Intel® XScale™ Microarchitecture

Programmers Reference Manual

February 2001

Order Number: 273436-001

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications.

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

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 htel® XScale™ microarchitecture may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

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

Copies of documents which have an 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

This Programmers Reference Manual documents all programming issues that are common to Intel® XScale™ core (ARM* architecture compliant) for the second-generation core of the ARM microprocessor family.

Intel Corporation assumes no responsibility for any errors that 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, feature, timings, and pin-out descriptions do not imply a commitment to implement them.

This chapter presents the following information:

- Product Overview: Presents a brief product description.
- About This Manual: Provides a list of all chapters, including chapter descriptions.
- Features and Benefits of the Intel® XScale™ microarchitecture: Summarizes the features and benefits of Intel® XScale™ microarchitecture.
- Related Information: Presents a list of all other related information including related documents, both hardcopy and online.

1.1 Product Overview

The Intel® XScale™ core is based on ARM processor family second-generation core and consists of innovative custom circuits, a proprietary design, and proprietary process techniques. This unique core enables ASSPs to operate on very low current while in Run and Low-power modes.

Designed to enable high performance, low power consumption, and systems integration, the Intel® XScale™ core empowers OEMs to develop smaller, more cost-effective, hand-held devices with longer battery life, while still providing the performance to run MIPS-intensive multimedia applications such as audio encode/decode, video compression, and speech.

The Intel® XScale™ microarchitecture extends to set-top boxes, networking, intelligent I/O, and remote-access servers. This unique processor engine design affords a substantial leadership position in the hand-held device market segment where *high performance*, *low power*, and *integration-per-cost effectiveness* are all critical factors.

The Intel® XScale™ core targets the portable information device segment, which consists of feature-rich hand-held devices such as (but not limited to) the following:

- Smart phones/3G+ multimedia phones
- PC companions
- Palm-size devices
- Vertical application devices

The processor is also packaged in a “smaller footprint, lower cost” version focused on palms and smart phones, and a “higher performance, higher cost” version for the PC companion and vertical application device segments. In addition to hand-held segments, the Intel® XScale™ core also provides a market entry to tethered applications such as screen phones, low-end settop boxes, web terminals, and other Internet appliances.

1.1.1 Features and Benefits of Intel® XScale™ Microarchitecture

- *Superpipelined RISC Technology* — Achieve high speed and ultra-low power with a 7-stage integer/8-stage memory superpipelined core.
- *Dynamic Voltage Management* — Obtain the right blend of performance and power with dynamic voltage and frequency scaling "on the fly."
- *Media Processing Technology* — Achieve efficient media processing with a multiply-accumulate coprocessor that performs two simultaneous 16-bit SIMD multiplies with 40-bit accumulation.
- *Power Management Unit* — Save power with idle, sleep, and quick wake-up modes.
- *128-entry Branch Target Buffer* — Maintain pipeline capacity with statistically correct branch choices
- *32 KB Instruction Cache* — Achieve high performance and low power consumption levels by keeping a local copy of important instructions.
- *2 KB Data Cache* — Avoid "thrashing" of the Data cache for frequently changing data streams.
- *32-Entry Instruction Memory Management Unit* — Enable logical-to-physical address translation, access permissions, and instruction-cache attributes.
- *Four Entry Fill and Pend Buffers* — Obtain core efficiency by allowing non-blocking and "hit-under-miss" operation with Data caches.
- *Performance Monitoring Unit* — Analyze hit rates with two 32-bit event counters and one 32-bit cycle counter.
- *Debug Unit* — Debug programs with hardware breakpoints and a 256-entry trace-history buffer (for flow change messages).
- *32-bit Coprocessor Interface* — Achieve a high-performance interface between the core and coprocessors.
- *64-bit Core Memory Bus with Simultaneous 32-bit Input Path and 32-bit Output Path* — Obtain up to 4.8 GBytes/sec @ 600 MHz bandwidth for internal accesses.
- *8-Entry Write Buffer* — Provides continuous core execution while data is written to memory.
- *Thumb* Instruction Set Supported* — Select the 16-bit Thumb instruction set from the current Program Status register.

1.2 About This Manual

The following chapters are provided in this manual:

- Chapter 1, "Introduction" – Introduces the processor and the manual contents.
- Chapter 2, "Programming Model" — Defines and describes the Intel® XScale™ microarchitecture programming model.
- Chapter 3, "Optimization Techniques" – Provides easy-to-use procedures on how to optimize your system including the optimization of Cache and Prefetch memories, your C library, and instruction-scheduling routines.
- Appendix A, "For OS Developers" — Describes features and implementation details that drive the operating systems issues.

1.3 Related Information

- *ARM ArchitectureReference Manual* — http://www.arm.com/Documentation/TRM/
- *Intel® XScale™ Microarchitecture Product Brief* — http://developer.intel.com/design/intelxscale/xscaleproductbriefweb.pdf
- *Intel® XScale™ Microarchitecture Technical Summary* — http://developer.intel.com/design/intelxscale/XScaleDatasheet4.htm
- *Intel® XScale™ Core Benchmarks* — http://developer.intel.com/design/intelxscale/benchmarks.htm

Programming Model 2

This chapter describes the programming model of Intel® XScale™ microarchitecture (ARM* architecture compliant), namely the implementation options and extensions to the ARM Version 5TE ISA programming model. The Intel® XScale™ core handles 8-, 16-, and 32-bit data types, and operates in one of seven processor modes (User, System, Supervisor, Abort, Undefined instruction, Fast interrupt, and Normal interrupt). The microarchitecture provides 16 general 32-bit registers (R0-R15), where R13 is the Stack pointer (SP), R14 is the Link register (LR), and R15 is the Program counter (PC). It also supplements the 16 registers (in addition to a Current Program Status register, CPSR) with 20 mode-dependent "shadow" registers.

2.1 ARM Architecture Compatibility

The Intel® XScale™ microarchitecture 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, specifically the addition of tiny pages (1 Kbyte), a new instruction (**CLZ**) that counts the leading zeroes in a data value, enhanced ARM-Thumb transfer instructions, and 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™ microarchitecture 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. To run in Big-Endian mode, the B-bit must be set before attempting any sub-word accesses to memory, or undefined results will occur. Note that this bit takes effect even if the MMU is disabled.

2.2.2 26-Bit Code

The Intel® XScale™ microarchitecture does not support 26-bit code.

2.2.3 Thumb

The Intel® XScale™ microarchitecture supports the 16-bit V5 Thumb instruction set.

2.2.4 ARM DSP-Enhanced Instruction Set

The Intel® XScale™ microarchitecture implements the ARM DSP-enhanced instruction set, which is a set of instructions that boosts the performance of signal-processing applications. New Multiply instructions operate on 16-bit data values, and new Saturation instructions are available as well (see below).

- SMLAxy 32<=16x16+32
- SMLAWy 32<=32x16+32
- SMLALxy 64<=16x16+64
- SMULxy 32<=16x16
- SMULWy 32<=32x16
- QADD adds two registers and saturates the result if an overflow occurred
- QDADD doubles and saturates one of the input registers then add and saturate
- QSUB subtracts two registers and saturates the result if an overflow occurred
- QDSUB doubles and saturates one of the input registers then subtract and saturate

The Intel® XScale™ microarchitecture 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. If the targeted cache line is already resident, this instruction has no affect.
- Both LDRD and STRD instructions will generate an alignment exception when the address bits [2:0] = 0b100.

MCRR and MRRC are supported in Intel® XScale™ microarchitecture only when directed to Coprocessor 0, and are used to access the internal accumulator. See Section 2.3.1.2 for more information. Access to any other coprocessor besides 0x0 is undefined.

2.2.5 Base Register Update

If a Data abort is signalled on a memory instruction that specifies Write-back, the contents of the Base register will not be updated. This behavior holds for all Load and Store instructions, and matches that of the first-generation Intel® StrongARM* processor (referred to in the ARM V5 architecture as the *Base Restored Abort Model*).

2.3 Extensions to ARM* Architecture

The Intel® XScale™ microarchitecture includes 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 that are discussed in the next subsections.

- A DSP coprocessor (CP0) has been added that contains a 40-bit accumulator and eight new instructions.
- New page attributes were added to the page table descriptors. The C- and B-page attribute encoding was extended by one additional bit to allow for more encodings: Write-Allocate and Mini-Data cache. An attribute specifying ECC for 1 MB regions was also added.
- Additional functionality has been added to Coprocessor 15, Coprocessor 14 also added.
- 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™ microarchitecture adds a DSP coprocessor to the architecture for increasing the performance and the precision of audio-processing algorithms. This coprocessor contains a 40-bit accumulator and eight 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. MAR and MRA 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. Note that only privileged software can set this bit in the Coprocessor Access register.

Two new instruction formats were added for coprocessor 0: Multiply with Internal Accumulate Format, and Internal Accumulate Access Format.

