1 cacheable in the page table, there are several scenarios in which 3rd generation microarchitecture automatically downgrades that memory region to be non-cacheable in the L1 data cache, to ensure coherency of shared data (assumes MMU enabled and L1 data cache enabled):

- L2 is not present OR
- L2 is present but disabled OR
- L2 is present and enabled BUT the region is non-cacheable in L2

The cache attributes in the page table also tell the caches how to handle write data that hits the L1 data cache. The two methods of handling write data are write-back and write-through. Write-back updates the data only in the L1 data cache, while write-through updates the data both in the L1 data cache and the backing memory. When the S bit is set for a memory region that meets the above requirements for cacheability in the L1 data cache, L1 data cache defaults to write-through.

The L1 caches only allocate a line in the cache for instruction or data reads that miss the cache (in other words, L1 caches only support read-allocate). Writes to addresses not contained in the L1 data cache never causes a cache line to be allocated. For microarchitectures in which the L2 does not exist, all write misses are placed directly on the internal bus.

For more information on the L1 caches refer to Chapter 4.0, "Instruction Cache" and Chapter 6.0, "Data Cache".



### 3.2.6 L2 Cache Behavior

3rd generation microarchitecture offers the option of an L2 cache. The discussion in this section assumes that the L2 is present. When the L2 cache is not present, ignore this section.

When the MMU is disabled or the L2 cache is disabled, neither instructions nor data are cached in the L2 cache. Accesses to addresses previously cached in the L2 cache do not result in a cache hit.

When the MMU and the L2 cache are enabled, instructions and data are cached in the L2 cache when the target region is defined as L2 cacheable/write-back by the page table attributes.

Page table accesses by the hardware table walk mechanism are cached in the L2 when the following conditions are met:

- MMU is enabled AND
- L2 cache is enabled AND
- Table Walk Outer Cache Attributes field (in the Translation Table Base Register) enables the caching of table walks in the L2 cache. See Section 7.2.3, "Register 2: Translation Table Base Register".

The page tables dictate the cacheability for associated memory regions. However, all cacheable accesses to the L2 is write-back and allocate a cache line on any cacheable miss (in other words, L2 cache is always write-back and write-allocate).

For more information on the L2 Cache refer to Chapter 8.0.

### 3.2.7 Exceptions

The MMU generates aborts on instruction fetches or data accesses.

For an instruction fetch, the MMU generates a prefetch abort for:

- translation faults
- external abort on translation
- domain faults
- permission faults

On a data access, the MMU generates a data abort for:

- alignment faults
- translation faults
- external abort on translation
- domain faults
- permissions faults.
- lock abort (data abort on TLB lock or IC fetch and lock)

Data address alignment checking is enabled by setting bit 1 of the Control Register (CP15, register 1). Alignment faults are still reported when the MMU is disabled. No other MMU exceptions are generated when the MMU is disabled.

Specific information about which abort was generated is reported in the Fault Status Register. In some cases, the target address is also reported in the Fault Address Register. More information on these registers is found in Chapter 7.0, "Configuration".

See Section 2.3.6, "Exception Architecture" on page 37 for additional information on 3rd generation microarchitecture exception reporting.



## 3.3 MMU Control and Management

### 3.3.1 MMU Control

Following a reset, the MMU Enable bit (bit 0) in coprocessor 15, register 1 (Control Register) is cleared and the MMU is disabled. In addition, the TLBs are unlocked and invalidated.

Software enables the MMU by setting this bit.

Software also clears this bit to disable MMU. While the MMU is disabled, no page table walks or TLB accesses occur.

Disabling and re-enabling the MMU in software does not affect the contents of the TLBs; valid TLB entries remain valid, locked TLB entries remain locked.

### 3.3.2 Invalidate TLB Operations

The instruction and data TLB are invalidated using TLB functions in CP15. The TLB functions allow the entire instruction and data TLBs to be invalidated (individually or both with a single command). In addition, individual entries within either TLB are invalidated based on a specified address. See Section 7.2.9, "Register 8: TLB Operations" for more details on these TLB invalidation operations.

### 3.3.3 Locking TLB Entries

Individual entries are locked into the instruction and data TLBs to improve performance of critical code. See Section 7.2.11, "Register 10: TLB Lock Down" for more information on the TLB lock/unlock functions.

When a lock operation finds the virtual address translation already resident in the TLB, the results are unpredictable. To ensure proper operation, software executes an invalidate by entry command before the lock command. Software also accomplishs this by invalidating all entries.

Locking entries into either the instruction TLB or data TLB reduces the available number of entries (by the number that was locked down) for hardware to cache other virtual to physical address translations.

When an MMU abort is generated during an instruction or data TLB lock operation, the Fault Status Register is updated to indicate a Lock Abort, and the exception is reported as a precise data abort.

### 3.3.4 Round-Robin Replacement Algorithm

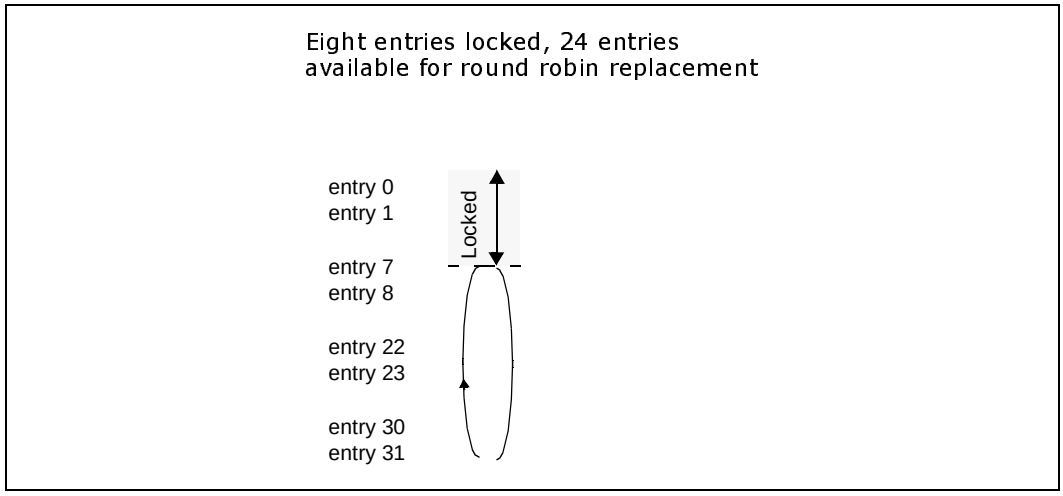
The line replacement algorithm for the TLBs is round-robin; there is a round-robin pointer that keeps track of the next entry to replace. The next entry to replace is the one sequentially after the last entry that was written. For example, when the last virtual to physical address translation was written into entry 5, the next entry to replace is entry 6.

At reset, the round-robin pointer is set to entry 31. Once a translation is written into entry 31, the round-robin pointer gets set to the next available entry, beginning with entry 0 when no entries have been locked down. Subsequent translations move the round-robin pointer to the next sequential entry until entry 31 is reached, where it wraps back to entry 0 upon the next translation. The round-robin algorithm does not search for the next available invalid entry in the TLB. Valid entries are replaced even when there are invalid entries in the TLB.

A lock pointer is used for locking entries into the TLB and is set to entry 0 at reset. A TLB lock operation places the specified translation at the entry designated by the lock pointer, moves the lock pointer to the next sequential entry, and resets the round-robin pointer to entry 31. Locking entries into either TLB effectively reduces the available entries for updating. For example, when the first three entries were locked down, the round-robin pointer is entry 3 after it rolled over from entry 31.

Only entries 0 through 30 are locked in either TLB; entry 31 is never locked. When the lock pointer is at entry 31, a lock operation updates the TLB entry with the translation and ignore the lock. In this case, the round-robin pointer stays at entry 31.

**Figure 3. Example of Locked Entries in TLB**



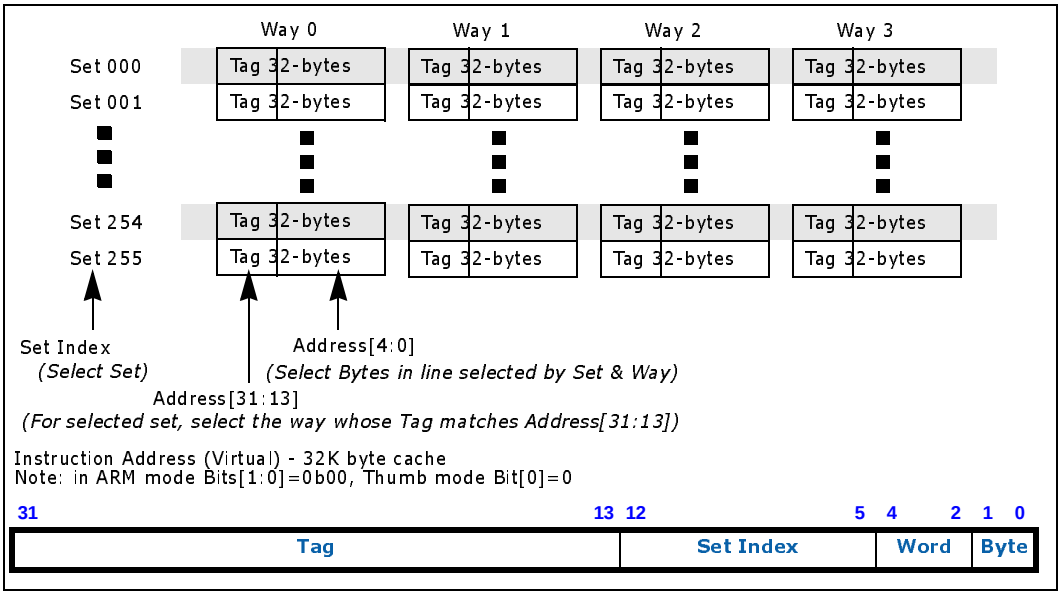
# 4.0 Instruction Cache

3rd generation Intel XScale® microarchitecture (3rd generation microarchitecture or 3rd generation) implements an instruction cache that is in the first level of the memory hierarchy. (This is referred to as Level 1). 3rd generation is configured with a unified level 2 (L2) cache, such that accesses that miss the instruction cache is directed to the L2 cache. The instruction cache enhances performance by reducing the number of instruction fetches from external memory. It also provides software the ability to lock down performance critical code.

## 4.1 Overview

Figure 4 shows cache organization and how the instruction address is used to access the cache.

**Figure 4. Instruction Cache Organization**




The instruction cache is a 32 Kbyte, 4-way set associative cache. There are 256 sets with each set containing four ways. Each way of a set contains eight 32-bit words and one valid bit (or line). The cache supports the ability to lock and unlock data on a line granularity (Section 4.3.4 has more information on locking.) The replacement policy used when all four ways are available (in other words, no lines are locked) is a pseudo-LRU algorithm. See Section 4.2.4 for more details about the replacement algorithm. The instruction cache is virtually addressed.

*Note:* The virtual address presented to the instruction cache is remapped by the PID register, which creates a modified virtual address (MVA). See Section 7.2.13, "Register 13: Process ID" for a description of the PID register.

## 4.2 Operation

### 4.2.1 Operation When Instruction Cache is Enabled

When the cache is enabled, it compares every instruction request address against the addresses of instructions that it is currently holding. When the cache contains the requested instruction and is valid, the access hits the cache, and the cache returns the requested instruction. When the cache does not contain the requested instruction, the access misses the cache, and the cache requests a fetch from backing memory of the 8-word line (32 bytes) that contains the requested instruction. As the fetch returns instructions to the cache, these are placed in one of two fetch buffers and the requested instruction is delivered to the instruction decoder.

A fetched line is written into the cache when it is cacheable and the cache is enabled. Code is designated as cacheable when the Memory Management Unit (MMU) is disabled or when the MMU is enabled and the page referenced is L1 cacheable. See Chapter 3.0, "Memory Management" for a discussion on page attributes.

*Note:* An instruction fetch misses the cache but hit one of the fetch buffers. This happens before a requested line is written into the cache. (See Section 4.2.3 for more details.) When a fetch buffer hit occurs, the requested instruction is delivered to the instruction decoder in the same manner as a cache hit.

### 4.2.2 Operation When Instruction Cache Is Disabled

Disabling the cache prevents any lines from being written into the instruction cache. Although the cache is disabled, it is still accessed and generates a hit when the data is already in the cache.

*Note:* This behavior (hitting the cache when disabled) is deprecated on 3rd generation microarchitecture, meaning it is supported in 3rd generation microarchitecture but not in future microarchitectures. Software must not rely on this feature after 3rd generation microarchitecture.

Disabling the instruction cache ***does not*** disable instruction buffering that occurs within the instruction fetch buffers. Two 8-word instruction fetch buffers are always enabled in the cache disabled mode. So long as instruction fetches continue to hit within either buffer (even in the presence of forward and backward branches), no external fetches for instructions are generated. A miss causes one or the other buffer to be filled from external memory. Note that these fetch buffers are invalidated. (See Section 4.3.3 for more details.)

Enabling the cache, after it has been disabled, does not modify the contents of the cache. For example, when a line is placed into the cache and then it is disabled and then re-enabled, the line is still in the cache.



### 4.2.3 Fetch Policy

An instruction cache miss occurs when the requested instruction is not found in the instruction fetch buffers or instruction cache. A fetch request is then made to the next level of memory (in other words, the L2 or memory external to 3rd generation microarchitecture). The fetch size is always 32 bytes, whether it is cacheable or non-cacheable and the fetch address is always aligned on a 32-byte boundary. The instruction cache handles up to two "misses". Each external fetch request uses a fetch buffer that holds 32 bytes.

A miss causes the following:

1. A fetch buffer is allocated
2. The instruction cache sends a fetch request to the next level of memory. This request is for a 32-byte line.
3. When the line requested is delivered from the L2, all 32 bytes are returned in one transfer. When the line is returned from memory external to 3rd generation microarchitecture, the transfer rate depends on the product configuration. Please refer to the ASSP product architecture specification for more information. As the bytes return, these are written into the fetch buffer.
4. As soon as the fetch buffer receives the requested instruction, it forwards the instruction to the instruction decoder for execution. As other instructions in the requested line return from external memory, these are placed in the fetch buffer. While there, these generate a hit when that instruction address is requested.
5. When all words have returned, the fetched line is written into the instruction cache, when cacheable and when the instruction cache is enabled. The line chosen for update in the cache is controlled by the replacement algorithm (see Section 4.2.4). This update replaces a valid line at that location.
6. Once the cache is updated, the fetch buffer is invalidated.

### 4.2.4 Replacement Algorithm

The line replacement algorithm for a set in the instruction cache is pseudo LRU when there are no lines locked. When one or more lines are locked in a set, the replacement algorithm become true LRU for that set.

Pseudo LRU works by keeping track of which pair of lines in each set was least-recently accessed and within each pair of lines which one was least-recently accessed. This requires three bits per set:

- One to designate either the first two lines (way 0 and way 1) or the last two lines (way 2 and way 3) as the least recently used pair
- Another bit to indicate which of the first two lines is least-recently used with respect to each other
- One more bit to indicate which of the last two lines is least-recently used with respect to each other.

Pseudo LRU doesn't always select the least-recently used line as the next one to replace. Consider the access sequence (in ways) 0-1-2-3-0, the line selected for replacement is line 2, not line 1 as is done in the true LRU algorithm. However, the algorithm does ensure that the line selected for replacement is either the least-recently or the second least-recently used line.

When three or fewer lines in a set are available for replacement (due to some lines being locked), true LRU are used to determine which line gets replaced. True LRU means the least-recently used line accessed in the set (ignoring locked lines) are replaced.

Lines are allocated in the following order after reset or global invalidation (assuming no lines were locked): way 0, way 2, way 1, way 3.

The "invalidate I cache line" function modifies the LRU bits to point to the line that was just invalidated. No other cache functions (Table 43) affect the LRU bits.

### 4.2.5 Parity Protection

The instruction cache is protected by parity to ensure data integrity. Each instruction cache word has one parity bit. (The instruction cache tag is not parity protected.) When a parity error is detected on an instruction cache access, a prefetch abort exception occurs when 3rd generation microarchitecture attempts to execute the instruction. Before servicing the exception (branching to the exception vector), hardware updates the Fault Status Register (Coprocessor 15, register 5). See Section 2.3.6.3, "Prefetch Aborts" on page 38 for the exact encoding.

A software exception handler recovers from an instruction cache parity error. This is accomplished by invalidating the instruction cache and the branch target buffer and then returning to the instruction that caused the prefetch abort exception. A more complex handler chooses to invalidate the specific line that caused the exception and then invalidate the BTB.

When a parity error occurs on an instruction that is locked in the cache, the software exception handler unlocks and invalidates the offending instruction cache line (by using "Invalidate Instruction Cache Line by MVA" function) and then re-lock the line in before it returns to the faulting instruction.

### 4.2.6 Instruction Fetch Latency

The minimum fetch latency for an L1 instruction cache miss, L2 cache hit is 15 cycles.

When the instruction fetch is to memory external to 3rd generation microarchitecture the latency is dependent on the microarchitecture to external memory frequency ratio, system bus bandwidth, system memory, etc., which are all particular to each ASSP.

### 4.2.7 Instruction Cache Coherency

The instruction cache does not detect modification to program memory by stores or actions of other bus masters. Several situations requires program memory modification, such as just-in-time compilation.

The application program is responsible for synchronizing code modification and invalidating the instruction cache and BTB. In general, software must ensure that modified code space is not accessed until modifications and invalidations are completed.

To achieve instruction cache coherence, the cache contents are invalidated after code modification in external memory is complete. Refer to Section 4.3.3, "Invalidating the Instruction Cache" on page 65 for more details on invalidating the instruction cache.

When the instruction cache is not enabled, or code is being written to a non-cacheable region, software must still invalidate the instruction cache, invalidate the BTB and execute a Prefetch Flush before using the newly-written code. This precaution ensures that state associated with the new code is not buffered elsewhere in the processor, such as the fetch buffers or the BTB.

When writing code as data, care must be taken to force it completely out of the L1 data cache and into the L2 cache or external memory before attempting to execute it. When writing into a non-cacheable or write-through region, executing a DWB (Section 7.2.8.3) is sufficient precaution. When writing to a cacheable writeback region, then the data cache is subjected to a Clean/Invalidate operation (see Section 7.2.8.1) to ensure coherency.

## 4.3 Instruction Cache Control

### 4.3.1 Instruction Cache State at Reset

After reset, the instruction cache is always disabled, unlocked, and invalidated.

### 4.3.2 Enabling/Disabling

The instruction cache is enabled by setting bit 12 in coprocessor 15, register 1 (see Section 7.2.2, "Register 1: Control and Auxiliary Control Registers").

### 4.3.3 Invalidating the Instruction Cache

The entire instruction cache along with the fetch buffers are invalidated by writing to coprocessor 15, register 7. (See Table 43 for the exact command.) This command does not unlock any lines that were locked in the instruction cache nor does it invalidate those locked lines. To invalidate the entire cache including locked lines, the unlock instruction cache command needs to be executed before the invalidate command. This unlock command is also found in Table 54.

3rd generation microarchitecture also supports invalidating an individual line in the instruction cache, specified by an MVA. See Table 43 for the exact command. This command also unlocks the entry when it was previously locked.

The Prefetch Flush function, Invalidate Instruction Cache function and Invalidate I Cache Line function invalidates the contents of the fetch buffers.

### 4.3.4 Locking Instructions in the Instruction Cache

Software has the ability to lock performance critical routines into the instruction cache. Lines are locked into the instruction cache by the "Fetch and Lock I Cache Line" function located in coprocessor 15, register 9, see Table 54 for exact command. Register *Rd* contains the modified virtual address of the line locked into the cache.

Lines are only locked in way1, way2 and way3, which means no more than 24K bytes of code is locked in the instruction cache. The "Fetch and Lock I Cache Line" function uses the replacement algorithm to decide which of the three ways (within the specified set) to allocate and lock. Attempting to lock a line in a set with 3 ways already locked result in the line being allocated in way0 and it is not locked.

3rd generation microarchitecture allows software to unlock individual lines or the entire instruction cache. The "Invalidate I Cache Line" function invalidates and unlocks the specified line and the "Unlock Instruction Cache" function unlocks the entire cache. (See Table 46 and Table 54 respectively, for the exact command.)

There are two requirements for locking down code:

1. The code being locked into the cache must be cacheable in the instruction cache.
2. The instruction cache must be enabled and lines targeted to be locked must not already be in the cache.

Failure to follow these requirements produces unpredictable results.

Software locks down several different routines located at different memory locations. This causes some sets to have more locked lines than others (for example, set 2 has way 1 and way 2 locked while set 34 only has way 3 locked).

*Note:* It is possible to receive an exception, known as a lock abort, while locking code (see Section 2.3.6, "Exception Architecture" on page 37).



# 5.0 Branch Target Buffer

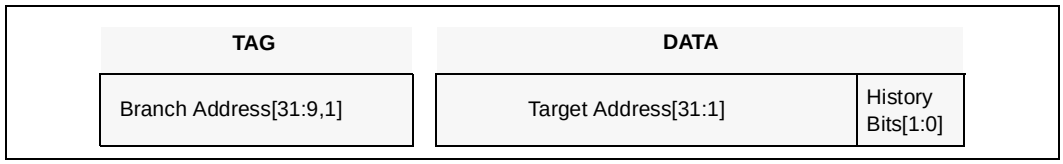
The 3rd generation Intel XScale® microarchitecture (3rd generation microarchitecture or 3rd generation) uses dynamic branch prediction to reduce the penalties associated with changing the flow of program execution. 3rd generation features a branch target buffer that provides the instruction cache with the target address of branch type instructions. The branch target buffer is implemented as a 128-entry, direct mapped cache.

This chapter is primarily intended for those optimizing their code for performance. An understanding of the branch target buffer is needed in this case so that code is scheduled to best utilize the performance benefits of the branch target buffer.

## 5.1 Branch Target Buffer (BTB) Operation

The BTB stores the history of branches that have executed along with their targets. Figure 5 shows an entry in the BTB, where the tag is the instruction address of a previously executed branch and the data contains the target address of the previously executed branch along with two bits of history information.

**Figure 5. BTB Entry Format**




The BTB takes the current instruction address and checks to see when this address is a branch that was previously seen. It uses bits [8:2] of the current address to read out the tag and then compares this tag to bits [31:9,1] of the current instruction address. When the current instruction address matches the tag in the cache and the history bits indicate that this branch has usually been taken in the past, the BTB uses the data (target address) as the next instruction address to send to the instruction cache.

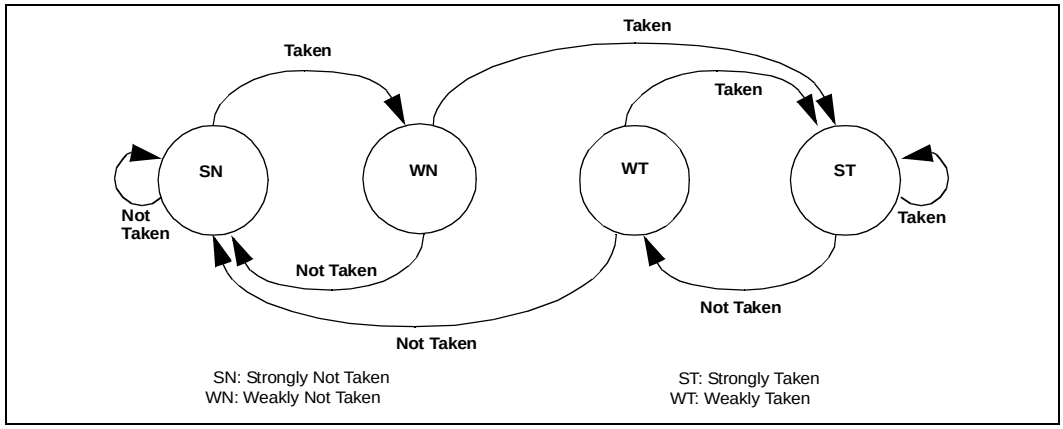
Instruction address Bit[1] is included in the tag comparison for Thumb execution support. This organization means that two consecutive Thumb branch (B) instructions, with instruction address bits[8:2] the same, contend for the same BTB entry. Thumb also requires 31 bits for branch target address. ARM mode = bit[1] is zero.

The history bits represent four possible prediction states for a branch entry in the BTB. Figure 6, "Branch History State Diagram" on page 67 shows these states along with the possible transitions. Every time a branch that exists in the BTB is executed, the history bits are updated to reflect the latest outcome of the branch, either taken or not-taken.

Chapter 13.0, "Performance Considerations" describes instructions that are dynamically predicted by the BTB and the performance penalty for mispredicting a branch.

The BTB is disabled by default following a reset and must be explicitly enabled. Once enabled, the BTB does not generally need to be managed by software; it is automatically invalidated by a global instruction cache invalidation or Process ID Register (Section 7.2.13) changes. However, certain situations require explicit management of the BTB. For example, modifying code in external memory and then invalidating an individual cache line to allow the modified code to execute requires explicit invalidation of the BTB. Section 5.2.2 describes BTB management methods.

**Figure 6. Branch History State Diagram**





### 5.1.1 Reset

After Processor Reset, the BTB is disabled and all entries are invalidated.

### 5.1.2 Update Policy

The following branch instructions update the BTB:

- **B** (ARM and Thumb)
- **BL** (ARM and Thumb)

A new entry is stored into the BTB when the following conditions are met:

- the BTB is enabled
- AND the branch instruction has executed
- AND the branch was taken
- AND the branch is not currently in the BTB.

The entry is then marked valid and the history bits are set to WT. When another valid branch exists at the same entry in the BTB, it is replaced by the new branch.

Once a branch is stored in the BTB, the history bits are updated upon every execution of the branch as shown in Figure 6.

## 5.2 BTB Control

### 5.2.1 Disabling/Enabling

The BTB is disabled following reset. Software enables the BTB by setting a bit in the coprocessor 15 control register (see Section 7.2.2).

### 5.2.2 Invalidation

There are four ways the contents of the BTB are invalidated.

1. The BTB is invalidated by a processor reset.
2. Software directly invalidates the BTB via a CP15, register 7 function. Refer to Section 7.2.8, "Register 7: Cache Functions".
3. The BTB is invalidated by a software write to the Process ID Register.
4. The BTB is invalidated by a global invalidation of the instruction cache via CP15, register 7 functions.



# 6.0 Data Cache

The 3rd generation Intel XScale® microarchitecture (3rd generation microarchitecture or 3rd generation) implements a data cache that is in the first level of the memory hierarchy. (This is referred to as Level 1). 3rd generation is configured with a unified level 2 (L2) cache, such that accesses that miss the data cache are directed to the L2 cache.

The data cache enhances performance by reducing the number of data accesses to and from external memory. The data cache is non-blocking, which means instruction execution proceeds when a data request is not serviced by the data cache. There is a 12 entry data request buffer (referred to as the memory buffer) to further decouple instruction execution from external memory accesses, which increases overall system performance.

# 6.1 Overview

## 6.1.1 Organization

The data cache is a 32 Kbyte, 4-way set associative cache; there are 256 sets with each set containing four ways. Each way of a set contains eight 32-bit words and one valid bit. The line size is 8 words. There also exists a dirty bit for every line; when a store hits the cache for a memory region marked as writeback, the dirty bit associated with that line is set. The cache supports the ability to lock and unlock data on a line granularity. (See Section 6.4 for more information on locking.) The replacement policy used when all four ways are available (in other words, no lines are locked) is a pseudo-LRU algorithm. (More details about the replacement algorithm is found in Section 6.2.4.)

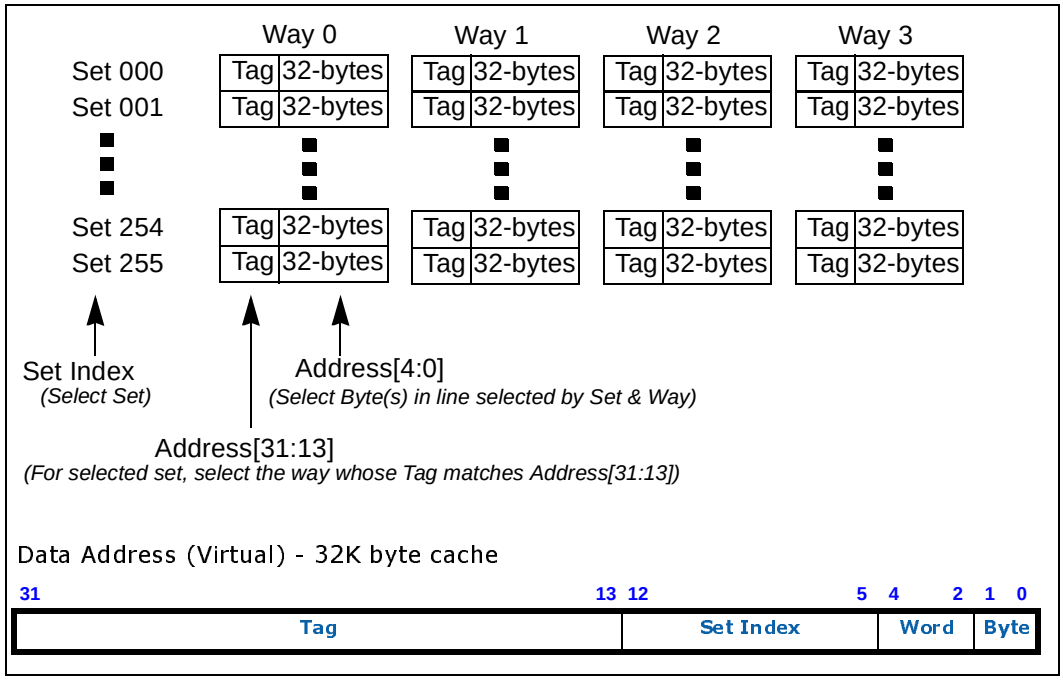
Figure 7, "Data Cache Organization" on page 71 shows the cache organization and how the data address is used to access the cache.

Cache policies are specified by the page attribute bits in the page table descriptors. See Section 3.2.3, "Memory Attributes" on page 47 for a description of these bits.

The data cache is virtually addressed. It supports write-back and write-through caching policies. The data cache only allocates a line in the cache when a cacheable read miss occurs (which includes a **PLD** instruction) or when the line-allocate command is used. Write allocation is not supported in the L1 data cache.

*Note:* The virtual address presented to the data cache is remapped by the PID register, which creates a modified virtual address (MVA). See Section 7.2.13, "Register 13: Process ID" for a description of the PID register.

**Figure 7. Data Cache Organization**





### 6.1.2 Low-Locality of Reference (LLR)

3rd generation microarchitecture provides unique caching for data that has a low temporal locality. This type of data is accessed for only a short period of time and when there is a large amount of data being processed, it pollutes the entire data cache. 3rd generation microarchitecture only allows this type of data to allocate in way 0 of the data cache, thus preserving the contents of other ways of the cache.

LLR caching is enabled through the attribute bits in the page table. (See Section 3.2.3.3, "Low Locality of Reference (LLR)" on page 48 for the exact encoding.) When LLR caching is specified, hardware looks in the Auxiliary Control Register to find out how the LLR data is cached in the data cache and L2 cache. (See the OC and IC fields of Table 34.)

### 6.1.3 Memory Buffer Overview

3rd generation microarchitecture implements a 12-entry memory buffer that holds data requests to the next level of memory. Each entry holds up to 32 bytes of data, all data being contained within a cache line boundary. Each entry generates only one request to external memory (or L2 cache) and each entry holds up to four cacheable load requests when the request is for an address that resides in the same cache line of the fill. In this case, all data requests share this one buffer.

The memory buffer supports the coalescing of multiple store requests to external memory. A store request that is marked as coalescable in the page table coalesces with any one of the 12 entries. The new store request is placed in the same entry as the existing store request when the address of the new store falls in the eight word aligned address of the existing entry. The data is updated in the memory buffer entry and no pend entry is used. (See Section 1.3.2, "Terminology and Acronyms" on page 25 for a definition of coalescing.)

The memory buffer is always enabled which means stores to external memory are buffered. The page attributes TEX[2:0], C, and B are used to defined whether coalescing is enabled for each region of memory. See Section 3.2.4, "Memory Attribute Encodings" on page 49 for more information on these attributes.

Data requests that allocate a new entry in the memory buffer are:

- a data cache line fill, due to a cacheable load, **PLD** instruction, or a swap instruction.
- a cacheable store when configured as write-through and doesn't coalesce with another store already in the memory buffer.
- a cacheable store when configured as writeback that misses the data cache and doesn't coalesce with another store already in the memory buffer.
- a data cache eviction of dirty data.
- a non-cacheable load.
- a non-cacheable store that doesn't coalesce with another store already in the memory buffer.

Data requests that pend against an existing entry in the memory buffer:

- an L1 cacheable load, that misses the data cache, and is for an address that resides in an existing data cache line fill.
- an L1 uncacheable, L2 cacheable load, that is for an address that resides in an existing data cache line fill.

There are several cases where the microarchitecture stalls due to the behavior of the memory buffer. Some of the more visible ones are:

- A cacheable store request that hits an outstanding fill in the memory buffer causes the microarchitecture to stall until the fill is complete.
- A cacheable load request that misses the cache and maps to an outstanding store in the memory buffer causes the microarchitecture to stall until the store is globally observed. See Chapter 10.0, "Memory Ordering" for a definition of globally observed.
- The next memory request causes the memory buffer to overflow meaning all 12 entries are occupied or the memory request coalesces to an entry that already has four pending requests.

### 6.1.3.1 Coalescing

3rd generation microarchitecture allows more opportunities for stores to coalesce in the memory buffer by delaying stores to the next level of memory. The oldest store is held in the memory buffer until one of the following occurs:

- The high-water mark is exceeded. Specifically, there are more than two coalescable store entries in the memory buffer.
- An explicit fence instruction is executed, which includes DMB and DWB. See Chapter 10.0, "Memory Ordering" for a description explicit fence instructions.
- The store part of a **SWP** instruction is written in the memory buffer.
- The time-out counter, associated with the oldest store, expires.The purpose of the time-out counter is to ensure that the oldest store is eventually sent to the next level of memory hierarchy, when none of the previously described events occur.

  The time-out counter counts when an instruction is executed and there is a coalescable store in the memory buffer. The count is associated with the oldest store and after approximately 127 instructions have executed the oldest store is no longer held in the memory buffer.

  After the oldest store is removed from the buffer, the timer starts again (at zero) for the next oldest store.

*Note:* This is not an exhaustive list. The conditions listed above are ones that are most visible to software.

## 6.2 Data Cache Operation

### 6.2.1 Operation When Data Cache is Enabled

When the data cache is enabled for an access, the data cache compares the address of requests against the addresses of data that it is currently holding. When the line containing the address of the request is resident in the cache, the access hits the cache. For a load operation the cache returns the requested data to the destination register and for a store operation the data is stored into the cache. The data associated with the store is also written to the next level of memory when write-through caching is specified for that area of memory. When the cache does not contain the requested data, the access misses the cache, and the sequence of events that follows depends on the configuration of the cache, the configuration of the MMU and the page attributes, which are described in Section 6.2.3.2, "Read Miss Policy" on page 76 and Section 6.2.3.3, "Write Miss Policy" on page 77.

### 6.2.2 Operation When Data Cache is Disabled

3rd generation microarchitecture allows the data cache to be disabled after it is enabled. The data cache management operations (Table 43) work as defined when the cache is disabled. However, when the data cache is accessed when it's disabled with load or store instructions, including the line allocate function (Table 46), the data associated with those accesses has unpredictable values.

When the data cache is re-enabled after it has been disabled, the contents remains since these were prior to it being disabled, as long as it was not accessed while it was disabled.

## 6.2.3 Cache Policies

### 6.2.3.1 Cacheability

Data at a specified address is cacheable given the following:

- the MMU is enabled
- the address is designated as L1 cacheable in the page table (see Chapter 3.0, "Memory Management", for more details)
- the data cache is enabled.

### 6.2.3.2 Read Miss Policy

The following sequence of events occurs when a cacheable (see Section 6.2.3.1, "Cacheability" on page 76) load operation misses the cache:

1. The data memory buffer is checked to see when an outstanding fill request already exists for that line.

   When so, the current request is placed in the same entry and waits until its requested data (being retrieved from the previously requested fill) returns, after which it writes the requested data to the destination register and the operation is complete.

   When there is no outstanding fill request for that line, the current load request is placed in a new memory buffer entry and a 32-byte read request is made to the next level of memory (in other words, L2 or memory external to 3rd generation microarchitecture). When the memory buffer is full, 3rd generation microarchitecture stalls until an entry is available.
2. A line is selected in the cache to receive the 32-bytes of fill data. The line selected is determined by the replacement algorithm. (See Section 6.2.4 for details on the replacement algorithm.)
3. When the data requested by the load is returned from external memory, it is immediately sent to the destination register specified by the load. A system that returns the requested data back first, with respect to the other bytes of the line, obtains the best performance. (This is commonly referred to as critical word first.)
4. After the entire line is returned from external memory it is written into the cache in the previously selected line. The line chosen contains a valid line previously allocated in the cache. In this case the dirty bit is examined and when set, the dirty line is evicted from the cache and written into the memory buffer. From there it is written to the next level of memory as an eight word burst operation.

A load operation that misses the L1 data cache and is not cacheable in the data cache but is cacheable in the L2 cache makes a line request to the L2 cache. The behavior is the same as mentioned above for cacheable misses except that data won't be written into the data cache (step #2 and step #4).

### 6.2.3.3 Write Miss Policy

An L1 data cacheable write miss is placed in the memory buffer and once it is removed from the memory buffer it generates a write request to the next level of memory for the exact data size specified by the store operation, assuming the write request doesn't coalesce with another write operation in the memory buffer.

The L1 data cache never allocates a line on a write miss.

### 6.2.3.4 Write-Back Versus Write-Through

3rd generation microarchitecture supports write-back caching or write-through caching, controlled through the MMU page attributes. When write-through caching is specified, all store operations not only update the data cache (when it hits and is cacheable), but are written to the next level of memory. This feature keeps the next level of memory coherent with the data cache, in other words, no dirty bits are set for this region of memory in the data cache.

*Note:* When shared memory is specified for a region of memory, the data cache defaults to write-through caching when the page table attributes had write-back caching designated and it defaults to uncacheable in the data cache when the memory region is marked as L2 uncacheable, L2 write-through, or there is no L2.

When write-back caching is specified for non-shared memory, a store operation that hits the cache does not generate a write to the next level of memory, thus reducing external memory traffic. It also sets the dirty bit for that line when it isn't already set.



### 6.2.4 Replacement Algorithm

The line replacement algorithm for a set in the data cache is pseudo LRU when there are no lines locked. When one or more lines are locked in a set, the replacement algorithm becomes true LRU for that set.

Pseudo LRU works by keeping track of which pair of lines in each set was least-recently accessed, and within each pair of lines which one was least-recently accessed. This requires three bits per set: one to designate either the first two lines (way 0 and way 1) or the last two lines (way 2 and way 3) as the least recently used pair, another bit to indicate which of the first two lines is least-recently used with respect to each other, and one more bit to indicate which of the last two lines is least-recently used with respect to each other.

Pseudo LRU doesn't always select the least-recently used line as the next one to replace. Consider the access sequence (in ways) 0-1-2-3-0, the line selected for replacement is line 2, not line 1 as is done in the true LRU algorithm. However, the algorithm does ensure that the line selected for replacement is either the least-recently or the second least-recently used line.

When three or fewer lines in a set are available for replacement (due to some lines being locked), true LRU are used to determine which line gets replaced. True LRU means the least-recently used line in the set (ignoring locked lines) gets replaced.

LLR caching always allocates to way 0 of the data cache; this gives the appearance of a direct mapped cache for LLR data. The LRU bits are updated after the allocation to identify way 0 as the most recently used line in the set.

### 6.2.5 Parity Protection

The data cache is protected by parity to ensure data integrity; there is one parity bit per byte of data. The tags are not parity protected. When a parity error is detected on a data cache read access or eviction, a data abort exception occurs. Before servicing the exception, hardware updates the Fault Status Register.

A data cache parity error causes an imprecise data abort, which means R14_ABORT does not point to the instruction that caused the parity error.

A data cache parity error is unrecoverable. For example, when the parity error occurred during a load, the targeted register is updated with incorrect data. Also when the error occurred on a line in the cache that has a writeback caching policy, prior updates to this line is lost.

### 6.2.6 Data Cache Miss Latency

The minimum result latency for load and store instructions that incur a L1 data cache miss, L2 cache hit is 15 cycles. Refer to Section 13.4.1, "Performance Terms" on page 222 for definition of minimum result latency.

## 6.3 Data Cache Control

### 6.3.1 Data Memory State After Reset

After processor reset the data cache is disabled, all valid bits are set to zero (invalid), all lines are unlocked, outstanding requests in the memory buffer are discarded and the replacement algorithm is reset.

### 6.3.2 Enabling/Disabling

The data cache is enabled by setting bit 2 in coprocessor 15, register 1 (Control Register). See Section 7.2.2, "Register 1: Control and Auxiliary Control Registers", for a description of this register.

The MMU must be enabled to use the data cache. Enabling the data cache and not the MMU produces unpredictable results.

### 6.3.3 Invalidate and Clean Operations

Individual entries are invalidated and cleaned in the data cache via coprocessor 15, register 7. Note that a line locked into the data cache is unlocked with invalidate by line functions that use a modified virtual address. Those functions that use set/way do not unlock lines. See Section 7.2.8.6, "Interaction of Cache Functions on Locked Entries" for more details.

This same register also provides the command to invalidate the entire data cache. Refer to Table 43 for a listing of the commands. These global invalidate commands have no effect on lines locked in the data cache. Locked lines must be unlocked before the cache is globally invalidated. This is accomplished by the Unlock Data Cache command found in Table 54.

There is no explicit command for globally cleaning the data cache. Software iterates through the cache using the clean by set/way command. See Table 44 for the proper usage.

## 6.4 Data Cache Locking

Software has the ability to lock lines in the data cache, thus creating the appearance of data RAM. Any subsequent access to this locked line always hits the cache unless it is invalidated. Once a line is locked into the data cache it is no longer available for replacement. Way 0 is not available for locking which means that the maximum locked size is 24 KB for the 32 KB cache.

There are two methods for locking lines into the data cache; method of choice depends on the application. One method is used to lock data that resides in external memory into the data cache, the other method is used to re-configure data cache lines as data RAM. Locking data from external memory into the data cache is useful for lookup tables, constants, and other data that is frequently accessed. Re-configuring a data cache portion as data RAM is useful when an application needs scratch memory (> the register file provides) for frequently used variables, which is strewn across memory, making it advantageous for software to pack these into data RAM memory.

Software maps any area of memory as data RAM. This is accomplished by using the "Data Cache Line Allocate" function. The line-allocate function does validate the target address with the MMU, so system software must ensure the memory has a valid descriptor in the page table that designates the area of memory as L1 cacheable. The 32 bytes of data located in a newly allocated line in the cache must be initialized by software before it is read. The line allocate operation does not initialize the 32 bytes and therefore reading from that line returns unpredictable values.

*Note:* The Data Cache Line Allocate function is deprecated on 3rd generation microarchitecture.

Lines are locked into the data cache by enabling the data cache lock mode bit located in coprocessor 15, register 9. (See Table 54 for the exact command.) Once enabled, any new lines allocated into the data cache are locked down.

To avoid undesirable locking behavior, software must use the Data Cache locking routine provided in the *3rd Generation Intel XScale® Microarchitecture Software Design Guide*. Any deviation from this routine result in unpredictable locking behavior. The *3rd Generation Intel XScale® Microarchitecture Software Design Guide* provides additional information on issues which the programmer must handle in their code.

Lines are only locked in way 1, way 2 and way 3, which means no more than 24 KB of data is locked in the data cache. The replacement algorithm is used to decide which of the three ways (within the specified set) to allocate and lock. Attempting to lock a line in a set with three ways already locked result in the line being allocated in way 0 and it is not locked.

Software locks down data sections located at different memory locations. This causes some sets to have more locked lines than others (for example, set 2 has way 1 and way 2 locked while set 34 has only way 3 locked).

3rd generation microarchitecture allows software to unlock individual lines or all the lines locked in the data cache. See Section 7.2.10, "Register 9: Cache Lock Down" and Table 47 for more information about locking and unlocking the data cache.

Before locking, the programmer must ensure that no part of the target data range is already resident in the cache. 3rd generation microarchitecture does not re-fetch such data, which results in it not being locked into the cache. When there is any doubt as to the location of the targeted memory data, clean the cache and invalidated to prevent this scenario. When the cache contains a locked region which the programmer wishes to lock again, then the cache must be unlocked before being cleaned and invalidated.

Attempting to lock data from a memory region marked as shared or LLR in the page tables, produces unpredictable results.

## 6.5 Memory Buffer Operation and Control

See Section 1.3.2, "Terminology and Acronyms" on page 25 for a definition of coalescing.

The memory buffer is always enabled. This means all writes to the next level of memory (L2 or memory external to 3rd generation microarchitecture) are buffered. See Section 6.1.3 for more details on the memory buffer.

The page attributes TEX[2:0], C, and B are examined to see when coalescing is enabled for each region of memory.

Software explicitly drains all buffered writes. For details on this operation, see the description of Data Write Barrier in Section 7.2.8, "Register 7: Cache Functions".

## 6.6 Memory Ordering

3rd generation microarchitecture implements a weakly ordered memory model, which means memory operations are reordered by the microarchitecture and explicit memory barrier instructions are required to keep program order when that is the desired effect. Refer to Chapter 10.0, "Memory Ordering"for more details.

## 6.7 Data Cache Coherency

The data cache provides hardware coherency for regions of memory that are marked as shared in the page table. It uses a Valid/Invalid protocol to maintain coherency. For more details, refer to Chapter 9.0, "Cache Coherence".

The data cache never operates in writeback mode when shared memory is referenced. It either defaults to write-through or non-cacheable, depending on the page table attributes. Refer to Chapter 3.0, "Memory Management" for more details.

3rd generation microarchitecture does not support hardware cache coherency when the L1 data cache is disabled.



# 7.0 Configuration

This chapter describes the internal co-processors within the 3rd generation Intel XScale® microarchitecture (3rd generation microarchitecture or 3rd generation). These internal co-processors include the System Control Co-processor (CP15), Co-processor 14 (CP14) and a portion of Co-processor 7 (CP7).

## 7.1 Overview

CP7, CP14 and CP15 contain internal co-processor registers used to configure various 3rd generation microarchitecture parts.

CP15 is used to configure the 3rd generation microarchitecture MMU, caches, buffers and other system attributes. CP14 contains the 3rd generation microarchitecture performance monitor registers, clock and power management registers and the debug registers.

In CP7, a portion of the co-processor registers are defined by 3rd generation microarchitecture. These 3rd generation microarchitecture CP7 co-processor registers are used for error logging for the L2 cache and BIU. The remaining registers in CP7 are defined as part of an ASSP specific co-processor.

Through the remainder of this chapter, references to CP7 only refer to the 3rd generation microarchitecture CP7 co-processor registers, unless otherwise noted.

For a description of any ASSP specific co-processors which are defined, refer to the relevant product documentation.

Table 25 shows the accessibility of each of the co-processor instructions to CP7, CP14 and CP15 co-processor registers.

**Table 25. Co-processor Instruction Accessibility to CP7, CP14 and CP15**

| CP# | MCR/MRC | CDP | LDC/STC | MCRR/MRRC | MRC2/MCR2 | CDP2 | LDC2/STC2 | MCRR2/MRRC2 |
|---|---|---|---|---|---|---|---|---|
| CP7[a] | Allowed | UNDEF[b] | N/A | N/A | UNDEF | UNDEF | N/A | N/A |
| CP14 | Allowed | UNDEF | Varies[c] | UNDEF | UNDEF | UNDEF | UNDEF | UNDEF |
| CP15 | Allowed | UNDEF | UNDEF | UNDEF | UNDEF | UNDEF | UNDEF | UNDEF |

a. CP7 registers are only accessed with instructions that are capable of specifying CRn and CRm. "N/A" indicates these instructions cannot be used to access the CP7 registers.
b. "UNDEF" indicates an undefined instruction exception is generated.
c. LDC/STC are only used to access CP14 registers for which CRm = 0.

There are four CP15 functions allowed in user mode (see Section 7.2.8, "Register 7: Cache Functions" on page 96); all other CP15 functions and registers must be accessed from privileged modes. Access to CP14 and CP7 registers is allowed only in privileged modes. Any access to CP7, CP14 or privileged CP15 co-processor registers in user mode causes an undefined instruction exception.

3rd generation microarchitecture includes an extra level of virtual address translation in the form of a Process ID (PID). For a detailed description of this facility, see Section 7.2.13, "Register 13: Process ID" on page 105. Privileged mode software must be aware of this facility when accessing CP15 because some addresses are modified by the PID and others are not. An address that has yet to be modified by the PID ("PIDified") is known as a *virtual address* (VA). An address that has been through the PID logic, but not translated into a physical address, is a *modified virtual address* (MVA).

## 7.2 CP15 Registers

Table 26 lists the CP15 registers implemented in 3rd generation microarchitecture.

**Table 26. CP15 Registers**

| Register (CRn) | Opc_1 | CRm | Opc_2 | Access | Description | Cross-Reference |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | Read / Write-Ignored | Main ID | Section 7.2.1, page 85 |
| 0 | 0 | 0 | 1 | Read / Write-Ignored | L1 Cache Type | |
| 0 | 1 | 0 | 0 | Read / Write-Ignored | L2 System ID | |
| 0 | 1 | 0 | 1 | Read / Write-Ignored | L2 Cache Type | |
| 1 | 0 | 0 | 0 | Read / Write | Control | Section 7.2.2, page 88 |
| 1 | 0 | 0 | 1 | Read / Write | Auxiliary Control | |
| 2 | 0 | 0 | 0 | Read / Write | Translation Table Base | Section 7.2.3, page 91 |
| 3 | 0 | 0 | 0 | Read / Write | Domain Access Control | Section 7.2.4, page 92 |
| 4 | - | - | - | Unpredictable | Reserved | |
| 5 | 0 | 0 | 0 | Read / Write | Fault Status | Section 7.2.6, page 93 |
| 6 | 0 | 0 | 0 | Read / Write | Fault Address | Section 7.2.7, page 95 |
| 7 | Varies[a] | Varies[a] | Varies[a] | Read-unpredictable / Write | Cache Operations | Section 7.2.8, page 96 |
| 8 | 0 | Varies[a] | Varies[a] | Read-unpredictable / Write | TLB Operations | Section 7.2.9, page 101 |
| 9 | Varies[a] | Varies[a] | Varies[a] | Varies[a] | Cache Lock Down | Section 7.2.10, page 102 |
| 10 | 0 | Varies[a] | Varies[a] | Read-unpredictable / Write | TLB Lock Down | Section 7.2.11, page 104 |
| 11 - 12 | - | - | - | Unpredictable | Reserved | |
| 13 | 0 | 0 | 0 | Read / Write | Process ID (PID) | Section 7.2.13, page 105 |
| 14 | 0 | Varies[a] | 0 | Read / Write | Breakpoint Registers | Section 7.2.14, page 106 |
| 15 | 0 | 1 | 0 | Read / Write | Co-processor Access | Section 7.2.15, page 107 |

a. The value varies depending on the specified function. Refer to the register description for a list of values.

## 7.2.1 Register 0: ID & Cache Type Registers

### Table 27. Register 0 Functions (CRn=0)

| Function | Opc_1 | CRm | Opc_2 | Instruction |
|---|---|---|---|---|
| Main ID Register (ID) | 0b0000 | 0b0000 | 0b000 | MRC p15, 0, Rd, c0, c0, 0 |
| L1 Cache Type Register (CTYPE) | 0b0000 | 0b0000 | 0b001 | MRC p15, 0, Rd, c0, c0, 1 |
| L2 System ID Register (L2ID) | 0b0001 | 0b0000 | 0b000 | MRC p15, 1, Rd, c0, c0, 0 |
| L2 Cache Type Register (L2CTYPE) | 0b0001 | 0b0000 | 0b001 | MRC p15, 1, Rd, c0, c0, 1 |

Register 0 houses four read-only registers that are used for part identification: the Main ID Register, the L1 Cache Type Register, the L2 System ID Register and the L2 Cache Type Register.

These registers are only readable from privileged modes. User mode access results in an undefined instruction exception.

The Main ID Register returns a code for the target product. A portion of the code is defined by the ASSP. Refer to the 3rd generation microarchitecture implementation options section of the relevant product documentation for the exact encoding.

### Table 28. Main ID Register

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | Revision | | | Product Number | | | | | | Product Revision | | | |

reset value: As Shown

| Bits | Access | Description |
|---|---|---|
| 31:24 | Read / Write Ignored | Implementation trademark<br>0x69: 'i'= Intel Corporation |
| 23:16 | Read / Write Ignored | Architecture version<br>0x05: *ARM Architecture Version 5TE Specification* |
| 15:13 | Read / Write Ignored | Microarchitecture Generation<br>0b011 = 3rd generation microarchitecture<br>This field reflects a specific set of architecture features supported by the microarchitecture. When new features are added/deleted/modified this field changes. This allows software that is not dependent on ASSP features to target code at a specific microarchitecture generation. |
| 12:10 | Read / Write Ignored | Microarchitecture Revision:<br>This field reflects revisions of microarchitecture generations. Differences include errata that dictate different operating conditions, software work-around, etc. |
| 9:4 | Read / Write Ignored | Product Number<br>Defined by the ASSP |
| 3:0 | Read / Write Ignored | Product Revision<br>Defined by the ASSP |

In the L2 System ID Register, only the implementation trademark field is valid. The rest of the bits are reserved and returns unpredictable values when read.

**Table 29. L2 System ID Register**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | Reserved | | | | | | | | | | | | | | | | | | | | | | | |

reset value: As Shown

| Bits | Access | Description |
|---|---|---|
| 31:24 | Read / Write Ignored | Implementation trademark<br>0x69: 'i'= Intel Corporation |
| 23:0 | Read-unpredictable / Write-unpredictable | Reserved |

The L1 Cache Type Register and the L2 Cache Type Register describe the configuration of the 3rd generation microarchitecture L1 and L2 caches, respectively.

