# *Intel® XScale™ Microarchitecture* for the PXA255 Processor

## *User's Manual*

*March, 2003*

Order Number: 278796

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

The ***Intel® XScale™ Microarchitecture Users Manual*** for the PXA255 processor may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

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

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

# Contents

## Figures

# Tables

# Introduction 1

## 1.1 About This Document

This document describes the Intel® XScale™ core as implemented in the PXA255 processor.

Intel Corporation assumes no responsibility for any errors which may appear 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 them.

### 1.1.1 How to Read This Document

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

Each chapter in this document focuses on a specific architectural feature of the Intel® XScale™ core.

- Chapter 2, “Programming Model”
- Chapter 3, “Memory Management”
- Chapter 4, “Instruction Cache”
- Chapter 5, “Branch Target Buffer”
- Chapter 6, “Data Cache”
- Chapter 7, “Configuration”
- Chapter 8, “Performance Monitoring”
- Chapter 10, “Software Debug”
- Chapter 11, “Performance Considerations”
- Appendix A, “Optimization Guide” covers instruction scheduling techniques.

***Note:*** Most of the “buzz words” and acronyms found throughout this document are captured in Section 1.3.2, “Terminology and Acronyms” on page 1-6, located at the end of this chapter.

### 1.1.2 Other Relevant Documents

- *ARM* Architecture Reference Manual* Document Number: ARM DDI 0100E
  This document describes the ARM* Architecture and is publicly available.
  See http://www.arm.com/ARMARM for details. Sold as:
  ARM* Architecture Reference Manual
  Second Edition, edited by David Seal: Addison-Wesley: ISBN 0-201-73719-1
- Intel® PXA255 Processor Developer’s Manual, Intel Order # 278693

- Intel® PXA255 Processor Design Guide, Intel Order # 278694
- *Intel® 80200 Processor Development Manual*, Intel Order #273411
  This document describes the first implementation of the Intel® XScale™ Microarchitecture in a microprocessor targeted at IO applications
  Available from http://developer.intel.com

## 1.2 High-Level Overview of the Intel® XScale™ core as Implemented in the Application Processors

The Intel® XScale™ core is an ARM* V5TE compliant microprocessor. It is a high performance and low-power device that leads the industry in MIPS/mW. The core 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 handheld devices, networking, storage, remote access servers, etc. The PXA255 processor is an example of an ASSP designed primarily for handheld devices. This document limits itself to describing the implementation of the Intel® XScale™ core as it is implemented in the PXA255 processor. In almost every attribute the Intel® XScale™ core used in the application processor is identical to the Intel® XScale™ core implemented in the Intel® 80200

The Intel® XScale™ core incorporates an extensive list of microarchitecture features that allow it to achieve high performance. This rich feature set lets you select the appropriate features that obtain the best performance for your application. Many of the micro-architectural features added to the Intel® XScale™ core help hide memory latency which often is a serious impediment to high performance processors. This includes:

- The ability to continue instruction execution even while the data cache is retrieving data from external memory
- A write buffer
- Write-back caching
- Various data cache allocation policies which can be configured differently for each application
- Cache locking

All these features improve the efficiency of the memory bus external to the core.

The Intel® XScale™ core efficiently handles audio processing through the support of 16-bit data types and enhanced 16-bit operations. These audio coding enhancements center around multiply and accumulate operations which accelerate many of the audio filtering and multimedia CODEC algorithms.

### 1.2.1 ARM* Compatibility

ARM* Version 5 (V5) Architecture added new features to ARM* Version 4, including among other inclusions, floating point instructions. The Intel® XScale™ core implements the integer instruction set of ARM* V5, but does not provide hardware support for any of the floating point instructions.

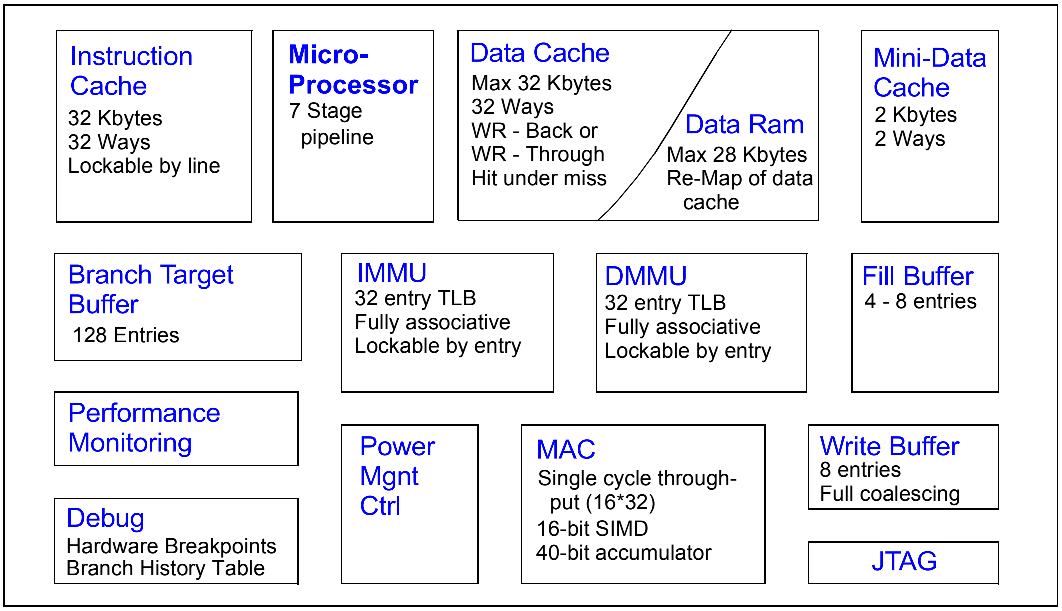
The Intel® XScale™ core provides the ARM* V5T Thumb instruction set and the ARM* V5E DSP extensions. To further enhance multimedia applications, the Intel® XScale™ core includes additional Multiply-Accumulate functionality as the first instantiation of Intel® Media Processing Technology. These new operations from Intel are mapped into ARM* coprocessor space.

Backward compatibility with StrongARM* products is maintained for user-mode applications. Operating systems may require modifications to match the specific hardware features of the Intel® XScale™ core and to take advantage of the performance enhancements added.

## 1.2.2 Features

Figure 1-1 shows the major functional blocks of the Intel® XScale™ core. The following sections give a brief, high-level overview of these blocks.

**Figure 1-1. Intel® XScale™ Microarchitecture Architecture Features**



### 1.2.2.1 Multiply/Accumulate (MAC)

The MAC unit supports early termination of multiplies/accumulates in two cycles and can sustain a throughput of a MAC operation every cycle. Several architectural enhancements were made to the MAC to support audio coding algorithms, which include a 40-bit accumulator and support for 16-bit packed data.

Refer to Section 2.3, "Extensions to ARM* Architecture" on page 2-2 for more information.

### 1.2.2.2 Memory Management

The Intel® XScale™ core implements the Memory Management Unit (MMU) Architecture specified in the *ARM* Architecture Reference Manual*. The MMU provides access protection and virtual to physical address translation.

The MMU Architecture also specifies the caching policies for the instruction cache and data cache. These policies are specified as page attributes and include:

- identifying code as cacheable or non-cacheable
- selecting between the mini-data cache or data cache
- write-back or write-through data caching
- enabling data write allocation policy
- enabling the write buffer to coalesce stores to external memory

Refer to Chapter 3, “Memory Management” for more information.

### 1.2.2.3 Instruction Cache

The Intel® XScale™ core implements a 32-Kbyte, 32-way set associative instruction cache with a line size of 32 bytes. All requests that “miss” the instruction cache generate a 32-byte read request to external memory. A mechanism to lock critical code within the cache is also provided.

Refer to Chapter 4, “Instruction Cache” for more information.

In addition to the main instruction cache there is a 2-Kbyte mini-instruction cache dedicated to advanced debugging features. Refer to Chapter 10, “Software Debug” for more information.

### 1.2.2.4 Branch Target Buffer

The Intel® XScale™ core 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. Refer to Chapter 5, “Branch Target Buffer” for more information.

### 1.2.2.5 Data Cache

The Intel® XScale™ core implements a 32-Kbyte, 32-way set associative data cache and a 2-Kbyte, 2-way set associative mini-data cache. Each cache has a line size of 32 bytes, supporting write-through or write-back caching.

The data/mini-data cache is controlled by page attributes defined in the MMU Architecture and by coprocessor 15.

Refer to Chapter 6, “Data Cache” for more information.

The Intel® XScale™ core allows applications to re-configure a portion of the data cache as data RAM. Software may place special tables or frequently used variables in this RAM. Refer to Section 6.4, “Re-configuring the Data Cache as Data RAM” on page 6-10 for more information.

### 1.2.2.6 Fill Buffer & Write Buffer

The Fill Buffer and Write Buffer enable the loading and storing of data to memory beyond the Intel® XScale™ core. The Write Buffer carries all write traffic beyond the core allowing data coalescing when both globally enabled, and when associated with the appropriate memory page types. The Fill buffer assists the loading of data from memory, which along with an associated Pend Buffer allows multiple memory reads to be outstanding. Another key function of the Fill

Buffer [along with the Instruction Fetch Buffers] is to allow the application processor external SDRAM to be read as 4-word bursts, rather than single word accesses, improving overall memory bandwidth.

Both the Fill, Pend and Write buffers help to decouple core speed from any limitations to accessing external memory. Further details on these buffers can be found in Section 6.5, “Write Buffer/Fill Buffer Operation and Control” on page 6-13

#### 1.2.2.7 Performance Monitoring

Two performance monitoring counters have been added to the Intel® XScale™ core that can be configured to monitor various events in the Intel® XScale™ core. These events allow a software developer to measure cache efficiency, detect system bottlenecks and reduce the overall latency of programs.

Refer to Chapter 8, “Performance Monitoring” for more information.

#### 1.2.2.8 Power Management

The Intel® XScale™ core incorporates a power and clock management unit that can assist ASSPs in controlling their clocking and managing their power. These features are described in Section 7.3, “CP14 Registers” on page 7-15.

#### 1.2.2.9 Debug

Intel® XScale™ core supports software debugging through two instruction address breakpoint registers, one data-address breakpoint register, one data-address/mask breakpoint register, a mini-instruction cache and a trace buffer.

Testability & hardware debugging is supported on the Intel® XScale™ core 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 the Intel® XScale™ core such as built-in self-test and boundary-scan.

The JTAG port can also be used as a hardware interface for debugger control of software. Refer to Chapter 10, “Software Debug” for more information.

## 1.3 Terminology and Conventions

### 1.3.1 Number Representation

All numbers in this document can be 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 would be represented as 0x6B in hexadecimal and 0b1101011 in binary.

Bitfields are expressed with a colon within square brackets, for example a * b [63:0] denotes a 64 bit arithmetic partial result.

## 1.3.2 Terminology and Acronyms

ASSP

Application Specific Standard Product. Defined for a specific purpose but not exclusively available to a single customer.

API

Application Programming Interface, typically a defined set of function calls and passed parameters defining how layers of software interact.

Assert

This term refers to the logically active value of a signal or bit.

BTB

Branch Target Buffer, a predictor of instructions that follow branches.

Clean

A ‘clean’ operation with regard to a data cache is the writing back of modified data to the external memory system, resulting in no ‘dirty’ lines remaining in the cache.

Coalescing

Coalescing means bringing together a new store operation with an existing store operation already issued. This includes, in Peripheral Component Interconnect [PCI] terminology, write merging, write collapsing, and write combining.

Deassert

This term refers to the logically inactive value of a signal or bit.

Flush

A ‘flush’ operation invalidates the location(s) in the cache by deasserting the valid bit. This now invalid cacheline will no longer be searched on cache accesses. A ‘flush’ operation on a write-back data cache does not implicitly imply a ‘clean’ operation.

NOP

Shortening of No OPeration, meaning an instruction with no state changing effect. A typical example might be Add-constant-zero without condition flag update

Privilege Mode

Any chip mode of operation that is not User Mode; the mode typically used for applications software. A Privileged Mode gains access to shared system resources.

Reserved

*A reserved* field is a register field that may be used by an implementation but not intended to be programmed. If the initial value of a reserved field is supplied by software, this value must be zero. Software should not modify reserved fields or depend on any values in reserved fields.

TLB

Translation Look-aside Buffer, a cache of Page Table descriptors loaded from memory to minimize page-table walking overhead.

# Programming Model 2

This chapter describes the programming model of the Intel® XScale™ core, namely the implementation options and extensions to the ARM* Version 5 architecture chosen for the PXA255 processor.

## 2.1 ARM* Architecture Compatibility

The Intel® XScale™ core implements the integer instruction set architecture specified in ARM* Version 5TE. T refers to the Thumb instruction set and E refers to the DSP-Enhanced instruction set.

ARM* Version 5 introduces a few more architecture features over Version 4;

- tiny pages of 1 Kbyte each
- a new instruction (**CLZ**) that counts the leading zeroes in a data value
- enhanced ARM-Thumb transfer instructions
- new breakpoint instructions (**BKPT**)
- a modification of the system control coprocessor, CP15.

## 2.2 ARM* Architecture Implementation Options

### 2.2.1 Big Endian versus Little Endian

The Intel® XScale™ core supports both big and little endian data representation. The B-bit of the Control Register (Coprocessor 15, register 1, bit 7) selects big and little endian mode.

The default behavior of the application processor at reset is little endian. To run in big endian mode, the B bit must be set before attempting any sub-word accesses to memory. Note that the endian bit takes effect even if the MMU is disabled.

The B-bit affects the data path, fill and write buffers and subword data location in memory. The LCD controller and DMA controller on the application processor can also switch endianism for data movement independent of the B-bit.

In concurrence with the changes introduced in ARM* V5, the Intel® XScale™ core does not support legacy code requiring the 26-bit address space.

### 2.2.2 Thumb

The Intel® XScale™ core supports the Thumb instruction set. These are 16-bit ARM* instructions that implement similar functions to the ARM* 32-bit instruction set, but offer advantages in reducing code size.

### 2.2.3 ARM* DSP-Enhanced Instruction Set

The Intel® XScale™ core implements ARM*'s DSP-enhanced instruction set. There are new multiply instructions that operate on 16-bit data values and new saturation instructions. Some of the new instructions are:

- SMLAxy32<=16x16+32
- SMLAWy 32<=32x16+32
- SMLALxy64<=16x16+64
- SMULxy32<=16x16
- SMULWy32<=32x16
- QADDadds two registers and saturates the result if an overflow occurred
- QDADDdoubles and saturates one of the input registers then add and saturate
- QSUBsubtracts two registers and saturates the result if an overflow occurred
- QDSUBdoubles and saturates one of the input registers then subtract and saturate

The Intel® XScale™ core also implements LDRD, STRD and PLD instructions with the following implementation notes:

- PLD is interpreted as a read operation by the MMU and is ignored by the data breakpoint unit, i.e., PLD will never generate data breakpoint events.
- PLD to a non-cacheable page performs no action. Also, if the targeted cache line is already resident, this instruction has no effect.
- Both LDRD and STRD instructions will generate an alignment exception when the address is not on a 64-bit boundary.

MCRR and MRRC are only supported on the Intel® XScale™ core when directed to coprocessor 0 and are used to access the internal accumulator. See Section 2.3.1.2 for more information. Access to other coprocessors on the application processor generates the Undefined instruction exception.

### 2.2.4 Base Register Update

If a data abort is signalled on a memory instruction that specifies writeback, the contents of the base register will not be updated. This holds for all load and store instructions. This behavior matches that of the first generation StrongARM* processor and is referred to in the ARM* V5 architecture as the *Base Restored Abort Model*.

## 2.3 Extensions to ARM* Architecture

The Intel® XScale™ core made a few extensions to the ARM* Version 5 architecture to meet the needs of various markets and design requirements. The following is a list of the extensions which are discussed in the next sections.

- A DSP coprocessor (CP0) has been added that contains a 40-bit accumulator and 8 new operations in coprocessor space, hereafter referred to as new instructions.

- New page attributes were added to the page table descriptors. The C and B page attribute encoding was extended by one more bit to allow for more encodings: write allocate and mini-data cache.
- Additional functionality has been added to coprocessor 15. Coprocessor 14 was created.
- Enhancements were made to the Event Architecture, instruction cache and data cache parity error exceptions, breakpoint events, and imprecise external data aborts.

## 2.3.1 DSP Coprocessor 0 (CP0)

The Intel® XScale™ core adds a DSP coprocessor to the architecture for the purpose of increasing the performance and the precision of audio processing algorithms. This coprocessor contains a 40-bit accumulator and 8 new instructions.

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. **MRA** and **MAR** 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 Coprocessor Access Register is set. Any access to CP0 when this bit is clear will cause an undefined exception. (See Section 7.2.13, "Register 15: Coprocessor Access Register" on page 7-14 for more details). Only privileged software can set this bit in the Coprocessor Access Register.

The 40-bit accumulator will need to be saved on a context switch if multiple processes are using it.

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

### 2.3.1.1 Multiply With Internal Accumulate Format

A new multiply format has been created to define operations on 40-bit accumulators. Table 2-1, "Multiply with Internal Accumulate Format" on page 2-4 shows the layout of the new format. The opcodes selected for this new format lie within the ARM* coprocessor register transfer instruction type. These instructions have their own syntax.

## Table 2-1. 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 | Description | Notes |
|---|---|---|
| 31:28 | **cond** - ARM* condition codes | - |
| 19:16 | **opcode_3** - specifies the type of multiply with internal accumulate | The Intel® XScale™ core 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** - Multiplier | |
| 7:5 | **acc** - select 1 of 8 accumulators | The Intel® XScale™ core only implements acc0; access to any other acc has unpredictable effect. |
| 3:0 | **Rm** - Multiplicand | - |

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. The Intel® XScale™ core defines a single 40-bit accumulator referred to as *acc0;* future implementations may define multiple internal accumulators. The Intel® XScale™ core uses opcode_3 to define six instructions, **MIA**, **MIAPH**, **MIABB, MIABT, MIATB and MIATT.**

## Table 2-2. 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 | 0 | 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])[39:0] + acc0[39:0]
Exceptions: none
Qualifiers  Condition Code
            No condition code flags are updated
Notes:      Early termination is supported. Instruction timings can be found
            in Section 11.2.4, “Multiply Instruction Timings” on page 11-5.
            Specifying R15 for register Rs or Rm has unpredictable results.
            acc0 is defined to be 0b000 on the Intel® XScale™ core.
```

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 will be 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 if the condition specified in the instruction matches the condition code status.

### Table 2-3. 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[15:0] * Rs[15:0]) +
                       sign_extend(Rm[31:16] * Rs[31:16]) +
                       acc0[39:0]
Exceptions: none
Qualifiers  Condition Code
            No condition code flags are updated
Notes:      Instruction timings can be found
            in Section 11.2.4, "Multiply Instruction Timings" on page 11-5.
            Specifying R15 for register Rs or Rm has unpredictable results.
            acc0 is defined to be 0b000 on the Intel® XScale™ core
```

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 and then added to the value in the 40-bit accumulator (acc0).

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

**Table 2-4. MIAxy{<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 can be found
            in Section 11.2.4, "Multiply Instruction Timings" on page 11-5.
            Specifying R15 for register Rs or Rm has unpredictable results.
            acc0 is defined to be 0b000 on the Intel® XScale™™ core.
```

The **MIAxy** instruction performs one16-bit signed multiply and accumulates these to a single 40-bit accumulator. **x** refers to either the upper half or lower half of register Rm (multiplicand) and **y** refers to the upper or lower half of Rs (multiplier). A value of 0x1 will select bits [31:16] of the register which is specified in the mnemonic as T (for top). A value of 0x0 will select bits [15:0] of the register which is specified in the mnemonic as B (for bottom).

**MIAxy** does not support unsigned multiplication; all values in Rs and Rm will be interpreted as signed data values.

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

### 2.3.1.2 Internal Accumulator Access Format

The Intel® XScale™ core defines a new instruction format for accessing internal accumulators in CP0. Table 2-5, "Internal Accumulator Access Format" on page 2-7 shows that the opcode falls into the coprocessor register transfer space.

The *RdHi* and *RdLo* fields allow up to 64 bits of data transfer between standard ARM* registers and an internal accumulator. The *acc* field specifies 1 of 8 internal accumulators to transfer data to/from. The Intel® XScale™ core implements a single 40-bit accumulator referred to as acc0; future implementations can specify multiple internal accumulators of varying sizes.

Access to the internal accumulator is allowed in all processor modes (user and privileged) as long as bit 0 of the Coprocessor Access Register is set. (See Section 7.2.13, "Register 15: Coprocessor Access Register" on page 7-14 for more details).

The Intel® XScale™ core implements two instructions **MAR** and **MRA** that move two ARM* registers to acc0 and move acc0 to two ARM* registers, respectively.

### Table 2-5. Internal Accumulator Access Format

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="4">cond</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>L</td><td colspan="4">RdHi</td><td colspan="4">RdLo</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td colspan="3">acc</td></tr>
</table>

| Bits | Description | Notes |
|---|---|---|
| 31:28 | **cond** - ARM* condition codes | - |
| 20 | **L** - move to/from internal accumulator<br>0= move to internal accumulator (MAR)<br>1= move from internal accumulator (MRA) | - |
| 19:16 | **RdHi** - specifies the high order eight (39:32) bits of the internal accumulator. | On a read of the acc, this 8-bit high order field will be sign extended.<br>On a write to the acc, the lower 8 bits of this register will be written to acc[39:32] |
| 15:12 | **RdLo** - specifies the low order 32 bits of the internal accumulator | - |
| 7:4 | Should be zero | This field could be used in future implementations to specify the type of saturation to perform on the read of an internal accumulator. (e.g., a signed saturation to 16-bits may be useful for some filter algorithms.) |
| 3 | Should be zero | - |
| 2:0 | **acc** - specifies 1 of 8 internal accumulators | The Intel® XScale™ core only implements acc0; access to any other acc is unpredictable |

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

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

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

### Table 2-6. 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 can be found in
            Section 11.2.4, "Multiply Instruction Timings" on page 11-5
            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 if the condition specified in the instruction matches the condition code status.

This instruction executes in any processor mode.

### Table 2-7. 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 can be found in
            Section 11.2.4, "Multiply Instruction Timings" on page 11-5
            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 if the condition specified in the instruction matches the condition code status.

This instruction executes in any processor mode.

## 2.3.2 New Page Attributes

The Intel® XScale™ core extends the ARM* page attributes defined by the C & B bits in the page descriptors with an additional X bit. This bit allows four more attributes to be encoded when X=1. These new encodings include allocating data for the mini-data cache and write-allocate caching. A full description of the encodings can be found in Section 3.2.3, “Data Cache and Write Buffer” on page 3-2.

The Intel® XScale™ core retains ARM* definitions of the C & B encoding when X = 0, which is different than the StrongARM* products. The memory attribute for the mini-data cache has been moved and replaced with the write-through caching attribute.

When write-allocate is enabled, a store operation that misses the data cache (cacheable data only) will generate a line fill. If disabled, a line fill only occurs when a load operation misses the data cache (cacheable data only).

Write-through caching causes all store operations to be written to memory, whether they are first written to the cache or not. This feature is useful for maintaining data cache coherency.

The Intel® XScale™ core also adds a P bit in the first level descriptors to allow an ASSP to identify a new memory attribute. The application processor doesn’t implement a function for this bit. All instances of the P bit in first-level descriptors must be written as zero. Bit 1 in the Control Register (coprocessor 15, register 1, opcode=1) that is used to interact with the P bit must also be written as zero.

The X, C, B & P attributes are programmed in the translation table descriptors, which are highlighted in Table 2-8, “First-level Descriptors” on page 2-9, Table 2-9, “Second-level Descriptors for Coarse Page Table” on page 2-9 and Table 2-10, “Second-level Descriptors for Fine Page Table” on page 2-10. Two second-level descriptor formats have been defined for the Intel® XScale™ core, one is used for the coarse page table and the other is used for the fine page table.

*AP* bits are ARM* Access Permission controls.

### Table 2-8. 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="22">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 colspan="4">SBZ</td><td colspan="4">TEX</td><td colspan="2">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="20">Fine page table base address</td><td colspan="2">SBZ</td><td>P</td><td colspan="4">Domain</td><td colspan="3">SBZ</td><td>1</td><td>1</td></tr>
</table>

### Table 2-9. 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 colspan="4">TEX</td><td colspan="2">AP3</td><td colspan="2">AP2</td><td colspan="2">AP1</td><td colspan="2">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 colspan="2">AP3</td><td colspan="2">AP2</td><td colspan="2">AP1</td><td colspan="2">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 colspan="2">SBZ</td><td colspan="4">TEX</td><td colspan="2">AP</td><td>C</td><td>B</td><td>1</td><td>1</td></tr>
</table>

**Table 2-10. Second-level Descriptors for Fine Page Table**

| 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SBZ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 0 | 0 |
| Large page base address | | | | | | | | | | | | | | | | TEX | | | | AP3 | | AP2 | | AP1 | | AP0 | | C | B | 0 | 1 |
| Small page base address | | | | | | | | | | | | | | | | | | | | AP3 | | AP2 | | AP1 | | AP0 | | C | B | 1 | 0 |
| Tiny Page Base Address | | | | | | | | | | | | | | | | | | | | | | TEX | | | | AP | | C | B | 1 | 1 |

The TEX (Type Extension) field is present in several of the descriptor types. In the Intel® XScale™ core, only the LSB of this field is used; this is called the X bit.

A Small Page descriptor does not have a TEX field. For these descriptors, TEX is implicitly zero; that is, they operate as if the X bit had a '0' value.

The X bit, when set, modifies the meaning of the C and B bits. Description of page attributes and their encoding can be found in Chapter 3, "Data Cache and Write Buffer".

## 2.3.3 Additions to CP15 Functionality

To accommodate the functionality in the Intel® XScale™ core, registers in CP15 and CP14 have been added or augmented. See Chapter 7, "Configuration" for details.

At times it is necessary to be able to guarantee exactly when a CP15 update takes effect. For example, when enabling memory address translation (turning on the MMU), it is vital to know when the MMU is actually guaranteed to be in operation. To address this need, a processor-specific code sequence is defined for the Intel® XScale™ core. The sequence -- called CPWAIT -- is shown in Example 2-1.

**Example 2-1. CPWAIT: Canonical method to wait for CP15 update**

```
;; The following macro should be used when software needs to be
;; assured that a CP15 update has taken effect.
;; It may only be used while in a privileged mode, because it
;; accesses CP15.

MACRO CPWAIT

       MRC P15, 0, R0, C2, C0, 0            ; arbitrary read of CP15
       MOV R0, R0                           ; wait for it
       SUB PC, PC, #4                       ; branch to next instruction

       ; At this point, any previous CP15 writes are
       ; guaranteed to have taken effect.
ENDM
```

When setting multiple CP15 registers, system software may opt to delay the assurance of their update. This is accomplished by executing CPWAIT only after a sequence of MCR instructions.

The CPWAIT sequence guarantees that CP15 side-effects are complete by the time the CPWAIT is complete. It is possible, however, that the CP15 side-effect will take place before CPWAIT completes or is issued. Programmers should take care that this does not affect the correctness of their code.

## 2.3.4 Event Architecture

### 2.3.4.1 Exception Summary

Table 2-11 shows all the exceptions that the Intel® XScale™ core may generate, and the attributes of each. Subsequent sections give details on each exception. A precise exception is defined as one where R14_*mode* always contains a pointer to locate the instruction that caused the exception. Imprecise exceptions know the last instruction, but must look beyond the ARM* registers for the cause of the exception.

**Table 2-11. Exception Summary**

| Exception Description | Exception Type[a] | Precise? |
|---|---|---|
| Reset | Reset | N |
| FIQ | FIQ | N |
| IRQ | IRQ | N |
| External Instruction | Prefetch | Y |
| Instruction MMU | Prefetch | Y |
| Instruction Cache Parity | Prefetch | Y |
| Lock Abort | Data | Y |
| MMU Data[b] | Data | Y |
| External Data | Data | N |
| Data Cache Parity | Data | N |
| Software Interrupt | Software Interrupt | Y |
| Undefined Instruction | Undefined Instruction | Y |
| Debug Events[c] | varies | varies |

[a] Exception types are those described in the ARM, section 2.5.
[b] Only exception that updates Fault Address Register, CP15 Register 6
[c] Refer to Chapter 10, “Software Debug” for more details

### 2.3.4.2 Event Priority

The Intel® XScale™ core follows the exception priority specified in the *ARM Architecture Reference Manual*. The processor has additional exceptions that might be generated while debugging. For information on these debug exceptions, see Chapter 10, “Software Debug”.

**Table 2-12. Event Priority (Sheet 1 of 2)**

| Exception | Priority |
|---|---|
| Reset | 1 (Highest) |
| Data Abort (Precise & Imprecise) | 2 |
| FIQ | 3 |

Table 2-12. Event Priority (Sheet 2 of 2)

| Exception | Priority |
|---|---|
| IRQ | 4 |
| Prefetch Abort | 5 |
| Undefined Instruction, SWI | 6 (Lowest) |

### 2.3.4.3 Prefetch Aborts

The Intel® XScale™ core detects three types of prefetch aborts: Instruction MMU abort, external abort on an instruction access, and an instruction cache parity error. These aborts are described in Table 2-13. Domains are defined in the *ARM* Architecture Reference Manual*. The Fault Address Register (FAR) in coprocessor 15 holds the address causing an abort under certain conditions.

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

Table 2-13. Intel® XScale™ Core Encoding of Fault Status for Prefetch Aborts

| Priority | Sources | FSR[10,3:0][a] | Domain | FAR |
|---|---|---|---|---|
| Highest | Instruction MMU Exception<br>Several exceptions can generate this encoding:<br>- translation faults<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<br>This exception occurs when the memory system beyond the core reports an error on an instruction cache fetch.<br>For the application processor this is an internal bus error as there is no signal pin to generate an error from external memory | 0b10110 | invalid | invalid |
| Lowest | Instruction Cache Parity Error Exception | 0b11000 | invalid | invalid |

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

### 2.3.4.4 Data Aborts

Two types of data aborts exist in the Intel® XScale™ core: precise and imprecise. A precise data abort is defined as one where R14_ABORT always contains the PC (+8) of the instruction that caused the exception. An imprecise abort is one where R14_ABORT contains the PC (+4) of the next instruction to execute and not the address of the instruction that caused the abort. In other words, instruction execution will have advanced beyond the instruction that caused the data abort.

On the Intel® XScale™ core precise data aborts are recoverable and imprecise data aborts are not recoverable.

#### *Precise Data Aborts*

- A lock abort is a precise data abort; the extended Status field of the Fault Status Register (FSR) is set to 0xb10100. This abort occurs when a lock operation directed to the MMU (instruction or data) or instruction cache causes an exception, due to either a translation fault, access permission fault or external bus fault.

The Fault Address Register (FAR) is undefined and R14_ABORT is the address of the aborted instruction + 8.

- A data MMU abort is precise. These are due to an alignment fault, translation fault, domain fault, permission fault or external data abort on an MMU translation. The status field is set to a predetermined ARM* definition which is shown in Table 2-14, "Intel® XScale™ Core Encoding of Fault Status for Data Aborts" on page 2-13.
The Fault Address Register is set to the effective data address of the instruction and R14_ABORT is the address of the aborted instruction + 8.

*First-level* and *Second-level* refer to page descriptor fetches. A *section* is a 1Mbyte memory area. For details see the *ARM* Architecture Reference Manual*.

### Table 2-14. Intel® XScale™ Core Encoding of Fault Status for Data Aborts

| Priority | Sources | | FSR[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<br>This data abort occurs on an MMU lock operation (data or instruction TLB) or on an Instruction Cache lock operation. | | 0b10100 | 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.

### *Imprecise data aborts*

- A data cache parity error is 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.

The Fault Address Register for all imprecise data aborts is undefined and R14_ABORT is the address of the next instruction to execute + 4, which is the same for both ARM* and Thumb mode.

Although the Intel® XScale™ core guarantees the *Base Restored Abort Model* for precise aborts, it cannot do so in the case of imprecise aborts. A Data Abort handler may encounter an updated base register if it is invoked because of an imprecise abort.

Imprecise data aborts may create scenarios that are difficult for an abort handler to recover. Both external data aborts and data cache parity errors may result in corrupted data in the targeted registers. Because these faults are imprecise, it is possible that the corrupted data will have been used before the Data Abort fault handler is invoked. Because of this, software should treat imprecise data aborts as unrecoverable.

Memory accesses marked as “stall until complete” (see Section 3.2.3) can also result in imprecise data aborts. For these types of accesses, the fault is easier to manage than the general case as it is guaranteed to be raised within three instructions of the instruction that caused it. In other words, if a “stall until complete” **LD** or **ST** instruction triggers an imprecise fault, then that fault will be seen by the program within three instructions.

With this knowledge, it is possible to write code that can reliably access faulting memory. Place three **NOP** instructions after such an access. If an imprecise fault occurs, it will do so during the **NOP**s; the data abort handler will see identical register and memory state as it would with a precise exception, and so would be able to recover. An example of this is shown in Example 2-2 on page 2-14.

### Example 2-2. Shielding Code from Potential Imprecise Aborts

```
;; Example of code that maintains architectural state through the
;; window where an imprecise fault might occur.

        LD      R0, [R1]                      ; R1 points to stall-until-complete
                                              ;   region of memory
        NOP
        NOP
        NOP
        ; Code beyond this point is guaranteed not to see any aborts
        ; from the LD.
```

Of course, if a system design precludes events that could cause external aborts, then such precautions are not necessary.

#### *Multiple Data Aborts*

Multiple data aborts may be detected by hardware but only the highest priority one will be reported. If the reported data abort is precise, software can correct the cause of the abort and re-execute the aborted instruction. If the lower priority abort still exists, it will be reported. Software can handle each abort separately until the instruction successfully executes.

If the reported data abort is imprecise, software needs to check the Saved Program Status Register (SPSR) to see if the previous context was executing in abort mode. If this is the case, the link back to the current process has been lost and the data abort is unrecoverable.

## 2.3.4.5 Events from Preload Instructions

A **PLD** instruction will never cause the Data MMU to fault for any of the following reasons:

- Domain Fault
- Permission Fault
- Translation Fault

If execution of the **PLD** would cause one of the above faults, then the **PLD** does not cause the load to occur.

This feature allows software to issue **PLD**s speculatively. Example 2-3 on page 2-15 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 will not cause a fault. Rather, it will be ignored and the loop will terminate normally.

### Example 2-3. 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
```

## 2.3.4.6 Debug Events

Debug events are covered in Section 10.4, “Debug Exceptions” on page 10-5.

# Memory Management 3

This chapter describes the memory management unit implemented in the Intel® XScale™ core.

## 3.1 Overview

The Intel® XScale™ core implements the Memory Management Unit (MMU) Architecture specified in the *ARM Architecture Reference Manual*. To accelerate virtual to physical address translation, the Intel® XScale™ core uses both an instruction Translation Look-aside Buffer (TLB) and a data TLB to cache the latest translations. Each TLB holds 32 entries and is fully-associative. Not only do the TLBs contain the translated addresses, but also the access rights for memory references.

If an instruction or data TLB miss occurs, a hardware translation-table-walking mechanism is invoked to translate the virtual address to a physical address. Once translated, the physical address is placed in the TLB along with the access rights and attributes of the page or section. These translations can also be locked down in either TLB to guarantee the performance of critical routines.

The Intel® XScale™ core allows system software to associate various attributes with regions of memory:

- Cacheable
- Bufferable
- Line allocate policy
- Write policy
- I/O
- Mini Data Cache
- Coalescing

See Section 3.2.3, “Data Cache and Write Buffer” on page 3-2 for a description of page attributes and Section 2.3.2, “New Page Attributes” on page 2-9 to find out where these attributes have been mapped in the MMU descriptors.

***Note:*** The virtual address with which the TLBs are accessed may be remapped by the PID register. See Section 7.2.11, “Register 13: Process ID” on page 7-12 for a description of the PID register.

## 3.2 Architecture Model

The following sub-sections describe the Architecture Model.

## 3.2.1 Version 4 vs. Version 5

ARM* MMU Version 5 Architecture introduces the support of tiny pages, which are 1 KByte in size. The reserved field in the first-level descriptor (encoding 0b11) is used as the fine page table base address. The exact bit fields and the format of the first and second-level descriptors are found in Section 2.3.2, “New Page Attributes” on page 2-9.

The attributes associated with a particular region of memory are configured in the memory management page table and control the behavior of accesses to the instruction cache, data cache, mini-data cache, and the write buffer. These attributes are ignored when the MMU is disabled.

To allow compatibility with older system software, the new Intel® XScale™ core attributes take advantage of encoding space in the descriptors that were formerly reserved and defaulted to zero.

## 3.2.2 Instruction Cache

When examining the X, C, and B bits in a descriptor, the Instruction Cache only utilizes the C bit. If the C bit is clear, the Instruction Cache considers a code fetch from that memory to be non-cacheable, and will not fill a cache entry. If the C bit is set, then fetches from the associated memory region will be cached.

## 3.2.3 Data Cache and Write Buffer

All of the X, C & B descriptor bits affect the behavior of the Data Cache and the Write Buffer.

If the X bit for a descriptor is zero, the C and B bits operate as mandated by the ARM* architecture. This behavior is detailed in Table 3-1.

If the X bit for a descriptor is one, the C and B bits’ meaning is extended, as detailed in Table 3-2.

**Table 3-1. Data Cache and Buffer Behavior when X = 0**

| C B | Cacheable? | Bufferable? | Write Policy | Line Allocation Policy | Notes |
|---|---|---|---|---|---|
| 0 0 | N | N | - | - | Stall until complete[a] |
| 0 1[b] | N | Y | - | - | |
| 1 0[c] | Y | Y | Write Through | Read Allocate | |
| 1 1 | Y | Y | Write Back | Read Allocate | |

[a] Normally, the processor will continue executing after a data access if no dependency on that access is encountered. With this setting, the processor will stall execution until the data access completes. This guarantees to software that the data access has taken effect by the time execution of the data access instruction completes. External data aborts from such accesses will be imprecise (but see Section 2.3.4.4 for a method to shield code from this imprecision).

[b] Different from StrongARM, which for these attributes did not coalesce in the Write Buffer

[c] Different from StrongARM, which for these attributes selected the mini data cache, see when X = 1

Table 3-2. Data Cache and Buffer Behavior when X = 1

| C B | Cacheable? | Bufferable? | Write Policy | Line Allocation Policy | Notes |
|---|---|---|---|---|---|
| 0 0 | - | - | - | - | Unpredictable -- do not use |
| 0 1 | N | Y | - | - | Writes will not coalesce into buffers[a] |
| 1 0 | (Mini Data Cache) | - | - | - | Cache policy is determined by MD field of Auxiliary Control register[b] |
| 1 1 | Y | Y | Write Back | Read/Write Allocate | |

[a] Normally, bufferable writes can coalesce with previously buffered data in the same address range

[b] See Section 7.2.2 for a description of this register

## 3.2.4 Details on Data Cache and Write Buffer Behavior

If the MMU is disabled all data accesses will be non-cacheable and non-bufferable. This is the same behavior as when the MMU is enabled, and a data access uses a descriptor with X, C, and B all set to 0.

The X, C, and B bits determine when the processor should place new data into the Data Cache. The cache places data into the cache in lines (also called blocks). Thus, the basis for making a decision about placing new data into the cache is a called a "Line Allocation Policy".

If the Line Allocation Policy is read-allocate, all load operations that miss the cache request a 32-byte cache line from external memory and allocate it into either the data cache or mini-data cache (this is assuming the cache is enabled). Store operations that miss the cache will not cause a line to be allocated.

If read/write-allocate is in effect, load or store operations that miss the cache will request a 32-byte cache line from external memory if the cache is enabled.

The other policy determined by the X, C, and B bits is the Write Policy. A write-through policy instructs the Data Cache to keep external memory coherent by performing stores to both external memory and the cache. A write-back policy only updates external memory when a line in the cache is cleaned or needs to be replaced with a new line. Generally, write-back provides higher performance because it generates less data traffic to external memory.

More details on cache policies can be found in Section 6.2.3, "Cache Policies" on page 6-4.

## 3.2.5 Memory Operation Ordering

A *fence* memory operation (memop) is one that guarantees all memops issued prior to the fence will execute before any memop issued after the fence. Thus software may issue a fence to impose a partial ordering on memory accesses.

Table 3-3 on page 3-4 shows the circumstances in which memops act as fences.

Table 3-3. Memory Operations that Impose a Fence

| operation | X | C | B |
|---|---|---|---|
| load | - | 0 | - |
| store | 1 | 0 | 1 |
| load or store | 0 | 0 | 0 |

Any swap (**SWP** or **SWPB**), to a page that would create a fence, is also a fence.

### 3.2.6 Exceptions

The MMU can generate prefetch aborts for instruction accesses and data aborts for data memory accesses. The types and priorities of these exceptions are described in Section 2.3.4, "Event Architecture" on page 2-11.

Data address alignment checking is enabled by setting bit 1 of the Control Register (CP15, register 1). Alignment faults are still reported even if the MMU is disabled. All other MMU exceptions are disabled when the MMU is disabled.

## 3.3 Interaction of the MMU, Instruction Cache, and Data Cache

The MMU, instruction cache, and data/mini-data cache may be enabled/disabled independently. The instruction cache can be enabled with the MMU enabled or disabled. However, the data cache can only be enabled when the MMU is enabled. Therefore only three of the four combinations of the MMU and data/mini-data cache enables are valid. The invalid combination will cause undefined results.

Table 3-4. Valid MMU & Data/mini-data Cache Combinations

| MMU | Data/mini-data Cache |
|---|---|
| Off | Off |
| On | Off |
| On | On |

## 3.4 Control

### 3.4.1 Invalidate (Flush) Operation

The entire instruction and data TLBs can be invalidated at the same time with one command or they can be invalidated separately. An individual entry in the data or instruction TLB can also be invalidated. See Table 7-13, "TLB Functions" on page 7-11 for a listing of commands supported by the Intel® XScale™ core.

Globally invalidating a TLB will not affect locked TLB entries. However, the invalidate-entry operations can invalidate individual locked entries. In this case, the locked entry remains in the TLB, but will never “hit” on an address translation. Effectively, a hole is in the TLB. This situation can be rectified by unlocking the TLB.

## 3.4.2 Enabling/Disabling

The MMU is enabled by setting bit 0 in coprocessor 15, register 1 (Control Register).

When the MMU is disabled, accesses to the instruction cache default to cacheable and all accesses to data memory are made non-cacheable.

A recommended code sequence for enabling the MMU is shown in Equation 3-1.