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” shows the layout of the new format. The opcode for this format lies within the 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	Intel® XScale™ microarchitecture defines the following: 0b0000 = **MIA** 0b1000 = **MIAPH** 0b1100 = **MIABB** 0b1101 = **MIABT** 0b1110 = **MIATB** 0b1111 = **MIATT** The effect of all other encodings are unpredictable.
15:12	**Rs** - Multiplier	
7:5	**acc** - select 1 of 8 accumulators	Intel® XScale™ microarchitecture 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 one of eight internal accumulators to operate on, and *opcode_3* defines the operation for this format. The Intel® XScale™ microarchitecture defines a single 40-bit accumulator referred to as *acc0;* future implementations may define multiple internal accumulators. The Intel® XScale™ microarchitecture 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.
            Specifying R15 for register Rs or Rm has unpredictable results.
            acc0 is defined to be 0b000 on IIntel® XScale™ microarchitecture.
```

The MIA instruction operates similarly to MLA except that it uses the 40-bit accumulator. 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 executed only 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[31:16] * Rs[31:16]) +
                       sign_extend(Rm[15:0] * Rs[15:0]) +
                       acc0[39:0]
Exceptions: none
Qualifiers  Condition Code
            S bit is always cleared; no condition code flags are updated
Notes:      Specifying R15 for register Rs or Rm has unpredictable results.
            acc0 is defined to be 0b000 on Intel® XScale™ microarchitecture
```

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

The instruction is executed only 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
            S bit is always cleared; no condition code flags are updated
Notes:      Specifying R15 for register Rs or Rm has unpredictable results.
            acc0 is defined to be 0b000 on Intel® XScale™ microarchitecture
```

The MIAxy instruction performs one16-bit signed multiply, and accumulates these to a single 40-bit accumulator. The 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 that is specified in the mnemonic as T (for top). A value of 0x0 will select bits [15:0] of the register that 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 executed only if the condition specified in the instruction matches the Condition-code status.

2.3.1.2 Internal Accumulator Access Format

The Intel® XScale™ microarchitecture defines a new instruction format for accessing internal accumulators in CP0. Table 2-5, “Internal Accumulator Access Format” on page 2-19 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 the registers and an internal accumulator. The *acc* field specifies one of eight internal accumulators to transfer data to/from. The Intel® XScale™ microarchitecture implements a single 40-bit accumulator referred to as *acc0*; future implementations can specify multiple internal accumulators of varying sizes, up to 64 bits. Access to the internal accumulator is allowed in all processor modes (user and privileged) as long bit 0 of the Coprocessor Access register is set.

The Intel® XScale™ microarchitecture implements two instructions (MAR and MRA) that move two registers to acc0, and move acc0 to two registers, respectively.

Table 2-5. Internal Accumulator Access Format

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

Bits	Description	Notes
31:28	**cond** - ARM condition codes	-
20	**L** - move to/from internal accumulator 0= move to internal accumulator (MAR) 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. 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	Intel® XScale™ microarchitecture 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 the ARM 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:   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 executed only 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:      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 RdHi register.

The instruction is executed only 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™ microarchitecture extends the page attributes defined by the C- and B-bits in the page descriptors with an additional X-bit, which 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.

The Intel® XScale™ microarchitecture retains ARM definitions of the C and B encoding when X=0, which is different than the first-generation Intel® 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 writes all Store operations to memory, whether they are cacheable or not cacheable. This feature is useful for maintaining data-cache coherency.

The Intel® XScale™ microarchitecture also added a P-bit in the first-level descriptors to identify which pages of memory are protected with ECC. A descriptor with the P-bit set indicates the corresponding page in memory is ECC-protected. If the BCU ECC mode is enabled, then writes to such a page will be accompanied with an ECC, and reads will be validated by an ECC.

Bit 1 in the Control register (Coprocessor 15, register 1, opcode=1) enables ECC protection for memory accesses made during page-table walks.

These attributes are programmed in the translation table descriptors, which are highlighted in Table 2-8 “First-Level Descriptors,” Table 2-9, “Second-Level Descriptors for Coarse Page Table,” and Table 2-10 “Second-Level Descriptors for Fine Page Table,” all on page 14. Two second-level descriptor formats have been defined for Intel® XScale™ microarchitecture: one is used for the coarse-page table, and the other is used for the fine-page table.

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

<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="22">Tiny Page Base Address</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>

The P-bit controls ECC.

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

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 value of 0.

The X-bit, when set, modifies the meaning of the C- and B-bits.

2.3.3 Additions to CP15 Functionality

To accommodate the functionality in Intel® XScale™ microarchitecture, registers in CP15 and CP14 have been added or augmented.

At times, it is necessary 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 each Intel® StrongARM processor. For the Intel® XScale™ microarchitecture, the sequence—called CPWAIT — is shown in Example 2-1 on page 2-23.

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 register
        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, which is achieved by executing CPWAIT only after the 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 occur before CPWAIT completes or is issued. Programmers should ensure 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™ microarchitecture may generate, and the attributes of each. Subsequent sections provide details on each exception.

Table 2-11. Exception Summary

Exception Description	Exception Type[a]	Precise?	Updates FAR?
Reset	Reset	N	N
FIQ	FIQ	N	N
IRQ	IRQ	N	N
External Instruction	Prefetch	Y	N
Instruction MMU	Prefetch	Y	N
Instruction Cache Parity	Prefetch	Y	N
Lock Abort	Data	Y	N
MMU Data	Data	Y	Y
External Data	Data	N	N
Data Cache Parity	Data	N	N
Software Interrupt	Software Interrupt	Y	N
Undefined Instruction	Undefined Instruction	Y	N
Debug events	varies	varies	N

[a] Exception types are those described in the ARM, section 2.5.

2.3.4.2 Event Priority

The Intel® XScale™ microarchitecture follows the exception priority specified in the *ARM Architecture Reference Manual*. The processor has additional exceptions that might be generated while debugging.

Table 2-12. Event Priority

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

2.3.4.3 Prefetch Aborts

The Intel® XScale™ microarchitecture 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.

When a Prefetch abort occurs, hardware reports the highest priority one in the Extended Status field of the Fault Status register. The value placed in R14_ABORT (the Link register in Abort mode) is the address of the aborted instruction + 4.

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

Priority	Sources	FS[10,3:0][a]	Domain	FAR
Highest	**Instruction MMU Exception** Several exceptions can generate this encoding: - translation faults - domain faults, and - permission faults It is up to software to figure out which one occurred.	0b10000	invalid	invalid
	External Instruction Error Exception This exception occurs when the external memory system reports an error on an instruction cache fetch.	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 Intel® XScale™ microarchitecture: *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 Data 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™ microarchitecture, precise Data aborts are recoverable; imprecise Data aborts are not.

- Precise Data Aborts
  - — A Lock abort is a precise Data abort; the extended Status field of the Fault Status register 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 a Translation fault, Access-Permission fault, or External-Bus fault.

    The Fault Address register is undefined, and R14_ABORT is the address of the aborted instruction + 8.
  - — A Data MMU abort is precise 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™ microarchitecture Encoding of Fault Status for Data Aborts”

    The Fault Address register is set to the effective instruction data address, and R14_ABORT is the address of the aborted instruction + 8.

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

Priority	Sources		FS[10,3:0][a]	Domain	FAR
Highest	**Alignment**		0b000x1	invalid	valid
	External Abort on Translation	First level Second level	0b01100 0b01110	invalid valid	valid valid
	Translation	Section Page	0b00101 0b00111	invalid valid	valid valid
	Domain	Section Page	0b01001 0b01011	valid valid	valid valid
	Permission	Section Page	0b01101 0b01111	valid valid	valid valid
	Lock Abort 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 those generated on 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.

The Intel® XScale™ microarchitecture generates external Data aborts on multibit ECC errors, and when the Abort pin is asserted on memory transactions. An external Data abort can occur on non-cacheable loads, Reads into the cache, cache evictions, or Stores to external memory.

Although Intel® XScale™ microarchitecture guarantees the *Base Restored Abort Model* for precise Data aborts, it cannot do so in the case of imprecise Data aborts. A Data-Abort handler may encounter an updated Base register if it is invoked because of an imprecise Data 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 targeted registers. Because these faults are imprecise, it is possible the corrupted data is used before the Data Abort Fault handler is invoked. Therefore, software should treat imprecise Data aborts as unrecoverable.

Note: Even memory accesses marked as “stall until complete” can result in imprecise Data aborts. For these types of accesses, the fault is somewhat less imprecise than general case: it is guaranteed to be raised within three instructions of instruction that caused it. In other words, if a “stall-until-complete” LD or ST instruction triggers an imprecise fault, the program sees that fault within three instructions.

With this knowledge, it is possible to write code that accesses “stall-until-complete” memory with exemption. Place several NOP instructions after such an access. If an imprecise fault occurs, it does so during NOPs; the Data-Abort handler sees identical register and memory states as it would with a precise exception, and so should recover. An example is shown in Example 2-2 on page 2-27.

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 is reported. If reported Data abort is precise, software can correct cause of abort and re-execute the aborted instruction. If the lower priority Data abort still exists, it will be reported. Software can handle each Data abort separately until the instruction successfully executes.

  If the reported Data abort is imprecise, software needs to check the SPSR to determine if the previous context was executing in Abort mode. If such 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, the PLD has no effect.

This feature allows software to issue PLDs speculatively. For example, Example 2-3 places a PLD instruction early in the loop. This PLD fetches 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
```

2.4 Performance Considerations

The following subsections describe relevant performance considerations that compiler writers, application programmers, and system designers need to be aware of to efficiently use Intel® XScale™ microarchitecture. Performance numbers discussed here include *Interrupt latency, Branch prediction*, and *Instruction latencies*.

2.4.1 Interrupt Latency

Table 2-15 shows the minimum Interrupt latency for Intel® XScale™ microarchitecture, which is 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.

Table 2-15. Minimum Interrupt Latency

# MCLK Clock Cycles	Description
3	Minimum Interrupt Latency. This is measured from the assertion of IRQ or FIQ interrupt pin to the execution of the first instruction of the interrupt event handler.

Note: This number assumes that the Interrupt vector resides in the Instruction cache. Program control can lock the vector, translation information, and the Interrupt service routine into the Instruction cache.