**Table 30. L1 Cache Type Register**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |

reset value: As Shown

| Bits | Access | Description |
|---|---|---|
| 31:29 | Read-as-Zero / Write Ignored | Reserved |
| 28:25 | Read / Write Ignored | Cache Class<br>0b0101: The caches support locking, write back and clean by Register 7 operations. |
| 24 | Read / Write Ignored | Harvard Cache |
| 23:21 | Read-as-Zero / Write Ignored | Reserved |
| 20:18 | Read / Write Ignored | Data Cache Size<br>0b110: 32 KB |
| 17:15 | Read / Write Ignored | Data Cache Associativity<br>0b010: 4-way |
| 14 | Read-as-Zero / Write Ignored | Reserved |
| 13:12 | Read / Write Ignored | Data Cache Line Length<br>0b10: 32 bytes/line |
| 11:9 | Read-as-Zero / Write Ignored | Reserved |
| 8:6 | Read / Write Ignored | Instruction Cache Size<br>0b110: 32 KB |
| 5:3 | Read / Write Ignored | Instruction Cache Associativity<br>0b010: 4-way |
| 2 | Read-as-Zero / Write Ignored | Reserved |
| 1:0 | Read / Write Ignored | Instruction Cache Line Length<br>0b10: 32 bytes/line |

For the L2 Cache Type Register, note that the bits 23:12 and bits 11:0 are duplicated. Bits 23:12 are defined as the data cache configuration and bits 11:0 are defined as the instruction cache configuration. However, since 3rd generation microarchitecture implements a unified L2 cache the information in bits 23:12 is required to be duplicated in both fields.

**Table 31. L2 Cache Type Register**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | Way Size | | | | Associativity | | | | | 0 | 0 | 0 | Way Size | | | | Associativity | | | | | 0 | 0 | 0 |
| reset value: As Shown | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

| Bits | Access | Description |
|---|---|---|
| 31:29 | Read-as-Zero / Write Ignored | Reserved |
| 28:25 | Read / Write Ignored | Cache Class<br>0b0101: The caches support locking, write back and clean by Register 7 operations. |
| 24 | Read / Write Ignored | Unified Cache |
| 23:20 | Read / Write Ignored | L2 Unified Cache Way Size<br>0b0010: 32KB<br>0b0011: 64KB |
| 19:15 | Read / Write Ignored | L2 Unified Cache Associativity<br>0b00000: L2 not present<br>0b01000: 8-way |
| 14 | Read-as-Zero / Write Ignored | Reserved |
| 13:12 | Read / Write Ignored | L2 Unified Cache Line Length<br>0b00: 32 bytes/line |
| 11:8 | Read / Write Ignored | L2 Unified Cache Way Size<br>0b0010: 32KB<br>0b0011: 64KB |
| 7:3 | Read / Write Ignored | L2 Unified Cache Associativity<br>0b00000: L2 not present<br>0b01000: 8-way |
| 2 | Read-as-Zero / Write Ignored | Reserved |
| 1:0 | Read / Write Ignored | L2 Unified Cache Line Length<br>0b00: 32 bytes/line |

### 7.2.2 Register 1: Control and Auxiliary Control Registers

**Table 32. Register 1 Functions (CRn=1)**

| Function | Opc_1 | CRm | Opc_2 | Instruction |
|---|---|---|---|---|
| Control Register (CTRL) | 0b000 | 0b0000 | 0b000 | MRC p15, 0, Rd, c1, c0, 0<br>MCR p15, 0, Rd, c1, c0, 0 |
| Auxiliary Control Register (AUXCTRL) | 0b000 | 0b0000 | 0b001 | MRC p15, 0, Rd, c1, c0, 1<br>MCR p15, 0, Rd, c1, c0, 1 |

Register 1 is made up of two registers; the Control Register is specified by the ARM Architecture; the Auxiliary Control Register is defined by 3rd generation microarchitecture.

These registers are only accessible from privileged modes. User mode access results in an undefined instruction exception.

The Exception Vector Relocation bit (bit 13 of the Control Register) allows the virtual address of the exception vectors to be mapped into high memory (starting at 0xffff0000) rather than their default location starting at address 0. This is useful for an application that uses the PID (see Section 7.2.13, "Register 13: Process ID" on page 105). Relocating the vector table to high memory prevents the vector address from being remapped via the usual translation mechanism involving the PID.

The L2 Unified Cache Enable bit (bit 26 of the Control Register) allows software to enable the L2 Cache. Following reset, the L2 Unified Cache Enable bit is cleared (in other words L2 Cache is disabled). To enable the L2 Cache, software sets this bit to a '1' before or at the same time as enabling the MMU. Enabling the L2 Cache after the MMU has been enabled or disabling the L2 Cache after the L2 Cache has been enabled, results in unpredictable behavior of the processor.

The definition of all other bits in the Control Register are found in the *ARM Architecture Version 5TE Specification*. Refer to the *3rd Generation Intel XScale® Microarchitecture Software Design Guide* for the proper method of programming the Control Register.

## Table 33. Control Register

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | L2 | | | | | | | | | | | | | V | I | Z | 0 | R | S | B | 1 | 1 | 1 | 1 | C | A | M |

**reset value: writable bits set to 0**

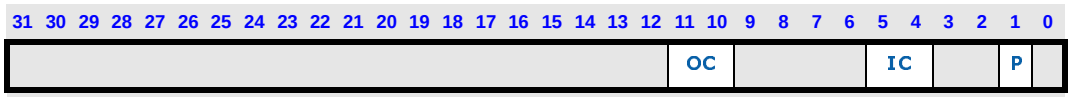
| Bits | Access | Description |
|---|---|---|
| 31:27 | Read-Unpredictable / Write-as-Zero | Reserved |
| 26 | Read / Write | L2 Unified Cache Enable (L2)<br>0 = Disabled<br>1 = Enabled |
| 25:14 | Read-Unpredictable / Write-as-Zero | Reserved |
| 13 | Read / Write | Exception Vector Relocation (V).<br>0 = Base address of exception vectors is 0x0000,0000<br>1 = Base address of exception vectors is 0xFFFF,0000 |
| 12 | Read / Write | Instruction Cache Enable (I)<br>0 = Disabled<br>1 = Enabled |
| 11 | Read / Write | Branch Target Buffer Enable (Z)<br>0 = Disabled<br>1 = Enabled |
| 10 | Read-as-Zero / Write-as-Zero | 0 |
| 9 | Read / Write | **ROM Protection (R)**<br>This selects the access checks performed by the memory management unit. See the *ARM Architecture Version 5TE Specification* for more information. |
| 8 | Read / Write | **System Protection (S)**<br>This selects the access checks performed by the memory management unit. See the *ARM Architecture Version 5TE Specification* for more information. |
| 7 | Read / Write | Big Endian Enable (B)<br>0 = Little-endian operation<br>1 = Big-endian operation |
| 6:3 | Read-as-One / Write-as-One | 0b1111 |
| 2 | Read / Write | Data Cache Enable (C)<br>0 = Disabled<br>1 = Enabled |
| 1 | Read / Write | Alignment Fault Enable (A)<br>0 = Disabled<br>1 = Enabled |
| 0 | Read / Write | Memory Management Unit Enable (M)<br>0 = Disabled<br>1 = Enabled |

The Auxiliary Control Register contains the cache attribute bits for the L1 data cache and L2 cache for Low-Locality of Reference (LLR) memory regions. (See Section 6.1.2, "Low-Locality of Reference (LLR)" on page 72 for more details on LLR.). This register also contains a bit which allows an ASSP defined memory attribute to be applied to translation table walks.

The configuration of LLR cache attributes are setup before any data access is made that is cached in the L1 data cache or L2 cache. Once data is cached, software must ensure that the L1 data cache and L2 cache have been cleaned and invalidated before the LLR cache attributes are changed. Software must also invalidate the ITLB and DTLB.

The Page Table Memory Attribute (P) bit allows an ASSP defined attribute to be applied for memory requests generated by the hardware when doing a translation table walk. Example behavior is enforcing ECC (error correction) on the memory access. Hardware logically OR this bit with Translation Table Base Register P bit. The P bit in the Auxiliary Control Register is deprecated on 3rd generation microarchitecture; the page table memory attribute is programmed through the Translation Table Base Register.

**Table 34. Auxiliary Control Register**




reset value: writable bits set to 0

| Bits | Access | Description |
|---|---|---|
| 31:12 | Read-Unpredictable / Write-as-Zero | Reserved |
| 11:10 | Read / Write | LLR Outer Cache Attributes (OC)<br>0b00 = Outer Non-cacheable<br>0b01 = Outer Write back, Write allocate<br>0b10 = Reserved<br>0b11 = Reserved |
| 9:6 | Read-Unpredictable / Write-as-Zero | Reserved |
| 5:4 | Read / Write | LLR Inner (Data) Cache Attributes (IC)<br>All configurations of LLR caching are cacheable, stores are buffered in the write buffer and stores coalesce in the write buffer.<br>Mapping LLR caching to shared memory changes the definition of this field on 3rd generation microarchitecture." See Chapter 3.0, "Memory Management" for details.<br>0b00 = Inner Write back, Read allocate<br>0b01 = Inner Write back, Read allocate<br>0b10 = Inner Write through, Read allocate<br>0b11 = Inner Write back, Read allocate |
| 3:2 | Read-Unpredictable / Write-as-Zero | Reserved |
| 1 | Read / Write | Page Table Memory Attribute (P)<br>Hardware logically OR the value of this bit with TTBASE.P. The P bit in the Auxiliary Control Register is deprecated on 3rd generation microarchitecture.<br>The effect of this bit is defined by the ASSP. Refer to the 3rd generation microarchitecture implementation options section of the relevant product documentation for more information.<br>0 = ASSP attribute not applied during page table access<br>1 = ASSP attribute is applied during page table access |
| 0 | Read-Unpredictable / Write-as-Zero | Reserved |

## 7.2.3 Register 2: Translation Table Base Register

### Table 35. Register 2 Functions (CRn=2)

| Function | Opc_1 | CRm | Opc_2 | Instruction |
|---|---|---|---|---|
| Translation Table Base Register (TTBASE) | 0b000 | 0b0000 | 0b000 | MRC p15, 0, Rd, c2, c0, 0<br>MCR p15, 0, Rd, c2, c0, 0 |

The Translation Table Base Register specifies the location of the first level translation table, as well as, memory attributes used while accessing the table. This register is only accessible from privileged modes. User mode access results in an undefined instruction exception.

The Translation Table Base field specifies the physical address of the first level page table used when the MMU is enabled. The first level page table must be aligned to a 16KB boundary.

The Table Walk Outer Cache Attributes (OC) field controls the L2 cacheability of the table walk. When the L2 cache is present and enabled and the OC field is programmed to make the table walk L2 cacheable, page table descriptors loaded during a table walk are cached in the L2. When the target descriptor is already cached in the L2, the table walk hits in the L2. On a miss, the table walk loads an entire cache line (8 descriptors) into the L2. When this field indicates L2 non-cacheable, table walks do not cache descriptors in the L2 cache and read page table descriptors directly from main memory.

The Table Walk Memory Attribute (P) bit allows an ASSP to define specific behavior for memory requests generated by the hardware when doing a translation table walk. Example behavior is enforcing ECC (error correction) on the memory access. Hardware logically OR this bit with Auxiliary Control Register P bit. The P bit in the Auxiliary Control Register is deprecated on 3rd generation microarchitecture; the page table memory attribute is programmed through this register.

### Table 36. Translation Table Base Register

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Translation Table Base | | | | | | | | | | | | | | | | | | | | | | | | | | | OC | | P | | |

reset value: unpredictable

| Bits | Access | Description |
|---|---|---|
| 31:14 | Read / Write | Translation Table Base<br>Physical address of the base of the first-level table |
| 13:5 | Read-unpredictable / Write-as-Zero | Reserved |
| 4:3 | Read / Write | Table Walk Outer Cache Attributes (OC)<br>0b00 = Outer Non-cacheable<br>0b01 = Reserved<br>0b10 = Outer Non-cacheable0b11 = Outer Write back |
| 2 | Read / Write | Table Walk Memory Attribute (P)<br>Hardware logically OR the value of this bit with the Auxiliary Control Register P bit. The P bit in the Auxiliary Control Register is deprecated on 3rd generation microarchitecture.<br>The effect of this bit is defined by the ASSP. Refer to the 3rd generation microarchitecture implementation options section of the relevant product documentation for more information.<br>0 = ASSP attribute not applied during page table access<br>1 = ASSP attribute is applied during page table access |
| 1:0 | Read-unpredictable / Write-as-Zero | Reserved |

## 7.2.4 Register 3: Domain Access Control Register

**Table 37. Register 3 Functions (CRn=3)**

| Function | Opc_1 | CRm | Opc_2 | Instruction |
|---|---|---|---|---|
| Domain Access Control Register (DACR) | 0b000 | 0b0000 | 0b000 | MRC p15, 0, Rd, c3, c0, 0<br>MCR p15, 0, Rd, c3, c0, 0 |

The ARM Architecture supports 16 domains. Each domain is a collection of sections and pages that share common access permissions. The Domain Access Control Register specifies the access permissions for each of the 16 domains. Refer to the *ARM Architecture Version 5TE Specification* for more information on domains.

This register is only accessible from privileged modes. User mode access results in an undefined instruction exception.

**Table 38. Domain Access Control Register**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| D15 | | D14 | | D13 | | D12 | | D11 | | D10 | | D9 | | D8 | | D7 | | D6 | | D5 | | D4 | | D3 | | D2 | | D1 | | D0 | |

reset value: unpredictable

| Bits | Access | Description |
|---|---|---|
| 31:0 | Read / Write | Access permissions for all 16 domains<br>The meaning of each field is found in the *ARM Architecture Version 5TE Specification*. |

### 7.2.5 Register 4: Reserved

Register 4 is reserved. Reading and writing this register yields unpredictable results.

### 7.2.6 Register 5: Fault Status Register

**Table 39. Register 5 Functions (CRn=5)**

| Function | Opc_1 | CRm | Opc_2 | Instruction |
|---|---|---|---|---|
| Fault Status Register (FSR) | 0b000 | 0b0000 | 0b000 | MRC p15, 0, Rd, c5, c0, 0<br>MCR p15, 0, Rd, c5, c0, 0 |

3rd generation microarchitecture updates the Fault Status Register (FSR) when a prefetch abort or data abort occurs. The data abort and prefetch abort handlers then use the FSR value to determine the specific type of abort reported.

This register is only accessible from privileged modes. User mode access results in an undefined instruction exception.

The ARM Architecture defines the encoding of the Domain and Status field for MMU generated data aborts. The Status Field Extension (X) bit extends the encoding of the status field for include prefetch aborts and additional types of data aborts. The ARM Architecture encodings and extended 3rd generation microarchitecture encodings are found in Section 2.3.6, "Exception Architecture" on page 37

The Debug Event (D) bit indicates when a debug exception has occurred. The exact source of the debug exception is found in the Debug Control and Status Register (see Section 7.3.3, "Software Debug Registers" on page 110). When bit 9 is set, the domain and extended status field are unpredictable.

Upon entry into the prefetch abort or data abort handler, this register is updated with the source of the exception. Software is not required to clear these fields.

**Table 40. Fault Status Register**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | | | | | | X | D | 0 | Domain | | | | Status | | | |

reset value: unpredictable

| Bits | Access | Description |
|---|---|---|
| 31:11 | Read-unpredictable / Write-as-Zero | Reserved |
| 10 | Read / Write | Status Field Extension (X)<br>This bit extends the encoding of the Status field for prefetch aborts and certain types of data aborts. The encoding of this field is found in Section 2.3.6, "Exception Architecture" on page 37 |
| 9 | Read / Write | Debug Event (D)<br>This bit indicates a debug event has occurred. The cause of the debug event is found in the MOE field of the Debug Control and Status Register (Section 12.3.2, "Debug Control and Status Register (DCSR)") |
| 8 | Read-as-zero / Write-as-Zero | 0 |
| 7:4 | Read / Write | Domain<br>Specifies which of the 16 domains was being accessed when a data abort occurred |
| 3:0 | Read / Write | Status<br>Type of prefetch or data abort that occurred. The encoding of this field is found in Section 2.3.6, "Exception Architecture" on page 37 |

## 7.2.7 Register 6: Fault address Register

**Table 41. Register 6 Functions (CRn=6)**

| Function | Opc_1 | CRm | Opc_2 | Instruction |
|---|---|---|---|---|
| Fault Address Register (FAR) | 0b000 | 0b0000 | 0b000 | MRC p15, 0, Rd, c6, c0, 0<br>MCR p15, 0, Rd, c6, c0, 0 |

The Fault Address Register (FAR) indicates the MVA of the data access that caused the previous data abort.

This register is only accessible from privileged modes. User mode access results in an undefined instruction exception.

The FAR is only valid for certain causes of data aborts. The specific types of aborts which update the FAR are found in Section 2.3.6, "Exception Architecture" on page 37.

Upon entry into the data abort handler, this register is updated with the source of the exception. Software is not required to clear these fields.

**Table 42. Fault Address Register**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Fault Virtual Address | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| reset value: unpredictable | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

| Bits | Access | Description |
|---|---|---|
| 31:0 | Read / Write | Fault Virtual Address<br>Contains the MVA of the data access that caused the data abort |

## 7.2.8 Register 7: Cache Functions

Register 7 contains functions for managing the instruction cache, data cache, L2 cache and branch target buffer (BTB). It also provides explicit memory barrier functions. Register 7 is accessed only with **MCR**, using **MRC** produces unpredictable results. Writing to register 7 with opc_1, CRm and opc_2 values other than those specified in the following tables produces unpredictable results.

### 7.2.8.1 Level 1 Cache and BTB Functions

Table 43 lists out the functions for controlling the instruction cache, data cache and BTB. These functions are only allowed in privileged mode; accessing these functions in user mode generates an undefined instruction exception.

These functions do not cause a page translation nor do these check permissions on the MVA, which means no precise data aborts are reported.

The invalidate instruction cache line command does not invalidate the BTB. When software invalidates a line from the instruction cache and modifies the same location in external memory, it must also invalidate the BTB. Failure to invalidate the BTB in this case causes unpredictable results.

All operations defined in Table 43 work regardless of whether the cache is enabled or disabled. When a function that operates on a line by MVA misses the cache it has no effect on the cache. When any clean function hits a line that is not dirty it also has no effect on the cache. The instruction cache functions work whether the MMU is enabled or disabled. The data cache functions only work when the MMU is enabled, and are unpredictable when the MMU is disabled.

**Table 43. L1 Cache Functions**

| Function | Opc_1 | CRm | Opc_2 | Data | Instruction |
|---|---|---|---|---|---|
| Invalidate I cache & BTB | 0b000 | 0b0101 | 0b000 | Ignored | MCR p15, 0, Rd, c7, c5, 0 |
| Invalidate I cache line | 0b000 | 0b0101 | 0b001 | MVA | MCR p15, 0, Rd, c7, c5, 1 |
| Invalidate BTB | 0b000 | 0b0101 | 0b110 | Ignored | MCR p15, 0, Rd, c7, c5, 6 |
| Invalidate D cache | 0b000 | 0b0110 | 0b000 | Ignored | MCR p15, 0, Rd, c7, c6, 0 |
| Invalidate D cache line | 0b000 | 0b0110 | 0b001 | MVA | MCR p15, 0, Rd, c7, c6, 1 |
| Invalidate I&D cache & BTB | 0b000 | 0b0111 | 0b000 | Ignored | MCR p15, 0, Rd, c7, c7, 0 |
| Clean D cache line | 0b000 | 0b1010 | 0b001 | MVA | MCR p15, 0, Rd, c7, c10, 1 |
| Clean D cache line | 0b000 | 0b1010 | 0b010 | set / way[a] | MCR p15, 0, Rd, c7, c10, 2 |
| Clean & Invalidate Dcache Line | 0b000 | 0b1110 | 0b001 | MVA | MCR p15, 0, Rd, c7, c14, 1 |
| Clean & Invalidate Dcache Line | 0b000 | 0b1110 | 0b010 | set / way[a] | MCR p15, 0, Rd, c7, c14, 2 |

a. Refer to Section 7.2.8.7, page 100 for details on the set/way format.

### 7.2.8.2 Level 2 Cache Functions

Table 44 lists out the functions for controlling the L2 cache. Refer to Chapter 8.0, "Level 2 Unified Cache (L2)" for more information on the Level 2 cache on 3rd generation microarchitecture.

The L2 cache functions are only allowed in privileged mode; user mode access generates an undefined instruction exception.

Functions which use an MVA cause a virtual to physical address page translation and generates data aborts. Refer to Section 7.2.8.5, "Precise Data Aborts" on page 99 for more information about the types of data abort generated.

**Table 44. L2 Cache Functions**

| Function | Opc_1 | CRm | Opc_2 | Data | Instruction |
|---|---|---|---|---|---|
| Invalidate L2 Cache Line | 0b001 | 0b0111 | 0b001 | MVA | MCR p15, 1, Rd, c7, c7, 1 |
| Clean L2 Cache Line | 0b001 | 0b1011 | 0b001 | MVA | MCR p15, 1, Rd, c7, c11, 1 |
| Clean L2 Cache Line | 0b001 | 0b1011 | 0b010 | set / way[a] | MCR p15, 1, Rd, c7, c11, 2 |
| Clean & Invalidate L2 Cache Line | 0b001 | 0b1111 | 0b010 | set / way[a] | MCR p15, 1, Rd, c7, c15, 2 |

a. Refer to Section 7.2.8.7, page 100 for details on the set/way format.

### 7.2.8.3 Explicit Memory Barriers

3rd generation microarchitecture provides three explicit memory barrier functions. These functions allow software to restrict the order in which certain types of memory accesses complete, before and after the functions. The type of memory accesses affected by the instruction depends on the function. These functions are described in detail in Chapter 10.0, "Memory Ordering".

These functions are available in user and privileged modes.

**Table 45. Explicit Memory Barrier Operations**

| Function | Opc_1 | CRm | Opc_2 | Data | Instruction |
|---|---|---|---|---|---|
| Prefetch Flush (PF) | 0b000 | 0b0101 | 0b100 | Ignored | MCR p15, 0, Rd, c7, c5, 4 |
| Data Write Barrier (DWB) | 0b000 | 0b1010 | 0b100 | Ignored | MCR p15, 0, Rd, c7, c10, 4 |
| Data Memory Barrier (DMB) | 0b000 | 0b1010 | 0b101 | Ignored | MCR p15, 0, Rd, c7, c10, 5 |

### 7.2.8.4 Data Cache Line Allocate Function

3rd generation microarchitecture provides a "Data Cache Line Allocate" function as a performance hint. This function allocates the line (in other words, write a tag) into the L1 data cache and causes a cache line eviction as a result of the allocation. The 32 bytes of data associated with the newly allocated line are not initialized so subsequent reads return unpredictable data until software writes to the line.

*Note:* The Data Cache Line Allocation Function is deprecated on 3rd generation microarchitecture.

The line allocate function does not affect the L2 cache, in other words, the function never allocates a line in the L2 cache. Thus, a line allocated in the L1 data cache does not allocate the line in the L2 cache, even when the target memory is L2 cacheable. However, when the allocated line is in a write-through region of memory (this includes a shared memory region which is L1 and L2 cacheable), stores to initialize the allocated line write through to the L2 cache. The line is then allocated in the L2 cache with a line fill from external memory. In the L2 cache, only the stored data is relied upon, the data in the rest of the L2 cache line is unpredictable until written to by software. When the allocated line is in shared memory, the value of uninitialized words in the cache line are also unpredictable to all other agents in the system.

*Note:* The unpredictability of the uninitialized data also includes the possibility that subsequent reads from the same address (from 3rd generation microarchitecture or when in a shared system, from other agents) returns different results.

Even though the data is unpredictable, hardware guarantees that the data in the newly allocated line won't be from another context that the current context doesn't have access rights to.

**Table 46. Line Allocate Function**

| Function | User Mode | Opc_1 | CRm | Opc_2 | Data | Instruction |
|---|---|---|---|---|---|---|
| Data Cache Line Allocate | Y | 0b000 | 0b0010 | 0b101 | VA | MCR p15, 0, Rd, c7, c2, 5 |

This function is available in user and privileged modes.

The Line Allocate Function takes a VA as the data (unlike other cache functions which take an MVA). As a result, the specified address is remapped by the Process ID (see Section 7.2.13, "Register 13: Process ID" on page 105). The final MVA value then goes through the normal MMU address translation mechanism, generating a table walk on a Data TLB miss. The line allocate function is interpreted as a write operation by the MMU for permission checking purposes.

Performing a line allocate function while the data cache is disabled or to a non-cacheable region of memory, has no effect on the cache. A line allocate that hits the cache has no effect. However, aborts are still reported.

On 3rd generation microarchitecture the DC Line Allocate function is treated as a store for data breakpoint purposes. When breakpoints are enabled, the function triggers a data breakpoint when an address match and access type match occurs. An address match occurs when the breakpoint address matches any byte within the cache line being allocated.

### 7.2.8.5 Precise Data Aborts

None of the L1 cache functions, listed in Table 43, generate precise data aborts. (See Section 2.3.6.4, "Data Aborts" on page 39 for more information on precise data aborts.) The MMU is not accessed with these commands.

The L2 cache functions, listed in Table 44, that use an MVA, cause a virtual to physical address page translation and generate precise data aborts. This includes: translation aborts or external abort on translation. However, the functions do not generate domain, permission, alignment or lock aborts. L2 cache functions that generate an abort do not affect the L2 cache.

The explicit memory barrier functions, listed in Table 45, does not generate precise data aborts.

The Data Cache Line Allocate function requires a VA and performs a virtual to physical address translation, which means precise data aborts is generated. This includes: translation aborts, external abort on translation, domain aborts and permission aborts, but does not include alignment or lock aborts. A Data Cache Line Allocate function that generates an abort does not affect the data cache.

### 7.2.8.6 Interaction of Cache Functions on Locked Entries

Table 47 and Table 48 list the affect the L1 and L2 cache functions have on locked entries. In summary, functions that operate on a line by set/way have no effect when the line is locked. Functions that invalidate a line by MVA unlocks the line and perform the function.

**Table 47. L1 Cache Functions Affect on Locked Entries**

| Function | Affect on Locked Entries |
|---|---|
| Invalidate I cache & BTB | Entries remain locked and valid |
| Invalidate I cache line (MVA) | Entry is unlocked and invalidated |
| Invalidate Branch Target Buffer | n/a |
| Invalidate D cache | Entries remain locked and valid |
| Invalidate D cache line (MVA) | Entry is unlocked and invalidated |
| Invalidate I&D cache & BTB | Entries remain locked and valid |
| Clean D cache line (MVA) | Entry is cleaned and remain locked |
| Clean D cache line (set/way) | Entry is not cleaned and remain locked |
| Clean & Invalidate Dcache Line (MVA) | Entry is cleaned and invalidated and unlocked |
| Clean & Invalidate Dcache Line (set/way) | Entry is not cleaned or invalidated and remain locked |
| DC Line Allocate (VA) | No effect on target line |

**Table 48. L2 Cache Functions Affect on Locked Entries**

| Function | Affect on Locked Entries |
|---|---|
| Invalidate L2 Cache Line (MVA) | Entry is unlocked and invalidated |
| Clean L2 Cache Line (MVA) | Entry is cleaned and remain locked |
| Clean L2 Cache Line (set/way) | Entry is not cleaned and remain locked |
| Clean & Invalidate L2 Cache Line (set/way) | Entry is not cleaned or invalidated and remain locked |

### 7.2.8.7 Set/Way Format

The format of the set/way register, which is used by several cache functions, is dependent on the organization and size of the target cache.

Table 49 shows the set/way format for the L1 D-cache set/way operations. The **way** field selects one of 4 ways (0-3) and the **set** field selects of 256 sets (0-255)

**Table 49. L1 DC Set/Way Format**

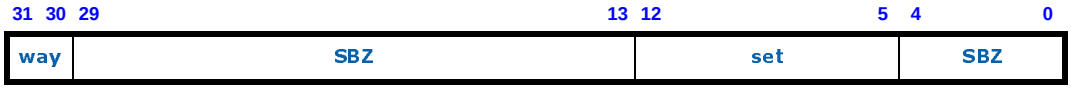



Table 50 and Table 51 show the set/way format for the L2 Unified cache set/way operations The **way** field selects one of 8 ways (0-7), regardless of the L2 cache size. The number of sets is dependant on the target cache size. For a 256KB L2 cache, the **set** field selects one of 1024 sets (0-1023). For a 512KB L2 cache, the **set** field selects on of 2048 sets (0-2047).

**Table 50. 256KB L2 Set/Way Format**

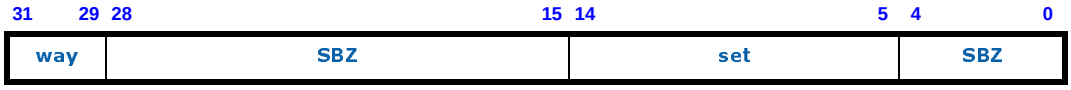
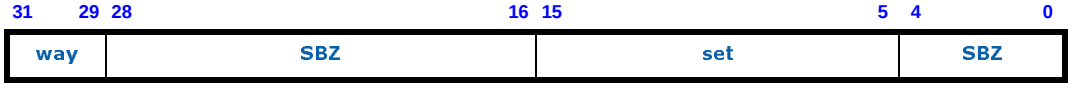



**Table 51. 512KB L2 Set/Way Format**



### 7.2.9 Register 8: TLB Operations

Register 8 contains functions for managing the 3rd generation microarchitecture TLBs. These allow the TLBs to be globally invalidated or invalidated by entry based on a specified modified virtual address.

These functions are only allowed in privileged mode; accessing these functions in user mode generates an undefined instruction exception. Also, these functions are accessed as write-only; accessing these functions with an **MRC** has unpredictable results.

All operations defined in Table 52 work regardless of whether the MMU is enabled or disabled. These operations do not generate precise data aborts.

**Table 52. TLB Functions**

| Function | Opc_1 | CRm | Opc_2 | Data | Instruction |
|---|---|---|---|---|---|
| Invalidate I&D TLB | 0b000 | 0b0111 | 0b000 | Ignored | MCR p15, 0, Rd, c8, c7, 0 |
| Invalidate I TLB | 0b000 | 0b0101 | 0b000 | Ignored | MCR p15, 0, Rd, c8, c5, 0 |
| Invalidate I TLB entry | 0b000 | 0b0101 | 0b001 | MVA | MCR p15, 0, Rd, c8, c5, 1 |
| Invalidate D TLB | 0b000 | 0b0110 | 0b000 | Ignored | MCR p15, 0, Rd, c8, c6, 0 |
| Invalidate D TLB entry | 0b000 | 0b0110 | 0b001 | MVA | MCR p15, 0, Rd, c8, c6, 1 |

Table 53 shows how these commands affect locked entries.

**Table 53. Interaction of TLB Functions with Locked Entries**

| Function | Affect on Locked Entries |
|---|---|
| Invalidate I&D TLB | Locked entry is not invalidated and not unlocked |
| Invalidate I TLB | Locked entry is not invalidated and not unlocked |
| Invalidate I TLB entry | Result is unpredictable when entry was locked |
| Invalidate D TLB | Locked entry is not invalidated and not unlocked |
| Invalidate D TLB entry | Result is unpredictable when entry was locked |

### 7.2.10 Register 9: Cache Lock Down

Register 9 is used for locking down entries into the instruction cache, data cache and L2 cache as shown in Table 54. The protocol for locking down entries is found in Chapter 4.0, "Instruction Cache", Chapter 6.0, "Data Cache" and Chapter 8.0, "Level 2 Unified Cache (L2)" respectively.

These functions are only accessible in privileged mode; accessing these functions in user mode generates an undefined instruction exception. Also, all functions, except the DC Lock Register, are accessed as write-only; accessing these functions with an **MRC** has unpredictable results. For the DC Lock Register, the Data Cache Lock Mode bit is readable and writable by privileged software.

Fetch and lock commands for the instruction cache and L2 cache explicitly specify a modified virtual address in Rd as the line to lock. The data cache locking mechanism follows a different procedure than the instruction cache and L2 cache. The data cache is placed in lock down mode such that all subsequent line fills to the data cache result in that line being locked in, as controlled by Table 55.

The "Allocate and Lock L2 Cache Line" command does not perform a fill operation. Instead the tag is written into the L2 cache and then locked. The data associated with the line has an unpredictable value, meaning subsequent reads returns unpredictable values.

Unlock Instruction Cache, Unlock Data Cache and Unlock L2 Cache are global operations; these unlock the entire target cache.

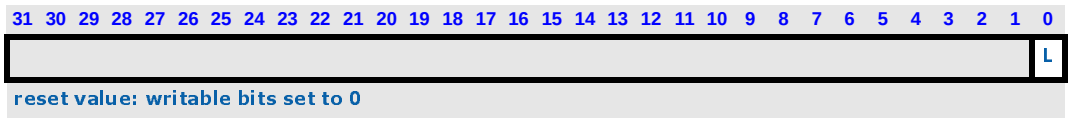
Lock/unlock operations on a disabled cache have an unpredictable effect. Lock operations by MVA to a non L2 cacheable memory location have unpredictable effect on the L2 cache.

Cache lockdown functions which operate on an MVA require an address translation when the MMU is enabled, and generates precise data aborts (see Section 7.2.10.1).

**Table 54. Cache Lockdown Functions**

| Function | Opc_1 | CRm | Opc_2 | Data | Instruction |
|---|---|---|---|---|---|
| Fetch and Lock I Cache Line | 0b000 | 0b0101 | 0b000 | MVA | MCR p15, 0, Rd, c9, c5, 0 |
| Fetch and Lock L2 Cache Line | 0b001 | 0b0101 | 0b000 | MVA | MCR p15, 1, Rd, c9, c5, 0 |
| Unlock Instruction Cache | 0b000 | 0b0101 | 0b001 | Ignored | MCR p15, 0, Rd, c9, c5, 1 |
| Unlock L2 Cache | 0b001 | 0b0101 | 0b001 | Ignored | MCR p15, 1, Rd, c9, c5, 1 |
| Allocate and Lock L2 Cache Line | 0b001 | 0b0101 | 0b010 | MVA | MCR p15, 1, Rd, c9, c5, 2 |
| Read Data Cache Lock Register | 0b000 | 0b0110 | 0b000 | lock mode value | MRC p15, 0, Rd, c9, c6, 0 |
| Write Data Cache Lock Register | 0b000 | 0b0110 | 0b000 | lock mode value | MCR p15, 0, Rd, c9, c6, 0 |
| Unlock Data Cache | 0b000 | 0b0110 | 0b001 | Ignored | MCR p15, 0, Rd, c9, c6, 1 |

**Table 55. Data Cache Lock Register**




| Bits | Access | Description |
|---|---|---|
| 31:1 | Read-unpredictable / Write-as-Zero | Reserved |
| 0 | Read / Write | **Data Cache Lock Mode (L)**<br>0 = fills to the data cache are not locked<br>1 = fills into the data cache get locked in |

### 7.2.10.1 Precise Data Aborts

The L2 cache lock functions that operate by MVA cause a virtual to physical address translation and generates precise data aborts. This includes: translation aborts and external abort on translation. These functions do not generate domain, permission, alignment or lock aborts. L2 cache lock functions that generate an abort do not affect the L2 cache.

The Fetch and Lock I Cache Line function also operate by MVA and cause a virtual to physical address translation. Data aborts detected during the address translation or fetch of the target line are reported as lock aborts. Only translation aborts, external abort on translation, or external bus errors are detected. The MMU does not do any access permission, domain or address alignment checking on a Fetch and Lock IC Line function. A Fetch and Lock IC Line function that generates an abort does not affect the instruction cache.

### 7.2.10.2 Legacy Support

The L1 cache lock/unlock functions have been moved for 3rd generation microarchitecture, however the previous encoding is also supported for legacy reasons. The legacy encoding is deprecated on 3rd generation microarchitecture; new software uses the encoding specified in Table 54.

**Table 56. Legacy Encoding for L1 Cache Lockdown Functions**

| Function | Opc_1 | CRm | Opc_2 | Data | Instruction |
|---|---|---|---|---|---|
| Fetch and Lock I Cache Line | 0b000 | 0b0001 | 0b000 | MVA | MCR p15, 0, Rd, c9, c1, 0 |
| Unlock Instruction Cache | 0b000 | 0b0001 | 0b001 | Ignored | MCR p15, 0, Rd, c9, c1, 1 |
| Read Data Cache Lock Register | 0b000 | 0b0010 | 0b000 | lock mode value | MRC p15, 0, Rd, c9, c2, 0 |
| Write Data Cache Lock Register | 0b000 | 0b0010 | 0b000 | lock mode value | MCR p15, 0, Rd, c9, c2, 0 |
| Unlock Data Cache | 0b000 | 0b0010 | 0b001 | Ignored | MCR p15, 0, Rd, c9, c2, 1 |



### 7.2.11 Register 10: TLB Lock Down

Register 10 is used for locking down entries into the instruction TLB and data TLB

These functions are only accessible in privileged mode; accessing these in user mode generates an undefined instruction exception. All TLB lock down functions are accessed as write-only. Access with an **MRC** produces unpredictable results.

Table 57 shows the command for locking down entries in the instruction TLB, and data TLB. The entry to lock is specified by the modified virtual address in Rd.

The "Translate and Lock" commands produces unpredictable results when the virtual address translation already exists in the TLB.

The TLB Lock and Unlock commands have an unpredictable effect when the MMU is disabled.

**Table 57. TLB Lockdown Functions**

| Function | Opc_1 | CRm | Opc_2 | Data | Instruction |
|---|---|---|---|---|---|
| Translate and Lock I TLB entry | 0b000 | 0b0100 | 0b000 | MVA | MCR p15, 0, Rd, c10, c4, 0 |
| Unlock I TLB | 0b000 | 0b0100 | 0b001 | Ignored | MCR p15, 0, Rd, c10, c4, 1 |
| Translate and Lock D TLB entry | 0b000 | 0b1000 | 0b000 | MVA | MCR p15, 0, Rd, c10, c8, 0 |
| Unlock D TLB | 0b000 | 0b1000 | 0b001 | Ignored | MCR p15, 0, Rd, c10, c8, 1 |

The Translation and Lock Functions operate by MVA and cause a virtual to physical address translation. Any data abort detected during the translation is reported as lock aborts. Only external abort on translation or translation abort is detected. The MMU does not do any access permission, domain or address alignment checking on these functions. Operations that generate an abort do not affect the target TLB.

### 7.2.12 Register 11-12: Reserved

These registers are reserved. Reading and writing these yields unpredictable results.

## 7.2.13 Register 13: Process ID

**Table 58. Register 13 Functions (CRn=13)**

| Function | Opc_1 | CRm | Opc_2 | Instruction |
|---|---|---|---|---|
| Process ID Register (PID) | 0b000 | 0b0000 | 0b000 | MRC p15, 0, Rd, c13, c0, 0<br>MCR p15, 0, Rd, c13, c0, 0 |

3rd generation microarchitecture supports the remapping of virtual addresses through a Process ID (PID) register. This remapping occurs before the instruction cache, instruction TLB, data cache and data TLB are accessed. The address resulting from the remapping of the PID with the VA is referred to as the modified virtual address (MVA).

The PID Register is only accessible in privileged mode; accessing it in user mode generates an undefined instruction exception.

The PID register is a 7-bit value that replaces bits 31:25 of the virtual address when these are zero. This effectively remaps the address to one of 128 "slots" in the 4 Gbytes of virtual address space. When bits 31:25 of the virtual address are not zero or the PID value is 0, no remapping occurs. This feature is useful for operating system management of processes that maps to the same virtual address space. In those cases, the virtually mapped caches on 3rd generation microarchitecture does not require invalidating on a process switch since the MVA in the cache tag contains the PID.

Any write to the PID register automatically invalidates the BTB.

**Table 59. Process ID Register**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Process ID | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

reset value: 0x0000,0000

| Bits | Access | Description |
|---|---|---|
| 31:25 | Read / Write | Process ID<br>This field is used for remapping the virtual address when bits 31-25 of the virtual address are zero. |
| 24:0 | Read-as-Zero / Write-as-Zero | Reserved |

### 7.2.13.1 The PID Register Effect On Addresses

Any address on a data access or instruction fetch is modified by the PID when the conditions described in the previous section are met. The only CP15 function to be remapped by the PID is the DC Line Allocate function. All other CP15 functions that require an address as data, require an MVA. The address provided must have the PID appropriately combined with the target VA by software

In general, addresses generated and used by User Mode code are eligible for being remapped by the PID. Privileged code, however, must be aware of certain special cases in which address generation does not follow the usual flow. Cache and TLB operations which require an MVA are not remapped by the PID. In addition CP15 registers such as the instruction and data breakpoint registers require an MVA and are not remapped by the PID.

### 7.2.14 Register 14: Breakpoint Registers

**Table 60. Register 14 Functions (CRn=14)**

| Function | Opc_1 | CRm | Opc_2 | Instruction |
|---|---|---|---|---|
| Instruction Breakpoint Register 0 (IBR0) | 0b000 | 0b1000 | 0b000 | MRC p15, 0, Rd, c14, c8, 0<br>MCR p15, 0, Rd, c14, c8, 0 |
| Instruction Breakpoint Register 1 (IBR1) | 0b000 | 0b1001 | 0b000 | MRC p15, 0, Rd, c14, c9, 0<br>MCR p15, 0, Rd, c14, c9, 0 |
| Data Breakpoint Register 0 (DBR0) | 0b000 | 0b0000 | 0b000 | MRC p15, 0, Rd, c14, c0, 0<br>MCR p15, 0, Rd, c14, c0, 0 |
| Data Breakpoint Register 1 (DBR1) | 0b000 | 0b0011 | 0b000 | MRC p15, 0, Rd, c14, c3, 0<br>MCR p15, 0, Rd, c14, c3, 0 |
| Data Breakpoint Control Register (DBCON) | 0b000 | 0b0100 | 0b000 | MRC p15, 0, Rd, c14, c4, 0<br>MCR p15, 0, Rd, c14, c4, 0 |

3rd generation microarchitecture contains two instruction breakpoint address registers (IBR0 and IBR1), one data breakpoint address register (DBR0), one configurable data breakpoint mask/address register (DBR1), and one data breakpoint control register (DBCON).

These breakpoint resources are only accessible in privileged mode; accessing these in user mode generates an undefined instruction exception.

Refer to Chapter 12.0, "Software Debug" for more information on using the 3rd generation microarchitecture breakpoint resources.

## 7.2.15 Register 15: Co-processor Access Register

**Table 61. Register 15 Functions (CRn=15)**

| Function | Opc_1 | CRm | Opc_2 | Instruction |
|---|---|---|---|---|
| Co-processor Access Register (CPAR) | 0b000 | 0b0001 | 0b000 | MRC p15, 0, Rd, c15, c1, 0<br>MCR p15, 0, Rd, c15, c1, 0 |

The Co-processor Access Register (CPAR) controls access rights to all the co-processors in the system except for CP15, CP14 and part of CP7. In CP7, the register only controls the rights for ASSP co-processor register, and not the 3rd generation microarchitecture defined co-processor registers. For more information on which co-processors are implemented in an ASSP see the 3rd generation microarchitecture implementation options section of the relevant product documentation.

CPAR also controls access to the 40-bit internal accumulator located in CP0 (see Section 2.3.1, "Media Processing Co-processor (CP0)" on page 28 for more information about the internal accumulator).

Table 62 shows the register format. The CPAR is only accessible in privileged mode; accessing it in user mode generates an undefined instruction exception.

**Table 62. Co-processor Access Register**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | | | CP13 | CP12 | CP11 | CP10 | CP9 | CP8 | CP7 | CP6 | CP5 | CP4 | CP3 | CP2 | CP1 | CP0 |

**reset value: 0x0000,0000**

| Bits | Access | Description |
|---|---|---|
| 31:14 | Read-unpredictable / Write-as-Zero | Reserved |
| 13:1 | Read / Write | Co-processor Access Rights<br>Each bit in this field corresponds to the access rights for each co-processor.[a] Refer to the 3rd generation microarchitecture implementation options section of the relevant product documentation to find out which, when any, co-processors exist and for the definition of these bits. |
| 0 | Read / Write | Co-processor 0 Access Rights<br>This bit corresponds to the access rights for CP0.<br>0 = Access denied. Any attempt to access the corresponding co-processor generates an undefined instruction exception.<br>1 = Access allowed. Includes read and write accesses. |

a. For CP7, this bit only controls access to ASSP defined registers, not the 3rd generation microarchitecture CP7 registers defined in Section 7.4

## 7.3 CP14 Registers

Table 63 lists the CP14 registers implemented in 3rd generation microarchitecture.

**Table 63. CP14 Registers**

| Register (CRn) | Opc_1 | CRm | Opc_2 | Access | Description |
|---|---|---|---|---|---|
| 0,1,4,5,8 | 0 | 1 | 0 | Read / Write | Performance Monitoring |
| 0-3 | 0 | 2 | 0 | | |
| 6,7 | 0 | 0 | 0 | Read / Write | Clock and Power Management |
| 8-14 | 0 | 0 | 0 | Varies[a] | Software Debug |

a. Access varies depending on the specified register.

All other registers are reserved in CP14. Reading and writing these yields unpredictable results.

All CP14 registers are only accessible in privileged mode; accessing these in user mode generates an undefined instruction exception.

### 7.3.1 Performance Monitoring Registers

The performance monitoring unit contains a control register (PMNC), a clock counter (CCNT), interrupt enable register (INTEN), overflow flag register (FLAG), event selection register (EVTSEL) and four event counters (PMN0 through PMN3). The format of these registers is found in Chapter 11.0, "Performance Monitoring", along with a description on how to use the performance monitoring facility.

These registers are not accessed by **LDC** and **STC** co-processor instructions.

**Table 64. Performance Monitoring Registers**

| Description | Opc_1 | CRn | CRm | Opc_2 | Instruction |
|---|---|---|---|---|---|
| Performance Monitor Control Register (PMNC) | 0b000 | 0b0000 | 0b0001 | 0b000 | MRC p14, 0, Rd, c0, c1, 0<br>MCR p14, 0, Rd, c0, c1, 0 |
| Clock Counter Register (CCNT) | 0b000 | 0b0001 | 0b0001 | 0b000 | MRC p14, 0, Rd, c1, c1, 0<br>MCR p14, 0, Rd, c1, c1, 0 |
| Interrupt Enable Register (INTEN) | 0b000 | 0b0100 | 0b0001 | 0b000 | MRC p14, 0, Rd, c4, c1, 0<br>MCR p14, 0, Rd, c4, c1, 0 |
| Overflow Flag Register (FLAG) | 0b000 | 0b0101 | 0b0001 | 0b000 | MRC p14, 0, Rd, c5, c1, 0<br>MCR p14, 0, Rd, c5, c1, 0 |
| Event Selection Register (EVTSEL) | 0b000 | 0b1000 | 0b0001 | 0b000 | MRC p14, 0, Rd, c8, c1, 0<br>MCR p14, 0, Rd, c8, c1, 0 |
| Performance Count Register 0 (PMN0) | 0b000 | 0b0000 | 0b0010 | 0b000 | MRC p14, 0, Rd, c0, c2, 0<br>MCR p14, 0, Rd, c0, c2, 0 |
| Performance Count Register 1 (PMN1) | 0b000 | 0b0001 | 0b0010 | 0b000 | MRC p14, 0, Rd, c1, c2, 0<br>MCR p14, 0, Rd, c1, c2, 0 |
| Performance Count Register 2 (PMN2) | 0b000 | 0b0010 | 0b0010 | 0b000 | MRC p14, 0, Rd, c2, c2, 0<br>MCR p14, 0, Rd, c2, c2, 0 |
| Performance Count Register 3 (PMN3) | 0b000 | 0b0011 | 0b0010 | 0b000 | MRC p14, 0, Rd, c3, c2, 0<br>MCR p14, 0, Rd, c3, c2, 0 |

## 7.3.2 Clock and Power Management Registers

### Table 65. Clock and Power Management Functions

| Function | Opc_1 | CRn | CRm | Opc_2 | Instruction |
|---|---|---|---|---|---|
| Power Mode Register (PWRMODE) | 0b000 | 0b0111 | 0b0000 | 0b000 | MRC p14, 0, Rd, c7, c0, 0<br>MCR p14, 0, Rd, c7, c0, 0 |
| Microarchitecture Clock Configuration Register (CCLKCFG) | 0b000 | 0b0110 | 0b0000 | 0b000 | MRC p14, 0, Rd, c6, c0, 0<br>MCR p14, 0, Rd, c6, c0, 0 |

These registers allow software to manage the microarchitecture clock and power management modes.

Power management modes are supported through the PWRMODE Register. The function and definition of these modes is defined by the ASSP. The user refers to the 3rd generation microarchitecture implementation options section of the relevant product documentation for specifics on the use of these registers.

Software enters a specific low power mode by writing the appropriate value to the register.

Software reads this register, but since software only runs during ACTIVE mode, it always reads zeros from the **M** field.

### Table 66. PWRMODE Register

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | M | | | |

reset value: writable bits set to 0

| Bits | Access | Description |
|---|---|---|
| 31:4 | Read-unpredictable / Write-as-Zero | Reserved |
| 3:0 | Read / Write | Mode (M)<br>0 = ACTIVE mode<br>All other values are defined by the ASSP |

Software changes the microarchitecture clock frequency by writing to the CCLKCFG register. This function informs the clocking unit (located external to 3rd generation microarchitecture) to change the microarchitecture clock frequency. Software reads CCLKCFG to determine current operating frequency. Exact definition of this register is determined by the ASSP and is found in the 3rd generation microarchitecture implementation option sections of the relevant product documentation.

### Table 67. CCLKCFG Register

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | CCLKCFG | | | |

reset value: unpredictable

| Bits | Access | Description |
|---|---|---|
| 31:4 | Read-unpredictable / Write-as-Zero | Reserved |
| 3:0 | Read / Write | Microarchitecture **Clock Configuration (CCLKCFG)**<br>This field is used to configure the microarchitecture clock frequency and is defined by the ASSP. |

### 7.3.3 Software Debug Registers

Software debug is supported by address breakpoint registers (Co-processor 15, register 14), serial communication over the JTAG interface and a trace buffer. Registers 8, 9 and 14 are used for the serial interface, register 10 is for general control and registers 11 through 13 support a 256 entry trace buffer. These registers are explained in more detail in Chapter 12.0, "Software Debug".

**Table 68. SW Debug Functions**

| Function | opc_1 | CRn | CRm | opc_2 | Instruction |
|---|---|---|---|---|---|
| Transmit Register (TX) | 0b000 | 0b1000 | 0b0000 | 0b000 | MCR p14, 0, Rd, c8, c0, 0 |
| Receive Register (RX) | 0b000 | 0b1001 | 0b0000 | 0b000 | MRC p14, 0, Rd, c9, c0, 0 |
| Debug Control and Status Register (DCSR) | 0b000 | 0b1010 | 0b0000 | 0b000 | MRC p14, 0, Rd, c10, c0, 0<br>MCR p14, 0, Rd, c10, c0, 0 |
| Trace Buffer Register (TBREG) | 0b000 | 0b1011 | 0b0000 | 0b000 | MRC p14, 0, Rd, c11, c0, 0 |
| Checkpoint 0 Register (CHKPT0) | 0b000 | 0b1100 | 0b0000 | 0b000 | MRC p14, 0, Rd, c12, c0, 0<br>MCR p14, 0, Rd, c12, c0, 0 |
| Checkpoint 1 Register (CHKPT1) | 0b000 | 0b1101 | 0b0000 | 0b000 | MRC p14, 0, Rd, c13, c0, 0<br>MCR p14, 0, Rd, c13, c0, 0 |
| Transmit/Receive Control Register (TXRXCTRL) | 0b000 | 0b1110 | 0b0000 | 0b000 | MRC p14, 0, Rd, c14, c0, 0<br>MCR p14, 0, Rd, c14, c0, 0 |

## 7.4 CP7 Registers

### Table 69. CP7 Registers

| Description | Opc_1 | CRn | CRm | Opc_2 | Instruction |
|---|---|---|---|---|---|
| L2 Cache and BIU Error Logging Register (ERRLOG) | 0b000 | 0b0000 | 0b0010 | 0b000 | MRC p7, 0, Rd, c0, c2, 0<br>MCR p7, 0, Rd, c0, c2, 0 |
| Error Lower Address Register (ERRADRL) | 0b000 | 0b0001 | 0b0010 | 0b000 | MRC p7, 0, Rd, c1, c2, 0<br>MCR p7, 0, Rd, c1, c2, 0 |
| Error Upper Address Register (ERRADRU) | 0b000 | 0b0010 | 0b0010 | 0b000 | MRC p7, 0, Rd, c2, c2, 0<br>MCR p7, 0, Rd, c2, c2, 0 |

The CP7 registers defined by 3rd generation microarchitecture use CRm=2. Registers with CRm!=2 are reserved to ASSP usage. Refer to the implementation options section of the relevant product documentation for details on ASSP specific co-processor registers. The details in this section only apply to the 3rd generation microarchitecture defined CP7 registers.

All registers with CRm=2 which are not defined in Table 69 are reserved. Reading and writing these yields unpredictable results.

The 3rd generation microarchitecture CP7 registers are accessible only in privileged mode, with **MRC** and **MCR** co-processor instructions. Accessing these in user mode generates an undefined instruction exception.

These registers listed in Table 69 are explained in more detail in Section 8.5, "Level 2 Cache and Bus Interface Unit Register Definitions" on page 132.