### Example 3-1. Enabling the MMU

```
; This routine provides software with a predictable way of enabling the MMU.
; After the CPWAIT, the MMU is guaranteed to be enabled. Be aware
; that the MMU will be enabled sometime after MCR and before the instruction
; that executes after the CPWAIT.
; Programming Note: This code sequence requires a one-to-one virtual to
; physical address mapping on this code since
; the MMU may be enabled part way through. This would allow the instructions
; after MCR to execute properly regardless of the state of the MMU.

MRC P15,0,R0,C1,C0,0; Read CP15, register 1
ORR R0, R0, #0x1; Turn on the MMU
MCR P15,0,R0,C1,C0,0; Write to CP15, register 1

; For a description of CPWAIT, see
; Section 2.3.3, "Additions to CP15 Functionality" on page 2-10
CPWAIT
; The MMU is guaranteed to be enabled at this point; the next instruction or
; data address will be translated.
```

## 3.4.3 Locking Entries

Individual entries can be explicitly loaded & locked into the instruction and data TLBs. See Table 7-16, “TLB Lockdown Functions” on page 7-12 for the exact commands.

*Note:* If a lock operation finds the virtual address translation already resident in the TLB, the results are unpredictable. An invalidate entry command, see Table 7-13, “TLB Functions” on page 7-11, before the lock command will ensure proper operation. Software can also accomplish this by invalidating all entries, as shown in Example 3-2 on page 3-6.

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.

A procedure for locking entries into the instruction TLB is shown in Example 3-2 on page 3-6.

If a MMU abort is generated during an instruction or data TLB lock operation, the Fault Status Register is updated to indicate a Lock Abort (see Section 2.3.4.4, “Data Aborts” on page 2-12), and the exception is reported as a data abort.

### Example 3-2. Locking Entries into the Instruction TLB

```
; R1, R2 and R3 contain the virtual addresses to translate and lock into
; the instruction TLB.

; The value in R0 is ignored in the following instruction.
; Hardware guarantees that accesses to CP15 occur in program order

MCR P15,0,R0,C8,C5,0      ; Invalidate the entire instruction TLB

MCR P15,0,R1,C10,C4,0     ; Translate virtual address (R1) and lock into
                          ; instruction TLB
MCR P15,0,R2,C10,C4,0  ; Translate
                          ; virtual address (R2) and lock into instruction TLB
MCR P15,0,R3,C10,C4,0     ; Translate virtual address (R3) and lock into
                          ; instruction TLB

CPWAIT
; For a description of CPWAIT, see
; Section 2.3.3, “Additions to CP15 Functionality” on page 2-10
; The MMU is guaranteed to be updated at this point; the next instruction will
; see the locked instruction TLB entries.
```

*Note:* If exceptions are allowed to occur in the middle of this routine, the TLB may end up caching a translation that is about to be locked. For example, if R1 is the virtual address of an interrupt service routine and that interrupt occurs immediately after the TLB has been invalidated, the lock operation will be ignored when the interrupt service routine returns back to this code sequence. Software should disable interrupts (FIQ or IRQ) in this case.

As a general rule, software should avoid locking in anything other than Supervisor mode.

The proper procedure for locking entries into the data TLB is shown in Example 3-3 on page 3-7.

**Example 3-3. Locking Entries into the Data TLB**

```
; R1, and R2 contain the virtual addresses to translate and lock into the data TLB

MCR  P15,0,R1,C8,C6,1      ; Invalidate the data TLB entry specified by the
                           ; virtual address in R1
MCR  P15,0,R1,C10,C8,0     ; Translate virtual address (R1) and lock into
                           ; data TLB

; Repeat sequence for virtual address in R2
MCR  P15,0,R2,C8,C6,1      ; Invalidate the data TLB entry specified by the
                           ; virtual address in R2
MCR  P15,0,R2,C10,C8,0     ; Translate virtual address (R2) and lock into
                           ; data TLB

CPWAIT                     ; wait for locks to complete
; For a description of CPWAIT, see
; Section 2.3.3, "Additions to CP15 Functionality" on page 2-10
; The MMU is guaranteed to be updated at this point; the next instruction will
; see the locked data TLB entries.
```

***Note:*** If exceptions are allowed to occur in the middle of this routine, the TLB may end up caching a translation into a data TLB entry that was about to be locked. In general exceptions should be avoided while locking TLB entries. Software should preferably lock TLBs in Supervisor mode.

## 3.4.4 Round-Robin Replacement Algorithm

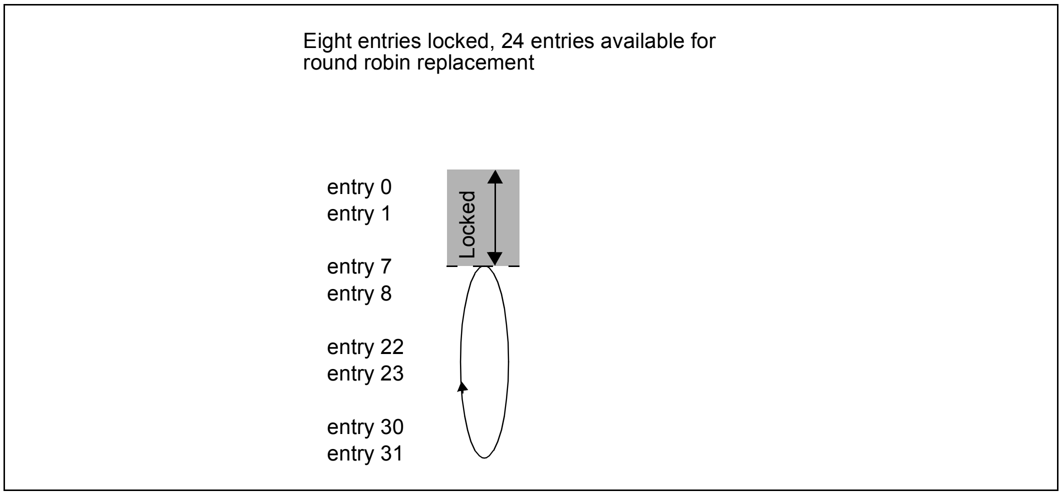
The line replacement algorithm for the TLBs is round-robin; there is a round-robin pointer that keeps track of the next entry to replace. The next entry to replace is the one sequentially after the last entry that was written. For example, if the last virtual to physical address translation was written 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 if 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 will wrap back to entry 0 upon the next translation.

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, if the first three entries were locked down, the round-robin pointer would be entry 3 after it rolled over from entry 31.

Only entries 0 through 30 can be locked in either TLB; entry 31 can never be locked. If the lock pointer is at entry 31, a lock operation will update the TLB entry with the translation and ignore the lock. In this case, the round-robin pointer will stay at entry 31.

Figure 3-1. Example of Locked Entries in TLB


Eight entries locked, 24 entries available for round robin replacement
entry 0
entry 1
entry 7
entry 8
entry 22
entry 23
entry 30
entry 31
Locked

# Instruction Cache 4

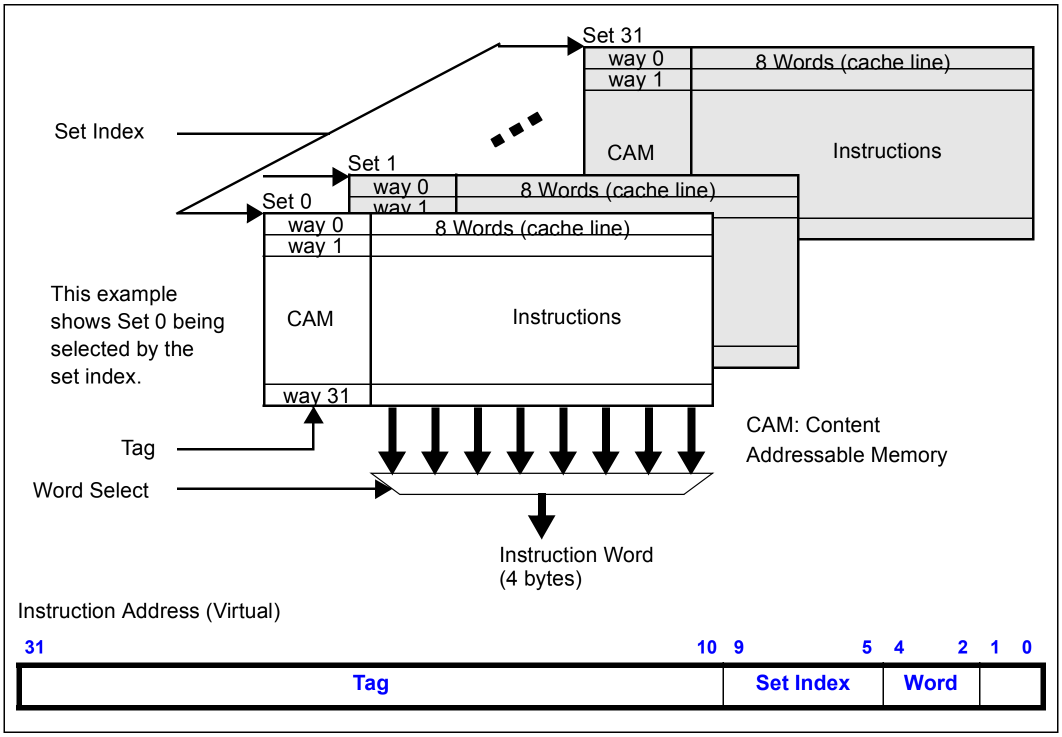
The Intel® XScale™ core instruction cache enhances performance by reducing the number of instruction fetches from external memory. The cache provides fast execution of cached code. Code can also be locked down when guaranteed or fast access time is required. An additional 2Kbyte mini instruction cache is used exclusively during debugging, see Section 10.13.6 for details.

## 4.1 Overview

Figure 4-1 shows the cache organization and how the instruction address is used to access the cache.

The instruction cache is a ***32-Kbyte, 32-way set associative cache***; this means there are 32 sets with each set containing 32 ways. Each way of a set contains eight 32-bit words and one valid bit, which is referred to as a line. The replacement policy is a round-robin algorithm. The cache also supports the ability to lock code in at a line granularity.

**Figure 4-1. Instruction Cache Organization**



The instruction cache is virtually addressed and virtually tagged. Tag bits[1:0] are ignored as far as the cache lookup is concerned. Bit 1 may be used subsequently to select a Thumb instruction.

*Note:* The virtual address presented to the instruction cache may be remapped by the PID register. See Section 7.2.11, “Register 13: Process ID” on page 7-12 for a description of the PID register.

# 4.2 Operation

## 4.2.1 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. If the cache contains the requested instruction, the access “hits” the cache, and the cache returns the requested instruction. If the cache does not contain the requested instruction, the access “misses” the cache, and the cache requests a fetch from external memory of the 8-word line (32 bytes) that contains the requested instruction using the fetch policy described in Section 4.2.3. As the fetch returns instructions to the cache, they are placed in one of two fetch buffers and the requested instruction is delivered to the instruction decoder.

A fetched line will be written from the fetch buffer into the cache if it is cacheable. Code is designated as eligible for caching when the Memory Management Unit (MMU) is disabled Section 4.2.2 or when the MMU is enabled and the cacheable (C) bit is set to 1 in it’s corresponding page. See Chapter 2, “New Page Attributes” for details on page attributes.

Disabling the MMU requires invalidation of the instruction cache first to prevent accidental matching of physical addresses with the existing virtual addresses in the cache.

*Note:* An instruction fetch may miss the cache but hit one of the fetch buffers. When this happens, the requested instruction will be delivered to the instruction decoder in the same manner as a cache hit.

## 4.2.2 The Instruction Cache Is Disabled

Disabling the cache prevents any lines from being loaded into the instruction cache from memory. While the cache is disabled, it is still accessed and may generate a hit if the instruction is already in the cache.

Disabling the instruction cache ***does not*** disable instruction buffering that may occur within the instruction fetch buffers. The two 32-byte instruction fetch buffers will always be enabled while the cache is disabled. 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 using the fetch policy described in Section 4.2.3.

To flush the Instruction Fetch Buffers see Section 4.3.3.

## 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 external memory. The instruction cache can handle up to two misses. Each external fetch request uses a fetch buffer that holds 32-bytes and eight valid bits, one for each word.

A miss causes the following:

1. A fetch buffer is allocated.

2. The instruction cache sends a fetch request to the external bus. This request is for a complete 32-byte cache line that will fill in sequence.
3. Instructions words are returned back from the external bus, at a maximum rate of 1 word per core cycle but typically much slower for Flash or SDRAM. As each word returns, the corresponding valid bit is set for the word in the currently selected fetch buffer.
4. As soon as the fetch buffer receives the requested instruction, it forwards the instruction to the instruction decoder for execution. Note at this point, with the pipeline restarting, another instruction cache miss may occur necessitating the use of the other Instruction Fetch Buffer.
5. When all words to an Instruction Fetch Buffer have returned, the fetched line will be written into the instruction cache if it's cacheable and if the instruction cache is enabled. The line chosen for update in the cache is controlled by the round-robin replacement algorithm. This update may evict a valid line at that location.
6. Once the cache is updated, the eight valid bits of the fetch buffer are invalidated and the cacheline valid bit is set.

## 4.2.4 Round-Robin Replacement Algorithm

The line replacement algorithm for the instruction cache is round-robin. Each set in the instruction cache has a round-robin pointer that keeps track of the next line (in that set) to replace. The next line to replace in a set is the one after the last line that was written from a fetch buffer. For example, if the line for the last external instruction fetch was written into way 5-set 2, the next line to replace for that set would be way 6. The other round-robin pointers for the other sets will not be affected.

After reset, way 31 is pointed to by the round-robin pointer for all the sets. Once a line is written into way 31, the round-robin pointer points to the first available way of a set, beginning with way0 if no lines have been locked into that particular set. Locking lines into the instruction cache reduces the available lines for cache updating. For example, if the first three lines of a set were locked down, the round-robin pointer would point to the line at way 3 after it rolled over from way 31. Refer to Section 4.3.4, "Locking Instructions in the Instruction Cache" on page 4-6 for more details on cache locking.

## 4.2.5 Parity Protection

The instruction cache is protected by parity to ensure data integrity. Each instruction cache word has 1 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 if the Intel® XScale™ core attempts to execute the instruction. Before servicing the exception, hardware places a notification of the error in the Fault Status Register (Coprocessor 15, register 5).

A software exception handler can recover from an instruction cache parity error. This can be accomplished by invalidating the instruction cache and the branch target buffer and then returning to the instruction that caused the prefetch abort exception. A simplified code example is shown in Example 4-1, Recovering from an Instruction Cache Parity Error. A more complex handler might choose to invalidate the specific line that caused the exception and then invalidate the BTB.

### Example 4-1. Recovering from an Instruction Cache Parity Error

```
; Prefetch abort handler
MCR P15,0,R0,C7,C5,0      ; Invalidate the instruction cache and branch target
                          ; buffer

CPWAIT                    ; wait for effect (see Section 2.3.3 for a
                      ; description of CPWAIT)

SUBS PC,R14,#4        ; Returns to the instruction that generated the
                          ; parity error
```

If a parity error occurs on an instruction that is locked in the cache, the software exception handler needs to unlock the instruction cache, invalidate the cache and then re-lock the code in before it returns to the faulting instruction.

## 4.2.6 Instruction Fetch Latency

The instruction fetch latency is dependent on the core to memory frequency ratio, system bus bandwidth, system memory, etc. The outstanding external memory bus activity on the PXA255 processor will have the highest impact on instruction fetch latency.

## 4.2.7 Instruction Cache Coherency

The instruction cache does not detect modification to program memory by loads, stores or actions of other bus masters. Several situations may require program memory modification, such as uploading code from disk.

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

To achieve cache coherence, instruction cache contents can be invalidated after code modification in external memory is complete. Refer to Section 4.3.3, “Invalidating the Instruction Cache” on page 4-5 for the proper procedure for invalidating the instruction cache.

If the instruction cache is not enabled, or code is being written to a non-cacheable region, software must still invalidate the instruction cache 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 into memory the writes involve the data cache not the instruction cache. Care must be taken to force writes completely out of the processor data path into external memory before attempting to execute the code. If writing into a non-cacheable region, flushing the write buffers is sufficient precaution (see Section 7.2.7 for a description of this operation). If writing to a cacheable region, then the data cache should be submitted to a Clean/Invalidate operation (see Section 6.3.3.1) to ensure coherency. Any data cache cleaning must be done before the previously mentioned instruction cache invalidation. This typically applies to code that is created, being modified, or simply written into memory, prior to code execution. Typical examples would be copying ROM code to DRAM, or dynamic libraries.

# 4.3 Instruction Cache Control

## 4.3.1 Instruction Cache State at RESET

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

## 4.3.2 Enabling/Disabling

The instruction cache is enabled by setting bit 12 in coprocessor 15, register 1 (Control Register). This process is illustrated in Example 4-2, Enabling the Instruction Cache.

### Example 4-2. Enabling the Instruction Cache

```
; Enable the Instruction Cache
MRC P15, 0, R0, C1, C0, 0             ; Read the control register
ORR R0, R0, #0x1000                   ; set bit 12 -- the I bit
MCR P15, 0, R0, C1, C0, 0             ; Write the control register
CPWAIT                    ; wait for effect, see Section 2.3.3
```

## 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 7-12, “Cache Functions” on page 7-9 for the exact command.) This command does not unlock any lines that were locked in the instruction cache or 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 can also be found in Table 7-14, “Cache Lockdown Functions” on page 7-11.

There is an inherent delay from the execution of the instruction cache invalidate command to where the next instruction will see the result of the invalidate. The following routine can be used to guarantee proper synchronization.

### Example 4-3. Invalidating the Instruction Cache

```
MCR P15,0,R1,C7,C5,0      ; Invalidate the instruction cache and branch
                          ; target buffer

CPWAIT                    ; wait for effect, see Section 2.3.3
; The instruction cache is guaranteed to be invalidated at this point; the next
; instruction sees the result of the invalidate command.
```

The Intel® XScale™ core also supports invalidating an individual line from the instruction cache. See Table 7-12, “Cache Functions” on page 7-9 for the exact command.

## 4.3.4 Locking Instructions in the Instruction Cache

Software has the ability to lock performance critical code routines into the instruction cache. Up to 28 lines in each set can be locked; hardware will ignore the lock command if software is trying to lock all the lines in a particular set i.e. ways 28-31 can never be locked. When this happens, the line will still be allocated into the cache but the lock will be ignored. The round-robin pointer will stay between way 28 and way 31 for that set.

Lines can be locked into the instruction cache by initiating a write to coprocessor 15. See Table 7-14, “Cache Lockdown Functions” on page 7-11 for the exact command.

Lines are locked into a set starting at way 0 and may progress up to way 27; which set a line gets locked into depends on the set index of the virtual address. Figure 4-2 is an example of where lines of code may be locked into the cache along with how the round-robin pointer is affected.

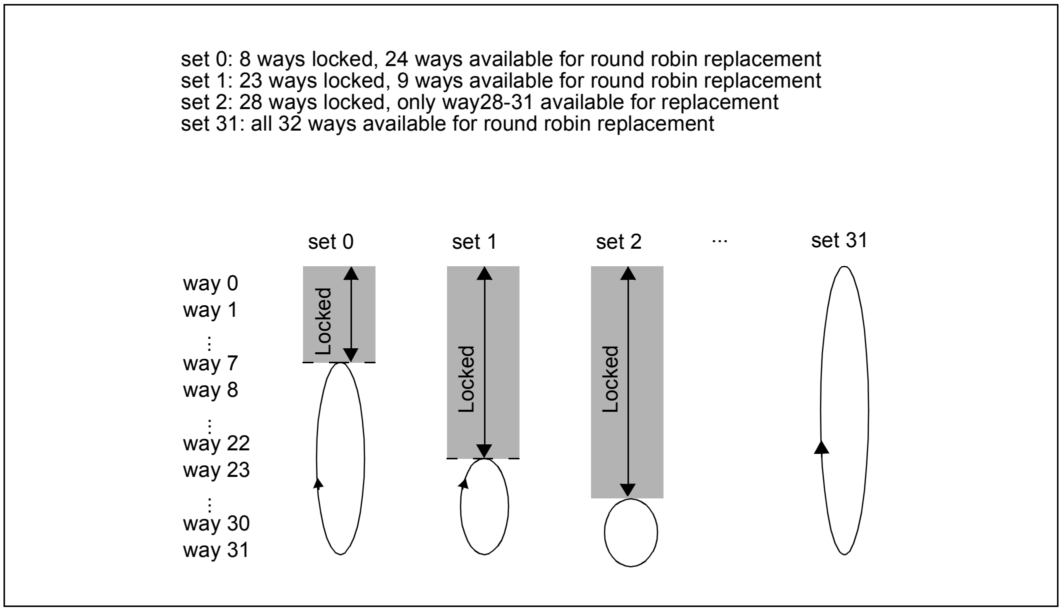
There are several requirements for locking down code. Failure to follow these requirements will produce unpredictable results when accessing the instruction cache.

- the routine used to lock lines down in the cache must be placed in non-cacheable memory, which means the MMU is enabled. Hence fetches of cacheable code must not occur while locking instructions into the cache.
- the code being locked into the cache must be cacheable.
- the instruction cache must be enabled and invalidated prior to locking down lines.

*Note:* Exceptions must be disabled during cache locking, to prevent the wrong contents from being locked down.

System programmers must ensure that the code to lock instructions into the cache is not closer than 128 bytes to a non-cacheable/cacheable page boundary. If the processor fetches ahead into a cacheable page, then the first requirement noted above could be violated

**Figure 4-2. Locked Line Effect on Round Robin Replacement**



set 0: 8 ways locked, 24 ways available for round robin replacement
set 1: 23 ways locked, 9 ways available for round robin replacement
set 2: 28 ways locked, only way28-31 available for replacement
set 31: all 32 ways available for round robin replacement

Software can lock down several different routines located at different memory locations. This may cause some sets to have more locked lines than others as shown in Figure 4-2.

Example 4-4, Locking Code into the Instruction Cache shows how a routine, called “lockMe” in this example, might be locked into the instruction cache. This example is incomplete as it doesn’t take into account the notes on page 4-6. However if this code runs with interrupts disabled and the locking code is placed by the assembler above the code to be locked in memory to avoid the prefetch issue, then the notes can be satisfied.

### Example 4-4. Locking Code into the Instruction Cache

```
lockMe:                    ; This is the code that will be locked into the cache
        mov r0, #5
        add r5, r1, r2
    . . .
lockMeEnd:
        . . .

codeLock:                  ; here is the code to lock the “lockMe” routine
        ldr r0, =(lockMe AND NOT 31); r0 gets a pointer to the 1st line locked

        ldr r1, =(lockMeEnd AND NOT 31); r1 contains a pointer to the last line

lockLoop:
        mcr p15, 0, r0, c9, c1, 0; lock next line of code into I-Cache
        cmp r0, r1         ; are we done yet?
        add r0, r0, #32    ; advance pointer to next line
        bne lockLoop       ; if not done, do the next line
```

## 4.3.5 Unlocking Instructions in the Instruction Cache

The Intel® XScale™ core provides a global unlock command for the instruction cache. There is no unlock function for individual lines in the cache.

Writing to coprocessor 15, register 9 unlocks all the locked lines in the instruction cache and leaves them valid. These lines then become available for the round-robin replacement algorithm. (See Table 7-14, “Cache Lockdown Functions” on page 7-11 for the exact command.)

# Branch Target Buffer 5

The Intel® XScale™ core uses dynamic branch prediction to reduce the penalties associated with changing the flow of program execution. The Intel® XScale™ core 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 for those optimizing their code for performance. An understanding of the branch target buffer is needed in this case so that code can be 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 5-1 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 5-1. BTB Entry**

| TAG | DATA | |
|---|---|---|
| Branch Address[31:9,1] | Target Address[31:1] | History Bits[1:0] |

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

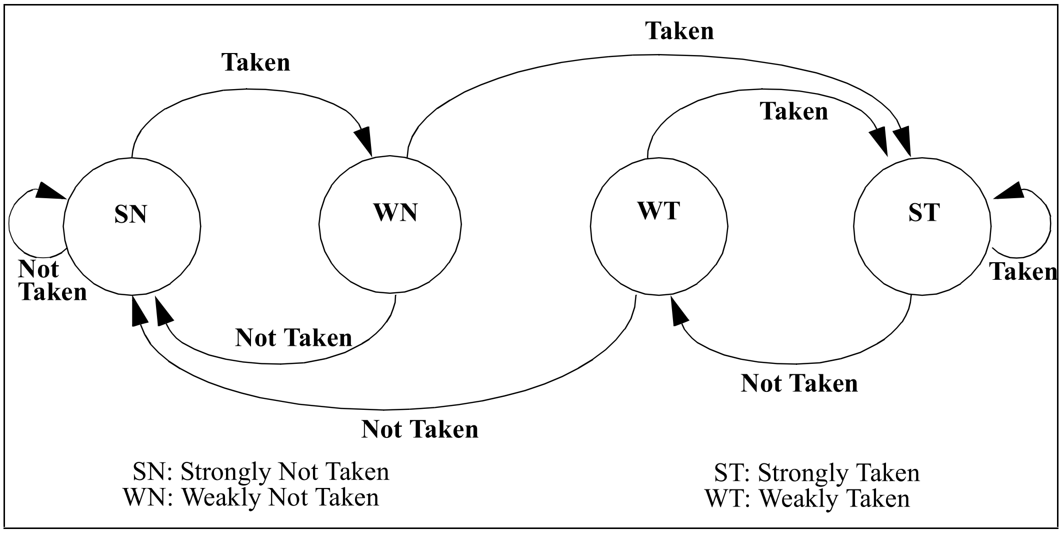
Bit[1] of the branch address is included in the tag comparison in order to support Thumb execution. This organization means that two consecutive Thumb branch (B) instructions, with instruction address bits[8:2] the same, will contend for the same BTB entry. Thumb also requires 31 bits for the branch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

The history bits represent four possible prediction states for a branch entry in the BTB. Figure 5-2, “Branch History” shows these states along with the possible transitions. The initial state for branches stored in the BTB is Weakly-Taken (WT). 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 11, “Performance Considerations” describes which instructions are dynamically predicted by the BTB and the performance penalty for mispredicting a branch.

The BTB does not have to be managed explicitly by software, it is disabled by default after reset and is invalidated when the instruction cache is invalidated.

Figure 5-2. Branch History



## 5.1.1 Reset

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

## 5.1.2 Update Policy

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

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

The entry is then marked valid and the history bits are set to WT. If another valid branch exists at the same entry in the BTB, it will be evicted 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 5-2.

# 5.2 BTB Control

## 5.2.1 Disabling/Enabling

The BTB is always disabled with Reset. Software enables the BTB through the Control Register bit[11] in coprocessor 15 (see Section 7.2.2).

Before enabling or disabling the BTB, software must invalidate it (described in the following section). This action will ensure correct operation in case stale data is in the BTB. Software must not place any branch instruction between the code that invalidates the BTB and the code that enables/disables it.

## 5.2.2 Invalidation

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

1. Reset
2. Software can directly invalidate the BTB via a CP15, register 7 function. Refer to Section 7.2.7, “Register 7: Cache Functions” on page 7-9.
3. The BTB is invalidated when the Process ID Register is written.
4. The BTB is invalidated when the instruction cache is invalidated via CP15, register 7 functions.

# Data Cache 6

The Intel® XScale™ core data cache enhances performance by reducing the number of data accesses to and from external memory. There are two data cache structures in the Intel® XScale™ core, a 32 Kbyte data cache and a 2 Kbyte mini-data cache. An eight entry write buffer and a four entry fill buffer are also implemented to decouple the Intel® XScale™ core instruction execution from external memory accesses, which increases overall system performance.

## 6.1 Overviews

### 6.1.1 Data Cache Overview

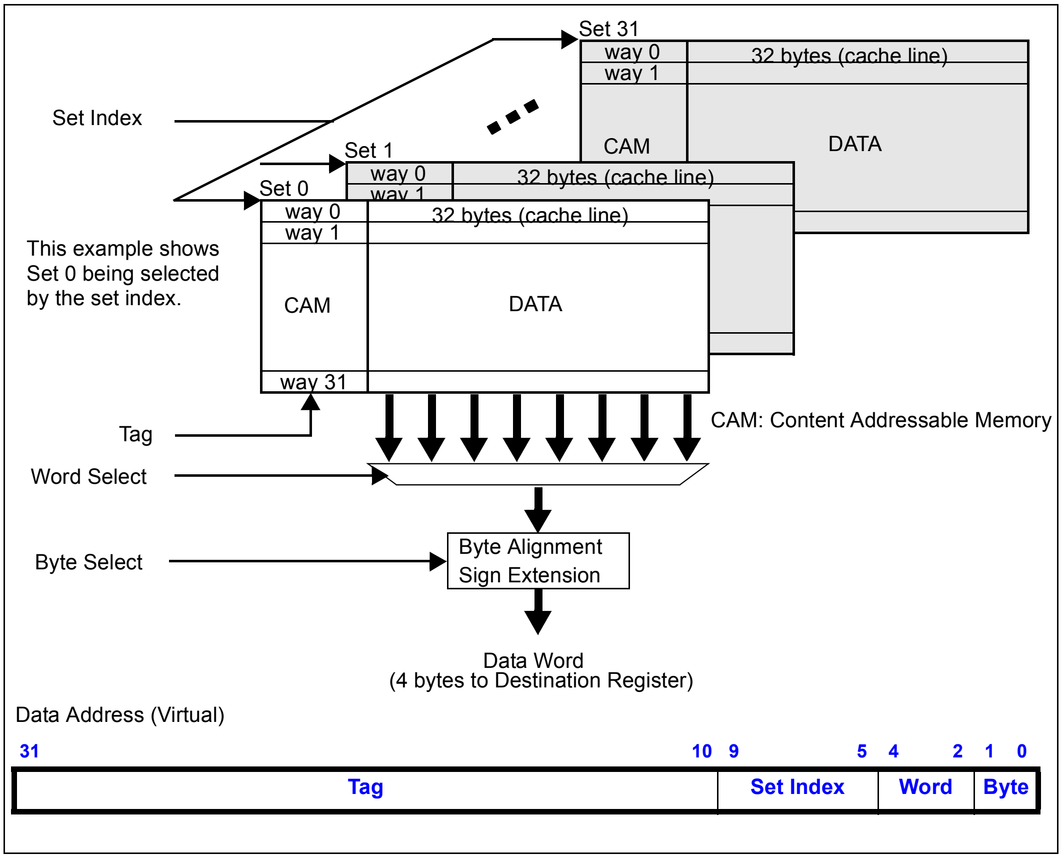
The data cache is a ***32-Kbyte, 32-way set associative cache***; this means there are 32 sets with each set containing 32 ways. Each way of a set contains 32 bytes (one cache line) and one valid bit. There also exist two dirty bits for every line, one for the lower 16 bytes and the other one for the upper 16 bytes. When a store hits the cache the dirty bit associated with that half of the cache line is set. The replacement policy is a round-robin algorithm and the cache also supports the ability to reconfigure each line as data RAM.

Figure 6-1 shows the cache organization and how the data address is used to access the cache.

Cache policies may be adjusted for particular regions of memory by altering page attribute bits in the MMU descriptor that controls that memory. See Section 3.2.3 for a description of these bits.

The data cache is virtually addressed and virtually tagged. It supports write-back and write-through caching policies. The data cache always allocates a line in the cache when a cacheable read miss occurs and will allocate a line into the cache on a cacheable write miss when write allocate is specified by its page attribute. Page attribute bits determine whether a line gets allocated into the data cache or mini-data cache.

Figure 6-1. Data Cache Organization



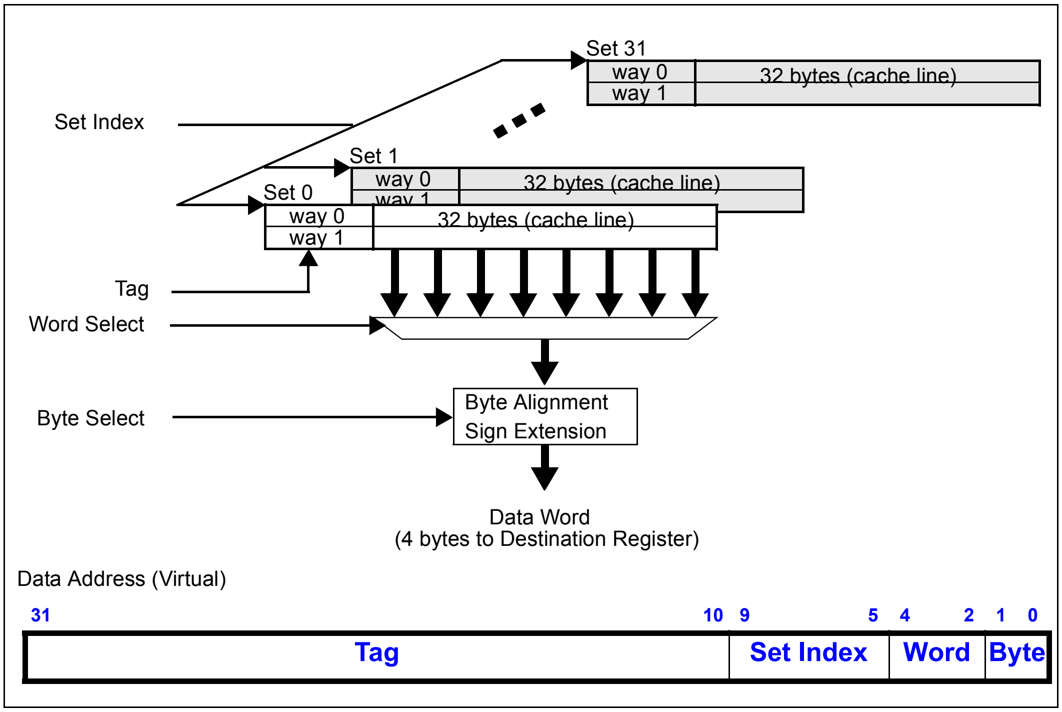
## 6.1.2 Mini-Data Cache Overview

The mini-data cache is a ***2-Kbyte, 2-way set associative cache***; this means there are 32 sets with each set containing 2 ways. Each way of a set contains 32 bytes (one cache line) and one valid bit. There are also 2 dirty bits for every line, one for the lower 16 bytes and the other one for the upper 16 bytes. When a store hits the cache the dirty bit associated with that half of the cacheline is set. The replacement policy is a round-robin algorithm.

Figure 6-2 shows the cache organization and how the data address is used to access the cache.

The mini-data cache is virtually addressed and virtually tagged and supports the same caching policies as the data cache. However, lines can not be locked into the mini-data cache.

Figure 6-2. Mini-Data Cache Organization



## 6.1.3 Write Buffer and Fill Buffer Overview

The Intel® XScale™ core employs an eight entry write buffer, each entry containing 16 bytes. Stores to external memory are first placed in the write buffer and subsequently taken out when the bus is available.

The write buffer supports the coalescing of multiple store requests to external memory where those stores are to a common 16-byte aligned address location. A store to memory may coalesce with any of the preceding eight entries so long as the store is to a bufferable memory page.

The fill buffer holds the external memory request information for a data cache or mini-data cache fill or non-cacheable read request. Up to four 32-byte read request operations can be outstanding in the fill buffer before the Intel® XScale™ core needs to stall.

The fill buffer has been augmented with a four entry pend buffer that captures data memory requests to outstanding fill operations. Each entry in the pend buffer contains enough data storage to hold one 32-bit word, specifically for store operations. Cacheable load or store operations that hit an entry in the fill buffer get placed in the pend buffer and are completed when the associated fill completes. Any entry in the pend buffer can be pended against any of the entries in the fill buffer; multiple entries in the pend buffer can be pended, or postponed, awaiting a particular entry in the fill buffer to complete.

Pended operations complete in program order.

## 6.2 Data Cache and Mini-Data Cache Operation

The following refers to the data cache and mini-data cache as one cache (data/mini-data) since their behavior is the same when accessed.

### 6.2.1 Operation When Caching is Enabled

When the data/mini-data cache is enabled for an access, the data/mini-data cache compares the address of the request against the addresses of data that it is currently holding. If 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 may also be written to external memory if write-through caching is specified for that area of memory. If 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. These are described in Section 6.2.3.2, "Read Miss Policy" and Section 6.2.3.3, "Write Miss Policy".

### 6.2.2 Operation When Data Caching is Disabled

The data/mini-data cache is still accessed even when it is disabled. If a load hits the cache it will return the requested data to the destination register. If a store hits the cache, the data is written into the cache.

Any access that misses the cache will not allocate a line in the cache when it's disabled, even if the MMU is enabled and the memory region's cacheability attribute is set. Any data reads or writes that miss in the cache will be directed to memory as controlled by the MMU. If both the data cache and MMU are disabled then cache misses will be issued as cycles on the application processor internal memory bus.

Disabling the cache prevents cache line refilling, but not the cache lookup from the processor.

### 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 cacheable attribute is set in the descriptor for the accessed address
- and the data/mini-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") load operation misses the data cache:

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

   If so, the current request is placed in the pending buffer and waits until the previously requested fill completes, after which it accesses the cache again, to obtain the request data and returns it to the destination register.

If there is no outstanding fill request for that line, the current load request is placed in the fill buffer and a 32-byte external memory read request is made. If the pending buffer or fill buffer is full, the Intel® XScale™ core will stall until an entry is available.

2. A line is allocated in the cache to receive the 32-bytes of fill data. The line selected is determined by the round-robin pointer (see Section 6.2.4, “Round-Robin Replacement Algorithm”). The line chosen may contain a valid line previously allocated in the cache. In this case both dirty bits are examined. Any half cache lines with a dirty bit that’s asserted will be written back to external memory as a four word burst operation.
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.
4. As data returns from external memory it is written to the cache into the previously allocated line.

A load operation that misses the cache and is NOT cacheable makes a request from external memory for the exact data size of the original load request. For example, **LDRH** requests exactly two bytes from external memory, **LDR** requests 4 bytes from external memory, etc. This request is placed in the fill buffer until, the data is returned from external memory, which is then forwarded back to the destination register(s).

For the PXA255 processor, the size of a data load depends also on the memory bank addressed in the access. For example, all 32-bit wide SDRAM reads are bursts of 4 words. All loads from this SDRAM generate a read of 4 words, despite that for uncacheable loads only the object the core requests will be used.

### 6.2.3.3 Write Miss Policy

A write operation that misses the cache will request a 32-byte cache line from external memory if the access is cacheable and write allocation is specified in the page. In this case the following sequence of events occur:

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

   If so, the current request is placed in the pending buffer and waits until the previously requested fill completes, after which it writes its data into the recently allocated cache line.

   If there is no outstanding fill request for that line, the current store request is placed in the fill buffer and a 32-byte external memory read request is made. If the pending buffer or fill buffer is full, the Intel® XScale™ core will stall until an entry is available.
2. The 32-bytes of data are returned back to the Intel® XScale™ core in the PXA255 processor in sequential order. Note that it does not matter, for performance reasons, which order the data is returned to the Intel® XScale™ core since the store operation has to wait until the entire line is written into the cache before it can complete.
3. When the entire 32-byte line has returned from external memory, a line is allocated in the cache, selected by the round-robin pointer (see Section 6.2.4, “Round-Robin Replacement Algorithm”). The line to be written into the cache may replace a valid line previously allocated in the cache. In this case both dirty bits are examined and if any are set, the four words associated with a dirty bit that’s asserted will be written back to external memory as a 4 word burst operation. This write operation will be placed in the write buffer.
4. The line is written into the cache along with the data associated with the store operation.

If the above condition for requesting a 32-byte cache line is not met, a write miss will cause a write request to external memory for the exact data size specified by the store operation, assuming the write request doesn’t coalesce with another write operation in the write buffer.

#### 6.2.3.4 Write-Back Versus Write-Through

The Intel® XScale™ core supports write-back caching or write-through caching, controlled through the MMU page attributes. When write-through caching is specified, all store operations are written to external memory even if the access hits the cache. This feature keeps the external memory coherent with the cache, i.e., no dirty bits are set for this region of memory in the data/mini-data cache. This however does not guarantee that the data/mini-data cache is coherent with external memory, which is dependent on the system level configuration, specifically if the external memory is shared by another master.

When write-back caching is specified, a store operation that hits the cache will not generate an immediate write to external memory, waiting instead for round-robin eviction or cache cleaning to write the data into external memory. Thus write-back caching is typically preferred for performance as it reduces external memory write traffic.

### 6.2.4 Round-Robin Replacement Algorithm

The line replacement algorithm for the data cache is round-robin. Each set in the data cache has a round-robin pointer that keeps track of the next line (in that set) to replace. The next line to replace in a set is the next sequential line after the last one that was just filled. For example, if the line for the last fill was written into way 5-set 2, the next line to replace for that set would be way 6. None of the other round-robin pointers for the other sets are affected when they are not the sets being accessed.

After reset, way 31 is pointed to by the round-robin pointer in each of the sets. Once a line is written into way 31, the round-robin pointer points to the first available way of a set, beginning with way 0 if no lines have been re-configured as data RAM in that particular set. Re-configuring lines as data RAM effectively reduces the available lines for cache updating. For example, if the first three lines of a set were re-configured, the round-robin pointer would point to the line at way 3 after it rolled over from way 31. Refer to Section 6.4, “Re-configuring the Data Cache as Data RAM” for more details on data RAM.

The mini-data cache follows the same round-robin replacement algorithm as the data cache except that there are only two lines the round-robin pointer can point to such that the round-robin pointer always points to the least recently filled line. A least recently used replacement algorithm is not supported because the purpose of the mini-data cache is to cache data that exhibits low temporal locality, i.e.,data that is placed into the mini-data cache is typically modified once and then written back out to external memory. Examples of data items that should be streamed through the mini data cache for improved system performance might be a audio or video bit stream such as MP3 or MPEG-4 data.

### 6.2.5 Parity Protection

The data cache and mini-data cache are 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/mini-data cache access, a data abort exception occurs. Before servicing the exception, hardware will set bit 10 of the Fault Status Register.

A data/mini-data cache parity error is an imprecise data abort, meaning R14_ABORT (+8) may not point to the instruction that caused the parity error. If the parity error occurred during a load, the targeted register may be updated with incorrect data.

A data abort due to a data/mini-data cache parity error may not be recoverable if the data address that caused the abort occurred on a line in the cache that has a write-back caching policy. Prior updates to this line may be lost; in this case the software exception handler should perform a “clean and clear” operation on the data cache, ignoring subsequent parity errors, and restart the offending process. The “clean & clear” operation is shown in Section 6.3.3.1.

### 6.2.6 Atomic Accesses

The **SWP** and **SWPB** instructions generate an atomic load and store operation to a common location, allowing a memory semaphore to be loaded and altered without interruption. These accesses may hit or miss the data/mini-data cache depending on the configuration of the cache, configuration of the MMU, and the page attributes.

The application processor guarantees that no other *on-chip* master (or process) divides a **SWP** or **SWPB** instruction. Note that there is no external bus lock pin, hence software coherency is compulsory if companion chips are to be required to access semaphores in external memory.