Many parameters can affect this best-case performance:

- Instruction currently executing: could be as bad as a 16-register LDM
- Fault status: processor could fault just when the interrupt arrives
- Stalls: processor could be waiting for data from a load, doing a page table walk, etc.
- Bus ratio: the best case assumes a 3:1 core:bus ratio. Higher ratios would slightly improve performance

2.4.2 Branch Prediction

The Intel® XScale™ microarchitecture 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. 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.

These instructions (ARM B, ARM BL and Thumb B) 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™ microarchitecture dynamically predicts the outcome of these instructions based on previous outcomes. Table 2-16 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™ microarchitecture can execute the next instruction in the program flow in the cycle following the Branch.

Table 2-16. Branch Latency Penalty

Core Clock Cycles		Description
ARM*	Thumb	
+0	+ 0	Predicted Correctly. The instruction is in the Branch target cache and is correctly predicted.
+4	+ 5	Mispredicted. There are three occurrences of Branch misprediction, all of which incur a 4-cycle Branch delay penalty. 1. The instruction is in the Branch target buffer and is predicted Not Taken, but is actually Taken. 2. The instruction is in the Branch target buffer and is predicted Taken, but is actually Not Taken 3. The instruction is not in the Branch target buffer and is a Taken Branch.

2.4.3 Addressing Modes

The Load-and-Store Addressing mode implemented in Intel® XScale™ microarchitecture does add to the instruction latencies numbers.

2.4.4 Instruction Latencies

The latencies for all the instructions are shown in the following subsections regarding their functional groups:

- Branch
- Data processing, Multiply
- Status register access
- Load/Store, Semaphore
- Coprocessor

The following subsection explains how to read these tables.

2.4.4.1 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 2-16, which is four cycles.

- 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 2-4. Computing Latencies

```
UMLALr6,r8,r0,r1
ADD r9,r10,r11
SUB r2,r8,r9
MOV r0,r1
```

Table 2-17 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 2-17, UMLAL starts to issue at cycle 0, and the SUB issues at cycle 5. Thus, the Result latency is five.

Table 2-17. 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

2.4.4.2 Branch Instruction Timings

Table 2-18. Branch Instruction Timings

Mnemonic	Minimum Issue Latency with Correct Branch Predictions	Minimum Issue Latency with Branch Misprediction
B	1	5
BL	1	5
BLX	N/A	5
BX	N/A	5
MOV PC,<>	N/A	5
LDR PC,<>	N/A	8
LDM SP,{PC}	N/A	Specifying PC in register list adds five cycles to latency of LDM.

2.4.4.3 Data Processing Instruction Timings

Table 2-19. 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.

2.4.5 Multiply Instruction Timings

Table 2-20. 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 or Rs[31:15] = 0x1FFFF	0	1	2	1
		1	2	2	2
	Rs[31:27] = 0x00 or 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 or Rs[31:15] = 0x1FFFF	0	1	2	1
		1	2	2	2
	Rs[31:27] = 0x00 or 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 or Rs[31:15] = 0x1FFFF	0	2	RdLo = 2; RdHi = 3	2
		1	3	3	3
	Rs[31:27] = 0x00 or 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 or Rs[31:15] = 0x1FFFF	0	1	RdLo = 2; RdHi = 3	2
		1	3	3	3
	Rs[31:27] = 0x00 or 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 2-20. 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 2-21. 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 or Rs[31:16] = 0xFFFF	1	1	1
	Rs[31:28] = 0x0 or 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 2-22. 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.

2.4.5.1 Saturated Arithmetic Instructions

Table 2-23. Saturated Data Processing Instruction Timings

Mnemonic	Minimum Issue Latency	Minimum Result Latency
QADD	1	2
QSUB	1	2
QDADD	1	2
QDSUB	1	2

2.4.5.2 Status Register Access Instructions

Table 2-24. Status Register Access Instruction Timings

Mnemonic	Minimum Issue Latency	Minimum Result Latency
MRS	1	2
MSR	2 (6 if updating mode bits)	1

2.4.5.3 Load/Store Instructions

Table 2-25. 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 2-26. 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.

2.4.5.4 Semaphore Instructions

Table 2-27. Semaphore Instruction Timings

Mnemonic	Minimum Issue Latency	Minimum Result Latency
SWP	5	5
SWPB	5	5

2.4.5.5 Coprocessor Instructions

Table 2-28. CP15 Register Access Instruction Timings

Mnemonic	Minimum Issue Latency	Minimum Result Latency
MRC	4	4
MCR	2	N/A

Table 2-29. 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

2.4.5.6 Miscellaneous Instruction Timing

Table 2-30. SWI Instruction Timings

Mnemonic	Minimum latency to first instruction of SWI exception handler
SWI	6

Table 2-31. Count Leading Zeros Instruction Timings

Mnemonic	Minimum Issue Latency	Minimum Result Latency
CLZ	1	1

2.4.5.7 Thumb Instructions

The timing of Thumb instructions is 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.

Optimization Techniques 3

3.1 The StrongARM* Pipeline

One of the biggest differences between the Intel® XScale™ core (ARM* architecture compliant) and first-generation Intel® StrongARM* processors is the pipeline. Many of the differences are summarized in Figure 3-1. This section provides a brief description of the structure and behavior of the Intel® XScale™ microarchitecture pipeline.

3.1.1 General Pipeline Characteristics

While the Intel® XScale™ microarchitecture 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. For additional information, refer to the *StrongARM Architecture Reference Manual*.

3.1.1.1 Number of Pipeline Stages

The Intel® XScale™ microarchitecture has a longer pipeline (seven stages versus five stages) that operates at a much higher frequency than its predecessors for greater overall performance. However, the longer Intel® XScale™ microarchitecture pipeline does offer some potential tradeoffs.

- Larger Branch misprediction penalty (four cycles in the Intel® XScale™ microarchitecture instead of one in the Intel® StrongARM* architecture) 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 delay cycles on the Intel® XScale™ core compared to the first-generation Intel® StrongARM* processors (**LDM**, **STM**).
- Decode and register-file lookups are spread out over two cycles in the Intel® XScale™ microarchitecture instead of one cycle.

3.1.1.2 Intel® XScale™ Microarchitecture Pipeline Organization

The Intel® XScale™ microarchitecture single-issue superpipeline consists of a main execution pipeline, MAC pipeline, and a memory-access pipeline, as shown in Figure 3-1, with the main execution pipeline shaded.

Figure 3-1. Intel® XScale™ Microarchitecture RISC Superpipeline

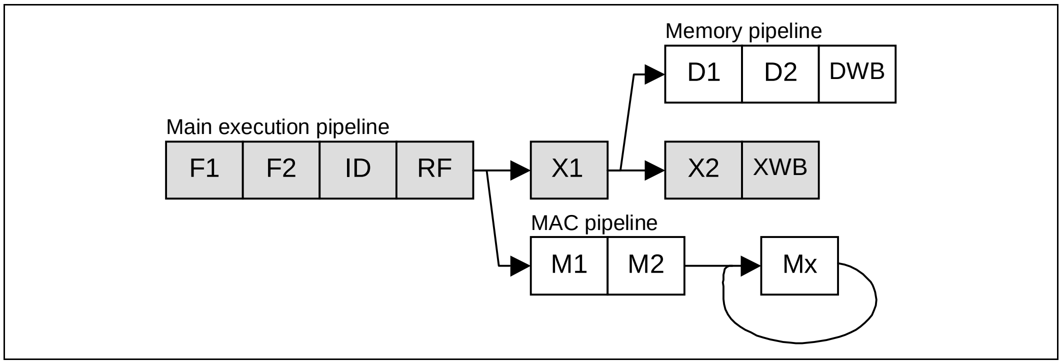

Table 3-1. Pipelines and Pipe Stages

Pipe / Pipe Stage	Description	Covered In
Main Execution Pipeline	Handles data processing instructions	Section 3.1.3
IF1/IF2	Instruction Fetch	Section 3.1.3
ID	Instruction Decode	Section 3.1.3
RF	Register File / Operand Shifter	Section 3.1.3
X1	ALU Execute	Section 3.1.3
X2	State Execute	Section 3.1.3
XWB	Write-back	Section 3.1.3
Memory Pipeline	Handles load/store instructions	Section 3.1.4
D1/D2	Data Cache Access	Section 3.1.4
DWB	Data cache writeback	Section 3.1.4
MAC Pipeline	Handles all multiply instructions	Section 3.1.5
M1-M5	Multiplier stages	Section 3.1.5
MWB (not shown)	MAC write-back - may occur during M2-M5	Section 3.1.5

3.1.1.3 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™ microarchitecture 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, causing instructions to 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 completed and the result has been written back to the register file.)

3.1.1.4 Register Scoreboarding

In certain situations, register dependencies between instructions may cause the pipeline to stall. 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. Only the destination of MAC operations and memory loads are scoreboarded; destinations of ALU instructions are not.

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 can complete independently.

3.1.1.5 Use of Bypassing

To minimize data hazards, the Intel® XScale™ microarchitecture pipeline makes extensive use of bypassing, which allows results-forwarding from multiple sources, eliminating pipeline stalls.

3.1.2 Instruction Flow Through the Pipeline

The Intel® XScale™ microarchitecture pipeline issues a single instruction per clock cycle. Instruction execution begins at the F1 pipestage and completes at the WB 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, each instruction can complete processing 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.

3.1.2.1 ARM* v5 Instruction Execution

Figure 3-2 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), causing M1 or X1 to be idle.