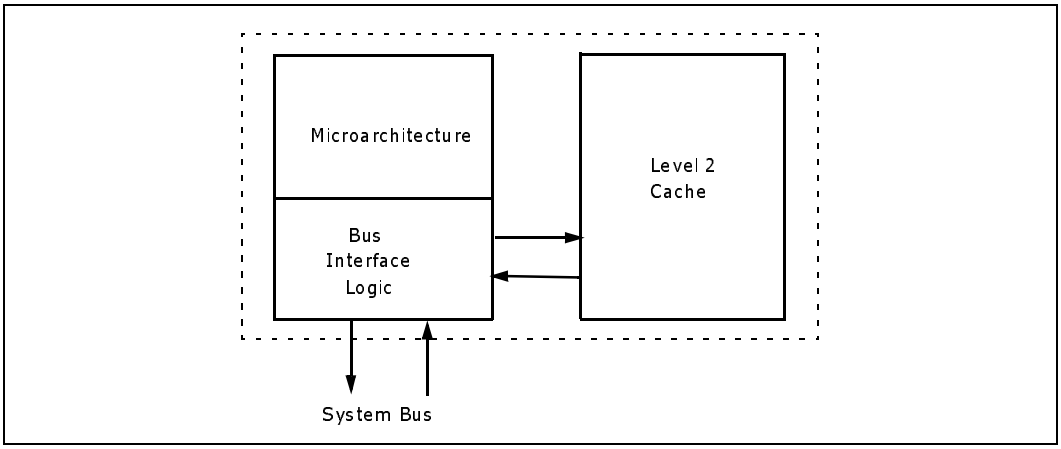
# 8.0 Level 2 Unified Cache (L2)

This chapter describes the behavior of the on-chip Level 2 Unified Cache (L2) and the Bus Interface Unit (BIU) of the 3rd generation Intel XScale® microarchitecture (3rd generation microarchitecture or 3rd generation).

## 8.1 Overviews

The L2 Unified Cache and Bus Interface Unit (BIU) work together to provide a high-performance memory subsystem for 3rd generation microarchitecture. The L2 and BIU are tightly coupled to the microarchitecture. Furthermore, the BIU interfaces to a high-performance on-chip system bus. Figure 8 shows the L2 cache and BIU in 3rd generation microarchitecture from a high-level perspective.

**Figure 8. 3rd Generation Microarchitecture High-Level Block Diagram**



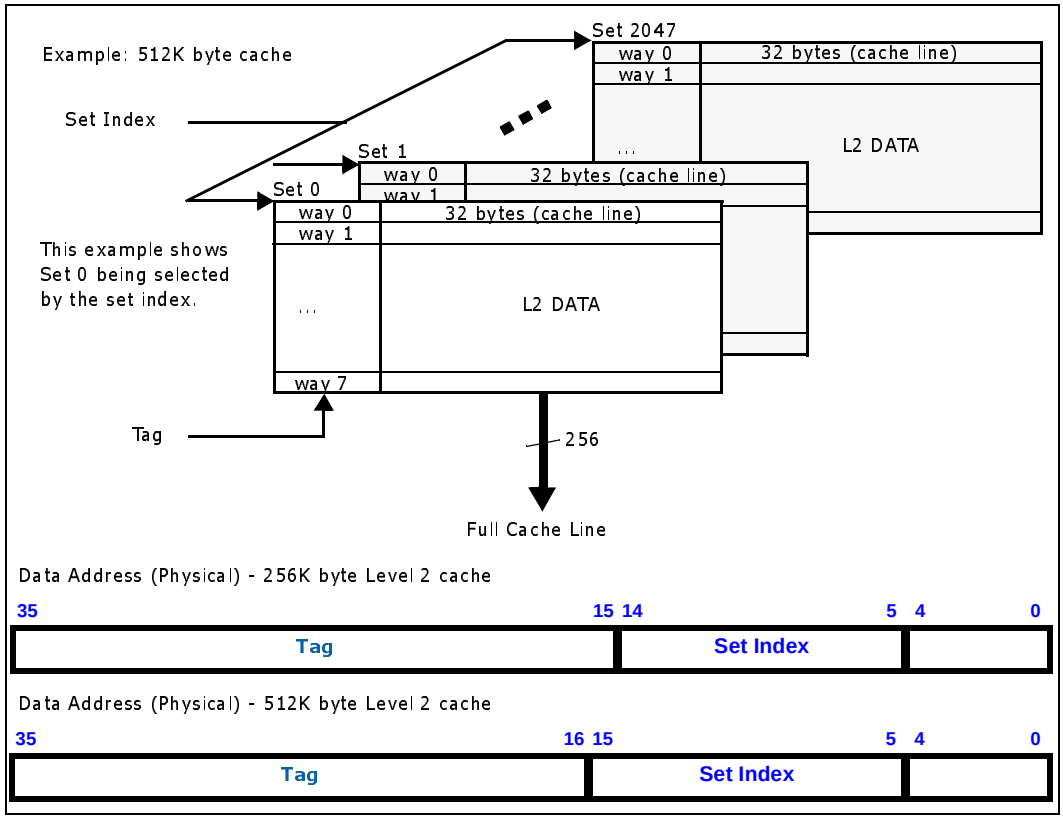
## 8.1.1 Level 2 Cache Overview

3rd generation microarchitecture implements an *8-way set associative* L2 cache. The cache line size is 32 bytes. The L2 cache controller supports two cache sizes: *256 KB or 512 KB*. The size determines the number of sets; a 256 KB cache has 1024 sets, while the 512 KB cache has 2048 sets. Each way of a set contains a cache line and three corresponding state bits indicating the state of the cache line (valid, modified, invalid, etc.). The replacement policy uses a NRU (not recently used) algorithm. The L2 cache is a fully pipelined, non-blocking cache, and operates at half the 3rd generation microarchitecture frequency. The L2 cache is unified in that it provides the ability to cache both instructions and data. The L2 cache is physically addressed using a 36-bit address, providing up to 64 GB of addressable memory.

*Note:* The 3rd generation microarchitecture is also available as an option without the L2 cache. Refer to the relevant product documentation to determine whether an L2 cache is supported or not.

Figure 9, "Level 2 Cache Organization" on page 113 shows the cache organization for a 512 KB cache and how the data or instruction address is used to access the cache. The 256 KB cache has the same organization, but there are half as many sets as in the 512 KB cache. Note that all accesses to the L2 array occur at a full cache line width.

**Figure 9. Level 2 Cache Organization**





L2 Cache policies such as cacheability and coherence, are adjusted for particular regions of memory by programming page attribute bits in the MMU descriptor that controls that memory. See Section 8.2.3, "Memory Attributes" on page 119 for a description of these bits.

The L2 cache supports write-back only caching. The L2 cache does not support write-through caching. Accesses to L2 cacheable memory marked as write-through are treated as L2 un-cacheable (see Section 8.2.3.2, "L2 Write Policy" on page 119). Data written to the L2 cache is only written to system memory when a line victimization due to replacement occurs, when a clean operation occurs on a modified line, or when a snoop probe forces a modified line to be written back to memory. The L2 cache always allocates a line into the cache in the event of a cacheable read-miss, or a cacheable write-miss.

The L2 cache supports hardware cache coherence, using the MOESI cache coherence protocol (Modified, Owned, Exclusive, Shared, Invalid). Hardware cache coherence allows multiple 3rd generation microarchitecture processors and I/O devices to share data without software intervention. In a coherent system, the L2 also supports a push-cache capability. This allows specially tagged write transactions on the system bus to push data directly into shared memory in the L2.

3rd generation microarchitecture provides several L2 cache maintenance operations to help manage the cache, including invalidate, clean, and clean & invalidate. In addition, there are special operations that allow the L2 cache to be locked on a per-line basis. These operations are describe more in Section 8.3, "Level 2 Cache Control" on page 126.

The L2 data array is ECC protected. Single bit errors are detected and corrected when encountered, while double bit errors are detected only. The L2 tag and state arrays are parity protected. This is described more in Section 8.2.6, "ECC and Parity Protection" on page 125.

## 8.1.2 Bus Interface Unit Overview

The Bus Interface Unit (BIU) accepts 3rd generation microarchitecture requests and schedules these to the L2 cache and/or the system bus. The BIU directs return data to 3rd generation microarchitecture to fulfill microarchitecture requested loads, instruction fetches or TLB table walks, and also to the L2 cache in the case of L2 cacheable memory accesses.

All microarchitecture requests to the BIU are physically addressed. During coherent memory operations, any snooping in the BIU and L2 cache is performed with physical addresses and a physical address is passed back to the microarchitecture for L1 cache invalidations as appropriate.

3rd generation microarchitecture supports a weakly ordered memory consistency model. As a result, the BIU reorders any cacheable requests from the microarchitecture. However, uncacheable requests are issued to the system bus in strict program order. There are no ordering dependencies between cacheable and uncacheable requests (in other words, cacheable requests in between uncacheable requests are reordered and/or issued prior to pending uncacheable requests). When ordering is desired, fence operations are used. For a full description of the 3rd generation microarchitecture memory ordering model and the available fencing operations, please see Chapter 10.0, "Memory Ordering"

## 8.2 Level 2 Unified Cache Operation

The L2 cache and BIU receive access requests from two places, either the microarchitecture, or from the system bus. Each of these types of requests is discussed in the subsequent sections.

### 8.2.1 L2 Cache / BIU Operations due to Microarchitecture Requests

The L2 cache and BIU receive many different types of requests from the microarchitecture. These requests include instruction fetches, data read and data write operations. These microarchitecture request types are briefly outlined in Table 70. These requests are either L2 cacheable or L2 uncacheable depending upon the memory attributes of the request. When L2 cacheable, these are presented to the L2; otherwise, these are forwarded to the system bus. The L2 cache and BIU also support many L2 cache maintenance operations. These are not listed in Table 70, but are described in more detail in Section 8.3.3, "Invalidate and Clean Operations" on page 127

**Table 70. Microarchitecture Request Types**

| Read Requests | Description |
|---|---|
| Instruction fetches | Request to load an instruction from a particular address. When L2 cacheable, this request is presented to the L2 cache as a read request. |
| Data loads | Request to load data from a particular address. When L2 cacheable, this request is presented to the L2 cache as a read request. |
| Instruction/Data TLB fetches | Request to load TLB data needed for a virtual to physical address translation. When the L2 is enabled, TLB information is cached in the L2 for higher performance. These types of requests are presented to the L2 cache as read requests. |
| SWAP | Request to perform the read portion of an atomic swap operation. When L2 cacheable, this request is presented to the L2 cache as a read request. |
| **Write Requests** | **Description** |
| L1 stores to write-through memory | Request to store data to L1 write-through memory. When L2 cacheable, this request is presented to the L2 cache as a write request. |
| L1 uncacheable stores | Request to store data when L1 is uncacheable. When L2 cacheable, this request is presented to the L2 cache as a write request. |
| L1 store miss requests to write-back memory | Request to store data to L1 write-through memory. When L1 misses, and the request is L2 cacheable, it is presented to the L2 cache as a write request. This is because the L1 cache does not support write allocate on miss. |
| L1 cache victimizations of lines in write-back memory | Request to store victim data when L1 victimizes a line from write back memory. When the request is L2 cacheable, it is presented to the L2 cache as a write request. |
| SWAP | Request to perform the write portion of an atomic swap operation. When L2 cacheable, this request is presented to the L2 cache as a write request. |

When an L2 cacheable read or write request is received from the microarchitecture, the L2 cache compares the address of the request against the addresses of the data currently in the cache. It also checks the state information in addition to the address. This information is used to determine whether or not the access results in a *cache hit* or a *cache miss.* The expected L2 cache behavior for each of these types of results is outlined in Table 71.

**Table 71. L2 Cache "Hit" Definition**

| Request Type | Cache Hit Definition | Resulting Behavior |
|---|---|---|
| Read | Request address matches the address of the data currently in the L2 cache **AND** the line is present in one of [M,O,E,S] states. Note that for a swap operation, a "hit" is only true when the line is present in one of [M,E] states. | The L2 cache returns the requested data to the 3rd generation microarchitecture. |
| Write | Request address matches the address of the data currently in the L2 cache **AND** the line is present in one of [M,E] states. | The provided data from the 3rd generation microarchitecture is written into the L2 cache. |

When a given request type does not meet the *cache hit* criteria defined Table 71, then the access results in a *cache miss*, and the L2 cache and BIU take the necessary steps to process the request. The sequence of these steps depends on the configuration of the cache and the configuration of the MMU and the page attributes. This is further described in Section 8.2.4.1, "Read Miss Policy"" and Section 8.2.4.2, "Write Miss Policy"" for a read miss and write miss respectively.

Note that some microarchitecture requests (such as stores) to shared memory finds the data in the L2 cache in the *shared* or *owned* state. These states do not give right to modify, and require a system bus transaction to maintain coherence.

Accesses to and from the actual L2 array occur only at line granularity (32 bytes). This means that any load requests for data cause the L2 cache to return the entire cache line to the microarchitecture. Any store requests that are less than the cache line width (32 bytes) cause a read-modify-write operation to occur in the L2 cache. When the line is present in the cache, it is first read out, and then it is merged with the write data provided by the microarchitecture in a merge buffer before writing it back into the L2 cache.

When the L2 cache is not present in an ASSP implementation, all microarchitecture requests are forwarded directly to the system bus.

When the L2 cache is disabled, all microarchitecture requests bypass the L2 cache and are forwarded directly to the system bus and the L2 cache are not accessed or updated in any way. The details of enabling the L2 cache are described in more detail in Chapter 7.0, "Configuration".

## 8.2.2 Level 2 Cache / BIU Operations Due to System Bus Requests

The L2 cache and BIU also receive access requests from the system bus. These request types are outlined in Table 72.

**Table 72. System Bus Requests to L2**

| Request Type | Description |
|---|---|
| Snoop Probes | A system bustransaction to coherent memory probes the L2 cache to see when it contains the desired cache line. This is the result of another system agent making a coherent memory request to the line. These types of requests only occurs to coherent (or shared) memory. |
| Push-Cache Requests | A specially tagged "push" transaction allows an agent on the system bus to *push* or write a full cache line of data directly into the L2 cache. These types of requests only occurs to coherent (or shared) memory. |

### 8.2.2.1 Snoop Probes

These types of requests only occurs to *coherent* memory (see Section 8.2.4.1, "Read Miss Policy" on page 121). When an L2 cacheable, coherent memory request is received from the system bus, the L2 cache compares the address of the request against the addresses of data currently in the cache. This is known as a *snoop probe*. When the cache line associated with the snoop is resident in the L2 cache in a valid state [M,O,E or S], then the access results in a *cache hit*. The status of the line is then reported to the bus as part of the snoop response protocol. In these cases, the data is provided from the L2 cache to the bus. When the L2 cache does not contain the requested data, then no snoop response is provided to the bus, indicating to the requesting agent that the line is not present in the L2 cache.

### 8.2.2.2 Push-Cache Requests

These types of requests allow a non-3rd generation microarchitecture agent on the system bus to push data into the L2. This allows bus agents to move critical data closer to the microarchitecture prior to use to reduce the new-data cache miss penalty. Push requests are only supported to coherent memory. As such, these types of transactions also results in a snoop probe. For non-target 3rd generation microarchitecture agents, when the cache line associated with the push request is resident in the L2 cache in a valid state [M,O,E or S], that cache line is invalidated [I]. For target 3rd generation microarchitecture agents, a cache line is allocated in a modified state, and the push data is written into the L2. More details on the push operations are also found in Chapter 9.0, "Cache Coherence."

*Note:* It is assumed that the push operation takes write precedence. When a push operation encounters any modified data in any 3rd generation microarchitecture agent L2 cache, that data is invalidated, without writing the modified data back to memory.

## 8.2.3 Memory Attributes

To support the L2 cache as well as hardware cache coherency for 3rd generation microarchitecture based products, 3rd generation microarchitecture uses a new memory attribute encoding. The new memory attributes affect aspects of L2 operation, including whether or not the access is *L2 Cacheable*, the *L2 Write Policy,* and whether or not the access is to *Shared* memory. These attributes also allow independent configuration of *outer* and *inner* caches. For 3rd generation microarchitecture, the L2 cache is considered an "outer" cache. All of these memory attributes are effectively ignored when the MMU is disabled. Full details of the outer and inner cache memory attribute encoding is found in Chapter 3.0, "Memory Management".

### 8.2.3.1 L2 Cacheability

The *outer cacheable* memory attribute encoding specifies that the associated memory is cacheable by the L2 cache.

When the MMU is disabled, the L2 unified cache is effectively disabled from caching. When the MMU is enabled, the L2 unified cache caches data for a region of memory when:

- the outer cacheable memory attribute encoding is set for the accessed address and
- the L2 unified cache is enabled.

When the outer cacheable memory attribute encoding is not set, access to that memory page is considered non-cacheable in the L2 cache, and the L2 cache is bypassed.

### 8.2.3.2 L2 Write Policy

The *outer write policy* memory attribute encoding allows outer caches to be configured as follows:

- *Write-Back* vs. *Write-Through*
- *Write-Allocate* vs. *Read-Allocate*

As previously mentioned, the only write policy the 3rd generation microarchitecture L2 cache supports is write-back, write-allocate. As a result of this, the 3rd generation microarchitecture L2 interprets the below attribute encodings as follows:

- *Outer Write-through* - on 3rd generation microarchitecture, an access of this type is L2 uncacheable
- *Outer Read-Allocate* - on 3rd generation microarchitecture, an access of this type is L2 write-allocate

For more details of the 3rd generation microarchitecture memory attribute encoding, please refer to Chapter 3.0, "Memory Management."



### 8.2.3.3 Shared Memory Attribute

The *shared* (S) memory attribute indicates that the memory region is shared by multiple processors or agents. From an L2 cache perspective, this means that any system bus transaction targeting memory that is marked *shared* and *L2 cacheable* causes the L2 cache to be snooped to see when it contains the desired cache line. When the cache line is present in the L2 cache, notification of this is sent to the bus, and the data is provided directly from the L2 cache to the requesting agent. When the S memory attribute is not set for a given transaction, then the system bus transaction does not trigger any snoop activity in the L2 cache. In this case, when it is desired to keep memory coherent, all accesses to memory being shared by multiple agents must be explicitly handled by software. See Chapter 9.0, "Cache Coherence" for more details on programming 3rd generation microarchitecture for hardware cache coherence.

In summary, hardware cache coherence and L2 snooping occurs on system bus transactions in a 3rd generation microarchitecture-based system with L2 given the following conditions:

- The MMU is enabled
- The L2 is present and enabled
- The bus transaction is L2 cacheable (as specified by the outer memory attribute encoding)
- The bus transaction is to shared memory (S=1)

When hardware cache coherence is supported, bus transactions are snooped, as is described in Section 8.2.2, "Level 2 Cache / BIU Operations Due to System Bus Requests" on page 118. Coherent memory is also supported without an L2 cache, but in this case, all requests are downgraded to both L1 and L2 uncacheable, thereby keeping memory coherent without hardware cache coherence support.

When an access is L1 cacheable, hardware cache coherence is only supported in a 3rd generation microarchitecture-based system when the L1 cache access is treated as write-through, and the access is L2 cacheable. Coherent memory transactions (S=1) on 3rd generation microarchitecture forces the L1 cache to be write-through for the given transaction, even though the page table specifies that the memory location in the L1 cache is write-back. Hardware cache coherence is also supported for L1 uncacheable accesses, just so long as the access is to a memory location that is specified as L2 cacheable.

When a memory access is marked shared (S=1), but the Level 2 cache is either disabled, or the access is L2 non-cacheable, the access is also forced to be L1 uncacheable. This ensures that shared memory remains coherent by making the access entirely uncacheable.

Please refer to Chapter 3.0, "Memory Management" for a full description of the format of the memory management page table.

## 8.2.4 Cache Policies

### 8.2.4.1 Read Miss Policy

The following sequence of events occurs when a L2 cacheable (see Section 8.2.3.1, "L2 Cacheability" on page 119) read operation from the microarchitecture misses the L2 cache:

1. The read operation is promoted to a line fill and a request for the data is made to system memory.
2. A line is allocated in the L2 cache to receive the 32-bytes of fill data. The line selected for replacement is determined by the NRU replacement algorithm (see Section 8.2.5, "Not Recently Used (NRU) Replacement Algorithm" on page 123). When the line chosen for replacement is dirty (M or O state), then the line is scheduled to be written back to memory.
3. When the data requested by the load or instruction fetch is returned from system memory, the data is buffered in the BIU and then forwarded to the microarchitecture.
4. As the data returns from system memory it is also written to the allocated line in L2.

A read operation that is not cacheable in both the L1 and L2 cache issues a read request to system memory via the internal system bus for the exact data size of the original load request. For example, **LDRH** results in a request for exactly two bytes from system memory, **LDR** results in a request for 4 bytes from system memory, etc.

A L1 instruction or data cache line fill that is not L2 cacheable results in a request to system memory for a cache line.

### 8.2.4.2 Write Miss Policy

The following sequence of events occurs when a L2 cacheable (see Section 8.2.3.1, "L2 Cacheability" on page 119) write operation misses the L2 cache. This request is either L1 cacheable, or L1 uncacheable:

1. The write operation is promoted to a line fill and a request for the data is made to system memory, since write allocation is supported by the L2 cache. When the access is to shared memory, the invalidating request is snooped by all system bus caching agents and any data are invalidated when found. Note that when the write operation is to a full cache line (either from a L1 victim or a coalesced store), the write miss allocates a line in the L2 cache, but does not cause a fill request to be made to memory.
2. A line is allocated in the L2 cache after the miss occurs. The line selected for replacement is determined by the NRU replacement algorithm (see Section 8.2.5, "Not Recently Used (NRU) Replacement Algorithm" on page 123). When the line chosen for replacement is dirty (M or O state), then the line is scheduled to be written back to memory.
3. When the line fill data returns, it is merged with the microarchitecture write data (for less than full line width stores), and written to the allocated line in L2.

A write operation that is not cacheable in both the L1 and L2 cache issues a write request to system memory via the internal system bus for the exact data size of the original store operation, assuming the write request does not coalesce with another write operation in the buffers of the L1 data cache.



### 8.2.4.3 L2 Write-Back Behavior

The L2 cache supports write-back caching only. This means that store operations from the L2 cache to memory only occur in the following cases:

- L2 Cache Victimizations - An L2 cache line is evicted. When the evicted line is dirty (M or O state), then it is written back to memory.
- L2 *Clean* and *Clean and Invalidate* Cache Maintenance Operations. These types of operations check to see when the requested data is dirty (M or O state) in the L2 cache. When so, the L2 writes the modified data back to memory, and appropriately update the state of the line. For clean ops, M state is updated to E, and O is updated to S, while for clean and invalidate ops, the state is updated to I.
- Snoop Write-Backs - When a cache coherent memory transaction occurs on the internal system bus, a snoop probe checks the L2 cache to see when the requested data is valid in the L2 cache. When so, it is possible for the L2 cache to intervene, and provide the data directly to the requesting agent via the internal system bus. In some cases, the snoop probe requires the L2 cache to write modified data back to memory (for example, when the requesting agent is not capable of cache ownership and wants to modify the line).

## 8.2.5 Not Recently Used (NRU) Replacement Algorithm

The L2 cache is an 8-way set associative cache. Whenever an L2 cacheable access misses the L2 cache, a line is allocated in the cache so that the requested line is brought in. This allocated line must come from one of the eight ways in the set. The choice of which way to select for replacement is determined by the *Not Recently Used* (NRU) replacement algorithm.

This algorithm targets for replacement the ways within a set that have not been recently used as a priority above those ways that have been accessed more recently. Every cache line has a *Used* bit associated with it. On every microarchitecture access, the Used bit corresponding to the line accessed is set. When all ways but one of the selected set have their Used bits set, then access to the line with the Used bit not set clears the Used bits on the remaining lines of the set, and sets the Used bit of the line just accessed.

To select a replacement candidate in a set on an allocation on miss, two parallel find-firsts are done across the ways of the set. The first looks for an invalid line to overwrite in one of the eight ways. The other looks for a line with the Used bit being clear in one of the eight ways. When an invalid line is found, then that line is used for replacement. Otherwise, the first line with the Used bit not set is used. When a locked line is found in one of the eight ways, then that line is not considered for replacement. Locking is covered in more detail in Section 8.3.5, "Level 2 Cache Locking" on page 128. In summary, for replacement, the L2 cache does the following when performing an line allocate due to a cache miss:

```
if all of the non-locked ways in the selected set are valid

    Use the set's Used bits (one per way) to identify the line to replace

    Update the new line's Used bit to mark the replaced way as used (Used=1)

    If all other ways in the set are marked used, clear all used bits except the one

       for the way that was just replaced.

    Read entire 32 byte line from system memory

    Write the 32 bytes of data from the line fill into that line

    Set the state bits to the appropriate fill state for this line

else {Invalid Lines Exist in the non-locked ways}

    Starting with way 0, identify the first invalid line

    Update the set's Used bits to mark the way as used (Used=1)

    If all other ways in the set are marked used, clear all used bits except the one

       for the way that was just replaced.

    Read entire 32 byte line from system memory

    Write the 32 bytes of data from the line fill into this line

    Set the state bits to the appropriate fill state for this line
```

*Note:* Every time a line in a set is either accessed or filled due to a microarchitecture or push access, the NRU **used** bits for that set are updated. The exception to this is for snoop probes, the NRU used bits are not updated, as the replacement policy is determined by the microarchitecture usage patterns only.



After reset, way 0 is filled first for all the sets, followed in order by way 1 through way 7. Also note that when a line is brought into memory, the state that it is filled in depends upon whether or not the line was from coherent or non-coherent memory. For non-coherent memory, the line is filled in *exclusive* (E) state, essentially acting like a "valid" state. For coherent memory, the line is filled in any of the [M,O,E,S,I] states, depending upon the situation.

## 8.2.6 ECC and Parity Protection

The L2 cache contains *Error Checking and Correction* (ECC) and *Parity Protection* to ensure data integrity on its various arrays as follows:

- The L2 data array is protected by ECC
- The L2 tag array is parity protected.
- The L2 MOESI state array is parity protected

For the data array, there are 10 ECC bits which are calculated and stored each time there is a store operation to the data array. These 10 bits are calculated for the entire 256-bit (32-Byte) line, and are stored along with that line. When a line is read from the L2 cache, the 10-bit ECC syndrome is recomputed on the read data, and then is compared with the stored ECC value. When a single bit error is detected in one of the 256 data bits, the error is seamlessly corrected, and the corrected data is returned to the requester. When a double bit error is detected in the data array, the error is not correctable.

For a single bit error (which has been detected and corrected), the hardware appropriately sets the *Single Bit Error* (bit 7) of the L2 Cache and BIU Error Logging Register 1, indicating that a single bit ECC error has been detected and corrected. In addition to this, the hardware signals an interrupt request to the 3rd generation microarchitecture. (see Section 8.5.2, page 8.0-133).

For a double-bit error (which has been detected only), the hardware appropriately sets the *Double Bit Error* (bit 2) of the L2 Cache and BIU Error Logging Register, indicating that a double-bit ECC error has been detected. In addition to this, the hardware signals a data abort or exception to the microarchitecture. The type of error reporting depends on what type of transaction actually caused the error. (see Section 8.5.2, page 8.0-133).

Note that for either type of ECC error (single or double bit), the physical address is not logged in the Error Address field of the L2 Cache and BIU Error Logging Registers 1 and 2.

For the tag array, there is a single parity bit protecting all 21 (20 for 512 KB) tag bits. Tag parity is checked against the tags from all eight ways on a miss, while only the accessed way's tag is checked on a hit. This ensures that any false miss conditions are detected. When a parity error is detected on a L2 cache access, a data abort or exception is signaled to the microarchitecture for this access. Before reporting the error, the hardware sets bit 0 of the L2 Cache and BIU Error Logging Register (see Section 8.5.2, page 8.0-133). This indicates that a tag parity error has occurred.

For the state/NRU array, there is a single parity bit protecting the 3 state bits. The NRU and lock bits are not parity protected. When a state parity error is detected on an L2 cache access, a data abort or exception is signaled to the microarchitecture for this access. Before reporting the error, the hardware sets bit 1 of the L2 Cache and BIU Error Logging Register (see Section 8.5.2, page 8.0-133). This indicates that a state parity error has occurred.

## 8.3 Level 2 Cache Control

### 8.3.1 Level 2 Cache Memory State After Reset

After processor reset, the L2 cache is disabled, and the state bits are reset such that all lines are invalid, and the NRU bits are all set to zero (not used). Any lines in the L2 cache that were locked before reset are unlocked after reset and therefore is available for replacement.

The L2 cache size configuration after processor reset is determined by reading bits [11:3] of the L2CTYPE register (see Section 7.2.1, "Register 0: ID & Cache Type Registers"). The actual L2 cache size is ASSP specific, but when the L2 is present, it is either 256 KB, or 512 KB.

### 8.3.2 Enabling the L2 Cache

The L2 cache is enabled by writing to bit [26] of the ARM Control register (CP15, register 1). This is the L2 unified cache enable bit (see Section 7.2.2, "Register 1: Control and Auxiliary Control Registers" for more details). This bit resets to 0 on power-up reset. Once written to 1, the L2 cache is enabled. The L2 cache behavior, when switched from the enabled to disabled state is architecturally unpredictable. Therefore, once enabled, the cache must remain enabled. Note that the L2 must be enabled to prior to, or at the same time as the MMU.

## 8.3.3 Invalidate and Clean Operations

The L2 cache provides several invalidate and clean operations which are controlled via coprocessor 15, register 7. These operations are performed on an individual line, either by specifying the address, or by directly specifying the set and way. Refer to Table 73 for a listing of the available L2 cache maintenance operations. Full details of the L2 invalidate and clean operations are found in the Chapter 7.0, "Configuration".

All of the L2 cache maintenance operations that operate on a modified virtual address (MVA) as shown in Table 73 honor address dependencies with other memory operations. However, L2 cache maintenance operations that operate on the entire L2 cache or directly on a set/way do not explicitly honor address dependencies. Therefore, when any specific ordering of these operations is desired with relation to each other, or with relation to other memory operations, an explicit data memory barrier (DMB) operation must be used. For example, when a Clean & Invalidate L2 Cache Line by Set/Way operation is followed by a read operation to the same address, it is strongly suggested that the Clean & Invalidate operation be globally observed before allowing the read operation to proceed. Otherwise, the read operation results in stale data being returned from memory instead of the data that was just "cleaned" from the L2 cache. The way to ensure this is to use a DMB between the Clean & Invalidate operation and the read operation. Full details of the memory ordering model are found in the Chapter 10.0, "Memory Ordering".

**Table 73. L2 Cache Maintenance Operations**

| Function | Opc_1 | CRm | Opc_2 | Data | Instruction |
|---|---|---|---|---|---|
| Invalidate L2 Cache Line | 0b001 | 0b0111 | 0b001 | MVA | MCR p15, 1, Rd, c7, c7, 1 |
| Clean L2 Cache Line | 0b001 | 0b1011 | 0b001 | MVA | MCR p15, 1, Rd, c7, c11, 1 |
| Clean L2 Cache Line | 0b001 | 0b1011 | 0b010 | set/way[a] | MCR p15, 1, Rd, c7, c11, 2 |
| Clean and Invalidate L2 Cache Line | 0b001 | 0b1111 | 0b010 | set/way[a] | MCR p15, 1, Rd, c7, c15, 2 |

a. Refer to Section 7.2.8.7, page 7.0-100 for details on the set/way format.

*Note:* The behavior of these operations is unpredictable when the L2 cache is disabled. When the L2 is not present, these operations perform no-ops.

## 8.3.4 Level 2 Cache Clean and Invalidate Operation

A simple software routine is used to clean and invalidate the *entire* L2 cache, by using the Clean and Invalidate operations listed in Table 73. Specifically, the *Clean and Invalidate Level 2 Cache by Line by Set/Way* operation is used to evict any dirty cache data back to system memory, and to mark all lines as invalid. This operation is used to specifically clean and invalidate a line directly, by providing a set and way. An example of code that cleans and invalidates the cache is found in the *3rd Generation Intel XScale® Microarchitecture Software Design Guide*. Note that when it is desired to clean and invalidate the entire cache, including locked entries, that the entire cache must be first unlocked, prior to the clean and invalidate routine. This is explained in more detail starting in Section 8.3.5.2, "Level 2 Cache Unlock Functions" on page 129.

When the clean and invalidate operation encounters a modified line in the L2 cache, the line is written to system memory before being marked as invalid. As a result, the time it takes to execute a clean and invalidate operation on the entire L2 cache depends on the number of modified lines in present in the L2 cache.

### 8.3.5 Level 2 Cache Locking

Software has the ability to lock lines in the L2 cache. Once a line has been locked in the cache, any access to the line always hits the cache unless it is invalidated. When a line is locked, in general, it is not considered for replacement. However, a locked line is in shared or non-shared memory. Since shared memory lines are invalidated by certain snoop operations including push transactions, it is possible for a locked line to be invalidated by an invalidating snoop operation. To prevent cache holes caused by locked invalid lines, when a line is invalidated, the line becomes unlocked. Therefore, when it is desirable to ensure that a cache line remains locked and not replaced, this line is mapped to non-shared memory.

The NRU algorithm is slightly modified to handle locked lines. In addition to the *used* bit, each cache line has a *lock* bit associated with it. The NRU algorithm first looks for any invalid lines in a given set, regardless of the lock bit state of the lines. The first invalid line, starting from way 0 and searching incrementally through the ways, is chosen for replacement. When no invalid lines are found, the NRU algorithm looks for the first line in a given set that is *not-used* and *not-locked*, starting from way 0 and searching incrementally through the ways.

There is no restriction with regard to which of the eight ways in a given set are locked. However, at most, only seven out of eight ways are available for locking in a given set. Thus, at least one way of the cache is left available for unlocked caching. When eight ways are attempted to be locked, the subsequent replacement behavior is architecturally unpredictable.

#### 8.3.5.1 Level 2 Cache Lock Functions

Level 2 cache locking is line granular and is initiated by having software issue one of two special CP15 instructions:

- fetch & lock
- allocate & lock

When either of these two different methods of locking is used, a line is allocated in the L2 (when not already present), its lock bit is set, and the appropriate action on the data taken, depending upon the lock method.

The CP15 lock instructions are listed below in Table 74.

**Table 74. Level 2 Cache CP15 Lock Operations**

| Operation | Opc_1 | CRm | Opc_2 | Data | Instruction |
|---|---|---|---|---|---|
| Fetch and Lock L2 Cache Line | 0b001 | 0b0101 | 0b000 | MVA | MCR p15, 1, Rd, c9, c5, 0 |
| Allocate and Lock L2 Cache Line | 0b001 | 0b0101 | 0b010 | MVA | MCR p15, 1, Rd, c9, c5, 2 |

The *Fetch and Lock L2 Cache Line* instruction serves to prime the L2 cache for future read operations (loads, instruction fetches, etc.). When the line is already present in the L2 cache, this instruction sets the lock bit for the line. When the line is not present in the L2 cache, this instruction fetches the line from system memory, places it in the L2 cache, and sets the lock bit for the line.

The *Allocate and Lock L2 Cache Line* instruction serves to prime the L2 cache for future write operations. When the line is already present in the L2 cache, this instruction sets the lock bit for the line. When the line is not present in the L2 cache, this instruction allocates a line in the L2 cache, and sets the lock bit for the line. Note that for shared memory, any matching lines in other agents are invalidated in this case. Also note that the data associated with the *allocate and lock* instruction has an unpredictable value, until explicitly written to, meaning subsequent reads before any write reads an unpredictable value.

### 8.3.5.2 Level 2 Cache Unlock Functions

Since an L2 line is unlocked by invalidating the line, the L2 invalidate operations are used to unlock lines in the L2 cache. An *unlock entire L2 cache* instruction is also provided. The CP15 unlock instructions are listed below in Table 75.

**Table 75. Level 2 Cache CP15 UnLock Operations**

| Operation | Opc_1 | CRm | Opc_2 | Data | Instruction |
|---|---|---|---|---|---|
| Invalidate L2 Cache Line | 0b001 | 0b0111 | 0b001 | MVA | MCR p15, 1, Rd, c7, c7, 1 |
| Unlock L2 Cache | 0b001 | 0b0101 | 0b001 | Ignored | MCR p15, 1, Rd, c9, c5, 1 |

The *Invalidate Line by MVA* instruction invalidates and thereby unlock any lines that have a matching address. Note that when it is desired to preserve any modified data in a locked line, a *Clean Line by MVA* instruction is used prior to invalidating/unlocking the line to ensure that the modified data gets written back to memory. To ensure that the clean happens prior to the invalidation, a DMB memory fence must be used, as previously described in *Section 8.3.3, "Invalidate and Clean Operations" on page 127*.

The *Unlock L2 Cache* instruction is used to unlock the entire L2 cache, without having to invalidate the entire L2 cache.

### 8.3.5.3 L2 Cache Maintenance Function Effect on Locked Lines

The remaining L2 cache maintenance operations as described in Table 73 on page 127 each interact with locked lines in different ways.

The *Clean L2 Cache Line by MVA* instruction writes back modified data to system memory when the specified address matches a cache entry with modified data. This behavior is the same, regardless of whether or not the entry is locked. Note that when the line is locked prior to the clean, it remains locked after the clean.

The *Clean L2 Cache Line by Set/Way* instruction does not impact the state of locked lines. This operation appears as a NOP to a locked line.

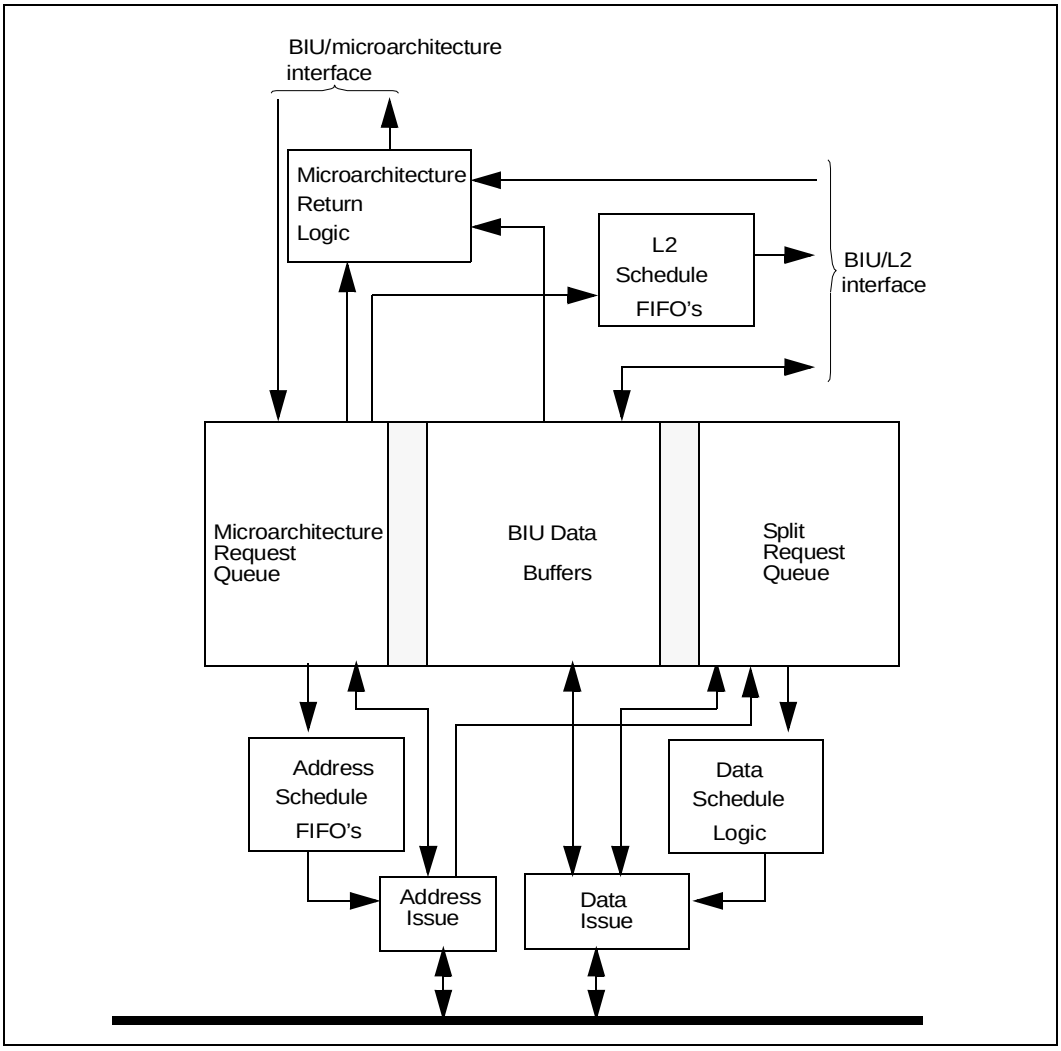
The *Clean & Invalidate L2 Cache Line by Set/Way* instruction also does not impact the state of locked lines. This operation also appears as a no-op to a locked line.

Since the Clean and Clean & Invalidate operations by set/way do not affect locked entries, these are used to clean or clean and invalidate entire sections of the cache, without affecting the status of locked lines.

## 8.4 Bus Interface Unit Operation

The BIU contains several queues and data structures to track 3rd generation microarchitecture requests through completion. These units are illustrated in Figure 10.

**Figure 10. High-Level Block Diagram of BIU**



### 8.4.1 Microarchitecture Request Queue (MRQ)

The *Microarchitecture Request Queue* (MRQ) contains the address and control fields to track a request from the 3rd generation microarchitecture through completion. The MRQ has four types of entries:

- fill entries
- merge entries
- victim entries
- push entries

For each of these types, there is a dedicated number of queue entries:

- Eight Fill Entries to track microarchitecture load requests (data cache load, instruction fetch, uncacheable load, TLB table walks, etc.)
- Four Merge Entries to track outstanding store requests (partial line stores, L1 data cache line evictions, uncacheable stores, etc.)
- Four Victim Entries to track L2 cache line evictions (in other words, modified lines evicted from the L2 cache due to way replacement)
- Two Push Entries to track system bus push cache line requests

All MRQ entries contain a 1:1 correspondence to a particular data buffer entry: fill buffer, merge buffer, victim buffer, or push buffer.

### 8.4.2 Request Scheduling

All microarchitecture requests requiring service by either the L2 cache and/or the system bus are scheduled into specific request FIFOs. Pending requests (in other words, those requests entered into a scheduling FIFO) are prioritized by the associated logic and issued to the L2 cache or the system bus. All cacheable requests are reordered in the BIU due to scheduling priority and/or due to being rejected or retried by the L2 or internal system bus respectively. Ordering is discussed in greater detail in Chapter 10.0, "Memory Ordering". Uncacheable requests are issued to the system bus strictly in-order.

Requests to the bus are prioritized as follows:

1. Victim Buffer Request (only when the victim buffer is full)
2. L2 Cache Miss Request
3. L2 Uncacheable Request
4. System Bus Address Issue Retry Request
5. Uncacheable Request
6. Victim Buffer Request

## 8.5 Level 2 Cache and Bus Interface Unit Register Definitions

Table 76 lists the registers necessary to support the L2 cache and bus interface unit operation.

**Table 76. L2 Unified Cache and BIU Registers**

| Section, Register Name - Acronym (Page) |
|---|
| Section 8.5.1, "Level 2 Cache ID and Cache Type Register" on page 132 |
| Section 8.5.2, "Level 2 Cache and Bus Error Logging Registers (ERRLOG, ERRADRL and ERRADRU)" on page 133 |

### 8.5.1 Level 2 Cache ID and Cache Type Register

The L2 Cache Type Register (L2CTYPE) and L2 System ID Register (L2ID) provide information regarding the configuration of the L2 Cache. These are used to determine whether the L2 cache is present, the size of the cache, the set and way configuration, etc.

The instruction encoding needed to access this register and the format of this register are found in Section 7.2.1, "Register 0: ID & Cache Type Registers".

## 8.5.2 Level 2 Cache and Bus Error Logging Registers (ERRLOG, ERRADRL and ERRADRU)

The L2 cache and bus error logging registers are used to log error information for any error that occurs in the L2 cache or the internal system bus as the result of 3rd generation microarchitecture transactions. ERRLOG contains error logging information, while ERRADRL and ERRADRU contain the physical address associated with the error. Note that addresses are only logged in ERRADRL and ERRADRU for implicit and explicit system bus address errors. When an error occurs, either an interrupt request or an imprecise abort to the microarchitecture occurs, as defined in Table 78, "L2 Cache and BIU Error Logging Register (ERRLOG)" on page 134. The error information associated with any error is logged in ERRLOG. All of the fields in the table are sticky, meaning that once any of the fields are set by hardware, these remain set until cleared by software.

**Table 77. L2 Cache and Bus Error Log Register Access**

| Function | Opc_1 | CRn | CRm | Opc_2 | Instruction |
|---|---|---|---|---|---|
| L2 Cache and BIU Error Logging Register (ERRLOG) | 0b000 | 0b0000 | 0b0010 | 0b000 | MRC p7, 0, Rd, c0, c2, 0<br>MCR p7, 0, Rd, c0, c2, 0 |
| Error Lower Address Register (ERRADRL) | 0b000 | 0b0001 | 0b0010 | 0b000 | MRC p7, 0, Rd, c1, c2, 0<br>MCR p7, 0, Rd, c1, c2, 0 |
| Error Upper Address Register (ERRADRU) | 0b000 | 0b0010 | 0b0010 | 0b000 | MRC p7, 0, Rd, c2, c2, 0<br>MCR p7, 0, Rd, c2, c2, 0 |

## Table 78. L2 Cache and BIU Error Logging Register (ERRLOG)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | | | | | | | | | SBE | OV | DA | EAE | IAE | DBE | SP | TP |

reset value: Writeable bits set to 0

| Bits | Access | Description |
|---|---|---|
| 31:8 | Read-Unpredictable / Write-as-Zero | Reserved |
| 7 | Read/Write | L2 Data Array Single Bit Error (SBE)<br>When set, this indicates that an L2 single bit ECC error has been detected and corrected as the result of a request to L2, and that an interrupt request was sent to the microarchitecture interface. |
| 6 | Read/Write | Overflow (OV)<br>When set, this indicates a second error has occurred after a prior error has already been logged. The overflow pertains to errors captured in bits [5:0] only. In this case, only the overflow bit is set, but all logged error information pertains to the previous error. Note that when two errors occur simultaneously, both errors are logged, and the overflow bit is not set. |
| 5 | Read/Write | Data Abort Error on the system bus (DA)<br>When set, this indicates that a data abort has occurred on the bus as the result of a request to the bus. |
| 4 | Read/Write | Explicit Address Error on the system bus EAE)<br>When set, this indicates that an explicit address error has occurred on the bus as the result of a request to the bus. |
| 3 | Read/Write | Implicit Address Error on the system bus (IAE)<br>When set, this indicates that an implicit address error has occurred on the bus as the result of a request to the bus. |
| 2 | Read/Write | L2 Data Array Double Bit Error (DBE)<br>When set, this indicates that an L2 double bit ECC error has been detected as the result of a request to L2. |
| 1 | Read/Write | L2 State Parity Error (SP)<br>When set, this indicates that an L2 state parity error has occurred as the result of a request to L2. |
| 0 | Read/Write | L2 Tag Parity Error (TP)<br>When set, this indicates that an L2 tag parity error has occurred as the result of a request to L2. |

Note that when any of the errors logged in bits[5:0] occur, these get reported as an Instruction MMU exception, an External Instruction Error exception, an External Abort on translation, or an Imprecise External Data abort. The type of error reporting depends on what type of transaction actually caused the error, in other words, whether or not the request was a load, store, code fetch, or table walk. More details on the 3rd generation microarchitecture exception handling are found in Chapter 2.0, "Programming Model". Note that when an error is detected as the result of an external snoop transaction, the resulting behavior is unpredictable in this case.

Certain system configurations also generate an imprecise external data abort to the microarchitecture via the use of an *asynchronous system bus error* pin. Please note that this type of error is not logged in the ERRLOG register, nor is this type of error set the overflow bit. Systems that use this pin must log the source of the error in a separate system error logging register, such as a memory mapped register, for example. Abort handlers in these systems must not only check the 3rd generation microarchitecture ERRLOG register, but also the appropriate system error register to determine the source of the error, since the ERRLOG register does not capture this type of error. Refer to the 3rd generation microarchitecture implementation options section of the relevant product documentation for more information about whether this pin is used.

When a single bit L2 ECC error occurs (as logged in bit 7), this generates an interrupt request to the microarchitecture interface. It is up to system interrupt controller logic to decide whether or not to take or mask the interrupt.

For address errors on the internal system bus, the physical address associated with the transaction is captured in the error address registers, as shown in Table 79 and Table 80.

### Table 79. L2 Cache and BIU Error Lower Address Register (ERRADRL)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ErrAdrl | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

reset value: Writeable bits set to 0

| Bits | Access | Description |
|---|---|---|
| 31:0 | Read/Write | ErrAdrL<br>Error Low Address[31:0] - when an internal system bus address error occurs, this field contains the lower 32 bits of the physical address of the transaction that generated the error. |

### Table 80. L2 Cache and BIU Error Upper Address Register (ERRADRU)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | ErrAdrU | | | |

reset value: Writeable bits set to 0

| Bits | Access | Description |
|---|---|---|
| 31:4 | Read Unpredictable/Write-as-Zero | Reserved |
| 3:0 | Read/Write | ErrAdrU<br>Error High Address[35:32] - when an internal system bus address error occurs, this field contains the upper 4 bits of the physical address of the transaction that generated the error. |



# 9.0 Cache Coherence

## 9.1 Introduction

This chapter defines the hardware-based cache coherence support in the 3rd generation Intel XScale® microarchitecture (3rd generation microarchitecture or 3rd generation).

Whenever writeable data in memory is placed in a cache for faster access, the issue of maintaining coherence between the data in the cache and that in main memory arises. For uniprocessor systems where only the processor accesses memory, this problem is typically solved in one of two ways:

- The first method is forcing all memory writes to update both cache and memory. This scheme is typically called write-through caching.
- The other common alternative is by tracking the modification (or *dirty*) status of the cached data, and ensuring dirty data is written back to memory when the cache line is replaced. This method is typically called write-back caching.

Cache coherence maintenance gets more complicated in systems where multiple agents access memory, and one or more agents have private cache(s) that contain copies of writeable data that multiple agents access.

A common example of such a system is a uniprocessor system, where non-processor agents, such as Direct Memory Access (DMA) agents, access memory. Hardware-based cache coherency ensures that the processor always reads the fresh data written by DMA, rather than a stale copy the processor has cached earlier.

Depending on the particular ASSP, 3rd generation microarchitecture is accompanied with a Level-2 cache (L2). Only products that include an L2 provide hardware cache coherency. Products without an L2 cache do not have hardware support for cache coherency. 3rd generation microarchitecture always have an *L1D* (Level-1 Data cache) and an *L1I* (Level-1 Instruction cache).

An L2 cache is a necessary, but not sufficient condition for hardware coherency support. Consult the relevant relevant product documentation to see whether the ASSP supports hardware coherency.

## 9.2 3rd Generation Microarchitecture Hardware Cache Coherence Solutions

3rd generation microarchitecture with L2 cache supports hardware cache coherence when enabled in a product. This section describes how that coherency operates.

### 9.2.1 Hardware Cache Coherence Configurations

#### 9.2.1.1 Configuration through Page Table Attributes

Hardware coherence support is configured at a page granular basis in memory, allowing ASSPs, operating systems, and other solution providers to partition memory into hardware coherent and non-hardware coherent areas.

When hardware based cache coherence is desired, the *Shared* memory attribute needs to be set in the page table entry (PTE) by writing '1' to its *S* bit and the PTE needs to be configured to enable L2 caching.

For a memory page where the Shared attribute is not set, all data coherence needs to be guaranteed by software, for example, by explicit cleaning of modifications.

Table 81 lists the page attribute encodings for cache coherence and the resulting coherence behavior when L2 is present and enabled.

**Table 81. Page Attributes Configuring Coherence and Cacheability**

| Shared Attribute | PTE: L1 Cacheable | PTE: L2 Cacheable | Coherence Type |
|---|---|---|---|
| 0 | 0 | 0 | Coherent due to no caching[a] |
| 0 | 0 | 1 | Non-Coherent |
| 0 | 1 | 0 | Non-Coherent |
| 0 | 1 | 1 | Non-Coherent |
| 1 | 0 | 0 | Coherent due to no caching |
| 1 | 0 | 1 | Coherent: hardware enforces L2 cache coherency |
| 1 | 1 | 0 | L1D defaults to uncacheable<br>Coherent due to no caching |
| 1 | 1 | 1 | Coherent: hardware enforces L1 and L2 cache coherency |

a. For forward compatibility, it is recommended pages of this type be instead replaced with the Shared Attribute = '1' analog



#### 9.2.1.2 Shared Attribute Precedence

The Shared attribute takes precedence over cacheability.
3rd generation microarchitecture does not support cache coherent operation for pages cacheable only in the L1D. When such a page is marked as shared, it is in fact treated as uncacheable in the L1D, and thus be made coherent by being made uncacheable.

For pages cacheable in both the L1D and L2, whenever the page attribute indicates shared, the L1D defaults to a write-through mode of operation.

#### 9.2.1.3 Non-coherent L1 Instruction Cache

The L1 instruction cache is not hardware coherent with the L1 data cache. Any memory modification that needs to be visible to the instruction cache requires explicit software control. More details on software controlled instruction coherence is found in3rd generation microarchitecture *EAS,* Chapter 4.0, "Instruction Cache".

#### 9.2.1.4 Swap Behavior

The **SWP** and **SWPB** instructions are fully supported to cache coherent space, and architecturally behave similarly to a write except that these also returns the memory value being swapped back to the register file. The swap is actually executed as a read followed by a write with 3rd generation microarchitecture guaranteeing atomicity of the swap by preventing accesses by other agents to the swapped memory location between the read and the write.

## 9.2.2 L1D Coherence

In the cache coherent mode of operation, the 3rd generation microarchitecture L1 data cache operates in write-through mode. This means that instruction writing memory updates both the L1D and the L2. Instructions that read memory still obtain their value from the L1D cache.

### 9.2.2.1 Coherent Read Behavior

All ARM load instructions addressing shared memory cacheable in both L1D and L2 access the L1D for a read operation. When the read hits (in other words, finds a matching line, then the value from the cache is simply returned to the appropriate register.

When the read misses, a read request propagates to the L2. When the L2 or memory returns the data, a line in the L1 is filled and the read data is returned to the appropriate register.

A read due to a swap (**SWP**) instruction is forced to miss L1D, regardless of the presence of the accessed line in L1D.

### 9.2.2.2 Coherent Write Behavior

All ARM store instructions addressing shared memory cacheable in both L1D and L2 accesses the L1D for a write operation. All such writes eventually propagate out of the L1D and writes through to L2 since L1D is write-through for cacheable shared memory. The L1D coalesces writes, so several writes are coalesced before being written through. More details on L1D write coalescing behavior is found in Chapter 6.0, "Data Cache".