## 6.3 Data Cache and Mini-Data Cache Control

### 6.3.1 Data Memory State After Reset

After processor reset, both the data cache and mini-data cache are disabled, all valid bits are set to zero (invalid), and the round-robin bit points to way 31. Any lines in the data cache that were configured as data RAM before reset are changed back to cacheable lines after reset, i.e., there are 32 KBytes of data cache and zero bytes of data RAM.

### 6.3.2 Enabling/Disabling

The data cache and mini-data cache are enabled by setting bit 2 in coprocessor 15, register 1 (Control Register). See Chapter 7, “Configuration”, for a description of this register and others.

Example 6-1 shows code that enables the data and mini-data caches. Note that the MMU must be enabled to use the data cache.

**Example 6-1. Enabling the Data Cache**

```
enableDCache:

        MCR p15, 0, r0, c7, c10, 4; Drain pending data operations...
                           ; (see Chapter 7.2.8, Register 7: Cache functions)
        MRC p15, 0, r0, c1, c0, 0; Get current control register
        ORR r0, r0, #4      ; Enable D-Cache by setting ‘C’ (bit 2)
        MCR p15, 0, r0, c1, c0, 0; And update the Control register
```

### 6.3.3 Invalidate & Clean Operations

Individual entries can be cleaned and invalidated in the data cache and mini-data cache via coprocessor 15, register 7. Note that a line locked into the data cache remains locked even after it has been subjected to an invalidate-entry operation. This will leave an unusable line in the cache until a global unlock or reset has occurred. For this reason, do not use these commands on locked lines.

This same register also provides the command to invalidate the entire data cache and mini-data cache. Refer to Table 7-12, “Cache Functions” on page 7-9 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 they can be invalidated. This is accomplished by the Unlock Data Cache command found in Table 7-14, “Cache Lockdown Functions” on page 7-11.

#### 6.3.3.1 Global Clean and Invalidate Operation

A simple software routine is used to globally clean the data cache. It takes advantage of the line-allocate data cache operation, which allocates a line into the data cache. This allocation evicts any cache dirty data back to external memory. Example 6-2 on page 6-9 shows how data cache can be cleaned.

### Example 6-2. Global Clean Operation

```
; Global Clean/Invalidate THE DATA CACHE
; R1 contains the virtual address of a region of cacheable memory reserved for
; this clean operation
; R0 is the loop count; Iterate 1024 times which is the number of lines in the
; data cache

;; Macro ALLOCATE performs the line-allocation cache operation on the
;; address specified in register Rx.
;;
MACRO ALLOCATE Rx
    MCR P15, 0, Rx, C7, C2, 5
ENDM

MOV  R0, #1024
LOOP1:
ALLOCATE R1                ; Allocate a line at the virtual address
                           ; specified by R1.
ADD R1, R1, #32        ; Increment the address in R1 to the next cache line
SUBS R0, R0, #1        ; Decrement loop count
BNE LOOP1
;
; Clean the Mini-data Cache
; Can’t use line-allocate command, so cycle 2KB of unused data through.
; R2 contains the virtual address of a region of cacheable memory reserved for
; cleaning the Mini-data Cache
; R0 is the loop count; Iterate 64 times which is the number of lines in the
; Mini-data Cache.

MOV  R0, #64
LOOP2:
LDR R3,[R2],#32 ; Load and increment to next cache line
SUBS R0, R0, #1        ; Decrement loop count
BNE LOOP2
;

; Invalidate the data cache and mini-data cache
MCR P15, 0, R0, C7, C6, 0
;
```

The line-allocate operation does not require physical memory to exist at the virtual address specified by the instruction, since it does not generate a load/fill request to external memory. Also, the line-allocate operation does not set the 32 bytes of data associated with the line to any known value. Reading this data will produce unpredictable results.

The line-allocate command will not operate on the mini Data Cache, so system software must clean this cache by reading 2KByte of contiguous unused data into it. This data must be unused and reserved for this purpose so that it will not already be in the cache. It must reside in a page that is marked as mini Data Cache cacheable (see Section 2.3.2).

The time it takes to execute a global clean operation depends on the number of dirty lines in cache.

## 6.4 Re-configuring the Data Cache as Data RAM

Software has the ability to lock tags associated with 32-byte lines in the data cache, thus creating the appearance of data RAM. Any subsequent access to this line will always hit the cache unless it is invalidated. Once a line is locked into the data cache it is no longer available for cache allocation on a line fill. Up to 28 lines in each set can be reconfigured as data RAM, such that the maximum data RAM size is 28 Kbytes.

Hardware does not support locking lines into the mini-data cache; any attempt to do this will produce unpredictable results.

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

Code examples for these two applications are shown in Example 6-3 on page 6-11 and Example 6-4 on page 6-12. The difference between these two routines is that Example 6-3 on page 6-11 actually requests the entire line of data from external memory and Example 6-4 on page 6-12 uses the line-allocate operation to lock the tag into the cache. No external memory request is made, which means software can map any unallocated area of memory as data RAM. However, the line-allocate operation does validate the target address with the MMU, so system software must ensure that the memory has a valid descriptor in the page table.

Another item to note in Example 6-4 on page 6-12 is that the 32 bytes of data located in a newly allocated line in the cache must be initialized by software before it can be read. The line allocate operation does not initialize the 32 bytes and therefore reading from that line without first writing to it will produce unpredictable results.

Any line locked in the data cache could be written to by software. The locking is more a function of the address tag and not the data values. If the cache values represent constants shared by software then the memory area should be protected by the MMU. Using the MMU data abort exception, software can decide whether to prevent the write or allow the cache value to be overwritten. The exception handler would determine whether a write to memory is also required to overcome any coherency issues.

In both examples, the code drains the pending loads before and after locking data. This step ensures that outstanding loads do not end up in the wrong place -- either unintentionally locked into the cache or mistakenly left out. A drain operation has been placed after the operation that locks the tag into the cache. This drain ensures predictable results if a programmer tries to lock more than 28 lines in a set; the tag will get allocated in this case but not locked into the cache.

The data cache can only be unlocked by using the global unlock command See Table 7-14, “Cache Lockdown Functions” on page 7-11. The invalidate-entry command should not be issued to a locked line as this will render the line useless until a global unlock is issued.

### Example 6-3. Locking Data into the Data Cache

```
; R1 contains the virtual address of a region of memory to lock,
; configured with C=1 and B=1
; R0 is the number of 32-byte lines to lock into the data cache. In this
; example 16 lines of data are locked into the cache.
; MMU and data cache are enabled prior to this code.

MACRO DRAIN
    MCR P15, 0, R0, C7, C10, 4    ; drain pending loads and stores
ENDM

DRAIN

MOV  R2, #0x1
MCR  P15,0,R2,C9,C2,0          ; Put the data cache in lock mode
CPWAIT                     ; wait for effect (see Section 2.3.3)
MOV  R0, #16
LOOP1:
MCR  P15,0,R1,C7,C10,1         ; Write back the line if it’s dirty in the cache
MCR  P15,0,R1, C7,C6,1         ; Flush/Invalidate the line from the cache
PLD  [R1], #32                 ; Load and lock 32 bytes of data located at [R1]
                               ; into the data cache. Post-increment the address
                               ; in R1 to the next cache line.
DRAIN
SUBS R0, R0, #1; Decrement loop count
BNE LOOP1

            ; Turn off data cache locking
DRAIN
MOV  R2, #0x0
MCR  P15,0,R2,C9,C2,0          ; Take the data cache out of lock mode.
CPWAIT
```

### Example 6-4. Creating Data RAM

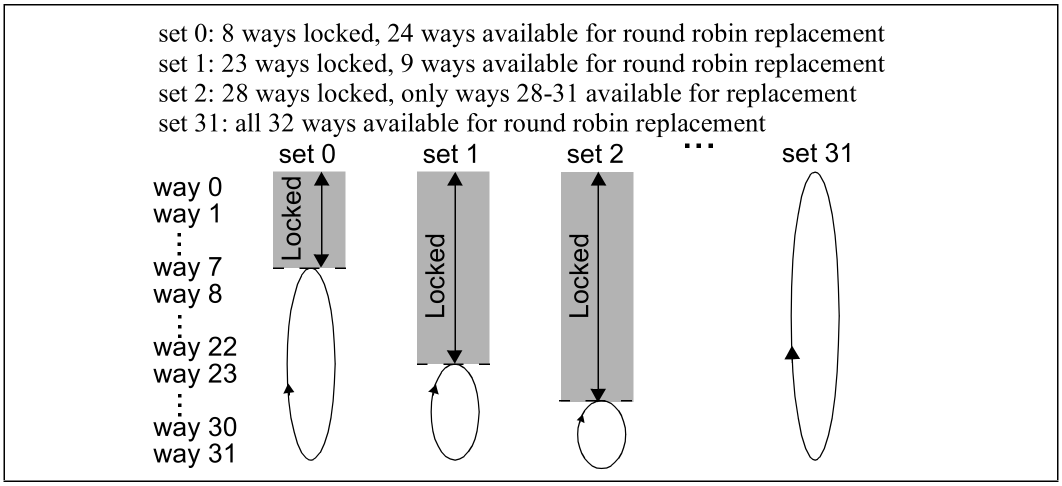
```
; R1 contains the virtual address of a region of memory to configure as data RAM,
; which is aligned on a 32-byte boundary.
; MMU is configured so that the memory region is cacheable.
; R0 is the number of 32-byte lines to designate as data RAM. In this example 16
; lines of the data cache are re-configured as data RAM.
; The inner loop is used to initialize the newly allocated lines
; MMU and data cache are enabled prior to this code.

MACRO ALLOCATE Rx
    MCR P15, 0, Rx, C7, C2, 5
ENDM

MACRO DRAIN
    MCR P15, 0, R0, C7, C10, 4    ; drain pending loads and stores
ENDM

DRAIN
MOV R4, #0x0
MOV R5, #0x0
MOV R2, #0x1
MCR P15,0,R2,C9,C2,0                     ; Put the data cache in lock mode
CPWAIT                    ; wait for effect (see Section 2.3.3)

MOV R0, #16
LOOP1:
ALLOCATE R1               ; Allocate and lock a tag into the data cache at
                          ; address [R1].
; initialize 32 bytes of newly allocated line
DRAIN
STRD R4, [R1],#4      ;
STRD R4, [R1],#4   ;
STRD R4, [R1],#4      ;
STRD R4, [R1],#4      ;

SUBS R0, R0, #1           ; Decrement loop count
BNE LOOP1
; Turn off data cache locking

DRAIN                             ; Finish all pending operations

MOV R2, #0x0
MCR P15,0,R2,C9,C2,0; Take the data cache out of lock mode.
CPWAIT
```

Tags can be locked into the data cache by enabling the data cache lock mode bit located in coprocessor 15, register 9. (See Table 7-14, “Cache Lockdown Functions” on page 7-11 for the exact command.) Once enabled, any new lines allocated into the data cache will be locked down.

Note that the **PLD** instruction will not affect the cache contents if it encounters an error while executing. For this reason, system software should ensure the memory address used in the **PLD** is correct. If this cannot be ascertained, replace the **PLD** with a **LDR** instruction that targets a scratch register.

Lines are locked into a set starting at way 0 and may progress up to way 27; which set a line gets locked into depends on the set index of the virtual address of the request. Figure 6-3, "Locked Line Effect on Round Robin Replacement" is an example of where lines of data may be locked into the cache along with how the round-robin pointer is affected.

**Figure 6-3. Locked Line Effect on Round Robin Replacement**



set 0: 8 ways locked, 24 ways available for round robin replacement
set 1: 23 ways locked, 9 ways available for round robin replacement
set 2: 28 ways locked, only ways 28-31 available for replacement
set 31: all 32 ways available for round robin replacement

Software can lock down data located at different memory locations. This may cause some sets to have more locked lines than others as shown in Figure 6-3.

Lines are unlocked in the data cache by performing an unlock operation. See Section 7.2.9, "Register 9: Cache Lock Down" on page 7-11 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. The Intel® XScale™ core will not refetch such data, which will result in it not being locked into the cache. If there is any doubt as to the location of the targeted memory data, the cache should be cleaned and invalidated to prevent this scenario. If the cache contains a locked region which the programmer wishes to lock again, then the cache must be unlocked before being cleaned and invalidated.

## 6.5 Write Buffer/Fill Buffer Operation and Control

The write buffer is always enabled which means stores to external memory will be buffered. The K bit in the Auxiliary Control Register (CP15, register 1) is a global enable/disable for allowing coalescing in the write buffer. When this bit disables coalescing, no coalescing will occur regardless of the value of the page attributes. If this bit enables coalescing, the page attributes X, C, and B are examined to see if coalescing is enabled for each region of memory.

Coalescing means that memory writes which occur with the same 16-byte aligned memory region can become one burst to external memory rather than distinct bus cycles. Merges can match with any write buffer entry, but they need to form contiguous data areas to coalesce. Byte writes may coalesce into halfwords, halfwords into words, or words into a multi-word burst to external memory. The Write Buffer attempts to replace distinct writes with burst writes to memory, greatly improving write performance to burst memory devices such as SDRAM.

All reads and writes to external memory occur in program order when coalescing is disabled in the write buffer. If coalescing is enabled in the write buffer, writes may occur out of program order to external memory. Program correctness is maintained in this case by comparing all store requests with all the valid entries in the fill buffer.

The write buffer and fill buffer support a drain operation, such that before the next instruction executes, all the Intel® XScale™ core data requests to external memory have completed. See Table 7-12, “Cache Functions” on page 7-9 for the exact command. Using this command, software running in a privileged mode can explicitly drain all buffered writes

Writes to a region marked non-cacheable/non-bufferable (page attributes C, B, and X all 0) will cause execution to stall until the write completes.

# Configuration 7

This chapter describes the System Control Coprocessor (CP15) and coprocessor 14 (CP14). CP15 configures the MMU, caches, buffers and other system attributes. Where possible, the definition of CP15 follows the definition of ARM* v5 products, see the *ARM* Architecture Reference Manual* for details. The PXA255 processor also include various extra device-specific configuration capabilities which are described here. CP14 contains the performance monitor registers and the trace buffer registers.

## 7.1 Overview

CP15 is accessed through **MRC** and **MCR** coprocessor instructions whose format is shown in Table 7-1, "MRC/MCR Format" on page 7-2. These instructions transfer data between ARM* registers and coprocessor registers. Coprocessor access is only allowed in a privileged mode. Any access to CP15 in user mode or with **LDC** or **STC** coprocessor instructions will cause an Undefined Instruction exception.

CP14 registers can be accessed through **MRC**, **MCR**, **LDC**, and **STC** coprocessor instructions and allowed only in privileged mode. **LDC** and **STC** transfer data between memory and coprocessor registers. See Table 7-2, "LDC/STC Format when Accessing CP14" on page 7-2. Any access to CP14 in user mode will cause an Undefined Instruction exception.

Coprocessors CP15 and CP14 on the Intel® XScale™ core do not support access via **CDP**, **MRRC**, or **MCRR** instructions. An attempt to access these coprocessors with these instructions will result in an Undefined Instruction exception.

Many of the MCR commands available in CP15 modify hardware state sometime after execution. A software sequence is available for those wishing to determine when this update occurs and can be found in Section 2.3.3, "Additions to CP15 Functionality" on page 2-10.

As in the Intel® SA-1110 product, and ARM* v5 Architecture specification, the Intel® XScale™ core includes an extra level of virtual address translation in the form of a PID (Process ID) register and associated logic. For a detailed description of this facility, see Section 7.2.11, "Register 13: Process ID" on page 7-12. Privileged code needs to be aware of this facility because when interacting with CP15 some addresses are modified by the PID and others are not.

An address that has yet to be modified by the PID 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). Non-privileged code always deals with VAs, while privileged code that programs CP15 occasionally needs to use MVAs.

The format of **MRC** and **MCR**, that move data to and from coprocessor registers, is shown in Table 7-1.

*cp_num* is defined for CP15, CP14 and CP0 on the Intel® XScale™ core. CP0 supports instructions specific for DSP and is described in Chapter 2, "Programming Model."

Unless otherwise noted, unused bits in coprocessor registers have unpredictable values when read. For compatibility with future implementations, software must program these bits as zero.

## Table 7-1. MRC/MCR 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 | opcode_1 | | | n | CRn | | | | Rd | | | | cp_num | | | | opcode_2 | | | 1 | CRm | | | |

| Bits | Description | Notes |
|---|---|---|
| 31:28 | **cond** - ARM* condition codes | - |
| 23:21 | **opcode_1** - Reserved | Must be programmed to zero for future compatibility |
| 20 | **n** - Read or write coprocessor register<br>0 = MCR<br>1 = MRC | - |
| 19:16 | **CRn** - specifies which coprocessor register | - |
| 15:12 | **Rd** - General Purpose Register, R0..R15 | - |
| 11:8 | **cp_num** - coprocessor number | The Intel® XScale™ core defines three coprocessors:<br>0b1111 = CP15<br>0b1110 = CP14<br>0x0000 = CP0 |
| 7:5 | **opcode_2** - Function bits | This field should be programmed to zero for future compatibility unless a value has been specified in the command. |
| 3:0 | **CRm** - Function bits | This field should be programmed to zero for future compatibility unless a value has been specified in the command. |

The format of **LDC** and **STC** for CP14 is shown in Table 7-2. **LDC** and **STC** follow the programming notes in the *ARM* Architecture Reference Manual* loading and storing coprocessor registers from/to memory. Access to CP15 with **LDC** and **STC** will cause an undefined exception as will access to coprocessors CP1 through CP13.

**LDC** and **STC** transfer a single 32-bit word between a coprocessor register and memory. These instructions do not allow the programmer to specify values for **opcode_1**, **opcode_2**, **or Rm**; those fields implicitly contain zero.

## Table 7-2. LDC/STC Format when Accessing CP14 (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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | | | | 1 | 1 | 0 | P | U | N | W | L | Rn | | | | CRd | | | | cp_num | | | | 8_bit_word_offset | | | | | | | |

| Bits | Description | Notes |
|---|---|---|
| 31:28 | **cond** - ARM* condition codes | - |
| 24:23,21 | **P, U, W** - specifies 1 of 3 addressing modes identified by addressing mode 5 in the *ARM Architecture Reference Manual*. | - |
| 22 | **N** - must be 0 for CP14 accesses. Setting this bit to 1 has will have an undefined effect. | |

### Table 7-2. LDC/STC Format when Accessing CP14 (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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | | | | 1 | 1 | 0 | P | U | N | W | L | Rn | | | | CRd | | | | cp_num | | | | 8_bit_word_offset | | | | | | | |

| Bits | Description | Notes |
|---|---|---|
| 20 | **L** - Load or Store<br>0 = STC<br>1 = LDC | - |
| 19:16 | **Rn** - specifies the base register | - |
| 15:12 | **CRd** - specifies the coprocessor register | - |
| 11:8 | **cp_num** - coprocessor number | The Intel® XScale™ core defines the following:<br>0b1110 = CP14<br>CP0-13 & CP15 = Undefined Exception |
| 7:0 | 8-bit word offset | - |

## 7.2 CP15 Registers

Table 7-3 lists the CP15 registers implemented in the Intel® XScale™ core.

### Table 7-3. CP15 Registers

| Register (CRn) | Opcode_2 | Access | Description |
|---|---|---|---|
| 0 | 0 | Read / Write-Ignored | ID |
| 0 | 1 | Read / Write-Ignored | Cache Type |
| 1 | 0 | Read / Write | Control |
| 1 | 1 | Read / Write | Auxiliary Control |
| 2 | 0 | Read / Write | Translation Table Base |
| 3 | 0 | Read / Write | Domain Access Control |
| 4 | 0 | Unpredictable | Reserved |
| 5 | 0 | Read / Write | Fault Status |
| 6 | 0 | Read / Write | Fault Address |
| 7 | 0 | Read-unpredictable / Write | Cache Operations |
| 8 | 0 | Read-unpredictable / Write | TLB Operations |
| 9 | 0 | Read / Write | Cache Lock Down |
| 10 | 0 | Read / Write | TLB Lock Down |
| 11 - 12 | 0 | Unpredictable | Reserved |
| 13 | 0 | Read / Write | Process ID (PID) |
| 14 | 0 | Read / Write | Breakpoint Registers |
| 15 | 0 | Read / Write | (CRm = 1) CP Access |

### 7.2.1 Register 0: ID & Cache Type Registers

Register 0 houses two read-only registers that are used for part identification: an ID register and a cache type register.

The ID Register is selected when *opcode_2*=0. This register returns the code for the application processor. Register 0 conforms with the values provided in the ARM* Architecture Reference Manual which should be consulted for alternate values representing other ARM* devices.

**Table 7-4. 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 | Core Gen | | | Core 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 = ARM* Version 5TE = 0b00000101 |
| 15:13 | Read / Write Ignored | Core Generation<br>0b001 = Intel® XScale™ core<br>This field reflects a specific set of architecture features supported by the core. If new features are added/deleted/ modified this field will change. |
| 12:10 | Read / Write Ignored | Core Revision:<br>0b000<br>This field reflects revisions of core generations. Differences may include errata that dictate different operating conditions, software work-arounds, etc. |
| 9:4 | Read / Write Ignored | Product Number<br>0b010000 |
| 3:0 | Read / Write Ignored | Product Revision<br>0b0000 for first stepping |

The Cache Type Register is selected when *opcode_2*=1 and describes the cache configuration of the Intel® XScale™ core. These values are device specific to the PXA255 processor, for the full set of potential values consult the *ARM* Architecture Reference Manual*.

**Table 7-5. 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 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 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 = 0b0101<br>The caches support locking, write back and round-robin replacement. They do not support address by index. |
| 24 | Read / Write Ignored | Harvard Cache = 1 |
| 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 = 0b101 = 32 |
| 14 | Read-as-Zero / Write Ignored | Reserved |
| 13:12 | Read / Write Ignored | Data cache line length = 0b10 = 8 words/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 = 0b101 = 32 |
| 2 | Read-as-Zero / Write Ignored | Reserved |
| 1:0 | Read / Write Ignored | Instruction cache line length = 0b10 = 8 words/line |

## 7.2.2 Register 1: Control & Auxiliary Control Registers

Register 1 is made up of two registers, one that is compliant with ARM* Version 5 and referred by *opcode_2* = 0x0, and the other which is specific to the Intel® XScale™ core is referred by *opcode_2* = 0x1. The latter is known as the Auxiliary Control Register.

The Exception Vector Relocation bit (bit 13 of the ARM* control register) allows the vectors to be mapped into high memory rather than their default location at address 0. This bit is readable and writable by software. If the MMU is enabled, the exception vectors will be accessed via the usual translation method involving the PID register (see Section 7.2.11, “Register 13: Process ID” on page 7-12) and the TLBs. To avoid automatic application of the PID to exception vector accesses, software may relocate the exceptions to high memory.

### Table 7-6. ARM* 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | | | V | I | Z | 0 | R | S | B | 1 | 1 | 1 | 1 | C | A | M |

reset value: writable bits set to 0

| Bits | Access | Description |
|---|---|---|
| 31:14 | 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/Disable (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 | Reserved |
| 9 | Read / Write | **ROM Protection (R bit)**<br>This selects the access checks performed by the memory management unit. See the *ARM Architecture Reference Manual* for more information. |
| 8 | Read / Write | **System Protection (S bit)**<br>This selects the access checks performed by the memory management unit. See the *ARM Architecture Reference Manual* for more information. |
| 7 | Read / Write | Big/Little Endian (B)<br>0 = Core Little-endian data operations<br>1 = Core Big-endian data operation |
| 6:3 | Read-as-One / Write-as-One | = 0b1111 |
| 2 | Read / Write | Data cache enable/disable (C)<br>0 = Disabled<br>1 = Enabled |
| 1 | Read / Write | Alignment fault enable/disable (A)<br>0 = Disabled<br>1 = Enabled |
| 0 | Read / Write | Memory management unit enable/disable (M)<br>0 = Disabled<br>1 = Enabled |

The mini-data cache attribute bits, in the Auxiliary Control Register, are used to control the allocation policy for the mini-data cache and whether it will use write-back caching or write-through caching.

The configuration of the mini-data cache must be setup before any data access is made that may be cached in the mini-data cache. Once data is cached, software must ensure that the mini-data cache has been cleaned and invalidated before the mini-data cache attributes can be changed.

### Table 7-7. Auxiliary 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | | | | | | | | | | | MD | | | | P | K |

reset value: writable bits set to 0

| Bits | Access | Description |
|---|---|---|
| 31:6 | Read-Unpredictable / Write-as-Zero | Reserved |
| 5:4 | Read / Write | Mini Data Cache Attributes (MD)<br>All configurations options for the Mini-data cache via these bits apply to mini-data cacheable accesses, stores are buffered in the write buffer and stores will coalesce in the write buffer as long as coalescing is globally enabled (**K** = 0 in this register).<br>0b00 = Write back, Read allocate<br>0b01 = Write back, Read/Write allocate<br>0b10 = Write through, Read allocate<br>0b11 = Unpredictable |
| 3:2 | Read-Unpredictable / Write-as-Zero | Reserved |
| 1 | Read / Write | Page Table Memory Attribute (P)<br>This field is undefined in the PXA255 processor implementation and must always be programmed as zero. |
| 0 | Read / Write | Write Buffer Coalescing Disable (K)<br>This bit globally disables the coalescing of all stores in the write buffer no matter what the value of the Cacheable and Bufferable bits are in the page table descriptors.<br>0 = Coalescing Enabled as Page Descriptors<br>1 = Coalescing Disabled |

## 7.2.3 Register 2: Translation Table Base Register

### Table 7-8. 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 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

reset value: unpredictable

| Bits | Access | Description |
|---|---|---|
| 31:14 | Read / Write | **Translation Table Base** - Physical address of the base of the first-level descriptor table |
| 13:0 | Read-unpredictable / Write-as-Zero | Reserved |

### 7.2.4 Register 3: Domain Access Control Register

Table 7-9. 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** - The meaning of each field can be found in the *ARM Architecture Reference Manual*. |

### 7.2.5 Register 5: Fault Status Register

The Fault Status Register (FSR) indicates which fault has occurred, which could be either a prefetch abort or a data abort. Bit 10 extends the encoding of the status field for prefetch aborts and data aborts. The definition of the extended status field is found in Section 2.3.4, "Event Architecture" on page 2-11. Bit 9 indicates that a debug event occurred and the exact source of the event is found in the debug control and status register (CP14, register 10). When bit 9 is set, the domain and extended status field are undefined.

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

Table 7-10. 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 is used to extend the encoding of the **Status** field, when there is a prefetch abort [See Table 2-13 on page 2-12] and when there is a data abort [See Table 2-14 on page 2-13]. |
| 9 | Read / Write | Debug Event (D)<br>This flag indicates a debug event has occurred and that the cause of the debug event is found in the MOE field of the debug control register (CP14, register 10) |
| 8 | Read-as-zero / Write-as-Zero | = 0 |
| 7:4 | Read / Write | **Domain** - Specifies which of the 16 domains was being accessed when a data abort occurred |
| 3:0 | Read / Write | **Status** - Used along with the **X**-bit above to determine the type of cycle type that generated the exception. See "Event Architecture" on page 2-11 |

## 7.2.6 Register 6: Fault Address Register

**Table 7-11. 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** - Contains the MVA of the data access that caused the memory abort |

## 7.2.7 Register 7: Cache Functions

All the cache functions defined in existing StrongARM* products appear here. The Intel® XScale™ core adds other functions as well. This register is write-only. Reads from this register, as with an MRC, have an undefined effect.

Disabling/enabling a cache has no effect on contents of the cache: valid data stays valid, locked items remain locked and accesses that hit in the cache will hit. To prevent cache hits after disabling the cache it is necessary to invalidate it. The way to prevent hits on the fill buffer is to drain it. All operations defined in Table 7-12 work regardless of whether the cache is enabled or disabled.

The Drain Write Buffer function not only drains the write buffer but also drains the fill buffer. The Intel® XScale™ core does not check permissions on addresses supplied for cache or TLB functions. Because only privileged software may execute these functions, full accessibility is assumed. Cache functions will not generate any of the following:

- translation faults
- domain faults
- permission faults

Since the Clean D Cache Line function reads from the data cache, it is capable of generating a parity fault. The other operations will not generate parity faults.

The invalidate instruction cache line command does not invalidate the BTB. If software invalidates a line from the instruction cache and modifies the same location in external memory, it needs to invalidate the BTB also. Not invalidating the BTB in this case will cause unpredictable results.

**Table 7-12. Cache Functions (Sheet 1 of 2)**

| Function | opcode_2 | CRm | Data | Instruction |
|---|---|---|---|---|
| Invalidate I&D cache & BTB | 0b000 | 0b0111 | Ignored | MCR p15, 0, Rd, c7, c7, 0 |
| Invalidate I cache & BTB | 0b000 | 0b0101 | Ignored | MCR p15, 0, Rd, c7, c5, 0 |
| Invalidate I cache line | 0b001 | 0b0101 | MVA | MCR p15, 0, Rd, c7, c5, 1 |
| Invalidate D cache | 0b000 | 0b0110 | Ignored | MCR p15, 0, Rd, c7, c6, 0 |
| Invalidate D cache line | 0b001 | 0b0110 | MVA | MCR p15, 0, Rd, c7, c6, 1 |
| Clean D cache line | 0b001 | 0b1010 | MVA | MCR p15, 0, Rd, c7, c10, 1 |

### Table 7-12. Cache Functions (Sheet 2 of 2)

| Function | opcode_2 | CRm | Data | Instruction |
|---|---|---|---|---|
| Drain Write (& Fill) Buffer | 0b100 | 0b1010 | Ignored | MCR p15, 0, Rd, c7, c10, 4 |
| Invalidate Branch Target Buffer | 0b110 | 0b0101 | Ignored | MCR p15, 0, Rd, c7, c5, 6 |
| Allocate Line in the Data Cache | 0b101 | 0b0010 | MVA | MCR p15, 0, Rd, c7, c2, 5 |

The line-allocate command allocates a tag into the data cache specified by bits [31:5] of Rd. If a valid dirty line (with a different MVA) already exists at this location it will be evicted. The 32 bytes of data associated with the newly allocated line are not initialized and therefore will generate unpredictable results if read.

This command may be used for cleaning the entire data cache on a context switch and also when re-configuring portions of the data cache as data RAM. In both cases, Rd is a virtual address that maps to some non-existent physical memory. When creating data RAM, software must initialize the data RAM before read accesses can occur. Specific uses of these commands can be found in Chapter 6, “Data Cache”.

Other items to note about the line-allocate command are:

- It forces all pending memory operations to complete.
- If the targeted cache line is already resident, this command has no effect.
- This command cannot be used to allocate a line in the mini Data Cache.
- The newly allocated line is not marked as “dirty”. However, if a valid store is made to that line it will be marked as “dirty” and will get written back to external memory if another line is allocated to the same cache location. This eviction will produce unpredictable results if the line-allocate command used a virtual address that mapped to non-existent memory.

  To avoid this situation, the line-allocate operation should only be used if one of the following can be guaranteed:

  — The virtual address associated with this command is not one that will be generated during normal program execution. This is the case when line-allocate is used to clean/invalidate the entire cache.

  — The line-allocate operation is used only on a cache region destined to be locked. When the region is unlocked, it must be invalidated before making another data access.

## 7.2.8 Register 8: TLB Operations

Disabling/enabling the MMU has no effect on the contents of either TLB: valid entries stay valid, locked items remain locked. To invalidate the TLBs the commands below are required. All operations defined in Table 7-13 work regardless of whether the cache is enabled or disabled.

This register is write-only. Reads from this register, as with an MRC, have an undefined effect.

### Table 7-13. TLB Functions

| Function | opcode_2 | CRm | Data | Instruction |
|---|---|---|---|---|
| Invalidate I&D TLB | 0b000 | 0b0111 | Ignored | MCR p15, 0, Rd, c8, c7, 0 |
| Invalidate I TLB | 0b000 | 0b0101 | Ignored | MCR p15, 0, Rd, c8, c5, 0 |
| Invalidate I TLB entry | 0b001 | 0b0101 | MVA | MCR p15, 0, Rd, c8, c5, 1 |
| Invalidate D TLB | 0b000 | 0b0110 | Ignored | MCR p15, 0, Rd, c8, c6, 0 |
| Invalidate D TLB entry | 0b001 | 0b0110 | MVA | MCR p15, 0, Rd, c8, c6, 1 |

## 7.2.9 Register 9: Cache Lock Down

Register 9 is used for locking down entries into the instruction cache and data cache. (The protocol for locking down entries can be found in Chapter 6, “Data Cache”.) Data can not be locked into the mini-data cache.

Table 7-14 shows the command for locking down entries in the instruction cache, instruction TLB, and data TLB. The cache entry to lock is specified by the virtual address in Rd. The data cache locking mechanism follows a different procedure than the instruction cache. The data cache is placed in lock down mode such that all subsequent fills to the data cache result in that line being locked in, as controlled by Table 7-15.

Lock/unlock operations on a disabled cache have an undefined effect. This register is write-only. Reads from this register, as with an MRC, have an undefined effect.

### Table 7-14. Cache Lockdown Functions

| Function | opcode_2 | CRm | Data | Instruction |
|---|---|---|---|---|
| Fetch and Lock I cache line | 0b000 | 0b0001 | MVA | MCR p15, 0, Rd, c9, c1, 0 |
| Unlock Instruction cache | 0b001 | 0b0001 | Ignored | MCR p15, 0, Rd, c9, c1, 1 |
| Read data cache lock register | 0b000 | 0b0010 | Read lock mode value | MRC p15, 0, Rd, c9, c2, 0 |
| Write data cache lock register | 0b000 | 0b0010 | Set/Clear lock mode | MCR p15, 0, Rd, c9, c2, 0 |
| Unlock Data Cache | 0b001 | 0b0010 | Ignored | MCR p15, 0, Rd, c9, c2, 1 |

### Table 7-15. Data Cache Lock 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | L |

reset value: writable bits set to 0

| Bits | Access | Description |
|---|---|---|
| 31:1 | Read-unpredictable / Write-as-Zero | Reserved |
| 0 | Read-unpredictable / Write | **Data Cache Lock Mode (L)**<br>0 = No locking occurs<br>1 = Any fill into the data cache while this bit is set gets locked in |

### 7.2.10 Register 10: TLB Lock Down

Register 10 is used for locking down entries into the instruction TLB, and data TLB. The protocol for locking down entries can be found in Chapter 3, "Memory Management". Lock/unlock operations on a TLB when the MMU is disabled have an undefined effect.

This register is write-only. Reads from this register, as with an MRC, have an undefined effect.

Table 7-16 shows the commands for locking down entries in the instruction TLB, and data TLB. The entry to lock is specified by the virtual address in Rd.

**Table 7-16. TLB Lockdown Functions**

| Function | opcode_2 | CRm | Data | Instruction |
|---|---|---|---|---|
| Translate and Lock I TLB entry | 0b000 | 0b0100 | MVA | MCR p15, 0, Rd, c10, c4, 0 |
| Translate and Lock D TLB entry | 0b000 | 0b1000 | MVA | MCR p15, 0, Rd, c10, c8, 0 |
| Unlock I TLB | 0b001 | 0b0100 | Ignored | MCR p15, 0, Rd, c10, c4, 1 |
| Unlock D TLB | 0b001 | 0b1000 | Ignored | MCR p15, 0, Rd, c10, c8, 1 |

### 7.2.11 Register 13: Process ID

The Intel® XScale™ core 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 PID register controls when virtual addresses are remapped and to what value.

The PID register is a 7-bit value that is ORed with bits 31:25 of the virtual address when they are zero. This effectively remaps the address to one of 128 "slots" in the 4 Gbytes of address space. If bits 31:25 are not zero, no remapping occurs. This feature is useful for operating system management of processes that may map to the same virtual address space. In those cases, the virtually mapped caches on the Intel® XScale™ core would not require invalidating on a process switch.

**Table 7-17. Accessing Process ID**

| Function | opcode_2 | CRm | Instruction |
|---|---|---|---|
| Read Process ID Register | 0b000 | 0b0000 | MRC p15, 0, Rd, c13, c0, 0 |
| Write Process ID Register | 0b000 | 0b0000 | MCR p15, 0, Rd, c13, c0, 0 |

Table 7-18. 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** - 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** - Must be programmed to zero for future compatibility |

### 7.2.11.1 The PID Register Affect On Addresses

All addresses generated and used by User Mode code are eligible for being translated using the PID register. Privileged code however, must be aware of certain special cases in which address generation does not follow the usual flow.

- The PID register is not used to remap the virtual address when accessing the Branch Target Buffer (BTB). Debug software reading the BTB needs to recognize addresses as MVAs. Any write to the PID register invalidates the BTB. This prevents any virtual addresses after the PID has changed from matching the incorrect Branch Target of any previously running process.
- A breakpoint address (see Section 7.2.12, “Register 14: Breakpoint Registers” on page 7-13) must be expressed as an MVA when written to the breakpoint register. This means the value of the PID must be combined appropriately with the address before it is written to the breakpoint register. All virtual addresses in translation descriptors (see Chapter 3, “Memory Management”) are MVAs.

## 7.2.12 Register 14: Breakpoint Registers

The Intel® XScale™ core contains two instruction breakpoint address registers (IBCR0 and IBCR1), one data breakpoint address register (DBR0), one configurable data mask/address register (DBR1), and one data breakpoint control register (DBCON). The Intel® XScale™ core also supports a 2K byte mini instruction cache for debugging and a 256 entry trace buffer that records program execution information. The registers to control the trace buffer are located in CP14.

Refer to Chapter 10, “Software Debug” for more information on these features of the Intel® XScale™ core.

Table 7-19. Accessing the Debug Registers (Sheet 1 of 2)

| Function | opcode_2 | CRm | Instruction |
|---|---|---|---|
| Read Instruction Breakpoint Register 0 (IBCR0) | 0b000 | 0b1000 | MRC p15, 0, Rd, c14, c8, 0 |
| Write IBCR0 | 0b000 | 0b1000 | MCR p15, 0, Rd, c14, c8, 0 |
| Read Instruction Breakpoint Register 1 (IBCR1) | 0b000 | 0b1001 | MRC p15, 0, Rd, c14, c9, 0 |
| Write IBCR1 | 0b000 | 0b1001 | MCR p15, 0, Rd, c14, c9, 0 |
| Read Data Breakpoint 0 (DBR0) | 0b000 | 0b0000 | MRC p15, 0, Rd, c14, c0, 0 |

### Table 7-19. Accessing the Debug Registers (Sheet 2 of 2)

| Function | opcode_2 | CRm | Instruction |
|---|---|---|---|
| Write DBR0 | 0b000 | 0b0000 | MCR p15, 0, Rd, c14, c0, 0 |
| Read Data Mask/Address Register (DBR1) | 0b000 | 0b0011 | MRC p15, 0, Rd, c14, c3, 0 |
| Write DBR1 | 0b000 | 0b0011 | MCR p15, 0, Rd, c14, c3, 0 |
| Read Data Breakpoint Control Register (DBCON) | 0b000 | 0b0100 | MRC p15, 0, Rd, c14, c4, 0 |
| Write DBCON | 0b000 | 0b0100 | MCR p15, 0, Rd, c14, c4, 0 |

## 7.2.13 Register 15: Coprocessor Access Register

Register 15: Coprocessor Access Register is selected when opcode_2 = 0 and CRm = 1.

This register controls access rights to all the coprocessors in the system except for CP15 and CP14. Both CP15 and CP14 can only be accessed in privilege mode. This register is accessed with an MCR or MRC with the CRm field set to 1.

This register controls access to CP0 on the application processors.

### Example 7-1. Disallowing access to CP0

```
;; The following code clears bit 0 of the CPAR.
;; This will cause the processor to fault if software
;; attempts to access CP0.

    LDR R0, =0x3FFE                 ; bit 0 is clear
    MCR P15, 0, R0, C15, C1, 0      ; move to CPAR
    CPWAIT                          ; wait for effect See Section 2.3.3
```

### Table 7-20. Coprocessor Access Register (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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | 0 | 0 | CP13 | CP12 | CP11 | CP10 | CP9 | CP8 | CP7 | CP6 | CP5 | CP4 | CP3 | CP2 | CP1 | CP0 |

reset value: 0x0000_0000

| Bits | Access | Description |
|---|---|---|
| 31:16 | Read-unpredictable / Write-as-Zero | **Reserved** - Should be programmed to zero for future compatibility |

### Table 7-20. Coprocessor Access Register (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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | 0 | 0 | CP13 | CP12 | CP11 | CP10 | CP9 | CP8 | CP7 | CP6 | CP5 | CP4 | CP3 | CP2 | CP1 | CP0 |

reset value: 0x0000_0000

| Bits | Access | Description |
|---|---|---|
| 15:14 | Read-as-Zero/Write-as-Zero | **Reserved** - Should be programmed to zero for future compatibility |
| 13:1 | Read / Write | **Coprocessor Access Rights**-<br>Each bit in this field corresponds to the access rights for each coprocessor. Only CP0 has any effect on the application processors CP1-CP13 must always be written as zero |
| 0 | Read / Write | **Coprocessor Access Rights**-<br>This bit corresponds to the access rights for CP0.<br>0 = Access denied. Any attempt to access the corresponding coprocessor will generate an Undefined exception, even in privileged modes.<br>1 = Access allowed. Includes read and write accesses. |

A typical use for this register is for an operating system to control resource sharing among applications. All applications can be denied access to CP0 by clearing the appropriate coprocessor bit in the Coprocessor Access Register. An application may request the use of the accumulator in CP0 by issuing an access to the resource, which will result in an undefined exception. The operating system may grant access to this coprocessor by setting the appropriate bit in the Coprocessor Access Register and return to the application where the access is retried. Sharing resources among different applications requires a state saving mechanism.

Two possibilities are:

- The operating system, during a context switch, could save the state of the coprocessor if the last executing process had access rights to the coprocessor.
- The operating system, during a request for access, saves off the old coprocessor state with the last process to have access to it.

Under both scenarios, the OS needs to restore state when a request for access is made. This means the OS has to maintain a list of what processes are modifying CP0 and their associated state.

A system programmer making this OS change should include code for coprocessors CP0 through CP13. Although the PXA255 processor only supports CP0, future products may implement additional coprocessor functionality from CP1-CP13.

## 7.3 CP14 Registers

Table 7-21 lists the CP14 registers implemented in the Intel® XScale™ core.

Table 7-21. CP14 Registers

| Register (CRn) | Access | Description |
|---|---|---|
| 0-3 | Read / Write | Performance Monitoring Registers |
| 4-5 | Unpredictable | Reserved |
| 6-7 | Read / Write | Clock and Power Management |
| 8-15 | Read / Write | Software Debug |

## 7.3.1 Registers 0-3: Performance Monitoring

The performance monitoring unit contains a control register (PMNC), a clock counter (CCNT), and two event counters (PMN0 and PMN1). The format of these registers can be found in Chapter 8, "Performance Monitoring", along with a description on how to use the performance monitoring facility.