All Load/Store instructions are routed to the Memory pipeline after the effective addresses have been calculated in X1.

The ARM v5 bx (Branch and Exchange) instruction, which is used to branch between ARM and THUMB code, flushes the entire pipeline (the *bx* instruction is not dynamically predicted by the BTB). If the processor is in Thumb mode, the ID pipestage dynamically expands each THUMB instruction into a normal ARM v5 RISC instruction, and execution resumes as usual.

3.1.2.2 Pipeline Stalls

The progress of an instruction can stall anywhere in the pipeline, and several pipestages may stall for various reasons. It is important to understand when and how hazards occur in the Intel® XScale™ microarchitecture pipeline as performance degradation can be significant if pipeline stalls are not minimized.

3.1.3 Main Execution Pipeline

3.1.3.1 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 Instruction Decode (ID) stage. Several important functional units reside within the F1 and F2 stages, including the Branch target buffer (BTB) and the Instruction Fetch unit (IFU).

An understanding of the BTB and IFU are important for performance considerations. A summary of operation is provided here so that readers can understand their role in the F1 pipestage.

- 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.
- IFU: The IFU is responsible for delivering instructions to the ID pipestage. An instruction could come from one of two sources: instruction cache or Fill buffers. One instruction word is delivered each cycle (if possible) to the ID.

3.1.3.2 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 accepts a new Instruction word from the IFU on every clock cycle in which there is no stall. The ID pipestage is responsible for the following:

- General instruction decoding (extracting opcode, operand addresses, destination addresses, and offset)
- Detecting undefined instructions and generating an exception
- Dynamic expansion of complex instructions into a sequence of simple instructions (complex instructions are defined as those that require more than one clock cycle to issue, such as LDM, STM, and SWP)

3.1.3.3 RF (Register File / Shifter) Pipestage

The main function of the RF pipestage is to read and write to the Register File unit (RFU), which provides source data to the following*:*

- 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 on 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 forward it to the next pipestage. When a register dependency does exist, the RFU will keep track of which register is unavailable. When the result is returned, the RFU will stop the pipe stall.

The ARM architecture specifies one of the data-processing instruction operands as 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.

3.1.3.4 X1 (Execute) Pipestages

The X1 pipestage performs the following functions:

- ALU calculations - the ALU performs arithmetic and logic operations, as required for data processing instructions and Load/Store index calculations.
- Determines 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.

3.1.3.5 X2 (Execute 2) Pipestage

The X2 pipestage contains the program status registers (PSRs). This pipestage selects what is going to be written to the RFU in the WB cycle: PSRs (MRS instruction), ALU output, or other items.

3.1.3.6 WB (Write back)

When an instruction reaches the Write-back stage, it is considered complete. Changes are written to the RFU.

3.1.4 Memory Pipeline

The Memory pipeline consists of two stages, D1 and D2. The Data cache unit (DCU) consists of the Data-cache array, Mini-Data cache, Fill buffers, and Write buffers. The Memory pipeline handles Load/Store instructions.

3.1.4.1 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 return 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.

3.1.5 Multiply/Multiply Accumulate (MAC) Pipeline

The Multiply-Accumulate (MAC) unit executes the Multiply and Multiply-Accumulate instructions supported by the Intel® XScale™ core. The MAC implements the 40-bit Intel® XScale™ microarchitecture Accumulator register, acc0, and handles the instructions that 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, which 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.

3.1.5.1 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 depends on the operand size) and returned to the register file.

An instruction that occupies the M1 or M2 pipestages will also occupy the X1 and X2 pipestages, respectively. Every cycle, a MAC operation progresses for M1 to M5, and a MAC operation may complete anywhere from M2-M5. If a MAC operation enters M3-M5, it is considered committed because it will modify the architectural state regardless of subsequent events.

3.2 Basic Optimization

This section outlines optimizations specific to the ARM architecture that have been modified for the Intel® XScale™ microarchitecture.

3.2.1 Conditional Instructions

The Intel® XScale™ microarchitecture executes instructions conditionally. This feature, combined with the ability of the Intel® XScale™ microarchitecture instructions to modify the condition codes, makes possible a wide array of optimizations.

3.2.1.1 Optimizing Condition Checks

The Intel® XScale™ microarchitecture instructions can selectively modify the state of the condition codes. When generating code for *if-else* and *loop* conditions, it helps to use this feature to set condition codes, thereby eliminating the need for a subsequent Compare instruction.

Consider the C code segment below.

```
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 by making use of the *add* instruction to set Condition codes:

```
;Assume r0 contains the value a, and r1 contains the value b
    adds  r0,r0,r1
```

The instructions that increment or decrement the loop counter can also modify the Condition codes, which 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 appear as

```
L6:
.
.
    subs r3, r3, #1
    bne  L6
```

It also helps 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 can be rewritten as follows, the code generated eliminates the Compare instruction to check for the Loop Exit condition (not all loops can be written this way).

```
for (i = 9; i >= 0; i--)
{
    do something;
}
```

3.2.1.2 Optimizing Branches

Branches decrease application performance by indirectly causing pipeline stalls. Branch prediction improves the performance by decreasing 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. Because the total number of branches executed in a program is relatively large compared to the size of the Branch target buffer; it is often helps 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 requires three cycles to execute the *else* portion and four cycles for the *if*, 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 percent of the time, and if we assume that both the *if* portion and the *else* portion are equally likely to be taken, on an average, the code above requires 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 require three cycles to execute, assuming best-case conditions. As can be seen, using conditional instructions significantly speeds up execution. However, the use of Conditional instructions should be carefully considered to ensure that performance does improve. To determine 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 where we are better off using branches over Conditional instructions.

```
    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 requires one cycle to execute, the *if* portion requires seven cycles to execute, and the *else* portion requires six cycles to execute. If we were to change the code above to eliminate the Branch instructions by making use of Conditional instructions, the *if-else* portion would always require 10 cycles to complete.

If we assume that both paths are equally likely to be taken, and that branches are mispredicted 50 percent of the time, the cost of using conditional execution versus using branches can be computed as follows:

$$1 + \left(\frac{50}{100} \times 10\right) + \left(\frac{50}{100} \times 10\right) = 11 \qquad cycles$$

Cost of using branches:

$$1 + \left(\frac{50}{100} \times 7\right) + \left(\frac{50}{100} \times 6\right) + \left(\frac{50}{100} \times 4\right) = 9.5 \qquad cycles$$

We get better performance by using branch instructions in the above example.

3.2.1.3 Optimizing Complex Expressions

Conditional instructions also should 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 manner improves performance by minimizing the number of branches, thereby minimizing the penalties caused by Branch mispredictions. This approach also reduces the use of Branch prediction resources.

3.2.2 Bit-Field Manipulation

The Intel® XScale™ microarchitecture 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
```

3.2.3 Optimizing the Use of Immediate (Constant) Values

Use the Intel® XScale™ core MOV or MVN instruction when loading an immediate (constant) value into a register. 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 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 no more than four instructions.

3.2.4 Optimizing Integer Multiply and Divide

Optimize multiplication by an integer constant 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
```

Note: Only use the above optimization 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
```

In the above example, the Add instruction would stall for one cycle. To prevent such a stall, fill in another instruction before Add.

3.2.5 Effective Use of Addressing Modes

The Intel® XScale™ microarchitecture provides a variety of addressing modes that make indexing an array of objects highly efficient. For a detailed description of these addressing modes, refer to the *ARM Architecture Reference Manual*. The following code samples illustrate how to optimize various kinds of array operations 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]!
```

3.3 Cache and Prefetch Optimizations

This section 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.

3.3.1 Instruction Cache

The Intel® XScale™ microarchitecture has separate Instruction and Data caches. Some data may reside 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. Using the I-cache provides no performance benefit. The one exception is that code, which locks code into the Instruction cache, must itself execute from non-cached memory.

3.3.1.1 Cache Miss Cost

The Intel® XScale™ microarchitecture performance depends highly on reducing the cache-miss rate. When an Instruction cache miss occurs, the timing to retrieve the next instruction is the same as that for retrieving data for the Data cache. Using the same assumptions as those for Data caches, the result is 60 to 90 core cycles to retrieve the first instruction. Once the first 4-byte word is read, it takes another six core cycles to read in the next two instructions, or a total of 78 to 108 clocks to fill a cache line. If the new instructions each execute in one core cycle, the processor stalls for four cycles, waiting for the next pair of instructions. Further, if the next pair of instructions each execute in one cycle each, the processor again stalls for four more cycles. From this it is clear that executing non-cached instructions severely curtails processor performance. It is very important to do everything possible to minimize cache misses.

3.3.1.2 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 the difficulty of predicting when (and over which cache lines) evictions occur. This information must be obtained by experimentation using performance profiling.

3.3.1.3 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 blocks modula 1024 of cache space. Any overused sets will thrash the cache. The ideal situation is for the software tools to distribute the code on a temporal evenness over this space.

Such a task 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.

3.3.1.4 Locking Code into the Instruction Cache

One very important instruction cache feature is the ability to lock code into the Instruction cache. There are two reasons for locking critical code. First, once locked into the Instruction cache, the code is always available for fast execution. Second, 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 include the following:

- 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 ensures even distribution across all cache ways. Though it is possible to choose randomly located functions for cache locking, this approach runs the risk of landing multiple cache ways in one set and few or none in another set.

3.3.2 Data and Mini-Cache

The Intel® XScale™ microarchitecture lets you define memory regions whose cache policies you can set. Supported policies and configurations include the following:

- 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

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.