### 9.2.2.3 Coherent Line Allocate Instruction Behavior

3rd generation microarchitecture features a line allocate instruction that allocates an L1D cache line with a specified address when the line is not already valid in L1D. Reads and writes to the allocated line are handled like other coherent reads and writes.

Note, that after an allocation, a read to any location contained within the allocated line by any agent in the system, returns unpredictable data value, unless a prior write to that location has been observed by the reading agent.

### 9.2.2.4 Replacement Behavior

A valid L1D line is overwritten on replacement without a memory write-back because the write-through policy ensures the replaced line is coherent with either L2 or memory.

When 3rd generation microarchitecture notices another agent reading or writing data in the L1D, then it invalidates that line in the L1D. This action never results in data loss because the L1D is write-through.

### 9.2.2.5 Locking and Shared Attributes

The L1 caches allow lines to be locked to prevent replacements. *Chapter 4.0, "Instruction Cache" and* Chapter 6.0, "Data Cache" have the full details. Memory locations configured as sharable in their PTEs are not locked in the L1 data cache. The result of attempting to lock shared memory locations in the L1 data cache is unpredictable.

### 9.2.3 L2 Coherence

In the cache coherent mode of operation (shared memory, cacheable in L2), the 3rd generation microarchitecture L2 data cache operates in write-back mode utilizing the MOESI coherence protocol.

#### 9.2.3.1 Coherent L2 Fetch and Lock

features an L2 Fetch and Lock instruction, that results in the addressed line becoming locked in the L2. When executed on a cache coherent location, from a coherence perspective, this instruction appears as and is treated like a read.

#### 9.2.3.2 Snoop Behavior

Writes by external agents, when these hit L2, cause a write-back and invalidation of a targeted line. Writes by an external agent to a line locked into the L2 results in unlocking that line. The contents of the line are valid but, like other lines, eligible for eviction.

#### 9.2.3.3 Intervention

When a memory request from another agent is snooped and found to hit in a

L2, microarchitecture features the capability to intervene and provide the line to the requestor instead of the line being provided from memory. This is both a power and a performance optimization because of the resulting reduced memory traffic.

#### 9.2.3.4 Push Cache

allows the L2 cache to be directly allocated and/or updated from another agent. This mode of L2 cache operation is referred to as a *push cache*. The size of the data pushed must be

cache line size (32B) and must be aligned on a a cache line (32B) boundary. Pushing is only allowed to memory locations marked as shared and L2 cacheable in their page attributes.

The data associated with the "push" becomes the value for the addressed locations.

See the relevant product documentation for information about how peripherals take advantage of push cache.

## 9.3 Non-Hardware Coherent Mode

### 9.3.1 Introduction

Prior generations of Intel XScale® microarchitecture, as well as other existing *ARM Architecture Version 5TE Specification* or earlier architecture implementations, do not support hardware-based cache coherence. When desired,

supports this non-hardware coherent mode by allowing pages to be marked as non-shared (see Section 9.2.1). Existing operating systems use this mode by default.

In this mode, coherence among multiple caching agents, when desired, needs to be maintained by software. For example, dirty data needs to be explicitly flushed by software at synchronization points. In this usage model, writeable data is only shared through synchronization.

### 9.3.2 L1 Data Cache Operation in Non-Cache Coherent Mode

#### 9.3.2.1 Read Behavior

The read behavior is identical to cache coherent mode.

#### 9.3.2.2 Write Behavior

The write behavior in write-through non-coherent mode, is similar to the coherent mode, in the sense that all writes propagate out of the L1 data cache regardless of hit status, and that write misses do not allocate. In non-coherent mode, the L1D also supports a write-back mode of operation, where a write hit to aline sets a *dirty* state for the line.

### 9.3.3 L2 Data Cache Operation in Non-Cache Coherent Mode

#### 9.3.3.1 Read Behavior

Software issuing reads to the L2 see the same behavior in both coherent and non-coherent memory.

#### 9.3.3.2 Write Behavior

The L1D uses a write-through policy when acting on a non-coherent memory region. Unlike the cache coherent L2 write behavior described previously, writes to L2 in non-coherent memory space also result from dirty L1D lines written back. These are treated as any other write.

# 10.0 Memory Ordering

## 10.1 Introduction

Memory ordering models, also known as memory consistency models, specify the order and visibility of memory accesses in systems where multiple processors and agents access shared memory and memory-mapped I/O devices. A variety of memory ordering models have been proposed in academia and implemented in commercial processors.

In this chapter, the memory ordering model supported by the 3rd generation Intel XScale® microarchitecture (3rd generation microarchitecture or 3rd generation) is specified. 3rd generation implements a *weak* consistency model, because it normally rearranges memory operations as needed to realize better performance. 3rd generation automatically honors data dependencies; programmers also explicitly force ordering with *fence* instructions.

Figure 11 is an example of how 3rd generation microarchitecture reorders memory operations. Assume that code fragment (a) in the example represents the memory operations of a program. The programmer has specified that memory pointed to by R0 is updated, followed by the location four bytes beyond the address in R0.

Because it implements a weak consistency model, 3rd generation microarchitecture updates memory in the order implied by fragment (b) of Figure 11. The microarchitecture chooses to do this for efficiency reasons. When the programmer is writing a device register, for example, then this results in unexpected behavior. Thus, a programmer doing system-level programming (interacting with devices) needs to be aware of 3rd generation microarchitecture consistency model.

**Figure 11. Memory Ordering Example**

| (a) Program Order | (b) Effect of Reordering |
|---|---|
| STR R4, [R0] | STR R5, [R0, #4] |
| STR R5, [R0, #4] | STR R4, [R0] |

Subsequent sections of this chapter describe the ordering model more formally, and give information on how programmers enforce a particular order when needed. Programmers writing "normal" application code need not be concerned with these issues. Programmers that need to understand the 3rd generation microarchitecture ordering model include those dealing with features like the following:

- Memory-mapped I/O
- Peripherals with side effects
- Memory shared by multiple peripherals

## 10.2 Visibility: Observation and Global Observation

A precise definition of visibility of a memory access is necessary for defining an ordering model. In 3rd generation microarchitecture, there are two notions of visibility: *observation* and *global observation*.

In the following definitions of observation and global observation, a memory system agent is any agent that reads from or write to memory including one or more 3rd generation microarchitecture processor agents as well as non-processor agents such as Direct Memory Access (DMA) agents.

### 10.2.1 Normal (Memory-like) Memory

The following apply to accesses to memory that does not contain I/O devices.

- A write to a location in normal memory is said to be *observed* by a memory system agent when a subsequent read of the location by the *same* memory system agent returns the value written by the write.
- A write to a location in normal memory is said to be *globally observed* when a subsequent read of the location by *any* memory system agent returns the value written by the write.
- A read from a location in normal memory is said to be *observed* by a memory system agent when a subsequent write to the location by the *same* memory system agent does not affect the value returned by the read.
- A read from a location in normal memory is said to be *globally observed* when a subsequent write to the location by *any* memory system agent does not affect the value returned by the read.

*Note:* The concept of observation applies to both *shared* and *non-shared* normal memory, while the concept of global observation only applies to shared normal memory.

### 10.2.2 I/O-like Memory

- A read or write to a location in I/O-like memory is said to be *observed* by a memory mapped peripheral device when the read or write begin to affect the state of the peripheral device and trigger any side effects that affect other peripheral devices and/or memory.
- A read or write to a location in I/O-like memory is said to be *globally observed* when the read or write has updated the state of the target peripheral device(s), and all resulting side effects that affect other peripheral devices and/or memory have become visible to the entire system.

*Note:* 3rd generation microarchitecture cannot ensure global observation of transactions to memory-mapped I/O because completion of side effects are not visible to the processor. See specific product documentation for information on how software ensures a memory-mapped I/O device has completed side effects.

### 10.2.3 Memory Types

Memory is generally segregated into three types: *Normal, Device,* and *Strongly Ordered*. Each addressed page in memory is placed into one of these categories by settings in the MMU descriptor of that page. See Section 3.2.4 for information on how these memory type attributes are specified in a page table.

Normal memory is, as the name implies, the type of memory used for regular program code and data. This memory type has the weakest ordering restrictions on it: hardware rearranges memory operations with impunity unless doing so violates the data dependencies implicit in the program. Normal memory is cacheable or uncacheable.

Device memory is intended for use with memory-mapped peripherals. Memory operations directed to this type of memory do not pass others directed to the same type of memory. However, the hardware permits Normal memory operations to pass Device memory operations.

Strongly Ordered memory has the most exacting ordering requirements. No memory operations of any type are allowed to pass memory operations to Strongly Ordered memory.

Device memory and Strongly Ordered memory are used when the programmer wishes to treat the target as I/O. In this chapter, sometimes these memory types are collectively called *I/O-like*.

Section 3.2.4 explicitly calls out page table entry encodings for Device memory and Strongly ordered memory. All other memory types are considered Normal.

### 10.2.4 Data Dependence

Accesses to the same normal memory location from the same
3rd generation microarchitecture honors data dependence. So, reads to a normal memory location *subsequent* in program order to a prior write to the same location, see the value updated by the write (*Read after Write)*. When there are two writes in program order to a normal memory location, the final value of the memory location after the two writes complete is the value updated by the second write in program order (*Write after Write*). When reads to a normal memory location precede a write in program order, the reads sees the value prior to the update by the write *(Write after Read)*.

## 10.3 Write Coalescing and Ordering

3rd generation microarchitecture allows write coalescing, which allows writes to Normal memory to be coalesced to improve write bandwidth. The effect of coalescing on ordering is two fold:

- First, writes coalesce around intervening reads in program order, thereby reordering the writes with respect to the reads.
- Second, multiple writes are globally observed simultaneously due to coalescing.

Write coalescing is one reason that Normal memory accesses are weakly ordered. Note that Normal memory is further specified as cacheable or uncacheable in a page table entry. In both cases 3rd generation microarchitecture uses write coalescing, so ordering is weak.

## 10.4 Instructions with Ordering Constraints

### 10.4.1 Safety Nets and Synchronization

Programs and systems often depend on certain memory operations to complete and become visible in a specified order and thus mechanisms are available in the 3rd generation microarchitecture memory ordering model to enforce a specified ordering where needed. These mechanisms are referred to as *fences*, to show that reordering is restricted "across" these fences. This section describes two *explicit* ordering fence instructions that impose an order, and also describes implicit fencing behavior of other instructions of 3rd generation microarchitecture.

### 10.4.2 Explicit Fence Instructions: DMB and DWB

An explicit ordering fence instruction restricts the order in which memory operations complete before and after it. The instruction itself does not access memory.

#### 10.4.2.1 Data Memory Barrier (DMB)

The *Data Memory Barrier (DMB),* specifies that all explicit normal memory accesses by instructions in program order *prior* to the DMB must be *globally observed* and all explicit I/O-like memory accesses must be *observed* by the target devices prior to any explicit memory accesses by instructions *subsequent* to the DMB in the program being *observed*.

Note that non-explicit accesses, such as instruction fetches and page walks, are not ordered by the DMB instruction. The DMB has no execution ordering constraint on non-memory access instructions.

#### 10.4.2.2 Data Write Barrier (DWB)

The *Data Write Barrier (DWB)* instruction specifies that completion of this instruction implies that all explicit Normal memory writes in program order *prior* to the DWB must be *globally observed*, and all I/O-like memory writes prior to the DWB must be *observed.* No instruction subsequent to DWB executes until DWB completes. Also, like the DMB, DWB does not impose any restrictions on non-explicit accesses.

#### 10.4.2.3 Effect of DMB and DWB on Write Coalescing

Because both of these fences require all prior writes be globally observed, before subsequent accesses, an implication of these fence executions is that no write prior in program order to a DMB or DWB coalesce with a write subsequent in program order.

### 10.4.3 Instruction Fence Instruction: Prefetch Flush (PF)

The Prefetch Flush (PF) instruction is needed in addition to a data memory fence instruction to enforce ordering between data and instruction accesses. The retirement of a PF guarantees the following:

- all outstanding instruction memory accesses and instruction page table walks have completed.
- all younger instructions in the processor pipeline after the PF are flushed.
- the next instruction to execute after the PF is fetched from cache or memory only after the above two conditions are satisfied.

In 3rd generation microarchitecture, a PF is required to be used in conjunction with a DWB to ensure any data memory hierarchy modifications (by **STR**, **SWP**, etc.) are visible to instruction memory hierarchy accesses as well as page table walks for both instruction and data page table entries.

### 10.4.4 Instruction Encodings

DMB, DWB and PF are encoded as these move to the coprocessor registers. The encodings are shown in Table 82.

DMB, DWB and PF are encoded as these move to coprocessor registers.

**Table 82. DMB, DWB and PF Instruction Encodings**

| Fence | opcode_2 | CRm | Rd | Instruction |
|---|---|---|---|---|
| Date Memory Barrier (DMB) | 0b101 | 0b1010 | Ignored | MCR p15, 0, Rd, c7, c10, 5 |
| Data Write Barrier (DWB) | 0b100 | 0b1010 | Ignored | MCR p15, 0, Rd, c7, c10, 4 |
| Prefetch Flush (PF) | 0b100 | 0b0101 | Ignored | MCR p15, 0, Rd, c7, c5, 4 |

### 10.4.5 Usage Examples of Fence Instructions

Compare the instruction sequence of Figure 11 with Figure 12. The code stream now has a DMB inserted between the stores. These accesses are now guaranteed to take effect in program order.

**Figure 12. Using DMB to Enforce Ordering**

```
(a) Program Order      (b) Observed Order

STR R4, [R0]           STR R4, [R0]

DMB

STR R5, [R0, #4]       STR R5, [R0, #4]
```

*Note:* The same program behavior is also achieved by replacing the DMB in the code a DWB. Had a memory-read been involved, then DMB is mandatory (DWB is only guaranteed to operate on memory-writes).

A programmer that wishes to ensure a specific order of observed memory accesses must utilize the appropriate fencing instructions. For example, when Figure 12 were updating device registers, then the system does not operate correctly when the stores were reordered.

Figure 13 shows an example of using PF to make visible an instruction modification. Note that in this example invalidation of the instruction cache and the BTB is also done, because it is assumed that the modified code location is cacheable and branch prediction has been enabled. Note that the DWB cannot be replaced by a DMB in this case, because unlike the DMB, the DWB disallows any instructions later in program order from *executing* until all stores prior in program order have been globally observed. Thus, only a DWB ensures that the NewCode update be observed by the instruction fetch to NewCode.

**Figure 13. Using PF to Enforce Data Write to Instruction Fetch Ordering**

```
STR R1, [NewCode];

INVIC [NewCode]; Invalidate I-Cache line containing code

INVBTB; Invalidate BTB, in case a branch moved



DWB ; Will not retire until STR observed

PF  ; Ensures any speculatively fetched...

    ; instructions are flushed
```

## 10.4.6 Implicit Fences

There are Intel XScale® microarchitecture instructions that have implicit fencing behavior. Unlike the explicit fence instructions, these instructions actually operate on memory but in addition have fencing behavior.

### 10.4.6.1 Swap

The **SWP** (or **SWPB**) instruction exchanges a value in a general register with a value in memory. When **SWP** operates on cache coherent memory space, the exchange is guaranteed to be atomic with respect to other memory agents.

A prevalent use of the **SWP** instruction is to implement semaphore-based thread synchronization. In this type of usage, it is important that memory operations subsequent to the **SWP** instruction in program order are not reordered to be observed prior to the **SWP** being globally observed, because doing so inadvertently allows code in a protected critical section to be executed when such execution is not allowed when program order was observed. The implicit fencing behavior of a **SWP** is defined to be that no explicit memory access *subsequent* in program order to a **SWP** to a memory location is *observed* until the **SWP** is *globally observed.*

A **SWP** to a memory region with Shared attributes in the page table are an implicit fencing operation. See Section 3.2.4 for information on how a page table entry is used to express a Shared region of memory. When **SWP** targets a Shared memory region, it is equivalent to the sequence: READ, WRITE, DMB.

A **SWP** to a region not configured as Shared does not ensure an implicit fence and is equivalent to: READ, WRITE.

### 10.4.6.2 Explicit Accesses to Strongly Ordered Memory

Explicit accesses to Strongly Ordered memoryact as implicit DMB fences, requiring all prior explicit normal memory accesses in program order to be *globally observed* and all prior explicit I/O-like memory accesses to be *observed* by the target device before the Strongly Ordered access is *observed*. Further, all subsequent explicit memory accesses in program order are only *observed* after the Strongly Ordered access is *globally observed*. Thus strongly ordered memory accesses are not only ordered among these, but are ordered with respect to all explicit.

## 10.5 Ordering Table

Table 83 summarizes the 3rd generation microarchitecture memory ordering rules described in this chapter.

- "-" indicates no ordering requirement exists.
- "O" indicates the operation subsequent in program order is ordered with respect to the operation prior in program order.

**Table 83. Ordering Rules**

| Prior in Program Order | | Subsequent in Program Order | | | | | |
|---|---|---|---|---|---|---|---|
| | | Normal | Device | | Strongly Ordered | DMB | DWB[a] |
| | | | Non-Shared | Shared | | | |
| Normal | | - | - | - | O | O | O |
| Device | Non-Shared | - | O | - | O | O | O |
| | Shared | - | - | O | O | O | O |
| Strongly Ordered | | O | O | O | O | O | O |
| DMB | | O | O | O | O | O | O |
| DWB | | O | O | O | O | O | O |

a. Ordered with respect to prior writes only (not prior reads)

As noted in Section 10.4.6, **SWP** instructions also act as fences. See that section for details.

## 10.6 I/O Ordering

Use of a fencing instruction (implicit or explicit) is not sufficient to ensure ordered execution of accesses to memory mapped locations which have side effects. The fences only ensure *observation* to the target devices for I/O like memory accesses, but not *global observation,* so there is no constraint placed on the side-effects in these devices. For these cases, software polling is required to determine when the side effects have completed. See your relevant product documentation for details on how to ensure peripheral accesses have taken effect.

Memory accesses are not ordered with respect to coprocessor accesses. Software ensures that a coprocessor write has occurred by reading from the CP location and creating a dependency by using the value.

## 10.7 Ordering Cache Management Operations

3rd generation microarchitecture has instructions to perform cache management operations, such as Clean, Invalidate, and Clean and Invalidate that operate either on a cacheable memory location or on a specified cache index. Chapter 7.0, "Configuration", details all these instructions.

All of the L2 cache management operations that operate on a modified virtual address (MVA) as shown in Table 73 in the EAS honor address dependencies with other memory operations. However, the L2 cache maintenance operations that operate on the entire L2 cache or directly on a set/way do not honor data dependencies. Therefore, when any specific ordering of these operations is desired with relation to each other, or with relation to other memory operations, an explicit data memory barrier (DMB) operation must be used.

This requirement is illustrated by the following example. When a Clean&Invalidate L2 Cache Line by Set/Way operation is followed by a LDR to a location within the cleaned line, the only method to ensure data dependency is honored between the two operations is to enforce that Clean&Invalidate operation be globally observed before allowing the LDR operation to proceed. Otherwise, the LDR reads in stale data from memory instead of the data that was just cleaned from the L2 cache. The way to ensure this is to use a DMB between the Clean&Invalidate operation and the LDR operation.

```
Clean & Invalidate L2 by Set/Way    ; line at address X

DMB                                 ; enforces ordering

LDR                                 ; loads from address XX
```



# 11.0 Performance Monitoring

This chapter describes the performance monitoring unit (PMU) facility of the 3rd generation Intel XScale® microarchitecture (3rd generation microarchitecture or 3rd generation). The events that are monitored provide performance information for compiler writers, system application developers and software programmers.

## 11.1 Overview

3rd generation microarchitecture hardware provides four 32-bit performance counters that allow four unique events to be monitored simultaneously. In addition, 3rd generation 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.

3rd generation microarchitecture monitors either occurrence events or duration events. When counting occurrence events, a counter is incremented each time a specified event takes place; when measuring duration, a counter counts the number of clocks (microarchitecture, bus or L2) that occur while a specified condition is true. When any of the five counters overflow, an interrupt request occurs when enabled.

Subsequent handling of PMU interrupt requests is ASSP defined, which typically contains an interrupt controller to manage interrupt priority, masking, steering to FIQ or IRQ, etc. Refer to the 3rd generation microarchitecture implementation options section of the relevant product documentation for more details.

Each counter has its own interrupt request enable. The counters continue to monitor events even after an overflow occurs, until disabled by software. Each of these counters are programmed to monitor any one of various events.

To further augment performance monitoring, the 3rd generation microarchitecture clock counter is used to measure the execution time of an application. This information combined with a duration event feeds back a percentage of time the event occurred with respect to overall execution time.

All of the performance monitoring registers are accessible through Coprocessor 14 (CP14). Refer to Table 84 for more details on accessing these registers with **MRC** and **MCR** coprocessor instructions. These registers are only accessible from privileged modes. User mode access results in an undefined instruction exception. Note that these registers cannot be accessed with **LDC** or **STC** coprocessor instructions.

**Table 84. Performance Monitoring Registers**

| CRn | CRm | Access | Description | Cross-Reference |
|---|---|---|---|---|
| 0 | 1 | Read / Write | Performance Monitor Control Register | Section 11.2.1, page 11.0-154 |
| 1 | 1 | Read / Write | Clock Counter Register | Section 11.2.2, page 11.0-155 |
| 4 | 1 | Read / Write | Interrupt Enable Register | Section 11.2.3, page 11.0-156 |
| 5 | 1 | Read / Write | Overflow Flag Register | Section 11.2.4, page 11.0-157 |
| 8 | 1 | Read / Write | Event Selection Register | Section 11.2.5, page 11.0-158 |
| 0 | 2 | Read / Write | Performance Count Register 0 | Section 11.2.6, page 11.0-159 |
| 1 | 2 | Read / Write | Performance Count Register 1 | |
| 2 | 2 | Read / Write | Performance Count Register 2 | |
| 3 | 2 | Read / Write | Performance Count Register 3 | |

## 11.2 Register Description

### 11.2.1 Performance Monitor Control Register (PMNC)

**Table 85. Performance Monitor Control Functions (CRn = 0, CRm = 1)**

| Function | CRn | CRm | Instruction |
|---|---|---|---|
| Performance Monitor Control Register (PMNC) | 0b0000 | 0b0001 | MRC p14, 0, Rd, c0, c1, 0<br>MCR p14, 0, Rd, c0, c1, 0 |

The performance monitor control register (PMNC) is a coprocessor register that:

- contains the PMU ID
- extend CCNT counting by six more bits (cycles between counter rollover = $2^{38}$)
- reset all counters to zero
- enables the clock count and all performance counters

Table 86 shows the format of the PMNC register.

**Table 86. Performance Monitor Control Register**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ID | | | | | | | | | | | | | | | | | | | | | | | | | | | M | D | C | P | E |

reset value: [2:0] = 0b0000, [31:24] = 0b00100100, others unpredictable

| Bits | Access | Description |
|---|---|---|
| 31:24 | Read / Write-Ignored | Performance Monitor Identification (ID)<br>3rd generation microarchitecture = 0x24 |
| 23:5 | Read-Unpredictable / Write-As-Zero | Reserved |
| 4 | Read/Write | Performance Counter Disable (M)<br>0 = performance counters are enabled (the E bit must also be enabled)<br>1 = performance counters are disabled |
| 3 | Read / Write | Clock Counter Divider (D)<br>0 = CCNT counts every clock cycle<br>1 = CCNT counts every 64$^{th}$ clock cycle |
| 2 | Read-as-0 / Write | Clock Counter Reset (C)<br>0 = no action<br>1 = reset the clock counter to '0x0' |
| 1 | Read-as-0 / Write | Performance Counter Reset (P)<br>0 = no action<br>1 = reset all performance counters to '0x0' |
| 0 | Read / Write | Enable (E)<br>0 = all counters are disabled<br>1 = all counters are enabled |

## 11.2.2 Clock Counter (CCNT)

**Table 87. Clock Count Functions (CRn = 1, CRm = 1)**

| Function | CRn | CRm | Instruction |
|---|---|---|---|
| Clock Counter Register (CCNT) | 0b0001 | 0b0001 | MRC p14, 0, Rd, c1, c1, 0<br>MCR p14, 0, Rd, c1, c1, 0 |

The clock counter is a counter that increments once for every microarchitecture clock or every 64 microarchitecture clocks when enabled (depending on the setting of the PMNC.D bit). The format of CCNT is shown in Table 88. The clock counter is reset to '0' by writing a '1' to the PMNC.C bit or is set to a predetermined value by directly writing to it. When CCNT reaches its maximum value 0xFFFFFFFF, the next increment causes it to roll over to zero and set the CCNT overflow flag bit (FLAG.C) in the Overflow Flag Status Register. An interrupt request occurs when enabled via the INTEN.C bit in the Interrupt Enable Register.

**Table 88. Clock Count Register (CCNT)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Clock Counter | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| reset value: unpredictable | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

| Bits | Access | Description |
|---|---|---|
| 31:0 | Read / Write | 32-bit clock counter<br>Reset to '0' by PMNC register. When the clock counter reaches its maximum value 0xFFFFFFFF, the next increment causes it to roll over to zero and generate an interrupt request when enabled. |

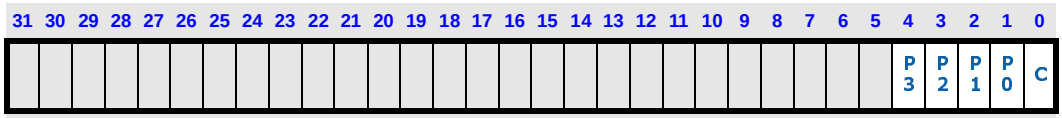
### 11.2.3 Interrupt Enable Register (INTEN)

**Table 89. Interrupt Enable Functions (CRn = 4, CRm = 1)**

| Function | CRn | CRm | Instruction |
|---|---|---|---|
| Interrupt Enable Register (INTEN) | 0b0100 | 0b0001 | MRC p14, 0, Rd, c4, c1, 0<br>MCR p14, 0, Rd, c4, c1, 0 |

Each counter generates an interrupt request when it overflows. The Interrupt Enable Register (INTEN) controls the interrupt request for each counter.

**Table 90. Interrupt Enable Register**




**reset value:** [4:0] = 0b00000, **others unpredictable**

| Bits | Access | Description |
|---|---|---|
| 31:5 | Read-Unpredictable / Write-As-Zero | Reserved |
| 4 | Read / Write | PMN3 Interrupt Enable (P3)<br>0 = disable interrupt<br>1 = enable interrupt |
| 3 | Read / Write | PMN2 Interrupt Enable (P2)<br>0 = disable interrupt<br>1 = enable interrupt |
| 2 | Read / Write | PMN1 Interrupt Enable (P1)<br>0 = disable interrupt<br>1 = enable interrupt |
| 1 | Read / Write | PMN0 Interrupt Enable (P0)<br>0 = disable interrupt<br>1 = enable interrupt |
| 0 | Read / Write | CCNT Interrupt Enable (C)<br>0 = disable interrupt<br>1 = enable interrupt |

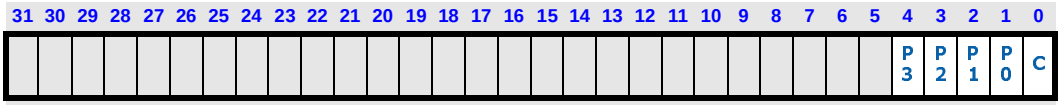
## 11.2.4 Overflow Flag Status Register (FLAG)

**Table 91. Overflow Flag Status Functions (CRn = 5, CRm = 1)**

| Function | CRn | CRm | Instruction |
|---|---|---|---|
| Overflow Flag Status Register (FLAG) | 0b0101 | 0b0001 | MRC p14, 0, Rd, c5, c1, 0<br>MCR p14, 0, Rd, c5, c1, 0 |

The Overflow Flag Status Register (FLAG) identifies which counter has overflowed and also indicates an interrupt has been requested when the corresponding interrupt enable bit (contained within INTEN) of the overflowing counter is asserted. An overflow is cleared by writing a '1' to the corresponding overflow bit.

**Table 92. Overflow Flag Status Register**




**reset value:** [4:0] = 0b00000, **others unpredictable**

| Bits | Access | Description |
|---|---|---|
| 31:5 | Read-Unpredictable / Write-As-Zero | Reserved |
| 4 | Read / Write | PMN3 Overflow Flag (P3)<br>Read Values:<br>0 = no overflow<br>1 = overflow has occurred<br>Write Values:<br>0 = no change<br>1 = clear this bit |
| 3 | Read / Write | PMN2 Overflow Flag (P2)<br>Read Values:<br>0 = no overflow<br>1 = overflow has occurred<br>Write Values:<br>0 = no change<br>1 = clear this bit |
| 2 | Read / Write | PMN1 Overflow Flag (P1)<br>Read Values:<br>0 = no overflow<br>1 = overflow has occurred<br>Write Values:<br>0 = no change<br>1 = clear this bit |
| 1 | Read / Write | PMN0 Overflow Flag (P0)<br>Read Values:<br>0 = no overflow<br>1 = overflow has occurred<br>Write Values:<br>0 = no change<br>1 = clear this bit |
| 0 | Read / Write | CCNT Overflow Flag (C)<br>Read Values:<br>0 = no overflow<br>1 = overflow has occurred<br>Write Values:<br>0 = no change<br>1 = clear this bit |

## 11.2.5 Event Select Register (EVTSEL)

**Table 93. Event Select Functions (CRn = 8, CRm = 1)**

| Function | CRn | CRm | Instruction |
|---|---|---|---|
| Event Select Register (EVTSEL) | 0b1000 | 0b0001 | MRC p14, 0, Rd, c8, c1, 0<br>MCR p14, 0, Rd, c8, c1, 0 |

EVTSEL is used to select events for PMN0, PMN1, PMN2 and PMN3. Refer to Table 97, "Performance Monitoring Events" on page 161 for a list of possible events. The event for a performance counter must be programmed while the PMU is disabled, otherwise the results are unpredictable.

**Table 94. Event Select Register**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| evtCount3 | | | | | | | | evtCount2 | | | | | | | | evtCount1 | | | | | | | | evtCount0 | | | | | | | |
| reset value: unpredictable | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

| Bits | Access | Description |
|---|---|---|
| 31:24 | Read / Write | Event Count 3 (evtCount3)<br>Identifies the source of events that PMN3 counts. See Table 97 for a description of the values this field contains. |
| 23:16 | Read / Write | Event Count 2 (evtCount2)<br>Identifies the source of events that PMN2 counts. See Table 97 for a description of the values this field contains. |
| 15:8 | Read / Write | Event Count 1 (evtCount1)<br>Identifies the source of events that PMN1 counts. See Table 97 for a description of the values this field contains. |
| 7:0 | Read / Write | Event Count 0 (evtCount0)<br>Identifies the source of events that PMN0 counts. See Table 97 for a description of the values this field contains. |

## 11.2.6 Performance Count Registers (PMN0 - PMN3)

**Table 95. Performance Count Functions (CRn = 0-3, CRm = 2)**

| Function | CRn | CRm | Instruction |
|---|---|---|---|
| Performance Count Register 0 (PMN0) | 0b0000 | 0b0010 | MRC p14, 0, Rd, c0, c2, 0<br>MCR p14, 0, Rd, c0, c2, 0 |
| Performance Count Register 1 (PMN1) | 0b0001 | 0b0010 | MRC p14, 0, Rd, c1, c2, 0<br>MCR p14, 0, Rd, c1, c2, 0 |
| Performance Count Register 2 (PMN2) | 0b0010 | 0b0010 | MRC p14, 0, Rd, c2, c2, 0<br>MCR p14, 0, Rd, c2, c2, 0 |
| Performance Count Register 3 (PMN3) | 0b0011 | 0b0010 | MRC p14, 0, Rd, c3, c2, 0<br>MCR p14, 0, Rd, c3, c2, 0 |

There are four 32-bit event counters; their format is shown in Table 96. The event counters are reset to '0' by writing a '1' to bit PMNC.P or is set to a predetermined value by directly writing to these. When an event counter reaches its maximum value 0xFFFFFFFF, the next event it needs to count causes it to roll over to zero and set its corresponding overflow flag (bit 1, 2, 3 or 4) in FLAG. An interrupt request is generated when its corresponding interrupt enable (bit 1, 2, 3 or 4) is set in INTEN.

**Table 96. Performance Monitor Count Register (PMN0 - PMN3)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Event Counter | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| reset value: unpredictable | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

| Bits | Access | Description |
|---|---|---|
| 31:0 | Read / Write | 32-bit event counter<br>Reset to '0' by PMNC register. When an event counter reaches its maximum value 0xFFFFFFFF, the next event it needs to count causes it to roll over to zero and generate an interrupt request when enabled. |

## 11.3 Managing the Performance Monitor

The following are a few notes about controlling the performance monitoring mechanism:

- An interrupt request is generated when a counter overflow flag is set and its associated interrupt enable bit is set in INTEN. The interrupt request remains asserted until software clears the overflow flag by writing a one to the flag that is set. (Note that the product specific interrupt unit and the CPSR must have enabled the interrupt in order for software to receive it.) The interrupt request is also de-asserted by clearing the corresponding interrupt enable bit. Disabling the facility (by setting PMNC.E to '0') doesn't de-assert the interrupt request. The count register must be cleared before enabling its corresponding interrupt.
- The counters continue to record events even after these overflow.
- To change an event for a performance counter, first disable the facility (by setting PMNC.M to "1" or PMNC.E to '0') and then modify EVTSEL. Not doing so causes unpredictable results.
- Resetting the performance counters while simultaneously disabling these (setting PMNC.P to '1' and either PMNC.E to '0' or PMNC.M to '1') causes unpredictable results. These must either be disabled and then separately reset or these reset at the time these are enabled.
- To increase the monitoring duration, software extends the count duration beyond 32 bits by counting the number of overflow interrupts each 32-bit counter generates. This is done in the interrupt service routine (ISR) where an increment to some memory location every time the interrupt occurs enables longer durations of performance monitoring. This does intrude slightly upon program execution but is negligible, since the ISR execution time is in the order of tens of cycles compared to the number of cycles it takes to generate an overflow interrupt ($2^{32}$).
- Power is saved by selecting event 0xFF for any unused event counter. This only applies when other event counters are in use. When the performance monitor is not used at all disable it by setting PMNC.E to '0'. The hardware then ensures minimal power consumption. The clock counter is used without the performance counters by setting PMNC.E to '1' and PMNC.M to '1'.

## 11.4 Performance Monitoring Events

Table 97 lists events that are monitored. Each of the Performance Monitor Count Registers (PMN0, PMN1, PMN2, and PMN3) counts any listed event. Software selects which event is counted by each count register by programming the corresponding event select field in EVTSEL. Other than the ASSP defined events (Events 0x80-0x87), the events in the table only count activity within the microarchitecture or activity which directly affects the microarchitecture (such as Event 0x17). The ASSP defined events are used to count activity outside of the microarchitecture as defined in the 3rd generation microarchitecture implementation options section of the relevant product documentation.

**Table 97. Performance Monitoring Events (Sheet 1 of 2)**

| Event Number (evtCount*n*) | Duration | Occurrence | Event Definition |
|---|---|---|---|
| 0x00 | | x | L1 Instruction cache miss requires fetch from external memory. |
| 0x01 | x | | L1 Instruction cache cannot deliver an instruction. This indicates an instruction cache or TLB miss. This event occurs every cycle in which the condition is present. |
| 0x02 | x | | Stall due to a microarchitecture register data dependency. This event occurs every cycle in which the condition is present. NOTE: this event does not count stalls due to co-processor register dependency. |
| 0x03 | | x | Instruction TLB miss. |
| 0x04 | | x | Data TLB miss. |
| 0x05 | | x | Branch instruction retired, branch has or does not have changed program flow. (Counts only B and BL instructions, in both ARM and Thumb mode) |
| 0x06 | | x | Branch mispredicted. (Counts only B and BL instructions, in both ARM and Thumb mode) |
| 0x07 | | x | Instruction retired. This event results in a count of the number of executed instructions. |
| 0x08 | x | | L1 Data cache buffer full stall. This event occurs every cycle in which the condition is present. |
| 0x09 | | x | L1 Data cache buffer full stall. This event occurs once for each contiguous sequence of this type of stall. |
| 0x0A | | x | L1 Data cache access, not including Cache Operations (defined in Section 7.2.8). All data accesses are treated as cacheable accesses and are counted here even when the cache is not enabled. |
| 0x0B | | x | L1 Data cache miss, not including Cache Operations (defined in Section 7.2.8). All data accesses are treated as cacheable accesses and are counted as misses when the data cache is not enabled. |
| 0x0C | | x | L1 Data cache write-back. This event occurs once for each line (32 bytes) that is written back from the cache. |
| 0x0D | | x | Software changed the PC (`b', `bx', `bl', `blx', `and', `eor', `sub', `rsb', `add', `adc', `sbc', `rsc', `orr', `mov', `bic', `mvn', `ldm Rn, {..., pc}', `ldr pc, [...]', pop {..., pc} is counted). The count does not increment when an exception occurs and the PC changes to the exception address (for example, IRQ, FIQ, SWI, etc...). |
| 0x0E | | x | Branch instruction retired. Branch has or does not have changed program flow. (Count ALL branch instructions, indirect as well as direct). |
| 0x0F | x | | Instruction issue cycle of retired instruction. This event is a count of the number of microarchitecture cycles each instruction requires to issue. |
| 0x17 | x | | Coprocessor stalled the instruction pipeline. This event occurs every cycle in which the condition is present. |
| 0x18 | | x | All changes to the PC. (includes software changes and exceptions) |
| 0x19 | | x | Pipeline flush due to branch mispredict or exception. |
| 0x1A | x | | The microarchitecture does not issue an instruction due to a backend stall. This event occurs every cycle in which the condition is present. |
| 0x1B | x | | Microarchitecture multiplier in use. This event occurs every cycle in which the multiplier is active. |
| 0x1C | x | | Microarchitecture multiplier stalled the instruction pipeline due to a resource stall. This event occurs every cycle in which the condition is present. |

**Table 97. Performance Monitoring Events (Sheet 2 of 2)**

| Event Number (evtCount*n*) | Duration | Occurrence | Event Definition |
|---|---|---|---|
| 0x1E | x | | Data Cache stalled the instruction pipeline. This event occurs every cycle in which the condition is present. |
| 0x20 | | x | Unified L2 cache request, not including cache operations (defined in Section 7.2.8). This event includes table walks, data and instruction requests. |
| 0x23 | | x | Unified L2 cache miss, not including Cache Operations (defined in Section 7.2.8). |
| 0x40 | | x | Address bus transaction. |
| 0x41 | | x | Self initiated (microarchitecture generated) address bus transaction. |
| 0x43 | x | | Bus clock. This event occurs once for each bus cycle. |
| 0x47 | x | | Self initiated (microarchitecture generated) data bus transaction. This event occurs once for each self initiated data bus cycle. |
| 0x48 | x | | Data bus transaction. This event occurs once for each data bus cycle. |
| 0x80 – 0x87 | ? | ? | ASSP Defined. See 3rd generation microarchitecture implementation options section of the relevant product documentation for more details. |
| 0xFF | - | - | Power savings event. This event deactivates the corresponding PMU event counter. |
| all others | - | - | Reserved, unpredictable results. |

Some typical combinations of counted events are listed in this section and summarized in Table 98. In this section, such an event combination is called a *mode*.

**Table 98. Some Common Uses of the PMU**

| Mode | EVTSEL.evtCount0 | EVTSEL.evtCount1 |
|---|---|---|
| Instruction Cache Efficiency | 0x07 (instruction count) | 0x00 (I-cache miss) |
| Data Cache Efficiency | 0x0A (D-cache access) | 0x0B (D-cache miss) |
| Instruction Fetch Latency | 0x01 (I-cache cannot deliver) | 0x00 (I-cache miss) |
| Data/Bus Request Buffer Full | 0x08 (D-buffer stall duration) | 0x09 (D-buffer stall) |
| Stall/Writeback Statistics | 0x02 (data stall) | 0x0C (D-cache writeback) |
| Instruction TLB Efficiency | 0x07 (instruction count) | 0x03 (I-TLB miss) |
| Data TLB Efficiency | 0x0A (D-cache access) | 0x04 (D-TLB miss) |
| Dynamic Block Length | 0x0D (software changed PC) | 0x07 (instruction count) |
| Table Walks | 0x03 (I-TLB miss) | 0x04 (D-TLB miss) |
| Microarchitecture Utilization | 0x0F (instruction issue cycles) | - |
| Exceptions | 0x18 (all changes to PC) | 0x0D (software changes to PC) |
| MAC Utilization | 0x1B (multiplier cycles) | 0x1C (multiplier stalled) |
| L2 Cache Efficiency | 0x20 (L2 cache access) | 0x23 (L2 cache miss) |
| Data Bus Utilization | 0x47 (initiated data bus cycles) | 0x48 (data bus cycles) |
| Address Bus Usage | 0x41 (self initiated address bus transactions) | 0x40 (address bus transactions) |

*Note:* PMN0 and PMN1 were used for illustration purposes only. Given there are four event counters, more elaborate combination of events is constructed. For example, one performance run selects 0xA, 0xB, 0xC, 0x9 events from which data cache performance statistics are gathered (like hit rates, number of writebacks per data cache miss, and number of times the data cache buffers fill up per request).

### 11.4.1 Instruction Cache Efficiency Mode

PMN0 totals the number of instructions that were executed (event 0x07), which does not include instructions fetched from the instruction cache that were never executed. This happens when a branch instruction changes the program flow; the instruction cache retrieves the next sequential instructions after the branch, before it receives the target address of the branch.

PMN1 counts the number of instruction fetch requests to external memory (event 0x00). Each of these requests loads 32 bytes at a time.

Statistics derived from these two events:

- Instruction cache miss-rate. This is derived by dividing PMN1 by PMN0.
- *The average number of cycles it took to execute an instruction or commonly referred to as cycles-per-instruction (CPI).* CPI is derived by dividing CCNT by PMN0, where CCNT was used to measure total execution time.

### 11.4.2 Data Cache Efficiency Mode

PMN0 totals the number of data cache accesses (event 0x0A), which includes cacheable and non-cacheable accesses and accesses made to locations configured as data RAM.

*Note:* **STM** and **LDM** each count as multiple accesses to the data cache depending on the number of registers specified in the register list. **LDRD** counts as two accesses.

PMN1 counts the number of data cache misses (event 0x0B). Cache operations do not contribute to this count. See Section 7.2.8 for a description of these operations.

The statistic derived from these two events is:

- Data cache miss-rate. This is derived by dividing PMN1 by PMN0.
- Data cache hit-rate. This is derived by subtracting PMN1 from PMN0 and dividing this result by PMN0.

### 11.4.3 Instruction Fetch Latency Mode

PMN0 accumulates the number of cycles when the instruction-cache is not able to deliver an instruction to 3rd generation microarchitecture due to an instruction-cache miss or instruction-TLB miss (event 0x01). This event means that the processor microarchitecture is stalled.

PMN1 counts the number of instruction fetch requests to external memory (event 0x00). Each of these requests loads 32 bytes at a time. This is the same event as measured in instruction cache efficiency mode.

Statistics derived from these two events:

- *The average number of cycles the processor stalled waiting for an instruction fetch from external memory to return.* This is calculated by dividing PMN0 by PMN1. When the average is high then 3rd generation microarchitecture is starved of the bus external to 3rd generation microarchitecture.
- *The percentage of total execution cycles the processor stalled waiting on an instruction fetch from external memory to return.* This is calculated by dividing PMN0 by CCNT, which was used to measure total execution time.

### 11.4.4 Data/Bus Request Buffer Full Mode

The Data Cache has buffers available to service cache misses or uncacheable accesses. For every memory request the Data Cache receives from the processor a buffer is speculatively allocated in case an external memory request is required or temporary storage is needed for an unaligned access. When no buffers are available, the Data Cache stalls the processor microarchitecture. How often the Data Cache stalls depends on the performance of the bus external to 3rd generation microarchitecture and what the memory access latency is for Data Cache miss requests to external memory. When 3rd generation microarchitecture memory access latency is high, possibly due to starvation, these Data Cache buffers becomes full. This performance monitoring mode is provided to see when 3rd generation microarchitecture is being starved off the bus external to 3rd generation microarchitecture, which affects the performance of the application running on the microarchitecture.

PMN0 accumulates the number of clock cycles the processor is being stalled due to this condition (event 0x08) and PMN1 monitors the number of times this condition occurs (event 0x09).

Statistics derived from these two events:

- *The average number of cycles the processor stalled on a data-cache access that overflows the data-cache buffers*. This is calculated by dividing PMN0 by PMN1. This statistic shows when the duration event cycles are due to many requests or are attributed to just a few requests. When the average is high then 3rd generation microarchitecture is starved of the bus external to 3rd generation microarchitecture.
- *The percentage of total execution cycles the processor stalled because a Data Cache request buffer was not available*. This is calculated by dividing PMN0 by CCNT, which was used to measure total execution time.

### 11.4.5 Stall/Writeback Statistics

When an instruction requires the result of a previous instruction and that result is not yet available, 3rd generation microarchitecture stalls in order to preserve the correct data dependencies. PMN0 counts the number of stall cycles due to data-dependencies (event 0x02). Not all data-dependencies cause a stall; only the following dependencies cause such a stall penalty:

- Load-use penalty: attempting to use the result of a load before the load completes. To avoid the penalty, software must delay using the result of a load until the load data is available. This penalty shows the latency effect of data-cache access.
- Multiply/Accumulate-use penalty: attempting to use the result of a multiply or multiply-accumulate operation before the operation completes. Again, to avoid the penalty, software must delay using the result until the load data is available.
- ALU use penalty: there are a few isolated cases where back to back ALU operations results in one cycle delay in the execution. These cases are defined in ***Chapter 13.0, "Performance Considerations".***

PMN1 counts the number of writeback operations emitted by the data cache (event 0x0C). These writebacks occur when the data cache evicts a dirty line of data to make room for a newly requested line or as the result of clean operation (P15, register 7).

Statistics derived from these two events:

- *The percentage of total execution cycles the processor stalled because of a data dependency*. This is calculated by dividing PMN0 by CCNT, which was used to measure total execution time. Often a compiler reschedules code to avoid these penalties when given the right optimization switches.
- Total number of data writeback requests to external memory are derived solely with PMN1.

### 11.4.6 Instruction TLB Efficiency Mode

PMN0 totals the number of instructions that were executed (event 0x07), which does not include instructions that were translated by the instruction TLB and never executed. This happens when a branch instruction changes the program flow; the instruction TLB translates the next sequential instructions after the branch, before it receives the target address of the branch.

PMN1 counts the number of instruction TLB table-walks (event 0x03), which occurs when there is a TLB miss. When the instruction TLB is disabled PMN1 does not increment.

The statistic derived from these two events:

- Instruction TLB miss-rate. This is derived by dividing PMN1 by PMN0.

### 11.4.7 Data TLB Efficiency Mode

PMN0 totals the number of data cache accesses (event 0x0A), which includes cacheable and non-cacheable accesses and accesses made to locations configured as data RAM.

*Note:* **STM** and **LDM** each count as several accesses to the data TLB depending on the number of registers specified in the register list. **LDRD** registers two accesses.

PMN1 counts the number of data TLB table-walks (event 0x04), which occurs when there is a TLB miss. When the data TLB is disabled PMN1 does not increment.

The statistic derived from these two events is:

- Data TLB miss-rate. This is derived by dividing PMN1 by PMN0.

### 11.4.8 Average Dynamic Block Length Mode

PMN0 totals the number of changes to the PC which indicates a program flow change (event 0x18). PMN1 totals the number of instructions executed (event 0x07).

The statistic derived from these two events is:

- Average Dynamic Block Length. This is derived by dividing PMN1 by PMN0.

### 11.4.9 Table Walk Mode

PMN0 counts the number of instruction TLB table-walks, which occurs when there is an instruction TLB miss (event 0x03). PMN1 counts the number of data TLB table-walks, which occurs when there is a data TLB miss (event 0x04).

The statistic derived from these two events is:

- Table Walks. This is derived by adding PMN0 to PMN1.

### 11.4.10 Microarchitecture Utilization Mode

The Microarchitecture Utilization Mode is used to determine software efficiency. PMN0 totals the number of instruction issue cycles (event 0x0F). CCNT totals the number of microarchitecture cycles. This statistic is used to determine how efficiently code is using the microarchitecture. This does not indicate code performance.

The statistic derived from these two events is:

- Microarchitecture Utilization. This is derived by dividing PMN0 by CCNT.

### 11.4.11 Exception Mode

PMN0 totals all changes to the program counter (event 0x18). PMN1 totals software changes to the program counter (event 0x0D).

The statistic derived from these two events is:

- Exceptions. This is derived by subtracting PMN1 from PMN0.

### 11.4.12 MAC Utilization Mode

PMN0 totals the number of active multiplier cycles (event 0x1B). PMN1 totals the number of stalled multiplier cycles (event 0x1C).

The statistic derived from these two events is:

- MAC Utilization. This is derived by subtracting PMN1 from PMN0 and dividing this result by PMN0.

### 11.4.13 L2 Cache Efficiency Mode

PMN0 totals the number of L2 cache accesses, which includes cacheable and non-cacheable accesses and accesses made to locations configured as data RAM (event 0x20).

*Note:* **STM** and **LDM** each count as several accesses to the data cache depending on the number of registers specified in the register list. **LDRD** registers two accesses.

PMN1 counts the number of L2 cache misses (event 0x23). Cache operations do not contribute to this count. See Section 7.2.8 for a description of these operations.

The statistic derived from these two events is:

- Data cache miss-rate. This is derived by dividing PMN1 by PMN0.
- Data cache hit-rate. This is derived by subtracting PMN1 from PMN0 and dividing this result by PMN0.

### 11.4.14 Data Bus Utilization Mode

PMN0 counts the number of self initiated data bus cycles (event 0x47). PMN1 counts the number of total data bus cycles (event 0x48).

The statistic derived from these two events is:

- Data Bus Utilization. This is derived by dividing PMN0 by PMN1.

### 11.4.15 Address Bus Usage Mode

PMN0 counts the number of self initiated address bus transactions (event 0x41). PMN1 counts the total number of address bus transactions (event 0x40).

The statistic derived from these two events is:

- Address Bus Usage. This is derived by dividing PMN0 by PMN1.

## 11.5 Multiple Performance Monitoring Run Statistics

There are times when more than four events need to be monitored for performance tuning. In this case, multiple performance monitoring runs are done, capturing different events from each run. For example, the first run monitors the events associated with instruction cache performance and the second run monitors the events associated with data cache performance. By combining the results, statistics like total number of memory requests are derived.

## 11.6 Examples

In this example, the events selected with the Instruction Cache Efficiency mode are monitored and CCNT is used to measure total execution time. Sampling time ends when PMN0 overflows which generates an IRQ interrupt.

**Example 2. Configuring the Performance Monitor**

```
; Configure the performance monitor with the following values:
;   EVTSEL.evtCount0 = 7, EVTSEL.evtCount1 = 0 instruction cache efficiency
;   INTEN.inten = 0x7 set all counters to trigger an interrupt on overflow
;   PMNC.C = 1   reset CCNT register
;   PMNC.P = 1   reset PMN0 and PMN1 registers
;   PMNC.E = 1   enable counting
    MOV  R0,#0x0007
    MCR  P14,0,R0,C8,c1,0  ; setup EVTSEL
    MOV  R0,#0x7
    MCR  P14,0,R0,C4,c1,0  ; setup INTEN
    MCR  P14,0,R0,C0,c1,0  ; setup PMNC, reset counters & enable
; Counting begins
```

Counter overflow is dealt with in the IRQ interrupt service routine as shown below:

**Example 3. Interrupt Handling**

```
IRQ_INTERRUPT_SERVICE_ROUTINE:
; Assume that performance counting interrupts are the only IRQ in the system
    MRC  P14,0,R1,C0,c1,0  ; read the PMNC register
    BIC  R2,R1,#1          ; clear the enable bit, preserve other bits in PMNC
    MCR  P14,0,R2,C0,c1,0  ; disable counting
    MRC  P14,0,R3,C1,c1,0  ; read CCNT register
    MRC  P14,0,R4,C0,c2,0  ; read PMN0 register
    MRC  P14,0,R5,C1,c2,0  ; read PMN1 register
; <process the results here>
    MRC p14, 0, R2, C5, C1, 0 ; Clear interrupt source by read/write of...
    MCR p14, 0, R2, C5, C1, 0 ; ...FLAG register
    SUBS PC,R14,#4         ; return from interrupt
```

As an example, assume the following values in CCNT, PMN0, PMN1 and PMNC:

**Example 4. Computing the Results**

```
; Assume CCNT overflowed

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
```

In the contrived example above, the instruction cache had a miss-rate of 5% and CPI was 2.4.

# 12.0 Software Debug

This chapter describes the software debug and related features implemented in the 3rd generation Intel XScale® microarchitecture (3rd generation microarchitecture or 3rd generation), namely:

- debug modes, registers, exceptions, breakpoint resources.
- a serial debug communication link via the JTAG interface.
- an on-microarchitecture trace buffer.
- on-microarchitecture Debug SRAM and a mechanism to load it via JTAG.

## 12.1 Additional Debug Documentation

In addition to the software debug features described in this chapter, additional documentation is available for debugger developers.

- *3rd Generation Intel XScale® Microarchitecture Software Debug Guide*
  This document describes additional software debug capabilities available on 3rd generation microarchitecture. It also provides information on developing a 3rd generation microarchitecture debug handler and porting handlers from a previous microarchitecture.

## 12.2 Definitions

**Table 99. Debug Terminology**

| Term | Meaning |
|---|---|
| debug handler | The debug handler is the routine that runs on 3rd generation microarchitecture when a debug exception occurs. |
| debugger | The debugger is software that runs on a host system outside of 3rd generation microarchitecture. |
| hot-debug | Hot-debug refers to connecting a debugger and starting a debug session on a target system, while an application is already running on the target. |

## 12.3 Microarchitecture Debug Capabilities

The 3rd generation microarchitecture debug capabilities, when used with a debugger application, allow software running on a 3rd generation microarchitecture to be debugged. The 3rd generation microarchitecture breakpoint resources allow a debugger to stop program execution and re-direct execution to a debug handling routine. Once program execution has stopped, the debugger examines or modify processor state, co-processor state, or memory. The debugger then restarts execution of the application.