Opcode_2 and CRm must be zero.

Table 7-22. Accessing the Performance Monitoring Registers

| Function | CRn (Register #) | Instruction |
|---|---|---|
| Read PMNC | 0b0000 | MRC p14, 0, Rd, c0, c0, 0 |
| Write PMNC | 0b0000 | MCR p14, 0, Rd, c0, c0, 0 |
| Read CCNT | 0b0001 | MRC p14, 0, Rd, c1, c0, 0 |
| Write CCNT | 0b0001 | MCR p14, 0, Rd, c1, c0, 0 |
| Read PMN0 | 0b0010 | MRC p14, 0, Rd, c2, c0, 0 |
| Write PMN0 | 0b0010 | MCR p14, 0, Rd, c2, c0, 0 |
| Read PMN1 | 0b0011 | MRC p14, 0, Rd, c3, c0, 0 |
| Write PMN1 | 0b0011 | MCR p14, 0, Rd, c3, c0, 0 |

.

## 7.3.2 Registers 6-7: Clock and Power Management

These registers contain functions for managing the core clock and power.

Power management modes are supported through register 7. Two low power modes are supported that are entered upon executing the functions listed in Table 7-25. To enter any of these modes, write the appropriate data to CP14, register 7 (PWRMODE). Software may read this register, but since software only runs during ACTIVE mode, it will always read zeroes from the **M** field.

### Table 7-23. PWRMODE Register 7

| 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 |
|---|---|---|
| 31:2 | Read-unpredictable / Write-as-Zero | Reserved |
| 1:0 | Read / Write | **Mode (M)**<br>0 = ACTIVE<br>1 = Idle Mode<br>2 = Reserved<br>3 = Sleep Mode |

Software can change core clock frequency by writing to CP 14 register 6, CCLKCFG.

### Table 7-24. CCLKCFG Register 6

| 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 | **Core Clock Configuration (CCLKCFG)**<br>0b0001 - Enter Turbo Mode<br>0b001x - Enter Frequency Change Sequence (Turbo Mode bit may be set or cleared in the same write)<br>Other values are reserved |

### Table 7-25. Clock and Power Management valid operations

| Function | Data | Instruction |
|---|---|---|
| Enter Idle Mode | 1 | MCR p14, 0, Rd, c7, c0, 0 |
| Reserved | 2 | MCR p14, 0, Rd, c7, c0, 0 |
| Enter Sleep Mode | 3 | MCR p14, 0, Rd, c7, c0, 0 |
| Read CCLKCFG | ignored | MRC p14, 0, Rd, c6, c0, 0 |
| Write CCLKCFG | CCLKCFG value | MCR p14, 0, Rd, c6, c0, 0 |

## 7.3.3 Registers 8-15: Software Debug

Software debug is supported by address breakpoint registers (Coprocessor 15, register 14), serial communication over the JTAG interface and a trace buffer. Registers 8 and 9 are used for the serial interface and registers 10 through 13 support a 256 entry trace buffer. Register 14 and 15 are the debug link register and debug SPSR (saved program status register). These registers are explained in more detail in Chapter 10, "Software Debug".

Opcode_2 and CRm must be zero.

### Table 7-26. Accessing the Debug Registers

| Function | CRn (Register #) | Instruction |
|---|---|---|
| Read Transmit Debug Register (TX) | 0b1000 | MRC p14, 0, Rd, c8, c0, 0 |
| Write TX | 0b1000 | MCR p14, 0, Rd, c8, c0, 0 |
| Read Receive Debug Register (RX) | 0b1001 | MRC p14, 0, Rd, c9, c0, 0 |
| Write RX | 0b1001 | MCR p14, 0, Rd, c9, c0, 0 |
| Read Debug Control and Status Register (DCSR) | 0b1010 | MRC p14, 0, Rd, c10, c0, 0 |
| Write DCSR | 0b1010 | MCR p14, 0, Rd, c10, c0, 0 |
| Read Trace Buffer Register (TBREG) | 0b1011 | MRC p14, 0, Rd, c11, c0, 0 |
| Write TBREG | 0b1011 | MCR p14, 0, Rd, c11, c0, 0 |
| Read Checkpoint 0 Register (CHKPT0) | 0b1100 | MRC p14, 0, Rd, c12, c0, 0 |
| Write CHKPT0 | 0b1100 | MCR p14, 0, Rd, c12, c0, 0 |
| Read Checkpoint 1 Register (CHKPT1) | 0b1101 | MRC p14, 0, Rd, c13, c0, 0 |
| Write CHKPT1 | 0b1101 | MCR p14, 0, Rd, c13, c0, 0 |
| Read Transmit and Receive Debug Control Register (TXRXCTRL) | 0b1110 | MRC p14, 0, Rd, c14, c0, 0 |
| Write TXRXCTRL | 0b1110 | MCR p14, 0, Rd, c14, c0, 0 |

# Performance Monitoring 8

This chapter describes the performance monitoring facility of the Intel® XScale™ core. The events that are monitored provide performance information for compiler writers, system application developers and software programmers.

## 8.1 Overview

The Intel® XScale™ core hardware provides two 32-bit performance counters that allow two unique events to be monitored simultaneously. In addition, the Intel® XScale™ core implements a 32-bit clock counter that can be used in conjunction with the performance counters; its sole purpose is to count the number of core clock cycles which is useful in measuring total execution time.

The Intel® XScale™ core can monitor either occurrence events or duration events. When counting occurrence events, a counter is incremented each time a specified event takes place and when measuring duration, a counter counts the number of processor clocks that occur while a specified condition is true. If any of the 3 counters overflow, an IRQ or FIQ will be generated if it's enabled. Each counter has its own interrupt enable. The counters continue to monitor events even after an overflow occurs, until disabled by software.

Each of these counters can be programmed to monitor any one of various events.

To further augment performance monitoring, the Intel® XScale™ core clock counter can be used to measure the executing time of an application. This information combined with a duration event can feedback a percentage of time the event occurred with respect to overall execution time.

Each of the three counters and the performance monitoring control register are accessible through Coprocessor 14 (CP14), registers 0-3. Refer to Section 7.3.1, "Registers 0-3: Performance Monitoring" on page 7-16 for more details on accessing these registers with **MRC**, **MCR**, **LDC**, and **STC** coprocessor instructions. Access is allowed in privileged mode only.

## 8.2 Clock Counter (CCNT; CP14 - Register 1)

The format of CCNT is shown in Table 8-1. The clock counter is reset to '0' by Performance Monitor Control Register (PMNC) or can be set to a predetermined value by directly writing to it. It counts core clock cycles. When CCNT reaches its maximum value 0xFFFF_FFFF, the next clock cycle will cause it to roll over to zero and set the overflow flag (bit 10) in PMNC. An IRQ or FIQ will be reported if it is enabled via bit 6 in the PMNC register.

The CCNT register continues running in DEBUG mode, yet will become unpredictable if the Power Mode register, see Section 7.3.2, "Registers 6-7: Clock and Power Management" on page 7-16 is written as non-ACTIVE.

Table 8-1. 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** - Reset to ‘0’ by PMNC register. When the clock counter reaches its maximum value 0xFFFF_FFFF, the next cycle will cause it to roll over to zero and generate an IRQ or FIQ if enabled. |

## 8.3 Performance Count Registers (PMN0 - PMN1; CP14 - Register 2 and 3, Respectively)

There are two 32-bit event counters; their format is shown in Table 8-2. The event counters are reset to ‘0’ by the PMNC register or can be set to a predetermined value by directly writing to them. When an event counter reaches its maximum value 0xFFFF_FFFF, the next event it needs to count will cause it to roll over to zero and set the overflow flag (bit 8 or 9) in PMNC. An IRQ or FIQ interrupt will be reported if it is enabled via bit 4 or 5 in the PMNC register.

Table 8-2. Performance Monitor Count Register (PMN0 and PMN1)

| 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** - Reset to ‘0’ by PMNC register. When an event counter reaches its maximum value 0xFFFF_FFFF, the next event it needs to count will cause it to roll over to zero and generate an IRQ interrupt if enabled. |

### 8.3.1 Extending Count Duration Beyond 32 Bits

To increase the monitoring duration, software can extend the count duration beyond 32 bits by counting the number of overflow interrupts each 32-bit counter generates. This can be done in the interrupt service routine (ISR) where an increment to some memory location every time the interrupt occurs will enable longer durations of performance monitoring. This intrudes upon program execution but is typically negligible, comparing the ISR execution time in the order of tens of cycles to the $2^{32}$ cycles it takes to generate an overflow interrupt.

## 8.4 Performance Monitor Control Register (PMNC)

The performance monitor control register (PMNC) is a coprocessor register that:

- controls which events PMN0 and PMN1 will monitor

- detects which counter overflowed
- enables/disables interrupt reporting
- resets all counters to zero
- and enables the entire mechanism

Table 8-3 shows the format of the PMNC register.

### Table 8-3. Performance Monitor Control Register (CP14, register 0) (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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | evtCount1 | | | | | | | | evtCount0 | | | | | | | | | flag | | | | inten | | | D | C | P | E |

**reset value:** E **and** inten **are 0, others unpredictable**

| Bits | Access | Description |
|---|---|---|
| 31:28 | Read-unpredictable / Write-as-0 | Reserved |
| 27:20 | Read / Write | **Event Count1** - identifies the source of events that PMN1 counts. See Table 8-4 for a description of the values this field may contain. |
| 19:12 | Read / Write | **Event Count0** - identifies the source of events that PMN0 counts. See Table 8-4 for a description of the values this field may contain. |
| 11 | Read-unpredictable / Write-as-0 | Reserved |
| 10:8 | Read / Write | **Overflow/Interrupt Flag** - identifies which counter overflowed<br>Bit 10 = clock counter overflow flag<br>Bit 9 = performance counter 1 overflow flag<br>Bit 8 = performance counter 0 overflow flag<br>Read Values:<br>0 = no overflow<br>1 = overflow has occurred<br>Write Values:<br>0 = no change<br>1 = clear this bit |
| 7 | Read-unpredictable / Write-as-0 | Reserved |
| 6:4 | Read / Write | **Interrupt Enable** - used to enable/disable interrupt reporting for each counter<br>Bit 6 = clock counter interrupt enable<br>0 = disable interrupt<br>1 = enable interrupt<br>Bit 5 = performance counter 1 interrupt enable<br>0 = disable interrupt<br>1 = enable interrupt<br>Bit 4 = performance counter 0 interrupt enable<br>0 = disable interrupt<br>1 = enable interrupt |
| 3 | Read / Write | **Clock Counter Divider (D)** -<br>0 = CCNT counts every processor clock cycle<br>1 = CCNT counts every 64th processor clock cycle |

### Table 8-3. Performance Monitor Control Register (CP14, register 0) (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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | evtCount1 | | | | | | | | evtCount0 | | | | | | | | | flag | | | | inten | | | D | C | P | E |

reset value: E and inten are 0, others unpredictable

| Bits | Access | Description |
|---|---|---|
| 2 | Read-unpredictable / Write | **Clock Counter Reset (C)** -<br>0 = no action<br>1 = reset the clock counter to ‘0x0’ |
| 1 | Read-unpredictable / Write | **Performance Counter Reset (P)** -<br>0 = no action<br>1 = reset both performance counters to ‘0x0’ |
| 0 | Read / Write | **Enable (E)** -<br>0 = all 3 counters are disabled<br>1 = all 3 counters are enabled |

### 8.4.1 Managing the PMNC

An interrupt will be reported when a counter’s overflow flag is set and its associated interrupt enable bit is set in the PMNC register. The interrupt will remain asserted until software clears the overflow flag by writing a one to the flag that is set. Note that the PXA255 processor Interrupt Controller and the CPSR interrupt bit must be enabled in order for software to receive the interrupt.

The PMCR registers continue running in DEBUG mode, yet will become unpredictable if the Power Mode register, see Section 7.3.2, “Registers 6-7: Clock and Power Management” on page 7-16 is written as non-ACTIVE.

*Note:* The counters continue to record events even after they overflow.

## 8.5 Performance Monitoring Events

Table 8-4 lists events that may be monitored by the PMU. Each of the Performance Monitor Count Registers (PMN0 and PMN1) can count any listed event. Software selects which event is counted by each PMNx register by programming the evtCountx fields of the PMNC register.

### Table 8-4. Performance Monitoring Events (Sheet 1 of 2)

| Event Number (evtCount0 or evtCount1) | Event Definition |
|---|---|
| 0x0 | Instruction cache miss requires fetch from external memory. |
| 0x1 | Instruction cache cannot deliver an instruction. This could indicate an I-Cache miss or an ITLB miss. This event will occur every cycle in which the condition is present. |
| 0x2 | Stall due to a data dependency. This event will occur every cycle in which the condition is present. |
| 0x3 | Instruction TLB miss. |
| 0x4 | Data TLB miss. |

### Table 8-4. Performance Monitoring Events (Sheet 2 of 2)

| Event Number (evtCount0 or evtCount1) | Event Definition |
|---|---|
| 0x5 | Branch instruction executed, branch may or may not have changed program flow. |
| 0x6 | Branch mispredicted. (B and BL instructions only.) |
| 0x7 | Instruction executed. |
| 0x8 | Stall because the data cache buffers are full. This event will occur every cycle in which the condition is present. |
| 0x9 | Stall because the data cache buffers are full. This event will occur once for each contiguous sequence of this type of stall, regardless the length of the stall. |
| 0xA | Data cache accesses, including misses and uncached accesses, but not including Cache Operations (defined in Section 7.2.7) |
| 0xB | Data cache misses, including uncached accesses but not including Cache Operations (defined in Section 7.2.7) |
| 0xC | Data cache write-back. This event occurs once for each 1/2 line (four words) that are written back from the cache. |
| 0xD | Software changed the PC. This event occurs any time the PC is changed by software and there is not a mode change. For example, a **mov** instruction with PC as the destination will trigger this event. Executing a **swi** from User mode will not trigger this event, because it will incur a mode change. |
| all others | Reserved, unpredictable results |

Some typical combination of counted events are listed in this section and summarized in Table 8-5. In this section, we call such an event combination a *mode*.

### Table 8-5. Some Common Uses of the PMU

| Mode | PMNC.evtCount0 | PMNC.evtCount1 |
|---|---|---|
| Instruction Cache Efficiency | 0x7 (instruction count) | 0x0 (I-Cache miss) |
| Data Cache Efficiency | 0xA (D-Cache access) | 0xB (D-Cache miss) |
| Instruction Fetch Latency | 0x1 (I-Cache cannot deliver) | 0x0 (I-Cache miss) |
| Data/Bus Request Buffer Full | 0x8 (D-Buffer stall duration) | 0x9 (D-Buffer stall) |
| Stall/Writeback Statistics | 0x2 (data stall) | 0xC (D-Cache writeback) |
| Instruction TLB Efficiency | 0x7 (instruction count) | 0x3 (ITLB miss) |
| Data TLB Efficiency | 0xA (D-cache access) | 0x4 (DTLB miss) |

## 8.5.1 Instruction Cache Efficiency Mode

PMN0 totals the number of instructions that were executed, which does not include instructions fetched from the instruction cache that were never executed. This can happen if a branch instruction changes the program flow; the instruction cache may retrieve 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. Each of these requests loads 32 bytes at a time due to the instruction fetch buffers, even when the memory page is marked as uncached.

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 can be derived by dividing CCNT by PMN0, where CCNT was used to measure total execution time.

### 8.5.2 Data Cache Efficiency Mode

PMN0 totals the number of data cache accesses, which includes cacheable and non-cacheable accesses, mini-data cache access and accesses made to locations configured as data RAM.

Note that **STM** and **LDM** will each count as several accesses to the data cache depending on the number of registers specified in the register list. **LDRD** will register two accesses.

PMN1 counts the number of data cache and mini-data cache misses. Cache operations do not contribute to this count. See Section 7.2.7 for a description of these operations.

The common statistic derived from these two events is:

- Data cache miss-rate. This is derived by dividing PMN1 by PMN0.

### 8.5.3 Instruction Fetch Latency Mode

PMN0 accumulates the number of cycles when the instruction-cache is not able to deliver an instruction to the Intel® XScale™ core due to an instruction-cache miss or instruction-TLB miss. This event means that the processor core is stalled.

PMN1 counts the number of instruction fetch requests to external memory. Each of these requests loads 32 bytes at a time. This is the same event as measured in instruction cache efficiency mode and is included in this mode for convenience so that only one performance monitoring run is need.

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. If the average is high then the Intel® XScale™ core may be starved of memory access due to other bus traffic.
- *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.

### 8.5.4 Data/Bus Request Buffer Full Mode

The Data Cache has buffers available to service cache misses or uncacheable accesses. For every memory request that the Data Cache receives from the processor core a buffer is speculatively allocated in case an external memory request is required or temporary storage is needed for an unaligned access. If no buffers are available, the Data Cache will stall the processor core. How often the Data Cache stalls depends on the performance of the bus external to the Intel® XScale™ core (the internal bus inside the application processor) and what the memory access latency is for Data Cache miss requests to external memory. If the Intel® XScale™ core memory access latency is high, possibly due to starvation, these Data Cache buffers will become full. This performance monitoring mode is provided to see if the Intel® XScale™ core is being starved of the bus external to the Intel® XScale™ core.

PMN0 accumulates the number of clock cycles the processor is being stalled due to this condition and PMN1 monitors the number of times this condition occurs.

Statistics derived from these two events:

- *The average number of cycles the processor stalled on a data-cache access that may overflow the data-cache buffers*. This is calculated by dividing PMN0 by PMN1. This statistic lets you know if the duration event cycles are due to many requests or are attributed to just a few requests. If the average is high then the Intel® XScale™ core may be starved from accessing the application processor internal bus due to other bus activity, e.g. companion chip bus cycles.
- *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.

## 8.5.5 Stall/Writeback Statistics Mode

When an instruction requires the result of a previous instruction and that result is not yet available, the Intel® XScale™ core stalls in order to preserve the correct data dependencies. PMN0 counts the number of stall cycles due to data-dependencies. 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 should delay using the result of a load until it's 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 should delay using the result until it's available.
- ALU use penalty: there are a few isolated cases where back to back ALU operations may result in one cycle delay in the execution. These cases are defined in Chapter 11, "Performance Considerations"**.**

PMN1 counts the number of writeback operations emitted by the data cache. 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 (CP15, 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 can reschedule code to avoid these penalties when given the right optimization switches.
- Total number of data writeback requests to external memory can be derived solely with PMN1.

### 8.5.6 Instruction TLB Efficiency Mode

PMN0 totals the number of instructions that were executed, which does not include instructions that were translated by the instruction TLB and never executed. This can happen if a branch instruction changes the program flow; the instruction TLB may translate 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, which occur when there is a TLB miss. If the instruction TLB is disabled PMN1 will not increment.

Statistics derived from these two events:

- Instruction TLB miss-rate. This is derived by dividing PMN1 by PMN0.
- CPI (See Section 8.5.1) can be derived by dividing CCNT by PMN0, where CCNT was used to measure total execution time.

### 8.5.7 Data TLB Efficiency Mode

PMN0 totals the number of data cache accesses, which includes cacheable and non-cacheable accesses, mini-data cache access and accesses made to locations configured as data RAM.

Note that **STM** and **LDM** will each count as several accesses to the data TLB depending on the number of registers specified in the register list. **LDRD** will register two accesses.

PMN1 counts the number of data TLB table-walks, which occur when there is a TLB miss. If the data TLB is disabled PMN1 will not increment.

The statistic derived from these two events is:

- Data TLB miss-rate. This is derived by dividing PMN1 by PMN0.

## 8.6 Multiple Performance Monitoring Run Statistics

Even though only two events can be monitored at any given time, multiple performance monitoring runs can be done, capturing different events from different modes. For example, the first run could monitor the number of writeback operations (PMN1 of mode, Stall/Writeback) and the second run could monitor the total number of data cache accesses (PMN0 of mode, Data Cache Efficiency). From the results, a percentage of writeback operations to the total number of data accesses can be derived.

## 8.7 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 will generate an IRQ interrupt.

### Example 8-1. Configuring the Performance Monitor

```
; Configure PMNC for instruction cache efficiency
;   evtCount0 = 7, evtCount1 = 0, flag = 0x7 to clear outstanding overflows
;   inten = 0x7set all counters to trigger an interrupt on overflow
;   C = 1       reset CCNT register
;   P = 1       reset PMN0 and PMN1 registers
;   E = 1       enable counting
MOV  R0,#0x7777
MCR  P14,0,R0,C0,c0,0  ; write R0 to PMNC
; Counting begins
```

Counter overflow can be dealt with in the IRQ interrupt service routine as shown below:

### Example 8-2. Interrupt Handling

```
IRQ_INTERRUPT_SERVICE_ROUTINE:
; Assume that performance counting interrupts are the only IRQ in the system
MRC  P14,0,R1,C0,c0,0  ; read the PMNC register
BIC  R2,R1,#1          ; clear the enable bit
MCR  P14,0,R2,C0,c0,0  ; clear interrupt flag and disable counting
MRC  P14,0,R3,C1,c0,0  ; read CCNT register
MRC  P14,0,R4,C2,c0,0  ; read PMN0 register
MRC  P14,0,R5,C3,c0,0  ; read PMN1 register

<process the results>
SUBS PC,R14,#4         ; return from interrupt
```

As an example, assume the following values in CCNT, PMN0, PMN1 and PMNC:

### Example 8-3. Computing the Results

```
; Assume CCNT overflowed

CCNT = 0x0000,0020 ;Overflowed and continued counting
Number of instructions executed = PMN0 = 0x6AAA,AAAA
Number of instruction cache miss requests = PMN1 = 0x0555,5555
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.

# Test 9

The application processor Test Access Port (TAP) conforms to the IEEE Std. 1149.1 – 1990, IEEE Std. 1149.1a-1993, *Standard Test Access Port and Boundary-Scan Architecture.* Refer to this standard for any explanations not covered in this section. This standard is more commonly referred to as JTAG, an acronym for the Joint Test Action Group.

The JTAG interface on the application processor can be used as a hardware interface for software debugging of PXA255 systems. This interface is described in Chapter 10, "Software Debug."

The JTAG hardware and test features of the application processor are discussed in the following sections.

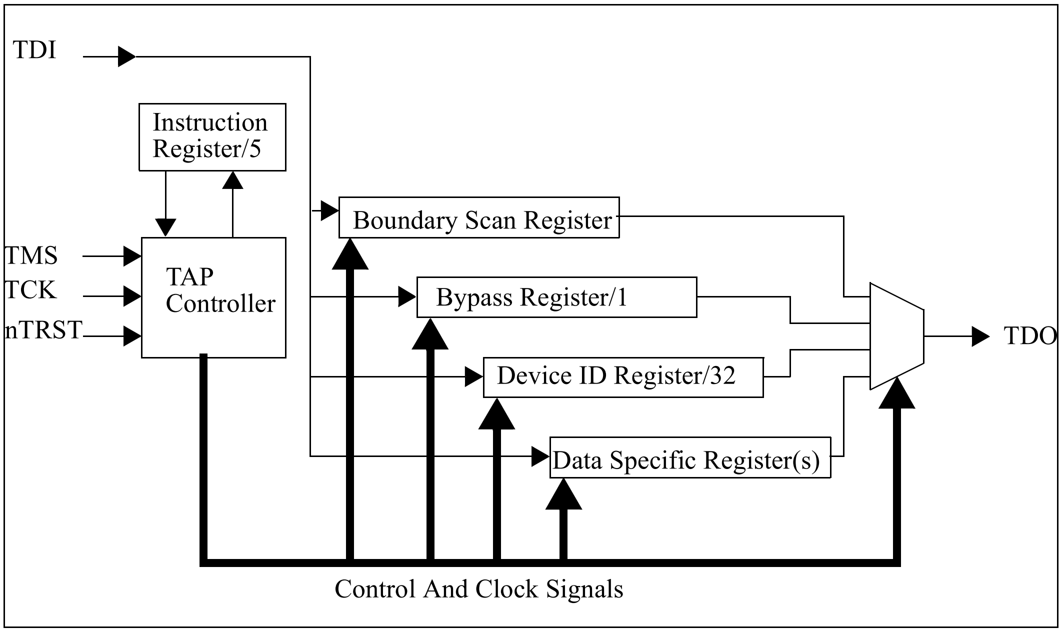
## 9.1 Boundary-Scan Architecture and Overview

The JTAG interface on the application processor provides a means of driving and sampling the external pins of the device irrespective of the core state. This feature is known as boundary scan.

Boundary scan permits testing of both the device's electrical connections to the circuit board and integrity of the circuit board connections between devices via linked JTAG interfaces. The interface intercepts external connections within the device via a boundary-scan cell, and each such "cell" is then connected together to form a serial shift register, called the boundary-scan register.

The boundary-scan test logic elements include the TAP pins, TAP Controller, instruction register, and a set of test data registers including: boundary-scan register, bypass register, device identification register, and data specific registers. This is shown in Figure 9-1.

**Figure 9-1. Test Access Port (TAP) Block Diagram**



The Test Access Port interface is controlled via five dedicated pins. These pins are described in Table 9-1.

Table 9-1. TAP Controller Pin Definitions

| Signal Name | Mnemonic | Type | Definition |
|---|---|---|---|
| Test Clock | TCK | Input | Clock input for the TAP controller, instruction register, and test data registers. |
| Test Mode Select | TMS | Input | Controls operation of the TAP controller. The TMS input is pulled high when not being driven. TMS is sampled on the rising edge of TCK. |
| Test Data In | TDI | Input | Serial data input to the instruction and test data registers. Data at TDI is sampled on the rising edge of TCK. TDI is pulled high when not being driven. |
| Test Data Out | TDO | Output | Serial data output. Data at TDO is clocked out on the falling edge of TCK. It provides an inactive (high-Z) state during non-shift operations to support parallel connection of TDO outputs at the board or module level. |
| Asynchronous Reset | nTRST | Input | Provides asynchronous initialization of the JTAG test logic. Assertion of this pin puts the TAP controller in the Test_Logic_Reset state. An external source must drive this signal from low to high for TAP controller operation. |

## 9.2 Reset

The boundary-scan interface includes a synchronous finite state machine, the TAP controller in Figure 9-1. In order to force the TAP controller into the correct state, a reset pulse must be applied to the nTRST pin.

*Note:* A clock on TCK is not necessary to reset the application processor.

To use the boundary-scan interface these points apply:

- During power-up only, drive nTRST from low to high either before or at the same time as nRESET.
- During power-up only, wait 10 µs after deassertion of nTRST before proceeding with any JTAG operation.
- Always drive the nBATT_FAULT and nVDD_FAULT pins high. An active low signal on either pin puts the device into sleep which powers down all JTAG circuitry.

The action of reset (either a pulse or a dc level) is:

- System mode is selected (the boundary-scan chain does NOT intercept any of the signals passing between the pads and the core.)
- **Idcode** instruction is selected. If TCK is pulsed, the contents of the ID register are clocked out of TDO.

If the boundary-scan interface is not to be used, then the nTRST pin may be tied permanently low or to the nRESET pin.

## 9.3 Instruction Register

The instruction register (IR) holds instruction codes shifted through the Test Data Input (TDI) pin.

Instruction codes in this register select the specific test operation performed and the test data register accessed. These instructions can be either mandatory or optional as set forth in the IEEE Std. 1149.1a-1993, user-defined, or private.

The instruction register is a 5-bit wide serial shift register. Data is loaded into the IR serially through the TDI pin clocked by the rising edge of TCK when the TAP controller is in the Shift_IR state.

The most significant bit of the IR is connected to TDI, and the least significant bit is connected to TDO. TDI is shifted into IR on the rising edge of TCK, as long as TMS remains asserted.

Upon activation of the nTRST pin, the latched instruction asynchronously changes to the **idcode** instruction.

### 9.3.1 Boundary-Scan Instruction Set

The application processor supports three mandatory public boundary scan instructions: **extest, sample/preload, bypass**. It also supports three optional public instructions: **idcode, clamp, highz**, four user-defined instructions: **dbgrx, ldic, dcsr, dbgtx**, and fourteen private instructions. The application processor does not support the optional public instructions **runbist, intest,** or **usercode**. Table 9-2 summarizes these boundary-scan instruction codes. Table 9-3 describes each of these instructions in detail.

Table 9-2. JTAG Instruction Codes

| Instruction Code | Instruction Name | Instruction Code | Instruction Name |
|---|---|---|---|
| $00000_2$ | extest | $01010_2$ | private |
| $00001_2$ | sample/preload | $01011_2$ | private |
| $00010_2$ | dbgrx | $01100_2$ | private |
| $00011_2$ | private | $01101_2$ | private |
| $00100_2$ | clamp | $01110_2$ - $01111_2$ | not used |
| $00101_2$ | private | $10000_2$ | dbgtx |
| $00110_2$ | not used | $10001_2$ - $11001_2$ | private |
| $00111_2$ | ldic | $11010_2$ - $11101_2$ | not used |
| $01000_2$ | highz | $11110_2$ | idcode |
| $01001_2$ | dcsr | $11111_2$ | bypass |

## Table 9-3. JTAG Instruction Descriptions

| Instruction / Requisite | Opcode | Description |
|---|---|---|
| **extest**<br>IEEE 1149.1<br>Required | $00000_2$ | The **extest** instruction initiates testing of external circuitry, typically board-level interconnects and off-chip circuitry. **extest** connects the Boundary-Scan register between TDI and TDO in the Shift_DR state only. When **extest** is selected, all output signal pin values are driven by values shifted into the Boundary-Scan register and may change only on the falling-edge of TCK in the Update_DR state. When **extest** is selected, all system input pin states must be loaded into the Boundary-Scan register on the rising-edge of TCK in the Capture_DR state. Values shifted into input latches in the Boundary-Scan register are never used by the processor's internal logic. |
| **sample**<br>IEEE 1149.1<br>Required | $00001_2$ | The **sample**/**preload** instruction performs two functions:<br>• When the TAP controller is in the Capture-DR state, the **sample** instruction occurs on the rising edge of TCK and provides a snapshot of the component's normal operation without interfering with that normal operation. The instruction causes Boundary-Scan register cells associated with outputs to **sample** the value being driven by the application processor.<br>• When the TAP controller is in the Update-DR state, the **preload** instruction occurs on the falling edge of TCK. This instruction causes the transfer of data held in the Boundary-Scan cells to the slave register cells. Typically the slave latched data is then applied to the system outputs by means of the **extest** instruction. |
| dbgrx | $00010_2$ | For Software Debug, see Section 10.10.5, "DBGRX JTAG Command" on page 10-20 |
| **clamp** | $00100_2$ | The **clamp** instruction allows the state of the signals driven from the application processor pins to be determined from the boundary-scan register while the Bypass register is selected as the serial path between TDI and TDO. Signals driven from the application processor pins will not change while the **clamp** instruction is selected. |
| **ldic** | $00111_2$ | For Software Debug, see Section 10.13.1, "LDIC JTAG Command" on page 10-30 |
| **highz** | $01000_2$ | The **highz** instruction floats all three-stateable output and in/out pins. Also, when this instruction is active, the Bypass register is connected between TDI and TDO. This register can be accessed via the JTAG Test-Access Port throughout the device operation. Access to the Bypass register can also be obtained with the **bypass** instruction**.** |
| dcsr | $01001_2$ | For Software Debug, see Section 10.3, "Debug Control and Status Register (DCSR)" on page 10-3 |
| dbgtx | $10000_2$ | For Software Debug, see Section 10.10.3, "DBGTX JTAG Command" on page 10-19 |
| **idcode**<br>IEEE 1149.1<br>Optional | $11110_2$ | The **idcode** instruction is used in conjunction with the device identification register. It connects the identification register between TDI and TDO in the Shift_DR state. When selected, **idcode** parallel-loads the hard-wired identification code (32 bits) on TDO into the identification register on the rising edge of TCK in the Capture_DR state.<br>Note: The device identification register is not altered by data being shifted in on TDI. |
| **bypass**<br>IEEE 1149.1<br>Required | $11111_2$ | The **bypass** instruction selects the Bypass register between TDI and TDO pins while in SHIFT_DR state, effectively bypassing the processor's test logic. $0_2$ is captured in the CAPTURE_DR state. While this instruction is in effect, all other test data registers have no effect on the operation of the system. Test data registers with both test and system functionality perform their system functions when this instruction is selected. |

## 9.4 Test Data Registers

The Test Data Registers are:

- Bypass Register
- Boundary-Scan Register
- Device Identification (ID) Code Register
- Data Specific Registers

### 9.4.1 Bypass Register

The Bypass register is a single-bit register that is selected as the path between TDI and TDO to allow the device to be bypassed during boundary-scan testing. This allows for more rapid movement of test data to and from other components on a board that are required to perform JTAG test operations.

When the **bypass**, **highz**, or **clamp** instruction is the current instruction in the instruction register, serial data is transferred from TDI to TDO in the Shift-DR state with a delay of one TCK cycle.

There is no parallel output from the bypass register.

A logic 0 is loaded from the parallel input of the bypass register in the Capture-DR state.

### 9.4.2 Boundary-Scan Register

The boundary-scan register consists of a serially connected set of cells around the periphery of the device at the interface between the core logic and the system input/output pads. This register can be used to isolate the pins from the core logic and then drive or monitor the system pins. The connected boundary-scan cells make up a shift-register.

The boundary-scan register is selected as the register to be connected between TDI and TDO only during the **sample/preload** and **extest** instructions. Values in the boundary-scan register are used, but are not changed, during the **clamp** instruction.

In the normal (system) mode of operation straight-through connections between the core logic and pins are maintained, and normal system operation is unaffected. Such is the case when the **sample/preload** instruction is selected.

In test mode when **extest** is the currently selected instruction, values can be applied to the output pins independently of the actual values on the input pins and core logic outputs. On the application processor, all of the boundary-scan cells include update registers with the exception of the nRESET_OUT and PWR_EN pins. In the case of the nRESET_OUT and PWR_EN pins, the contents of the scan latches are not placed on the pins. This is to prevent a scan operation from disabling power to the device and/or resetting external components.

The following pins are not part of the boundary-scan shift-register:

- PEXTAL
- PXTAL
- TEXTAL

- TXTAL
- XM
- XP
- YM
- YP
- REF
- The five TAP Controller pins

Also, JTAG operations cannot be performed in sleep, i.e. the nBATT_FAULT and nVDD_FAULT pins must always be driven high during JTAG operation.

The **extest** guard values should be clocked into the boundary-scan register (using the **sample/ preload** instruction) before the **extest** instruction is selected to ensure that known data is applied to the core logic during the test. These guard values should also be used when new EXTEST vectors are clocked into the boundary-scan register.

The values stored in the boundary-scan register after power-up are not defined. Similarly, the values previously clocked into the boundary-scan register are not guaranteed to be maintained across a JTAG reset (from forcing nTRST low or entering the Test Logic Reset state).

The PXA255 256-pin PBGA package boundary scan pin order is shown in Figure 9-2 on page 9-6.

### Figure 9-2. BSDL code for 256-MBGA package