If the application is running under an OS, then the OS may restrict you from using certain cache policies.

3.3.2.1 Non-Cacheable Regions

Non-cached memory (X=0, C=0, and B=0) should be used only if necessary as is often true for I/O devices. Accessing non-cacheable memory is likely to cause frequent processor stalls due to the long latency of memory Reads.

3.3.2.2 Write-through and Write-back Cached Memory Regions

Write-through memory regions generate more data traffic on the bus. Therefore, it is recommended that the Write-back policy must be used whenever possible.

However, in a multiprocessor environment, it will be necessary to use a Write-through policy if data is shared across multiple processors. In such a situation, all shared memory regions should use the Write-through policy. Memory regions that are private to a particular processor should use the Write-back policy.

3.3.2.3 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, which causes the cache line to be loaded on first reference. 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.

3.3.2.4 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. Improved application performance can occur by converting a part of the cache into on-chip RAM and allocating frequently allocated variables to it. Due to the Intel® XScale™ microarchitecture round-robin replacement policy, all data will eventually be evicted unless locked. 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. If the data in the on-chip RAM is to be initialized to zero, the locking process can be accelerated by using the CP15 “prefetch zero” function. This function does not generate external memory references.

When creating the on-chip RAM, ensure that all sets in the on-chip RAM area of the Data cache have approximately the same number of ways locked; otherwise, some sets will have more ways locked than the others. This uneven allocation will increase the thrashing level in some sets and leave other sets underutilized.

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 set2, as a result three ways in both sets set0 and set1, will be locked, leaving 28 ways for use by other variables.

This can be overcome by allocating on-chip RAM data in sequential order. In the above example, allocating arr2 to address 64, and arr3 to address 128 allows the three arrays to use only one way in sets 0 through 8.

3.3.2.5 Mini-Data Cache

The Mini-Data cache is best used for data structures having 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, which will reduce performance. Placing this data in Mini-Data cache memory region instead 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 its use does not trash the main Data cache. This allocation would keep 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 can occupy the whole cache.

Overuse of the Mini-Data cache will thrash the cache. This is easy to do because the Mini-Data cache has just two ways per set. For example, a loop that 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 array is aligned on a 1-K boundary will quickly thrash the cache.

3.3.2.6 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, 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, 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.

Additional Prefetch considerations are discussed in greater detail in following sections.

3.3.2.7 Literal Pools

The Intel® XScale™ microarchitecture does not have a single instruction that can move all Literals (a constant or address) to a register. One technique for loading registers with Literals in the Intel® XScale™ microarchitecture 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*. Refer to the *ARM Architecture Reference Manual*, Section B, for details on Literal pools. It is advantageous to place all the Literals together in 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 wasted 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.

3.3.3 Cache Considerations

3.3.3.1 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, refer to Section 3.3.5.2, “Prefetch Loop Scheduling”.

3.3.4 Memory Page Thrashing

Memory-page thrashing occurs because of the nature of SDRAM. SDRAMs are typically divided into four banks, where 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 two to three bus clocks. Thrashing occurs when subsequent memory accesses within the same memory bank access different pages. The memory page change adds three to four 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. Instruction and data sections must reside in different memory banks, or they will continually thrash the memory page selection.

3.3.5 Prefetch Considerations

The Intel® XScale™ microarchitecture has a true Prefetch Load instruction (PLD). This instruction preloads data into the Data and Mini-Data caches. Data prefetching hides 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 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™ microarchitecture 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 that 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, thus increasing register pressure. By contrast, the Intel Intel® XScale™ core prefetch reduces register pressure instead of increasing it.

The Intel® XScale™ core 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, the processor will ignore the Prefetch instruction and continue processing the next instruction. This operation is particularly advantageous in the case where a Null pointer terminates a linked list or recursive data structure. A prefetch of the Null pointer will not fault program flow.

3.3.5.1 Prefetch Distances in the Intel® XScale™ Core

Scheduling the Prefetch instruction requires an understanding of system latency times and system resources that affect when to use the Prefetch instruction. This section considers three timing elements.

1. N_{cwf} critical word first
2. N_{clxfer} full cache line transfer time
3. $N_{subissue}$ subsequent Prefetch issue time to insure uninterrupted transfers

The Memory Latency times presented here assume typical SDRAM that is currently available and working with the Intel® XScale™ microarchitecture. It is assumed that the SDRAM supports “critical word first” transfers (when a cache line is being transferred, the first word transferred corresponds to the one the processor needs immediately, as opposed to transferring the data from lowest address first).

The cycle times assume that the core is running six times a fast as the Memory-Transfer bus. Further, the example values presented here apply to the current processor implementation. The Intel® XScale™ microarchitecture is different for future implementations.

N_{cwf} is the number of core cycles required to transfer the first critical word of a Prefetch or Load operation:

$$N_{cwf} = N_{lookup} + N_{cwfxfer}$$

Where:

N_{lookup} The number of core clocks required for the processor to issue a Memory-Transfer request to the SDRAM plus the time the SDRAM requires to locate the data.

$$N_{lookup} = N_{processor} + N_{memwait} + N_{mempagewait}$$

The Intel® XScale™ microarchitecture needs seven bus clocks to process a memory request to the SDRAM ($N_{processor}$). Typical SDRAM needs two to three bus clocks to select the memory locations, provided the current SDRAM memory page is selected ($N_{memwait}$). If the current SDRAM memory page is not selected, then an additional three to four bus cycles are needed to look up the memory data locations ($N_{mempagewait}$). Thus, the look-up time can range from nine to 14 bus clock cycles. Translating this to core cycles at a ratio of six to one means between 54 and 84 core clocks.

$N_{cwfxfer}$ This is the number of core clocks needed to transfer the first critical word of a Cache line-fill operation. It takes one bus clock to transfer the first word if the data is in the lower word address of the transfer, and one additional core clock if the word is in the upper word address range of the transfer. Thus, for the examples presented here, this translates to six or seven core clock cycles.

N_{cwf} For the Intel® XScale™ microarchitecture, this works out to be 60 instructions, assuming two wait- state SDRAM and that the current SDRAM memory page is selected. The second 64 bits of data will be available at the next bus cycle or six core clocks.

N_{clxfer} The minimal number of cycles to prefetch ahead for an entire cache line:

$$N_{clxfer} = N_{lookup} + N_{linexfer}$$

Where:

$N_{linexfer}$ The number of core clocks required to transfer one complete cache line. The Intel® XScale™ microarchitecture requires four bus cycles to transfer four 64-bit words of a full cache line. Given the six-to-one core-to-bus clock ratio, this translates to 24 core clock cycles.

N_{clxfer} Works out to be about 78 cycles for the Intel® XScale™ microarchitecture when using a Two-Bus-Cycle Wait state.

$N_{subissue}$ This is the maximum number of core clocks that a subsequent bus-transfer request must be made to guarantee that transfer occurs immediately after the previous request has completed its transfer. If a transfer is not made within this time, idle bus cycles will occur, thereby reducing efficiency. If the previous transfer was for a full Cache-Line Read or Write, it would require 24 core cycles at a six-to-one ratio between core and bus clocks. If the previous operation was for a Half Cache line, it would require 12 core clocks.

Consider the following code sample:

```
    add   r1, r1, #1
; Sequence of instructions that use r2.
    ldr   r2, [r3]
    add   r3, r3, #4
    mov   r4, r3
    sub   r2, r2, #1
```

The *sub* instruction above would stall if the data being loaded misses the cache. These stalls can be avoided by using a *pld* instruction well ahead of the *sub* instruction, as shown below. The number of instructions required to ensure a stall does not occur is proportional to N_{cwf} for a given system.

```
    pld   [r3]
    add   r1, r1, #1
; Sequence of instructions that use r2. These instructions leave r3 unchanged.
    ldr   r2, [r3]
    add   r3, r3, #4
    mov   r4, r3
    sub   r2, r2, #1
```

3.3.5.2 Prefetch Loop Scheduling

When adding prefetch to a loop that operates on arrays, it's a good idea 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 arrangement makes it easy to predict where to fetch the data. The number of iterations to prefetch ahead is refered to as the *prefetch scheduling distance*, or PSD. For the Intel® XScale™ microarchitecture, this PSD can be calculated as

$$PSD = floor\left(\frac{(N_{lookup} + N_{linexfer} \times N_{pref} + N_{hwlinexfer} \times N_{evict})}{(CPI \times N_{inst})}\right)$$

Where:

N_{pref} The number of cache lines to be prefetched for both reading and writing

N_{evict} The number of first-time half-line evictions mentioned

N_{inst} The number of instructions executed in one iteration of the loop

$N_{hwlinexfer}$ The number of core clocks required to write half a cache line as if only one of the *cache line dirty* bits were set when a line eviction occurred. For the Intel® XScale™ microarchitecture, this process requires two bus clocks or 12 core clocks.

CPI The average number of core clocks per instruction

The *PSD* number in the above equation provides is a good starting point, but may not be the most ideal consideration. Estimating N_{evict} is very difficult from static code; however, if the operational data uses the Mini-Data cache, and if the loop operations should overflow the Mini-Data cache, a first-order estimate of N_{evict} would be the number of bytes written pre-loop iteration divided by a half cache line size of 16 bytes. Cache overflow can be estimated by the number of cache lines transferred each iteration and the number of expected loop iterations. N_{evict} and CPI can be estimated by profiling the code using the performance monitor "Cache Write-back" Event count.

3.3.5.3 Prefetch Loop Limitations

It is not always advantageous to add Prefetch to a loop. Loop characteristics that limit the use value of Prefetch are discussed below.