3rd generation microarchitecture runs in one of two debug modes:

- Halt Mode

  Halt Mode is a JTAG debug mode which uses an on-microarchitecture Debug SRAM, separate from the application memory space, to hold a debug handler routine. A debugger loads the debug handler into the Debug SRAM through JTAG prior to starting a debug session. Having the debug handler reside in the on-microarchitecture RAM allows initial debug in a non-functional system, since functional external memory is not required.

  During Halt Mode, all debug exceptions vector to the debug handler, at address 0 of the Debug SRAM. The processor switches into DEBUG mode (CPSR[4:0] = 0x15) and enters Special Debug State. Once in the debug handler, a debugger communicates with the handler through JTAG, and send commands to examine or modify processor or system state.

- Monitor Mode

  Monitor Mode is a software debug mode used for debugging software such as interrupt handlers and other system-level routines, as well as systems that have real-time requirements. In this mode, debug exceptions are handled as prefetch aborts or data aborts, depending on the cause of the exception.

  When a debug exception occurs in Monitor Mode, the processor switches to abort mode and branches to a debug monitor loaded in system memory. The monitor then enables interrupts to allow real-time handling of system events.

  NOTE: System-on-a-chip (SOC) debug exceptions in Monitor Mode are handled differently than other Monitor Mode debug exceptions: the processor enters DEBUG mode and Special Debug State, which is similar to debug exceptions in Halt Mode. However, in Monitor Mode the processor branches to address 0 (or 0xffff0000, when vector table is relocated) in the application space, instead of address 0 of the Debug SRAM.

### 12.3.1 Debug Registers

The debug registers reside in CP14 and CP15. CP15 contains the HW breakpoint resources. CP14 contains the global debug control and status register, the JTAG communications registers and trace buffer registers. Table 100 and Table 101 show the software debug registers.

CP15 registers are only accessible via software, using MRC and MCR. CRn and CRm specify the register to access. The opcode_1 and opcode_2 fields are not used and need to be set to 0. Direct access to these registers through JTAG is not supported.

**Table 100. CP15 Software Debug Registers**

| CRn | CRm | Access | Register | Cross-Reference |
|---|---|---|---|---|
| 14 | 8 | Read / Write | Instruction Breakpoint Register 0 (IBR0) | See Section 12.3.7, "HW Breakpoint Resources" on page 184 |
| 14 | 9 | Read / Write | Instruction Breakpoint Register 1 (IBR1) | |
| 14 | 0 | Read / Write | Data Breakpoint Register 0 (DBR0) | |
| 14 | 3 | Read / Write | Data Breakpoint Register 1 (DBR1) | |
| 14 | 4 | Read / Write | Data Breakpoint Control Register (DBCON) | |

CP14 registers are accessible using MRC and STC (for software readable registers) and MCR and LDC (for software writable registers). CDP to any CP14 registers causes an undefined instruction exception. CRn and CRm specify the register to access. The opcode_1 and opcode_2 fields are not used and need to be set to 0.

Within CP14, the TX and RX registers, certain bits in the TXRXCTRL register, and certain bits in the DCSR is also accessed by a debugger directly through the JTAG interface. Refer to the description of these registers for complete details.

**Table 101. CP14 Software Debug Registers**

| CRn | CRm | Access[a] | Register | Cross-Reference |
|---|---|---|---|---|
| 8 | 0 | SW Read / Write<br>JTAG Read-Only | Transmit Register (TX) | See Section 12.4.2, "Transmit Register (TX)" on page 193 |
| 9 | 0 | SW Read-Only<br>JTAG Write-Only | Receive Register (RX) | See Section 12.4.3, "Receive Register (RX)" on page 193 |
| 10 | 0 | Varies[b] | Debug Control and Status Register (DCSR) | See Section 12.3.2, "Debug Control and Status Register (DCSR)" on page 175 |
| 11 | 0 | Read-Only | Trace Buffer Register (TBREG) | See Section 12.6.2.2, "Trace Buffer Register (TBREG)" on page 202 |
| 12 | 0 | Read / Write | Checkpoint 0 Register (CHKPT0) | See Section 12.6.2.1, "Checkpoint Registers" on page 200 |
| 13 | 0 | Read / Write | Checkpoint 1 Register (CHKPT1) | |
| 14 | 0 | Varies[b] | TXRX Control Register (TXRXCTRL) | See Section 12.4.1, "Transmit/ Receive Control Register (TXRXCTRL)" on page 189 |

a. Unless otherwise stated, access refers to software access and direct JTAG access is not supported.
b. JTAG and software access to these registers varies depending on the bit, refer to the register description for further details.

Software access to all debug registers must be done from a privileged mode. User mode access generates an undefined instruction exception. Specifying registers which do not exist has unpredictable results.

## 12.3.2 Debug Control and Status Register (DCSR)

**Table 102. Debug Control and Status Register (CRn = 10, CRm = 0)**

| Function | CRn | CRm | Instruction |
|---|---|---|---|
| Debug Control and Status Register (DCSR) | 0b1010 | 0b0000 | MRC p14, 0, Rd, c10, c0, 0<br>MCR p14, 0, Rd, c10, c0, 0 |

The DCSR register is the main debug control register. Table 103 shows the format of the register. The register is accessed in privileged modes by software running on the microarchitecture or by a debugger through the JTAG interface. Refer to Section 12.5.1, "SELDCSR JTAG Register" on page 194 for details about accessing the DCSR through JTAG.

**Table 103. Debug Control and Status Register (DCSR) (Sheet 1 of 2)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| GE | H | B | | | | | | TF | TI | | TD | TA | TS | TU | TR | | | | | | | | | | TT | SA | MOE | | | M | E |

reset value: unpredictable

| Bits | Access | Description | Reset Value | TRST Value |
|---|---|---|---|---|
| 31 | SW Read / Write<br>JTAG Read-Only / Write-Ignored | Global Enable (GE)<br>0 = disables all debug functionality<br>1 = enables all debug functionality | 0 | unchanged |
| 30 | SW Read-Only / Write-Ignored<br>JTAG Read / Write | Halt Mode (H)<br>0 = Monitor Mode<br>1 = Halt Mode | unchanged | 0 |
| 29 | SW Read-Only / Write-Ignored<br>JTAG Read-Only / Write-Ignored | SOC Break (B)<br>value of SOC break input pin | unpredictable | unpredictable |
| 28:24 | Read-Unpredictable / Write-As-Zero | Reserved | unpredictable | unpredictable |
| 23 | SW Read-Only / Write-Ignored<br>JTAG Read / Write | Trap FIQ (TF) | unchanged | 0 |
| 22 | SW Read-Only / Write-Ignored<br>JTAG Read / Write | Trap IRQ (TI) | unchanged | 0 |
| 21 | Read-Unpredictable / Write-As-Zero | Reserved | unpredictable | unpredictable |
| 20 | SW Read-Only / Write-Ignored<br>JTAG Read / Write | Trap Data Abort (TD) | unchanged | 0 |
| 19 | SW Read-Only / Write-Ignored<br>JTAG Read / Write | Trap Prefetch Abort (TA) | unchanged | 0 |
| 18 | SW Read-Only / Write-Ignored<br>JTAG Read / Write | Trap Software Interrupt (TS) | unchanged | 0 |
| 17 | SW Read-Only / Write-Ignored<br>JTAG Read / Write | Trap Undefined Instruction (TU) | unchanged | 0 |
| 16 | SW Read-Only / Write-Ignored<br>JTAG Read / Write | Trap Reset (TR) | unchanged | 0 |
| 15:7 | Read-Unpredictable / Write-As-Zero | Reserved | unpredictable | unpredictable |
| 6 | SW Read / Write<br>JTAG Read-Only / Write-Ignored | Thumb Trace (TT)<br>0 = Disable Thumb Trace<br>1 = Enable Thumb Trace | 0 | unchanged |
| 5 | SW Read / Write<br>JTAG Read-Only / Write-Ignored | Sticky Abort (SA) | 0 | unchanged |

## Table 103. Debug Control and Status Register (DCSR) (Sheet 2 of 2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| GE | H | B | | | | | | TF | TI | | TD | TA | TS | TU | TR | | | | | | | | | | TT | SA | MOE | | | M | E |

reset value: unpredictable

| Bits | Access | Description | Reset Value | TRST Value |
|---|---|---|---|---|
| 4:2 | SW Read / Write<br>JTAG Read-Only / Write-Ignored | Method Of Entry (MOE)<br>000: Processor Reset<br>001: Instruction Breakpoint Hit<br>010: Data Breakpoint Hit<br>011: BKPT Instruction Executed<br>100: JTAG Debug Break OR SOC Debug Break Occurred<br>101: Vector Trap Occurred<br>110: Trace-Buffer-Full Break Occurred<br>111: Reserved | 0b000 | unchanged |
| 1 | SW Read / Write<br>JTAG Read-Only / Write-Ignored | Trace Buffer Mode (M)<br>0 = Wrap around mode<br>1 = fill-once mode | 0 | unchanged |
| 0 | SW Read / Write<br>JTAG Read-Only / Write-Ignored | Trace Buffer Enable (E)<br>0 = Disabled<br>1 = Enabled | 0 | unchanged |

#### 12.3.2.1 Global Enable Bit (GE)

The Global Debug Enable bit disables and enables all debug functionality, except reset vector trap and JTAG debug breaks. Following a processor reset, this bit is clear so all debug functionality is disabled. When debug functionality is disabled, the BKPT instruction becomes a NOP and hardware breakpoints, and non-reset vector traps are ignored.

Reset vector traps and JTAG debug breaks are not qualified by the Global Debug Enable bit. The reset vector trap allows a debugger to gain control of the system following a processor reset. The JTAG debug break allows the debugger to stop the microarchitecture to initiate a hot-debug session.

#### 12.3.2.2 Halt Mode Bit (H)

The Halt Mode bit configures the debug unit for either Halt Mode or Monitor Mode.

#### 12.3.2.3 System-on-a-Chip (SOC) Break Flag (B)

Reading the SOC Break flag returns the value of the SOC break input to the microarchitecture[1].

#### 12.3.2.4 Vector Trap Bits (TF,TI,TD,TA,TS,TU,TR)

The Vector Trap bits allow the debugger to set breakpoints on exception vectors without using the HW breakpoint resources. When a bit is set, the processor acts like when an instruction breakpoint was set up on the corresponding exception vector. A debug exception is generated before the instruction in the exception vector executes.

The Vector Trap bits are only set by a debugger through the JTAG interface. A non-reset vector trap exception only occurs when the processor is configured for Halt Mode and the Global Debug Enable bit is set.

A reset vector trap is not qualified by global debug enable. However, the processor must be in Halt Mode. The reset vector trap and Halt Mode bits are set up before or during a processor reset. When processor reset is de-asserted, a debug exception occurs before the instruction in the reset vector executes.

#### 12.3.2.5 Thumb Trace Bit (TT)

The Thumb Trace Bit, when set, enables the trace buffer to provide Thumb/ARM information as part of branch target addresses in the Trace Buffer and Checkpoint registers. Refer to Section 12.6.4, "Tracing Thumb Code" on page 206 for more details on using this bit to trace Thumb code.

The reset value of the Thumb Trace Bit is '0', disabling this feature.

To use this feature software must set this bit before (or at the same time as) enabling the Trace Buffer. Once this bit is set, software must not change it while tracing is enabled, otherwise the Trace Buffer contents are unpredictable.

1. Use of the SOC break input (for system-on-a-chip debug) is ASSP specific. Refer to the 3rd generation microarchitecture implementation options section of the relevant product documentation to determine whether this feature is used.

#### 12.3.2.6 Sticky Abort Bit (SA)

The Sticky Abort bit is only valid in Halt Mode. It indicates a data abort occurred during Special Debug State (see Section 12.3.4, "Halt Mode" on page 180). The microarchitecture does not generate exceptions during SDS, thus, for data aborts, it sets the Sticky Abort bit to indicate a data abort was detected.

The processor also sets up the Fault Status Register (FSR) and Fault Address Register (FAR) since it normally is for data aborts. The debugger uses the Sticky Abort bit and the fault information to determine when a data abort was detected during the Special Debug State and take appropriate actions.

The Sticky Abort bit must be cleared by SW before exiting the debug handler.

#### 12.3.2.7 Method of Entry Bits (MOE)

The Method of Entry field specifies the cause of the most recent debug exception. When multiple exceptions occur in parallel, the processor places the highest priority exception (based on the priorities in Table 104) in the MOE field.

#### 12.3.2.8 Trace Buffer Mode Bit (M)

The Trace Buffer Mode bit selects one of two trace buffer modes:

- Wrap-around mode - Trace buffer fills up and wraps around until a debug exception occurs.
- Fill-once mode - Trace buffer fills up and generates a trace-buffer-full break.

The Trace Buffer Mode bit must not be modified while tracing is enabled, otherwise the contents of the trace buffer are unpredictable.

#### 12.3.2.9 Trace Buffer Enable Bit (E)

The Trace Buffer Enable bit enables and disables the trace buffer. Both DCSR.e and DCSR.ge must be set to enable the trace buffer. The processor automatically clears this bit, disabling the trace buffer, when any debug exception occurs. For more details on the trace buffer refer to Section 12.6, "Trace Buffer" on page 199.

### 12.3.3 Debug Exceptions

A debug exception causes the processor to re-direct execution to a debug event handling routine. The 3rd generation microarchitecture debug architecture defines the following debug exceptions:

- instruction breakpoint
- data breakpoint
- software breakpoint
- JTAG debug break
- exception vector trap
- trace-buffer-full break
- SOC debug break

When a debug exception occurs, the processor actions depend on whether the debug unit is configured for Halt Mode or Monitor Mode.

Table 104 shows the priority of debug exceptions relative to other processor exceptions.

**Table 104. Event Priority**

| Event | Priority |
|---|---|
| Vector Trap | 1 (highest) |
| Reset | 2 |
| data abort (precise) | 3 |
| data breakpoint | 4 |
| data abort (imprecise) | 5 |
| JTAG debug break; SOC debug break; trace-buffer-full break | 6 |
| FIQ | 7 |
| IRQ | 8 |
| instruction breakpoint | 9 |
| pre-fetch abort | 10 |
| undef, SWI, software breakpoint | 11 |

### 12.3.4 Halt Mode

For Halt Mode debugging, the debugger must load a debug handler into the Debug SRAM starting at address 0, prior to beginning a debug session. For more details on using the Debug SRAM refer to Section 12.7, "Debug SRAM" on page 209. When a debug exception occurs in Halt Mode, the processor executes the debug handler out of Debug SRAM, allowing the debugger to examine or modify state in the target system.

During Halt Mode, writes to the HW breakpoint resources and the DCSR are ignored, unless the processor is in Special Debug State (SDS). For more details on SDS, refer to the SDS description below.

When a debug exception occurs during Halt Mode, or an SOC debug break occurs in Monitor Mode, the processor takes the following actions:

- disables the trace buffer
- sets DCSR.MOE encoding
- enters Special Debug State (SDS)
- R14_dbg is updated as follows:

**Table 105. R14_dbg Updating - Halt Mode**

| Debug Exception | R14_dbg Value | |
|---|---|---|
| | ARM Mode | THUMB Mode |
| Data Breakpoint | PC of next instruction to execute + 4 | PC of next instruction to execute + 4 |
| Instruction Breakpoint, SW Breakpoint | PC of breakpointed instruction + 4 | PC of breakpointed instruction + 4 |
| Vector Trap | PC of trapped exception vector + 4 | NA |
| Trace-buffer-full Break, SOC debug Break, JTAG Debug Break | PC of next instruction to execute + 4 | PC of next instruction to execute + 4 |

- SPSR_dbg = CPSR
- CPSR[4:0] = 0b10101 (DEBUG mode)
- CPSR[5] = 0
- CPSR[6] = 1
- CPSR[7] = 1
- PC is determined as follows:
  For all debug exceptions in Halt Mode: PC = 0 of Debug SRAM;
  For SOC debug break from Monitor Mode: PC = VA 0 in application space (or 0xffff0000, when exception vector table is relocated).

The FSR.D bit, which is set for all debug exceptions during Monitor Mode (including the SOC debug break) to indicate that a debug exception occurred, is unaffected by debug exceptions during Halt Mode.

Entering SDS following a Halt Mode debug exception has the following effect:

- The processor ignores all exceptions. SWI and undefined instructions have unpredictable results. The processor ignores pre-fetch aborts, FIQ and IRQ (SDS disables FIQ and IRQ regardless of the enable values in the CPSR) and all debug exceptions. The processor reports data aborts detected during SDS by setting the Sticky Abort bit in the DCSR, but does not generate an exception.
- The hardware breakpoint resources and DCSR are software writable.

SDS remains in effect regardless of the processor mode. This allows the debug handler to switch to other modes, maintaining SDS functionality. However, entering User mode causes unpredictable behavior.

The processor exits SDS following a CPSR restore operation. When exiting, the debug handler uses:

```
subs pc, lr, #4
```

This restores the CPSR, turns off all of SDS functionality, and branches to the target instruction.

### 12.3.5 Monitor Mode

In Monitor Mode, the processor handles debug exceptions like normal ARM exceptions (except for SOC debug breaks; refer to the Section 12.3.4, "Halt Mode" on page 180 to see how Monitor Mode SOC debug breaks are handled). The processor generates a data abort or a pre-fetch abort depending on the type of debug exception.

The following debug exceptions cause data aborts:

- data breakpoint
- JTAG debug break
- trace-buffer full break

The following debug exceptions cause prefetch aborts:

- instruction breakpoint
- BKPT instruction

The processor ignores vector traps during Monitor Mode.

When a debug exception occurs in Monitor Mode, the processor takes the following actions:

- disables the trace buffer
- sets DCSR.MOE encoding
- sets Fault Status Register (FSR) bit 9 (see Chapter 7.0, "Register 5: Fault Status Register")
- R14_abt is updated as follows:

**Table 106. R14_abt Updating - Monitor Mode**

| Debug Exception | R14_abt Value | |
|---|---|---|
| | ARM Mode | THUMB Mode |
| Data Breakpoint | PC of next instruction to execute + 4 | PC of next instruction to execute + 4 |
| Instruction Breakpoint, SW Breakpoint | PC of breakpointed instruction + 4 | PC of breakpointed instruction + 4 |
| Trace-buffer-full Break, JTAG Debug Break | PC of next instruction to execute + 4 | PC of next instruction to execute + 4 |

- SPSR_abt = CPSR
- CPSR[4:0] = 0b10111 (ABORT mode)
- CPSR[5] = 0
- CPSR[6] = unchanged
- CPSR[7] = 1
- PC = 0xc or 0xffff000c (for Prefetch Aborts) OR
  PC = 0x10 or 0xffff0010 (for Data Aborts)

During abort mode, the processor pends JTAG debug breaks and trace buffer full breaks. When the processor exits abort mode, either through a CPSR restore or a write directly to the CPSR, the pended debug breaks immediately generates a debug exception. Any pending debug breaks are cleared out when any type of debug exception occurs. Note that SOC debug breaks are not pended in abort mode; these occur immediately when detected.

To return to the application after handling the debug exception the handler uses:

```
subs pc, lr, #4
```

### 12.3.6 Summary of Debug Modes

Table 107 summarizes special 3rd generation microarchitecture behavior for Halt and Monitor Modes.

**Table 107. Special Behavior for Halt and Monitor Mode**

| Feature | Monitor Mode | | Halt Mode | |
|---|---|---|---|---|
| | non-SDS | SDS[a] | non-SDS | SDS |
| Debug SRAM used | NO | NO | NO | YES |
| instruction address translation disabled | NO | NO | NO | YES |
| data address translation disabled | NO | NO | NO | NO |
| instruction protection checking disabled | NO | NO | NO | YES |
| data protection checking disabled | NO | NO | NO | NO |
| BTB disabled | NO | NO | NO | YES |
| PID disabled on instruction accesses | NO | NO | NO | YES |
| PID disabled on data accesses | NO | NO | NO | NO |
| all exceptions ignored | NO | YES | NO | YES |
| access to debug registers allowed (DCSR, IBR[0,1], DBR[0,1], DBCON) | YES | YES | NO | YES |

a. In Monitor Mode, SDS is only entered when an SOC debug break occurs. All other debug exceptions in Monitor Mode are either prefetch or data aborts.

### 12.3.7 HW Breakpoint Resources

On 3rd generation microarchitecture, two instruction and two data breakpoint registers, denoted IBR0/IBR1 and DBR0/DBR1, are available. The data breakpoint address registers also have a separate control register, DBCON.

The instruction and data breakpoint registers are 32-bit registers. The instruction breakpoint causes a break before execution of the target instruction. The data breakpoint causes a break after the memory access has been issued.

In this section the term Modified Virtual Address (MVA) is used to refer to the virtual address modified with the PID. Refer to Section 7.2.13, "Register 13: Process ID" on page 105 for more details on the PID. The processor does not OR the PID with the specified breakpoint address prior to doing address comparison. The programmer must write the MVA to the breakpoint address register. This applies for instruction and data breakpoints.

#### 12.3.7.1 Instruction Breakpoints

**Table 108. Instruction Breakpoint Resources (CRn = 14, CRm = 8,9)**

| Function | CRn | CRm | Instruction |
|---|---|---|---|
| Instruction Breakpoint Register 0 (IBR0) | 0b1110 | 0b1000 | MRC p15, 0, Rd, c14, c8, 0<br>MCR p15, 0, Rd, c14, c8, 0 |
| Instruction Breakpoint Register 1 (IBR1) | 0b1110 | 0b1001 | MRC p15, 0, Rd, c14, c9, 0<br>MCR p15, 0, Rd, c14, c9, 0 |

3rd generation microarchitecture defines two instruction breakpoint registers (IBR0, IBR1). The format of these registers is shown in Table 109. In ARM mode, the upper 30 bits contain a word aligned MVA to break on. In Thumb mode, the upper 31 bits contain a half-word aligned MVA to break on. In both modes, bit 0 enables and disables that instruction breakpoint register.

**Table 109. Instruction Breakpoint Register (IBRx)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Address | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | E |

reset value: address unpredictable, disabled

| Bits | Access | Description |
|---|---|---|
| 31:1 | Read / Write | Address<br>Instruction Breakpoint MVA |
| 0 | Read / Write | IBRx Enable (E)<br>0 = Breakpoint disabled<br>1 = Breakpoint enabled |

Enabling instruction breakpoints while debug is globally disabled results in unpredictable behavior.

When an address match occurs, the processor generates a debug exception before the instruction at the address specified in the matching IBRx executes.

Software must disable the breakpoint before exiting the handler. This allows the breakpointed instruction to execute after the exception is handled.

Single step execution is accomplished using the instruction breakpoint registers and is handled in software.

### 12.3.7.2 Data Breakpoints

**Table 110. Data Breakpoint Resources (CRn = 14, CRm = 0,3,4)**

| Function | CRn | CRm | Instruction |
|---|---|---|---|
| Data Breakpoint Register 0 (DBR0) | 0b1110 | 0b0000 | MRC p15, 0, Rd, c14, c0, 0<br>MCR p15, 0, Rd, c14, c0, 0 |
| Data Breakpoint Register 1 (DBR1) | 0b1110 | 0b0011 | MRC p15, 0, Rd, c14, c3, 0<br>MCR p15, 0, Rd, c14, c3, 0 |
| Data Breakpoint Control Register (DBCON) | 0b1110 | 0b0100 | MRC p15, 0, Rd, c14, c4, 0<br>MCR p15, 0, Rd, c14, c4, 0 |

3rd generation microarchitecture provides two data breakpoint registers (DBR0, DBR1). The format of the registers is shown in Table 111.

**Table 111. Data Breakpoint Register (DBRx)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Address/Mask | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| reset value: unpredictable | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

| Bits | Access | Description |
|---|---|---|
| 31:0 | Read / Write | Address/Mask<br>DBR0: Data Breakpoint MVA<br>DBR1: Address Mask or Data Breakpoint MVA |

DBR0 is a dedicated data address breakpoint register. DBR1 is programmed for 1 of 2 operations: Address mask for DBR1, OR Second data address breakpoint

The DBCON register controls the behavior of the data address breakpoint registers.

**Table 112. Data Breakpoint Controls Register (DBCON)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | | | | | | | | M | | | | | E1 | | E0 | |
| reset value: 0x00000000 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

| Bits | Access | Description |
|---|---|---|
| 31:9 | Read-Unpredictable / Write-As-Zero | Reserved |
| 8 | Read / Write | DBR1 Mode (M)<br>0 = DBR1 = Data Address Breakpoint<br>1 = DBR1 = Data Address Mask |
| 7:4 | Read-Unpredictable / Write-As-Zero | Reserved |
| 3:2 | Read / Write | DBR1 Enable (E1)<br>When DBR1 = Data Address Breakpoint<br>0b00: DBR1 disabled<br>0b01: DBR1 enabled, Store only<br>0b10: DBR1 enabled, Any data access, load or store<br>0b11: DBR1 enabled, Load only<br>When DBR1 = Data Address Mask this field has no effect |
| 1:0 | Read / Write | DBR0 Enable (E0)<br>0b00: DBR0 disabled<br>0b01: DBR0 enabled, Store only<br>0b10: DBR0 enabled, Any data access, load or store<br>0b11: DBR0 enabled, Load only |

When DBR1 is programmed as a data address mask, it is used in conjunction with the address in DBR0. Using DBR1 as a data address mask allows a range of addresses to generate a data breakpoint. The bits set in DBR1 causes the processor to ignore those bits when comparing the address of a memory access with the address in DBR0. The processor ignores the E1 field of DBCON when DBR1 is selected as a data address mask. The mask is used only when DBR0 is enabled.

When DBR1 is programmed as a second data address breakpoint, it functions independently of DBR0. In this case, the DBCON.E1 controls DBR1.

Only program data breakpoint address registers while that address register is disabled in DBCON. Programming a DBR register while it is enabled results in unpredictable behavior.

A data breakpoint is triggered when the memory access matches the access type and the address of any byte within the memory access matches the address in DBRx. For example, **LDR** triggers a breakpoint when DBCON.E0 is 0b10 or 0b11, and the address of any of the 4 bytes accessed by the load matches the address in DBR0.

The processor does not trigger data breakpoints for the **PLD** instruction or any CP15, register 7, 8, 9, or 10 functions (with the exception of Allocate L1 Data Cache Line). Any other type of memory access triggers a data breakpoint.

The Allocate L1 Data Cache Line function in CP15, register 7 is treated as store for data breakpoint purposes. This function takes a VA as an operand, but the address comparison occurs on the MVA. Thus, an address match occurs when any MVA within the allocated cache line matches the programmed data breakpoint address.

For data breakpoint purposes the **SWP** and **SWPB** instructions are treated as stores - these do not cause a data breakpoint when the breakpoint is set up to break on loads only and an address match occurs.

On unaligned memory accesses, the addresses used for the breakpoint address comparison are aligned down to the natural boundary of the instruction (in other words, half-word access aligned down to half-word boundary, word access aligned down to word-boundary, etc.).

When a memory access triggers a data breakpoint, the breakpoint is reported after the access is issued. The memory access is not aborted by the processor. However, the data breakpoint generates an exception before the next instruction executes. The actual timing of when the access completes with respect to the start of the debug handler depends on the memory configuration.

### 12.3.8 Software Breakpoints

Software breakpoints are generated using the BKPT instruction.

| | |
|---|---|
| Mnemonic: | BKPT (See *ARM Architecture Version 5TE Specification*) |
| Operation: | When DCSR[31] = 0, BKPT is a NOP;<br>When DCSR[31] =1, BKPT causes a debug exception |

## 12.4 JTAG Communications

A debug handler running on 3rd generation microarchitecture communicates with a host debugger using the transmit (TX) and receive (RX) registers on the microarchitecture. A debugger accesses these registers through the JTAG interface, using the DBGTX and DBGRX JTAG instructions.

Handshaking between a debug handler and a debugger ensures synchronized access of the TX and RX registers. Handshaking bits are available on the microarchitecture transmit/receive control register (TXRXCTRL). A debugger accesses the same handshaking bits through the DBGRX and DBGTX JTAG registers.

*Note:* While the following sections specifically refer to communications between a debug handler and a debugger, it really applies to communications between any privileged SW running on 3rd generation microarchitecture and an external JTAG controller.

This section discusses the JTAG communications registers and capabilities from the point of view of SW running on 3rd generation microarchitecture. Section 12.5, "Debug JTAG Access" on page 194 discusses the JTAG communications from the point of view of an external JTAG controller.

## 12.4.1 Transmit/Receive Control Register (TXRXCTRL)

**Table 113. Transmit/Receive Control Register (CRn = 14, CRm = 0)**

| Function | CRn | CRm | Instruction |
|---|---|---|---|
| Transmit/Receive Control Register (TXRXCTRL) | 0b1110 | 0b0000 | MRC p14, 0, Rd, c14, c0, 0<br>MCR p14, 0, Rd, c14, c0, 0 |

The TXRXCTRL register contains handshaking bits used by the debug handler to synchronize access to the TX and RX registers. The TX and RX registers have individual synchronization bits.

The TXRXCTRL register also contains two other bits to support high-speed download. One bit indicates an overflow condition that occurs when the debugger attempts to write the RX register before the debug handler has read the previous data written to RX. The other bit is used by the debug handler as a branch flag during high-speed download.

All of the bits in the TXRXCTRL register are placed such that these are read directly into the CC flags in the CPSR with an MRC (with Rd = PC). The subsequent instruction then conditionally executes based on the updated CC value.

**Table 114. TXRX Control Register (TXRXCTRL)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RR | OV | D | TR | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

| Bits | Access | Description | Reset Value | TRST Value |
|---|---|---|---|---|
| 31 | SW Read-Only / Write-Ignored<br>JTAG Read / Write | RX Ready Flag (RR) | 0 | 0 |
| 30 | SW Read / Write | RX overflow flag (OV) | 0 | unchanged |
| 29 | SW Read-Only / Write-Ignored<br>JTAG Write-Only | High-speed download flag (D) | unchanged | 0 |
| 28 | SW Read-Only / Write-Ignored<br>JTAG Read-Only | TX Ready (TR) | 0 | unchanged |
| 27:0 | Read-Unpredictable / Write-As-Zero | Reserved | unpredictable | unpredictable |

#### 12.4.1.1 RX Register Ready Bit (RR)

The debugger and debug handler use the RR bit to synchronize accesses to RX. Normally, the debugger and debug handler use a handshaking scheme that requires both sides to poll the RR bit. To support higher download performance for large amounts of data, a high-speed download handshaking scheme is used. In this scheme, only the debug handler polls the RR bit before accessing the RX register, while the debugger continuously downloads a stream of data.

Table 115 shows the normal handshaking used to access the RX register.

**Table 115. Normal RX Handshaking**

| Debugger Actions |
|---|
| Debugger wants to send data to debug handler.<br>Before writing new data to the RX register, the debugger polls RR through JTAG until the bit is cleared.<br>After the debugger reads a '0' from the RR bit, it scans data into JTAG to write to the RX register and sets the valid bit. The write to the RX register automatically sets the RR bit. |
| **Debug Handler Actions** |
| Debug handler is expecting data from the debugger.<br>The debug handler polls the RR bit until it is set, indicating data in the RX register is valid.<br>Once the RR bit is set, the debug handler reads the new data from the RX register. The read operation automatically clears the RR bit. |

When data is being downloaded by the debugger, part of the normal handshaking is bypassed to allow the download rate to be increased. Table 116 shows the handshaking used when the debugger is doing a high-speed download. Note that before the high-speed download starts, both the debugger and debug handler must be synchronized, such that the debug handler is executing a routine that supports the high-speed download.

Although it is similar to the normal handshaking, the debugger polling of RR is bypassed with the assumption that the debug handler reads the previous data from RX before the debugger scans in the new data.

**Table 116. High-Speed Download Handshaking States**

| Debugger Actions |
|---|
| Debugger wants to transfer code into 3rd generation microarchitecture system memory.<br>Prior to starting download, the debugger must poll RR bit until it is clear. Once the RR bit is clear, indicating the debug handler is ready, the debugger starts the download.<br>The debugger scans data into JTAG to write to the RX register with the download bit and the valid bit set. Following the write to RX, the RR bit and D bit are automatically set in TXRXCTRL.<br>Without polling of RR to see whether the debug handler has read the data just scanned in, the debugger continues scanning in new data into JTAG for RX, with the download bit and the valid bit set.<br>An overflow condition occurs when the debug handler does not read the previous data before the debugger completes scanning in the new data, (see Section 12.4.1.2, "Overflow Flag (OV)" on page 191 for more details on the overflow condition).<br>After completing the download, the debugger clears the D bit allowing the debug handler to exit the download loop. |
| **Debug Handler Actions** |
| Debug handler is in a routine waiting to write data out to memory. The routine loops based on the D bit in TXRXCTRL.<br>The debug handler polls the RR bit until it is set. It then reads the Rx register, and writes it out to memory. The handler loops, repeating these operations until the debugger clears the D bit. |

### 12.4.1.2 Overflow Flag (OV)

The Overflow flag is a sticky flag that is set when the debugger writes to the RX register while the RR bit is set.

The flag is used during high-speed download to indicate that some data was lost. The assumption during high-speed download is that the time it takes for the debugger to shift in the next data word is greater than the time necessary for the debug handler to process the previous data word. So, before the debugger shifts in the next data word, the handler is polling for that data.

However, when the handler incurs stalls that are long enough such that the handler is still processing the previous data when the debugger completes shifting in the next data word, an overflow condition occurs and the OV bit is set.

Once set, the overflow flag remains set, until cleared by a write to TXRXCTRL by software. After the debugger completes the download, it examines the OV bit to determine when an overflow occurred. The debug handler software is responsible for saving the address of the last valid store before the overflow occurred.

### 12.4.1.3 Download Flag (D)

The value of the download flag is set by the debugger through JTAG. The debug handler uses this flag during high-speed download in place of a loop counter.

The download flag becomes especially useful when an overflow occurs. When a loop counter is used, and an overflow occurs, the debug handler cannot determine how many data words overflowed. Therefore the debug handler counter gets out of sync with the debugger - the debugger finishes downloading the data, but the debug handler counter indicates there is more data to be downloaded - this results in unpredictable behavior of the debug handler.

Using the download flag, the debug handler loops until the debugger clears the flag. Therefore, when doing a high-speed download, for each data word downloaded, the debugger sets the D bit.

### 12.4.1.4 TX Register Ready Bit (TR)

The debugger and debug handler use the TR bit to synchronize accesses to the TX register. The debugger and debug handler must poll the TR bit before accessing the TX register. Table 117 shows the handshaking used to access the TX register.

**Table 117. TX Handshaking**

| Debugger Actions |
|---|
| Debugger is expecting data from the debug handler.<br>Before reading data from the TX register, the debugger polls the TR bit through JTAG until the bit is set. NOTE: while polling TR, the debugger must scan out the TR bit and the TX register data.<br>Reading a `1' from the TR bit, indicates that the TX data scanned out is valid<br>The action of scanning out data when the TR bit is set, automatically clears TR. |
| **Debug Handler Actions** |
| Debug handler wants to send data to the debugger (in response to a previous request).<br>The debug handler polls the TR bit to determine when the TX register is empty (any previous data has been read out by the debugger). The handler polls the TR bit until it is clear.<br>Once the TR bit is clear, the debug handler writes new data to the TX register. The write operation automatically sets the TR bit. |

### 12.4.1.5 Conditional Execution Using TXRXCTRL

All of the bits in TXRXCTRL are placed such that these are read directly into the CC flags using an MCR instruction. To simplify the debug handler, read the TXRXCTRL register using the following instruction:

```
mrc p14, 0, r15, C14, C0, 0
```

This instruction directly updates the condition codes in the CPSR. The debug handler then conditionally executes based on each CC bit. Table 118 shows the mnemonic extension to conditionally execute based on whether the TXRXCTRL bit is set or clear.

**Table 118. TXRXCTRL Mnemonic Extensions**

| TXRXCTRL Bit | Mnemonic Extension to Execute When Bit Sset | Mmnemonic Eextension to Execute When Bit Clear |
|---|---|---|
| 31 (to N flag) | MI | PL |
| 30 (to Z flag) | EQ | NE |
| 29 (to C flag) | CS | CC |
| 28 (to V flag) | VS | VC |

The following example is a code sequence in which the debug handler polls the TXRXCTRL handshaking bit to determine when the debugger has completed its write to RX and the data is ready for the debug handler to read.

```
loop:   mrc     p14, 0, r15, c14, c0, 0# read the handshaking bit in TXRXCTRL
        mrcmi   p14, 0, r0, c9, c0, 0  # if RX is valid, read it
        bpl     loop                   # if RX is not valid, loop
```

## 12.4.2 Transmit Register (TX)

**Table 119. Transmit Register (CRn = 8, CRm = 0)**

| Function | CRn | CRm | Instruction |
|---|---|---|---|
| Transmit Register (TX) | 0b1000 | 0b0000 | MCR p14, 0, Rd, c8, c0, 0 |

The TX register is the debug handler transmit buffer. The debug handler sends data to the debugger through this register.

**Table 120. TX Register**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| TX | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| reset value: unpredictable | | | | | | | | | | | | | | | | TRST Value: unchanged | | | | | | | | | | | | | | | |

| Bits | Access | Description |
|---|---|---|
| 31:0 | SW Read / Write<br>JTAG Read-Only | Debug handler writes data to send to debugger |

Since the TX register is accessed by the debug handler (using MCR) and the debugger (through JTAG), handshaking is required to prevent the debug handler from writing new data before the debugger reads the previous data.

The TX register handshaking is described in Table 117, "TX Handshaking" on page 191.

## 12.4.3 Receive Register (RX)

**Table 121. Receive Register (CRn = 9, CRm = 0)**

| Function | CRn | CRm | Instruction |
|---|---|---|---|
| Receive Register (RX) | 0b1001 | 0b0000 | MRC p14, 0, Rd, c9, c0, 0 |

The RX register is the receive buffer used by the debug handler to get data sent by the debugger through the JTAG interface.

**Table 122. RX Register**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RX | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| reset value: unpredictable | | | | | | | | | | | | | | | | TRST value: unpredictable | | | | | | | | | | | | | | | |

| Bits | Access | Description |
|---|---|---|
| 31:0 | SW Read-Only / Write-Unpredictable<br>JTAG Write-Only | Software reads to receives data/commands from debugger |

Since the RX register is accessed by the debug handler (using MRC) and the debugger (through JTAG), handshaking is required to prevent the debugger from writing new data to the register before the debug handler reads the previous data out. The handshaking is described in Section 12.4.1.1, "RX Register Ready Bit (RR)" on page 190.

## 12.5 Debug JTAG Access

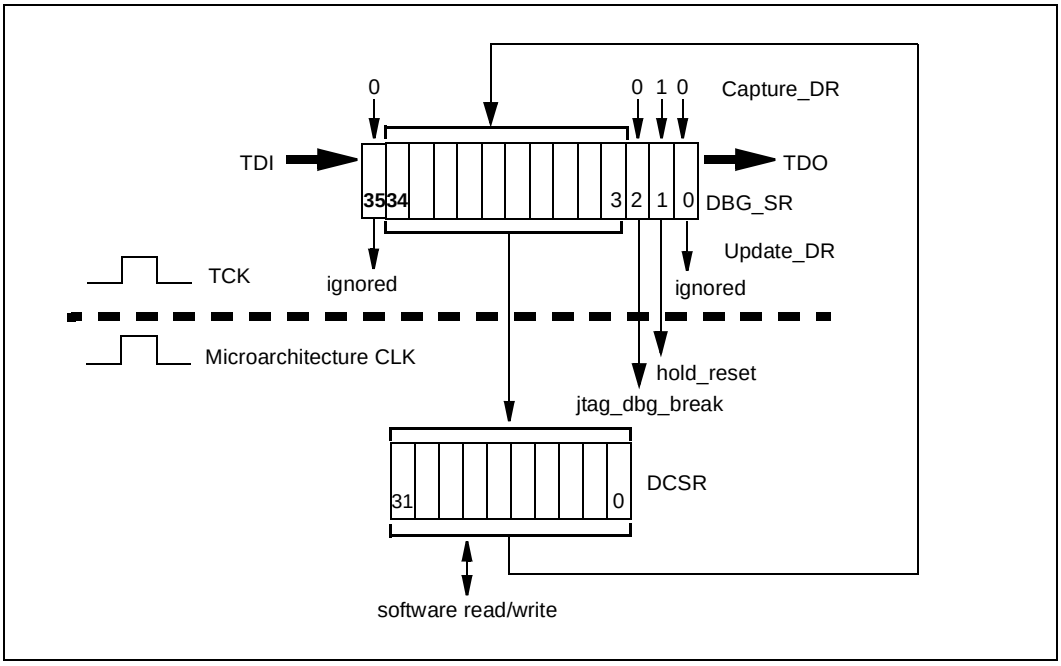
There are four JTAG instructions used by the debugger during software debug: LDSRAM, SELDCSR, DBGTX and DBGRX. LDSRAM is described in Section 12.7, "Debug SRAM" on page 209. The other three JTAG instructions are described in this section. SELDCSR, DBGTX and DBGRX each use a 36-bit shift register to scan in new data and scan out captured data.

### 12.5.1 SELDCSR JTAG Register

The 'SELDCSR' JTAG instruction selects the DCSR JTAG data register. The JTAG opcode is '0b0001001'. When the SELDCSR JTAG instruction is in the JTAG instruction register, the debugger directly accesses the Debug Control and Status Register (DCSR). The debugger only modifies certain bits through JTAG but reads the entire register.

The SELDCSR instruction also allows the debugger to generate an external debug break and set the hold_reset signal, which is used when downloading code into the Debug SRAM during reset.

**Figure 14. SELDCSR**




A Capture_DR loads the current DCSR value into DBG_SR[34:3]. The other bits in DBG_SR are loaded as shown in Figure 14.

A new DCSR value is scanned into DBG_SR, and the previous value out, during the Shift_DR state. When scanning in a new DCSR value into the DBG_SR, care must be taken to also set up DBG_SR[2:1] to prevent undesirable behavior.

Update_DR parallel loads the new DCSR value into the DCSR. All bits defined as JTAG writable in Table 103, "Debug Control and Status Register (DCSR)" on page 175 are updated.

Access to the DCSR must be synchronized between the debugger and debug handler. When one side writes the DCSR at the same side the other side reads the DCSR, the results are unpredictable.

#### 12.5.1.1 hold_reset

The debugger uses hold_reset when loading code into the Debug SRAM during a processor reset. Details about loading code into the Debug SRAM are in Section 12.7, "Debug SRAM" on page 209.

The debugger must set hold_reset before or during assertion of the reset pin. Once hold_reset is set, the reset pin is de-asserted, and the processor internally remains in reset. The debugger then loads debug handler code into the Debug SRAM before the processor begins executing any code.

Once the code download is complete, the debugger must clear hold_reset. This allows the processor to come out of reset, and execution begins at the reset vector.

A debugger sets hold_reset in one of two ways:

- Either by taking the JTAG state machine into the Capture_DR state, which automatically loads DBG_SR[1] with '1', then the Exit2 state, followed by the Update_Dr state. This sets the hold_reset, clear jtag_dbg_break, and leave the DCSR unchanged (the DCSR bits captured in DBG_SR[34:3] are written back to the DCSR on the Update_DR).
- Alternatively, a '1' is scanned into DBG_SR[1], with the appropriate value scanned in for the DCSR and ext_dbg_break. The hold_reset bit updates following entry into the Update_DR state.

The hold_reset bit is cleared by scanning in a '0' to DBG_SR[1] and scanning in the appropriate values for the DCSR and jtag_dbg_break. The hold_reset bit is also cleared following a JTAG Reset.

#### 12.5.1.2 jtag_dbg_break

The jtag_dbg_break allows the debugger to asynchronously generate a JTAG debug break and re-direct execution on the microarchitecture to a debug handling routine. Note that jtag_dbg_break is not qualified with global debug enable. This allows a debugger to generate a debug break at anytime (for example, to initiate a hot-debug session).

A debugger sets a JTAG debug break by scanning a '1' into DBG_SR[2] (and scanning in the desired value for the DCSR JTAG writable bits in DBG_SR[34:3]) and entering the Update_DR state.

Once jtag_dbg_break is set, it remains set internally until a debug exception occurs or a new value is scanned in which clears the bit. In Monitor Mode, JTAG debug breaks detected during abort mode are pended until the processor exits abort mode. In Halt Mode, JTAG debug breaks detected during SDS are pended until the processor exits SDS. When a JTAG debug break is detected outside of these two cases, the processor ceases executing instructions as quickly as possible, clears the internal jtag_dbg_break bit, and branches to the debug handler (Halt Mode) or abort handler (Monitor Mode).

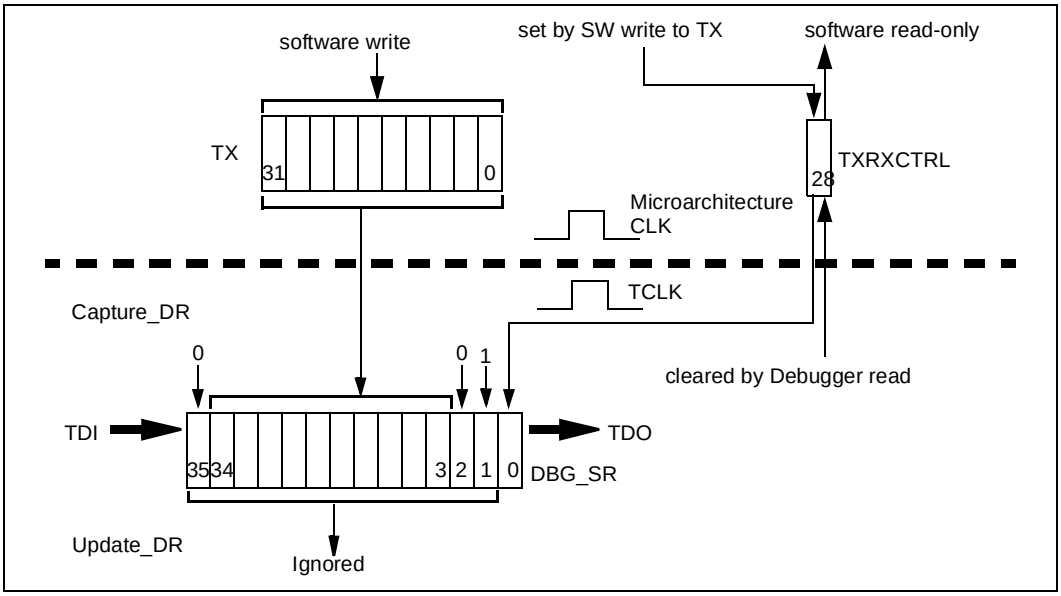
#### 12.5.1.3 DCSR (DBG_SR[34:3])

The JTAG writable bits in the DCSR are updated with the value loaded into DBG_SR[34:3] following an Update_DR.

### 12.5.2 DBGTX JTAG Register

The 'DBGTX' JTAG instruction selects the DBGTX JTAG data register. The JTAG opcode for this instruction is '0b0010000'. The debug handler uses the DBGTX data register to send data to the debugger. A protocol is setup between the debugger and debug handler to allow the debug handler to signal an entry into debug mode, and once in debug mode to transmit data requested by the debugger.

**Figure 15. DBGTX**




A Capture_DR loads the TX register value into DBG_SR[34:3] and TXRXCTRL.TR into DBG_SR[0]. The other bits in DBG_SR are loaded as shown in Figure 15.

The captured TX value is scanned out during the Shift_DR state. Entering Shift_DR after capturing a '1' in DBG_SR[0] automatically clears TXRXCTRL.TR. Note that the Shift_DR must immediately follow the Capture_DR to ensure that TXRXCTRL.TR gets cleared.

Data scanned in is ignored on an Update_DR.

#### 12.5.2.1 DBG_SR[0]

DBG_SR[0] is used for part of the synchronization that occurs between the debugger and debug handler for accessing TX. The debugger polls DBG_SR[0] to determine when the TX register contains valid data from the debug handler.

A '1' captured in DBG_SR[0] indicates valid captured TX data. After capturing valid data, the act of shifting out the data automatically clears TXRXCTRL.TR. Therefore, the debugger must not go through the Update_DR state when the TX data is valid, without first scanning out the entire TX register value.

A '0' indicates there is no new data from the debug handler in the TX register.

#### 12.5.2.2 TX (DBG_SR[34:3])

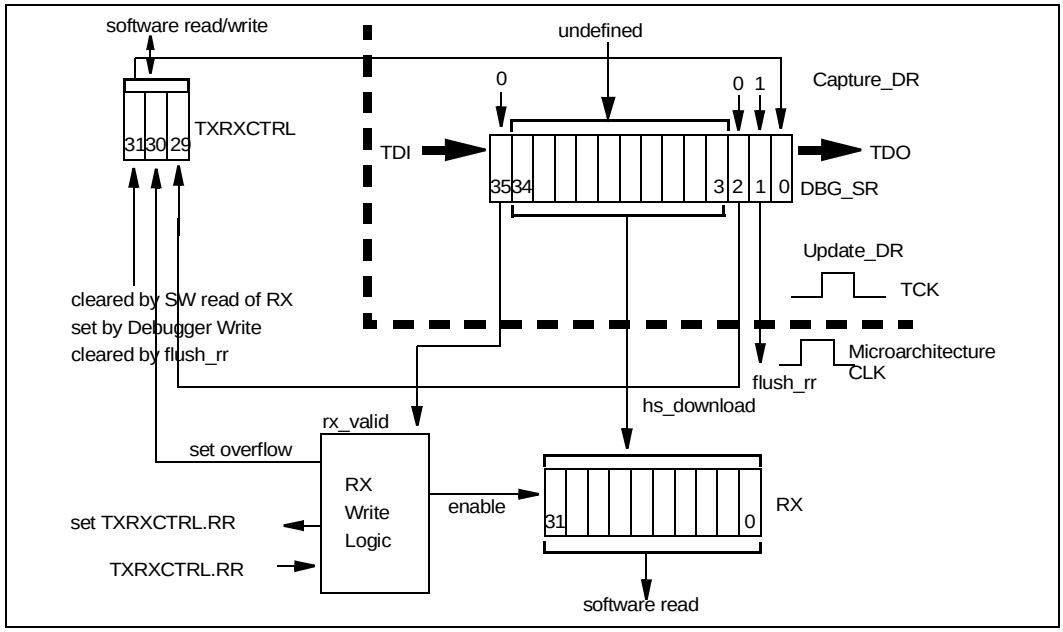
DBG_SR[34:3] is updated with the contents of the TX register following an Update_DR. When DBG_SR[0] is '0' following an Update_DR, the contents of DBG_SR[34:3] are unpredictable.

## 12.5.3 DBGRX JTAG Register

The ‘DBGRX’ JTAG instruction selects the DBGRX JTAG data register. The JTAG opcode for this instruction is ‘0b0000010’. The debug handler uses the DBGRX data register to receive information from the debugger. A protocol is setup between the debugger and debug handler to allow the handler to identify data values and commands.

The DBGRX data register also contain bits to support high-speed download and to “invalidate” the contents of the RX register.

**Figure 16. DBGRX**




A Capture_DR loads the value of TXRXCTRL.RR into DBG_SR[0]. The other bits in DBG_SR are loaded as shown in Figure 16.

The captured data is scanned out during the Shift_DR state. Care must be taken while scanning in data. While polling TXRXCTRL.RR, incorrectly setting rx_valid or flush_rr causes unpredictable behavior following an Update_DR.

Following an Update_DR the scanned in data takes effect.

### 12.5.3.1 RX Write Logic

The RX write logic (Figure 16) serves the following functions:

1. RX Write Enable: RX register only gets updated when rx_valid is set and unaffected when rx_valid is clear or an overflow occurs. In particular, when the debugger is polling DBG_SR[0], as long as rx_valid is 0, Update_DR does not modify RX.
2. Set TXRXCTRL.RR: When debugger writes new data to RX, TXRXCTRL.RR is automatically set signalling to debug handler that RX register contains valid data.
3. Set TXRXCTRL.OV: When debugger scans in a value with rx_valid set and TXRXCTRL.RR already set, TXRXCTRL.OV is automatically set. For instance, during high-speed download, the debugger does not poll to see when the handler has read previous data. When the debug handler stalls long enough, the debugger tries to write a new data to RX before the handler has read previous data. When occurs, RX write logic sets TXRXCTRL.OV and blocks the RX register write.

#### 12.5.3.2 DBG_SR[0]

DBG_SR[0] is used for part of the synchronization that occurs between the debugger and debug handler for accessing RX. The debugger polls DBG_SR[0] to determine when the handler has read the previous data from RX, and it is safe to write new data.

A '1' read in DBG_SR[0] indicates that the RX register contains valid data which has not yet been read by the debug handler. A '0' indicates it is safe for the debugger to write new data to the RX register.

#### 12.5.3.3 flush_rr

The flush_rr bit allows the debugger to flush a previous data value written to RX, assuming the debug handler has not read that value yet. Setting flush_rr clears TXRXCTRL.RR.

#### 12.5.3.4 hs_download

The hs_download bit is provided for use during high speed download. This bit is written directly to TXRXCTRL.D. The debugger uses this bit to improve performance when downloading a block of code or data to the target system memory.

A protocol is setup between the debugger and debug handler using this bit. For example, while this bit is set, the debugger continuously downloads new data without polling TXRXCTRL.RR. The debug handler uses TXRXCTRL.D as a branch flag to loop while there is more data to come. The debugger clears this bit to indicate the end of the block and allow the debug handler to exit its loop.

Using hs_download as a branch flags eliminates the need for a loop counter in the debug handler code. This avoids the problem where the debugger loop counter is out of synchronization with the debug handler counter because of overflow conditions that have occurred.

#### 12.5.3.5 RX (DBG_SR[34:3])

DBG_SR[34:3] is written to RX following an Update_DR when the RX Write Logic enables the RX register to be updated.

#### 12.5.3.6 rx_valid

The debugger sets the rx_valid bit to indicate the data scanned into DBG_SR[34:3] is valid data to be written to RX. When this bit is set, the data scanned into the DBG_SR is written to RX following an Update_DR. When rx_valid is not set Update_DR does not affect RX.