```
-- A full BSDL file for this part is available from Intel
entity processor_jtag is
generic(PHYSICAL_PIN_MAP : string := "MBGA-256");
  port ( gpio        : inout bit_vector(80 DOWNTO 0);
         scl         : inout bit;
         sda         : inout bit;
         usb_n       : inout bit;
         usb_p       : inout bit;
         mmdat       : inout bit;
         mmcmd       : inout bit;
         md          : inout bit_vector(31 DOWNTO 0);
         pwr_en      : out bit;
         nreset_out  : out bit;
         ac_reset_n  : out bit;
         rdnwr       : out bit;
         sdclk_0     : out bit;
         sdclk_1     : out bit;
         sdclk_2     : out bit;
         sdcke       : out bit_vector(1 DOWNTO 0);
         nsdcs_0     : out bit;
         nsdcs_1     : out bit;
         nsdcs_2     : out bit;
         nsdcs_3     : out bit;
         dqm_0       : out bit;
         dqm_1       : out bit;
         dqm_2       : out bit;
         dqm_3       : out bit;
         nsdcas      : out bit;
nsdras       : out bit;
nwe          : out bit;
noe          : out bit;
ncs_0        : out bit;
ma           : out bit_vector(25 DOWNTO 0);
test         : in bit;
testclk      : in bit;
nvdd_fault   : in bit;
nbatt_fault  : in bit;
boot_sel     : in bit_vector(2 DOWNTO 0);
nreset       : in bit;
pextal       : out bit;
textal       : out bit;
yp           : in bit;
ym           : in bit;
xp           : in bit;
xm           : in bit;
ref          : in bit;
pxtal        : in bit;
txtal        : in bit;
tms          : in bit;
tck          : in bit;
tdi          : in bit;
tdo          : out bit;
ntrst        : in bit);
```

### 9.4.3 Device Identification (ID) Code Register

The Device Identification register is used to read the 32-bit device identification code. No programmable supplementary identification code is provided.

When the **idcode** instruction is current, the ID register is selected as the serial path between TDI and TDO.

The format of the ID register is as follows:

| 31 28 | 27 12 | 11 0 |
|---|---|---|
| **Version** | **Part Number** | **JEDEC Code** |

The high-order 4 bits of the ID register contains the version number of the silicon and changes with each new revision.

There is no parallel output from the ID register.

The 32-bit device identification code is loaded into the ID register from its parallel inputs during the CAPTURE-DR state.

### 9.4.4 Data Specific Registers

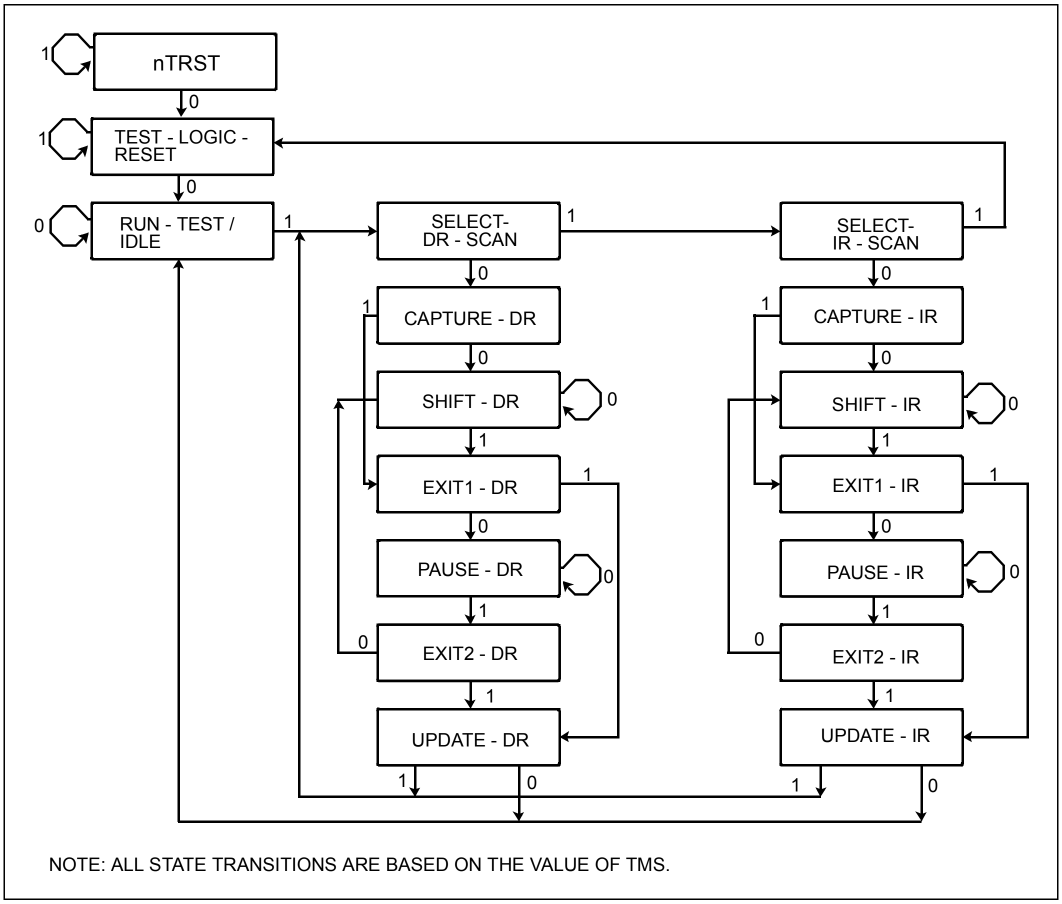
Data Specific Registers are used for the application processor instruction cache initialization and software debugging. For further information see Section 10.3, “Debug Control and Status Register (DCSR)” on page 10-3, Section 10.10.2, “SELDCSR JTAG Register” on page 10-17, Section 10.13.2, “LDIC JTAG Data Register” on page 10-31, Section 10.10.4, “DBGTX JTAG Register” on page 10-19 and Section 10.10.6, “DBGRX JTAG Register” on page 10-20.

## 9.5 TAP Controller

The TAP controller is a 16-state synchronous finite state machine that controls the sequence of test logic operations. The TAP can be controlled via a bus master. The bus master can be either automatic test equipment or a programmable logic device that interfaces to the Test Access Port (TAP). The TAP controller changes state only in response to a rising edge of TCK or power-up. The value of the test mode state (TMS) input signal at a rising edge of TCK controls the sequence of state changes. The TAP controller is automatically initialized on power–up. In addition, the TAP controller can be initialized by applying a high signal level on the TMS input for five TCK periods.

Behavior of the TAP controller and other test logic in each controller state is described in the following sub-sections. Figure 9-3 shows the state transitions that occur in the TAP controller. Note that all application processor digital signals participate in the boundary scan except the PWR_EN pin. This prevents a scan operation from turning off power to the application processor. For greater detail on the state machine and the public instructions, refer to IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture Document.

Figure 9-3. TAP Controller State Diagram



## 9.5.1 Test Logic Reset State

In this state, test logic is disabled to allow normal operation of the application processor. Test logic is disabled by loading the **idcode** register. No matter what the state of the controller, it enters Test-Logic-Reset state when the TMS input is held high (1) for at least five rising edges of TCK. The controller remains in this state while TMS is high. The TAP controller is also forced to enter this state by enabling nTRST.

If the controller exits the Test-Logic-Reset controller states as a result of an erroneous low signal on the TMS line at the time of a rising edge on TCK (for example, a glitch due to external interference), it returns to the test logic reset state following three rising edges of TCK with the TMS line at the intended high logic level. Test logic operation is such that no disturbance is caused to on-chip system logic operation as the result of such an error.

## 9.5.2 Run-Test/Idle State

The TAP controller enters the Run-Test/Idle state between scan operations. The controller remains in this state as long as TMS is held low. In the Run-Test/Idle state the **runbist** instruction is performed; the result is reported in the RUNBIST register. Instructions that do not call functions generate no activity in the test logic while the controller is in this state. The instruction register and all test data registers retain their current state. When TMS is high on the rising edge of TCK, the controller moves to the Select-DR-Scan state.

## 9.5.3 Select-DR-Scan State

The Select-DR-Scan state is a temporary controller state. The test data registers selected by the current instruction retain their previous state. If TMS is held low on the rising edge of TCK when the controller is in this state, the controller moves into the Capture-DR state and a scan sequence for the selected test data register is initiated. If TMS is held high on the rising edge of TCK, the controller moves into the Select-IR-Scan state.

The instruction does not change while the TAP controller is in this state.

## 9.5.4 Capture-DR State

When the controller is in this state and the current instruction is **sample**/**preload**, the Boundary-Scan register captures input pin data on the rising edge of TCK. Test data registers that do not have parallel input are not changed. Also if the **sample**/**preload** instruction is not selected while in this state, the Boundary-Scan registers retain their previous state.

The instruction does not change while the TAP controller is in this state.

If TMS is high on the rising edge of TCK, the controller enters the Exit1-DR. If TMS is low on the rising edge of TCK, the controller enters the Shift-DR state.

## 9.5.5 Shift-DR State

In this controller state, the test data register, which is connected between TDI and TDO as a result of the current instruction, shifts data one bit position nearer to its serial output on each rising edge of TCK. Test data registers that the current instruction selects but does not place in the serial path, retain their previous value during this state.

The instruction does not change while the TAP controller is in this state.

If TMS is high on the rising edge of TCK, the controller enters the Exit1-DR state. If TMS is low on the rising edge of TCK, the controller remains in the Shift-DR state.

## 9.5.6 Exit1-DR State

This is a temporary controller state. When the TAP controller is in the Exit1-DR state and TMS is held high on the rising edge of TCK, the controller enters the Update-DR state, which terminates the scanning process. If TMS is held low on the rising edge of TCK, the controller enters the Pause-DR state.

The instruction does not change while the TAP controller is in this state. All test data registers selected by the current instruction retain their previous value during this state.

### 9.5.7 Pause-DR State

The Pause-DR state allows the test controller to temporarily halt the shifting of data through the test data register in the serial path between TDI and TDO. The test data register selected by the current instruction retains its previous value during this state. The instruction does not change in this state.

The controller remains in this state as long as TMS is low. When TMS goes high on the rising edge of TCK, the controller moves to the Exit2-DR state.

### 9.5.8 Exit2-DR State

This is a temporary state. If TMS is held high on the rising edge of TCK, the controller enters the Update-DR state, which terminates the scanning process. If TMS is held low on the rising edge of TCK, the controller enters the Shift-DR state.

The instruction does not change while the TAP controller is in this state. All test data registers selected by the current instruction retain their previous value during this state.

### 9.5.9 Update-DR State

The Boundary-Scan register is provided with a latched parallel output. This output prevents changes at the parallel output while data is shifted in response to the **extest**, **sample**/**preload** instructions. When the Boundary-Scan register is selected while the TAP controller is in the Update-DR state, data is latched onto the Boundary-Scan register's parallel output from the shift-register path on the falling edge of TCK. The data held at the latched parallel output does not change unless the controller is in this state.

While the TAP controller is in this state, all of the test data register's shift-register bit positions selected by the current instruction retain their previous values.

The instruction does not change while the TAP controller is in this state.

When the TAP controller is in this state and TMS is held high on the rising edge of TCK, the controller enters the Select-DR-Scan state. If TMS is held low on the rising edge of TCK, the controller enters the Run-Test/Idle state.

### 9.5.10 Select-IR Scan State

This is a temporary controller state. The test data registers selected by the current instruction retain their previous state. In this state, if TMS is held low on the rising edge of TCK, the controller moves into the Capture-IR state and a scan sequence for the instruction register is initiated. If TMS is held high on the rising edge of TCK, the controller moves to the Test-Logic-Reset state.

The instruction does not change in this state.

### 9.5.11 Capture-IR State

When the controller is in the Capture-IR state, the shift register contained in the instruction register loads the fixed value $0001_2$ on the rising edge of TCK.

The test data register selected by the current instruction retains its previous value during this state. The instruction does not change in this state. While in this state, holding TMS high on the rising edge of TCK causes the controller to enter the Exit1-IR state. If TMS is held low on the rising edge of TCK, the controller enters the Shift-IR state.

## 9.5.12 Shift-IR State

When the controller is in this state, the shift register contained in the instruction register is connected between TDI and TDO and shifts data one bit position nearer to its serial output on each rising edge of TCK. The test data register selected by the current instruction retains its previous value during this state. The instruction does not change.

If TMS is held high on the rising edge of TCK, the controller enters the Exit1-IR state. If TMS is held low on the rising edge of TCK, the controller remains in the Shift-IR state.

## 9.5.13 Exit1-IR State

This is a temporary state. If TMS is held high on the rising edge of TCK, the controller enters the Update-IR state, which terminates the scanning process. If TMS is held low on the rising edge of TCK, the controller enters the Pause-IR state.

The test data register selected by the current instruction retains its previous value during this state.

The instruction does not change and the instruction register retains its state.

## 9.5.14 Pause-IR State

The Pause-IR state allows the test controller to temporarily halt the shifting of data through the instruction register. The test data registers selected by the current instruction retain their previous values during this state.

The instruction does not change and the instruction register retains its state.

The controller remains in this state as long as TMS is held low. When TMS goes high on the rising edges of TCK, the controller moves to the Exit2-IR state.

## 9.5.15 Exit2-IR State

This is a temporary state. If TMS is held high on the rising edge of TCK, the controller enters the Update-IR state, which terminates the scanning process. If TMS is held low on the rising edge of TCK, the controller enters the Shift-IR state.

This test data register selected by the current instruction retains its previous value during this state. The instruction does not change and the instruction register retains its state.

## 9.5.16 Update-IR State

The instruction shifted into the instruction register is latched onto the parallel output from the shift-register path on the falling edge of TCK. Once latched, the new instruction becomes the current instruction. Test data registers selected by the current instruction retain their previous values.

If TMS is held high on the rising edge of TCK, the controller enters the Select-DR-Scan state. If TMS is held low on the rising edge of TCK, the controller enters the Run-Test/Idle state.

# Software Debug 10

This chapter describes the software debug and related features implemented in the Intel® XScale™ core, namely:

- debug modes, registers and exceptions
- a serial debug communication link via the JTAG interface
- a trace buffer
- a mini Instruction Cache
- a mechanism to load the instruction cache through JTAG
- Debug Handler software issues

## 10.1 Introduction

Two key terms that require clear definition in debugging are the differences between the host and target ends of a debugging scenario. The following text in this chapter refers to a *debugger* and a *debug handler.*

The debugger is software that runs on a host system outside of the Intel® XScale™ core. The debug handler is an event handler that runs on the Intel® XScale™ core, when a debug event occurs.

The Intel® XScale™ core debug unit, when used with a debugger application, allows software running on an Intel® XScale™ core target to be debugged. The debug unit allows the debugger to stop program execution and re-direct execution to a debug handling routine. Once program execution has stopped, the debugger can examine or modify processor state, co-processor state, or memory. The debugger can then restart execution of the application.

The external debug interface to the PXA255 processor is via the JTAG port. Further details on the JTAG interface can be found in Section 9, “Test”.

On the Intel® XScale™ core, one of two debug modes can be entered:

- Halt mode
- Monitor mode

### 10.1.1 Halt Mode

When the debug unit is configured for halt mode, the reset vector is overloaded to serve as the debug vector. A new processor mode, DEBUG mode (CPSR[4:0] = 0x15), is added to allow debug exceptions to be handled similarly to other types of ARM* exceptions.

When a debug exception occurs, the processor switches to debug mode and redirects execution to a debug handler, via the reset vector. After the debug handler begins execution, the debugger can communicate with the debug handler to examine or alter processor state or memory through the JTAG interface.

The debug handler can be downloaded and locked directly into the instruction cache through the JTAG interface so external memory is not required to contain debug handler code.

### 10.1.2 Monitor Mode

In monitor mode, debug exceptions are handled like ARM* prefetch aborts or ARM* data aborts, depending on the cause of the exception.

When a debug exception occurs, the processor switches to abort mode and branches to a debug handler using the pre-fetch abort vector or data abort vector. The debugger then communicates with the debug handler to access processor state or memory contents.

## 10.2 Debug Registers

CP15 registers are accessible using MRC and MCR. CRn and CRm specify the register to access. The opcode_1 and opcode_2 fields are not used and must be set to 0. Software access to all debug registers must be done in privileged mode. User mode access will generate an undefined instruction exception. Specifying registers which do not exist has unpredictable results.

**Table 10-1. Coprocessor 15 Debug Registers**

| Register name | CRn | CRm |
|---|---|---|
| Instruction breakpoint register 0 (IBCR0) | 14 | 8 |
| Instruction breakpoint register 1 (IBCR1) | 14 | 9 |
| Data breakpoint register 0 (DBR0) | 14 | 0 |
| Data breakpoint register 1 (DBR1) | 14 | 3 |
| Data breakpoint control register (DBCON) | 14 | 4 |

CP14 registers are accessible using MRC, MCR, LDC and STC (CDP to any CP14 registers will cause an undefined instruction trap). The CRn field specifies the number of the register to access. The CRm, opcode_1, and opcode_2 fields are not used and must be set to 0.

**Table 10-2. Coprocessor 14 Debug Registers**

| Register name | CRn | CRm |
|---|---|---|
| TX Register (TX) | 8 | 0 |
| RX Register (RX) | 9 | 0 |
| Debug Control and Status Register (DCSR) | 10 | 0 |
| Trace Buffer Register (TBREG) | 11 | 0 |
| Checkpoint Register 0 (CHKPT0) | 12 | 0 |
| Checkpoint Register 1 (CHKPT1) | 13 | 0 |
| TXRX Control Register (TXRXCTRL) | 14 | 0 |

The TX and RX registers, certain bits in the TXRXCTRL register, and certain bits in the DCSR can be accessed by a debugger through the JTAG interface. This is to allow an external debugger to have access to the internal state of the processor. For the details of which bits can be accessed see Table 10-8, Table 10-12 and Table 10-3.

## 10.3 Debug Control and Status Register (DCSR)

The DCSR register is the main control register for the debug unit. Table 10-3 shows the format of the register. The DCSR register can be accessed in privileged modes by software running on the core or by a debugger through the JTAG interface. Refer to Section 10, "SELDCSR JTAG Register" for details about accessing DCSR through JTAG.

For the Trap bits in Table 10-3 writing a one enables the trap behavior, while writing a zero will disable the trap.

### Table 10-3. 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 | | | | | | | TF | TI | | TD | TA | TS | TU | TR | | | | | | | | | | | SA | MOE | | | M | E |

| Bits | Access | Description | Reset Value | TRST Value |
|---|---|---|---|---|
| 31 | Software Read / Write<br>JTAG Read-Only | Global Enable (GE)<br>0: disables all debug functionality<br>1: enables all debug functionality | 0 | unchanged |
| 30 | Software Read Only<br>JTAG Read / Write | Halt Mode (H)<br>0: Monitor Mode<br>1: Halt Mode | unchanged | 0 |
| 29:24 | Read-undefined / Write-As-Zero | Reserved | undefined | undefined |
| 23 | Software Read Only<br>JTAG Read / Write | Trap FIQ (TF) | unchanged | 0 |
| 22 | Software Read Only<br>JTAG Read / Write | Trap IRQ (TI) | unchanged | 0 |
| 21 | Read-undefined / Write-As-Zero | Reserved | undefined | undefined |
| 20 | Software Read Only<br>JTAG Read / Write | Trap Data Abort (TD) | unchanged | 0 |
| 19 | Software Read Only<br>JTAG Read / Write | Trap Prefetch Abort (TA) | unchanged | 0 |
| 18 | Software Read Only<br>JTAG Read / Write | Trap Software Interrupt (TS) | unchanged | 0 |
| 17 | Software Read Only<br>JTAG Read / Write | Trap Undefined Instruction (TU) | unchanged | 0 |
| 16 | Software Read Only<br>JTAG Read / Write | Trap Reset (TR) | unchanged | 0 |

### Table 10-3. 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 |  |  |  |  |  |  | TF | TI |  | TD | TA | TS | TU | TR |  |  |  |  |  |  |  |  |  |  | SA | MOE |  |  | M | E |

| Bits | Access | Description | Reset Value | TRST Value |
|---|---|---|---|---|
| 15:6 | Read-undefined / Write-As-Zero | Reserved | undefined | undefined |
| 5 | Software Read / Write<br>JTAG Read-Only | Sticky Abort (SA) | 0 | unchanged |
| 4:2 | Software Read / Write<br>JTAG Read-Only | 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: External Debug Event Asserted<br>101: Vector Trap Occurred<br>110: Trace Buffer Full Break<br>111: Reserved | 0b000 | unchanged |
| 1 | Software Read / Write<br>JTAG Read-Only | Trace Buffer Mode (M)<br>0: Wrap around mode<br>1: fill-once mode | 0 | unchanged |
| 0 | Software Read / Write<br>JTAG Read-Only | Trace Buffer Enable (E)<br>0: Disabled<br>1: Enabled | 0 | unchanged |

## 10.3.1 Global Enable Bit (GE)

The Global Enable bit disables and enables all debug functionality (except the reset vector trap). 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 external debug breaks, hardware breakpoints, and non-reset vector traps are ignored.

## 10.3.2 Halt Mode Bit (H)

The Halt Mode bit configures the debug unit for either halt mode or monitor mode.

## 10.3.3 Vector Trap Bits (TF,TI,TD,TA,TS,TU,TR)

The Vector Trap bits allow instruction breakpoints to be set on exception vectors without using up any of the breakpoint registers. When a bit is set, it acts as if an instruction breakpoint was set up on the corresponding exception vector. A debug exception is generated before the instruction in the exception vector executes.

Software running on the Intel® XScale™ core must set the Global Enable bit and the debugger must set the Halt Mode bit and the appropriate vector trap bit through JTAG to set up a non-reset vector trap.

To set up a reset vector trap, the debugger sets the Halt Mode bit and reset vector trap bit through JTAG. The Global Enable bit does not effect the reset vector trap. A reset vector trap can be 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.

### 10.3.4 Sticky Abort Bit (SA)

The Sticky Abort bit is only valid in Halt mode. It indicates a data abort occurred within the Special Debug State (see Section 10, “Halt Mode”). Since Special Debug State disables all exceptions, a data abort exception does not occur. However, the processor sets the Sticky Abort bit to indicate a data abort was detected. The debugger can use this bit to determine if a data abort was detected during the Special Debug State. The sticky abort bit must be cleared by the debug handler before exiting the debug handler.

### 10.3.5 Method of Entry Bits (MOE)

The Method of Entry bits specify 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 10-4) in the MOE field.

### 10.3.6 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 - The trace buffer automatically generates a debug exception (trace buffer full break) when it becomes full.

### 10.3.7 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 to disable the trace buffer when a debug exception occurs. For more details on the trace buffer refer to Section 10, “Trace Buffer”.

## 10.4 Debug Exceptions

A debug exception causes the processor to re-direct execution to a debug event handling routine. The Intel® XScale™ core debug architecture defines the following debug exceptions:

1. instruction breakpoint
2. data breakpoint
3. software breakpoint
4. external debug break
5. exception vector trap
6. trace-buffer full break

When a debug exception occurs, the processor's actions depend on whether the debug unit is configured for Halt mode or Monitor mode.

Table 10-4 shows the priority of debug exceptions relative to other processor exceptions.

**Table 10-4. Event Priority**

| Event | Priority |
|---|---|
| Reset | 1 |
| Vector Trap[a] | 2 |
| Data Abort (precise) | 3 |
| Data Breakpoint | 4 |
| Data Abort (imprecise) | 5 |
| External debug break, Trace-buffer full | 6 |
| FIQ | 7 |
| IRQ | 8 |
| Instruction Breakpoint | 9 |
| Prefetch Abort | 10 |
| Undefined, **SWI**, **BKPT** | 11 |

[a] See "Vector Trap Bits (TF,TI,TD,TA,TS,TU,TR)" on page 10-4 for vector trap options

## 10.4.1 Halt Mode

The debugger turns on Halt mode through the JTAG interface by scanning in a value that sets the bit in DCSR. The debugger turns off Halt mode through JTAG, either by scanning in a new DCSR value or by a TRST. Processor reset does not effect the value of the Halt mode bit.

When halt mode is active, the processor uses the reset vector as the debug vector. The debug handler and exception vectors can be downloaded directly into the instruction cache, to intercept the default vectors and reset handler, or they can be resident in external memory. Downloading into the instruction cache allows a system with memory problems, or no external memory, to be debugged. Refer top Section 10.13, "Downloading Code into the Instruction Cache" on page 10-30 for details about downloading code into the instruction cache.

During Halt mode, software running on the Intel® XScale™ core cannot access DCSR, or any of hardware breakpoint registers, unless the processor is in Special Debug State (SDS), described below.

When a debug exception occurs during Halt mode, the processor takes the following actions:

- disables the trace buffer
- sets DCSR.moe encoding
- processor enters a Special Debug State (SDS)
- for data breakpoints, trace buffer full break, and external debug break:
  R14_dbg = PC of the next instruction to execute + 4
  for instruction breakpoints and software breakpoints and vector traps:
  R14_dbg = PC of the aborted instruction + 4
- SPSR_dbg = CPSR

- CPSR[4:0] = 0b10101 (DEBUG mode)
- CPSR[5] = 0
- CPSR[6] = 1
- CPSR[7] = 1
- PC = 0x0[1]

Following a debug exception, the processor switches to debug mode and enters SDS, which allows the following special functionality:

- All events are disabled. SWI or 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). The processor reports data aborts detected during SDS by setting the Sticky Abort bit in the DCSR, but does not generate an exception (processor also sets up FSR and FAR as it normally would for a data abort).
- Normally, during halt mode, software cannot write the hardware breakpoint registers or the DCSR. However, during the SDS, software has write access to the breakpoint registers (see Section 10, "HW Breakpoint Resources") and the DCSR (see Table 10-3, "Debug Control and Status Register (DCSR)" on page 10-3).
- The IMMU is disabled. In halt mode, since the debug handler would typically be downloaded directly into the instruction cache, it would not be appropriate to do TLB accesses or translation walks, since there may not be any external memory or if there is, the translation table or TLB may not contain a valid mapping for the debug handler code. To avoid these problems, the processor internally disables the IMMU during SDS.
- The PID is disabled for instruction fetches. This prevents fetches of the debug handler code from being remapped to a different address than where the code was downloaded.

The SDS remains in effect regardless of the processor mode. This allows the debug handler to switch to other modes, maintaining SDS functionality. Entering user mode will cause unpredictable behavior. The processor exits SDS following a CPSR restore operation.

When exiting, the debug handler should use:

```
subs pc, lr, #4
```

This restores CPSR, turns off all of SDS functionality, and branches to the target instruction.

## 10.4.2 Monitor Mode

In monitor mode, the processor handles debug exceptions like normal ARM* exceptions. If debug functionality is enabled (DCSR[31] = 1) and the processor is in Monitor mode, debug exceptions cause either a data abort or a pre-fetch abort.

The following debug exceptions cause data aborts:

- data breakpoint
- external debug break
- trace-buffer full break

[1] When the vector table is relocated (CP15 Control Register[13] = 1), the debug vector is relocated to 0xFFFF_0000

The following debug exceptions cause pre-fetch aborts:

- instruction breakpoint
- BKPT instruction

The processor ignores vector traps during monitor mode.

When an exception occurs in monitor mode, the processor takes the following actions:

1. disables the trace buffer
2. sets DCSR.moe encoding
3. sets FSR[9]
4. R14_abt = PC of the next instruction to execute + 4 (for Data Aborts)
   R14_abt = PC of the faulting instruction + 4 (for Prefetch Aborts)
5. SPSR_abt = CPSR
6. CPSR[4:0] = 0b10111 (ABORT mode)
7. CPSR[5] = 0
8. CPSR[6] = unchanged
9. CPSR[7] = 1
10. PC = 0xc (for Prefetch Aborts),
    PC = 0x10 (for Data Aborts)

During Abort mode, external Debug breaks and trace buffer full breaks are internally postponed. When the processor exits Abort mode, either through a CPSR restore or a write directly to the CPSR, the postponed Debug breaks will immediately generate a Debug exception. Any of these postponed Debug breaks are cleared once any one Debug exception occurs.

When exiting, the debug handler should do a CPSR restore operation that branches to the next instruction to be executed in the program under debug.

## 10.5 HW Breakpoint Resources

The Intel® XScale™ core debug architecture defines two instruction and two data breakpoint registers, denoted IBCR0, IBCR1, DBR0, and DBR1.

The instruction and data address 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 Modified Virtual Address (MVA) refers to the virtual address ORed with the PID. Refer to Section 7.2.11, “Register 13: Process ID” on page 7-12 for more details on the PID. The processor does not OR the PID with the specified breakpoint address prior to doing address comparison. This must be done by the programmer and written to the breakpoint register as the MVA. This applies to data and instruction breakpoints.

## 10.5.1 Instruction Breakpoints

The Debug architecture defines two instruction breakpoint registers (IBCR0 and IBCR1). The format of these registers is shown in Table 10-5., Instruction Breakpoint Address and Control Register (IBCRx). 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. Enabling instruction breakpoints while debug is globally disabled (DCSR.GE=0) will result in unpredictable behavior.

Table 10-5. Instruction Breakpoint Address and Control Register (IBCRx)

| 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| IBCRx | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | E |

reset value: unpredictable address, disabled

| Bits | Access | Description |
|---|---|---|
| 31:1 | Read / Write | Instruction Breakpoint MVA<br>in ARM* mode, IBCRx[1] is ignored |
| 0 | Read / Write | **IBCRx Enable (E)** -<br>0 = Breakpoint disabled<br>1 = Breakpoint enabled |

An instruction breakpoint will generate a debug exception before the instruction at the address specified in the IBCR executes. When an instruction breakpoint occurs, the processor sets the DBCR[**MOE**] bits to 0b001.

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 must be completely handled in software (either on the host or by the debug handler).

## 10.5.2 Data Breakpoints

The Intel® XScale™ core debug architecture defines two data breakpoint registers (DBR0, DBR1). The format of the registers is shown in Table 10-6.

Table 10-6. 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DBRx | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

reset value: unpredictable

| Bits | Access | Description |
|---|---|---|
| 31:0 | Read / Write | **DBR0**: Data Breakpoint MVA<br>**DBR1**:<br>Data Address Mask OR<br>Data Breakpoint MVA |

DBR0 is a dedicated data address breakpoint register. DBR1 can be programmed for 1 of 2 operations:

- data address mask
- second data address breakpoint

The DBCON register controls the functionality of DBR1, as well as the enables for both DBRs. DBCON also controls what type of memory access to break on.

### Table 10-7. 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: 0x0000_0000

| Bits | Access | Description |
|---|---|---|
| 31:9 | Read-as-Zero / Write-ignored | Reserved |
| 8 | Read / Write | **DBR1 Mode (M)** -<br>0: DBR1 = Data Address Breakpoint<br>1: DBR1 = Data Address Mask |
| 7:4 | Read-as-Zero / Write-ignored | 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. The bits set in DBR1 are ignored by the processor when comparing the address of a memory access with the address in DBR0. Using DBR1 as a data address mask allows a range of addresses to generate a data breakpoint. When DBR1 is selected as a data address mask, it is unaffected by the E1 field of DBCON. 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.

A data breakpoint is triggered if 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 if 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. Any other type of memory access can trigger a data breakpoint. For data breakpoint purposes the **SWP** and **SWPB** instructions are treated as stores - they will not cause a data breakpoint if the breakpoint is set up to break on loads only and an address match occurs.

On unaligned memory accesses, breakpoint address comparison is done on a word-aligned address (aligned down to word boundary).

When a memory access triggers a data breakpoint, the breakpoint is reported after the access is issued. The memory access will not be aborted by the processor. The actual timing of when the access completes with respect to the start of the debug handler depends on the memory configuration.

On a data breakpoint, the processor generates a debug exception and re-directs execution to the debug handler before the next instruction executes. The processor reports the data breakpoint by setting the DCSR.moe to 0b010. The link register of a data breakpoint is always PC (of the next instruction to execute) + 4, regardless of whether the processor is configured for monitor mode or halt mode.

## 10.6 Software Breakpoints

| | |
|---|---|
| Mnemonics: | **BKPT** (See ARM* Architecture Reference Manual, ARMv5T) |
| Operation: | If DCSR[31] = 0, BKPT is a **NOP**;<br>If DCSR[31] =1, BKPT causes a debug exception |

The processor handles the software breakpoint as described in Section 10.4, “Debug Exceptions” on page 10-5.

## 10.7 Transmit/Receive Control Register (TXRXCTRL)

Communications between the debug handler and debugger are controlled through handshaking bits that ensure the debugger and debug handler make synchronized accesses to TX and RX. The debugger side of the handshaking is accessed through the DBGTX (Section 10, “DBGTX JTAG Register”) and DBGRX (Section 10, “DBGRX JTAG Register”) JTAG Data Registers, depending on the direction of the data transfer. The debug handler uses separate handshaking bits in TXRXCTRL register for accessing TX and RX.

The TXRXCTRL register also contains two other bits that 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 they can be read directly into the CC flags in the CPSR with an MRC (with Rd = PC). The subsequent instruction can then conditionally execute based on the updated CC value

### Table 10-8. TX RX 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 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

reset value: 0x0000_0000

| Bits | Access | Description |
|---|---|---|
| 31 | Software Read-only / Write-ignored<br>JTAG Write-only | RR<br>1=RX Register Ready |
| 30 | Software Read / Write | OV<br>1=RX overflow sticky flag |
| 29 | Software Read-only/ Write-ignored<br>JTAG Write-only | D<br>High-speed download flag |
| 28 | Software Read-only/ Write-ignored<br>JTAG Write-only | TR<br>1=TX Register Ready |
| 27:0 | Read-as-Zero / Write-ignored | Reserved |

## 10.7.1 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 can be used in which only the debug handler polls the RR bit before accessing the RX register, while the debugger continuously downloads data.

Table 10-9 shows the normal handshaking used to access the RX register.

### Table 10-9. 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 can be bypassed to allow the download rate to be increased. Table 10-10 shows the handshaking used when the debugger is doing a high-speed download. Before the high-speed download can start, 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 can read the previous data from RX before the debugger can scan in the new data.

**Table 10-10. High-Speed Download Handshaking States**

| Debugger Actions |
|---|
| Debugger wants to transfer code into the Intel® XScale™ core system memory.<br>Prior to starting download, the debugger must poll the 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 if the debug handler does not read the previous data before the debugger completes scanning in the new data, (see Section 10, "Overflow Flag (OV)" 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. |

## 10.7.2 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 will be polling for that data.

However, if 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 will remain set, until cleared by a write to TXRXCTRL with an MCR. After the debugger completes the download, it can examine the OV bit to determine if an overflow occurred. The debug handler software is responsible for saving the address of the last valid store before the overflow occurred.

## 10.7.3 Download Flag (D)

The value of the download flag is set by the debugger through JTAG. This flag is asserted during high-speed download to replace a loop counter.

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 should set the D bit. On completing the download the debugger clears the D bit releasing the debug handler to take the data.

The download flag becomes especially useful when an overflow occurs. If a loop counter is used, and an overflow occurs, the debug handler cannot determine how many data words overflowed. Therefore the debug handler counter may get out of sync with the debugger - the debugger may finish downloading the data, but the debug handler counter may indicate there is more data to be downloaded - this results in unpredictable behavior of the debug handler.

## 10.7.4 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 10-11 shows the handshaking used to access the TX register.

### Table 10-11. 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. |

## 10.7.5 Conditional Execution Using TXRXCTRL

All of the bits in TXRXCTRL are placed such that they can be read directly into the CC flags using an MCR instruction. To simplify the debug handler, the TXRXCTRL register should be read using the following instruction:

```
mrc p14, 0, r15, C14, C0, 0
```

This instruction will directly update the condition codes in the CPSR. The debug handler can then conditionally execute based on each CC bit. Table 10-12 shows the mnemonic extension to conditionally execute based on whether the TXRXCTRL bit is set or clear.

### Table 10-12. TXRXCTRL Mnemonic Extensions

| TXRXCTRL bit | mnemonic extension to execute if bit set | mnemonic extension to execute if 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:   mcr    p14, 0, r15, c14, c0, 0# read the handshaking bit in TXRXCTRL
```

```
mcrmi   p14, 0, r0, c9, c0, 0  # if RX is valid, read it
bpl     loop                   # if RX is not valid, loop
```

## 10.8 Transmit Register (TX)

The TX register is the debug handler transmit buffer. The debug handler sends data to the debugger through this register.

### Table 10-13. 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**

| Bits | Access | Description |
|---|---|---|
| 31:0 | Software 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/MRC) 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 10-11, "TX Handshaking" on page 10-14.

## 10.9 Receive Register (RX)

The RX register is the receive buffer used by the debug handler to get data sent by the debugger through the JTAG interface.

### Table 10-14. 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**

| Bits | Access | Description |
|---|---|---|
| 31:0 | Software Read-only<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 10, "RX Register Ready Bit (RR)".

## 10.10 Debug JTAG Access

There are four JTAG instructions used by the debugger during software debug: LDIC, SELDCSR, DBGTX and DBGRX. LDIC is described in Section 10, “Downloading Code into the Instruction Cache”. The other three JTAG instructions are described in this section.

SELDCSR, DBGTX and DBGRX use a common 36-bit shift register (DBG_SR). New data is shifted in and captured data out through the DBG_SR. In the UPDATE_DR state, the new data is shifted into the appropriate data register. Details of the JTAG state machine can be found in Section 9, “Test”.

### 10.10.1 SELDCSR JTAG Command

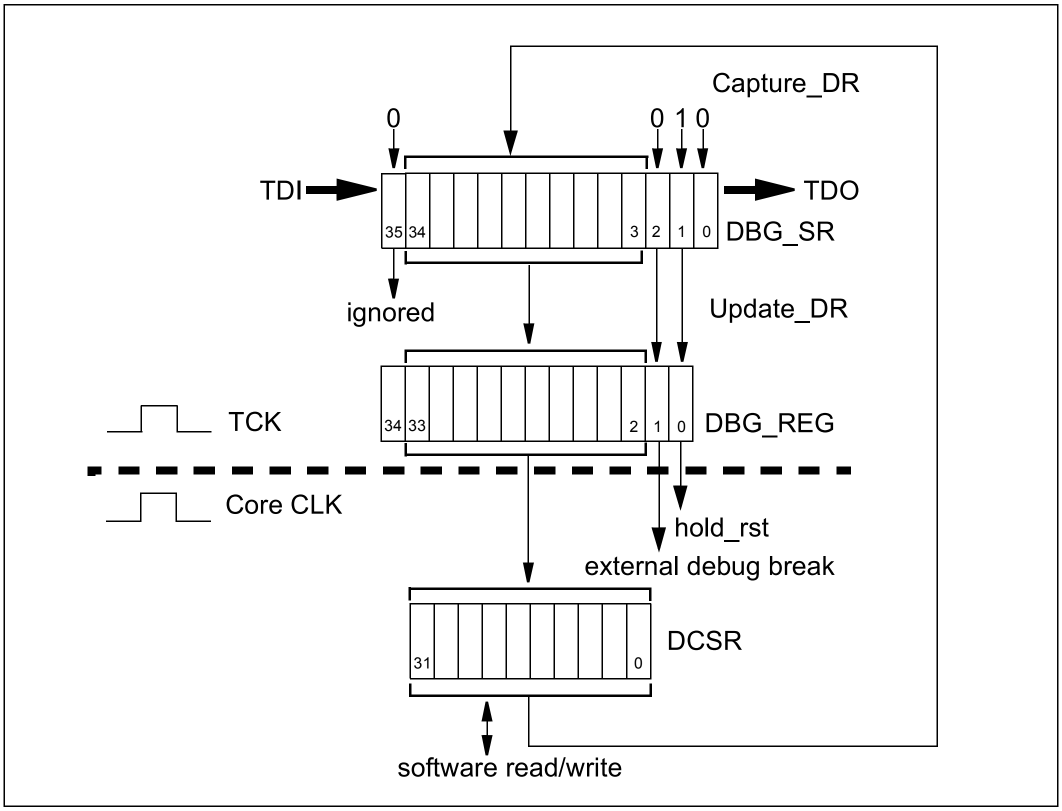
The ‘SELDCSR’ JTAG instruction selects the DCSR JTAG data register. The JTAG opcode is ‘01001’. When the SELDCSR JTAG instruction is in the JTAG instruction register, the debugger can directly access the Debug Control and Status Register (DCSR). The debugger can only modify certain bits through JTAG, but can read the entire register.

The SELDCSR instruction also allows the debugger to generate an external debug break.

### 10.10.2 SELDCSR JTAG Register

Placing the "SELDCSR" JTAG instruction in the JTAG IR, selects the DCSR JTAG Data register (Figure 10-1), allowing the debugger to access the DCSR, generate an external debug break, set the hold_rst signal, which is used when loading code into the instruction cache during reset.

**Figure 10-1. SELDCSR Hardware**



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 10-1.

A new DCSR value can be 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 DBG_REG[33:2]. This value is then loaded into the actual DCSR register. All bits defined as JTAG writable in Table 10-3, "Debug Control and Status Register (DCSR)" on page 10-3 are updated.

An external host and the debug handler running on the Intel® XScale™ core must synchronize access to the DCSR. If one side writes the DCSR at the same time the other side reads the DCSR, the results are unpredictable.

### 10.10.2.1 DBG.HLD_RST

The debugger uses DBG.HLD_RST when loading code into the instruction cache during a processor reset. Details about loading code into the instruction cache are in Section 10, “Downloading Code into the Instruction Cache”.

The debugger must set DBG.HLD_RST before or during assertion of the reset pin. Once DBG.HLD_RST is set, the reset pin can be de-asserted, and the processor will internally remain in reset. The debugger can then load debug handler code into the instruction cache before the processor begins executing any code.

Once the code download is complete, the debugger must clear DBG.HLD_RST. This takes the processor out of reset, and execution begins at the reset vector.

A debugger sets DBG.HLD_RST in one of 2 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 DBG.HLD_RST, clear DBG.BRK, and leave the DCSR unchanged (the DCSR bits captured in DBG_SR[34:3] are written back to the DCSR on the Update_DR). Refer to Figure 9-3, “TAP Controller State Diagram” on page 9-9.
- Alternatively, a ‘1’ can be scanned into DBG_SR[1], with the appropriate value scanned in for the DCSR and DBG.BRK.

DBG.HLD_RST can only be cleared by scanning in a ‘0’ to DBG_SR[1] and scanning in the appropriate values for the DCSR and DBG.BRK.

### 10.10.2.2 DBG.BRK

DBG.BRK allows the debugger to generate an external debug break and asynchronously re-direct execution to a debug handling routine.

A debugger sets an external debug break by scanning data into the DBG_SR with DBG_SR[2] set and the desired value to set the DCSR JTAG writable bits in DBG_SR[34:3].

Once an external debug break is set, it remains set internally until a debug exception occurs. In Monitor mode, external debug breaks detected during abort mode are postponed until the processor exits abort mode. In Halt mode, breaks detected during SDS are postponed until the processor exits SDS. When an external debug break is detected outside of these two cases, the processor ceases executing instructions as quickly as the current pipeline contents can be completed. This improves breakpoint accuracy by reducing the number of instructions that can execute after the external debug break is requested. However, the processor will continue to process any instructions which have already begun execution. Debug mode will not be entered until all processor activity has ceased in an orderly fashion.

### 10.10.2.3 DBG.DCSR

The DCSR is updated with the value loaded into DBG.DCSR following an Update_DR. Only bits specified as writable by JTAG in Table 10-3 are updated.

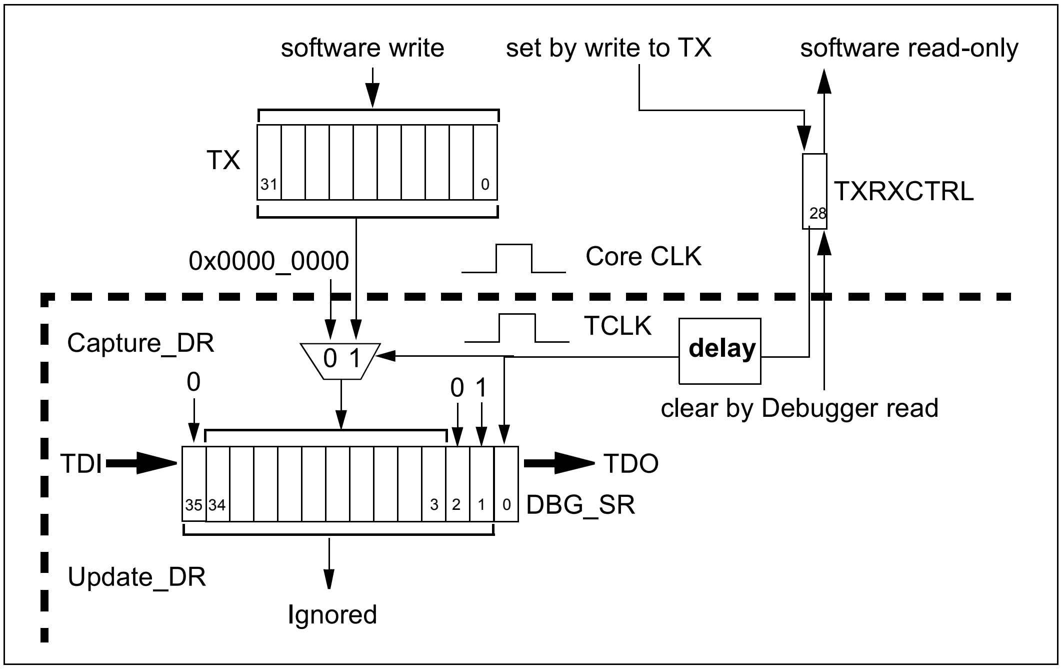
## 10.10.3 DBGTX JTAG Command

The ‘DBGTX’ JTAG instruction selects the DBGTX JTAG data register. The JTAG opcode for this instruction is ‘0b10000’. Once the DBGTX data register is selected, the debugger can receive data from the debug handler.

## 10.10.4 DBGTX JTAG Register

The DBGTX JTAG instruction selects the Debug JTAG Data register (Figure 10-2). The debugger uses the DBGTX data register to poll for breaks (internal and external) both to cause an entry into Debug mode and once in Debug mode, to read data from the debug handler.

**Figure 10-2. DBGTX Hardware**



A Capture_DR loads the TX register value into DBG_SR[34:3] and TXRXCTRL[28] into DBG_SR[0]. The other bits in DBG_SR are loaded as shown in Figure 10-3.

The captured TX value is scanned out during the Shift_DR state.

Data scanned in is ignored on an Update_DR.

A ‘1’ captured in DBG_SR[0] indicates the captured TX data is valid. After doing a Capture_DR, the debugger must place the JTAG state machine in the Shift_DR state to guarantee that a debugger read clears TXRXCTRL[28].

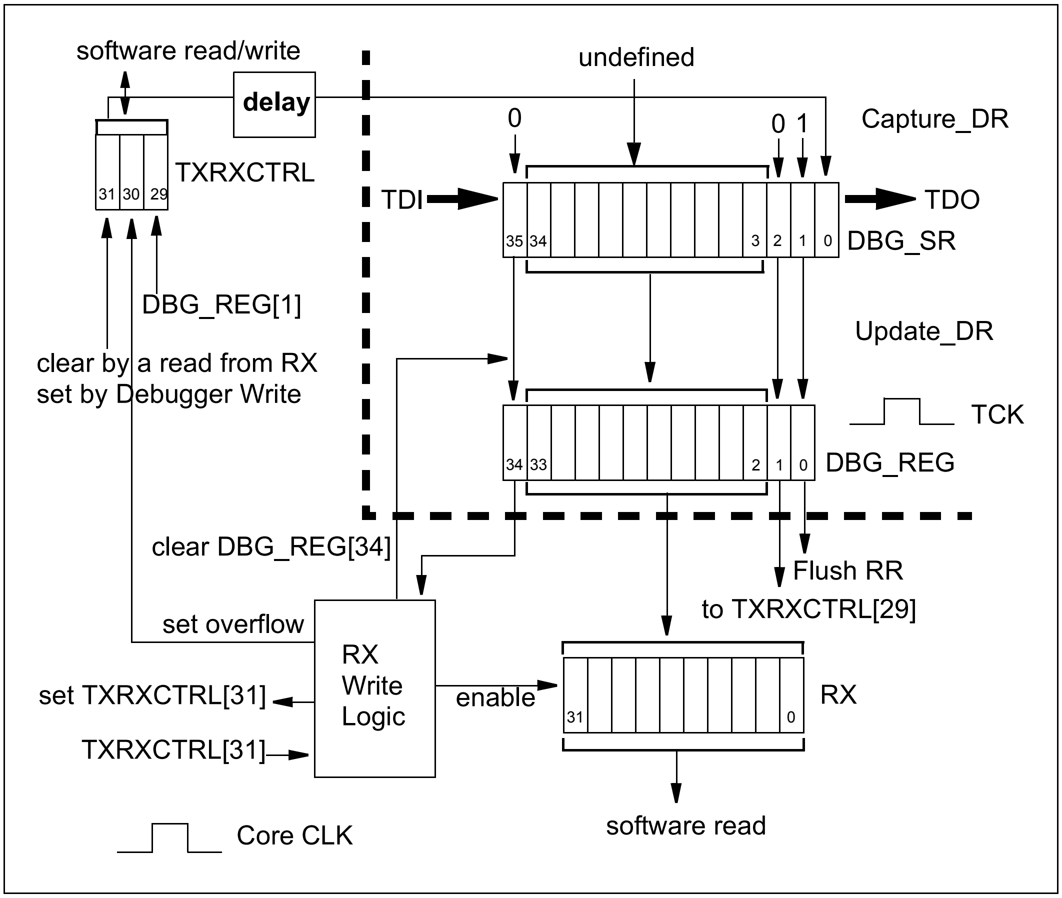
## 10.10.5 DBGRX JTAG Command

The ‘DBGRX’ JTAG instruction selects the DBGRX JTAG data register. The JTAG opcode for this instruction is ‘0b00010’. Once the DBGRX data register is selected, the debugger can send data to the debug handler through the RX register.

## 10.10.6 DBGRX JTAG Register

The DBGRX JTAG instruction selects the DBGRX JTAG Data register. The debugger uses the DBGRX data register to send data or commands to the debug handler.

Figure 10-3. DBGRX Hardware



A Capture_DR loads TXRXCTRL[31] into DBG_SR[0]. The other bits in DBG_SR are loaded as shown in Figure 10-3.