3.3.5.4 Compute vs. Data Bus Bound

At the extreme, a Data Bus Bound loop will not benefit from Prefetch because all the system resources for transferring data are quickly allocated, and there are no instructions that can be profitably executed. On the other end of the scale, Compute Bound loops completely hide all Data Transfer latencies.

3.3.5.5 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 using scheduled Prefetch instructions.

3.3.5.6 Bandwidth Limitations

Prefetch overuse can usurp resources and degrade performance because once bus-traffic requests exceed the system resource capacity, the processor stalls. The Intel® XScale™ microarchitecture data-transfer resources are the following:

- 4 Fill buffers
- 4 Pending buffers
- 8 Half-Cache Line Write buffer

SDRAM resources are typically:

- 4 Memory banks
- 1 Page buffer per bank referencing a 4K address range
- 4 Transfer-Request buffers

Consider how these resources all 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, and a Write buffer is required for each memory Write to non-cached memory that is non-coalescing. Consequently, an 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). 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, ensure that the combination of Prefetch and Instruction bus requests does not exceed the system resource capacity described above or performance will degrade instead of improve. It's important to spread Prefetch operations over calculations so as to allow bus traffic to flow free, and to minimize the number of necessary prefetches.

3.3.5.7 Cache Memory Considerations

Stride—the way data structures are walked through—can affect temporal data quality and reduce or increase cache conflicts. The Intel® XScale™ microarchitecture Data cache and Mini-Data caches each have 32 sets of 32 bytes, which means that each cache line in a set is on a modular 1K-address boundary. Be careful to choose data structure sizes and stride requirements that do not overwhelm a given set that could cause conflicts and increased register pressure. Register pressure can be increased because additional registers are needed to track Prefetch addresses. These effects can be minimized by rearranging data structure components to use more parallel access to search and compare elements. Similarly, rearranging often-written sections of data structures so that they fit in the same half cache line (16 bytes for the Intel® XScale™ microarchitecture) 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_array[NMAX];
int b_array[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 written-to-often 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, rarely change. If the fields are laid out as shown in the above example—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 one by rearranging the fields in the above data structure as shown below:

```
struct employee {
    struct employee *prev;
    struct employee *next;
    int ssno;
    int empid;
    float Year2DatePay;
    float Year2DateTax;
    float Year2Date401KDed;
    float Year2DateOtherDed;
};
```

3.3.5.8 Cache Blocking

Cache-blocking techniques, such as Strip mining, are used to improve temporal locality of data. Given a large data set that can be reused across multiple passes of a loop, data blocking divides the data into smaller chunks that can be loaded into the cache during the first loop, and then be available for processing on subsequence 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][j] 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];
                }
```

3.3.5.9 Prefetch Unrolling

When iterating through a loop, Data Transfer latency can be hidden by prefetching ahead one or more iterations. The solution incurs several unwanted side effects: the final interactions of a Loop load useless data into the cache, pollute the cache, increase bus traffic, and possibly evict valuable temporal data. This problem can be resolved by Prefetch unrolling. For example, consider the following:

```
for(i=0; i<NMAX; i++)
{
    prefetch(data[i+2]);
    sum += data[i];
}
```

Interactions i-1 and i, will prefetch superfluous data. Unrolling the end of the loop avoids this problem.

```
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.

3.3.5.10 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 loop iteration. Linked lists should be converted to arrays as often as possible.

```
while(p) {
    prefetch(p->next);
    do_something(p->data);
    p = p->next;
}
```

Recursive Data-Structure traversal, which is similar to Linked-List traveral, is another construct where prefetching can be applied. 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 loop iteration.

```
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 use. If a cache conflict causes data to be evicted from the cache, only the data from the first Prefetch is lost.

3.3.5.11 Loop Interchange

As mentioned earlier, the sequence in which data is accessed affects Cache thrashing. Usually, it is best to access data in a spatially contiguous address range. However, Data arrays may have been laid out such that indexed elements are not physically adjacent. Consider the following C code that 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 adjacent. This situation increases 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 then becomes

```
for(i=0; i<NMAX; i++)
    for(j=0; j<NMAX; j++)
    {
        prefetch(A[i][j+1]);
        sum += A[i][j];
    }
```

3.3.5.12 Loop Fusion

Loop fusion is a process of combining multiple loops, which reuse the same data, in to 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], c[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];
}
```

3.3.5.13 Prefetch to Reduce Register Pressure

Prefetch can reduce register pressure. When data is needed for an operation, 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 > 60 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 then 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.

3.4 Instruction Scheduling

This section discusses Instruction-Scheduling optimizations. Instruction scheduling refers to the rearrangement of a sequence of instructions for minimizing pipeline stalls, which improves application performance. While making this rearrangement, ensure that the rearranged sequence of instructions has the same effect as the original sequence of instructions.

3.4.1 Scheduling Loads

With the Intel® XScale™ core, an LDR instruction has a three-cycle Result latency, assuming the data being loaded resides in the Data cache. If the instruction after the LDR needs to use the Load result, then it would stall for two 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 two cycles because it uses the Load result. 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. 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 requires 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 the load result. Note that this may at times cause certain register values to be spilled over to memory due to the increase in register pressure. In such cases, use a Preload instruction or a *Preload hint* to ensure that the data access in the LDR instruction hits the cache when it executes. A Preload hint should be used in cases where there is a question as to whether the Load instruction would be executed. A *Preload instruction* should be used in cases where there is no question 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, ADD and LDR instructions can be moved before the MOV instruction. Note that this would prevent pipeline stalls if the load hits the Data cache. However, if 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 the contents of the register r6 can 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 register r6 contents have been spilled to the stack and subsequently loaded back to 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
```

Intel® XScale™ microarchitecture has four Fill buffers that fetch data from external memory when a Data-Cache miss occurs. The Intel® XScale™ microarchitecture stalls when all Fill buffers are in use. This happens when more than four loads are outstanding and are being fetched from memory. As a result, the code written should ensure that no more than four loads are outstanding at the same time. For example, the number of loads issued sequentially should not exceed four. 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 arrive at the number of loads that are 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 Write-back policy, 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 limited by this fact.

3.4.1.1 Scheduling Load and Store Double (LDRD/STRD)

The Intel® XScale™ microarchitecture 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. Two important restrictions exist on how these instructions can 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 LDRDs targeting R12 because it incurs an extra cycle of Issue latency.

The LDRD instruction has a Result latency of three or four cycles, depending on the Destination register being accessed (assuming the data being loaded is in the Data cache).

```
    add   r6, r7, r8
    sub   r5, r6, r9
; The following ldrd instruction 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 three cycles because of the four-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 one cycle.

```
; The str instruction below would stall for 1 cycle
    ldrd  r0, [r3]
    str   r4, [r5]
```

3.4.1.2 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 two 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 Load results. 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 addition 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 require 11 cycles to complete. Rewriting the code as shown below using LDRD instruction would need only seven 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 five cycles to complete.

```
    stm   r0, {r2, r3}
    add   r1, r1, #1
```

The alternative version (shown below) would only take three cycles to complete.

```
    strd  r2, [r0]
    add   r1, r1, #1
```

3.4.2 Scheduling Data Processing Instructions

Most Intel® XScale™ microarchitecture data-processing instructions have a one-cycle Result latency, which means that the current instruction can use the result from the previous data-processing instruction. However, the Result latency is two 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 one-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 one-cycle stall:

```
add   r1, r2, r3
sub   r6, r7, r8
mov   r4, r1, LSL #2
```

All data-processing instructions incur a two-cycle Issue penalty and a two-cycle Result penalty when the Shifter operand is a Shift/Rotate by a register or the Shifter operand is RRX. Because the next instruction would always incur a two-cycle Issue penalty, there is no way to avoid such a stall except by rewriting 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 one-cycle stall due to the Issue latency of the Add instruction as the Shifter operand is shifted 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
```

3.4.3 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 zero to three 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 one to three 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 entire pipeline. A four-cycle Multiply operation that sets the Condition codes behaves the same as a four-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 three cycles if the Multiply takes four 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
```

Refer to “Instruction Latencies” in the *ARM Architecture Reference Manual* to get the Instruction latencies for various Multiply instructions. The Multiply instructions should be scheduled taking into consideration these Instruction latencies.

3.4.4 Scheduling SWP and SWPB Instructions

The SWP and SWPB instructions have a five-cycle Issue latency. As a result of this latency, the instruction following the SWP/SWPB instruction would stall for four 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 two 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 requires nine cycles to complete. The rewritten code below needs only six cycles to execute:

```
; Swap the contents of memory locations pointed to by r0 and r1
    ldr   r2, [r0]
    ldr   r3, [r1]
    str   r2, [r1]
    str   r3, [r0]
```

3.4.5 Scheduling the MRA and MAR Instructions (MRRC/MCRR)

The MRA (MRRC) instruction has a one-cycle Issue latency, a two-to-three cycle Result latency (depending on the Destination register value being accessed), and a two-cycle Resource latency.

Consider the code sample:

```
mra   r6, r7, acc0
mra   r8, r9, acc0
add   r1, r1, #1
```

The code shown above would incur a one-cycle stall due to the two-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 two-cycle penalty due to the three-cycle Result latency for the second Destination register.

```
mra   r6, r7, acc0
mov   r1, r7
mov   r0, r6
add   r2, r2, #1
```

The stalls incurred by the code shown above 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 two cycles. Due to the two-cycle Issue latency, the pipeline would always stall for one cycle following a MAR instruction. Therefore, the MAR instruction should be used only where absolutely necessary.