*Note:* The actions of flush_rr and hs_download are not qualified with rx_valid.

*Note:* Setting rx_valid and flush_rr at the same time result in unpredictable behavior.

## 12.6 Trace Buffer

3rd generation microarchitecture has a 256 entry trace buffer that provides the ability to capture control flow information for debugging an application. Two modes are supported:

1. Fill-once Mode: The trace buffer fills up completely and generates a debug exception.
2. Wrap-around Mode: The trace buffer continuously fills up and wraps around until it is disabled (either by a debug exception or by software).

### 12.6.1 Definitions

In the description of the trace buffer, the following terminology is used:

**Table 123. Trace Buffer Terminology**

| Term | Meaning |
|---|---|
| trace buffer entry | an individual 8-bit unit of the trace buffer. The trace buffer contains 256 of these 8-bit units. |
| trace message | a group of 1 or more entries. Trace messages indicate a type of program flow change and any related address information |
| message header | a specific entry which contains the encoding and incremental instruction count value of the current trace message. A 1-entry trace message is just a message header. |

## 12.6.2 Trace Buffer Registers

A summary of trace buffer registers is shown in Table 101, "CP14 Software Debug Registers" on page 174. The following sections provide a detailed description of these registers.

### 12.6.2.1 Checkpoint Registers

**Table 124. Checkpoint Registers (CRn = 12,13, CRm = 0)**

| Function | CRn | CRm | Instruction |
|---|---|---|---|
| Checkpoint Register 0 (CHKPT0) | 0b1100 | 0b0000 | MRC p14, 0, Rd, c12, c0, 0<br>MCR p14, 0, Rd, c12, c0, 0 |
| Checkpoint Register 1 (CHKPT1) | 0b1101 | 0b0000 | MRC p14, 0, Rd, c13, c0, 0<br>MCR p14, 0, Rd, c13, c0, 0 |

The two checkpoint registers (CHKPT0, CHKPT1) provide a reference address to the debugger for reconstructing a trace history.

The debugger reconstructs a trace history, starting at the oldest trace message going forward, to the most recent trace message. In fill-once mode and wrap-around mode, before the trace buffer wraps around, the trace is reconstructed by starting from the point in the code where the trace buffer was first enabled (typically this occurs at the end of the debug handler, where the exit of the debug handler is traced as an indirect branch, providing a reference starting address).

The difficulty occurs in wrap-around mode when the trace buffer wraps around at least once. In this case the debugger gets a snapshot of the last N control flow changes in the program, where N <= size of buffer. The debugger does not know the starting address of the oldest trace message read from the trace buffer. In this case, the checkpoint registers are used to identify a starting address for reconstructing the trace history.

**Table 125. Checkpoint Register (CHKPTx)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CHKPTx | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | T |

reset value: Unpredictable

| Bits | Access | Description |
|---|---|---|
| 31:1 | Read / Write | CHKPTx<br>target address for corresponding entry in trace buffer |
| 0 | Read / Write | Thumb/ARM (T)<br>indicates whether target address is in ARM or Thumb mode. (see Section 12.6.4, "Tracing Thumb Code" on page 206)<br>0 = ARM target<br>1 = Thumb target |

When the trace buffer is enabled, reading and writing to either checkpoint register has unpredictable results. When the trace buffer is disabled, writing to a checkpoint register sets the register to the value written. Reading the checkpoint registers returns the value of the register.

In normal usage, the checkpoint registers hold target addresses of checkpointed trace messages in the trace buffer. Only direct and indirect trace messages are checkpointed. Exception and roll-over messages are never checkpointed. The processor sets bit 6 of the message header to indicate that a trace message has been checkpointed (refer to Table 128).

The trace buffer contains no more than two checkpointed trace messages at any given time. When the trace buffer contains only one checkpointed message, the corresponding checkpoint register is CHKPT0. When the trace buffer wraps around, two messages typically are checkpointed, usually about half a buffers length apart. In this case, the first (oldest) checkpointed message read from the trace buffer corresponds to CHKPT1, the second checkpointed message corresponds to CHKPT0.

Although the checkpoint registers are provided for wrap-around mode, these are still valid in fill-once mode.

### 12.6.2.2 Trace Buffer Register (TBREG)

**Table 126. Trace Buffer Register (CRn = 11, CRm = 0)**

| Function | CRn | CRm | Instruction |
|---|---|---|---|
| Trace Buffer Register (TBREG) | 0b1011 | 0b0000 | MRC p14, 0, Rd, c11, c0, 0 |

Software reads the contents of the trace buffer through TBREG. Software only reads the trace buffer when it is disabled. Reading the trace buffer while it is enabled, causes unpredictable behavior of the trace buffer. Writes to the trace buffer have unpredictable results.

Reading TBREG pops the oldest trace buffer entry in the least significant 8 bits of the register. The entry is either a message header or part of the 32-bit address associated with an indirect branch message.

Table 127 shows the format of the Trace Buffer Register.

**Table 127. Trace Buffer Register (TBREG)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | | | | | | | | | Data | | | | | | | |

reset value: Unpredictable

| Bits | Access | Description |
|---|---|---|
| 31:8 | Read-Unpredictable / Write-Unpredictable | Reserved |
| 7:0 | Read / Write-Unpredictable | Data<br>Trace Buffer Data |

## 12.6.3 Trace Messages

Trace messages consist of one or more trace buffer entries. Most messages are a single entry consisting of a message header indicating a type of control flow change or a counter rollover.

The target address for single entry trace messages is either encoded in the message header (as for exceptions), or determined by looking at the instruction word in system memory (as for direct branches).

Indirect branch messages require five entries. One entry is the message header identifying it as an indirect branch. The target address of the indirect branch makes up the other four entries.

The following sections describe the trace messages in detail.

### 12.6.3.1 Trace Message Formats

There are two message header formats, (exception and non-exception) as shown in Figure 17.

**Figure 17. Message Header Formats**

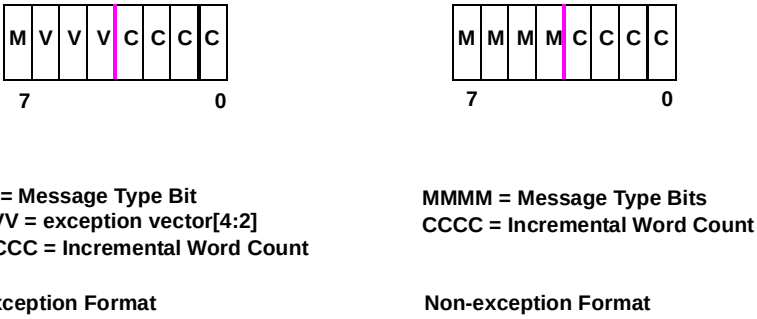



Table 128 shows the individual types of trace messages and their formats.

**Table 128. Trace Messages**

| Message Name | Message Type | Message Header Format | # address bytes |
|---|---|---|---|
| Exception | exception | 0b0VVV CCCC | 0 |
| Direct Branch[a] | non-exception | 0b1000 CCCC | 0 |
| Checkpointed Direct Branch[a] | non-exception | 0b1100 CCCC | 0 |
| Indirect Branch[b] | non-exception | 0b1001 CCCC | 4 |
| Checkpointed Indirect Branch[b] | non-exception | 0b1101 CCCC | 4 |
| Roll-over | non-exception | 0b1111 1111 | 0 |

a. Direct branches include ARM and THUMB bl, b
b. Indirect branches include ARM ldm, ldr, and dproc to PC; ARM and THUMB bx, blx(1) and blx(2); and THUMB pop.

### 12.6.3.2 Exception Messages

When any kind of exception occurs, an exception message, consisting simply of a message header is placed in the trace buffer. In the message header, the message type bit (M) is always set to 0. The exception vector (VVV) field specifies bits[4:2] of the vector address (offset from the base of default or relocated vector table). This information allows the debugger to determine which exception occurred.

The incremental word count (CCCC) is the instruction count since the last control flow change (not including the current instruction for undef, SWI, and pre-fetch abort). The instruction count includes instructions that were executed and conditional instructions that were not executed due to the condition of the instruction not matching the CC flags.

An incremental word count of 0 indicates that 0 instructions executed since the last control flow change and the current exception. For example, when a branch is immediate followed by a SWI, a direct branch message (for the branch) is followed by an exception message (for the SWI) in the trace buffer. The incremental word count in the exception message is 0, meaning that 0 instructions executed after the last control flow change (the branch) and before the current control flow change (the SWI). Instead of the SWI, when an IRQ was handled immediately after the branch (before any other instructions executed), the incremental word count is still be 0, since no instructions executed after the branch and before the interrupt was handled.

An incremental word count of 0b1111 indicates that 15 instructions executed between the last branch and the exception. In this case, an exception was either caused by the 16th instruction (when it is an undefined instruction exception, pre-fetch abort, or SWI) or generated before the 16th instruction executed (for FIQ, IRQ, or data abort).

*Note:* There is an incremental word count special case related with precise data aborts. For a precise data abort on a load to the PC (LDR or LDM), the incremental word count is consistent with the above description (in other words aborting instruction is not counted). For all other precise data aborts, the instruction that causes the data abort is included in the incremental word count in the exception message.

### 12.6.3.3 Non-exception Messages

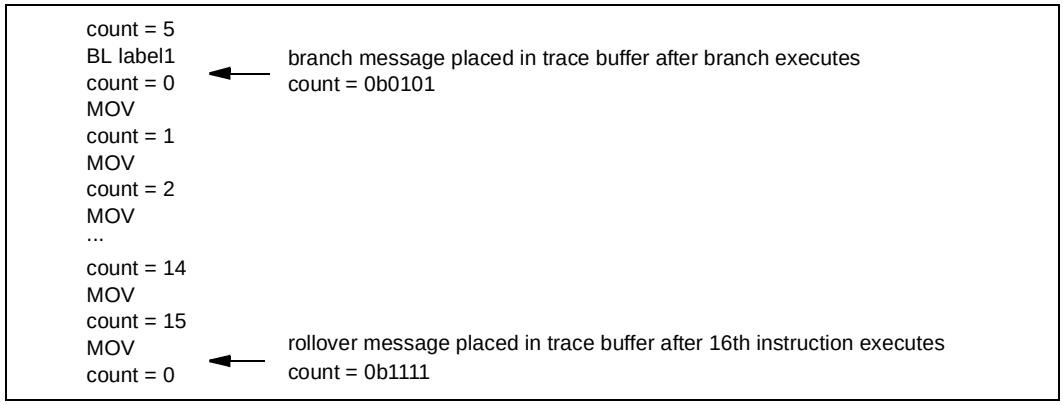
Non-exception messages used for direct and indirect branches, and rollover messages. The 4-bit message type field (MMMM) specifies type of message (Table 128).

The incremental word count (CCCC) is the instruction count since the last control flow change (excluding current branch). This includes executed and conditional instructions, that were not executed due to the condition of the instruction not matching the CC flags. In the case of back-to-back branches the incremental word count is 0 indicating no instructions executed after the last branch and before the current one.

A rollover message is used to keep track of long traces of code that do not have control flow changes. The rollover message means that 16 instructions have executed since the last program flow change or rollover message.

When the incremental word count reaches its maximum value of 15, a rollover message is written to the trace buffer following the next instruction (which is the 16th instruction to execute), as shown in Example 5. The incremental word count in the rollover message is 0b1111, indicating that 15 instructions have executed after the last branch and before the current non-branch instruction causing the rollover message.

**Example 5. Rollover Messages Examples**

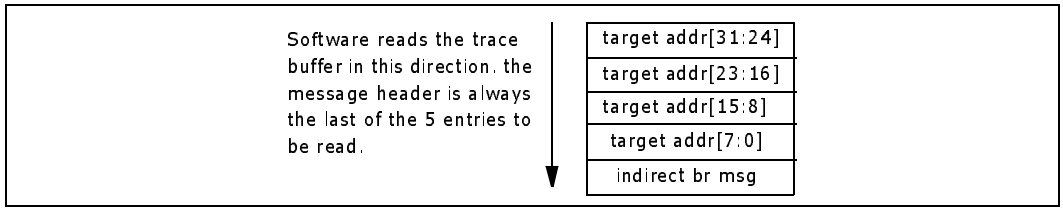



When the 16th instruction is a branch (direct or indirect), the appropriate message is placed in the trace buffer instead of the roll-over message. The incremental word count is still set to 0b1111, meaning 15 instructions executed between the last branch and the current branch.

### 12.6.3.4 Reading Indirect Branch Messages

Only indirect branch messages contain additional address information. Indirect branch messages have four address entries specifying the target of that branch. When reading the trace buffer the MSB of the target address is read out first; the LSB is the fourth entry read out; and the indirect branch message header is the fifth entry read out. The entry organization of an indirect branch message is shown in Figure 18.

**Figure 18. Indirect Branch Message Organization**





### 12.6.4 Tracing Thumb Code

The trace buffer provides the capability to indicate whether the traced code is branching to code executing in Thumb or ARM mode. This capability is controlled by the Thumb Trace bit in the DCSR (DCSR[6]).

When this feature is enabled, the Trace Buffer uses bit 0 of branch target addresses placed in the trace buffer (indirect branch target addresses) and in the Checkpoint registers (indirect or direct branch target addresses) to indicate whether the target of the branch is in ARM mode or Thumb mode.

On a branch to ARM mode (from ARM or Thumb mode), the Trace Buffer places a '0' in bit 0 of the target address. On a branch to Thumb mode (from ARM or Thumb mode), the Trace Buffer places a '1' in bit 0 of the target address.

All transitions into and out of Thumb Mode are traced as indirect branches. So, assuming the Trace Buffer does not wrap around, all of the Thumb entry and exit points are identifiable. Even when the trace buffer wraps around and the Thumb entry point is lost, all indirect branches from Thumb mode that remain in Thumb mode set bit 0 of the indirect branch target address to '1'. This allows the trace tools to correctly trace Thumb code from the first indirect branch address (or checkpointed address) in the trace buffer. Since all exceptions exit Thumb mode, an exception trace message implies a Thumb exit point.

When this feature is disabled, all branch target addresses in the Trace Buffer and Checkpoint registers have bit 0 set to '0'.

## 12.6.5 Trace Buffer Usage

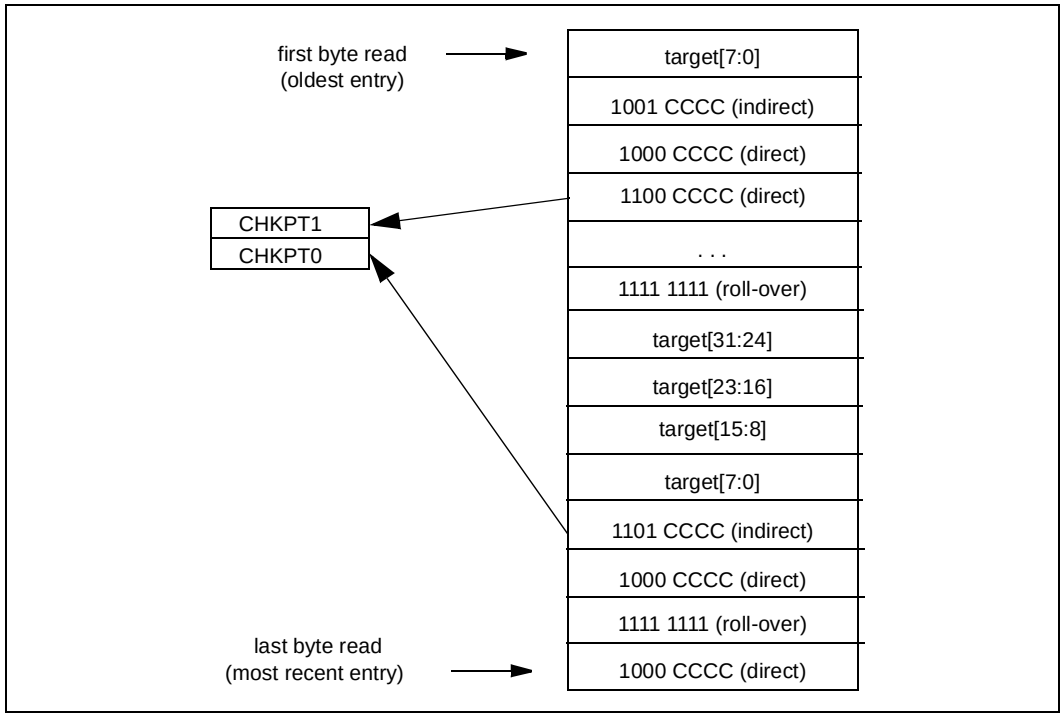
The trace buffer must be initialized prior to each usage. Initialization is done by reading the entire trace buffer through TBREG. The process of reading the trace buffer also clears it out (all entries are set to 0b0000 0000). Therefore, reading the contents of the trace buffer after capturing a trace re-initializes it for its next usage.

The trace buffer is used to capture a trace up to a processor reset or debug exception. Neither processor reset nor debug exceptions generate a trace message in the trace buffer.

Following a processor reset or debug exception, the trace buffer is disabled, however the contents are unaffected, so the debugger still reconstructs the trace up-to the disabling event. The debugger must read the entire trace buffer prior to re-enabling it.

After capturing a trace, the debugger must read the entire trace buffer before reconstructing the trace. The first entry read from the buffer represents the oldest trace history information in the buffer. The last (256th) entry read represents the most recent data in the buffer and is always a message header. The last entry provides the debugger with a well defined starting point for parsing individual trace messages from the buffer. Figure 19 is a high level view of the trace buffer.

**Figure 19. High Level View of Trace Buffer**




Since the trace buffer is cleared out prior to each use, all entries are initially 0b0000 0000. In cases where the trace buffer does not wrap-around (in fill-once mode or wrap-around mode), the debugger finds entries containing all 0s. The debugger identifies the end of the valid trace buffer contents by identifying the first message header containing 0s - since this is not a valid message header value.

In wrap-around mode, the debugger must be aware that the oldest trace message is a partial message. The debugger identifies a partial trace message by parsing the trace buffer and looking for an indirect branch message that does not have all four address entries.



Once the debugger has read and parsed the trace buffer, it re-creates the trace history starting with the oldest trace message working its way to the most recent.

- In fill-once mode, the return from the debug handler to the application generates an indirect branch trace message. The target address placed in the trace buffer is return address within the target application. This serves as the starting point for re-constructing the trace in fill-once mode.
- In wrap-around mode, the debugger uses the checkpoint registers and indirect branch trace messages to identify starting points for re-creating the trace.

In wrap-around mode, some of the older trace messages are unusable depending on where these are relative to the first checkpointed entry or indirect branch trace message.

The best case is when the oldest message in the trace buffer is checkpointed or is an indirect branch trace message. In this case the entire trace buffer contains valid data.

In the worst case, the first checkpointed entry is in the middle of the trace buffer. When the debugger cannot identify an older reference address, only 1/2 of the trace buffer contains usable trace information.

In fill-once mode, the entire trace buffer is usable, since the oldest entry is the indirect branch used to return to the application from the debug handler.

## 12.7 Debug SRAM

3rd generation microarchitecture has a on-microarchitecture Debug SRAM, for holding the debug handling routine used during JTAG debugging. A debugger downloads the code directly into the Debug SRAM through JTAG either during reset or while a program is running.

The remainder of this section describes the Debug SRAM in more details, as well as the methods for loading the SRAM through JTAG.

### 12.7.1 Debug SRAM Overview

The Debug SRAM is a 2 KB instruction RAM located on the 3rd generation microarchitecture.

The Debug SRAM is only programmed through JTAG. The target address is loaded through JTAG along with eight instruction words to place in the Debug SRAM starting at the specified address. The details for programming the Debug SRAM are discussed in the following sections. Any code already in the Debug SRAM at the target addresses is overwritten. The contents of the Debug SRAM are unaffected by a processor reset or a JTAG reset (assertion of TRST or transition of TAP controller into TLRS).

Instruction fetches are directed to the Debug SRAM following a debug exception in Halt Mode. When a debug exception occurs, execution begins at address 0 of the SRAM. Execution continues out of the Debug SRAM until the debug handler does a CPSR restore.

The Debug SRAM is a separate memory space from the application memory. Code in the Debug SRAM cannot be affected by application code. Also, a debug handler executing out of the Debug SRAM cannot branch to code in the application memory, without doing a CPSR restore.

Instruction accesses to the Debug SRAM have the following characteristics:

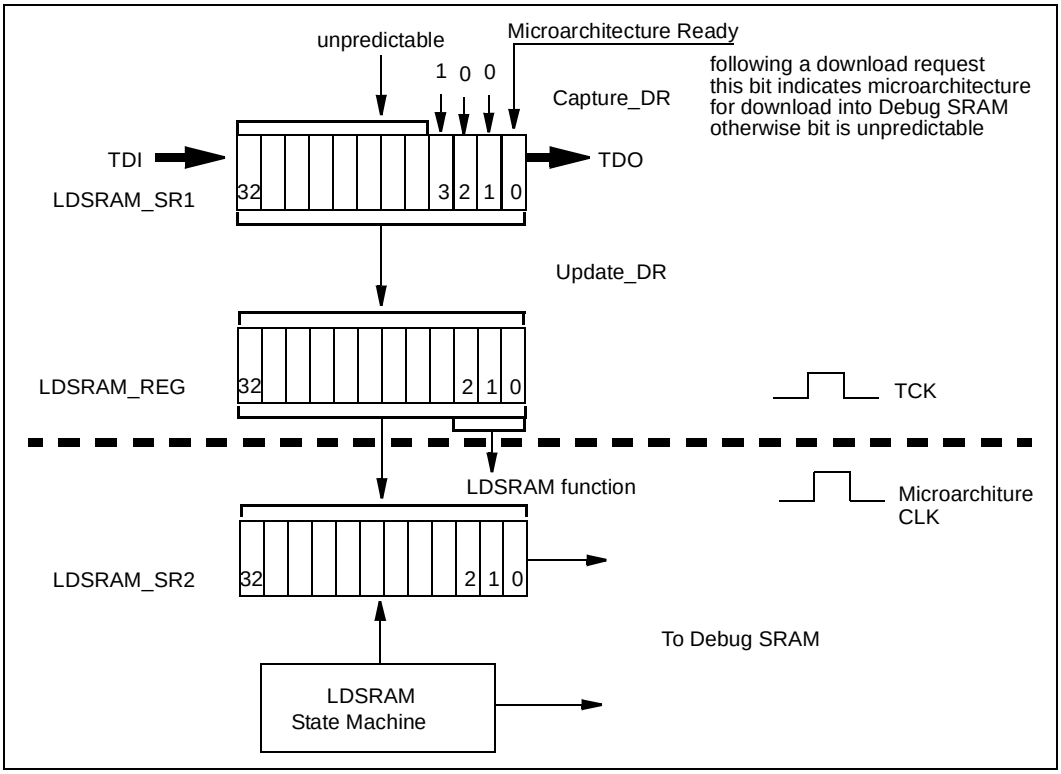
- no memory management protection checks;
- no memory management address translation;
- no PID remapping.
- no BTB interaction.
- Fetches past the end of the 2 KB Debug SRAM result in unpredictable behavior;

Data accesses never go to the Debug SRAM. Any data access by software running out of the Debug SRAM goes to the application memory space and uses the application's memory management setup and PID remapping.

## 12.7.2 LDSRAM JTAG Register

The LDSRAM JTAG instruction selects the JTAG data register for loading code into the Debug SRAM. The JTAG opcode for this instruction is '0b0000111'. The LDSRAM instruction must be in the JTAG instruction register in order to load code into the Debug SRAM through JTAG or to use any of the other LDSRAM functions listed in Table 129.

**Figure 20. LDSRAM JTAG Data Register**




The data loaded into LDSRAM_SR1 during a Capture_DR is as shown in Section 20, "LDSRAM JTAG Data Register" on page 210. Note that the values captured into LDSRAM_SR1 are used to facilitate the Download Request function (and its associated polling loop, see Section 12.7.3.1).

All specific LDSRAM functions and associated data are downloaded in 33-bit packets which are scanned into LDSRAM_SR1 during the Shift_DR state.

Update_DR parallel loads LDSRAM_SR1 into LDSRAM_REG which is then synchronized with the 3rd generation microarchitecture clock and loaded into the LDSRAM_SR2. When the function is set to Load Debug SRAM, the LDSRAM state machine kicks off and begins shifting code to the Debug SRAM.

Note that, when loading the Debug SRAM, there is a delay from the time of the Update_DR to the time the entire contents of LDSRAM_SR2 have been shifted to the Debug SRAM. Removing the LDSRAM JTAG instruction from the JTAG IR before the entire contents of LDSRAM_SR2 have been transferred, results in unpredictable behavior. Therefore, following the Update_DR for the last LDSRAM packet, the LDSRAM instruction must remain in the JTAG IR for a minimum of 20 TCKs. This ensures the last packet is correctly sent to the Debug SRAM.

### 12.7.3 LDSRAM Functions

3rd generation microarchitecture supports three LDSRAM JTAG functions as shown in Table 129. All other functions are NOPs or reserved.

**Table 129. LDSRAM JTAG Functions**

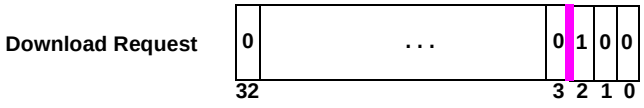
| Function | Encoding | Arguments | |
|---|---|---|---|
| | | Address | # Data Words |
| NOP | 0b000 | - | 0 |
| NOP | 0b001 | - | 0 |
| RESERVED | 0b010 | - | - |
| Load Debug SRAM | 0b011 | Address of line to load | 8 |
| Download Request | 0b100 | - | 0 |
| Download Complete | 0b101 | - | 0 |
| RESERVED | 0b100-0b111 | - | - |

#### 12.7.3.1 Download Request / Download Complete Functions

The Download Request function is used with the Download Complete function to support loading code into the Debug SRAM while the target application code is executing. In particular, this feature is used for hot-debug, which requires downloading a debug handler into the Debug SRAM, while the application is running. The steps for loading the Debug SRAM for hot-debug are described in Section 12.7.5.

The Download Request function allows the debugger to inform the microarchitecture that a download to the Debug SRAM is about to occur. This allows the microarchitecture to halt any activity which interferes with the download. The format of the Download Request function is shown in Figure 21

**Figure 21. Format of Download Request function**

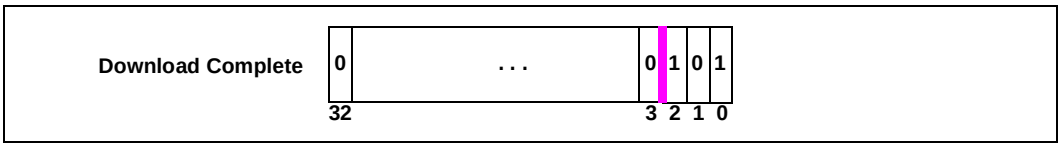



Following a Download Request, the debugger must poll the Microarchitecture Ready flag (LDSRAM_SR1[0], see Figure 20, "LDSRAM JTAG Data Register" on page 210) before downloading any code into the Debug SRAM. This flag provides an acknowledgement from the microarchitecture indicating that it is ready for the download. The polling is basically done by continuously scanning in a Download Request and checking the value of the Microarchitecture Ready flag scanned out.

Once the Microarchitecture Ready flag is read as a '1' by the debugger, it proceeds with downloading code into the Debug SRAM. The entire time the debugger is scanning out the Microarchitecture Ready flag, it must scan in the Download Request function. For each iteration of the polling loop, the debugger must take the TAP controller through the Capture_DR state. This ensures that the debugger sees the correct value of the Microarchitecture Reset flag.

After completing the code download into the Debug SRAM, the debugger must scan in the Download Complete function, informing the microarchitecture that it resumes normal activity. The Format of the Download Complete function is shown in Figure 22.

**Figure 22. Format of Download Complete function**



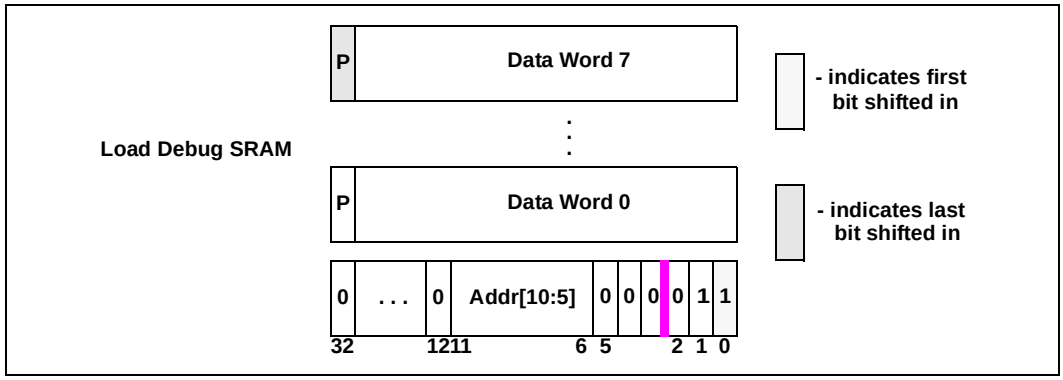

The debugger must not switch the JTAG instruction register value between the time of the initial Download Request function and the final Download Complete function, otherwise the results of the download are unpredictable.

LDSRAM_SR1[32:3] is set to 0 for the Download Request and Download Complete functions.

### 12.7.3.2 Load Debug SRAM Function

The debugger uses the Load Debug SRAM function to program code into the Debug SRAM. The function takes a target address and eight words of instructions to load. The address and data information is downloaded through JTAG in 33-bit packets. Figure 23 shows the packet format. The Load Debug SRAM function requires nine packets.

**Figure 23. Format of Load Debug SRAM function**




All packets are 33 bits in length. Bits [2:0] of the first packet specify the function to execute; bits[11:6] of the first packet specify an 8-word aligned address within the 2 KB Debug SRAM; bits[32:12,5:3] is set to 0.

Eight additional data packets are used to specify eight ARM instructions to be loaded into the Debug SRAM. Bits[31:0] of each data packet contains the instruction to download. Bit[32] of each data packet is the value of the parity for the data in that packet. (Parity = XOR of first 32 bits).

As shown in Figure 23, the first bit shifted in TDI is bit 0 of the first packet. After each 33-bit packet, the debugger must take the JTAG state machine into the Update_DR state. Following an Update_DR, the debugger immediately returns to the Shift_DR state (via Capture_DR) and begin shifting in the next 33-bit packet.

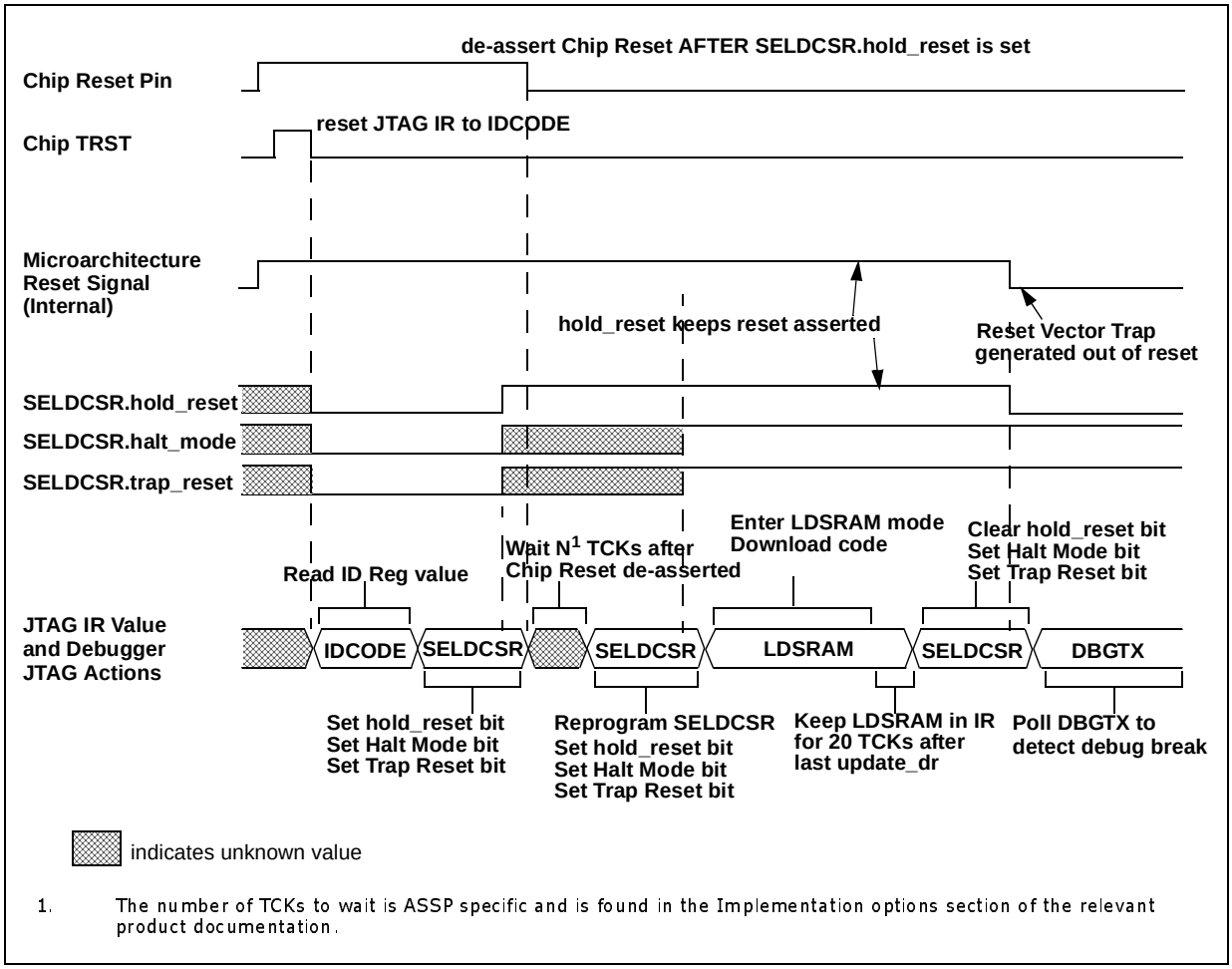
*Note:* When a TRST occurs in the middle of the Load Debug SRAM function, the results of the entire function are unpredictable (in other words, code loaded by the debugger before the TRST is or is not updated in the Debug SRAM).

## 12.7.4 Loading Debug SRAM During Reset

Code is downloaded into the Debug SRAM through JTAG during a processor reset. This feature is used during software debug to download the debug handler prior to starting a debug session. Immediately out of reset, the debugger intercepts the reset vector and take control of the system. The debugger then initializes the system as necessary and begin the application program.

Following a cold reset, the contents of the Debug SRAM are unpredictable. The contents of the Debug SRAM are unaffected by a warm reset. The steps for loading during a cold reset vs. a warm reset are the same, as shown in Figure 24.

**Figure 24. Code Download During a Cold Reset For Debug**




1. The number of TCKs to wait is ASSP specific and is found in the Implementation options section of the relevant product documentation.

Table 130 describe the actions a debugger takes to load code into the Debug SRAM during reset:

**Table 130. Steps For Loading Debug SRAM During Reset**

| Step # | Action | Notes |
|---|---|---|
| 1 | Assert Chip Reset and Chip TRST | This resets the JTAG IR to IDCODE and ensures the TAP controller is in a known state. |
| 2 | Read ID Register value | |
| 3 | Program SELDCSR JTAG register:<br>Halt Mode=1<br>Trap Reset=1<br>hold_reset=1 | SELDCSR details are found in Section 12.5.1.<br>Depending on ASSP implementation, the Halt Mode bit and Trap Reset bit is or is not actually be set to the programmed value. The hold reset bit is set to the programmed value. |
| 4 | De-assert Chip Reset | Internally the microarchitecture remains held in reset due to hold_reset being set. |
| 5 | Wait N TCKs | N is a ASSP specific number and is found in the Implementation options section of the relevant product documentation. This delay ensures that the microarchitecture is stable before proceeding. |
| 6 | Program SELDCSR JTAG register:<br>Halt Mode=1<br>Trap Reset=1<br>hold_reset=1 | The SELDCSR instruction must be reloaded into the JTAG IR. Failure to reload the JTAG IR results in unpredictable behavior.<br>Reprogramming of the SELDCSR JTAG register guarantees that the Halt Mode bit and Trap Reset bit are set before loading the Debug SRAM. |
| 7 | Load LDSRAM JTAG instruction and download the debug handler into Debug SRAM. | Loading into the Debug SRAM is described in Section 12.7.3, "LDSRAM Functions" on page 211 |
| 8 | Clock a minimum of 20 TCKs before changing the JTAG IR. | The LDSRAM JTAG instruction must remain in the JTAG instruction register for at least 20 TCKs following the update_dr for the last line of code. This ensures that the last line is correctly loaded into the Debug SRAM. Changing the JTAG IR within 20 cycles my result in unpredictable behavior. |
| 9 | Program SELDCSR JTAG register:<br>Halt Mode bit = 1<br>Trap Reset bit = 1<br>hold_reset = 0 | Clearing the hold_reset bit allows the microarchitecture to come out of reset and begin execution from address 0. |
| 10 | poll the DBGTX register | Immediately out of reset, a reset vector trap occurs and the debug handler begins execution. The debugger must poll DBGTX for a message from the debug handler to identify when this has happened. |

### 12.7.5 Loading Debug SRAM After Reset

3rd generation microarchitecture provides support to allow a debugger to load code into the Debug SRAM while the microarchitecture is not held in reset. This is referred to as loading the Debug SRAM "on the fly". A debugger loads the Debug SRAM on the fly when downloading dynamic debug handler functions or when downloading the full debug handler prior to initiating a hot-debug session.

Due to the limited size of the Debug SRAM, the main code of the debug handler is limited to the more frequently used functions. Functions which are used less frequently are downloaded, as needed, into any space available in the Debug SRAM. Debug handler functions which are downloaded on the fly are referred to as dynamic functions. Correctly downloading dynamic functions requires software synchronization between the debugger and debug handler. This is described in Section 12.7.5.1.

For hot-debug, a debugger downloads the debug handler into the Debug SRAM while the application program is still running. Strict hardware synchronization between the debugger and 3rd generation microarchitecture ensures that the debug handler code is correctly downloaded. This hardware synchronization is described in Section 12.7.5.2.

#### 12.7.5.1 Software Synchronization for Loading Debug SRAM

Software synchronization for loading the Debug SRAM is only used when there is very tight coupling between the debugger and the code running on 3rd generation microarchitecture. This is true, in particular, when the debug handler is executing, and dynamic functions need to be downloaded. The protocol between the debugger and debug handler is tightly controlled allowing software synchronization to work.

The software synchronization between the debugger and debug handler ensures that

- the debug handler is not executing from the address in the Debug SRAM which the debugger is downloading to;
- the debug handler is not doing an operation which interferes with download.

The software synchronization is accomplished by handshaking through the TX and RX registers.

The debug handler and debugger synchronize the start of the download through the TX register. The debug handler writes a value to the debugger via TX as an indication that the handler is ready for the download.

The debug handler and the debugger synchronize completion of the download using the RX register. While the download is in progress, the debug handler is in a polling loop waiting for a response from the debugger in RX. Once the debugger completes the download, it writes a value to the RX register through JTAG, allowing the debug handler to exit the polling loop.

As an example, the debug handler sends a "ready-for-next-command" message to the debugger, through TX. The handler then enters its command loop, polling RX for the next command from the debugger. In the meantime, the debugger downloads to some other part of the Debug SRAM. After completing the download, the debugger sends a command to the handler via RX, to execute the downloaded function. Upon seeing the command in RX, the handler exits it polling loop and executes the specified command.

### 12.7.5.2 Hardware Synchronization for Loading Debug SRAM

Hardware synchronization mainly applies when a download into the Debug SRAM is required, but the debugger cannot be closely coupled with the code executing on the microarchitecture. This is true for hot-debug, in which the debugger tries to download a debug handler (and start a debug session) while some unknown application code is executing. Since the debugger does not have any control of the application, it must rely on hardware synchronization to ensure that the debug handler is correctly downloaded.

The following steps are required by the debugger prior to loading code into the Debug SRAM for hot-debug:

1. First the debugger issues a "download request" function through JTAG.
2. Then the debugger polls the microarchitecture_ready flag (LDSRAM_SR1[0]), to determine when the microarchitecture is ready for the download. Reading a '1' in this bit indicates that the microarchitecture is ready.
3. Once the microarchitecture is ready the debugger proceeds to download code into the Debug SRAM.
4. After the debugger completes the download, it sends the "download complete" function through JTAG.

Following the download of the debug handler for a hot-debug session, the debugger places 3rd generation microarchitecture in Halt Mode and program a JTAG debug break when it is ready to stop the microarchitecture.

## 12.8 JTAG Device Identification Register

3rd generation microarchitecture provides a 32-bit Device Identification register containing the manufacturer identification code, part number code, and version code. The Device Identification register is selected by placing the IDCODE JTAG instruction in the JTAG IR. When the TAP controller enters the Test_Logic_Reset state, the IDCODE JTAG instruction is automatically loaded into the JTAG IR.

Table 131 shows the Device Identification register format and values of the fields which are standard for all 3rd generation microarchitecture-based ASSPs. The Product Version and Model fields are ASSP specific. This information is found in the 3rd generation microarchitecture implementation options section of the relevant product documentation.

**Table 131. JTAG Device Identification Register**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Version | | | | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | Model | | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |

| Bits | Access | Description |
|---|---|---|
| 31:28 | Read / Write-Ignored | Product Version<br>This field reflects the product revision/stepping. |
| 27:17 | Read / Write-Ignored | 0b 1110 0110 010 |
| 16:12 | Read / Write-Ignored | Model<br>This field specifies a unique product ID. |
| 11:0 | Read / Write-Ignored | 0b 0000 0001 0011 |

## 12.9 Debug Changes from previous generations to 3rd Generation Microarchitecture

Following is a list of changes to the SW debug capabilities between 3rd generation microarchitecture and previous generations. Refer to the *3rd Generation Intel XScale® Microarchitecture Software Debug Guide* for additional information on these changes.

- Microarchitecture debug capabilities
  - JTAG debug break, not qualified with debug enable in 3rd generation microarchitecture
  - SDS definition changed (instruction MMU not turned off by SDS, but turned off by execution from Debug SRAM. Mainly allows instruction fetches following an SOC break to be remapped in Monitor Mode.
- JTAG communications
  - no changes
- Trace Buffer
  - Thumb Trace capability added for 3rd generation microarchitecture; new bit in DCSR added, to enable this feature.
- Loading Debug SRAM
  - name changed from previous generations (LDIC) to 3rd generation microarchitecture (LDSRAM)
  - some previous generations LDIC JTAG commands removed (defined as NOPs)
  - new 3rd generation microarchitecture LDIC JTAG commands added for hot-debug and loading dynamic functions.
  - load Mini-IC command on previous generations mapped to load SRAM command on 3rd generation microarchitecture (same syntax)
  - load Main-IC command removed.
- JTAG Device ID value



# 13.0 Performance Considerations

This chapter describes relevant performance considerations that compiler writers, application programmers and system designers need to optimize code that efficiently uses the 3rd generation Intel XScale® microarchitecture (3rd generation microarchitecture or 3rd generation). Performance numbers discussed here include interrupt latency, branch prediction, and instruction latencies.

## 13.1 Interrupt Latency

Refer to the 3rd generation microarchitecture implementation option section of the relevant product documentation for information on interrupt latency.

*Minimum Interrupt Latency* is defined as the minimum number of cycles from the assertion of an interrupt signal (IRQ or FIQ) to the issue clock of the instruction at the vector for that interrupt. The point at which the assertion begins depends on the interrupt controller implementation as defined in the relevant product documentation. This number assumes best case conditions exist when the interrupt is asserted (for example, the system is not waiting on the completion of some other operation).

A more useful number to work with is the *Maximum Interrupt Latency*. The *Maximum Interrupt Latency* also depends on the interrupt controller implementation and depends on what else is going on in the system at the time the interrupt is asserted. Some events adversely affect interrupt latency by preventing the microarchitecture from servicing the interrupt:

- execution of multiple issue cycle instructions (**LDM**, **STM**, **MCR**, **MRC**, etc.).
- disabled interrupts (due to faults or software interrupts).
- pipeline stalls (data cache buffers full, performing a page table walk, etc.).
- high microarchitecture to system (bus) clock ratios.

Interrupt latency is reduced by:

- ensuring that the interrupt vector and interrupt service routine are resident in the instruction cache. This is accomplished by locking these down into the cache.
- removing or reducing the occurrences of hardware page table walks. This also is accomplished by locking down the application's page table entries into the TLBs, along with the page table entry for the interrupt service routine.

## 13.2 Branch Prediction

3rd generation microarchitecture implements dynamic branch prediction (see Chapter 5.0, "Branch Target Buffer") for the **B** and **BL** instructions. **BX**, **BLX** and any instruction that specifies the PC as the destination are predicted as not taken. For example, an **LDR** or a **MOV** that loads or moves directly to the PC is predicted not taken and incur a branch latency penalty.

These instructions -- **B** and **BL** -- enter into the branch target buffer when these are "taken" for the first time (a "taken" branch refers to when the condition code is evaluated to be true). Once in the branch target buffer,
3rd generation microarchitecture dynamically predicts the outcome of these instructions based on previous outcomes. Table 132, "Branch Latency Penalty", shows the branch latency penalty when these instructions are correctly predicted and when these are not. A penalty of zero for correct prediction means that
3rd generation microarchitecture executes the next instruction in the program flow in the cycle following the branch.

**Table 132. Branch Latency Penalty**

| Microarchitecture Clock Cycles | Description |
|---|---|
| + 0 | ***Predicted Correctly.*** The instruction is in the branch target cache and is correctly predicted. |
| +4 | ***Mispredicted***. There are three cases of branch misprediction, all of which incur a 4-cycle branch delay penalty:<br>1. The instruction is in the branch target buffer and is predicted not-taken, but is actually taken.<br>2. The instruction is not in the branch target buffer and is a taken branch.<br>3. The instruction is in the branch target buffer and is predicted taken, but is actually not-taken |

## 13.3 Addressing Modes

Using the various addressing modes for load and store instructions typically does not affect the instruction issue latencies. See Table 141, "Load and Store Instruction Timings" for exceptions. Base register update latencies only apply for load or store, pre-indexed or post-indexed addressing modes.

## 13.4 Instruction Latencies

The latencies for all the instructions are shown in the following sections with respect to their functional groups:

- branch
- data processing
- multiply
- status register access
- load/store
- semaphore
- coprocessor

The following section explains how to read these tables.

### 13.4.1 Performance Terms

- Issue Clock (cycle 0)
  The cycle when an instruction is decoded *and* allowed to proceed to further stages in the execution pipeline (in other words, when the instruction is actually issued).
- Cycle Distance from A to B
  The cycle distance from cycle A to cycle B is (B-A) -- that is, the number of cycles from the start of cycle A to the start of cycle B. Example: the cycle distance from cycle 3 to cycle 4 is one cycle.
- Issue Latency
  The cycle distance from the issue clock of the current instruction to the issue clock of the next instruction. The number of cycles is influenced by cache-misses, data dependency stalls, and resource availability conflicts.
- Minimum Issue Latency (without Branch Misprediction)
  The minimum cycle distance from the issue clock of the current instruction to the issue clock of the next instruction assuming best case conditions (in other words, that the issuing of the next instruction is not stalled due to a data dependency stall; the next instruction is immediately available from the cache or memory interface; the current instruction does not incur a resource availability stall; and when the instruction uses dynamic branch prediction, correct prediction is assumed).
- Minimum Issue Latency (with Branch Misprediction)
  The minimum cycle distance from the issue clock of the current branching instruction to the issue clock of the next instruction. This definition is identical to *Minimum Issue Latency (without Branch Misprediction)* except that the branching instruction has been mispredicted. It is calculated by adding *Minimum Issue Latency (without Branch Misprediction)* to the branch latency penalty number from Table 132, "Branch Latency Penalty", which is four cycles.
- Result Latency
  The cycle distance from the issue clock of the current instruction to the issue clock of the next instruction that uses the result including any data dependency induced stalls. The number of cycles are influenced by cache-misses, data dependency stalls, and resource availability conflicts.
- Minimum Result Latency
  The minimum cycle distance from the issue clock of the current instruction to the issue clock of the instruction that uses the result without incurring a data dependency stall assuming best case conditions (in other words, that the issuing of the next instruction is not stalled due to a data dependency stall; the next instruction is immediately available from the cache or memory interface; and the current instruction does not incur a resource availability stall during execution that is not detected at issue time).
- Minimum Resource Latency
  The minimum cycle distance from the issue clock of the current multiply instruction to the issue clock of the next multiply instruction assuming the second multiply does not incur a data dependency stall and is immediately available from the instruction cache or memory interface.

The code fragment in Example 6 is used for demonstration purposes relating to issue, result and resource latencies.

### Example 6. Latency Example Code

```
UMLAL   r6, r8, r0, r1

ADD     r9, r10, r11

SUB     r2, r8, r9

MOV     r0, r1
```

Example 7, "Latency Example", shows the instruction flow of our example code through the instruction pipeline. Looking at the issue column, the **UMLAL** instruction issues in cycles 0 and 1 with **ADD** issuing in cycle 2. This shows that the Issue Latency for **UMLAL** is two. Also, from the code example, is seen a data dependency on the result placed in R8 by the **UMLAL** instruction and used by the **SUB** instruction. Again, looking at Example 7, the **UMLAL** instruction issues at cycle 0. The results of the **UMLAL** return in cycles 3 and 4 for R6 and R8 respectively. This corresponds to result latencies of 3 for RdLo and 4 for RdHi. Note that the **UMLAL** instruction occupies the MAC from cycle 1 to cycle 3 which creates a MAC resource latency of 3 cycles. Even though the result in R8 appears to be available for the **SUB** in cycle 4 it is not used by the **SUB** until the following cycle causing a pipe stall.

### Example 7. Latency Example

| Cycle | Issue | Executing MACALU | | Results |
|---|---|---|---|---|
| 0 | umlal (1st cycle) | | | |
| 1 | umlal (2nd cycle) | umlal | | |
| 2 | add | umlal | | |
| 3 | sub | umlal | add | R6, R9 |
| 4 | mov (stalled) | -- | sub (stalled) | R8 |
| 5 | mov | -- | sub | R2 |
| 6 | | -- | mov | R0 |

## 13.4.2 Branch Instruction Timings

### Table 133. Branch Instruction Timings (Those predicted by the BTB)

| Mnemonic | Minimum Issue Latency | | Minimum Result Latency (R14) with Branch Taken | |
|---|---|---|---|---|
| | Predicted Correctly | Mispredicted | Predicted Correctly | Mispredicted |
| B | 1 | 5 | N/A | N/A |
| BL | 1 | 5 | 2 | 5 |

### Table 134. Branch Instruction Timings (Those not predicted by the BTB)

| Mnemonic | Minimum Issue Latency[a,b] | |
|---|---|---|
| | Not Taken | Taken[c] |
| BX, BLX | 1 | 5 |
| ADC,ADD,AND,BIC,EOR, MOV,MVN,ORR,RSB,RSC, SBC,SUB with PC as the destination register | Minimum Issue Latency from Table 135 | 4 + (Minimum Issue latency from Table 135) |
| LDR PC, [...] | 2 | 8 |
| LDM Rn, {... PC} | max (3, 1 + N) | max (8, 5 + N) |

a. "N" is the number of registers in the register list $\{R_1, \ldots R_n\}$ including the PC.
b. When the LDR PC, [...] uses RRX in an addressing mode then one extra cycle of latency must be added to the given latency.
c. R14 Minimum Result Latency for BLX is 5 cycles.

## 13.4.3 Data Processing Instruction Timings

### Table 135. Data Processing Instruction Timings

| Mnemonic | <shifter operand> is NOT a Shift/ Rotate by Register | | <shifter operand> is a Shift/ Rotate by Register OR <shifter operand> is RRX | |
|---|---|---|---|---|
| | Minimum Issue Latency | Minimum Result Latency | Minimum Issue Latency | Minimum Result Latency |
| ADC,ADD,AND,BIC, EOR,MOV,MVN,ORR,RSB, RSC,SBC,SUB | 1 | 1[a] | 2 | 2[a] |
| CMN,CMP,TEQ,TST | 1 | 1 | 2 | 2 |

a. When an instruction needs to use the result of the data processing instruction as Rm in a shift by immediate or as Rn in a QDADD or QDSUB, one extra cycle must be added to the given result latency.

## 13.4.4 Multiply Instruction Timings

### Table 136. Multiply Instruction Timings

| Mnemonic | Rs Value (Early Termination) | S-Bit Value | Minimum Issue Latency | Minimum Result Latency[a] | Minimum Resource Latency (Throughput) |
|---|---|---|---|---|---|
| MLA | Rs[31:15] = 0x00000 or Rs[31:15] = 0x1FFFF | 0 | 1 | 2 | 1 |
| | | 1 | 2 | 2 | 2 |
| | all others | 0 | 1 | 3 | 2 |
| | | 1 | 3 | 3 | 3 |
| MUL | Rs[31:15] = 0x00000 or Rs[31:15] = 0x1FFFF | 0 | 1 | 2 | 1 |
| | | 1 | 2 | 2 | 2 |
| | all others | 0 | 1 | 3 | 2 |
| | | 1 | 3 | 3 | 3 |
| SMLAL | Rs[31:15] = 0x00000 or Rs[31:15] = 0x1FFFF | 0 | 2 | RdLo = 2; RdHi = 3 | 2 |
| | | 1 | 3 | RdLo = 3; RdHi = 3 | 3 |
| | all others | 0 | 2 | RdLo = 3; RdHi = 4 | 3 |
| | | 1 | 4 | RdLo = 4; RdHi = 4 | 4 |
| SMLALxy | N/A | N/A | 2 | RdLo = 2; RdHi = 3 | 2 |
| SMLAWy | N/A | N/A | 1 | 3 | 2 |
| SMLAxy | N/A | N/A | 1 | 2 | 1 |
| SMULL | Rs[31:15] = 0x00000 or Rs[31:15] = 0x1FFFF | 0 | 1 | RdLo = 2; RdHi = 3 | 2 |
| | | 1 | 3 | RdLo = 3; RdHi = 3 | 3 |
| | all others | 0 | 1 | RdLo = 3; RdHi = 4 | 3 |
| | | 1 | 4 | RdLo = 4; RdHi = 4 | 4 |
| SMULWy | N/A | N/A | 1 | 3 | 2 |
| SMULxy | N/A | N/A | 1 | 2 | 1 |
| UMLAL | Rs[31:15] = 0x00000 or Rs[31:15] = 0x1FFFF | 0 | 2 | RdLo = 2; RdHi = 3 | 2 |
| | | 1 | 3 | RdLo = 3; RdHi = 3 | 3 |
| | all others | 0 | 2 | RdLo = 3; RdHi = 4 | 3 |
| | | 1 | 4 | RdLo = 4; RdHi = 4 | 4 |
| UMULL | Rs[31:15] = 0x00000 or Rs[31:15] = 0x1FFFF | 0 | 1 | RdLo = 2; RdHi = 3 | 2 |
| | | 1 | 3 | RdLo = 3; RdHi = 3 | 3 |
| | all others | 0 | 1 | RdLo = 3; RdHi = 4 | 3 |
| | | 1 | 4 | RdLo = 4; RdHi = 4 | 4 |

a. When an instruction needs to use the result of the multiply as Rm in a shift by immediate or as Rn in a QDADD or QDSUB, one extra cycle must be added to the given result latency except for RdLo with S-Bit=1.