The captured data is scanned out during the Shift_DR state.

While polling TXRXCTRL[31], incorrectly setting DBG_SR[35] or DBG_SR[1] will cause unpredictable behavior following an Update_DR.

Update_DR parallel loads DBG_SR[35:1] into DBG_REG[34:0]. Whether the new data gets written to the RX register or an overflow condition is detected depends on the inputs to the RX write logic.

### 10.10.6.1 RX Write Logic

The RX write logic (Figure 10-4) serves 4 functions:

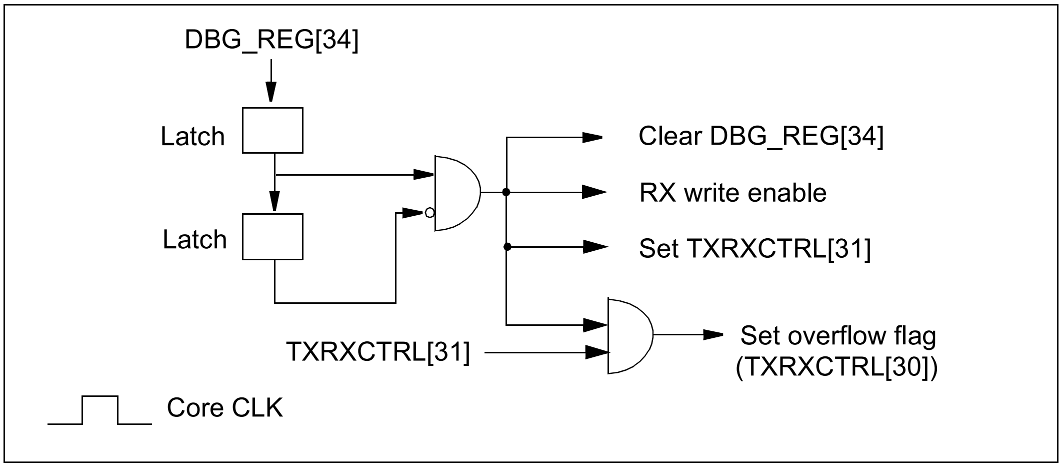
1) Enable the debugger write to RX - the logic ensures only new, valid data from the debugger is written to RX. In particular, when the debugger polls TXRXCTRL[31] to see whether the debug handler has read the previous data from RX. The JTAG state machine must go through Update_DR, which should not modify RX.

2) Clear DBG_REG[34] - mainly to support high-speed download. During high-speed download, the debugger continuously scans in data to send to the debug handler and sets DBG_REG[34] to signal the data is valid. Since DBG_REG[34] is never cleared by the debugger in this case, the ‘0’ to ‘1’ transition used to enable the debugger write to RX would not occur.

3) Set TXRXCTRL[31] - When the debugger writes new data to RX, the logic automatically sets TXRXCTRL[31], signalling to the debug handler that the data is valid.

4) Set the overflow flag (TXRXCTRL[30] - During high-speed download, the debugger does not poll to see if the handler has read the previous data. If the debug handler stalls long enough, the debugger may overwrite the previous data before the handler can read it. The logic sets the overflow flag when the previous data has not been read yet, and the debugger has just written new data to RX.

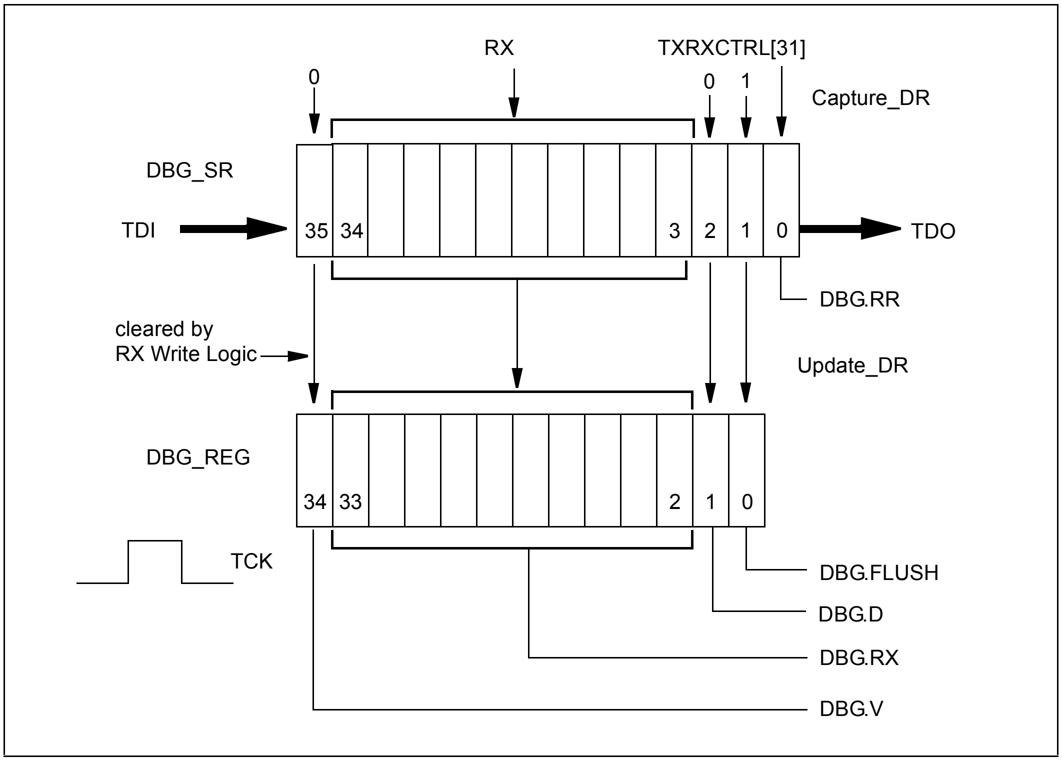
Figure 10-4. RX Write Logic



### 10.10.6.2 DBGRX Data Register

The bits in the DBGRX data register (Figure 10-5) are used by the debugger to send data to the processor. The data register also contains a bit to flush previously written data and a high-speed download flag.

Figure 10-5. DBGRX Data Register



## 10.10.6.3 DBG.RR

The debugger uses DBG.RR as part of the synchronization that occurs between the debugger and debug handler for accessing RX. This bit contains the value of TXRXCTRL[31] after a Capture_DR. The debug handler automatically sets TXRXCTRL[31] by doing a write to RX.

The debugger polls DBG.RR to determine when the handler has read the previous data from RX.

The debugger sets TXRXCTRL[31] by setting the DBG.V bit.

## 10.10.6.4 DBG.V

The debugger sets this bit to indicate the data scanned into DBG_SR[34:3] is valid data to write to RX. DBG.V is an input to the RX Write Logic and is also cleared by the RX Write Logic.

When this bit is set, the data scanned into the DBG_SR will be written to RX following an Update_DR. If DBG.V is not set and the debugger does an Update_DR, RX will be unchanged.

This bit does not affect the actions of DBG.FLUSH or DBG.D.

## 10.10.6.5 DBG.RX

DBG.RX is written into the RX register based on the output of the RX Write Logic. Any data that needs to be sent from the debugger to the processor must be loaded into DBG.RX with DBG.V set to 1. DBG.RX is loaded from DBG_SR[34:3] when the JTAG enters the Update_DR state.

DBG.RX is written to RX following an Update_DR when the RX Write Logic enables the RX register.

### 10.10.6.6 DBG.D

DBG.D is provided for use during high speed download. This bit is written directly to TXRXCTRL[29]. The debugger sets DBG.D when downloading a block of code or data to the Intel® XScale™ core system memory. The debug handler then uses TXRXCTRL[29] as a branch flag to determine the end of the loop.

Using DBG.D as a branch flags eliminates the need for a loop counter in the debug handler code. This avoids the problem were the debugger's loop counter is out of synchronization with the debug handler's counter because of overflow conditions that may have occurred.

### 10.10.6.7 DBG.FLUSH

DBG.FLUSH allows the debugger to flush any previous data written to RX. Setting DBG.FLUSH clears TXRXCTRL[31].

## 10.10.7 Debug JTAG Data Register Reset Values

Upon asserting TRST, the DEBUG data register is reset. Assertion of the reset pin does not affect the DEBUG data register. Table 10-15 shows the reset and TRST values for the data register. Note: these values apply for DBG_REG for SELDCSR, DBGTX and DBGRX.

Table 10-15. DEBUG Data Register Reset Values

| Bit | TRST | RESET |
|---|---|---|
| DBG_REG[0] | 0 | unchanged |
| DBG_REG[1] | 0 | unchanged |
| DBG_REG[33:2] | unpredictable | unpredictable |
| DBG_REG[34] | 0 | unchanged |

# 10.11 Trace Buffer

The 256 entry trace buffer provides the ability to capture control flow information to be used for debugging an application. Two modes are supported:

1. The buffer fills up completely and generates a debug exception. Then software empties the buffer.
2. The buffer fills up and wraps around until it is disabled. Then software empties the buffer.

## 10.11.1 Trace Buffer CP Registers

CP14 defines three registers (see Table 10-16) for use with the trace buffer. These CP14 registers are accessible using **MRC**, **MCR**, **LDC** and **STC** (**CDP** to any CP14 registers will cause an undefined instruction trap). The CRn field specifies the number of the register to access. The CRm, opcode_1, and opcode_2 fields are not used and must be set to 0.

### Table 10-16. CP 14 Trace Buffer Register Summary

| CP14 Register Number | Register Name |
|---|---|
| 11 | Trace Buffer Register (TBREG) |
| 12 | Checkpoint 0 Register (CHKPT0) |
| 13 | Checkpoint 1 Register (CHKPT1) |

Any access to the trace buffer registers in User mode will cause an undefined instruction exception. Specifying registers which do not exist has unpredictable results.

## 10.11.1.1 Checkpoint Registers

When the debugger reconstructs a trace history, it is required to start at the oldest trace buffer entry and construct a trace going forward. In fill-once mode and wrap-around mode when the buffer does not wrap around, the trace can be reconstructed by starting from the point in the code where the trace buffer was first enabled.

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 is less than of equal to the size of the buffer. The debugger does not know the starting address of the oldest entry read from the trace buffer. The checkpoint registers provide reference addresses to help reduce this problem.

### Table 10-17. 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 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| reset value: Unpredictable | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

| Bits | Access | Description |
|---|---|---|
| 31:0 | Read/Write | **CHKPTx**:<br>target address for corresponding entry in trace buffer |

The two checkpoint registers (CHKPT0, CHKPT1) on the Intel® XScale™ core provide the debugger with two reference addresses to use for re-constructing the trace history.

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 are used to hold the target addresses of specific entries in the trace buffer. Direct and indirect entries written into the trace buffer are marked as checkpoints with the corresponding target address being automatically written into the checkpoint registers. Exception and roll-over messages never use the checkpoint registers. When a checkpoint register value is updated, the processor sets bit 6 of the message byte in the trace buffer to indicate that the update occurred. (refer to Table 10-19., Message Byte Formats)

When the trace buffer contains only one entry relating to a checkpoint, the corresponding checkpoint register is CHKPT0. When the trace buffer wraps around, two entries will typically be marked as relating to checkpoint register values, usually about half the trace buffer length apart.

This is always the case as the messages in the trace buffer vary in length. With two entries, the first (oldest) entry that set a checkpoint in the trace buffer corresponds to CHKPT1, the second entry that set a checkpoint corresponds to CHKPT0.

Although the checkpoint registers are provided for wrap-around mode, they are still valid in fill-once mode.

#### 10.11.1.2 Trace Buffer Register (TBREG)

The trace buffer is read through TBREG, using MRC and MCR. Software can only read the trace buffer when it is disabled. Reading the trace buffer while it is enabled, will cause unpredictable behavior of the trace buffer. Writes to the trace buffer have unpredictable results. Reading the trace buffer returns the oldest byte in the trace buffer in the least significant byte of TBREG. The byte is either a message byte or one byte of the 32 bit address associated with an indirect branch message.Table 10-18 shows the format of the trace buffer register.

Table 10-18. TBREG Format

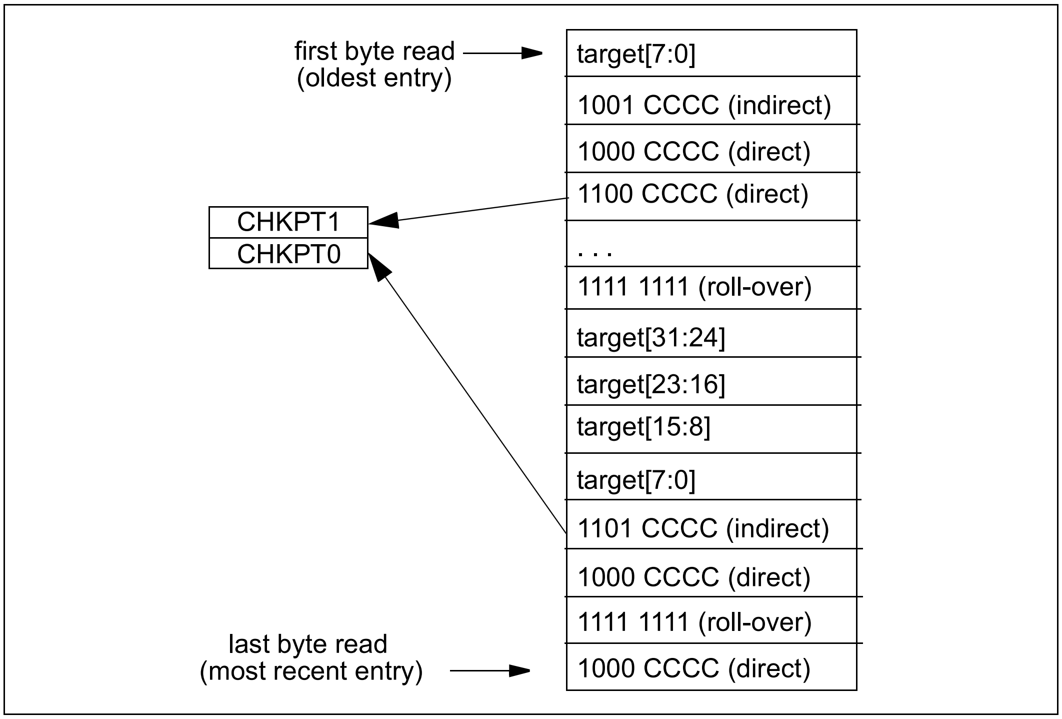
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |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

| Bits | Access | Description |
|---|---|---|
| 31:8 | Read-as-Zero/Write-ignored | Reserved |
| 7:0 | Read / Write-unpredictable | Message Byte or Address Byte |

### 10.11.2 Trace Buffer Usage

The Intel® XScale™ core trace buffer is 256 bytes in length. The first byte read from the buffer represents the oldest trace history information in the buffer. The last (256th) byte read represents the most recent entry in the buffer. The last byte read from the buffer will always be a message byte. This provides the debugger with a starting point for parsing the entries out of the buffer. Because the debugger needs the last byte as a starting point when parsing the buffer, the entire trace buffer must be read (256 bytes on the Intel® XScale™ core) before the buffer can be parsed. Figure 10-6 is a high level view of the trace buffer.

Figure 10-6. High Level View of Trace Buffer



The trace buffer must be initialized prior to its initial usage, then again prior to each subsequent usage. Initialization is done be reading the entire trace buffer. The process of reading the trace buffer also clears it out (all entries are set to 0b00000000), so when the trace buffer has been used to capture a trace, the process of reading the captured trace data also re-initializes the trace buffer for its next usage.

The trace buffer can be used to capture a trace up to a processor reset. A processor reset disables the trace buffer, but does not affect the contents. The trace buffer does not capture reset events or debug exceptions.

Since the trace buffer is cleared out before it is used, all entries are initially 0b00000000. In fill-once mode, these 0's can be used to identify the first valid entry in the trace buffer. In wrap around mode, in addition to identifying the first valid entry, these 0 entries can be used to determine whether a wrap around occurred.

As the trace buffer is read, the oldest entries are read first. Reading a series of 5 (or more) consecutive "0b00000000" entries in the oldest entries indicates that the trace buffer has not wrapped around and the first valid entry will be the first non-zero entry read out.

Reading 4 or less consecutive "0b00000000" entries requires a bit more intelligence in the host software. The host software must determine whether these 0's are part of the address of an indirect branch message, or whether they are part of the "0b00000000" that the trace buffer was initialized with. If the first non-zero message byte is an indirect branch message, then these 0's are part of the address since the address is always read before the indirect branch message (see Section 10, "Address Bytes"). If the first non-zero entry is any other type of message byte, then these 0's indicate that the trace buffer has not wrapped around and that first non-zero entry is the start of the trace.

If the oldest entry from the trace buffer is non-zero, then the trace buffer has either wrapped around or just filled up.

Once the trace buffer has been read and parsed, the host software must re-create the trace history from oldest trace buffer entry to latest. Trying to re-create the trace going backwards from the latest trace buffer entry will not work in most cases, because once a branch message is encountered, it may not be possible to determine the source of the branch.

In fill-once mode, the return from the debug handler to the application should generate an indirect branch message. The address placed in the trace buffer will be that of the target application instruction. Using this as a starting point, re-creating a trace going forward in time is straightforward.

In wrap around mode, the host software uses the checkpoint registers and address bytes from indirect branch entries to re-create the trace going forward. The drawback is that some of the oldest entries in the trace buffer may be untraceable, depending on where the earliest checkpoint (or indirect branch entry) is located. The best case is when the oldest entry in the trace buffer set a checkpoint, so the entire trace buffer can be used to re-create the trace. The worst case is when the first checkpoint is in the middle of the trace buffer and no indirect branch messages exist before this checkpoint. In this case, the host software would have to start at its known address (the first checkpoint) which is half way through the buffer and work forward from there.

## 10.12 Trace Buffer Entries

Trace buffer entries consist of either one or five bytes. Most entries are one byte messages indicating the type of control flow change. The target address of the control flow change represented by the message byte is either encoded in the message byte (as for exceptions) or can be determined by looking at the instruction word (like for direct branches). Indirect branches require five bytes per entry. One byte is the message byte identifying it as an indirect branch. The other four bytes make up the target address of the indirect branch. The following sections describe the trace buffer entries in detail.

### 10.12.1 Message Byte

There are two message formats, (exception and non-exception) as shown in Figure 10-7.

Figure 10-7. Message Byte Formats

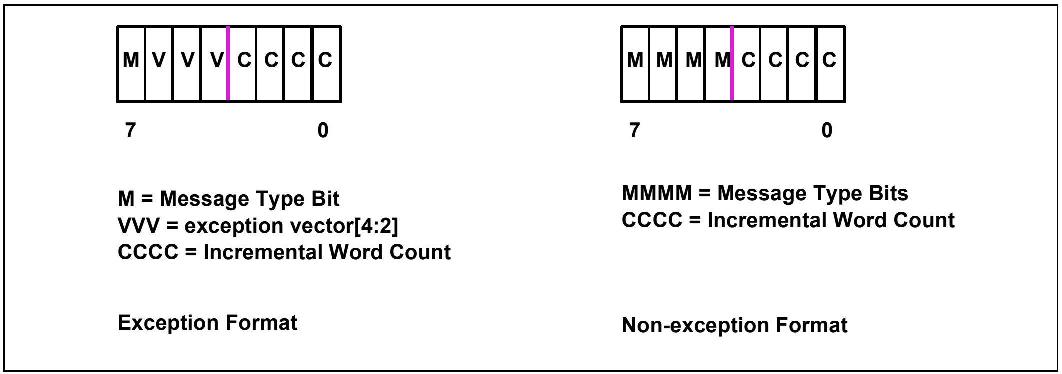


Table 10-19 shows all of the possible trace messages.

### Table 10-19. Message Byte Formats

| Message Name | Message Byte Type | Message Byte format | # address bytes |
|---|---|---|---|
| Exception | exception | 0b0VVV CCCC | 0 |
| Direct Branch[a] | non-exception | 0b1000 CCCC | 0 |
| Direct Branch with checkpoint[ab] | non-exception | 0b1100 CCCC | 0 |
| Indirect Branch[c] | non-exception | 0b1001 CCCC | 4 |
| Indirect Branch with checkpoint[b] | non-exception | 0b1101 CCCC | 4 |
| Roll-over | non-exception | 0b1111 1111 | 0 |

a. Direct branches include ARM* and THUMB bl, b

b. These message types correspond to trace buffer updates to the checkpoint registers

c. Indirect branches include ARM* ldm, ldr, and dproc to PC; ARM* and THUMB bx, blx and THUMB pop.

## 10.12.1.1 Exception Message Byte

When any kind of exception occurs, an exception message is placed in the trace buffer. In an exception message byte, the message type bit (M) is always 0.

The vector exception (VVV) field is used to specify bits[4:2] of the vector address (offset from the base of default or relocated vector table). The vector allows the host software to identify 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.

A count value of 0 indicates that 0 instructions executed since the last control flow change and the current exception. For example, if a branch is immediate followed by a SWI, a direct branch exception message (for the branch) is followed by an exception message (for the SWI) in the trace buffer. The count value in the exception message will be 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, if an IRQ was handled immediately after the branch (before any other instructions executed), the count would still be 0, since no instructions executed after the branch and before the interrupt was handled.

A 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 (if it is an undefined instruction exception, pre-fetch abort, or SWI) or handled before the 16th instruction executed (for FIQ, IRQ, or data abort).

## 10.12.1.2 Non-exception Message Byte

Non-exception message bytes are used for direct branches, indirect branches, and rollovers.

In a non-exception message byte, the 4-bit message type field (MMMM) specifies the type of message (refer to Table 10-19).

The incremental word count (CCCC) is the instruction count since the last control flow change (excluding the current branch). 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. In the case of back-to-back branches the word count would be 0 indicating that 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 message byte was written to the trace buffer.

If the incremental counter reaches its maximum value of 15, a rollover message is written to the trace buffer following the next instruction (which will be the 16th instruction to execute). This is shown in Example 10-1. The count in the rollover message is 0b1111, indicating that 15 instructions have executed after the last branch and before the current non-branch instruction that caused the rollover message.

**Example 10-1. Rollover Messages Examples**

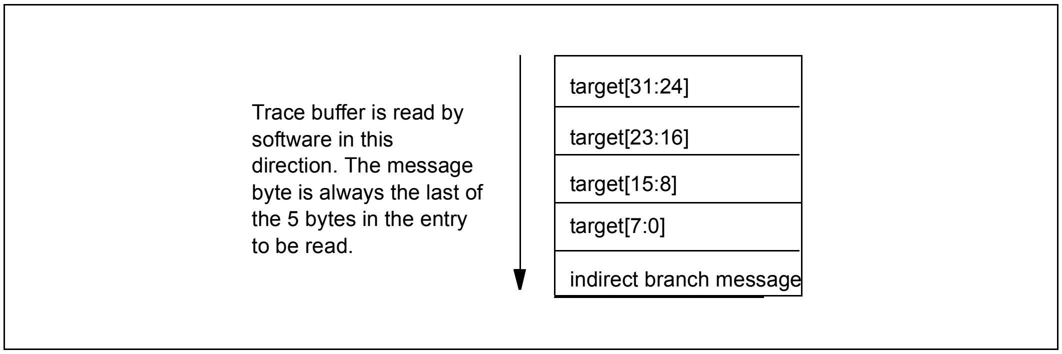
```
count = 5
BL label1                 branch message placed in trace buffer after branch executes
count = 0     ←———        count = 0b0101
MOV
count = 1
MOV
count = 2
MOV
...

count = 14
MOV
count = 15
MOV                       rollover message placed in trace buffer after 16th instruction executes
count = 0     ←———        count = 0b1111
```

If the 16th instruction is a branch (direct or indirect), the appropriate branch message is placed in the trace buffer instead of the roll-over message. The incremental counter is still set to 0b1111, meaning 15 instructions executed between the last branch and the current branch.

## 10.12.1.3 Address Bytes

Only indirect branch entries contain address bytes in addition to the message byte. Indirect branch entries always have four address bytes indicating the target of that indirect branch. When reading the trace buffer the MSB of the target address is read out first; the LSB is the fourth byte read out; and the indirect branch message byte is the fifth byte read out. The byte organization of the indirect branch message is shown in Figure 10-8.

**Figure 10-8. Indirect Branch Entry Address Byte Organization**


Trace buffer is read by software in this direction. The message byte is always the last of the 5 bytes in the entry to be read.
target[31:24]
target[23:16]
target[15:8]
target[7:0]
indirect branch message

## 10.13 Downloading Code into the Instruction Cache

On the Intel® XScale™ core, a 2K mini instruction cache, physically separate[1] from the 32K main instruction cache can be used as an on-chip instruction RAM. An external host can download code directly into either the mini or main instruction cache through JTAG. In addition to downloading code, several cache functions are supported.

The Intel® XScale™ core supports loading either instruction cache during reset and during program execution. Loading the instruction cache during normal program execution requires a strict handshaking protocol between software running on the Intel® XScale™ core and the external host.

In the remainder of this section the term ‘instruction cache’ applies to either main or mini instruction cache.

### 10.13.1 LDIC JTAG Command

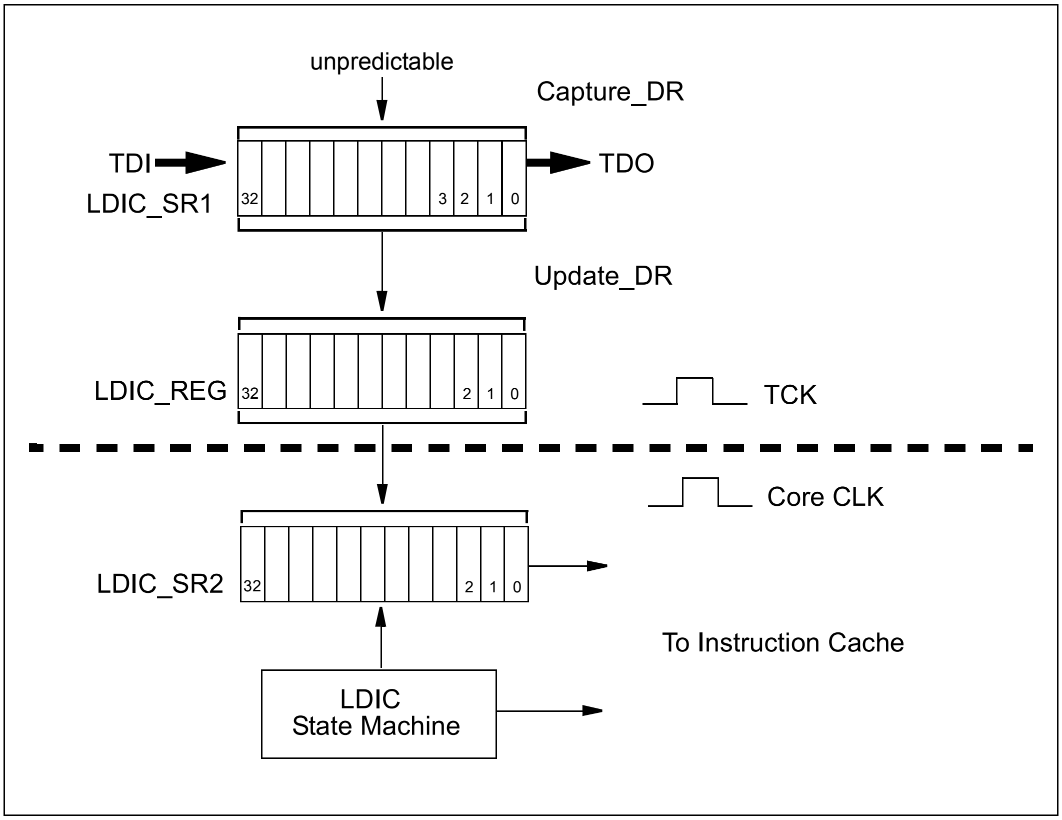
The LDIC JTAG instruction selects the JTAG data register for loading code into the instruction cache. The JTAG opcode for this instruction is ‘00111’. The LDIC instruction must be in the JTAG instruction register in order to load code directly into the instruction cache through JTAG.

[1] A cache line fill from external memory will never be written into the mini-instruction cache. The only way to load a line into the mini-instruction cache is through JTAG.

### 10.13.2 LDIC JTAG Data Register

The LDIC JTAG Data Register is selected when the LDIC JTAG instruction is in the JTAG IR. An external host can load and invalidate lines in the instruction cache through this data register.

Figure 10-9. LDIC JTAG Data Register Hardware



The data loaded into LDIC_SR1 during a Capture_DR is unpredictable.

All LDIC functions and data consists of 33 bit packets which are scanned into LDIC_SR1 during the Shift_DR state.

Update_DR parallel loads LDIC_SR1 into LDIC_REG which is then synchronized with the Intel® XScale™ core clock and loaded into the LDIC_SR2. Once data is loaded into LDIC_SR2, the LDIC State Machine turns on and serially shifts the contents if LDIC_SR2 to the instruction cache.

*Note:* There is a delay from the time of the Update_DR to the time the entire contents of LDIC_SR2 have been shifted to the instruction cache. Removing the LDIC JTAG instruction from the JTAG IR before the entire contents of LDIC_SR2 are sent to the instruction cache will cause unpredictable behavior. Therefore, following the Update_DR for the last LDIC packet, the LDIC instruction must remain in the JTAG IR for a minimum of 15 TCKs. This ensures the last packet is correctly sent to the instruction cache.

## 10.13.3 LDIC Cache Functions

The Intel® XScale™ core supports four cache functions that can be executed through JTAG. Two functions allow an external host to download code into the main instruction cache or the mini instruction cache through JTAG. Two additional functions are supported to allow lines to be invalidated in the instruction cache. The following table shows the cache functions supported through JTAG.

### Table 10-20. LDIC Cache Functions

| Function | Encoding | Arguments | |
|---|---|---|---|
| | | Address | # Data Words |
| Invalidate IC Line | 0b000 | VA of line to invalidate | 0 |
| Invalidate Mini IC | 0b001 | - | 0 |
| Load Main IC | 0b010 | VA of line to load | 8 |
| Load Mini IC | 0b011 | VA of line to load | 8 |
| RESERVED | 0b100-0b111 | - | - |

Invalidate IC line invalidates the line in the instruction cache containing specified virtual address. If the line is not in the cache, the operation has no effect. It does not take any data arguments.

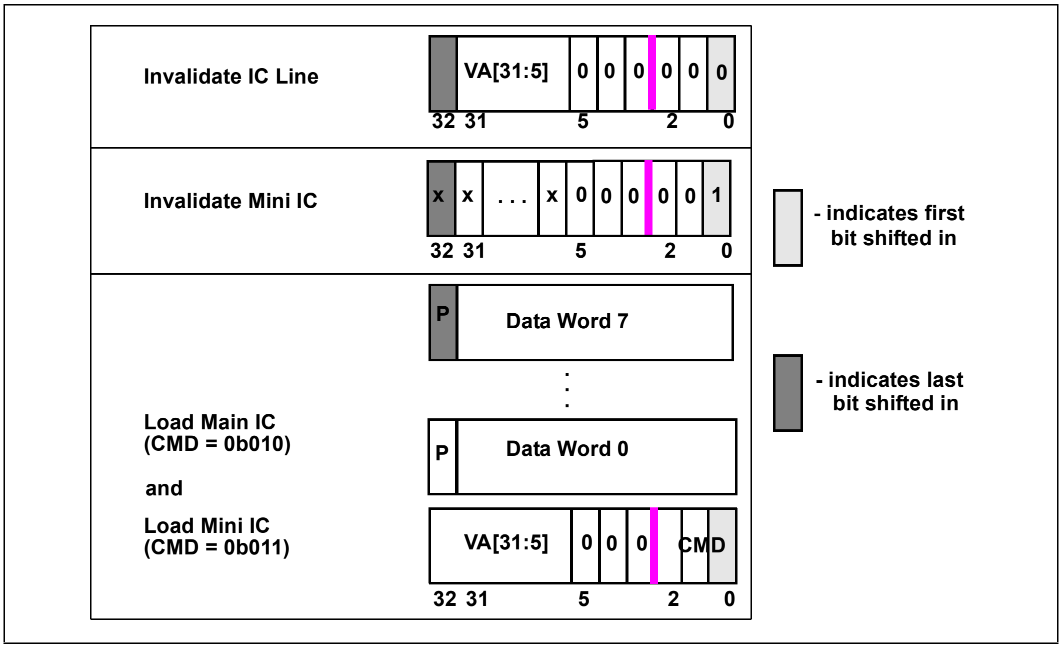
Invalidate Mini IC will invalidate the entire mini instruction cache. It does not effect the main instruction cache. It does not require a virtual address or any data arguments.

Load Main IC and Load Mini IC write one line of data (8 ARM* instructions) into the specified instruction cache at the specified virtual address.

The LDIC Invalidate Mini I-Cache function does not invalidate the BTB (like the CP15 Invalidate IC function) so software must do this manually where appropriate.

Each cache function is downloaded through JTAG in 33 bit packets. Figure 10-10 shows the packet formats for each of the JTAG cache functions. Invalidate IC Line and Invalidate Mini IC each require 1 packet. Load Main IC and Load Mini IC each require 9 packets.

**Figure 10-10. Format of LDIC Cache Functions**



All packets are 33 bits in length. Bits [2:0] of the first packet specify the function to execute. For functions that require an address, bits[32:6] of the first packet specify an 8-word aligned address (Packet1[32:6] = VA[31:5]). For Load Main IC and Load Mini IC, 8 additional data packets are used to specify 8 ARM* instructions to be loaded into the target instruction cache. Bits[31:0] of the data packets contain the data to download. Bit[32] of each data packet is the value of the parity for the data in that packet.

As shown in Figure 10-10, the first bit shifted in TDI is bit 0 of the first packet. After each 33-bit packet, the host must take the JTAG state machine into the Update_DR state. After the host does an Update_DR and returns the JTAG state machine back to the Shift_DR state, the host can immediately begin shifting in the next 33-bit packet.

## 10.13.4 Loading IC During Reset

Code can be downloaded into the instruction cache through JTAG during a processor reset. This feature is used during software debug to download the debug handler prior to starting an application program. The downloaded handler can then intercept the reset vector and do any necessary setup before the application code executes.

Any code downloaded into the instruction cache through JTAG, must be downloaded to addresses that are not already valid in the instruction cache. Failure to meet this requirement will result in unpredictable behavior by the processor. During a processor reset, the instruction cache is typically invalidated, with the exception of the following modes:

- LDIC mode: active when LDIC JTAG instruction is loaded in the JTAG IR; prevents the mini instruction cache and the main instruction cache from being invalidated during reset.

- HALT mode: active when the Halt Mode bit is set in the DCSR; prevents only the mini instruction cache from being invalidated; main instruction cache is invalidated by reset.

During a cold reset (in which both a processor reset and a JTAG reset occurs) it can be guaranteed that the instruction cache will be invalidated since the JTAG reset takes the processor out of any of the modes listed above.

During a warm reset, if a JTAG reset does not occur, the instruction cache is not invalidated by reset when any of the above modes are active. This situation requires special attention if code is downloaded during the warm reset.

***Note:*** While Halt Mode is active, reset can invalidate the main instruction cache. Thus debug handler code downloaded during reset can only be loaded into the mini instruction cache. However, code can be dynamically downloaded into the main instruction cache. (refer to Section 10, "Dynamically Loading IC After Reset").

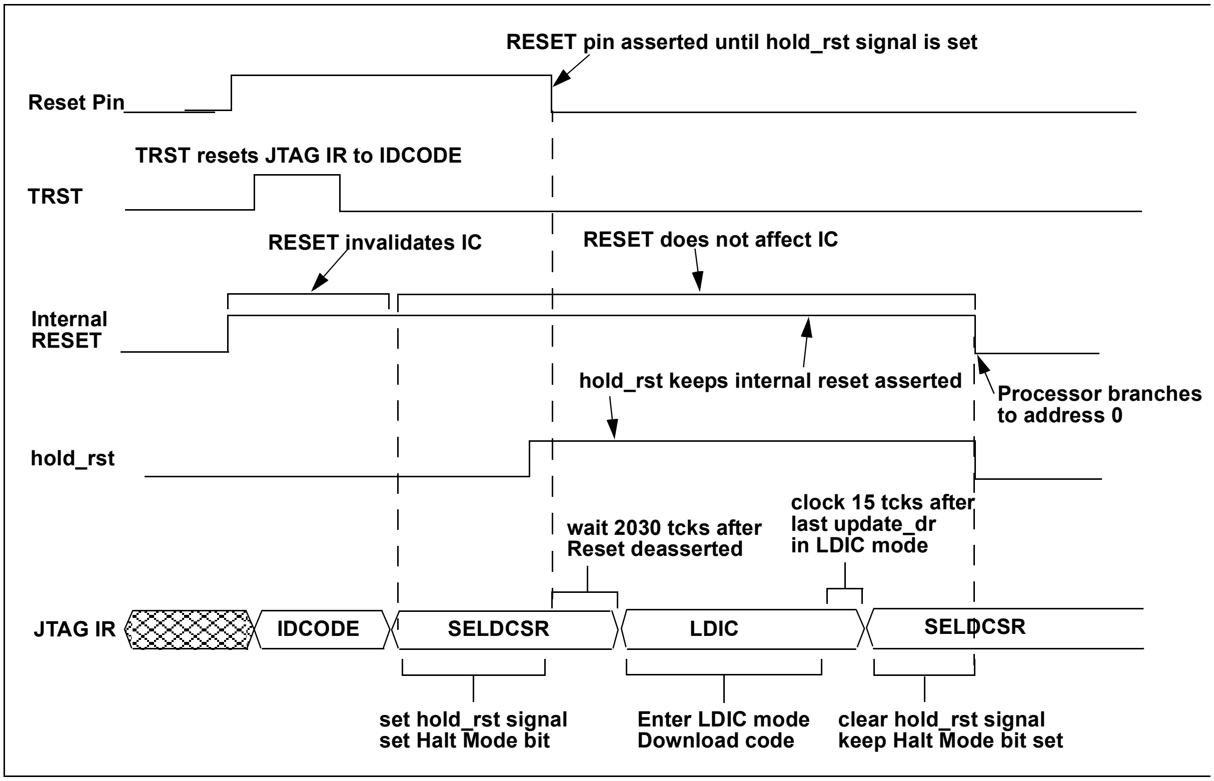
The following sections describe the steps necessary to ensure code is correctly downloaded into the instruction cache.

### 10.13.4.1 Loading IC During Cold Reset for Debug

The Figure 10-11 shows the actions necessary to download code into the instruction cache during a cold reset for debug.

NOTE: In the Figure 10-11 hold_rst is a signal that gets set and cleared through JTAG When the JTAG IR contains the SELDCSR instruction, the hold_rst signal is set to the value scanned into DBG_SR[1].

**Figure 10-11. Code Download During a Cold Reset For Debug**


RESET pin asserted until hold_rst signal is set
Reset Pin
TRST resets JTAG IR to IDCODE
TRST
RESET invalidates IC
RESET does not affect IC
Internal RESET
hold_rst keeps internal reset asserted
Processor branches to address 0
hold_rst
wait 2030 tcks after Reset deasserted
clock 15 tcks after last update_dr in LDIC mode
JTAG IR
IDCODE
SELDCSR
LDIC
SELDCSR
set hold_rst signal set Halt Mode bit
Enter LDIC mode Download code
clear hold_rst signal keep Halt Mode bit set

An external host should take the following steps to load code into the instruction cache following a cold reset:

1. Assert the Reset and TRST pins: This resets the JTAG IR to IDCODE and invalidates the instruction cache (main and mini).
2. Load the SELDCSR JTAG instruction into JTAG IR and scan in a value to set the Halt Mode bit in DCSR and to set the hold_rst signal. For details of the SELDCSR, refer to Section 10.10.2.
3. After hold_rst is set, de-assert the Reset pin. Internally the processor remains held in reset.
4. After Reset is de-asserted, wait 2030 TCKs.
5. Load the LDIC JTAG instruction into JTAG IR.
6. Download code into instruction cache in 33-bit packets as described in Section 10, "LDIC Cache Functions".
7. After code download is complete, clock a minimum of 15 TCKs following the last update_dr in LDIC mode.
8. Place the SELDCSR JTAG instruction into the JTAG IR and scan in a value to clear the hold_rst signal. The Halt Mode bit must remain set to prevent the instruction cache from being invalidated.
9. When hold_rst is cleared, internal reset is de-asserted, and the processor executes the reset vector at address 0.

An additional issue for debug is setting up the reset vector trap. This must be done before the internal reset signal is de-asserted. As described in Section 10.3.3, the Halt Mode and the Trap Reset bits in the DCSR must be set prior to de-asserting reset in order to trap the reset vector. There are two possibilities for setting up the reset vector trap:

- The reset vector trap can be set up before the instruction cache is loaded by scanning in a DCSR value that sets the Trap Reset bit in addition to the Halt Mode bit and the hold_rst signal.
- The reset vector trap can be set up after the instruction cache is loaded. In this case, the DCSR should be set up to do a reset vector trap, with the Halt Mode bit and the hold_rst signal remaining set.

In either case, when the debugger clears the hold_rst bit to de-assert internal reset, the debugger must have already set the Halt Mode and Trap Reset bits in the DCSR.

### 10.13.4.2 Loading IC During a Warm Reset for Debug

Loading the instruction cache during a warm reset is a slightly different situation than during a cold reset. For a warm reset, the main issue is whether the instruction cache gets invalidated by the processor reset or not.

There are several possible scenarios:

- While reset is asserted, TRST is also asserted.