3.4.6 Scheduling the MIA and MIAPH Instructions

The MIA instruction has an Issue latency of one cycle. The Result and Resource latency can vary from one to three 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 zero to two cycles, depending on the values in the registers r2 and r3 due to the one-to-three-cycle Resource latency.

Similarly, consider the following code sample:

```
mia   acc0, r2, r3
mra   r4, r5, acc0
```

The MRA instruction above can stall from zero to two cycles, depending on the values in the registers r2 and r3 due to the one-to-three-cycle Result latency.

The MIAPH instruction has an Issue latency of one cycle, Result latency of two cycles, and a Resource latency of two cycles.

Consider the code sample shown below:

```
add   r1, r2, r3
miaph acc0, r3, r4
miaph acc0, r5, r6
mra   r6, r7, acc0
sub   r8, r3, r4
```

The second MIAPH instruction would stall for one cycle due to a two-cycle Resource latency. The MRA instruction would stall for one cycle due to a two-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
```

3.4.7 Scheduling MRS and MSR Instructions

The MRS instruction has an Issue latency of one cycle and a Result latency of two cycles. The MSR instruction has an Issue latency of two cycles (six, if updating the *mode* bits), and a Result latency of one cycle.

Consider the code sample:

```
mrs   r0, cpsr
orr   r0, r0, #1
add   r1, r2, r3
```

The ORR instruction above would incur a one-cycle stall due to the two-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.

3.4.8 Scheduling CP15 Coprocessor Instructions

The MRC instruction has an Issue latency of one cycle and a Result latency of three cycles. The MCR instruction has an Issue latency of one 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 two-cycle latency due to the three-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
```

3.5 Optimizing C Libraries

Many of the standard C library routines can benefit greatly by being optimized for the Intel® XScale™ microarchitecture technology. The following string and memory-manipulation routines should be tuned to obtain the best performance from the Intel® XScale™ microarchitecture (Instruction selection, Cache usage, and Data prefetch):

strcat, strchr, strcmp, strcoll, strcpy, strcspn, strlen, strncat, strncmp, strpbrk, strrchr, strspn, strstr, strtok, strxfrm, memchr, memcmp, memcpy, memmove, memset

3.6 Optimizing for Size

For applications such as cell-phone software, the code must be optimized for improved performance while minimizing code size. Optimizing for smaller code size will, in general, lower the performance of your application. This section contains techniques for code-size optimization using the Intel® XScale™ microarchitecture instruction set.

3.6.1 Space/Performance Trade Off

Many optimizations mentioned in the previous sections improve ARM code; however, using these instructions will result in increased code size. Use the following optimizations to reduce the application code space requirements.

3.6.1.1 Multiple Word Load and Store

The LDM/STM instructions are one word long and simultaneously load or store multiple registers. Use the LDM/STM instructions instead of a sequence of Loads/Stores to consecutive addresses in memory whenever possible.

3.6.1.2 Use of Conditional Instructions

Using conditional instructions to expand If-Then-Else statements will result in increasing the size of the generated code. Therefore, do not use Conditional instructions if application code space requirements are an issue.

3.6.1.3 Use of PLD Instructions

The Preload instruction PLD is only a hint; it does not change the processor architectural state. Using or not using them will not change code behavior; therefore, avoid using these instructions when optimizing for space.

For OS Developers A

A.1 Introduction

A.1.1 Intended Audience

This document is a compendium of information of specific interest to operating systems or firmware developers. It is not intended for use by application developers.

Readers should be familiar with the definition of the ARM* V5TE architecture, with the *Intel® XScale™ Microarchitecture Developers Manual* (ARM* architecture compliant), and with the *Intel® XScale™ Microarchitecture Programmers Reference Manual*. Some familiarity with the Intel® StrongARM* 11x0 processors is helpful, but not essential.

A.2 Document Organization

A.2.1 Related Documents

- ARM V5TE definition from ARM (*ARM* Architecture Reference Manual*)
- *Intel® XScale™ Microarchitecture Technical Summary*
- *Intel® XScale™ Microarchitecture Product Brief*

A.3 Salient Features of the Intel® XScale™ Core MegaCell

This section describes those features and implementation details of the Intel® XScale™ microarchitecture that drive the operating systems issues addressed in the remainder of this appendix. Some of these features are mandated by the ARM V5TE architecture; others are implementation options. The Intel® XScale™ microarchitecture offers the following:

- Implementation of a Harvard (split) L1 cache, providing a 32 Kbyte Instruction cache and a 32 Kbyte Data cache
- Caches that use virtual address indices (or tags)
- 2 Kbyte Mini-Data cache, separate from the Data cache
- Locked entries in the TLBs
- Locked cache lines in the caches
- Instruction sequence that acts as a synchronization barrier when modifying registers in Coprocessor 15
- Implements the PID register in Coprocessor 15
- Vector re-map capability in Coprocessor 15
- Extends the Exception model to include some imprecise Data and Prefetch aborts
- Extends Coprocessor 15 to provide access control to other implemented coprocessors
- Expands the set of supported cache-control options, using the X-bit in the implementation-dependent TEX field of both 1st- and 2nd-level descriptors
- Defines a Branch target buffer for Dynamic Branch prediction

A.4 Enabling the Caches

Any StrongARM-based chip is intended to perform best with both caches enabled and in Virtual mode. While the Instruction cache can be enabled while running in Physical mode, enabling the Data cache (and the Mini-Data cache) requires that Address translation be enabled. There is a significant performance gain from enabling the Instruction cache. The recommended temporal sequence is as follows:

1. Enable the Instruction cache
2. Enable translation
3. Drain the Write buffer
4. Enable the Data cache

These operations should not be combined.

A.5 Using the PID Register

The PID register, register 13 in Coprocessor 15, alleviates the need for cache flushes on Context switches. If a memory reference is made to a virtual address in which bits [31:25] are zeroes, the access is actually made with a modified virtual address, which is constructed by ORing the address with the value contained in the PID register. As a result, use of the PID register and process or thread execution in the virtual-address range [0:0x1ffffff] limits the size of that process or thread virtual address range to 32 Mbytes. The virtual-address range of a thread or process can grow beyond 32 Mbytes by allowing access to addresses outside this range.

In SA1 implementations, bit 31 was not defined in the PID register.

Using the PID register eliminates the need for many cache flushes until such time as the PID value rolls over, or a PID value is reused, or the processor is placed in Sleep mode.

Addresses used in Coprocessor 15 actions, such as flushing a single cache line, are not modified by the value of the PID register.

A.6 Exception Vector Remapping

Effective use of the PID register depends on re-mapping the Exception vectors from 0x0 to 0xFFFF0000. This is accomplished by setting bit 13 of register 1 in Coprocessor 15. Failure to do so would require every process virtual-address range to contain the Exception vectors located at 0x0.

A.7 Instruction Stream Barrier Code

Intel® XScale™ microarchitecture gets its speed from (among other implementation details) its ability to speculatively execute instructions out of order, although it cannot retire instructions out of order. However, for certain operations, it is imperative that the code knows with certainty that all instructions earlier in the instruction stream have completed and that no instructions later in the instruction stream have been dispatched. An example of this is the effect of enabling (or disabling) virtual-to-physical address translation, when it is necessary to know which instructions were fetched before the address range switch took effect, and which instructions will be fetched after the address range switch takes effect.

The following code is guaranteed to execute before the switch and no instruction following this code will be fetched until the switch has occurred. This code sequence is called the CPWAIT sequence. It is privileged code.

```
MACRO CPWAIT

    MRC P15, 0, Rx, Cj, C0, 0        ; read some register in CP15
    MOV Rx, Rx         ; wait for the read to complete
    SUB PC, PC, #4     ; branch to the next instruction,
                       ; flushing the instruction pipeline

MEND
```

At this point in the execution, all prior writes to registers in Coprocessor 15 are guaranteed to have completed.

A.8 Memory Management Concerns

ARM V5 introduces the concept of *tiny* (1 Kbyte) *pages*, while continuing to support *coarse* (4 Kbyte and 64 Kbyte) *pages* and *segments* (1 Mbyte).

Pages of any supported size can be mixed and matched within a 2nd-level page table. The 1st-level descriptor type indicates whether the 2nd-level table defines coarse pages or fine (tiny) pages.

In a coarse 2nd-level table, each page is assumed to be 4 Kbyte long. Defining a 64-Kbyte page requires that the same descriptor appear 16 times to cover the range.

In a fine-page table, it is assumed each page is 1 Kbyte in size. Consequently, a fine-page table is four times the size of a coarse-page table. Defining a 4 Kbyte page requires four copies of the descriptor to cover the range; defining a 64 Kbyte page requires 64 copies of the descriptor to cover the range.

Regardless, a 2nd-level page table should be aligned on a 1-Kbyte boundary.

Intel® XScale™ microarchitecture has extended the set of cache control attributes in the descriptors. In addition to the C and B bits, the X bit in the TEX field plays a role in the definition of the cache attributes of the virtual range. When the X bit is clear, the semantics of the C- and B-bits are as specified by the ARM architecture. This differs from SA1-based chips, in which the use of the Mini-Data cache overloaded the semantics of the B-bit. When the X-bit is set, the additional semantics are defined, including use of the Mini-Data cache and Write Allocation of cache lines. This is all summarized in Table A-1:

Table A-1. Cache Control Attributes

X C B	Cacheable?	Bufferable?	Write Policy	Notes
0 0 0	N	N		Stall until complete
0 0 1	N	Y		
0 1 0	Y	Y	Write Thru	
0 1 1	Y	Y	Write Back	
1 0 0				Do not use
1 0 1	N	Y		No coalescing
1 1 0				Mini Data cache*
1 1 1	Y	Y	Write Back	Read/Write Allocate

Note: * See details of the Auxiliary Control Register, Coprocessor 15.