### Table 137. Multiply Implicit Accumulate Instruction Timings

| Mnemonic | Rs Value (Early Termination) | Minimum Issue Latency | Minimum Result Latency | Minimum Resource Latency (Throughput) |
|---|---|---|---|---|
| MIA | Rs[31:15] = 0x00000 or Rs[31:15] = 0x1FFFF | 1 | 1 | 1 |
| | all others | 1 | 2 | 2 |
| MIAxy | N/A | 1 | 1 | 1 |
| MIAPH | N/A | 1 | 2 | 2 |

### Table 138. Implicit Accumulator Access Instruction Timings

| Mnemonic | Minimum Issue Latency | Minimum Result Latency[a/b] | Minimum Resource Latency (Throughput)[b] |
|---|---|---|---|
| MAR | 1 | 1 | 1 |
| MRA | 1 | RdLo = 2; RdHi = 3 | 2 |

a. When the next instruction needs to use the result of the MRA as Rm in a shift by immediate or as Rn in a QDADD or QDSUB, one extra cycle must be added to the given result latency.
b. When there are two pending MRA's then one extra cycle must be added to the given latency.

## 13.4.5 Saturated Arithmetic Instructions

### Table 139. Saturated Data Processing Instruction Timings

| Mnemonic | Minimum Issue Latency | Minimum Result Latency |
|---|---|---|
| QADD, QSUB | 1 | 2 |
| QDADD, QDSUB | 1 | 2 |

## 13.4.6 Status Register Access Instructions

### Table 140. Status Register Access Instruction Timings

| Mnemonic | Minimum Issue Latency | Minimum Result Latency |
|---|---|---|
| MRS | 2 | 3 |
| MSR | 2 (6 when updating mode bits) | 2 (6 when updating mode bits) |

## 13.4.7 Load/Store Instructions

### Table 141. Load and Store Instruction Timings

| Mnemonic | Minimum Issue Latency[a] | Minimum Result Latency[a] | Minimum Base Writeback Latency[b] |
|---|---|---|---|
| LDR,LDRB, LDRT,LDRBT | 1 | 3 | 3[a] |
| LDRD | 1[c] | 3 for Rd; 4 for Rd+1 | 1 |
| LDRH,LDRSB,LDRSH | 1 | 3 | 3 |
| PLD | 1 | N/A | N/A |
| STR,STRB, STRT,STRBT | 1 | N/A | 1 |
| STRD | 2 | N/A | 2 |
| STRH | 1 | N/A | 1 |

a. When the instruction uses RRX in an addressing mode, one extra cycle must be added to the given latency.
b. When an instruction needs to use the base register as Rm in a shift by immediate or as Rn in a QDADD or QDSUB, one extra cycle must be added to the base writeback latency.
c. When a load, PLD, or CP15 operation immediately follows an LDRD, one extra cycle must be added to the issue latency.

### Table 142. Load and Store Multiple Instruction Timings

| Mnemonic | {..., PC}? | Executed? | Minimum Issue Latency[a] | Minimum Result Latency[a] | Minimum Base Writeback Latency[a,b] |
|---|---|---|---|---|---|
| LDM | Yes | Yes | max(8, 5 + N) | max(8, 5 + N) | max(8, 5 + N) |
| | | No | max(3, 1 + N) | N/A | N/A |
| | No | Yes | max(3, N) | max(2 + n, N) for load of $R_n$ | max(3, N) |
| | | No | max(3, n) | N/A | N/A |
| STM | - | - | max(3, N) | N/A | max(3, N) |

a. "N" is the number of registers in the register list $\{R_1, \ldots R_n\}$. Note that the register ordering is that imposed by hardware and not by any software notation.
b. For LDMDA, LDMIB, STMDA or STMIB with at least three registers in the list, unless it is an LDM with either R13 or the PC in the register list, one additional cycle must be added to the given latency.

## 13.4.8 Semaphore Instructions

### Table 143. Semaphore Instruction Timings

| Mnemonic | Minimum Issue Latency | Minimum Result Latency |
|---|---|---|
| SWP, SWPB | 4 | 4 |

## 13.4.9 Coprocessor Instructions

### Table 144. CP15 Register Access Instruction Timings (Sheet 1 of 2)

| Instruction[a] | Description | Minimum Issue Latency | Minimum Result Latency |
|---|---|---|---|
| MRC p15, 0, Rd, c0, c0, 0 | Main ID | 4 | 4 |
| MRC p15, 1, Rd, c0, c0, 0 | L2 System ID | 4 | 4 |
| MRC p15, 0, Rd, c0, c0, 1 | L1 Cache Type | 4 | 4 |
| MRC p15, 1, Rd, c0, c0, 1 | L2 Cache Type | 4 | 4 |
| MRC p15, 0, Rd, c1, c0, 0 | Control (CTRL) | 4 | 4 |
| MRC p15, 0, Rd, c1, c0, 1 | Auxiliary Control (AUXCTRL) | 4 | 4 |
| MRC p15, 0, Rd, c2, c0, 0 | Translation Table Base (TTBASE) | 4 | 4 |
| MRC p15, 0, Rd, c3, c0, 0 | Domain Access Control (DACR) | 4 | 4 |
| MRC p15, 0, Rd, c5, c0, 0 | Fault Status (FSR) | 4 | 4 |
| MRC p15, 0, Rd, c6, c0, 0 | Fault Address (FAR) | 4 | 4 |
| MRC p15, 0, Rd, c13, c0, 0 | Process ID (PID) | 4 | 4 |
| MRC p15, 0, Rd, c9, c6, 0 | Data Cache Lock | 4 | 4 |
| MRC p15, 0, Rd, c14, c0, 0 | Data Breakpoint (DBR0) | 4 | 4 |
| MRC p15, 0, Rd, c14, c3, 0 | Data Breakpoint (DBR1) | 4 | 4 |
| MRC p15, 0, Rd, c14, c8, 0 | Instruction Breakpoint (IBR0) | 11 | 11 |
| MRC p15, 0, Rd, c14, c9, 0 | Instruction Breakpoint (IBR1) | 11 | 11 |
| MRC p15, 0, Rd, c14, c4, 0 | Data Bkpt. Control (DBCON) | 4 | 4 |
| MRC p15, 0, Rd, c15, c1, 0 | Coprocessor Access (CPAR) | 4 | 4 |
| MCR p15, 0, Rd, c1, c0, 0 | Control | 12 | N/A |
| MCR p15, 0, Rd, c1, c0, 1 | Auxiliary Control (AUX) | 4 | N/A |
| MCR p15, 0, Rd, c2, c0, 0 | Translation Table Base (TTBR) | 4 | N/A |
| MCR p15, 0, Rd, c3, c0, 0 | Domain Access Control (DACR) | 11 | N/A |
| MCR p15, 0, Rd, c5, c0, 0 | Fault Status (FSR) | 4 | N/A |
| MCR p15, 0, Rd, c6, c0, 0 | Fault Address (FAR) | 4 | N/A |
| MCR p15, 0, Rd, c7, c2, 5 | Allocate Line | 2 | N/A |
| MCR p15, 0, Rd, c7, c5, 0 | Invalidate I-Cache & BTB | 12 | N/A |
| MCR p15, 0, Rd, c7, c5, 1 | Invalidate I-Cacheline by MVA | 2 | N/A |
| MCR p15, 0, Rd, c7, c5, 4 | Prefetch Flush (PF) | 12 | N/A |
| MCR p15, 0, Rd, c7, c5, 6 | Invalidate BTB | 12 | N/A |
| MCR p15, 0, Rd, c7, c7, 0 | Invalidate I/D-Cache & BTB | 12 | N/A |
| MCR p15, 0, Rd, c7, c6, 0 | Invalidate D-Cache | 7 | N/A |
| MCR p15, 0, Rd, c7, c6, 1 | Invalidate D-Cacheline by MVA | 2 | N/A |
| MCR p15, 0, Rd, c7, c10, 1 | Clean D-Cacheline by MVA | 2 | N/A |
| MCR p15, 0, Rd, c7, c10, 2 | Clean D-Cacheline by set/way | 3 | N/A |
| MCR p15, 0, Rd, c7, c10, 4 | Data Write Barrier (DWB) | 2 | N/A |
| MCR p15, 0, Rd, c7, c10, 5 | Data Memory Barrier (DMB) | 2 | N/A |
| MCR p15, 0, Rd, c7, c14, 1 | Cln/Inv D-Cacheline by MVA | 2 | N/A |
| MCR p15, 0, Rd, c7, c14, 2 | Cln/Inv D-Cacheline by set/way | 3 | N/A |
| MCR p15, 1, Rd, c7, c7, 1 | Invalidate L2 Cacheline by MVA | 2 | N/A |
| MCR p15, 1, Rd, c7, c11, 1 | Clean L2 Cacheline by MVA | 2 | N/A |
| MCR p15, 1, Rd, c7, c11, 2 | Clean L2 Cacheline by set/way | 2 | N/A |
| MCR p15, 1, Rd, c7, c15, 2 | Cln/Inv L2 Cacheline by set/way | 2 | N/A |
| MCR p15, 0, Rd, c8, c5, 0 | Invalidate I TLB | 12 | N/A |
| MCR p15, 0, Rd, c8, c5, 1 | Invalidate I TLB Entry | 12 | N/A |
| MCR p15, 0, Rd, c8, c6, 0 | Invalidate D TLB | 2 | N/A |
| MCR p15, 0, Rd, c8, c6, 1 | Invalidate D TLB Entry | 2 | N/A |
| MCR p15, 0, Rd, c8, c7, 0 | Invalidate I/D TLB | 12 | N/A |

## Table 144. CP15 Register Access Instruction Timings (Sheet 2 of 2)

| Instruction[a] | Description | Minimum Issue Latency | Minimum Result Latency |
|---|---|---|---|
| MCR p15, 0, Rd, c9, c5, 0 | Fetch & Lock I-Cacheline[b] | 26 | N/A |
| MCR p15, 0, Rd, c9, c5, 1 | Unlock I-Cache | 12 | N/A |
| MCR p15, 0, Rd, c9, c6, 0 | Data Cache Lock | 4 | N/A |
| MCR p15, 0, Rd, c9, c6, 1 | Unlock D-Cache | 7 | N/A |
| MCR p15, 1, Rd, c9, c5, 0 | Fetch & Lock L2 Cacheline | 2 | N/A |
| MCR p15, 1, Rd, c9, c5, 1 | Unlock L2 Cache | 2 | N/A |
| MCR p15, 1, Rd, c9, c5, 2 | Allocate & Lock L2 Cacheline | 2 | N/A |
| MCR p15, 0, Rd, c10, c4, 0 | Translate & Lock I TLB[b] | 15 | N/A |
| MCR p15, 0, Rd, c10, c4, 1 | Unlock I TLB | 12 | N/A |
| MCR p15, 0, Rd, c10, c8, 0 | Translate & Lock D TLB[b] | 19 | N/A |
| MCR p15, 0, Rd, c10, c8, 1 | Unlock D TLB | 2 | N/A |
| MCR p15, 0, Rd, c13, c0, 0 | Process ID (PID) | 12 | N/A |
| MCR p15, 0, Rd, c14, c0, 0 | Data Breakpoint (DBR0) | 4 | N/A |
| MCR p15, 0, Rd, c14, c3, 0 | Data Breakpoint (DBR1) | 4 | N/A |
| MCR p15, 0, Rd, c14, c4, 0 | Data Bkpt. Control (DBCON) | 4 | N/A |
| MCR p15, 0, Rd, c14, c8, 0 | Instruction Breakpoint (IBR0) | 15 | N/A |
| MCR p15, 0, Rd, c14, c9, 0 | Instruction Breakpoint (IBR1) | 15 | N/A |
| MCR p15, 0, Rd, c15, c1, 0 | Coprocessor Access (CPAR) | 8 | N/A |

a. MRC or MCR with Rd = R15 is unpredictable
b. The latency given assumes the unified L2 cache is hit

## Table 145. CP14 Register Access Instruction Timings

| Mnemonic | Description | Minimum Issue Latency[a] | Minimum Result Latency[a] |
|---|---|---|---|
| MRC p14, 0, Rd, c6, c0, 0 | Clock Config (CCLKCFG) | 12 | 12 |
| MRC p14, 0, Rd, c7, c0, 0 | Power Mode (PWRMODE) | 12 | 12 |
| MRC p14, 0, Rd, c9, c0, 0 | Receive Register (RX) | 12 | 12 |
| MRC p14, 0, Rd, c10, c0, 0 | Debug Control / Status (DCSR) | 12 | 12 |
| MRC p14, 0, Rd, c11, c0, 0 | Trace Buffer (TBREG) | 12 | 12 |
| MRC p14, 0, Rd, c12, c0, 0 | Checkpoint (CHKPT0) | 12 | 12 |
| MRC p14, 0, Rd, c13, c0, 0 | Checkpoint (CHKPT1) | 12 | 12 |
| MRC p14, 0, Rd, c14, c0, 0 | TX/RX Control (TXRXCTRL) | 12 | 12 |
| MRC p14, 0, Rd, c0, c1, 0 | PMU Control (PMNC) | 12 | 12 |
| MRC p14, 0, Rd, c1, c1, 0 | Clock Counter (CCNT) | 12 | 12 |
| MRC p14, 0, Rd, c4, c1, 0 | Interrupt Enable (INTEN) | 12 | 12 |
| MRC p14, 0, Rd, c5, c1, 0 | Overflow Flag Status (FLAG) | 12 | 12 |
| MRC p14, 0, Rd, c8, c1, 0 | Event Select (EVTSEL) | 12 | 12 |
| MRC p14, 0, Rd, c0, c2, 0 | Performance Counter (PMN0) | 12 | 12 |
| MRC p14, 0, Rd, c1, c2, 0 | Performance Counter (PMN1) | 12 | 12 |
| MRC p14, 0, Rd, c2, c2, 0 | Performance Counter (PMN2) | 12 | 12 |
| MRC p14, 0, Rd, c3, c2, 0 | Performance Counter (PMN3) | 12 | 12 |
| MCR p14, 0, Rd, c6, c0, 0 | Clock Config (CCLKCFG) | 12 | N/A |
| MCR p14, 0, Rd, c7, c0, 0 | Power Mode (PWRMODE) | 18 | N/A |
| MCR p14, 0, Rd, c8, c0, 0 | Transmit Register (TX) | 12 | N/A |
| MCR p14, 0, Rd, c10, c0, 0 | Debug Control / Status (DCSR) | 12 | N/A |
| MCR p14, 0, Rd, c12, c0, 0 | Checkpoint (CHKPT0) | 12 | N/A |
| MCR p14, 0, Rd, c13, c0, 0 | Checkpoint (CHKPT1) | 12 | N/A |
| MCR p14, 0, Rd, c14, c0, 0 | TX/RX Control (TXRXCTRL) | 12 | N/A |
| MCR p14, 0, Rd, c0, c1, 0 | PMU Control (PMNC) | 12 | N/A |
| MCR p14, 0, Rd, c1, c1, 0 | Clock Counter (CCNT) | 12 | N/A |
| MCR p14, 0, Rd, c4, c1, 0 | Interrupt Enable (INTEN) | 12 | N/A |
| MCR p14, 0, Rd, c5, c1, 0 | Overflow Flag Status (FLAG) | 12 | N/A |
| MCR p14, 0, Rd, c8, c1, 0 | Event Select (EVTSEL) | 12 | N/A |
| MCR p14, 0, Rd, c0, c2, 0 | Performance Counter (PMN0) | 12 | N/A |
| MCR p14, 0, Rd, c1, c2, 0 | Performance Counter (PMN1) | 12 | N/A |
| MCR p14, 0, Rd, c2, c2, 0 | Performance Counter (PMN2) | 12 | N/A |
| MCR p14, 0, Rd, c3, c2, 0 | Performance Counter (PMN3) | 12 | N/A |
| STC p14, c6, <addr_mode> | Clock Config (CCLKCFG) | 12 | N/A |
| STC p14, c7, <addr_mode> | Power Mode (PWRMODE) | 12 | N/A |
| STC p14, c9, <addr_mode> | Receive Register (RX) | 12 | N/A |
| STC p14, c10, <addr_mode> | Debug Control / Status (DCSR) | 12 | N/A |
| STC p14, c11, <addr_mode> | Trace Buffer (TBREG) | 12 | N/A |
| STC p14, c12, <addr_mode> | Checkpoint (CHKPT0) | 12 | N/A |
| STC p14, c13, <addr_mode> | Checkpoint (CHKPT1) | 12 | N/A |
| STC p14, c14, <addr_mode> | TX/RX Control (TXRXCTRL) | 12 | N/A |
| LDC p14, c6, <addr_mode> | Clock Config (CCLKCFG) | 25 | N/A |
| LDC p14, c7, <addr_mode> | Power Mode (PWRMODE) | 25 | N/A |
| LDC p14, c8, <addr_mode> | Transmit Register (TX) | 15 | N/A |
| LDC p14, c10, <addr_mode> | Debug Control / Status (DCSR) | 15 | N/A |
| LDC p14, c12, <addr_mode> | Checkpoint (CHKPT0) | 15 | N/A |
| LDC p14, c13, <addr_mode> | Checkpoint (CHKPT1) | 15 | N/A |
| LDC p14, c14, <addr_mode> | TX/RX Control (TXRXCTRL) | 15 | N/A |

a. When the MRC destination register is R15 then one additional cycle must be added to the latency given.

### Table 146. CP7 Register Access Instruction Timings

| Mnemonic | Description | Minimum Issue Latency[a] | Minimum Result Latency[a] |
|---|---|---|---|
| MRC p7, 0, Rd, c0, c2, 0 | L2 Cache / BIU Error Log (ERRLOG) | 14 | 14 |
| MRC p7, 0, Rd, c1, c2, 0 | Error Address Lower (ERRADRL) | 14 | 14 |
| MRC p7, 0, Rd, c2, c2, 0 | Error Address Upper (ERRADRU) | 14 | 14 |
| MCR p7, 0, Rd, c0, c2, 0 | L2 Cache / BIU Error Log (ERRLOG) | 14 | N/A |
| MCR p7, 0, Rd, c1, c2, 0 | Error Address Lower (ERRADRL) | 14 | N/A |
| MCR p7, 0, Rd, c2, c2, 0 | Error Address Upper (ERRADRU) | 14 | N/A |

a. When the MRC destination register is R15 then one additional cycle must be added to the latency given.

## 13.4.10 Miscellaneous Instruction Timing

### Table 147. Exception-Generating Instruction Timings

| Mnemonic | Minimum latency to first instruction of exception handler |
|---|---|
| SWI | 7 |
| BKPT | 7 |
| UNDEFINED | 7 |

### Table 148. Count Leading Zeros Instruction Timings

| Mnemonic | Minimum Issue Latency[a] | Minimum Result Latency[a] |
|---|---|---|
| CLZ | 1 | 1 |

a. When the next instruction needs to use the result of the CLZ as Rm in a shift by immediate or as Rn in a QDADD or QDSUB, one extra cycle must be added to the given latency.

## 13.4.11 Thumb Instructions

With the exception of the Thumb BL and BLX(1) instructions, the instructions timings are the same as their equivalent ARM instructions. The mapping of Thumb instructions to ARM instructions is found in the *ARM Architecture Version 5TE Specification*.

### Table 149. Thumb Instruction Timings

| Mnemonic | Minimum Issue Latency | Minimum Result Latency (R14) |
|---|---|---|
| BL, BLX(1) | 2 | 3 |

### 13.4.12 Result Latency Summary

**Figure 25. 3rd Generation Microarchitecture Pipeline Data Flow**

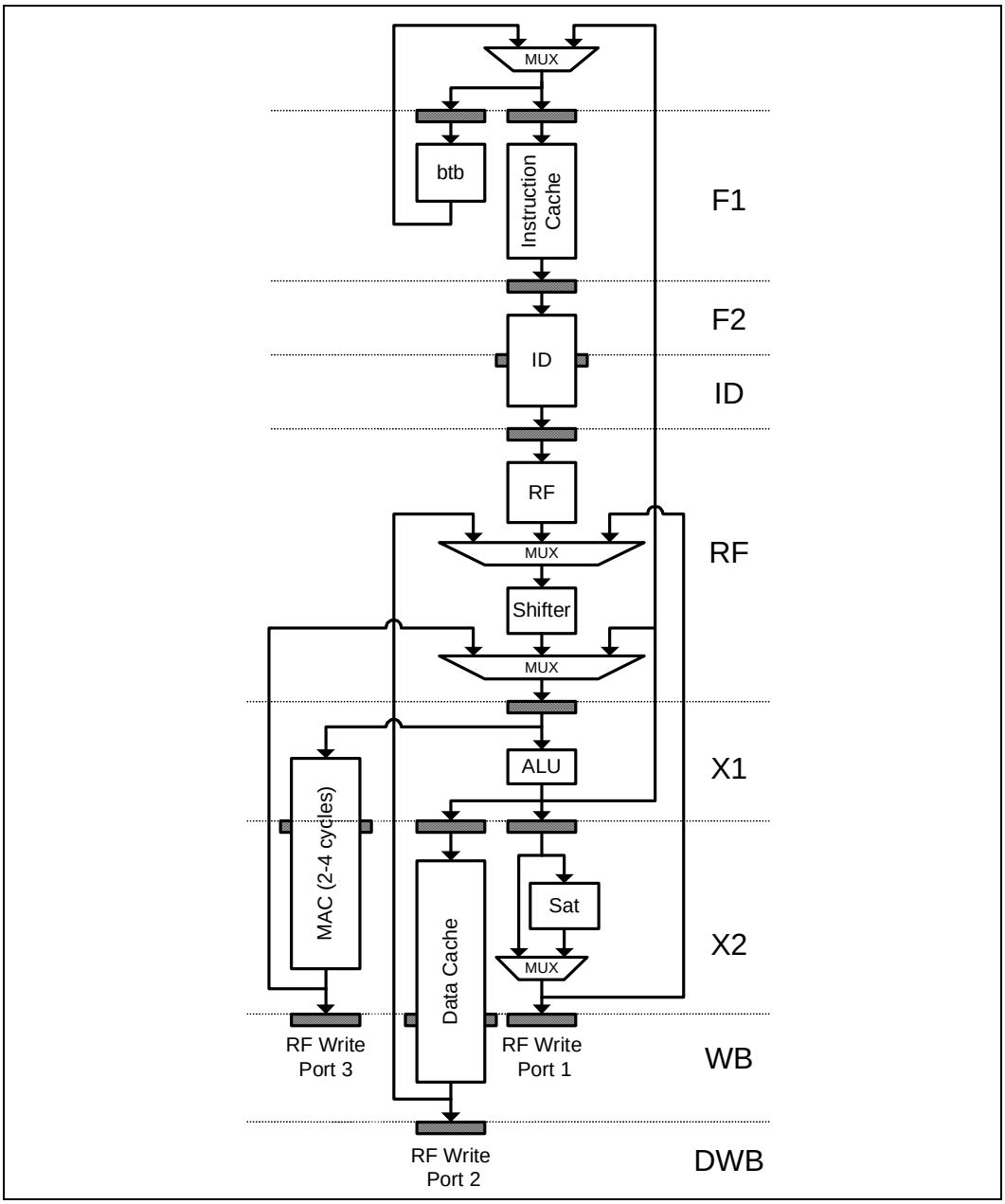



Figure 25, "3rd Generation Microarchitecture Pipeline Data Flow", shows the data flow in the pipeline responsible for the result latencies. ALU and MAC operations are bypassed from the X1 stage and are available to the instruction in the next issue cycle. When a shifted operand is required for an instruction, an extra cycle is required before the data is made available through the shifter. This includes load and store addressing modes that involve shifter operations. Certain instructions always return the data through the shifter path such as loads and the saturated DSP extensions.

## 13.4.13 Shifter Latency Summary

A result dependency is created when an output register from one instruction is used as a source register in a following instruction. The following instruction becomes dependent on the result of the source instruction. The common register causing the dependency is referred to as the dependent register or simply as the dependency.

A shifter stall results when the Cycle Distance from the source instruction to the dependent instruction is less than the Minimum Result latency for the dependency when used as a shifter source operand.

**Table 150. Shifter Dependencies**

| Source Instruction | Dependency |
|---|---|
| ADC,ADD,AND,BIC, EOR,MOV,MVN,ORR, RSB,RSC,SBC,SUB, MLA,MUL,CLZ, SMLAxy,SMLAWy, SMULWy,SMULxy | Rd |
| LDR,LDRB,LDRT,LDRBT LDRD,LDRH,LDRSB,LDRSH STR,STRB,STRT,STRBT STRD,STRH,PLD using pre-indexed or post-indexed addressing modes | Rn |
| UMULL,SMULL,SMLAL,UMLAL, SMLALxy | RdLo, RdHi |

| Dependent Instruction | Dependency |
|---|---|
| ADC, ADD, AND, BIC, EOR, MOV, MVN, ORR, RSB, RSC, SBC, SUB, LDR, LDRB, LDRT, LDRBT LDRD, LDRH, LDRSB, LDRSH STR, STRB, STRT, STRBT STRD, STRH, PLD using shift by immediate | Rm |
| QDADD, QDSUB shift implicit in instruction | Rn |

Table 150, "Shifter Dependencies" shows a list of source and dependent instructions that result in an additional cycle of result latency. The additional cycle of result latency ends in a shifter dependency stall. Instructions are scheduled accordingly. Addressing modes and register mnemonics are as defined in the *ARM Architecture Version 5TE Specification*.



# Appendix A Optimization Guide

## A.1 Introduction

This document contains optimization techniques for achieving the highest performance from 3rd generation Intel XScale® microarchitecture (3rd generation microarchitecture or 3rd generation). It is written for developers who are optimizing compilers or performance analysis tools for this processor. It is also used by application developers to obtain the best performance from their assembly language code.

The instruction set is based on the *ARM Architecture Version 5TE Specification* with some additional instructions. Code generated for the v5TE processor and processors based on the previous generation Intel XScale® microarchitecture, execute on this 3rd generation microarchitecture; however, to obtain the maximum performance of application code, please optimize for 3rd generation microarchitecture.

### A.1.1 Quick Start for Optimization

Techniques to get significant software speed-ups with least amount of work include:

- Scheduling memory operations: "Load and Store Instructions" on page 274
- Enabling hardware optimization features such as the L2 cache and BTB
- Preload data when possible: "Preload Considerations" on page 268
- Avoiding shifter dependencies: "Scheduling Data Processing Instructions" on page 281

Readers with the time and inclination benefit from reading all sections of this document and applying the techniques described therein.

### A.1.2 About This Guide

This guide assumes that the user is familiar with the ARM instruction set and the C language. It consists of the following sections:

- Section A.1, "Introduction". Outlines the contents of this guide.
- Section A.2, "3rd Generation Microarchitecture Pipeline". Provides an overview of pipeline behavior.
- Section A.3, "Basic Optimizations". Outlines basic optimizations that are applied.
- Section A.4, "Cache and preload Optimizations". Contains optimizations for efficient use of caches. Also included are optimizations that take advantage of the preload instructions.
- Section A.5, "Instruction Scheduling". Shows how to optimally schedule code for the pipeline.
- Section A.6, "Optimizing C Libraries". Contains information relating to C library routine optimizations.
- Section A.7, "Optimizations for Size". Contains optimizations that reduce the size of the generated code. Thumb optimizations are also included.

## A.2 3rd Generation Microarchitecture Pipeline

This section provides a brief description of the structure and behavior of 3rd generation microarchitecture pipeline.

### A.2.1 General Pipeline Characteristics

While the processor is scalar and in-order issue, instructions occupies the main pipeline and both sub-pipelines at once (See Figure 26, "Pipeline Diagram" on page 238). Out of order completion is possible. The following sections discuss general pipeline characteristics.

#### A.2.1.1 Number of Pipeline Stages

The processor has a long pipeline (7 stages) which operates at a higher frequency than its predecessors. This allows for greater overall performance. The long pipeline has some drawbacks however:

- Large branch misprediction penalty (four cycles). This is mitigated by dynamic branch prediction.
- Load use delay (LUD). LUDs arise from load-use dependencies. A load-use dependency gives rise to a LUD when the result of the load instruction cannot be made available by the pipeline in time for the dependent instruction. An optimizing compiler finds independent instructions to fill the slot following the load.

### A.2.1.2 Pipeline Organization

The single-issue pipeline consists of a main execution pipeline, MAC pipeline, and a memory access pipeline. These are shown in Figure 26, with the main execution pipeline shaded.

**Figure 26. Pipeline Diagram**

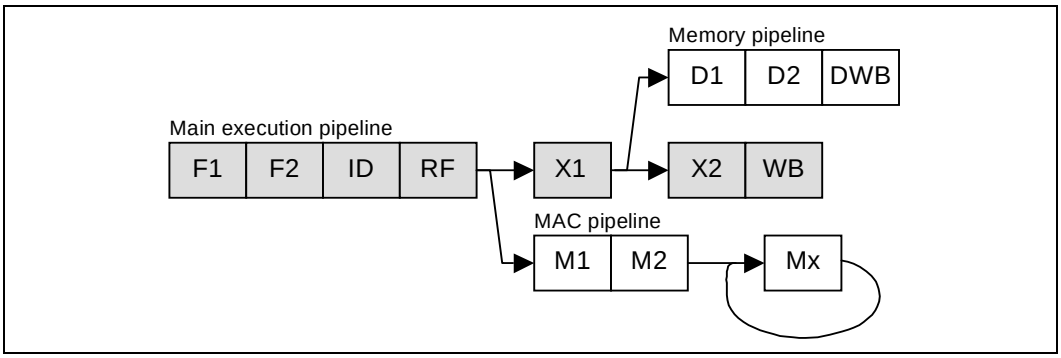



Table 151 gives a brief description of each pipe-stage.

**Table 151. Pipelines and Pipe Stages**

| Pipe / Pipestage | Description | Covered In |
|---|---|---|
| Main Execution Pipeline | Handles data processing instructions | Section A.2.3 |
| F1/F2 | Instruction Fetch | " |
| ID | Instruction Decode | " |
| RF | Register File / Operand Shifter | " |
| X1 | ALU Execute | " |
| X2 | State Execute | " |
| WB | Write-back | " |
| Memory Pipeline | Handles load/store instructions | Section A.2.4 |
| D1/D2 | Data Cache Access | " |
| DWB | Data cache writeback | " |
| MAC Pipeline | Handles all multiply instructions | Section A.2.5 |
| M1-M4 | Multiplier stages | " |
| MWB | MAC write-back (occur during M3-M5) | " |

### A.2.1.3 Out Of Order Completion

The microarchitecture issues instructions in-order, but the main execution pipeline, memory, and MAC pipelines are not lock-stepped and therefore have different execution times. This means that instructions finish out of program order. Short 'younger' instructions are finished earlier than long 'older' ones. (The term 'to finish' is used here to indicate that the operation has been completed and the result has been written back to the register file.)

Programmers need not worry about correctness being affected by out of order completion. The processor preserves effective program order of execution even though instructions complete out of order.

### A.2.1.4 Register Scoreboarding

In certain situations, the pipeline needs to be stalled because of register dependencies between instructions. A register dependency occurs when a previous MAC or load instruction is about to modify a register value that has not been returned to the register file and the current instruction needs access to the same register. When no register dependencies exist, the pipeline need not be stalled. For example, when a load operation has missed the data cache, subsequent instructions that do not depend on the load completes independently.

### A.2.1.5 Use of Bypassing

The pipeline makes extensive use of bypassing to minimize data hazards. Bypassing allows result forwarding from multiple sources reducing the need to stall the pipeline.

### A.2.2 Instruction Flow Through the Pipeline

The 3rd generation microarchitecture pipeline typically issues a single instruction per clock cycle. Instruction execution begins at the F1 pipestage and completes at the WB pipestage.

Although a single instruction is issued per clock cycle, all three sub-pipelines (MAC, memory, and main execution) are processing instructions simultaneously. When there are no data dependencies then each instruction completes independently of the others.

Each pipestage takes a single clock cycle or machine cycle to perform its subtask with the exception of the MAC unit.

#### A.2.2.1 Instruction Execution

Figure 26 uses arrows to show the possible flow of instructions in the pipeline. Instruction execution flows from the F1 pipestage to the RF pipestage. The RF pipestage issues a single instruction to either the X1 pipestage or the MAC unit (multiply instructions go to the MAC while all others continue to X1). This means that M1 or X1 are idle.

All load/store instructions are routed to the memory pipeline after the effective addresses have been calculated in X1.

Indirect branches, mispredicted direct branches, and exceptions cause the F1, F2, ID, RF, and X1 stages of the pipeline to be flushed.

When the processor is in Thumb mode then the ID pipestage dynamically expands each Thumb instruction into a normal ARM instruction, and execution continues as usual.

#### A.2.2.2 Pipeline Stalls

The progress of an instruction stalls anywhere in the pipeline. Several pipestages stalls for various reasons. It is important to understand when and how hazards occur in the pipeline. Performance degradation is significant when care is not taken to minimize pipeline stalls.

## A.2.3 Main Execution Pipeline

### A.2.3.1 F1 / F2 (Instruction Fetch) Pipestages

The job of the instruction fetch stages F1 and F2 is to present the next instruction to be executed to the ID stage. Several important functional units reside within the F1 and F2 stages including:

- Branch Target Buffer (BTB)
- Instruction Fetch Unit (IFU)

An understanding of the BTB (See Chapter 5.0, "Branch Target Buffer") and IFU are important for performance considerations. A summary of operation is provided here so that the reader understands its role in the F1 pipestage.

- Branch Target Buffer (BTB)
  The BTB predicts the outcome of branch type instructions. Once a branch type instruction reaches the X1 pipestage, its target address is known. When this address is different from the address that the BTB predicted the pipeline is flushed, execution starts at the new target address, and the branch's history is updated in the BTB.
- Instruction Fetch Unit (IFU)
  The IFU is responsible for delivering instructions to the *instruction decode* (ID) pipestage. One instruction word is delivered each cycle (when possible) to the ID. The instruction comes from one of two sources: instruction cache or fetch buffers.

### A.2.3.2 ID (Instruction Decode) Pipestage

The ID pipestage accepts an instruction word from the IFU and sends register decode information to the RF pipestage. The ID is able to accept a new instruction word from the IFU on every clock cycle in which there is no stall. The ID pipestage is responsible for:

- General instruction decoding (extracting the opcode, operand addresses, destination addresses, and the offset).
- Detecting undefined instructions and generating an exception.
- Dynamic expansion of complex instructions into sequence of simple instructions. Complex instructions are defined as ones that take more than one clock cycle to issue, such as **LDM**, **STM**, and **SWP**.

### A.2.3.3 RF (Register File / Shifter) Pipestage

The main function of the RF pipestage is to read and write to the *register file unit*, or *RFU*. It provides source data for:

- ALU operations
- Multiply operations
- Memory writes
- Coprocessor operations

The ID unit decodes the instruction and specifies which registers are accessed in the RFU. Based upon this information, the RFU determines when it needs to stall the pipeline due to a register dependency. A register dependency occurs when a previous instruction is about to modify a register value that has not been returned to the RFU and the current instruction needs to access that same register. When no dependencies exist, the RFU selects the appropriate data from the register file and pass it to the next pipestage. When a register dependency does exist, the RFU keeps track of which register is unavailable and when the result is returned the RFU stops stalling the pipe.

The ARM architecture specifies that one of the operands for data processing instructions as the shifter operand, where a 32-bit shift is performed before it, is used as an input to the ALU. This shifter is located in the second half of the RF pipestage.

### A.2.3.4 X1 (Execute) Pipestage

The X1 pipestage performs the following functions:

- ALU calculation - the ALU performs arithmetic and logic operations as required for data processing instructions and load/store index calculations.
- Determine conditional instruction execution - The instruction condition is compared to the CPSR prior to execution of each instruction. Any instruction with a false condition is cancelled, and does not cause any architectural state changes including modifications of registers, memory, and PSR.
- Branch target determination - When a branch was mispredicted by the BTB, the X1 pipestage flushes all of the instructions in the previous pipestages and sends the branch target address to the BTB, which restarts the pipeline

### A.2.3.5 X2 (Execute 2) Pipestage

The X2 pipestage contains the *program status registers* (PSRs). This pipestage selects what is going to be written to the RFU in the WB cycle: PSRs (MRS instruction), ALU output, or other items.

### A.2.3.6 WB (write-back)

When an instruction has reached the write-back stage, it is considered complete. Changes are written to the RFU.

### A.2.4 Memory Pipeline

The memory pipeline consists of two stages: D1 and D2. The *data cache unit*, or DCU, consists of the data-cache array and buffers. The memory pipeline handles load / store instructions.

#### A.2.4.1 D1 and D2 Pipestage

Operation begins in D1 after the X1 pipestage has calculated the effective address for load/stores. The data cache returns the destination data in the D2 pipestage. Before data is returned in the D2 pipestage sign extension and byte alignment occurs for byte and half-word loads.

### A.2.5 Multiply/Multiply Accumulate (MAC) Pipeline

The *multiply-accumulate unit*, or MAC, executes the multiply and multiply-accumulate instructions supported by Intel XScale® microarchitecture. The MAC implements the 40-bit accumulator register acc0, and handles the instructions which transfer its value to and from general-purpose ARM registers.

The following are important characteristics about the MAC:

- The MAC is not truly pipelined, as the processing of a single instruction requires use of the same datapath resources for several cycles before a new instruction is accepted. The type of instruction and source arguments determines the number of cycles required.
- No more than two instructions occupy the MAC pipeline concurrently.
- When the MAC is processing an instruction, another instruction does not enter M1 unless the original instruction completes in the next cycle.
- The MAC unit operates on 16-bit packed signed data. This reduces register pressure and memory traffic size. Two 16-bit data items are loaded into a register with one LDR.
- The MAC achieves throughput of one multiply per cycle when performing a 16- by 32-bit multiply.

#### A.2.5.1 Behavioral Description

The execution of the MAC unit starts at the beginning of the M1 pipestage where it receives two 32-bit source operands. Results are completed N cycles later (where N is dependent on the operand size) and returned to the register file. For more information on MAC instruction latencies refer to Section 13.4, "Instruction Latencies".

An instruction that occupies the M1 pipestage also occupies the X1 pipestage. Each cycle, a MAC operation progresses from M1 to M4. A MAC operation completes anywhere from M2 to M4. When a MAC operation enters M3, it is considered committed because it modifies architectural state regardless of subsequent events.



## A.3 Basic Optimizations

This section outlines optimizations specific to the ARM architecture. These optimizations have been modified to suit the 3rd generation microarchitecture where needed.

### A.3.1 Conditional Instructions

The processor provides the ability to execute instructions conditionally. This feature combined with the ability of instructions to modify the condition codes makes possible a wide array of optimizations.

### A.3.1.1 Optimizing Condition Checks

Some instructions modify the condition codes state. When generating if-else code and loop conditions, it is often beneficial to make use of this feature to set condition codes, thereby eliminating the need for a subsequent compare instruction. Consider the C code fragment:

```
if (a + b) ...;
```

Code generated for the if condition without using an add instruction to set condition codes is:

```
;Assume r0 contains the value a, and r1 contains the value b

    add   r0, r0, r1

    cmp   r0, #0
```

However, code is optimized as follows making use of the add instruction to set condition codes:

```
;Assume r0 contains the value a, and r1 contains the value b

    adds  r0, r0, r1
```

The instructions that increment or decrement the loop counter are also used to modify the condition codes. This eliminates the need for a subsequent compare instruction. A conditional branch instruction is then used to exit or continue with the next loop iteration.

Consider the following C code segment:

```
for (i = 10; i != 0; i--)

{

    do something;

}
```

The optimized code generated for the above code segment looks like:

```
    mov r3, #10

L6:

.

.

    subs r3, r3, #1

    bne  .L6
```



It is also beneficial to rewrite loops whenever possible so as to make the loop exit conditions check against the value 0. For example, the code generated for the code segment below needs a compare instruction to check for the loop exit condition.

```
for (i = 0; i < 10; i++)

{

    do something;

}
```

When the loop were rewritten as follows, the code generated avoids using the compare instruction to check for the loop exit condition.

```
for (i = 9; i >= 0; i--)

{

    do something;

}
```

### A.3.1.2 Optimizing Branches

Branches decrease application performance by indirectly causing pipeline stalls. Branch prediction improves the performance by lessening the delay inherent in fetching a new instruction stream. The number of branches that accurately predicted is limited by the size of the branch target buffer. Since the total number of branches executed in a program is relatively large compared to the size of the branch target buffer; it is often beneficial to minimize the number of branches in a program. Consider the following C code segment.

```
int foo(int a)

{

    if (a > 10)

        return 0;

    else

        return 1;

}
```

The code generated for the if-else portion of this code segment using branches is:

```
    cmp   r0, #10

    ble   L1

    mov   r0, #0

    b     L2

L1:

    mov   r0, #1

L2:
```

The code generated above takes three cycles to execute the else-part and four cycles for the if-part, assuming best case conditions and no branch misprediction penalties. In the case of this generation of the microarchitecture, a branch misprediction incurs a penalty of four cycles. When the branch is mispredicted 50% of the time, and when assumed that both the if-part and the else-part are equally likely to be taken, on an average the code above takes 5.5 cycles to execute.

$$\left(\frac{50}{100} \times 4 + \frac{3+4}{2}\right) = 5.5 \quad \text{cycles}.$$

When using 3rd generation microarchitecture to execute instructions conditionally, the code generated for the above if-else statement is:

```
    cmp   r0, #10

    movgt r0, #0

    movle r0, #1
```

The above code segment does not incur any branch misprediction penalties and takes three cycles to execute assuming best case conditions. As is seen, using conditional instructions speeds up execution significantly. However, the use of conditional instructions are carefully considered to ensure that it does improve performance. To decide when to use conditional instructions over branches consider the following hypothetical code segment:

```
if (cond)

    if_stmt

else

    else_stmt
```

Assume that having the following data:

$N1_B$ Number of cycles to execute the if_stmt assuming the use of branch instructions

$N2_B$ Number of cycles to execute the else_stmt assuming the use of branch instructions

P1 Percentage of times the if_stmt is likely to be executed

P2 Percentage of times to likely incur a branch misprediction penalty

$N1_C$ Number of cycles to execute the if-else portion using conditional instructions assuming the if-condition to be true

$N2_C$ Number of cycles to execute the if-else portion using conditional instructions assuming the if-condition to be false

Once the above data is had, use conditional instructions when:

$$\left(N1_C \times \frac{P1}{100}\right) + \left(N2_C \times \frac{100-P1}{100}\right) \leq \left(N1_B \times \frac{P1}{100}\right) + \left(N2_B \times \frac{100-P1}{100}\right) + \left(\frac{P2}{100} \times 4\right)$$

The following example illustrates a situation which is better off using branches over conditional instructions. Consider the code sample shown below:

```
    cmp   r0, #0
    bne   L1
    add   r0, r0, #1
    add   r1, r1, #1
    add   r2, r2, #1
    add   r3, r3, #1
    add   r4, r4, #1
    b     L2
L1:
    sub   r0, r0, #1
    sub   r1, r1, #1
    sub   r2, r2, #1
    sub   r3, r3, #1
    sub   r4, r4, #1
L2:
```

In the above code sample, the cmp instruction takes 1 cycle to execute, the if-part takes 7 cycles to execute, and the else-part takes 6 cycles to execute. When changing the code above so as to eliminate the branch instructions by making use of conditional instructions, the if-else part always takes 10 cycles to complete.

When making the assumptions that both paths are equally likely to be taken and that branches are mis-predicted 50% of the time, the costs of using conditional execution *vs.* using branches is computed as follows:

Cost of using conditional instructions:

$$1 + \left(\frac{50}{100} \times 10\right) + \left(\frac{50}{100} \times 10\right) = 11 \quad \text{cycles}$$

Cost of using branches:

$$1 + \left(\frac{50}{100} \times 7\right) + \left(\frac{50}{100} \times 6\right) + \left(\frac{50}{100} \times 4\right) = 9.5 \quad \text{cycles}$$

As is seen, there is better performance by using branch instructions in the above scenario.

### A.3.1.3 Optimizing Complex Expressions

Conditional instructions are also used to improve the code generated for complex expressions such as the C shortcut evaluation feature. Consider the following C code segment:

```
int foo(int a, int b)

{

    if (a != 0 && b != 0)

        return 0;

    else

        return 1;

}
```

The optimized code for the if condition is:

```
    cmp   r0, #0

    cmpne r1, #0
```

Similarly, the code generated for the following C segment

```
int foo(int a, int b)

{

    if (a != 0 || b != 0)

        return 0;

    else

        return 1;

}

is:

    cmp   r0, #0

    cmpeq r1, #0
```

The use of conditional instructions in the above fashion improves performance by minimizing the number of branches thereby minimizing the penalties caused by branch mispredictions. This approach also reduces the utilization of branch prediction resources.



### A.3.2 Bit Field Manipulation

Shift and logical operations provide a useful way of manipulating bit fields. Bit field operations are optimized as follows:

```
;Set the bit number specified by r1 in register r0

    mov   r2, #1

    orr   r0, r0, r2, asl r1

;Clear the bit number specified by r1 in register r0

    mov   r2, #1

    bic   r0, r0, r2, asl r1

;Extract the bit-value of the bit number specified by r1 of the

;value in r0 storing the value in r0

    mov   r1, r0, asr r1

    and   r0, r1, #1

;Extract the higher order 8 bits of the value in r0 storing

;the result in r1

    mov   r1, r0, lsr #24
```

### A.3.3 Optimizing the Use of Immediate Values

**MOV** or **MVN** instructions are used when loading an immediate (constant) value into a register. Please refer to the *ARM Architecture Version 5TE Specification* for the set of immediate values that are used in a **MOV** or **MVN** instruction. It is also possible to generate a whole set of constant values using a combination of **MOV**, **MVN**, **ORR**, **BIC**, **ADD**, and related instructions.

A **LDR** instruction is used to load a constant from memory; this is not always the highest performance method of creating an immediate value. The **LDR** instruction has the potential of incurring a cache miss in addition to polluting the data and instruction caches. Programmers thus avoid using a **LDR** instruction to load a constant when a sequence of one or two data-processing instructions are instead used.

The code samples below illustrate cases where a combination of the above instructions are used to set a register to a constant value:

```
;Set the value of r0 to 127

    mov   r0, #127

;Set the value of r0 to 0xfffffefb.

    mvn   r0, #260

;Set the value of r0 to 257

    mov   r0, #1

    orr   r0, r0, #256

;Set the value of r0 to 0x51f

    mov   r0, #0x1f

    orr   r0, r0, #0x500

;Set the value of r0 to 0xf100ffff

    mvn   r0, #0xff, 16

    bic   r0, r0, #0xe, 8

; Set the value of r0 to 0x12341234

    mov   r0, #0x8d, 30

    orr   r0, r0, #0x1, 20

    add   r0, r0, r0, LSL #16 ; shifter delay of 1 cycle
```

*Note:* It is possible to load any 32-bit value into a register using a sequence of at most four instructions.

### A.3.4 Optimizing Integer Multiply and Divide

Multiplication by an integer constant is optimized to make use of the shift operation whenever possible.

```
;Multiplication of r0 by 2n

    mov   r0, r0, LSL #n

;Multiplication of R0 by 2^n+1

    add   r0, r0, r0, LSL #n
```

Multiplication by an integer constant that is expressed as $(2^n + 1) \cdot (2^m)$ is similarly optimized as:

```
;Multiplication of r0 by an integer that is

;expressed as (2^n+1)*(2^m)

    add   r0, r0, r0, LSL #n

    mov   r0, r0, LSL #m
```

Please note that the above optimization is only used in cases where the multiply operation cannot be advanced far enough to prevent pipeline stalls.

Dividing an unsigned integer by an integer constant is optimized to make use of the shift operation whenever possible.

```
;Dividing r0 containing an unassigned value by an integer constant

;that is represented as 2^n

    mov   r0, r0, LSR #n
```

Dividing a signed integer by an integer constant is optimized to make use of the shift operation whenever possible.

```
;Dividing r0 containing a signed value by an integer constant

;that is represented as 2^n

    mov   r1, r0, ASR #31

    add   r0, r0, r1, LSR #(32 - n)

    mov   r0, r0, ASR #n
```

The **ADD** instruction stalls for 1 cycle. The stall is prevented by filling in another instruction before the **ADD**.

### A.3.5 Effective Use of Addressing Modes

The processor provides a variety of addressing modes that make indexing an array of objects highly efficient. For a detailed description of these addressing modes please refer to the *ARM Architecture Version 5TE Specification.* The following code samples illustrate how various kinds of array operations are optimized to make use of these addressing modes:

```
;Set the contents of the word pointed to by r0 to the value

;contained in r1 and make r0 point to the next word

    str   r1, [r0], #4

;Increment the contents of r0 to make it point to the next word

;and set the contents of the word pointed to the value contained

;in r1

    str   r1, [r0, #4]!

;Set the contents of the word pointed to by r0 to the value

;contained in r1 and make r0 point to the previous word

    str   r1, [r0], #-4

;Decrement the contents of r0 to make it point to the previous

;word and set the contents of the word pointed to the value

;contained in r1

    str   r1, [r0, #-4]!
```



## A.4 Cache and preload Optimizations

The caches are limited resources and need to be effectively managed to obtain optimum application performance. This section considers how to use the various cache memories in all their modes and examines when and how to use preload to improve execution efficiencies.

### A.4.1 L1 Instruction Cache

The Intel XScale® microarchitecture has separate L1 instruction and L1 data caches. Only fetched instructions are held in the instruction cache even though both data and instructions reside within the same memory space with each other. Functionally, the instruction cache is either enabled or disabled. There is no performance benefit of not using the instruction cache.

#### A.4.1.1 Cache Miss Cost

Performance is highly dependent on reducing the cache miss rate. Refer to the implementation options section of the relevant product documentation for more information on the cycle penalty associated with cache misses. Note that this cycle penalty becomes significant when the processor is running much faster than external memory. This penalty is mitigated by use of the unified L2 cache. Executing non-cached instructions severely curtails the processor performance in this case and it is very important to do everything possible to minimize cache misses.

#### A.4.1.2 Pseudo-LRU Replacement Cache Policy

Both the L1 instruction and L1 data caches use a pseudo-LRU replacement policy to evict a cache line. The simple consequence of this is that at sometime every line is evicted assuming a non-trivial program. The less obvious consequence is that predicting when and over which cache lines evictions take place is difficult to predict. This information must be gained by experimentation using performance profiling.

#### A.4.1.3 Code Placement to Reduce Instruction Cache Misses

Code placement greatly affects cache misses.

One way to view the L1 instruction cache is as a collection of 256 sets, each of which is fixed at an address (modulo 8192). Elements of a set are 32-byte lines. For example, set 0 contains instructions at address 0x0..0x1F, or at address 0x2000..0x201F, or at address 0x4000..0x401F, etc.

Each set contains up to four lines at its address. When a fifth line of code is needed that maps into that set, then one of the existing lines must be displaced.

Code that exhibits a high degree of spatial locality relative to any set causes excessive cache line evictions (thrashing the cache). The ideal situation is for the software tools to distribute the code to achieve a low spatial locality over this space.

This is very difficult when not impossible for a compiler to do. Most of the input needed to best estimate how to distribute the code comes from profile based compiler optimizations.

### A.4.1.4 Locking Code into Instruction Cache

One very important instruction cache feature is the ability to lock code into the instruction cache. Once locked into the instruction cache, the code is always available for fast execution. Another reason for locking critical code into cache is that with only four ways per set, the pseudo-LRU replacement policy "age out" the code even when it is a very frequently executed function. Key code components to consider for locking are:

- Interrupt handlers
- Real time clock handlers
- OS critical code
- Time critical application code

The disadvantage to locking code into the cache is that it reduces the cache size for the rest of the program. This results in thrashing the remaining cache. How much code to lock is very application dependent and requires experimentation to optimize.

Code locked into the instruction cache is placed sequentially together as tightly as possible so as not to waste precious cache space. Making the code sequential also ensures even distribution across all cache ways. Though it is possible to choose randomly located functions for cache locking, this approach runs the risk of locking multiple cache ways in one set and few or none in another set. This distribution unevenness leads to excessive thrashing of the instruction cache.



## A.4.2 L1 Data Cache

The microarchitecture allows the user to define memory regions whose cache policies are set by the user (see Section 6.2.3, "Cache Policies"). To support allocating variables to these various memory regions, the tool chain (compiler, assembler, linker, and debugger) must implement named sections.

The performance of the application code depends on what cache policy is being used for data objects. Guidelines on when to use a particular policy are described below.

When the application is running under an OS, then the OS restricts using certain cache policies.

### A.4.2.1 Cache Conflicts, Pollution and Pressure

Cache pollution occurs when unused data is loaded in the cache and cache pressure occurs when data that is not temporal to the current process is loaded into the cache. For an example see Section A.4.5.2, "Preload Loop Scheduling" below.