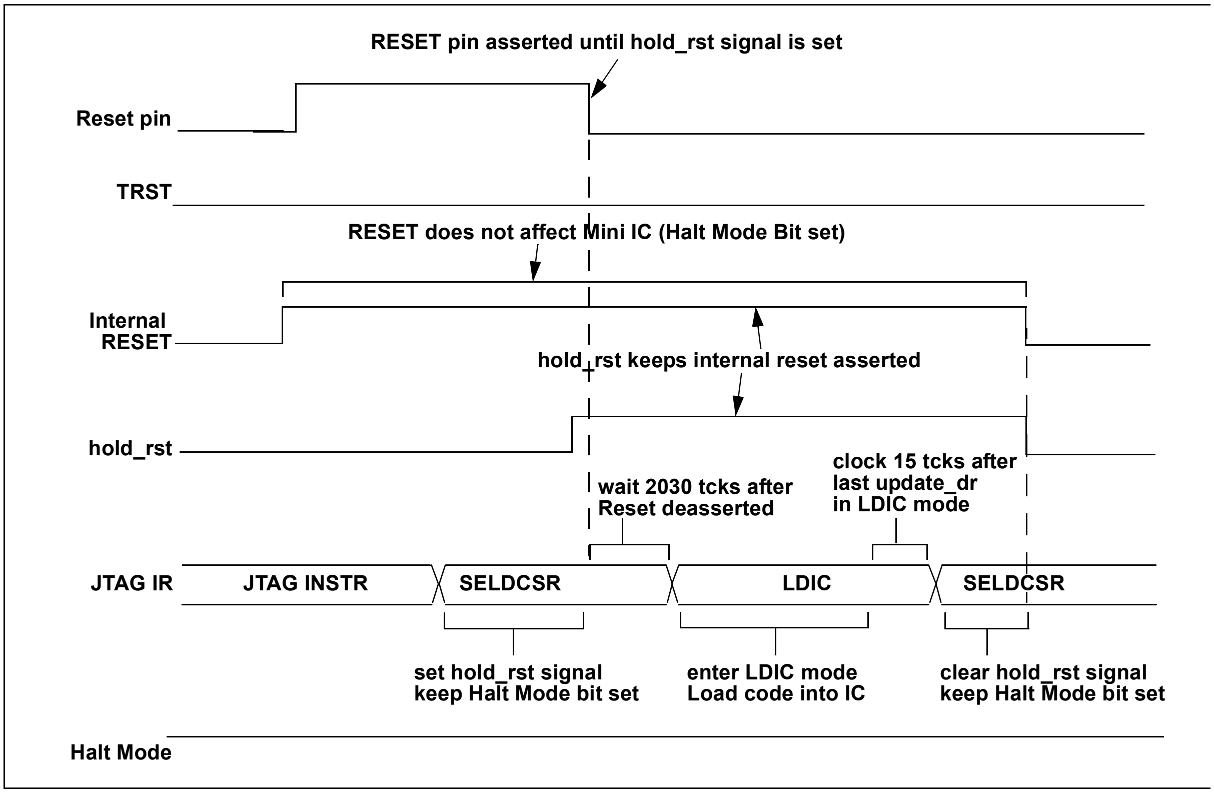
  In this case the instruction cache is invalidated, so the actions taken to download code are identical to those described in Section 10.13.4.1
- When reset is asserted, TRST is not asserted, but the processor is not in Halt Mode.

  In this case, the instruction cache is also invalidated, so the actions are the same as described in Section 10.13.4.1, after the LDIC instruction is loaded into the JTAG IR.
- When reset is asserted, TRST is not asserted, and the processor is in Halt Mode.

  In this last scenario, the mini instruction cache does not get invalidated by reset, since the processor is in Halt Mode. This scenario is described in more detail in this section.

The last scenario described above is shown in Figure 10-12.

Figure 10-12. Code Download During a Warm Reset For Debug


RESET pin asserted until hold_rst signal is set
Reset pin
TRST
RESET does not affect Mini IC (Halt Mode Bit set)
Internal RESET
hold_rst keeps internal reset asserted
hold_rst
wait 2030 tcks after Reset deasserted
clock 15 tcks after last update_dr in LDIC mode
JTAG IR
JTAG INSTR
SELDCSR
LDIC
SELDCSR
set hold_rst signal keep Halt Mode bit set
enter LDIC mode Load code into IC
clear hold_rst signal keep Halt Mode bit set
Halt Mode

As shown in Figure 10-12, reset does not invalidate the instruction cache because the processor is in Halt Mode. Since the instruction cache was not invalidated, it may contain valid lines. The host must avoid downloading code to virtual addresses that are already valid in the instruction cache (mini IC or main IC), otherwise the processor will behave unpredictably.

There are several possible solutions that ensure code is not downloaded to a VA that already exists in the instruction cache.

1) Since the mini instruction cache was not invalidated, any code previously downloaded into the mini IC is valid in the mini IC, so it is not necessary to download the same code again.

If it is necessary to download code into the instruction cache then:

2) Assert TRST, halting the device awaiting activity on the JTAG interface.

3) Clear the Halt Mode bit through JTAG. This allows the instruction cache to be invalidated by reset.

4) Place the LDIC JTAG instruction in the JTAG IR, then proceed with the normal code download, using the Invalidate IC Line function before loading each line. This requires 10 packets to be downloaded per cache line instead of the 9 packets as described in Section 10.13.3

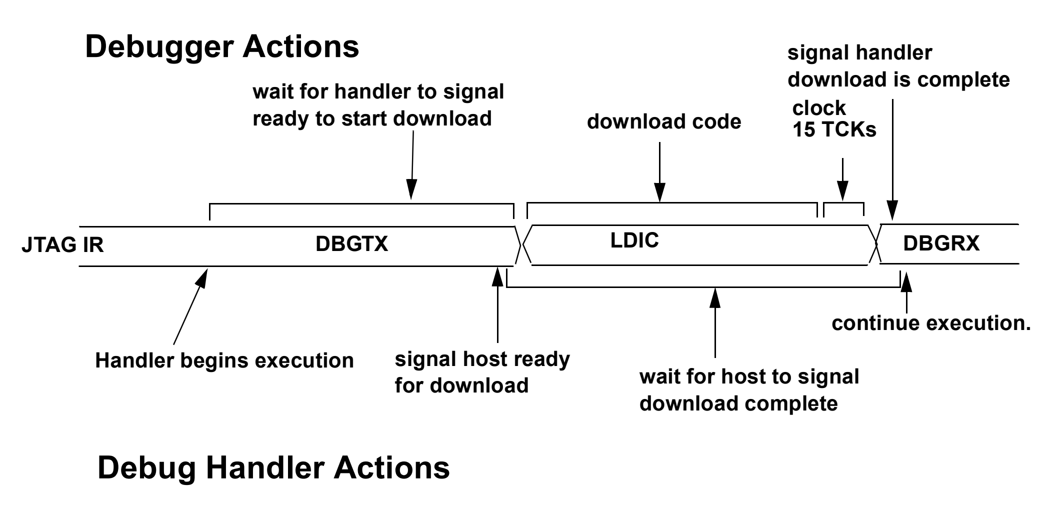
### 10.13.5 Dynamically Loading IC After Reset

An external host can load code into the instruction cache “on the fly” or “dynamically”. This occurs when the host downloads code while the processor is not being reset. However, this requires strict synchronization between the code running on the Intel® XScale™ core and the external host. The guidelines for downloading code during program execution must be followed to ensure proper operation of the processor. The description in this section focuses on using a debug handler running on the Intel® XScale™ core to synchronize with the external host, but the details apply for any application that is running while code is dynamically downloaded.

To dynamically download code during software debug, there must be a minimal debug handler stub, responsible for doing the handshaking with the host, resident in the instruction cache. This debug handler stub can be downloaded into the instruction cache during processor reset using the method described in Section 10.13.4. Section 10, “Dynamic Code Download Synchronization” describes the details for implementing the handshaking in the debug handler.

Figure 10-13 shows a high level view of the actions taken by the host and debug handler during dynamic code download.

**Figure 10-13. Downloading Code in IC During Program Execution**


Debugger Actions
wait for handler to signal ready to start download
download code
signal handler download is complete
clock 15 TCKs
JTAG IR
DBGTX
LDIC
DBGRX
Handler begins execution
signal host ready for download
wait for host to signal download complete
continue execution.
Debug Handler Actions

The following steps describe the details for downloading code:

1. Since the debug handler is responsible for synchronization during the code download, the handler must be executing before the host can begin the download. The debug handler execution starts when the application running on the Intel® XScale™ core generates a debug exception or when the host generates an external debug break.
2. While the DBGTX JTAG instruction is in the JTAG IR (see Section 10, “DBGTX JTAG Command”), the host polls DBG_SR[0], waiting for the debug handler to set it.
3. When the debug handler gets to the point where it is ready to begin the code download, it writes to TX, which automatically sets DBG_SR[0]. This signals the host that it can begin the download. The debug handler then begins polling TXRXCTRL[31] waiting for the host to clear it through the DBGRX JTAG register (to indicate the download is complete).
4. The host writes LDIC to the JTAG IR, and downloads the code. For each line downloaded, the host must invalidate the target line before downloading code to that line. Failure to invalidate a line prior to writing it will cause unpredictable operation by the processor.

5. When the host completes its download, the host must wait a minimum of 15 TCKs, then switch the JTAG IR to DBGRX, and complete the handshaking (by scanning in a value that sets DBG_SR[35]). This clears TXRXCTL[31] and allows the debug handler code to exit the polling loop.
6. After the handler exits the polling loop, it branches to the downloaded code.

***Note:*** The debug handler stub must reside in the instruction cache and execute out of the cache while doing the synchronization. The processor must not be doing any code fetches to external memory while code is being downloaded.

### 10.13.5.1 Dynamic Code Download Synchronization

The following pieces of code are necessary in the debug handler to implement the synchronization used during dynamic code download. The pieces must be ordered in the handler as shown below.

```
# Before the download can start, all outstanding instruction fetches must complete.
# The MCR invalidate IC by line function serves as a barrier instruction in
# the core. All outstanding instruction fetches are guaranteed to complete before
# the next instruction executes.
# NOTE1: the actual address specified to invalidate is implementation defined, but
# must not have any harmful effects.
# NOTE2: The placement of the invalidate code is implementation defined, the only
# requirement is that it must be placed such that by the time the debugger starts
# loading the instruction cache, all outstanding instruction fetches have completed
    mov r5, address
    mcr p15, 0, r5, c7, c5, 1

# The host waits for the debug handler to signal that it is ready for the
# code download. This can be done using the TX register access handshaking
# protocol. The host polls the TR bit through JTAG until it is set, then begins
# the code download. The following MCR does a write to TX, automatically
# setting the TR bit.
# NOTE: The value written to TX is implementation defined.
    mcr p14, 0, r6, c8, c0, 0

# The debug handler waits until the download is complete before continuing. The
# debugger uses the RX handshaking to signal the debug handler when the download
# is complete. The debug handler polls the RR bit until it is set. A debugger write
# to RX automatically sets the RR bit, allowing the handler to proceed.
# NOTE: The value written to RX by the debugger is implementation defined - it can
be a bogus value signalling the handler to continue or it can be a target address
for the handler to branch to.
loop:
    mrc     p14, 0, r15, c14, c0, 0       @ handler waits for signal from debugger
    bpl     loop
    mrc     p14, 0, r0, c8, c0, 0         @ debugger writes target address to RX
    bx      r0
```

In a very simple debug handler stub, the above parts may form the complete handler downloaded during reset (with some handler entry and exit code). When a debug exception occurs, routines can be downloaded as necessary. This allows the entire handler to be dynamic.

Another possibility is for a more complete debug handler to be downloaded during reset. The debug handler may support some operations, such as read memory, write memory, etc. However, other operations, such as reading or writing a group of CP registers, can be downloaded dynamically. This method could be used to dynamically download infrequently used debug handler functions, while the more common operations remain static in the mini-instruction cache.

### 10.13.6 Mini Instruction Cache Overview

The mini instruction cache is a smaller version of the main instruction cache (Refer to Chapter 4 for more details on the main instruction cache). It is a 2KB, 2-way set associative cache. There are 32 sets, each containing two ways; each way contains 8 words. The cache uses the round-robin replacement policy for lines overloaded from the debugger.

Normal application code is never cached in the mini instruction cache on an instruction fetch. The only way to get code into the mini instruction cache is through the JTAG LDIC function. Code downloaded into the mini instruction cache is essentially locked - it cannot be overwritten by application code running on the Intel® XScale™ core. It is not locked against code downloaded through the JTAG LDIC functions.

Application code can invalidate a line in the mini instruction cache using a CP15 Invalidate IC line function to an address that hits in the mini instruction cache. However, a CP15 global invalidate IC function does not affect the mini instruction cache.

The mini instruction cache can be globally invalidated through JTAG by the LDIC Invalidate IC function or by a processor reset when the processor is not in HALT or LDIC mode. A single line in the mini instruction cache can be invalidated through JTAG by the LDIC Invalidate IC-line function.

The mini instruction cache is virtually addressed and addresses may be remapped by the PID. However, since the debug handler executes in Special Debug State, address translation and PID remapping are turned off. For application code, accesses to the mini instruction cache use the normal address translation and PID mechanisms.

## 10.14 Halt Mode Software Protocol

This section describes the overall debug process in Halt Mode. It describes how to start and end a debug session and provides details for implementing a debug handler. Intel may provide a standard Debug Handler that implements some of the techniques in this chapter. This code and other documentation describing additional handler implementation techniques and requirements is intended for manufacturers of debugging tools.

### 10.14.1 Starting a Debug Session

Prior to starting a debug session in Halt Mode, the debugger must download code into the instruction cache during reset, via JTAG. (Section 10, “Downloading Code into the Instruction Cache”). This downloaded code should consist of:

- a debug handler;
- an override default vector table;
- an override relocated vector table (if necessary).

While the processor is still in reset, the debugger sets up the DCSR to trap the reset vector. This causes a debug exception to occur immediately when the processor comes out of reset. Execution is redirected to the debug handler allowing the debugger to perform any necessary initialization. The reset vector trap is the only debug exception that can occur with debug globally disabled (DCSR[31]=0). Therefore, the debugger must also enable debug prior to exiting the handler to ensure all subsequent debug exceptions correctly break to the debug handler.

### 10.14.1.1 Setting up Override Vector Tables

The override default vector table intercepts the reset vector and branches to the debug handler when a debug exception occurs. If the vector table is relocated, the debug vector is relocated to address 0xFFFF_0000. Thus, an override relocated vector table is required to intercept vector 0xFFFF_0000 and branch to the debug handler.

Both override vector tables also intercept the other debug exceptions, so they must be set up to either branch to a debugger specific handler or go to the application's handlers.

It is possible that the application modifies its vector table in memory, so the debugger may not be able to set up the override vector table to branch to the application's handlers. The Debug Handler may be used to work around this problem by reading memory and branching to the appropriate address. Vector traps can be used to get to the debug handler, or the override vector tables can redirect execution to a debug handler routine that examines memory and branches to the application's handler.

### 10.14.1.2 Placing the Handler in Memory

The debug handler is not required to be placed at a specific pre-defined address. However, there are some limitations on where the handler can be placed due to the override vector tables and the 2-way set associative mini instruction cache.

In the override vector table, the reset vector must branch to the debug handler using:

- a direct branch, which limits the start of the handler code to within 32 MB of the reset vector,

  or

- an indirect branch with a data processing instruction. The data processing instruction creates an address using immediate operands and then branches to the target. An **LDR** to the PC does not work because the debugger cannot set up data in memory before starting the debug handler.

The 2-way set associative limitation is due to the fact that when the override default and relocated vector tables are downloaded, they take up both ways of Set 0 (w/ addresses 0x0 and 0xFFFF_0000). Therefore, debug handler code cannot be downloaded to an address that maps into Set 0, otherwise it will overwrite one of the vector tables (avoid addresses w/ lower 12 bits=0).

The instruction cache 2-way set limitation is not a problem when the reset vector uses a direct branch, since the branch offset can be adjusted accordingly. However, it makes using indirect branches more complicated. Now, the reset vector actually needs multiple data processing instructions to create the target address and branch to it.

One possibility is to set up vector traps on the non-reset exception vectors. These vector locations can then be used to extend the reset vector.

Another solution is to have the reset vector do a direct branch to some intermediate code. This intermediate code can then use several instructions to create the debug handler start address and branch to it. This would require another line in the mini instruction cache, since the intermediate code must also be downloaded. This method also requires that the layout of the debug handler be well thought out to avoid the intermediate code overwriting a line of debug handler code, or vice versa.

For the indirect branch cases, a temporary scratch register may be necessary to hold intermediate values while computing the final target address. DBG_r13 can be used for this purpose (see Section 10, "Debug Handler Restrictions" for restrictions on DBG_r13 usage).

## 10.14.2 Implementing a Debug Handler

The debugger uses the debug handler to examine or modify processor state by sending commands and reading data through JTAG. The software interface between the debugger and debug handler is specific to a debugger implementation.

### 10.14.2.1 Debug Handler Entry

When the debugger requests an external debug break or is waiting for an internal break, it then polls the TR bit through JTAG to determine when the processor has entered Debug Mode. The debug handler entry code must do a write to TX to signal the debugger that the processor has entered Debug Mode. The write to TX sets the TR bit, signalling the host that a debug exception has occurred and the processor has entered Debug Mode. The value of the data written to TX is implementation defined (debug break message, contents of register to save on host, etc.).

### 10.14.2.2 Debug Handler Restrictions

The Debug Handler executes in Debug Mode which is similar to other privileged processor modes, however, there are some differences. Following are restrictions on Debug Handler code and differences between Debug Mode and other privileged modes.

- The processor is in Special Debug State following a debug exception, and thus has special functionality as described in Section 10, "Halt Mode".
- Although address translation and PID remapping are disabled for instruction accesses (as defined in Special Debug State), data accesses use the normal address translation and PID remapping mechanisms.
- Debug Mode does not have a dedicated stack pointer, DBG_r13. Although DBG_r13 exists, it is not a general purpose register. Its contents are unpredictable and cannot be relied upon across any instructions or exceptions. However, DBG_r13 can be used, by data processing (non RRX) and **MCR**/**MRC** instructions, as a temporary scratch register.
- The following instructions must not be executed in Debug Mode as they will result in unpredictable behavior:

  LDM

  **LDR** w/ Rd=PC

  **LDR** w/ RRX addressing mode

  SWP

  LDC

  STC

- The handler executes in Debug Mode and can be switched to other modes to access banked registers. The handler must not enter User Mode; any User Mode registers that need to be accessed can be accessed in System Mode. Entering User Mode will cause unpredictable behavior.

### 10.14.2.3 Dynamic Debug Handler

On the Intel® XScale™ core, the debug handler and override vector tables may reside in the 2 KB mini instruction cache, separate from the main instruction cache. A “static” Debug Handler is downloaded during reset. This is the base handler code, necessary to do common operations such as handler entry/exit, parse commands from the debugger, read/write ARM* registers, read/write memory, etc.

Some functions may require large amounts of code or may not be used very often. As long as there is space in the mini-instruction cache, these functions can be downloaded as part of the static Debug Handler. However, if space is limited, the debug handler also has a dynamic capability that allows a function to be downloaded when it is needed. There are three methods for implementing a dynamic debug handler (using the mini instruction cache, main instruction cache, or external memory). Each method has limitations and advantages. Section 10, “Dynamically Loading IC After Reset” describes how to dynamically load the mini or main instruction cache.

1. using the Mini IC

   The static debug handler can support a command which can have functionality dynamically mapped to it. This dynamic command does not have any specific functionality associated with it until the debugger downloads a function into the mini instruction cache. When the debugger sends the dynamic command to the handler, new functionality can be downloaded, or the previously downloaded functionality can be used.

   There are also variations in which the debug handler supports multiple dynamic commands, each mapped to a different dynamic function; or a single dynamic command that can branch to one of several downloaded dynamic functions based on a parameter passed by the debugger.

   Debug Handlers that allow code to be dynamically downloaded into the mini instruction cache must be carefully written to avoid inadvertently overwriting a critical piece of debug handler code. Dynamic code is downloaded to the way pointed to by the round-robin pointer. Thus, it is possible for critical debug handler code to be overwritten, if the pointer does not select the expected way.

   To avoid this problem, the debug handler should be written to avoid placing critical code in either way of a set that is intended for dynamic code download. This allows code to be downloaded into either way, and the only code that is overwritten is the previously downloaded dynamic function. This method requires that space within the mini instruction cache be allocated for dynamic download, limiting the space available for the static Debug Handler. Also, the space available may not be suitable for a larger dynamic function.

   Once downloaded, a dynamic function essentially becomes part of the Debug Handler. If written in the mini instruction cache, it does not get overwritten by application code. It remains in the cache until it is replaced by another dynamic function or the lines where it is downloaded are invalidated.

2. Using the Main IC.

   The steps for downloading dynamic functions into the main instruction cache is similar to downloading into the mini instruction cache. However, using the main instruction cache has its advantages.

   Using the main instruction cache eliminates the problem of inadvertently overwriting static Debug Handler code by writing to the wrong way of a set, since the main and mini instruction caches are separate. The debug handler code does not need to be specially mapped out to avoid this problem. Also, space for dynamic functions does not need to be allocated in the mini instruction cache and dynamic functions are not limited to the size allocated.

The dynamic function can actually be downloaded anywhere in the address space. The debugger specifies the location of the dynamic function by writing the address to RX when it signals to the handler to continue. The debug handler then does a branch-and-link to that address.

If the dynamic function is already downloaded in the main instruction cache, the debugger immediately downloads the address, signalling the handler to continue.

The static Debug Handler only needs to support one dynamic function command. Multiple dynamic functions can be downloaded to different addresses and the debugger uses the function’s address to specify which dynamic function to execute.

Since the dynamic function is being downloaded into the main instruction cache, the downloaded code may overwrite valid application code, and conversely, application code may overwrite the dynamic function. The dynamic function is only guaranteed to be in the cache from the time it is downloaded to the time the debug handler returns to the application (or the debugger overwrites it).

3. External memory

   Dynamic functions can also we downloaded to external memory (or they may already exist there). The debugger can download to external memory using the write-memory commands. Then the debugger executes the dynamic command using the address of the function to identify which function to execute. This method has many of the same advantages as downloading into the main instruction cache.

   Depending on the memory system, this method could be much slower than downloading directly into the instruction cache. Another problem is the application may write to the memory where the function is downloaded. If it can be guaranteed by software design that the application does not modify the downloaded dynamic function, the debug handler can save the time it takes to re-download the code. Otherwise, to ensure the application does not corrupt the dynamic functions, the debugger should re-download any dynamic functions it uses.

For all three methods, the downloaded code executes in the context of the debug handler. The processor will be in Special Debug State, so all of the special functionality applies.

The downloaded functions may also require some common routines from the static debug handler, such as the polling routines for reading RX or writing TX. To simplify the dynamic functions, the debug handler should define a set of registers to contain the addresses of the most commonly used routines. The dynamic functions can then access these routines using indirect branches (BLX). This helps reduce the amount of code in the dynamic function since common routines do not need to be replicated within each dynamic function.

### 10.14.2.4 High-Speed Download

Special debug hardware has been added to support a high-speed download mode to increase the performance of downloads to system memory (vs. writing a block of memory using the standard handshaking).

The basic assumption is that the debug handler can read any data sent by the debugger and write it to memory, before the debugger can send the next data. Thus, in the time it takes for the debugger to scan in the next data word and do an Update_DR, the handler is already in its polling loop, waiting for it. Using this assumption, the debugger does not have to poll RR to see whether the handler has read the previous data - it assumes the previous data has been consumed and immediately starts scanning in the next data word.

The pitfall is when the write to memory stalls long enough that the assumption fails. In this case the download with normal handshaking can be used (or high-speed download can still be used, but a few extra TCKs in the Pause_DR state may be necessary to allow a little more time for the store to complete).

The hardware support for high-speed download includes the Download bit (DCSR[29]) and the Overflow Flag (DCSR[30]).

The download bit acts as a branch flag, signalling to the handler to continue with the download. This removes the need for a counter in the debug handler.

The overflow flag indicates that the debugger attempted to download the next word before the debug handler read the previous word.

More details on the Download bit, Overflow flag and high-speed download, in general, can be found in Section 10, "Transmit/Receive Control Register (TXRXCTRL)".

Following is example code showing how the Download bit and Overflow flag are used in the debug handler:

```
hs_write_word_loop:
hs_write_overflow:
    bl      read_RX                 @ read data word from host

    @@ read TXRXCTRL into the CCs
    mrc      p14, 0, r15, c14, c0, 0
    bcc     hs_write_done       @ if D bit clear, download complete, exit loop.
    beq     hs_write_overflow   @ if overflow detected, loop until host clears D bit

    str     r0, [r6], #4        @ store only if there is no overflow.

    b       hs_write_word_loop      @ get next data word

hs_write_done:
    @@ after the loop, if the overflow flag was set, return error message to host
    moveq   r0, #OVERFLOW_RESPONSE
    beq     send_response
    b       write_common_exit
```

## 10.14.3 Ending a Debug Session

Prior to ending a debug session, the debugger must take the following actions:

1. Clear the DCSR (disable debug, exit Halt Mode, clear all vector traps, disable the trace buffer)
2. turn off all breakpoints;
3. invalidate the mini instruction cache;
4. invalidate the main instruction cache;
5. invalidate the BTB;

These actions ensure that the application program executes correctly after the debugger has been disconnected.

## 10.15 Software Debug Notes

1) Trace buffer message count value on data aborts:

**LDR** to non-PC that aborts gets counted in the exception message. But an **LDR** to the PC that aborts does not get counted as an exception message.

2) Software note on data abort generation in Special Debug State.

1) Avoid code that could generate precise data aborts.

2) If this cannot be done, then handler needs to be written such that a memory access is followed by 1 **NOP**. In this case, certain memory operations must be avoided - **LDM**, **STM**, **STRD**, **LDC**, **SWP**.

3) Data abort on Special Debug State:

When write-back is on for a memory access that causes a data abort, the base register is updated with the write-back value. This is inconsistent with normal (non-SDS) behavior where the base remains unchanged if write-back is on and a data abort occurs.

4) Trace Buffer wraps around and loses data in Halt Mode when configured for fill-once mode:

It is possible to overflow (and lose) data from the trace buffer in fill-once mode, in Halt Mode. When the trace buffer fills up, it has space for 1 indirect branch message (5 bytes) and 1 exception message (1 byte).

If the trace buffer fills up with an indirect branch message and generates a trace buffer full break at the same time as a data abort occurs, the data abort has higher priority, so the processor first goes to the data abort handler. This data abort is placed into the trace buffer without losing any data.

However, if another imprecise data abort is detected at the start of the data abort handler, it will have higher priority than the trace buffer full break, so the processor will go back to the data abort handler. This 2nd data abort also gets written into the trace buffer. This causes the trace buffer to wrap-around and one trace buffer entry is lost (oldest entry is lost). Additional trace buffer entries can be lost if imprecise data aborts continue to be detected before the processor can handle the trace buffer full break (which will turn off the trace buffer).

This trace buffer overflow problem can be avoided by enabling vector traps on data aborts.

5) The TXRXCTRL.OV bit (overflow flag) does not get set during high-speed download when the handler reads the RX register at the same time the debugger writes to it.

If the debugger writes to RX at the same time the handler reads from RX, the handler read returns the newly written data and the previous data is lost. However, in this specific case, the overflow flag does not get set, so the debugger is unaware that the download was not successful.

# Performance Considerations 11

This chapter describes performance considerations that compiler writers, application programmers and system designers need to be aware of to efficiently use the Intel® XScale™ core. Performance numbers discussed here include branch prediction, and instruction latencies.

The timings in this section are specific to the PXA255 processor, and how it implements the ARM* v5TE architecture. This is not a summary of all possible optimizations nor is it an explanation of the ARM* v5TE instruction set. For information on instruction definitions and behavior consult the *ARM* Architecture Reference Manual*.

## 11.1 Branch Prediction

The Intel® XScale™ core implements dynamic branch prediction for the ARM* instructions **B** and **BL** and for the Thumb instruction **B**. Any instruction that specifies the PC as the destination is predicted as not taken, and is not entered into the BTB. For example, an **LDR** or a **MOV** that loads or moves directly to the PC will be predicted not taken and incur a branch latency penalty.

The instructions **B** and **BL (**including Thumb) enter into the branch target buffer when they are *taken* for the first time. A taken branch refers to when they are evaluated to be true. Once in the branch target buffer, the Intel® XScale™ core dynamically predicts the outcome of these instructions based on previous outcomes. Table 11-1 shows the branch latency penalty when these instructions are correctly predicted and when they are not. A penalty of zero for correct prediction means that the Intel® XScale™ core can execute the next instruction in the program flow in the cycle following the branch.

### Table 11-1. Branch Latency Penalty

| Core Clock Cycles | | Description |
|---|---|---|
| ARM* | Thumb | |
| +0 | + 0 | ***Predicted Correctly***. The instruction matches in the branch target buffer and is correctly predicted. |
| +4 | + 5 | ***Mispredicted***. There are three occurrences 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 |

## 11.2 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, and coprocessor.

The load and store addressing modes implemented in the Intel® XScale™ core do not add to the instruction latencies numbers.

The following section explains how to read these tables.

### 11.2.1 Performance Terms

- Issue Clock (cycle 0)

  The first cycle when an instruction is decoded ***and*** allowed to proceed to further stages in the execution pipeline (i.e., 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 first issue clock of the current instruction ***to*** the issue clock of the next instruction. The actual number of cycles can be influenced by cache-misses, resource-dependency stalls, and resource availability conflicts.

- Result Latency

  The cycle distance ***from*** the first issue clock of the current instruction ***to*** the issue clock of the first instruction that can use the result without incurring a resource dependency stall. The actual number of cycles can be influenced by cache-misses, resource-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 first possible issue clock of the next instruction assuming best case conditions (i.e., that the issuing of the next instruction is not stalled due to a resource dependency stall; the next instruction is immediately available from the cache or memory interface; the current instruction does not incur resource dependency stalls during execution that can not be detected at issue time; and if the instruction uses dynamic branch prediction, correct prediction is assumed).

- Minimum Result Latency

  The required minimum cycle distance ***from*** the issue clock of the current instruction ***to*** the issue clock of the first instruction that can use the result without incurring a resource dependency stall assuming best case conditions (i.e., that the issuing of the next instruction is not stalled due to a resource dependency stall; the next instruction is immediately available from the cache or memory interface; and the current instruction does not incur resource dependency stalls during execution that can not be detected at issue time).

- Minimum Issue Latency (with Branch Misprediction)

  The minimum cycle distance ***from*** the issue clock of the current branching instruction ***to*** the first possible issue clock of the next instruction. This definition is identical to *Minimum Issue Latency* except that the branching instruction has been mispredicted. It is calculated by adding

*Minimum Issue Latency (without Branch Misprediction)* to the minimum branch latency penalty number from Table 11-1.

- 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 and is immediately available from the instruction cache or memory interface.

  For the following code fragment, here is an example of computing latencies:

### Example 11-1. Computing Latencies

```
UMLAL    r6,r8,r0,r1
ADD      r9,r10,r11
SUB      r2,r8,r9
MOV      r0,r1
```

Table 11-2 shows how to calculate Issue Latency and Result Latency for each instruction. Looking at the issue column, the **UMLAL** instruction starts to issue on cycle 0 and the next instruction, **ADD,** issues on cycle 2, so the Issue Latency for **UMLAL** is two. From the code fragment, there is a result dependency between the **UMLAL** instruction and the **SUB** instruction. In Table 11-2, **UMLAL** starts to issue at cycle 0 and the **SUB** issues at cycle 5. thus the Result Latency is five.

### Table 11-2. Latency Example

| Cycle | Issue | Executing |
|---|---|---|
| 0 | umlal (1st cycle) | -- |
| 1 | umlal (2nd cycle) | umlal |
| 2 | add | umlal |
| 3 | sub (stalled) | umlal & add |
| 4 | sub (stalled) | umlal |
| 5 | sub | umlal |
| 6 | mov | sub |
| 7 | -- | mov |

## 11.2.2 Branch Instruction Timings

### Table 11-3. Branch Instruction Timings (Those predicted by the BTB)

| Mnemonic | Minimum Issue Latency when Correctly Predicted by the BTB | Minimum Issue Latency with Branch Misprediction |
|---|---|---|
| B | 1 | 5 |
| BL | 1 | 5 |

### Table 11-4. Branch Instruction Timings (Those not predicted by the BTB)

| Mnemonic | Minimum Issue Latency when the branch is not taken | Minimum Issue Latency when the branch is taken |
|---|---|---|
| BLX(1) | N/A | 5 |
| BLX(2) | 1 | 5 |
| BX | 1 | 5 |
| Data Processing Instruction with PC as the destination | Same as Table 11-5 | 4 + numbers in Table 11-5 |
| LDR PC,<> | 2 | 8 |
| LDM with PC in register list | 3 + numreg[a] | 10 + max (0, numreg-3) |

[a] numreg is the number of registers in the register list including the PC.

## 11.2.3 Data Processing Instruction Timings

### Table 11-5. 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[a] | Minimum Issue Latency | Minimum Result Latency[a] |
| ADC | 1 | 1 | 2 | 2 |
| ADD | 1 | 1 | 2 | 2 |
| AND | 1 | 1 | 2 | 2 |
| BIC | 1 | 1 | 2 | 2 |
| CMN | 1 | 1 | 2 | 2 |
| CMP | 1 | 1 | 2 | 2 |
| EOR | 1 | 1 | 2 | 2 |
| MOV | 1 | 1 | 2 | 2 |
| MVN | 1 | 1 | 2 | 2 |
| ORR | 1 | 1 | 2 | 2 |
| RSB | 1 | 1 | 2 | 2 |
| RSC | 1 | 1 | 2 | 2 |
| SBC | 1 | 1 | 2 | 2 |
| SUB | 1 | 1 | 2 | 2 |
| TEQ | 1 | 1 | 2 | 2 |
| TST | 1 | 1 | 2 | 2 |

[a] If the next instruction needs to use the result of the data processing for a shift by immediate or as Rn in a QDADD or QDSUB, one extra cycle of result latency is added to the number listed.

## 11.2.4 Multiply Instruction Timings

### Table 11-6. Multiply Instruction Timings (Sheet 1 of 2)

| Mnemonic | Rs Value (Early Termination) | S-Bit Value | Minimum Issue Latency | Minimum Result Latency[a] | Minimum Resource Latency (Throughput) |
|---|---|---|---|---|---|
| MLA | Rs[31:15] = 0x00000<br>or<br>Rs[31:15] = 0x1FFFF | 0 | 1 | 2 | 1 |
| | | 1 | 2 | 2 | 2 |
| | Rs[31:27] = 0x00<br>or<br>Rs[31:27] = 0x1F | 0 | 1 | 3 | 2 |
| | | 1 | 3 | 3 | 3 |
| | all others | 0 | 1 | 4 | 3 |
| | | 1 | 4 | 4 | 4 |
| MUL | Rs[31:15] = 0x00000<br>or<br>Rs[31:15] = 0x1FFFF | 0 | 1 | 2 | 1 |
| | | 1 | 2 | 2 | 2 |
| | Rs[31:27] = 0x00<br>or<br>Rs[31:27] = 0x1F | 0 | 1 | 3 | 2 |
| | | 1 | 3 | 3 | 3 |
| | all others | 0 | 1 | 4 | 3 |
| | | 1 | 4 | 4 | 4 |
| SMLAL | Rs[31:15] = 0x00000<br>or<br>Rs[31:15] = 0x1FFFF | 0 | 2 | RdLo = 2; RdHi = 3 | 2 |
| | | 1 | 3 | 3 | 3 |
| | Rs[31:27] = 0x00<br>or<br>Rs[31:27] = 0x1F | 0 | 2 | RdLo = 3; RdHi = 4 | 3 |
| | | 1 | 4 | 4 | 4 |
| | all others | 0 | 2 | RdLo = 4; RdHi = 5 | 4 |
| | | 1 | 5 | 5 | 5 |
| 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<br>or<br>Rs[31:15] = 0x1FFFF | 0 | 1 | RdLo = 2; RdHi = 3 | 2 |
| | | 1 | 3 | 3 | 3 |
| | Rs[31:27] = 0x00<br>or<br>Rs[31:27] = 0x1F | 0 | 1 | RdLo = 3; RdHi = 4 | 3 |
| | | 1 | 4 | 4 | 4 |
| | all others | 0 | 1 | RdLo = 4; RdHi = 5 | 4 |
| | | 1 | 5 | 5 | 5 |
| SMULWy | N/A | N/A | 1 | 3 | 2 |
| SMULxy | N/A | N/A | 1 | 2 | 1 |

### Table 11-6. Multiply Instruction Timings (Sheet 2 of 2)

| Mnemonic | Rs Value (Early Termination) | S-Bit Value | Minimum Issue Latency | Minimum Result Latency[a] | Minimum Resource Latency (Throughput) |
|---|---|---|---|---|---|
| UMLAL | Rs[31:15] = 0x00000 | 0 | 2 | RdLo = 2; RdHi = 3 | 2 |
| | | 1 | 3 | 3 | 3 |
| | Rs[31:27] = 0x00 | 0 | 2 | RdLo = 3; RdHi = 4 | 3 |
| | | 1 | 4 | 4 | 4 |
| | all others | 0 | 2 | RdLo = 4; RdHi = 5 | 4 |
| | | 1 | 5 | 5 | 5 |
| UMULL | Rs[31:15] = 0x00000 | 0 | 1 | RdLo = 2; RdHi = 3 | 2 |
| | | 1 | 3 | 3 | 3 |
| | Rs[31:27] = 0x00 | 0 | 1 | RdLo = 3; RdHi = 4 | 3 |
| | | 1 | 4 | 4 | 4 |
| | all others | 0 | 1 | RdLo = 4; RdHi = 5 | 4 |
| | | 1 | 5 | 5 | 5 |

a. If the next instruction needs to use the result of the multiply for a shift by immediate or as Rn in a QDADD or QDSUB, one extra cycle of result latency is added to the number listed.

### Table 11-7. Multiply Implicit Accumulate Instruction Timings

| Mnemonic | Rs Value (Early Termination) | Minimum Issue Latency | Minimum Result Latency | Minimum Resource Latency (Throughput) |
|---|---|---|---|---|
| MIA | Rs[31:16] = 0x0000<br>or<br>Rs[31:16] = 0xFFFF | 1 | 1 | 1 |
| | Rs[31:28] = 0x0<br>or<br>Rs[31:28] = 0xF | 1 | 2 | 2 |
| | all others | 1 | 3 | 3 |
| MIAxy | N/A | 1 | 1 | 1 |
| MIAPH | N/A | 1 | 2 | 2 |

### Table 11-8. Implicit Accumulator Access Instruction Timings

| Mnemonic | Minimum Issue Latency | Minimum Result Latency | Minimum Resource Latency (Throughput) |
|---|---|---|---|
| MAR | 2 | 2 | 2 |
| MRA | 1 | (RdLo = 2; RdHi = 3)[a] | 2 |

a. If the next instruction needs to use the result of the MRA for a shift by immediate or as Rn in a QDADD or QDSUB, one extra cycle of result latency is added to the number listed.

## 11.2.5 Saturated Arithmetic Instructions

Table 11-9. Saturated Data Processing Instruction Timings

| Mnemonic | Minimum Issue Latency | Minimum Result Latency |
|---|---|---|
| QADD | 1 | 2 |
| QSUB | 1 | 2 |
| QDADD | 1 | 2 |
| QDSUB | 1 | 2 |

### 11.2.6 Status Register Access Instructions

Table 11-10. Status Register Access Instruction Timings

| Mnemonic | Minimum Issue Latency | Minimum Result Latency |
|---|---|---|
| MRS | 1 | 2 |
| MSR | 2 (6 if updating mode bits) | 1 |

### 11.2.7 Load/Store Instructions

Table 11-11. Load and Store Instruction Timings

| Mnemonic | Minimum Issue Latency | Minimum Result Latency |
|---|---|---|
| LDR | 1 | 3 for load data; 1 for writeback of base |
| LDRB | 1 | 3 for load data; 1 for writeback of base |
| LDRBT | 1 | 3 for load data; 1 for writeback of base |
| LDRD | 1 (+1 if Rd is R12) | 3 for Rd; 4 for Rd+1; 2 for writeback of base |
| LDRH | 1 | 3 for load data; 1 for writeback of base |
| LDRSB | 1 | 3 for load data; 1 for writeback of base |
| LDRSH | 1 | 3 for load data; 1 for writeback of base |
| LDRT | 1 | 3 for load data; 1 for writeback of base |
| PLD | 1 | N/A |
| STR | 1 | 1 for writeback of base |
| STRB | 1 | 1 for writeback of base |
| STRBT | 1 | 1 for writeback of base |
| STRD | 2 | 1 for writeback of base |
| STRH | 1 | 1 for writeback of base |
| STRT | 1 | 1 for writeback of base |

Table 11-12. Load and Store Multiple Instruction Timings

| Mnemonic | Minimum Issue Latency[a] | Minimum Result Latency |
|---|---|---|
| LDM | 3 - 23 | 1-3 for load data; 1 for writeback of base |
| STM | 3 - 18 | 1 for writeback of base |

[a] LDM issue latency is 7 + N if R15 is in the register list and 2 + N if it is not. STM issue latency is calculated as 2 + N. N is the number of registers to load or store.

## 11.2.8 Semaphore Instructions

Table 11-13. Semaphore Instruction Timings

| Mnemonic | Minimum Issue Latency | Minimum Result Latency |
|---|---|---|
| SWP | 5 | 5 |
| SWPB | 5 | 5 |

## 11.2.9 Coprocessor Instructions

Table 11-14. CP15 Register Access Instruction Timings

| Mnemonic | Minimum Issue Latency | Minimum Result Latency |
|---|---|---|
| MRC | 4 | 4 |
| MCR | 2 | N/A |

Table 11-15. CP14 Register Access Instruction Timings

| Mnemonic | Minimum Issue Latency | Minimum Result Latency |
|---|---|---|
| MRC | 7 | 7 |
| MCR | 7 | N/A |
| LDC | 10 | N/A |
| STC | 7 | N/A |

## 11.2.10 Miscellaneous Instruction Timing

Table 11-16. SWI Instruction Timings

| Mnemonic | Minimum latency to first instruction of SWI exception handler |
|---|---|
| SWI | 6 |

Table 11-17. Count Leading Zeros Instruction Timings

| Mnemonic | Minimum Issue Latency | Minimum Result Latency |
|---|---|---|
| CLZ | 1 | 1 |

### 11.2.11 Thumb Instructions

The timing of Thumb instructions are the same as their equivalent ARM* instructions. This mapping can be found in the *ARM* Architecture Reference Manual*. The only exception is the Thumb BL instruction when H = 0; the timing in this case would be the same as an ARM* data processing instruction.

## 11.3 Interrupt Latency

*Minimum Interrupt Latency* is defined as the minimum number of cycles from the assertion of any interrupt signal (IRQ or FIQ) to the execution of the instruction at the vector for that interrupt. An active system responding to an interrupt will typically depend predominantly on the PXA255 processor's internal & external bus activity.

Assuming best case conditions exist when the interrupt is asserted, e.g., the system isn't waiting on the completion of some other operation, the core will recognize an interrupt approximately 6 core clock cycles after the application processors interrupt controller detects an interrupt.

A sometimes more useful concept to work with is the *Maximum Interrupt Latency*. This is typically a complex calculation that depends on what else is going on in the system at the time the interrupt is asserted. Some examples that can adversely affect interrupt latency are:

- the instruction currently executing could be a 16-register LDM,
- the processor could fault just when the interrupt arrives,
- the processor could be waiting for data from a load, doing a page table walk, etc., and
- high core to system (bus) clock ratios.

Maximum Interrupt Latency can be reduced by:

- ensuring that the interrupt vector and interrupt service routine are resident in the instruction cache. This can be accomplished by locking them down into the cache.
- removing or reducing the occurrences of hardware page table walks. This also can be accomplished by locking down the application's page table entries into the TLBs, along with the page table entry for the interrupt service routine.

# Optimization Guide A

## A.1 Introduction

This document contains optimization techniques for achieving the highest performance from the Intel® XScale™ core architecture. It is written for developers who are optimizing compilers or performance analysis tools for the Intel® XScale™ core based processors. It can also be used by application developers to obtain the best performance from their assembly language code. The optimizations presented in this chapter are based on the Intel® XScale™ core, and hence can be applied to all products that are based on it including the PXA255 processor.

The Intel® XScale™ core architecture includes a superpipelined RISC architecture with an enhanced memory pipeline. The Intel® XScale™ core instruction set is based on ARM* v5 architecture; however, the Intel® XScale™ core includes new instructions. Code generated for the SA-110, SA-1100 and SA-1110 executes on Intel® XScale™ core based processors, however to obtain the maximum performance of your application code, it should be optimized for the Intel® XScale™ core using the techniques presented here.

### A.1.1 About This Guide

This guide assumes that you are 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, "Intel® XScale™ Core Pipeline". This chapter provides an overview of the Intel® XScale™ core pipeline behavior.

Section A.3, "Basic Optimizations". This chapter outlines basic optimizations that can be applied to the Intel® XScale™ core.

Section A.4, "Cache and Prefetch Optimizations". This chapter contains optimizations for efficient use of caches. Also included are optimizations that take advantage of the prefetch instruction of the Intel® XScale™ core.

Section A.5, "Instruction Scheduling". This chapter shows how to optimally schedule code for the Intel® XScale™ core pipeline.

Section A.6, "Optimizations for Size". This chapter contains optimizations that reduce the size of the generated code.

## A.2 Intel® XScale™ Core Pipeline

One of the biggest differences between the Intel® XScale™ core and StrongARM processors is the pipeline. Many of the differences are summarized in Figure A-1. This section provides a brief description of the structure and behavior of the Intel® XScale™ core pipeline.

## A.2.1 General Pipeline Characteristics

While the Intel® XScale™ core pipeline is scalar and single issue, instructions may occupy all three pipelines at once. Out of order completion is possible. The following sections discuss general pipeline characteristics.

### A.2.1.1. Number of Pipeline Stages

The Intel® XScale™ core has a longer pipeline (7 stages versus 5 stages for StrongARM*) which operates at a much higher frequency than its predecessors do. This allows for greater overall performance. The longer Intel® XScale™ core pipeline has several negative consequences, however:

- Larger branch misprediction penalty (4 cycles in the Intel® XScale™ core instead of 1 in StrongARM Architecture). This is mitigated by dynamic branch prediction.
- Larger load use delay (LUD) - LUDs arise from load-use dependencies. A load-use dependency gives rise to a LUD if the result of the load instruction cannot be made available by the pipeline in due time for the subsequent instruction. An optimizing compiler should find independent instructions to fill the slot following the load.
- Certain instructions incur a few extra cycles of delay on the Intel® XScale™ core as compared to StrongARM processors (**LDM**, **STM**).
- Decode and register file lookups are spread out over 2 cycles in the Intel® XScale™ core, instead of 1 cycle in predecessors.

### A.2.1.2. Intel® XScale™ Core Pipeline Organization

The Intel® XScale™ core single-issue superpipeline consists of a main execution pipeline, MAC pipeline, and a memory access pipeline. These are shown in Figure A-1, with the main execution pipeline shaded.

Figure A-1. Intel® XScale™ Core RISC Superpipeline

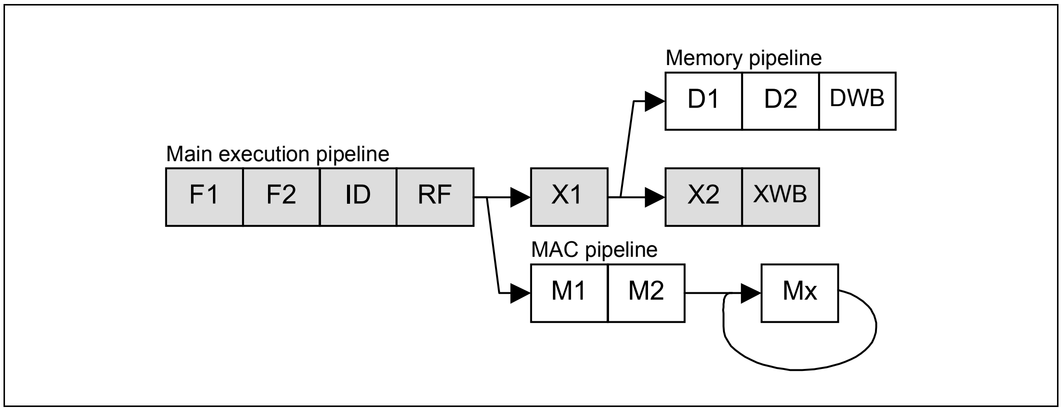


Table A-1 gives a brief description of each pipe-stage.

Table A-1. Pipelines and Pipe stages

| Pipe / Pipestage | Description | Covered In |
|---|---|---|
| Main Execution Pipeline | Handles data processing instructions | Section A.2.3 |
| F1/F2 | Instruction Fetch | Section A.2.3 |
| ID | Instruction Decode | Section A.2.3 |
| RF | Register File / Operand Shifter | Section A.2.3 |
| X1 | ALU Execute | Section A.2.3 |
| X2 | State Execute | Section A.2.3 |
| XWB | Write-back | Section A.2.3 |
| Memory Pipeline | Handles load/store instructions | Section A.2.4 |
| D1/D2 | Data Cache Access | Section A.2.4 |
| DWB | Data cache writeback | Section A.2.4 |
| MAC Pipeline | Handles all multiply instructions | Section A.2.5 |
| M1-M5 | Multiplier stages | Section A.2.5 |
| MWB (not shown) | MAC write-back - may occur during M2-M5 | Section A.2.5 |

### A.2.1.3. Out Of Order Completion

Sequential consistency of instruction execution relates to two aspects: first, to the order in which the instructions are completed; and second, to the order in which memory is accessed due to load and store instructions. The Intel® XScale™ core preserves a weak processor consistency because instructions may complete out of order, provided that no data dependencies exist.

While instructions are issued in-order, the main execution pipeline, memory, and MAC pipelines are not lock-stepped, and, therefore, have different execution times. This means that instructions may finish out of program order. Short ‘younger’ instructions may be 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.

### A.2.1.4. Register Dependencies

In certain situations, the pipeline may need 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.

If no register dependencies exist, the pipeline will not be stalled. For example, if a load operation has missed the data cache, subsequent instructions that do not depend on the load may complete independently.

### A.2.1.5. Use of Bypassing

The Intel® XScale™ core pipeline makes extensive use of bypassing to minimize data hazards. Bypassing allows results forwarding from multiple sources, eliminating the need to stall the pipeline.

## A.2.2 Instruction Flow Through the Pipeline

The Intel® XScale™ core pipeline issues a single instruction per clock cycle. Instruction execution begins at the F1 pipestage and completes at the XWB pipestage.

Although a single instruction may be issued per clock cycle, all three pipelines (MAC, memory, and main execution) may be processing instructions simultaneously. If there are no data hazards, then each instruction may complete 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. ARM* v5 Instruction Execution

Figure A-1 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 may issue 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 at any instant either M1 or X1 will be idle.

All load/store instructions are routed to the memory pipeline after the effective addresses have been calculated in X1.

The ARM* v5 BLX (branch and exchange) instruction, which is used to branch between ARM* and THUMB code, causes the entire pipeline to be flushed (The BLX instruction is not dynamically predicted by the BTB). If the processor is in Thumb mode, then the ID pipestage dynamically expands each Thumb instruction into a normal ARM* v5 RISC instruction and execution resumes as usual.

### A.2.2.2. Pipeline Stalls

The progress of an instruction can stall anywhere in the pipeline. Several pipestages may stall for various reasons. It is important to understand when and how hazards occur in the Intel® XScale™ core pipeline. Performance degradation could be significant if 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, “Branch Target Buffer”) and IFU are important for performance considerations. A summary of operation is provided here so that the reader may understand 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. If 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 (if possible) to the ID. The instruction could come from one of two sources: instruction cache or fill 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 a 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 to:*

- EX for ALU operations
- MAC for multiply operations
- Data Cache for memory writes
- Coprocessor interface

The ID unit decodes the instruction and specifies which registers are accessed in the RFU. Based upon this information, the RFU determines if 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. If no dependencies exist, the RFU will select the appropriate data from the register file and pass it to the next pipestage. When a register dependency does exist, the RFU will keep track of which register is unavailable and when the result is returned, the RFU will stop stalling the pipe.

The ARM* architecture specifies that one of the operands for data processing instructions is the shifter operand, where a 32-bit shift can be 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) Pipestages

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's condition is compared to the CPSR prior to execution of each instruction. Any instruction with a false condition is

cancelled, and will not cause any architectural state changes, including modifications of registers, memory, and PSR.

- Branch target determination - If 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 will restart 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 XWB cycle: PSRs (MRS instruction), ALU output, or other items.

### A.2.3.6. XWB (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, mini-data cache, fill buffers, and writebuffers. The memory pipeline solely handles load and 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 and mini-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 (MAC) unit executes all multiply and multiply-accumulate instructions supported by the Intel® XScale™ core. The MAC implements the 40-bit Intel® XScale™ core 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 may require use of the same datapath resources for several cycles before a new instruction can be accepted. The type of instruction and source arguments determines the number of cycles required.
- No more than two instructions can occupy the MAC pipeline concurrently.
- When the MAC is processing an instruction, another instruction may not enter M1 unless the original instruction completes in the next cycle.
- The MAC unit can operate on 16-bit packed signed data. This reduces register pressure and memory traffic size. Two 16-bit data items can be loaded into a register with one LDR.
- The MAC can achieve 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 11.2, “Instruction Latencies”.

An instruction that occupies the M1 or M2 pipestages will also occupy the X1 and X2 pipestage, respectively. Each cycle, a MAC operation progresses for M1 to M5. A MAC operation may complete anywhere from M2-M5. If a MAC operation enters M3-M5, it is considered committed because it will modify architectural state regardless of subsequent events.

# A.3 Basic Optimizations

This chapter outlines optimizations specific to the ARM* architecture. These optimizations have been modified to suit the Intel® XScale™ core where needed.

## A.3.1 Conditional Instructions

The Intel® XScale™ core architecture provides the ability to execute instructions conditionally. This feature combined with the ability of the Intel® XScale™ core instructions to modify the condition codes makes possible a wide array of optimizations.

### A.3.1.1. Optimizing Condition Checks

The Intel® XScale™ core instructions can selectively modify the state of the condition codes. When generating code for if-else 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 segment:

```
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 can be optimized as follows making use of an 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 can also be used to modify the condition codes. This eliminates the need for a subsequent compare instruction. A conditional branch instruction can then be 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 would look like:

```
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 will need a compare instruction to check for the loop exit condition.

```
for (i = 0; i < 10; i++)
{
    do something;
}
```

If 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 can accurately be 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 the Intel® XScale™ core, a branch misprediction incurs a penalty of four cycles. If the branch is mispredicted 50% of the time, and if we assume 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.

$$\left(\frac{50}{100} \times 4 + \frac{3+4}{2}\right) = 5.5 \qquad cycles\ .$$

If we were to use the Intel® XScale™ core 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 would not incur any branch misprediction penalties and would take three cycles to execute assuming best case conditions. As can be seen, using conditional instructions speeds up execution significantly. However, the use of conditional instructions should be 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 we have 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 we are likely to 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 we have the above data, 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 in which we are 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. If we were to change the code above so as to eliminate the branch instructions by making use of conditional instructions, the if-else part would always take 10 cycles to complete.

If we make the assumptions that both paths are equally likely to be taken and that branches are mispredicted 50% of the time, the costs of using conditional execution Vs using branches can be computed as follows:

Cost of using conditional instructions:

$$1 + \left(\frac{50}{100} \times 10\right) + \left(\frac{50}{100} \times 10\right) = 11 \qquad \textit{cycles}$$

Cost of using branches:

$$1 + \left(\frac{50}{100} \times 7\right) + \left(\frac{50}{100} \times 6\right) + \left(\frac{50}{100} \times 4\right) = 9.5 \qquad \textit{cycles}$$

As can be seen, we get better performance by using branch instructions in the above scenario.

### A.3.1.3. Optimizing Complex Expressions

Conditional instructions should also be 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

The Intel® XScale™ core shift and logical operations provide a useful way of manipulating bit fields. Bit field operations can be 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

The Intel® XScale™ core **MOV** or **MVN** instruction should be used when loading an immediate (constant) value into a register. Please refer to the *ARM* Architecture Reference Manual* for the set of immediate values that can be 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**, and **ADD** instructions. The **LDR** instruction has the potential of incurring a cache miss in addition to polluting the data and instruction caches. The code samples below illustrate cases when a combination of the above instructions can be 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 that it is possible to load any 32-bit value into a register using a sequence of four instructions.

## A.3.4 Optimizing Integer Multiply and Divide

Multiplication by an integer constant should be optimized to make use of the shift operation whenever possible.

```
;Multiplication of R0 by 2^n
    mov   r0, r0, LSL #n
;Multiplication of R0 by 2^n+1
    add   r0, r0, r0, LSL #n
```

Multiplication by an integer constant that can be expressed as $(2^n + 1) \cdot (2^m)$ can similarly be optimized as:

```
;Multiplication of r0 by an integer constant that can be
;expressed as (2^n+1)*(2^m)
    add   r0, r0, r0, LSL #n
    mov   r0, r0, LSL #m
```

Please note that the above optimization should only be used in cases where the multiply operation cannot be advanced far enough to prevent pipeline stalls.

Dividing an unsigned integer by an integer constant should be optimized to make use of the shift operation whenever possible.

```
;Dividing r0 containing an unsigned value by an integer constant
;that can be represented as 2^n
    mov   r0, r0, LSR #n
```

Dividing a signed integer by an integer constant should be optimized to make use of the shift operation whenever possible.

```
;Dividing r0 containing a signed value by an integer constant
;that can be represented as 2^n
    mov   r1, r0, ASR #31
    add   r0, r0, r1, LSR #(32 - n)
    mov   r0, r0, ASR #n
```

The add instruction would stall for 1 cycle. The stall can be prevented by filling in another instruction before add.

### A.3.5 Effective Use of Addressing Modes

The Intel® XScale™ core 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 Reference Manual.* The following code samples illustrate how various kinds of array operations can be 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 Prefetch Optimizations

This chapter considers how to use the various cache memories in all their modes and then examines when and how to use prefetch to improve execution efficiencies.

## A.4.1 Instruction Cache

The Intel® XScale™ core has separate instruction and data caches. Only fetched instructions are held in the instruction cache even though both data and instructions may reside within the same memory space with each other. Functionally, the instruction cache is either enabled or disabled. There is no performance benefit in not using the instruction cache. The exception is that code, which locks code into the instruction cache, must itself execute from non-cached memory.

### A.4.1.1. Cache Miss Cost

The Intel® XScale™ core performance is highly dependent on reducing the cache miss rate. Note that this cache miss penalty becomes significant when the core is running much faster than external memory. Executing non-cached instructions severely curtails the processor's performance in this case and it is very important to do everything possible to minimize cache misses.

### A.4.1.2. Round Robin Replacement Cache Policy

Both the data and the instruction caches use a round robin replacement policy to evict a cache line. The simple consequence of this is that at sometime every line will be evicted, assuming a non-trivial program. The less obvious consequence is that predicting when and over which cache lines evictions take place is very difficult to predict. This information must be gained by experimentation using performance profiling.

### A.4.1.3. Code Placement to Reduce Cache Misses

Code placement can greatly affect cache misses. One way to view the cache is to think of it as 32 sets of 32 bytes, which span an address range of 1024 bytes. When running, the code maps into 32 modular blocks of 1024 bytes of cache space (See Figure 6-1 on page 6-2). Any sets, which are overused, will thrash the cache. The ideal situation is for the software tools to distribute the code on a temporal evenness over this space.

This is very difficult if not impossible for a compiler to do. Most of the input needed to best estimate how to distribute the code will come from profiling followed by compiler based two pass optimizations.

### A.4.1.4. Locking Code into the 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 the round robin replacement policy, eventually the code will be evicted, even if 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. How much code to lock is very application dependent and requires experimentation to optimize.

Code placed into the instruction cache should be aligned on a 1024 byte boundary and placed sequentially together as tightly as possible so as not to waste precious memory space. Making the code sequential also insures even distribution across all cache ways. Though it is possible to choose randomly located functions for cache locking, this approach runs the risk of landing multiple cache ways in one set and few or none in another set. This distribution unevenness can lead to excessive thrashing of the Data and Mini Caches

## A.4.2 Data and Mini Cache

The Intel® XScale™ core allows the user to define memory regions whose cache policies can be set by the user (see Section 6.2.3, “Cache Policies”). Supported policies and configurations are:

- Non Cacheable with no coalescing of memory writes.
- Non Cacheable with coalescing of memory writes.
- Mini-Data cache with write coalescing, read allocate, and write-back caching.
- Mini-Data cache with write coalescing, read allocate, and write-through caching.
- Mini-Data cache with write coalescing, read-write allocate, and write-back caching.
- Data cache with write coalescing, read allocate, and write-back caching.
- Data cache with write coalescing, read allocate, and write-through caching.
- Data cache with write coalescing, read-write allocate, and write-back caching.

To support allocating variables to these various memory regions, the tool chain (compiler, assembler, linker and debugger), must implement these named sections.

The performance of your application code depends on what cache policy you are using for data objects. A description of when to use a particular policy is described below.

The Intel® XScale™ core allows dynamic modification of the cache policies at run time, however, the operation does require considerable processing time and therefore should not be recommended for use by applications.

If the application is running under an OS, then the OS may restrict you from using certain cache policies.

### A.4.2.1. Non Cacheable Regions

It is recommended that non-cache memory (X=0, C=0, and B=0) be used only if necessary as is often necessary for I/O devices. Accessing non-cacheable memory is likely to cause the processor to stall frequently due to the long latency of memory reads.

### A.4.2.2. Write-through and Write-back Cached Memory Regions

Write-through memory regions generate more data traffic on the bus. Therefore use the write-back policy in preference to the write-through policy whenever possible.

In an external DMA environment it may be necessary to use a write through policy where data is shared with external companion devices. In such a situation all shared memory regions should use write through policy to save regular cache cleaning. Memory regions that are private to a particular processor should use the write back policy.

### A.4.2.3. Read Allocate and Read-write Allocate Memory Regions

Most of the regular data and the stack for your application should be allocated to a read-write allocate region. It is expected that you will be writing and reading from them often.

Data that is write only (or data that is written to and subsequently not used for a long time) should be placed in a read allocate region. Under the read-allocate policy if a cache write miss occurs a new cache line will not be allocated, and hence will not evict critical data from the Data cache.

### A.4.2.4. Creating On-chip RAM

Part of the Data cache can be converted into fast on-chip RAM. Access to objects in the on-chip RAM will not incur cache miss penalties, thereby reducing the number of processor stalls. Application performance can be improved by converting a part of the cache into on-chip RAM and allocating frequently used variables to it. Due to the Intel® XScale™ core round-robin replacement policy, all data will eventually be evicted. Therefore to prevent critical or frequently used data from being evicted it should be 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, "Re-configuring the Data Cache as Data RAM" for more details). If the data in the on-chip RAM is to be initialized to zero, then the locking process can be made quicker by using the CP15 prefetch zero function. This function does not generate external memory references.

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. An uneven allocation may increase the level of thrashing in some sets while leaving 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 1024, and the address of arr3 is 2048. All three arrays will be within the same sets, i.e. set0 and set1, as a result three ways in both sets set0 and set1, will be locked, leaving 29 ways for use by other variables.

This can be improved 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.4.2.5. Mini-data Cache

The mini-data cache is best used for data structures, which have short temporal lives, and/or cover vast amounts of data space. Addressing these types of data spaces from the Data cache would corrupt much if not all of the Data cache by evicting valuable data. Eviction of valuable data will reduce performance. Placing this data instead in a Mini-data cache memory region would prevent Data cache corruption while providing the benefits of cached accesses.

A prime example of using the mini-data cache would be for caching the procedure call stack. The stack can be allocated to the mini-data cache so that it's use does not trash the main data cache. This would separate local variables from global data.

Following are examples of data that could be assigned to the mini-data cache:

- The stack space of a frequently occurring interrupt, the stack is used only during the duration of the interrupt, which is usually very small.
- Video buffers, these are usual large and would otherwise more than occupy the main cache allowing for little or no reuse of cached data.
- Streaming data such as Music or Video files that will be read sequentially with little data reuse.

Over use of the Mini-Data cache will thrash the cache. This is easy to do because the Mini-Data cache only has two ways 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 mini-data cache memory region and each is array is aligned on a 1K boundary will quickly thrash the cache.

### A.4.2.6. Data Alignment

Cache lines begin on 32-byte address boundaries. To maximize cache line use and minimize cache pollution, data structures should be aligned on 32 byte boundaries and sized to multiple cache line sizes. Aligning data structures on cache address boundaries simplifies later addition of prefetch instructions to optimize performance.

Not aligning data on cache lines has the disadvantage of moving the prefetch 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++)
{
    PREFETCH(tdata[i+1]);
    tdata[i].ia = tdata[i].ib + tdata[i].ic - tdata[i].id];
    ....
    tdata[i].id = 0;
}
```

In this case if tdata[] is not aligned to a cache line, then the prefetch using the address of tdata[i+1].ia may not include element id. If the array was aligned on a cache line + 12 bytes, then the prefetch would have to be placed on &tdata[i+1].id.

If the structure is not sized to a multiple of the cache line size, then the prefetch address must be advanced appropriately and will require extra prefetch instructions. Consider the following example:

```
struct {
    long ia;
    long ib;
    long ic;
    long id;
    long ie;
} tdata[IMAX];

ADDRESS preadd = tdata

for (i=0, i<IMAX; i++)
{
    PREFETCH(predata+=16);
    tdata[i].ia = tdata[i].ib + tdata[i].ic - tdata[i].id + tdata[i].ie;
    ....
    tdata[i].ie = 0;
}
```

In this case, the prefetch address was advanced by size of half a cache line and every other prefetch instruction is ignored. Further, an additional register is required to track the next prefetch address.

Generally, not aligning and sizing data will add extra computational overhead.

### A.4.2.7. Literal Pools

The Intel® XScale™ core does not have a single instruction that can move all literals (a constant or address) to a register. One technique to load registers with literals in the Intel® XScale™ core 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 as a literal pool. These data blocks are located in the text or code address space so that they can be 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 may be present in both the data and instruction caches, resulting in waste of space.

For maximum efficiency, the compiler should align 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 would be to 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 will be available immediately from the data cache.

## A.4.3 Cache Considerations

### A.4.3.1. 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.4.4.2., "Prefetch Loop Scheduling" below.

### A.4.3.2. Memory Page Thrashing

Memory page thrashing occurs because of the nature of SDRAM. SDRAMs are typically divided into multiple banks. Each bank can have one selected page where a page address size for current memory components is often defined as 4k. Memory lookup time or latency time for a selected page address is currently 2 to 3 bus clocks. Thrashing occurs when subsequent memory accesses within the same memory bank access different pages. The memory page change adds 3 to 4 bus clock cycles to memory latency. This added delay extends the prefetch distance correspondingly making it more difficult to hide memory access latencies. This type of thrashing can be resolved by placing the conflicting data structures into different memory banks or by paralleling the data structures such that the data resides within the same memory page. It is also extremely important to insure that instruction and data sections are in different memory banks, or they will continually trash the memory page selection.

## A.4.4 Prefetch Considerations

The Intel® XScale™ core has a true prefetch load instruction (PLD). The purpose of this instruction is to preload data into the data and mini-data caches. Data prefetching allows hiding of memory transfer latency while the processor continues to execute instructions. The prefetch is important to compiler and assembly code because judicious use of the prefetch instruction can enormously improve throughput performance of the Intel® XScale™ core. Data prefetch can be applied not only to loops but also to any data references within a block of code. Prefetch also applies to data writing when the memory type is enabled as write allocate

The Intel® XScale™ core prefetch load instruction is a true prefetch instruction because the load destination is the data or mini-data cache and not a register. Compilers for processors which have data caches, but do not support prefetch, 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 prefetch can be used to reduce register pressure instead of increasing it.

The prefetch load is a hint instruction and does not guarantee that the data will be loaded. Whenever the load would cause a fault or a table walk, then the processor will ignore the prefetch instruction, the fault or table walk, and continue processing the next instruction. This is particularly advantageous in the case where a linked list or recursive data structure is terminated by a NULL pointer. Prefetching the NULL pointer will not fault program flow.

### A.4.4.1. Prefetch Distances

Scheduling the prefetch instruction requires some understanding of the system latency times and system resources which affect when to use the prefetch instruction. For the PXA255 processor a cache line fill of 8 words from external memory will take more than 10 memory clocks, depending on external RAM speed and system timing configuration. With the core running faster than memory, data from external memory may take many tens of core clocks to load, especially when the data is the last in the cacheline. Thus there can be considerable savings from prefetch loads being used many instructions before the data is referenced.

### A.4.4.2. Prefetch Loop Scheduling

When adding prefetch to a loop which operates on arrays, it may be advantageous to prefetch 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 prefetch ahead is referred to as the prefetch scheduling distance.

It is not always advantageous to add prefetch to a loop. Loop characteristics that limit the use value of prefetch are discussed below.

### A.4.4.3. Compute vs. Data Bus Bound

At the extreme, a loop, which is data bus bound, will not benefit from prefetch because all the system resources to transfer data are quickly allocated and there are no instructions that can profitably be executed. On the other end of the scale, compute bound loops allow complete hiding of all data transfer latencies.

### A.4.4.4. Low Number of Iterations

Loops with very low iteration counts may have the advantages of prefetch completely mitigated. A loop with a small fixed number of iterations may be faster if the loop is completely unrolled rather than trying to schedule prefetch instructions.

### A.4.4.5. Bandwidth Limitations

Overuse of prefetches can usurp resources and degrade performance. This happens because once the bus traffic requests exceed the system resource capacity, the processor stalls. The Intel® XScale™ core data transfer resources are:

4 fill buffers
4 pending buffers
8 half cache line write buffer

SDRAM resources are typically:

1-4 memory banks
1 page buffer per bank referencing a 4K address range
4 transfer request buffers

Consider how these resources work together. A fill buffer is allocated for each cache read miss. A fill buffer is also allocated for each cache write miss if the memory space is write allocate along with a pending buffer. A subsequent read to the same cache line does not require a new fill buffer, but does require a pending buffer and a subsequent write will also require a new pending buffer. A fill buffer is also allocated for each read to a non-cached memory page and a write 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 will use eight write buffers assuming they don't coalesce and two write buffers if they do coalesce. A cache eviction requires a write buffer for each dirty bit set in the cache line. The prefetch instruction requires a fill buffer for each cache line and 0, 1, or 2 write buffers for an eviction.

When adding prefetch instructions, caution must be asserted to insure that the combination of prefetch and instruction bus requests do not exceed the system resource capacity described above or performance will be degraded instead of improved. The important points are to spread prefetch operations over calculations so as to allow bus traffic to free flow and to minimize the number of necessary prefetches.

### A.4.4.6. Cache Memory Considerations

Stride, the way data structures are walked through, can affect the temporal quality of the data and reduce or increase cache conflicts. The Intel® XScale™ core data cache and mini-data caches each have 32 sets of 32 bytes. This means that each cache line in a set is on a modular 1K-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 can be increased because additional registers are required to track prefetch addresses. The effects can be affected by rearranging data structure components to use more parallel accesses to search and compare elements. Similarly rearranging sections of data structures so that sections often written fit in the same half cache line [16 bytes for the Intel® XScale™ core] can reduce cache eviction write-backs. On a global scale, techniques such as array merging can enhance the spatial locality of the data.

As an example of array merging, consider the following code:

```
int a [NMAX];
int b [NMAX];
int 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 can place a and b spatially close.

```
struct {
    int a;
    int b;
} c_arrays;

int 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 arrays 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. If 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 write buffers when the data is written out to memory. However, we can restrict the number of write buffers that are commonly used to 1 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.4.4.7. Cache Blocking

Cache blocking techniques, such as strip-mining, are used to improve temporal locality of the data. Given a large data set that can be reused across multiple passes of a loop, data blocking divides the data into smaller chunks which can be loaded into the cache during the first loop and then be available for processing on subsequent loops thus minimizing cache misses and reducing bus traffic.

As an example of cache blocking consider the following code:

```
for(i=0; i<10000; i++)
    for(j=0; j<10000; j++)
        for(k=0; k<10000; k++)
            C[j][k] += A[i][k] * B[j][i];
```

The variable A[i][k] is completely reused. However, accessing C[j][k] in the j and k loops can displace A[i][k] from the cache. Using blocking the code becomes:

```
for(i=0; i<10000; i++)
    for(j1=0; j<100; j++)
        for(k1=0; k<100; k++)
            for(j2=0; j<100; j++)
                for(k2=0; k<100; k++)
                {
                    j = j1 * 100 + j2;
                    k = k1 * 100 + k2;
                    C[j][k] += A[i][k] * B[j][i];
                }
```

### A.4.4.8. Prefetch Unrolling

When iterating through a loop, data transfer latency can be hidden by prefetching 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 can be resolved by prefetch unrolling. For example consider:

```
for(i=0; i<NMAX; i++)
{
    prefetch(data[i+2]);
    sum += data[i];
}
```

The last two iterations will prefetch superfluous data. The problem can be avoid by unrolling the end of the loop.

```
for(i=0; i<NMAX-2; i++)
{
    prefetch(data[i+2]);
    sum += data[i];
}
sum += data[NMAX-2];
sum += data[NMAX-1];
```

Unfortunately, prefetch loop unrolling does not work on loops with indeterminate iterations.

### A.4.4.9. Pointer Prefetch

Not all looping constructs contain induction variables. However, prefetching techniques can still be 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* can be prefetched to reduce data transfer latency for the next iteration of the loop. Linked lists should be converted to arrays as much as possible.

```
while(p) {
    prefetch(p->next);
    do_something(p->data);
    p = p->next;
}
```

Recursive data structure traversal is another construct where prefetching can be 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* can be prefetched for the next iteration of the loop.

```
preorder(treeNode *t) {
    if(t) {
        prefetch(t->right);
        prefetch(t->left);
        process(t->data);
        preorder(t->left);
        preorder(t->right);
    }
}
```

Note the order reversal of the prefetches in relationship to the usage. If there is a cache conflict and data is evicted from the cache then only the data from the first prefetch is lost.

### A.4.4.10. Loop Interchange

As mentioned earlier, the sequence in which data is accessed affects cache thrashing. Usually, it is best to access data in a contiguous spatially address range. However, arrays of data may have been laid out such that indexed elements are not physically next to each other. Consider the following C code which places array elements in row major order.

```
for(j=0; j<NMAX; j++)
    for(i=0; i<NMAX; i++)
    {
        prefetch(A[i+1][j]);
        sum += A[i][j];
    }
```

In the above example, A[i][j] and A[i+1][j] are not sequentially next to each other. This situation causes an increase in bus traffic when prefetching loop data. In some cases where the loop mathematics are unaffected, the problem can be resolved by induction variable interchange. The above example becomes:

```
for(i=0; i<NMAX; i++)
    for(j=0; j<NMAX; j++)
    {
        prefetch(A[i][j+1]);
        sum += A[i][j];
    }
```

### A.4.4.11. Loop Fusion

Loop fusion is a process of combining multiple loops, which reuse the same data, into one loop. The advantage of this is that the reused data is immediately accessible from the data cache. Consider the following example:

```
for(i=0; i<NMAX; i++)
{
    prefetch(A[i+1], b[i+1], c[i+1]);
    A[i] = b[i] + c[i];
}
for(i=0; i<NMAX; i++)
{
    prefetch(D[i+1], c[i+1], A[i+1]);
    D[i] = A[i] + c[i];
}
```

The second loop reuses the data elements A[i] and c[i]. Fusing the loops together produces:

```
for(i=0; i<NMAX; i++)
{
    prefetch(D[i+1], A[i+1], c[i+1], b[i+1]);
    ai = b[i] + c[i];
    A[i] = ai;
    D[i] = ai + c[i];
}
```

### A.4.4.12. Prefetch to Reduce Register Pressure

Prefetch can be 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 can be used. For example:

```
    ldr   r2, [r0]
; Process code { not yet cached latency > 30 core clocks }
    add   r1, r1, r2
```

In the above case, r2 is unavailable for processing until the add statement. Prefetching the data load frees the register for use. The example code becomes:

```
    pld    [r0] ;prefetch the data keeping r2 available for use
; Process code
    ldr    r2, [r0]
; Process code { ldr result latency is 3 core clocks }
    add    r1, r1, r2
```

With the added prefetch, register r2 can be used for other operations until just before it is needed.

# A.5 Instruction Scheduling

This chapter 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 should be taken to ensure that the rearranged sequence of instructions has the same effect as the original sequence of instructions.

## A.5.1 Scheduling Loads

On the Intel® XScale™ core, an **LDR** instruction has a result latency of 3 cycles assuming the data being loaded is in the data cache. If the instruction after the **LDR** needs to use the result of the load, then it would stall for 2 cycles. If possible, the instructions surrounding the **LDR** instruction should be 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** would stall for 2 cycles because it uses the result of the load. The code can be 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 may not be 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. Noting the conditional behavior, one could 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 if the data being loaded is not in the data cache. To minimize the number of pipeline stalls in such a situation the **LDR** instruction should be moved as far away as possible from the instruction that uses result of the load. Note that this may at times cause certain register values to be spilled to memory due to the increase in register pressure. In such cases, use a prefetch load instruction as a preload hint, to ensure that the data access in the **LDR** instruction hits the cache when it executes. A **PLD** instruction should be used in cases where we can be sure that the load instruction would be executed. 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 can be moved before the **MOV** instruction. Note that this would prevent pipeline stalls if the load hits the data cache. However, if 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 would change the program semantics. It is possible to move the **ADD** and the **LDR** instructions before the **SUB** instruction if we allow 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 can be 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
```

The Intel® XScale™ core has 4 fill-buffers that are used to fetch data from external memory when a data-cache miss occurs. The Intel® XScale™ core stalls when all fill buffers are in use. This happens when more than 4 loads are outstanding and are being fetched from memory. As a result, the code written should ensure that no more than 4 loads are outstanding at the same time. For example, the number of loads issued sequentially should not exceed 4. Also note that a preload instruction may cause a fill buffer to be used. As a result, the number of preload instructions outstanding should also be considered to derive how many loads are simultaneously outstanding.

Similarly, the number of write buffers also limits the number of successive writes that can be issued before the processor stalls. No more than eight stores can be issued. Also note that if the data caches are using the write-allocate with writeback policy, then a load operation may cause stores to the external memory if the read operation evicts a cache line that is dirty (modified). The number of sequential stores may be further limited by these other writes.

### A.5.1.1. Scheduling Load and Store Double (LDRD/STRD)

The Intel® XScale™ core introduces two new double word instructions: **LDRD** and **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 may be used:

- the effective address must be aligned on an 8-byte boundary
- the specified register must be even (r0, r2, etc.).

If 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 issues in four clock cycles. Avoid **LDRD**s targeting R12; this incurs an extra cycle of issue latency.

The **LDRD** instruction has a result latency of 3 or 4 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 would load 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 would stall for 3 cycles because of the 4 cycle result latency for the second destination register of an **LDRD** instruction. The code shown above can be rearranged to remove the pipeline stalls:

```
; The following ldrd instruction would load 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 memory operation following a **LDRD** instruction (**LDR**, **LDRD**, **STR** and so on) would stall for 1 cycle. This stall time could be used to execute a data processing instruction.

```
; The str instruction below would stall for 1 cycle
    ldrd  r0, [r3]
    str   r4, [r5]
```

## A.5.1.2. Scheduling Load and Store Multiple (LDM/STM)

**LDM** and **STM** instructions have an issue latency of 2-20 cycles depending on the number of registers being loaded or stored. The issue latency is typically 2 cycles plus an additional cycle for each of the registers being loaded or stored assuming a data cache hit. The instruction following an **LDM** would stall 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 should be 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 can be 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
    ldm   r0, {r2, r3}
    ldm   r1, {r4, r5}
    adds  r0, r2, r4
    adc   r1, r3, r5
```

If the code were written as shown above, assuming all the accesses hit the cache, the code would take 11 cycles to complete. Rewriting the code as shown below using **LDRD** instruction would take only 7 cycles to complete. The performance would increase further if we can fill in other instructions 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 5 cycles to complete.

```
    stm   r0, {r2, r3}
    add   r1, r1, #1
```

The alternative version which is shown below would only take 3 cycles to complete.

```
    strd  r2, [r0]
    add   r1, r1, #1
```

## A.5.2 Scheduling Data Processing Instructions

Most Intel® XScale™ core data processing instructions have a result latency of 1 cycle. This means that the current instruction is able to use the result from the previous data processing instruction. However, the result latency is 2 cycles if 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 would incur a 1 cycle stall for the **MOV** instruction:

```
sub   r6, r7, r8
add   r1, r2, r3
mov   r4, r1, LSL #2
```

The code above can be rearranged as follows to remove the 1 cycle stall:

```
add   r1, r2, r3
sub   r6, r7, r8
mov   r4, r1, LSL #2
```

All data processing instructions incur a 2 cycle issue penalty and a 2 cycle result penalty when the shifter operand is a shift/rotate by a register or shifter operand is RRX. Since the next instruction would always incur a 2 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 would incur a 1 cycle stall due to the issue latency of the add instruction as the shifter operand is shift by a register. The issue latency can be 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 can cause pipeline stalls due to either resource conflicts or result latencies. The following code segment would incur a stall of 0-3 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 would incur a stall of 1-3 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 4 cycle multiply operation that sets the condition codes behaves the same as a 4 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 would stall for 3 cycles if the multiply takes 4 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 11.2, “Instruction Latencies” to get the instruction latencies for the multiply instructions. The multiply instructions should be scheduled taking into consideration these instruction latencies.

## A.5.4 Scheduling SWP and SWPB Instructions

The **SWP** and **SWPB** instructions have a 5 cycle issue latency. As a result of this latency, the instruction following the **SWP**/**SWPB** instruction would stall for 4 cycles. **SWP** and **SWPB** instructions should, therefore, be used only where absolutely needed.

For example, the following code may be used to swap the contents of 2 memory locations:

```
; Swap the contents of memory locations pointed to by r0 and r1
    ldr   r2, [r0]
    swp   r2, [r1]
    str   r2, [r1]
```

The code above takes 9 cycles to complete. The rewritten code below, takes 6 cycles to execute; assuming the availability of r3.

```
; 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 1 cycle, a result latency of 2 or 3 cycles depending on the destination register value being accessed and a resource latency of 2 cycles.

Consider the code sample:

```
    mra   r6, r7, acc0
    mra   r8, r9, acc0
    add   r1, r1, #1
```

The code shown above would incur a 1-cycle stall due to the 2-cycle resource latency of an **MRA** instruction. The code can be 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 would incur a 2 cycle penalty due to the 3-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 can be 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 latency, a result latency, and a resource latency of 2 cycles. Due to the 2-cycle issue latency, the pipeline would always stall for 1 cycle following a **MAR** instruction. The use of the **MAR** instruction should, therefore, be used only where absolutely necessary.

## A.5.6 Scheduling the MIA and MIAPH Instructions

The **MIA** instruction has an issue latency of 1 cycle. The result and resource latency can vary from 1 to 3 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 can stall from 0 to 2 cycles depending on the values in the registers r2 and r3 due to the 1 to 3 cycle resource latency.

Similarly, consider the following code sample:

```
mia   acc0, r2, r3
mra   r4, r5, acc0
```

The **MRA** instruction above can stall from 0 to 2 cycles depending on the values in the registers r2 and r3 due to the 1 to 3 cycle result latency.

The **MIAPH** instruction has an issue latency of 1 cycle, result latency of 2 cycles and a resource latency of 2 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 would stall for 1-cycle due to a 2-cycle resource latency. The **MRA** instruction would stall for 1-cycle due to a 2-cycle result latency. These stalls can be 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 1 cycle and a result latency of 2 cycles. The **MSR** instruction has an issue latency of 2 cycles (6 if updating the mode bits) and a result latency of 1 cycle.

Consider the code sample:

```
    mrs   r0, cpsr
    orr   r0, r0, #1
    add   r1, r2, r3
```

The **ORR** instruction above would incur a 1 cycle stall due to the 2-cycle result latency of the **MRS** instruction. In the code example above, the **ADD** instruction can be moved before the **ORR** instruction to prevent this stall.

### A.5.8 Scheduling Coprocessor Instructions

The **MRC** instruction has an issue latency of 1 cycle and a result latency of 3 cycles. The **MCR** instruction has an issue latency of 1 cycle.

Consider the code sample:

```
    add   r1, r2, r3
    mrc   p15, 0, r7, C1, C0, 0
    mov   r0, r7
    add   r1, r1, #1
```

The **MOV** instruction above would incur a 2-cycle latency due to the 3-cycle result latency of the **MRC** instruction. The code shown above can be rearranged as follows to avoid these stalls:

```
    mrc   p15, 0, r7, C1, C0, 0
    add   r1, r2, r3
    add   r1, r1, #1
    mov   r0, r7
```

## A.6 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 will, in general, lower the performance of your application. These are some techniques for optimizing for code size using the Intel® XScale™ core instruction set.

Many optimizations mentioned in the previous chapters improve the performance of ARM* code. However, using these instructions will result in increased code size. Use the following optimizations to reduce the space requirements of the application code.

### A.6.1 Multiple Word Load and Store

The **LDM**/**STM** instructions are one word long and let you load or store multiple registers at once. Use the **LDM**/**STM** instructions instead of a sequence of loads/stores to consecutive addresses in memory whenever possible.

### A.6.2 Use of Conditional Instructions

Using conditional instructions to expand if-then-else statements as described in Section A.3.1, “Conditional Instructions” may result in increasing or decreasing the size of the generated code. Compare the savings made by any removal of branch instructions to determine whether conditional execution reduces code size. If the conditional components of both the ‘if’ and ‘else’ are more than two instructions it would be more compact code to use branch instructions instead.

### A.6.3 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 them will not change the behavior of your code, therefore, you should avoid using these instructions when optimizing for space.

### A.6.4 Thumb Instructions

The best opportunity for code compaction is to utilize the ARM* Thumb instructions. These instructions are additions to the ARM* architecture primarily for the purpose of code size reduction.

16-bit Thumb instructions have less functionality than their 32-bit equivalents, hence Thumb code is typically slower than 32-bit ARM* code. However, in some unusual cases where Instruction Cache size is a significant influence, being able to hold more Thumb instructions in cache may aid performance. Whatever the performance outcome, Thumb coding significantly reduces code size.