### A.4.2.2 Write-through and Write-back Cached Memory Regions

Write-through memory regions generate more second level memory traffic, therefore, it is recommended that use of write-through be minimized. This additional traffic is mitigated by use of the unified L2 cache. The write back policy, however, is used whenever possible. When a memory region is marked shareable, the L1 data cache policy for that region is forced to write-through to maintain data coherency.

One reason that system software designates a page as write-through (or uncacheable) is that the target memory needs to be coherent with the contents of the cache. For example, data that is updated by a DMA device is marked as uncacheable so that software running on the microarchitecture always sees the latest updated value in that memory.

On products where I/O-coherency is enabled, consider using that facility to keep the microarchitecture view of memory synchronized with other readers/writers. Keeping coherent with I/O coherency is higher performance than using reduced cacheability. See 3.2.3.2 for details on enabling shared memory. Also, consult your product documentation to see when it enables coherency.

### A.4.2.3 L1 Data Cache Organization

Stride, the way data structures are walked through, affects the temporal quality of the data and reduce or increase cache conflicts. The Intel XScale® microarchitecture data cache has 256 sets of 32 bytes. This means that each cache line in a set is on a modulo 8 KB address boundary. The caution is to choose data structure sizes and stride requirements that do not overwhelm a given set causing conflicts and increased register pressure.

Register pressure is increased because additional registers are required to track preload addresses. The effects are affected by rearranging data structure components to use more parallel access to search and compare elements. Similarly rearranging sections of data structures so that sections often written fit in the same cache line, 32 bytes reduces cache eviction write-backs. On a global scale, techniques such as array merging enhance the spatial locality of the data.

As an example of array merging, consider the following code:

```
int a[NMAX];

int b[NMAX];

int i, ix;

for (i=0; i < NMAX; i++)

{

    ix = b[i];

    if (a[i] != 0)

        ix = a[i];

    do_other calculations;

}
```



In the above code, data is read from both arrays a and b, but a and b are not spatially close. Array merging places a and b spatially close.

```
struct {

    int a;

    int b;

} c[NMAX];

int i, ix;

for (i=0; i < NMAX; i++)

{

    ix = c[i].b;

    if (c[i].a != 0)

        ix = c[i].a;

    do_other_calculations;

}
```

As an example of rearranging often written to sections in a structure, consider the code sample:

```
struct employee {

    struct employee *prev;

    struct employee *next;

    float Year2DatePay;

    float Year2DateTax;

    int ssno;

    int empid;

    float Year2Date401KDed;

    float Year2DateOtherDed;

};
```

In the data structure shown above, the fields Year2DatePay, Year2DateTax, Year2Date401KDed, and Year2DateOtherDed are likely to change with each pay check. The remaining fields however change very rarely. When the fields are laid out as shown above, assuming that the structure is aligned on a 32-byte boundary, modifications to the Year2Date fields is likely to use two memory buffers when the data is written out to memory. However, restrict the number of write buffers that are commonly used to one by rearranging the fields in the above data structure as shown below:

```
struct employee {

    struct employee *prev;

    struct employee *next;

    int ssno;

    int empid;

    float Year2DatePay;

    float Year2DateTax;

    float Year2Date401KDed;

    float Year2DateOtherDed;

};
```

### A.4.2.4 Cache Line Preallocation

The 3rd generation microarchitecture L1 cache only allocates space for new data (a line) when it processes a read transaction. Writes to the cache do not allocate a line. This policy is called *read allocate*.

In some cases, it is known in advance that a large amount of generated data is read back in and processed again. The read allocate cache policy causes data in this situation to be written out, missing the cache, and read back, possibly causing cache line evictions.

The way to reduce bandwidth, in this case, is to preallocate the cache space for data in question. The generated data then hits the cache and is read back without causing a second level memory request. Eventually the data is written out but only one memory request is made per cache line instead of three.

There are several ways to preallocate a line:

- with a read to the line
- with a **PLD** instruction
- with a line-allocate operation (when all bytes in the line are destined to be written)

### A.4.2.5 Creating On-chip RAM

Part of the L1 data cache is converted into fast on chip RAM. Access to objects in the on-chip RAM do not incur cache miss penalties thereby reducing the number of processor stalls. Application performance is improved by converting a part of the cache into on chip RAM and allocating frequently used variables to it. Due to pseudo-LRU replacement policy, all data is eventually evicted. Therefore, to prevent critical or frequently used data from being evicted it is allocated to on-chip RAM.

The following variables are good candidates for allocating to the on-chip RAM:

- Frequently used global data used for storing context for context switching.
- Global variables that are accessed in time critical functions such as interrupt service routines.

The on-chip RAM is created by locking a memory region into the data cache (see Section 6.4, "Data Cache Locking" for more details).

When creating the on-chip RAM, care must be taken to ensure that all sets in the on-chip RAM area of the data cache have approximately the same number of ways locked, otherwise some sets have more ways locked than the others. This uneven allocation increases the level of thrashing in some sets and leave other sets under utilized.

For example, consider three arrays arr1, arr2, and arr3 of size 64 bytes each that are being allocated to the on-chip RAM, and assume that the address of arr1 is 0, address of arr2 is 8192, and the address of arr3 is 16384. All three arrays are within the same sets, in other words, set0 and set1, as a result three ways in both sets set0 and set1 are locked, leaving 1 way for use by other variables.

This is overcome by allocating on-chip RAM data in sequential order. In the above example, allocating arr2 to address 64 and arr3 to address 128 allows the three arrays to use only 1 way in sets 0 through 5.

### A.4.2.6 LLR Cache Policy

The LLR cache policy is best used for large data structures that have a high spatial locality within the data cache. Addressing this type of data from the data cache quickly pollutes much when not all of the data cache. Eviction of valuable data reduces overall performance by requiring constant reloads of the evicted data. Placing this type of data in a LLR cacheable region prevents data cache pollution while providing some of the benefits of cached access.

An example of data that is assigned to LLR cache is a video buffer. Video buffers are usually large and occupies the entire cache. Over use of the LLR cache region causes thrashing within the LLR cache space. This is easy to do because the LLR cache policy only has one way per set. For example, a loop which uses a simple statement such as:

```
for (i=0; I< IMAX; i++)

{

    A[i] = B[i] + C[i];

}
```

where A, B, and C reside in a LLR cache region and each array aligned on a 8192-byte boundary quickly thrashes LLR cache space.

LLR cacheable regions are part of the main data cache and use impacts data cache usage. See Section 6.1.2, "Low-Locality of Reference (LLR)" for more information.



### A.4.2.7 Data Alignment

Cache lines begin on 32-byte address boundaries. To maximize cache line use and minimize cache pollution, data structures are aligned on 32-byte boundaries and sized to multiple cache line sizes. Aligning large data structures on cache address boundaries simplifies later addition of preload instructions to optimize performance.

Not aligning data on cache lines has the disadvantage of moving the preload address correspondingly to the misalignment. Consider the following example:

```
struct {

    long ia;

    long ib;

    long ic;

    long id;

} tdata[IMAX];

for (i=0, i<IMAX; i++)

{

    PRELOAD(tdata[i+1]);

    tdata[i].ia = tdata[i].ib + tdata[i].ic + tdata[i].id;

    ....

    tdata[i].id = 0;

}
```

In this case when tdata[] is not aligned to a cache line then the preload using the address of tdata[i+1].ia is not include element id. When the array was aligned on a cache line + 12 bytes then the preload halves to be placed on &tdata[i+1].id.

When the structure is not sized to a multiple of the cache line size then the preload address must be advanced appropriately and requires extra preload instructions.

Generally, not aligning and sizing data adds extra computational overhead.

Additional preload considerations are discussed in greater detail in following sections.

### A.4.2.8 Literal Pools

The processor does not have a single instruction that moves all literals (a constant or address) to a register. One technique to load registers with literals is by loading the literal from a memory location that has been initialized with the constant or address. These blocks of constants are referred to as literal pools. See Section A.3, "Basic Optimizations" for more information on how to do this. It is advantageous to place all the literals together in a pool of memory known a literal pool. These data blocks are located in the text or code address space so that these are loaded using PC relative addressing. However, references to the literal pool area load the data into the data cache instead of the instruction cache. Therefore, it is possible that the literal is present in both the data and instruction caches resulting in waste of space.

For maximum efficiency, the compiler aligns all literal pools on cache boundaries and size each pool to a multiple of 32 bytes (the size of a cache line). One additional optimization is group highly used literal pool references into the same cache line. The advantage is that once one of the literals has been loaded the other seven are available immediately from the data cache.

### A.4.3 L2 Unified Cache

3rd generation microarchitecture has an optional 256KB or 512KB L2 unified cache (see Chapter 8.0, "Level 2 Unified Cache (L2)"). This cache acts to reduce the latency of memory requests. The L2 cache is physically addressed, and buffers information for instruction, data and TLB requests. The L2 cache operates at half the microarchitecture frequency, and supply entire cache lines to either the L1 instruction cache or L1 data cache.

The L2 is not enabled by default at reset. Make sure that your operating system has enabled the L2 to get better performance.

The L2 cache is used to cache parts of the page table. For higher performance, ensure that your operating system has enabled this option.

The L2 cache supports write-back only caching, and does not support write-through caching. Accesses to L2 cacheable memory marked as write-through are treated as L2 un-cacheable. Supported policies are:

- Non L2 cacheable
- L2 cacheable, write allocate and write-back.

#### A.4.3.1 Locking Code or Data into L2 Unified Cache

One important L2 cache feature is the ability to lock code or data into the L2 cache. Once locked into the L2 cache, the code or data is always available for fast access. Key components to consider for locking are:

- Interrupt handlers
- Real time clock handlers
- OS critical code
- Time critical application code

The disadvantage to locking code or data into the L2 cache is that it reduces the cache size for the rest of the program. The L2 cache is slower but larger than the L1 cache. How much of the L2 cache to lock is very application dependent and requires experimentation to optimize. Code and/or data is placed sequentially together as tightly as possible so as not to waste precious cache space. Making the code and/or data sequential also ensures even distribution across all cache ways. Though it is possible to choose randomly located code or data for cache locking, this approach runs the risk of co-locating multiple cache lines in one set (increasing the cache spatial locality) and few or none in another set. This uneven distribution leads to excessive L2 cache thrashing.

### A.4.3.2 Creating On-chip RAM

Part of the L2 Cache is converted into a fast on chip RAM. Access to objects in the on-chip RAM do not incur large cache miss penalties, thereby reducing the number and duration of processor stalls. Application performance is improved by converting a part of the L2 cache into on chip RAM and allocating frequently used variables to it.

The following variables are good candidates for allocating to the on-chip RAM:

- Audio and video buffers
- Direct memory access (DMA) descriptors
- Global variables that are access in time critical functions such as interrupt service routines.

The on-chip RAM is created by locking a memory region into the L2 cache (see Section 8.3.5, "Level 2 Cache Locking" for more details).

When creating the on-chip RAM, care must be taken to ensure that all sets in the on-chip RAM area of the L2 cache have approximately the same number of ways locked, otherwise some sets have more ways locked than the others. This uneven allocation increases the level of thrashing in those sets and leave other sets under utilized.

### A.4.4 Classical Array Optimizations

Consult standard references for classical optimizations on loop/array code. These include blocking, and loop fusion and interchange.

### A.4.5 Preload Considerations

The processor has a true preload load instruction (PLD). The purpose of this instruction is to preload data into the data cache. Data preloading allows hiding of memory transfer latency while the processor continues to execute instructions. The preload is important to compiler and assembly code because judicious use of the preload instruction enormously improves throughput performance. Data preload is applied not only to loops but also to any data references within a block of code.

The preload instruction loads data into the data cache and not a register. Compilers for processors which have data caches, but do not support preload, sometimes use a load instruction to preload the data cache. This technique has the disadvantages of using a register to load data and requiring additional registers for subsequent preloads and thus increasing register pressure. By contrast, the preload is used to reduce register pressure instead of increasing it.

#### A.4.5.1 Preload Distances

Scheduling the preload instruction requires understanding the system latency times and system resources which affect when to use the preload instruction. Refer to the 3rd generation implementation options section of the relevant product documentation for more information.

#### A.4.5.2 Preload Loop Scheduling

When adding preload to a loop which operates on arrays, it is advantageous to preload ahead one, two, or more iterations. The data for future iterations is located in memory by a fixed offset from the data for the current iteration. This makes it easy to predict where to fetch the data. The number of iterations to preload ahead is referred to as the preload scheduling distance. Refer to the implementation options section of the relevant product documentation for more information.

#### A.4.5.3 Preload Loop Limitations

It is not always advantageous to add preload to a loop. Loop characteristics that limit the use value of preload are discussed below.

#### A.4.5.4 Compute vs. Data Bus Bound

At the extreme, a loop, which is data bus bound, does not benefit from preload because all the system resources to transfer data are quickly allocated and there are no instructions that are profitably executed. On the other end of the scale, compute bound loops allow complete hiding of all data transfer latencies.

#### A.4.5.5 Low Number of Iterations

Loops with very low iteration counts have the advantages of preload completely nullified. A loop with a small fixed number of iterations is faster when the loop is completely unrolled rather than trying to schedule preload instructions.

### A.4.5.6 Bandwidth Limitations

Overuse of preloads usurps resources and degrade performance. This happens because once the bus traffic requests exceed the system resource capacity, the processor stalls. Microarchitecture data transfer resources are:

Twelve memory buffers

Four request buffers per memory buffer

SDRAM resources are typically:

Four memory banks

One page buffer per bank referencing a 4 K address range

Four transfer request buffers

Consider how these resources work together. A memory buffer is allocated for each cache read miss. A subsequent read to the same cache line does not require a new fill buffer but does require a request buffer, and a subsequent write also require a new memory buffer. A memory buffer is also allocated for each read to non-cached memory and a memory buffer is needed for each memory write to non-cached memory that is non-coalescing. Consequently, a **STM** instruction listing eight registers and referencing non-cached memory uses eight memory buffers assuming these do not coalesce and one or two memory buffers when these do coalesce. A cache eviction requires a memory buffer for each dirty cache line. The preload instruction requires a memory buffer for each cache line and zero or one memory buffers for an eviction.

When adding preload instructions, take caution to ensure that the combination of preload and instruction bus requests do not exceed the system resource capacity or performance are degraded instead of improved. The important points are to spread preload operations over calculations so as to allow bus traffic to flow freely and to minimize the number of necessary preloads.

Rules of thumb on when not to use a PLD:

- On very speculative loads
- When doing so is likely to force a table walk for an invalid page (for example, a NULL pointer)
- When the targeted data is probably already resident in the cache



### A.4.5.7 Preload Unrolling

When iterating through a loop, data transfer latency is hidden by preloading ahead one or more iterations. The solution incurs an unwanted side affect that the final interactions of a loop loads useless data into the cache, polluting the cache, increasing bus traffic, and possibly evicting valuable temporal data. This problem is resolved by preload unrolling. For example consider:

```
for(i=0; i<NMAX; i++)

{

    PRELOAD(data[i+2]);

    sum += data[i];

}
```

Interactions i-1 and i preloads superfluous data. The problem is avoid by unrolling the end of the loop.

```
for(i=0; i<NMAX-2; i++)

{

    PRELOAD(data[i+2]);

    sum += data[i];

}

sum += data[NMAX-2];

sum += data[NMAX-1];
```

Unfortunately, preload loop unrolling does not work on loops with indeterminate iterations. Additionally, preloads beyond the end of data causes undesired table walks to occur thus reducing performance.

### A.4.5.8 Pointer Preload

Not all looping constructs contain induction variables. However, preloading techniques are still applied. Consider the following linked list traversal example:

```
while(p) {

    do_something(p->data);

    p = p->next;

}
```

The pointer variable *p* becomes a pseudo induction variable and the data pointed to by *p->next* is preloaded to reduce data transfer latency for the next iteration of the loop. Linked lists is converted to arrays as much as possible.

```
while(p) {

    PRELOAD(p->next);

    do_something(p->data);

    p = p->next;

}
```

Recursive data structure traversal is another construct where preloading is applied. This is similar to linked list traversal. Consider the following pre-order traversal of a binary tree:

```
preorder(treeNode *t) {

    if(t) {

        process(t->data);

        preorder(t->left);

        preorder(t->right);

    }

}
```



The pointer variable *t* becomes the pseudo induction variable in a recursive loop. The data structures pointed to by the values *t->left* and *t->right* is preloaded for the next iteration of the loop.

```
preorder(treeNode *t) {

    if(t) {

        PRELOAD(t->right);

        PRELOAD(t->left);

        process(t->data);

        preorder(t->left);

        preorder(t->right);

    }

}
```

Note the order reversal of the preloads in relationship to the usage. When there is a cache conflict and data is evicted from the cache then only the data from the first preload is lost.

Preloading a NULL pointer reduces performance by causing a table walk for page zero. This occurs on leaf nodes of a tree traversal. When the TLB entry for page zero is set to cause a translation fault then a table walk occurs on every preload. To improve performance, a page table entry that has permissions set to no access is used instead. The translation fault entry does not get cached in the TLB where as the permission fault entry does. Approximately half the node pointers in a binary tree are NULL pointers so this is a large performance impact when using preloading on tree traversal. Note that ANSI conforming C compilers do not have to equate the "NULL" pointer to a binary value of 0 but most do for simplicity of implementation.

### A.4.5.9 Preload to Reduce Register Pressure

Preload is used to reduce register pressure. When data is needed for an operation then the load is scheduled far enough in advance to hide the load latency. However, the load ties up the receiving register until the data is used. For example:

```
    ldr   r2, [r0]

; Process code {not yet cached latency > 60 core clocks}

    add   r1, r1, r2
```

In the above case, r2 is unavailable for processing until the add statement. Preloading the data load frees the register for use. The example code becomes:

```
    pld   [r0] ;preload the data keeping r2 available for use

;

; Process code -- a significant amount of code here hides the latency of

; the data from the preload returning. The number of saved cycles depends on

; the system's memory configuration.

;

    ldr   r2, [r0]

; Process code {ldr result latency is 3 core clocks}

    add   r1, r1, r2
```

With the added preload, register r2 is used for other operations until almost just before it is needed.

## A.5 Instruction Scheduling

This section discusses instruction scheduling optimizations. Instruction scheduling refers to the rearrangement of a sequence of instructions for the purpose of minimizing pipeline stalls. Reducing the number of pipeline stalls improves application performance. While making this rearrangement, care is taken to ensure that the rearranged sequence of instructions has the same effect as the original sequence of instructions. See Chapter 13.0, "Performance Considerations", for additional timing information.

### A.5.1 Load and Store Instructions

The 3rd generation microarchitecture has twelve memory buffers used for loading from and storing to external memory or L2 cache. Each of these buffers holds a request for up to a cache line worth of data and any given cacheable line is only allocated to one buffer at a time. A buffer holds up to four load requests, a preload request, or one or more (coalesced) store requests. Non-cacheable non-coalesceable stores and non-cacheable loads also use the buffers at a rate of one buffer per access.

The processor stalls when all memory buffers are in use and another memory buffer is needed. When any buffer has four load requests, any load miss (a load that misses both the data cache and the fill buffers) causes a stall until a load request is satisfied regardless of address. When any buffer has three or more load requests, any load double miss regardless of address causes a stall until all buffers have at least two available request slots.

Although this is not a general concern, certain code sequences cause the processor to stall due to a lack of available memory buffers. As a result, code attempts to keep the buffers less than full; when possible each outstanding cache line has:

- no more than four outstanding loads against it, or
- no more than two outstanding load doubles against it

When any of the buffers have the indicated content, then these cause a stall on the next issued memory operation.

### A.5.1.1 Scheduling Loads

On the 3rd generation microarchitecture, an **LDR** instruction has a result latency of three cycles assuming the data being loaded is in the data cache. When the instruction after the **LDR** needs to use the result of the load then it stalls for two cycles. When possible, the instructions surrounding the **LDR** instruction is rearranged to avoid this stall.

Consider the following example:

```
add   r1, r2, r3

ldr   r0, [r5]

add   r6, r0, r1

sub   r8, r2, r3

mul   r9, r2, r3
```

In the code shown above, the **ADD** instruction following the **LDR** stalls for two cycles because it uses the result of the load. The code is rearranged as follows to prevent the stalls:

```
ldr   r0, [r5]

add   r1, r2, r3

sub   r8, r2, r3

add   r6, r0, r1

mul   r9, r2, r3
```

Note that this rearrangement is not always possible. Consider the following example:

```
cmp   r1, #0

addne r4, r5, #4

subeq r4, r5, #4

ldr   r0, [r4]

cmp   r0, #10
```

In the example above, the **LDR** instruction cannot be moved before the **ADDNE** or the **SUBEQ** instructions because the **LDR** instruction depends on the result of these instructions. Rewrite the above code to make it run faster at the expense of increasing code size:

```
cmp   r1, #0

ldrne r0, [r5, #4]

ldreq r0, [r5, #-4]

addne r4, r5, #4

subeq r4, r5, #4

cmp   r0, #10
```

The optimized code takes six cycles to execute compared to the seven cycles taken by the unoptimized version.



The result latency for an **LDR** instruction is significantly higher when the data being loaded is not in the data cache. To minimize the number of pipeline stalls in such a situation the **LDR** instruction is moved as far away as possible from the instruction that uses result of the load. Note that this at times causes certain register values to be spilled to memory due to the increase in register pressure. In such cases, use a preload instruction or a preload hint to ensure that the data access in the **LDR** instruction hits the cache when it executes. A preload instruction is used in cases where the load instruction is sure to execute. Consider the following code sample:

```
; all other registers are in use

    sub   r1, r6, r7

    mul   r3, r6, r2

    mov   r2, r2, LSL #2

    orr   r9, r9, #0xf

    add   r0, r4, r5

    ldr   r6, [r0]

    add   r8, r6, r8

    add   r8, r8, #4

    orr   r8, r8, #0xf

; The value in register r6 is not used after this
```

In the code sample above, the **ADD** and the **LDR** instruction are moved before the **MOV** instruction. Note that this prevents pipeline stalls when the load hits the data cache. However, when the load is likely to miss the data cache, move the **LDR** instruction so that it executes as early as possible - before the **SUB** instruction. However, moving the **LDR** instruction before the **SUB** instruction changes the program semantics. It is possible to move the **ADD** and the **LDR** instructions before the **SUB** instruction when allowing the contents of the register R6 to be spilled and restored from the stack as shown below:

```
; all other registers are in use

    str   r6, [sp, #-4]!

    add   r0, r4, r5

    ldr   r6, [r0]

    mov   r2, r2, LSL #2

    orr   r9, r9, #0xf

    add   r8, r6, r8

    ldr   r6, [sp], #4

    add   r8, r8, #4

    orr   r8, r8, #0xf

    sub   r1, r6, r7

    mul   r3, r6, r2

; The value in register r6 is not used after this
```

As is seen above, the contents of the register R6 have been spilled to the stack and subsequently loaded back to the register R6 to retain the program semantics. Another way to optimize the code above is with the use of the preload instruction as shown below:

```
; all other registers are in use

    add   r0, r4, r5

    pld   [r0]

    sub   r1, r6, r7

    mul   r3, r6, r2

    mov   r2, r2, LSL #2

    orr   r9, r9, #0xf

    ldr   r6, [r0]

    add   r8, r6, r8

    add   r8, r8, #4

    orr   r8, r8, #0xf

; The value in register r6 is not used after this
```

### A.5.1.2 Scheduling Load and Store Double (LDRD/STRD)

**LDRD** loads 64-bits of data from an effective address into two consecutive registers, conversely **STRD** stores 64-bits from two consecutive registers to an effective address. There are two important restrictions on how these instructions are used:

- the effective address must be aligned on an 8-byte boundary
- the specified register must be even (R0, R2, etc.).

When this situation occurs, using **LDRD**/**STRD** instead of **LDM**/**STM** to do the same thing is more efficient because **LDRD**/**STRD** issues in only one/two clock cycle(s), as opposed to **LDM**/**STM** which always issue in three or more clock cycles.

The **LDRD** instruction has a result latency of three or four cycles depending on the destination register being accessed (assuming the data being loaded is in the data cache).

```
    add   r6, r7, r8

    sub   r5, r6, r9

; The following ldrd instruction loads values

; into registers r0 and r1

    ldrd  r0, [r3]

    orr   r8, r1, #0xf

    mul   r7, r0, r7
```

In the code example above, the **ORR** instruction stalls for three cycles because of the four cycle result latency for the second destination register of an **LDRD** instruction. The code shown above is rearranged to remove the pipeline stalls:

```
; The following ldrd instruction loads values

; into registers r0 and r1

    ldrd  r0, [r3]

    add   r6, r7, r8

    sub   r5, r6, r9

    mul   r7, r0, r7

    orr   r8, r1, #0xf
```

Any load operation (**PLD**, **LDR**, **LDRB**, and so on) directly following a **LDRD** instruction stalls for one cycle.

```
; The ldr instruction below stalls for 1 cycle

    ldrd   r0, [r3]

    ldr    r4, [r5]
```

Similarly, any read-from-CP15 operation (**MRC** P15, ...) after a **LDRD** exacts an additional issue cycle.

The processor stalls when any memory buffer has four active requests and another memory operation is issued. For example, when there are 4 LDR instructions pending against a memory buffer, then another LDR operation causes a stall, regardless of the address or hit/miss status for that final LDR.

Similarly, when any buffer has three or more load requests, an issued **LDRD** — regardless of address — causes a stall until all buffers have at least two available request slots.

A store that "hits" a pending load causes the machine to stall until the load completes.

### A.5.1.3 Scheduling Load and Store Multiple (LDM/STM)

**LDM** and **STM** instructions have an issue latency of three to twenty one cycles depending on the number of registers being loaded or stored. The issue latency is typically one cycle for each of the registers being loaded or stored assuming a data cache hit. The instruction following an **LDM** stalls whether or not this instruction depends on the results of the load. A **LDRD** or **STRD** instruction does not suffer from this drawback (except when followed by a memory operation) and is used where possible. Consider the task of adding two 64-bit integer values. Assume that the addresses of these values are aligned on an 8-byte boundary. This is achieved using the **LDM** instructions as shown below:

```
; r0 contains the address of the value being copied

; r1 contains the address of the destination location

    ldmia r0, {r2, r3}

    ldmia r1, {r4, r5}

    adds  r0, r2, r4

    adc   r1, r3, r5
```

When the code were written as shown above, assuming all the accesses hit the cache, the code takes eight cycles to complete. Rewriting the code as shown below using **LDRD** instruction takes seven cycles to complete. The performance increases when other instructions are filled in after **LDRD** to reduce the stalls due to the result latencies of the **LDRD** instructions.

```
; r0 contains the address of the value being copied

; r1 contains the address of the destination location

    ldrd  r2, [r0]

    ldrd  r4, [r1]

    adds  r0, r2, r4

    adc   r1,r3, r5
```

Similarly, the code sequence shown below takes four cycles to complete.

```
    stmia r0, {r2, r3}

    add   r1, r1, #1
```

The alternative version which is shown below takes three cycles to complete.

```
    strd  r2, [r0]

    add   r1, r1, #1
```

A rule of thumb for choosing **LDM** or **LDR/LDRD**: use **LDM** only when more than two registers are read.

## A.5.2 Scheduling Data Processing Instructions

Most 3rd generation microarchitecture data processing instructions have a result latency of one cycle. This means that the current instruction is able to use the result from the previous data processing instruction. However, the result latency is two cycles when the current instruction needs to use the result of the previous data processing instruction for a shift by immediate. As a result, the following code segment incurs a one cycle stall for the mov instruction:

```
sub   r6, r7, r8

add   r1, r2, r3

mov   r4, r1, LSL #2
```

The code above is rearranged as follows to remove the one cycle stall:

```
add   r1, r2, r3

sub   r6, r7, r8

mov   r4, r1, LSL #2
```

All data processing instructions incur a one cycle issue penalty and a one cycle result penalty when the shifter operand is a shift/rotate by a register or shifter operand is RRX. Since the next instruction always incurs a one cycle issue penalty, there is no way to avoid such a stall except by re-writing the assembler instruction. Consider the following segment of code:

```
mov   r3, #10

mul   r4, r2, r3

add   r5, r6, r2, LSL r3

sub   r7, r8, r2
```

The subtract instruction incurs a one cycle stall due to the issue latency of the add instruction as the shifter operand is shift by a register. The issue latency is avoided by changing the code as follows:

```
mov   r3, #10

mul   r4, r2, r3

add   r5, r6, r2, LSL #10

sub   r7, r8, r2
```



### A.5.3 Scheduling Multiply Instructions

Multiply instructions cause pipeline stalls due to either resource conflicts or result latencies. The following code segment incurs a stall of up to two cycles depending on the values in registers r1, r2, r4 and r5 due to resource conflicts.

```
mul   r0, r1, r2

mul   r3, r4, r5
```

The following code segment incurs a stall of one to two cycles depending on the values in registers r1 and r2 due to result latency.

```
mul   r0, r1, r2

mov   r4, r0
```

Note that a multiply instruction that sets the condition codes blocks the whole pipeline. A three cycle multiply operation that sets the condition codes behaves the same as a three cycle issue operation. Consider the following code segment:

```
muls  r0, r1, r2

add   r3, r3, #1

sub   r4, r4, #1

sub   r5, r5, #1
```

The add operation above stalls for two cycles when the multiply takes three cycles to complete. It is better to replace the code segment above with the following sequence:

```
mul   r0, r1, r2

add   r3, r3, #1

sub   r4, r4, #1

sub   r5, r5, #1

cmp   r0, #0
```

Please refer to Section 13.4, "Instruction Latencies" to get the instruction latencies for various multiply instructions. The multiply instructions is scheduled taking into consideration these instruction latencies.

The processor lifts certain operand restrictions on multiply instructions. For example, prior Intel XScale® microarchitectures required that *Rd* and *Rm* be different registers. 3rd generation microarchitecture has no such restriction, which enables simpler scheduling of multiplies in some situations.

### A.5.4 Scheduling SWP and SWPB Instructions

The **SWP** and **SWPB** instructions have a four cycle issue latency. As a result of this latency, the instruction following the **SWP**/**SWPB** instruction stalls for three cycles. **SWP** and **SWPB** instructions therefore are used only where absolutely needed.

For example, the following code is used to swap the contents of two memory locations:

```
; Swap the contents of memory locations pointed to by r0 and r1

    ldr   r2, [r0]

    swp   r2, [r1]

    str   r2, [r0]
```

The code above takes eight cycles to complete with instructions and data residing in the cache. The rewritten code below, takes five cycles to execute:

```
; Swap the contents of memory locations pointed to by r0 and r1

    ldr   r2, [r0]

    ldr   r3, [r1]

    str   r2, [r1]

    str   r3, [r0]
```



### A.5.5 Scheduling the MRA and MAR Instructions (MRRC/MCRR)

The **MRA** (**MRRC**) instruction has an issue latency of one cycle, a result latency of two or three cycles depending on the destination register value being accessed and a resource latency of two cycles.

Consider the code sample:

```
mra   r6, r7, acc0

mra   r8, r9, acc0

add   r1, r1, #1
```

The code shown above incurs a one cycle stall due to the two cycle resource latency of an **MRA** instruction. The code is rearranged as shown below to prevent this stall.

```
mra   r6, r7, acc0

add   r1, r1, #1

mra   r8, r9, acc0
```

Similarly, the code shown below incurs a two cycle penalty due to the three cycle result latency for the second destination register.

```
mra   r6, r7, acc0

mov   r1, r7

mov   r0, r6

add   r2, r2, #1
```

The stalls incurred by the code shown above are prevented by rearranging the code:

```
mra   r6, r7, acc0

add   r2, r2, #1

mov   r0, r6

mov   r1, r7
```

The **MAR** (**MCRR**) instruction has an issue, result, and resource latency of one cycle.

### A.5.6 Scheduling the MIA and MIAPH Instructions

The **MIA** instruction has an issue latency of one cycle. The result and resource latency varies from one to two cycles depending on the values in the source register.

Consider the following code sample:

```
mia   acc0, r2, r3

mia   acc0, r4, r5
```

The second **MIA** instruction above stalls for one cycle depending on the values in the registers r2 and r3 due to the one to two cycle resource latency.

Similarly, consider the following code sample:

```
mia   acc0, r2, r3

mra   r4, r5, acc0
```

The **MRA** instruction above stalls for one cycle depending on the values in the registers r2 and r3 due to the one to two cycle result latency. The **MIAPH** instruction has an issue latency of one cycle, result latency of two cycles and a resource latency of two cycles.

Consider the code sample shown below:

```
add   r1, r2, r3

miaph acc0, r3, r4

miaph acc0, r5, r6

mra   r6, r7, acc0

sub   r8, r3, r4
```

The second **MIAPH** instruction stalls for one cycle due to a two cycle resource latency. The **MRA** instruction stalls for one cycle due to a two cycle result latency. These stalls are avoided by rearranging the code as follows:

```
miaph acc0, r3, r4

add   r1, r2, r3

miaph acc0, r5, r6

sub   r8, r3, r4

mra   r6, r7, acc0
```



### A.5.7 Scheduling MRS and MSR Instructions

The **MRS** instruction has an issue latency of two cycles and a result latency of three cycles. The **MSR** instruction has an issue latency of two cycles (six when updating the mode bits).

Consider the code sample:

```
mrs   r0, cpsr

orr   r0, r0, #1

add   r1, r2, r3
```

The **ORR** instruction above incurs a one cycle stall due to the three cycle result latency of the **MRS** instruction. In the code example above, the **ADD** instruction is moved before the **ORR** instruction to prevent this stall.

### A.5.8 Scheduling CP15 Coprocessor Instructions

All CP15 operations stall the microarchitecture until complete and therefore are not overlapped with other operations. See Section 13.4.9, "Coprocessor Instructions" on page 229 for additional timing information.

## A.6 Optimizing C Libraries

Many of the standard C library routines benefit greatly by being optimized for 3rd generation microarchitecture. The following string and memory manipulation routines are tuned to obtain the best performance from the processor architecture (instruction selection, cache usage and data preload):

strcat, strchr, strcmp, strcoll, strcpy, strcspn, strlen, strncat, strncmp, strpbrk, strrchr, strspn, strstr, strtok, strxfrm, memchr, memcmp, memcpy, memmove, memset

## A.7 Optimizations for Size

For applications such as cell phone software it is necessary to optimize the code for improved performance while minimizing code size. Optimizing for smaller code size, in general, lowers the performance of your application. This section contains techniques for optimizing for code size using the 3rd generation microarchitecture instruction set.

### A.7.1 Space/Performance Trade Off

Many optimizations mentioned in the previous sections, improve the performance of ARM code. However, using these instructions results in increased code size. Use the following optimizations to reduce the space requirements of the application code.

#### A.7.1.1 Multiple Word Load and Store

The **LDM**/**STM** instructions are one word long and allow loading or storing multiple registers at once. Use the **LDM**/**STM** instructions instead of a sequence of loads/stores to consecutive addresses in memory whenever possible.

#### A.7.1.2 Use of Conditional Instructions

Using conditional instructions to expand if-then-else statements as described in Section A.3.1, "Conditional Instructions" results in increasing the size of the generated code, therefore, do not use conditional instructions when application code space requirements are an issue.

#### A.7.1.3 Use of PLD Instructions

The preload instruction **PLD** is only a hint, it does not change the architectural state of the processor. Using or not using these do not change the behavior of the code, therefore, avoid using these instructions when optimizing for space.

### A.7.2 Thumb

The microarchitecture supports the Thumb instruction set, which uses a 16-bit encoding. Programs compiled to this ISA typically are smaller than those targeting the 32-bit ARM ISA.



# Appendix B Microarchitecture Compatibility Guide

## B.1 Overview

This appendix describes new features in 3rd generation Intel XScale® microarchitecture (3rd generation microarchitecture or 3rd generation), compatibility of features relative to previous generations and features in that are no longer available in 3rd generation.

## B.2 New Features

This section lists new features for the 3rd generation microarchitecture.

### B.2.1 MMU Features

- Supersection page table descriptor supporting a physical addressing range of 36 bits
- Support for caching of page table descriptors in L2 cache
- New memory attribute encodings in the page table descriptor
- Page table descriptors to support shared and coherent memory

See Chapter 3.0, "Memory Management"

### B.2.2 New L1 Cache Functions

- Clean Data cache line by Set and Way
- Clean and invalidate data cache line by MVA
- Clean and invalidate data cache line by Set and Way

See Section 7.2.8, "Register 7: Cache Functions"

### B.2.3 LLR

- Data cache support for Low-locality of reference data

See Section 6.1.2, "Low-Locality of Reference (LLR)"

### B.2.4 Optional Level 2 Cache

- No L2, or lockable 256K/512K options
- Physically addressed using 36 bit addressing

See Section 8.0, "Level 2 Unified Cache (L2)"

### B.2.5 Support For Hardware Cache Coherency

- Memory hierarchy supports coherency
- External bus master pushes data directly into the cache

See Chapter 9.0, "Cache Coherence"

### B.2.6 Memory Ordering

- Weak memory consistency model defined
- Software has access to explicit ordering instructions

See Chapter 10.0, "Memory Ordering"

### B.2.7 PMU features

- New PMU events
- Performance counters are disabled independently of the clock counter

See Chapter 11.0, "Performance Monitoring"

### B.2.8 Instruction Behavior

- **MUL** and **MLA** support for same register for Rd, Rm, e.g **MUL R1,R1,R1**
- **SMULL, SMLAL, UMULL,** and **UMLAL** support for same register in RdhiandRm or RdLo and Rm.

## B.3 Features No Longer Available

This section lists features that were available for the previous generations and no longer available on 3rd generation processors.

### B.3.1 Memory Coalescing

- The previous generations K bit (disable-coalescing) in the AUX control register is not defined in 3rd generation microarchitecture

### B.3.2 Mini Data Cache

- The previous generations Mini Data cache is not present on 3rd generation microarchitecture instead a LLR memory attribute is used. See Appendix B.2.3,"LLR."for more details

## B.4 Compatibility With Respect To Previous Generation Microarchitecture

This section discusses the features on the previous generations that have been changed or replaced with different functionality on 3rd generation. This section describes how these features behave differently and how to utilize alternate features on the processor.

### B.4.1 Page Table Memory Attributes

Previous Generation Microarchitecture traditionally has 3 bits to define memory attributes, XCB.

3rd generation microarchitecture has 6 bits to define memory attributes,TEXCB and S.

The TEX bits [2:1] were previously defined to be zero and the S bit was defined as Should Be Zero (SBZ). Well behaved code that followed this definition function correctly on 3rd generation microarchitecture as shown in Table 152.

Table 152 shows the XCB encodings for previous generationss and mappings to 3rd generation microarchitecture.

**Table 152. Previous Generation Microarchitecture Page Table Attribute Encoding Compatibility**

| Previous Generation Microarchitecture | | | microarchitecture | |
|---|---|---|---|---|
| XCB | Description | | TEXCB | Description[a] |
| 0b000 | I/O memory | | 0b00000 | Strongly Ordered |
| 0b001 | Uncacheable | | 0b00001 | Uncacheable |
| 0b010 | Write Through; Read Allocate | | 0b00010 | Write Through; Read Allocate |
| 0b011 | Write Back; Read Allocate | | 0b00011 | Write Back; Read Allocate |
| 0b100 | Unpredictable | | 0b00100 | Uncacheable |
| 0b101 | Non-Coalescing | | 0b00101 | Shared Device |
| 0b110 | Mini Data Cache | | 0b00110 | LLR |
| 0b111 | Write Back<br>Write Allocate | | 0b00111 | Write Back<br>Read Allocate |

a. See subsequent sections in this chapter for more information

## B.4.2 Behavior Of Strongly Ordered Memory

### B.4.2.1 Behavioral Difference

On previous generationss, access to I/O memory stalls the pipeline until the memory operation has been sent out of the microarchitecture.

On the 3rd generation microarchitecture an access to strongly ordered memory waits until all prior explicit memory accesses have been observed. After a memory access to strongly ordered memory all subsequent memory accesses are stalled until the initial memory access has been observed. However after a memory access to strongly ordered memory other non-memory access instructions continue to execute.

### B.4.2.2 Compatibility Implication

Accesses to strongly ordered memory do not ensure timing of side effect completion (such as, doing a store to strongly ordered memory that maps to an interrupt controller to disable an interrupt, does not ensure that the interrupt is disabled for subsequent code). See ASSP documentation for details on how to ensure a device-update has occurred. One common scheme is to read back the just-written I/O location.

While previous generations did not stall until stores have completed these, but did stall longer than 3rd generation microarchitecture, so badly behaved code, that does not poll for effects, has different behavior when run on 3rd generation microarchitecture.

### B.4.2.3 Performance Difference

Instruction throughput is higher since code continues to execute while there is a load/ store to strongly ordered memory.

## B.4.3 Behavior Of Device Memory

### B.4.3.1 Behavioral Difference

Previously, a page table cache attribute of XCB = 0b101 was implemented as Non-cacheable, bufferable, but non coalescing. On a 3rd generation microarchitecture this memory attribute encoding is Device Memory.

After a memory access to shared device memory all subsequent memory accesses to shared device memory is sent out in the executed order, hence memory ordering to shared device memory is ensured. However after the initial memory access to shared device memory other memory accesses not to shared device memory and other non memory access instructions continue to execute.

Like previous generations this type of memory is uncacheable and non-coalescing.

### B.4.3.2 Compatibility Implication

Memory accesses not to shared device memory is re-ordered with respect to shared device memory.

Code polls devices for side effects, such as, when configuring a memory controller that is accessed as device memory, and then reading data from the memory which is configured as normal memory, this does not ensure that read to memory occurs before the memory controller is configured. To prevent this a fence is used such as a DWB.

### B.4.3.3 Performance Difference

Accesses to device memory on 3rd generation microarchitecture allows executing code to have a greater through put where there are no dependencies on the data from device memory.

## B.4.4 Low Locality Of Reference (LLR) Cache Usage

### B.4.4.1 Behavioral Difference

The page table attribute encoding that specifies LLR in 3rd generation microarchitecture, was the same encoding as the Mini-Data cache on previous generations. These two features are compatible in terms of most desired effects.

On 3rd generation microarchitecture LLR is a subset of the data cache, where as the previous generations Mini-Data cache was separate from the main data cache.

LLR on 3rd generation microarchitecture is also outer (L2) cacheable.

Previously, the bit position for the S bit is defined as Should be Zero (SBZ). Well behaved code that followed this definition behaves in a similar way on 3rd generation microarchitecture depending on the value of the Aux Control register as shown in the table below.

Most of the settings in the Aux Control register result in the same cache policy. The only exception is the previous generations policy of "Write back, Read/Write allocate" is now implemented as "Write back, Read allocate". 3rd generation microarchitecture L1 data cache is always "Read allocate".

See Section B.4.5, "L1 Allocation Policy" on page 294, for compatibly differences of switching from Read/Write allocate to Read allocate.

**Table 153. Auxiliary Control Register Bits [5:4]**

| Bits [5:4] | Previous Generation Microarchitecture | 3rd Generation Microarchitecture |
|---|---|---|
| 0b00 | Write back, Read allocate | Write back, Read allocate |
| 0b01 | Write back, Write allocate | Write back, Read allocate |
| 0b10 | Write through, Read allocate | Write through, Read allocate |
| 0b11 | Reserved | Write back, Read allocate |

### B.4.4.2 Compatibility Implication

Any code that tried to use the mini-data cache as on chip SRAM, by relying on the fact that it is not replaced, cannot make this assumption anymore because the LLR resides in the data cache.

Any code that relied on having 32KB of data cache, and an additional 2KB of mini-data cache, is disappointed.

### B.4.4.3 Performance Difference

Performance is affected, since the LLR pollutes 8K of the Data Cache



## B.4.5 L1 Allocation Policy

### B.4.5.1 Behavioral Difference

On 3rd generation microarchitecture the L1 Data cache does not support write allocate.

Any policy that is set to L1 write allocate is now interpreted to be L1 read allocate.

### B.4.5.2 Compatibility Implication

Behavior compatible, except where code explicitly expects line to be allocated on write, for example, using a **STR** to lock a line into the Data Cache

### B.4.5.3 Performance Difference

To exhibit a write allocate behavior in the memory hierarchy, the L2 is used with write allocate.

Products, with no L2, that write entire cache lines of data, and read these back at a later time, achieves similar performance to write allocate systems by using the DC Line allocate functions

## B.4.6 DC Line Allocate

### B.4.6.1 Behavioral Difference

DC line allocate is done in usr mode. In previous generations DC Line Allocate is only done in a privileged mode.

3rd generation microarchitecture generates a store breakpoint on a DC Line Allocate, previous generations do not.

3rd generation microarchitecture uses the VA, previous generations uses the MVA. In other words. On 3rd generation microarchitecture, the address to be allocated is first modified through PID. For an explanation of how the PID works see Section 7.2.13, "Register 13: Process ID".

3rd generation microarchitecture does a TLB walk for a DC line allocate, this causes MMU aborts. previous generations do not.

Reading the data from a newly Line Allocated line, while resulting in unpredictable values, does not cause unpredictable behavior. Previous Generation Microarchitecture causes an exception when the data were read before an explicit software write.

### B.4.6.2 Compatibility Implication

When the PID is set to any value greater than zero and code 'DC line allocates' an address where bits [31:25] of that address are zero. The 'DC line allocate' now allocates to the MVA remapped through the PID.

For example:

PID = 0xC0000000.
The address supplied to the to be 'DC line allocate' function = 0x00002000.

On previous generations 0x00002000 is allocated.

On 3rd generation microarchitecture 0xC0002000 is allocated.

Any code that relies on this instruction generating an Undef user mode no longer observes this behavior.

*Note:* Line Allocate is a deprecated feature. Future microarchitectures do not implement this command.

### B.4.6.3 Performance Difference

A DC line allocate on 3rd generation microarchitecture takes longer when the associated page table descriptor is not in the TLB.

A DC line allocate on 3rd generation microarchitecture is used more efficiently than previous generations implementation.

### B.4.7 Translation Table Register - Page Table Memory Attribute (P) Bit

#### B.4.7.1 Behavioral Difference

The P bit in the Auxiliary Control Register is deprecated. It is now logical ORed with the new P bit in the Translation Table base register.

#### B.4.7.2 Compatibility Implication

Software that ran on previous generationss acted on setting/clearing the P bit in the Aux Control Register, now on 3rd generation microarchitecture software needs to be aware that this bit is set in the Translation table base register. When software clears a P bit, it only observes the effect when both P bits are cleared.

#### B.4.7.3 Performance Difference

No differences in performance are foreseeable from this change.

### B.4.8 Drain Write Buffer

#### B.4.8.1 Behavioral Difference

**DWB** on 3rd generation microarchitecture is done in user mode. In previous generations DWB is only done in a privileged mode.

On previous generations, a DWB drains the write and fill buffer.

On 3rd generation microarchitecture, write buffers are drained, but loads are not.

#### B.4.8.2 Compatibility Implication

**DWB** does not guaranty ordering of loads with respect to subsequent loads after the drain write buffer.

In the below code segment the **DWB** does not prevent **LDR R3,[R4]** from occurring before **LDR R1,[R2]**. To prevent this happening a **DMB** is used.

```
LDR R1,[R2]

DWB

LDR R3,[R4]
```

#### B.4.8.3 Performance Difference

No impact to typical code.

## B.4.9 L1 Cache Invalidate Function

### B.4.9.1 Behavioral Difference

On previous generationss "Invalidate cache line by MVA" on a locked line invalidates the line, but not unlock it, leaving an empty hole in the cache.

On 3rd generation microarchitecture "Invalidate cache line by MVA" unlocks a locked line, as well as invalidating it.

### B.4.9.2 Compatibility Implication

No compatibility implications are foreseeable from this change.

### B.4.9.3 Performance Difference

3rd generation microarchitecture does not get unused holes appearing in the cache.

## B.4.10 Cache Organization, Locking And Unlocking

### B.4.10.1 Behavioral Difference

3rd generation microarchitecture L1 caches are 4 way as opposed to previous generations's 32 way caches.

On 3rd generation microarchitecture 3/4 ways are locked, as opposed to 28/32 ways locked on the previous generations.

### B.4.10.2 Compatibility Implication

Cache clean/unlocking algorithms need to be modified.

Any code that relies on locking 28/32 ways is now only able lock 3/4 ways.

### B.4.10.3 Performance Difference

Performance changes for code with unusual instruction or data access patterns.

## B.4.11 Data Cache Replacement Algorithm

### B.4.11.1 Behavioral Difference

Previous Generation Microarchitecture uses round robin when deciding which line to evict from the caches.

3rd generation microarchitecture uses pseudo LRU when deciding which line to evict from the L1 caches.

### B.4.11.2 Compatibility Implication

Cache clean/unlocking algorithms need to be modified, to use cache set and way functions.

### B.4.11.3 Performance Difference

Pseudo LRU is often a more efficient replacement algorithm, meaning code or data with a higher temporal locality stays in the cache longer.

## B.4.12 PLD

### B.4.12.1 Behavioral Difference

On previous generationss, a **PLD** instruction that requires a page table entry that is not in the TLB functions as a **NOP**.

On 3rd generation microarchitecture a **PLD** instruction when required walks the page table and load the entry into the TLB.

The **PLD** instruction does not generate any precise aborts on 3rd generation microarchitecture or previous generationss.

### B.4.12.2 Compatibility Implication

No compatibility implications are foreseeable from this change.

### B.4.12.3 Performance Difference

On previous generationss, **PLD**s that required a table walk did not provide any performance improvement.

On 3rd generation microarchitecture all **PLD**s begins preloading for the subsequent access to that data.

In some cases the walking the page table when not necessary reduces overall performance.

## B.4.13 SWP

### B.4.13.1 Behavioral Difference

On 3rd generation microarchitecture when a SWP is performed to a region of memory that is marked as shared it behaves differently from previous generationss.

The 3rd generation microarchitecture page table descriptor shared (S) bit was defined on previous generationss as Should Be Zero (SBZ). On
3rd generation microarchitecture, well behaved code that followed this definition behaves the same as previous generationss.

### B.4.13.2 Compatibility Implication

No compatibility implications are foreseeable from this change when the S bit is set to zero.

### B.4.13.3 Performance Difference

No performance differences are foreseeable from this change when the S bit is set to zero

## B.4.14 Page Table Walks

### B.4.14.1 Behavioral Difference

3rd generation microarchitecture translation table base register bits [4:3] allow page table walks to be cached in the L2. TTBASE bits [4:3] are defined on previous generationss as Should Be Zero (SBZ). On 3rd generation microarchitecture, well behaved code that followed this definition is fully compatible.

When TTBASE bits [4:3] are set to 0b11 and an L2 cache is present on
3rd generation microarchitecture then all table walks are L2 cacheable.

### B.4.14.2 Compatibility Implication

No compatibility implications are foreseeable from this change when the TTBASE bits [4:3] bits are set to zero.

When the table walks are set to L2 cacheable and the page table resides in memory marked as not L2 cacheable, then manipulation of page table descriptors are modifying uncached entries. To prevent this happening two things are done

1) Make translation table region L2 cacheable

2) Invalidate cached page table entries after modification

### B.4.14.3 Performance Difference

No performance differences are foreseeable from this change when the TTBASE bits [4:3] are set to zero.

## B.4.15 Coalescing

### B.4.15.1 Behavioral Difference

On previous generations, coalescing only occurs when: the memory region is coalescable, the external bus is busy and the K bit in the Aux control register is cleared.

On 3rd generation microarchitecture, when the memory region is coalescable the stores to that region wait in the memory buffer to coalesce. For details see Section 6.0, "Data Cache".

There is no K bit in the 3rd generation microarchitecture Aux control register.

On previous generationss, coalescing completes when the external bus is available for writing — the coalescing buffer is written to the bus and invalidated. On 3rd generation microarchitecture, coalescing continues until either:

- a hardware time-out expires
- the buffer is needed for another purpose

### B.4.15.2 Compatibility Implication

On 3rd generation microarchitecture, global coalescing cannot be disabled. When coalescing is not desired, then either a non coalescable region or explicit DWB is used.

### B.4.15.3 Performance Difference

On 3rd generation microarchitecture coalescing occurs more frequently making more efficient use of the bus.

## B.4.16 Buffers

### B.4.16.1 Behavioral Difference

Previous Generation Microarchitecture has separate Write / Fill / Pend buffers.

3rd generation microarchitecture has memory buffers that are write or fill. Each buffer pends.

### B.4.16.2 Compatibility Implication

No compatibility implications are foreseeable from this change.

### B.4.16.3 Performance Difference

More efficient use of buffers results in better performance.

Unusual data access patterns causes different buffer stall behavior.

## B.4.17 LDC

### B.4.17.1 Behavioral Difference

When the Coprocessor Access Register (CPAR) allows access to a coprocessor, previous generationss raise an undefined instruction exception when a **LDC** is done to a coprocessor that does not exist. No implicit load is performed.

On 3rd generation microarchitecture when the Coprocessor Access Register (CPAR) allows access to a coprocessor and a **LDC** is done to a coprocessor that does not exist, an implicit load is done before the undefined instruction exception is raised. When this load aborted, 3rd generation microarchitecture encounters a data abort instead of raising an undefined instruction exception. Likewise it also generates a Data Breakpoint when the address matches that in the data break point register.

### B.4.17.2 Compatibility Implication

On previous generationss a **LDC** to CP that does not exist raises an undefined instruction exception.

On 3rd generation microarchitecture a **LDC** to CP that does not exist generates a Data abort or breakpoint. To achieve the same behavior as previous generations, the software clears the CPAR permission bits for all Coprocessors that do not exist.

### B.4.17.3 Performance Difference

Accessing unimplemented Coprocessors is slower.

## B.4.18 Instruction Timings

Some instruction timings have changed on 3rd generation microarchitecture. See Section 13.4, "Instruction Latencies" on page 221 for exact timings.

## B.4.19 Debug

Some debug features have changed. This is relevant to vendors writing debug monitors. To non debug handler code the effects are not visible. Below is a brief summary of the changes.

The previous generations mini-instruction cache is replaced on 3rd generation microarchitecture with Debug SRAM.

3rd generation microarchitecture supports Hot debug to download code to Debug SRAM while the microarchitecture is executing.

The definition of SDS debug mode has changed. See Section 12.0, "Software Debug" for more details.

The trace buffer on 3rd generation microarchitecture support tracing Thumb. By default this feature is disabled, thus allowing the trace buffer to maintain compatibility with previous generations.