It is not possible to define a page as Write Allocate/Write Through.

The Mini-Data cache does not support per-descriptor characteristics. Cache behavior is selected with the MD bit field in the Auxiliary Control register in Coprocessor 15.

The Intel® XScale™ microarchitecture Cache Control extensions provide several types of functionality. Some define genuine new possibilities (such as Cache Line Allocate on Write Access), some refine and control unusual functionality (such as the Mini-Data cache), and some provide a finer granularity of control (such as precluding coalescing in the Write buffer).

If and how these capabilities are used in a system are part of the design and implementation details of the operating system.

The Intel® XScale™ core defines a set of events that cause Prefetch aborts. This is an extension to the architecture that can be used by the Prefetch Abort handler as needed. A Prefetch abort is signaled only when the instruction is required, or when the cache line containing the required instruction must be fetched.

The Intel® XScale™ core extends the set of events causing Data aborts and introduces several imprecise exceptions. Those causes that are specific to the Intel® XScale™ core are tagged by setting bit 10 in the Fault Status register. Imprecise exceptions should be considered unrecoverable. Recognizing and responding to the additional causes of precise Data aborts requires enhancement of the Data Abort handler.

A.9 Locking TLB Entries

The Intel® XScale™ microarchitecture allows individual entries to be locked in the TLBs. Each locked TLB entry reduces the number of TLB entries available to hold other translation information. The entries one would expect to lock in the TLBs are those used during access to locked cache lines.

A TLB Global Invalidate does not affect locked entries.

A.10 DSP Coprocessor 0

The Intel® XScale™ core has defined a DSP coprocessor containing a 40-bit accumulator and 8 instructions to manipulate it, as a performance boost for audio processing algorithms. CP0 is accessible in User mode. Saving and restoring the contents of the accumulator must be addressed during context switch.

Because the amount of data residing in CP0 is small, it is not unreasonable to simply add the contents of the accumulator to the context definition. The “double width” memory referencing instructions *LDRD* and *STRD* are defined in the extended instruction set for this purpose.

Support for an alternative implementation is provided. In CP15, register 15 (the CP Access register) can be used to grant/prohibit User mode access to CP0. Any User mode access attempt to use the coprocessor while access is prohibited results in an *undefined* exception, which allows the operating system to implement late Save/Restore of the accumulator contents by implementing the notion of “CP0 owner,” in a manner analogous to “floating point registers owner.”

A.11 Instruction Cache Invalidation

The Intel® XScale™ core implements both single cache line and entire cache invalidation. Either operation requires only one instruction using register 7 of Coprocessor 15. Invalidating the entire cache invalidates the Branch Target buffer automatically. It is strongly recommended that if a single cache line is invalidated, the Branch Target buffer also be invalidated.

A.12 Data Cache Flushing

The Intel® XScale™ core provides cache functions to force a cache line Write back if dirty, to invalidate a single cache line, and to invalidate the entire Data cache.

Completely flushing Data cache, to cause all dirty data to be written out, is a costly operation. There are two considerations here: (1) if possible, data RAM should be used to cause dirty data to be written back to memory instead of a data region that must be read into the cache from memory; and (2) if possible, a 32-Kbyte virtual address range should be defined to be used for no other purpose than flushing the Data cache, since doing so reduces by half the effort required to flush the Data cache.

Assuming both conditions are met, the following algorithm (using data RAM) will cause all dirty data in the Data cache to be written back to memory.

```
;  set baseaddress as the first virtual address of a 32kbyte range used only
;  to flush the Data cache.   base address should be aligned on a 32byte
;  boundary.  if the virtual range beginning with baseaddress is used for any
;  purpose other than flushing the Data cache, then cachelinecount must be
;  doubled, to 2048.

set     cachelinecount, 1024
set     cachelinesize, 32

    mov r1, #cachelinecount
    ldr r0, =baseaddress
l1:
    mcr p15, 0, r0, c7, c2, 5; allocate a cache line for  [r0]
    add r0, r0, #cachelinesize
    subsr1, r1, #1
    bne l1
```

Flushing the Mini-Data cache requires the same sort of analysis. If the Mini-Data cache is configured as Write-through, no further action is required. If a 2-Kbyte virtual-address range can be defined that is used for no purpose other than flushing the dirty data from the Mini-Data cache, then that is preferred. If that is not possible, it will be necessary to read a 4-Kbyte virtual-address range into the Mini-Data cache to guarantee that all dirty data is written back. The Mini-Data cache, of course, need not be flushed if the operating system provides no mechanism for its use.

```
;  SET BASEADDRESS AS THE FIRST VIRTUAL ADDRESS OF A 2KBYTE RANGE USED ONLY TO
;  flush the mini Data cache.  baseaddress should be aligned on a 32byte
;  boundary.  if the virtual range beginning with baseaddress is used for any
;  purpose other then flushing the mini Data cache, then cachelinecount must
; be doubled, to 128.

    set cachelinecount, 64
    set cachelinesize, 32

    mov r1, #cachelinecount
    ldr r0, =baseaddress
l2:
    ldr r2, [r0], #cachelinesize
    subsr1, r1, #1
    bne L2
```

Finally, invalidate the Data caches, since the contents are now unpredictable.

```
;  invalidate the data and mini Data caches.

    mcr p15, 0, r0, c7, c6, 0
```

A.13 Locking and Unlocking the Caches

Cache contents can be incrementally locked on a line-by-line basis. Only global cache unlocking is supported. To simulate incremental unlocking of cache contents, the operating system must be prepared to relock the cache lines it does not want unlocked.

A.14 Locking Code in the Instruction Cache

The Intel® XScale™ core allows code to be locked in the Instruction cache. Up to 28 cache lines can be locked in a set. Any attempt to lock more than 28 cache lines in a set is silently ignored.

There are several requirements that must be met when locking code in the Instruction cache:

1. The Instruction cache must be enabled
2. Translation must be enabled
3. The code that performs the locking must be cache inhibited
4. No Instruction cache-line fill can occur while the locking activity is in progress

This last requirement implies care in the placement of the code that performs the locking; that code cannot reside too close to a cacheable region from which a prefetch might occur. The locking code cannot reside within 128 bytes of a cacheable region.

A.15 Unlocking the Instruction Cache

Only global Instruction-cache unlocking is supported. The contents of the cache remain valid after unlocking.

A.16 Locking Data in the Data Cache

The Intel® XScale™ core allows cache lines to be locked in the Data cache. Up to 28 cache lines can be locked in a set. Any attempt to lock more than 28 cache lines in a set is silently ignored. The Mini- Data cache does not support locking cache lines.

There are two distinct ways to lock data in the Data cache. The first, called *data locking*, requires that the virtual addresses be backed by physical memory. The second, called *data RAM*, allows the definition of a virtual-address range that is not backed by physical memory. While locked data may be either Write-back or Write-through, data RAM must be Write-back.

Although the virtual range defined as data RAM does not get backed by physical memory, the page-table descriptors must be completed so that the necessary permissions checking can be performed.

A.17 Unlocking the Data Cache

Only global Data cache unlocking is supported. The data RAM must be invalidated, not cleaned, once the Data cache is unlocked.

A.18 Branch Target Buffer (BTB)

The Intel Intel® XScale™ microarchitecture implements a Dynamic-Branch Prediction scheme to reduce the penalties associated with program-flow changes. The scheme rests upon the contents of the Branch-Target buffer, a 128-entry, direct-mapped cache. The BTB is disabled and invalidated on reset. It is automatically invalidated whenever the PID register (Coprocessor 15, register 13) is written, and whenever the instruction cache is invalidated. The BTB is enabled manually by writing a 1 to bit 11 in Coprocessor 15, register 1. The BTB is manually invalidated using a Coprocessor 15, register 7 function. If the PID register is not being used, the BTB should be flushed on context switch.

A.19 Exception Model

The Intel® XScale™ core implements the *Base Restored Abort Model.* If a precise Data abort is signaled during the execution of a Memory-Referencing instruction, the base register is not updated. This allows any instruction to be restarted after servicing the Data abort.

The Intel® XScale™ core has extended the list of conditions raising undefined exceptions, Prefetch aborts, and Data aborts. The Exception handler needs to be prepared to deal with these extensions.

It is strongly recommended that the first task completed in an Exception handler is to “save state.”

A.20 Power Management

The Intel® XScale™ microarchitecture defines three low-power modes: Idle, Drowsy and Sleep. All state information is lost on entering Sleep mode. The only way to exit Sleep mode is through the reset sequence. State is retained in Idle and Drowsy modes. Both Idle and Drowsy modes are exited by interrupt, even if the interrupt is masked. A single Coprocessor 14 register Write is used to enter any low-power mode.

If the system provides a data repository (a set of registers) that is preserved across Sleep mode, it is possible to save and recover state. Any support for this is external to the Intel® XScale™ microarchitecture.

A.21 Assembly Language Considerations

Refer to this *Programmers Reference Manual* for information that describes some performance implications in assembly language programming.

ARM Glossary B

Term	Description
26-bit architecture	Earlier versions of the ARM* architecture (ARMv1, ARMv2, and ARMv2a) that implement only a 26-bit address space.
32-bit architecture	Versions of ARM architecture (version 3 and above) that implement a 32-bit address space.
Abort	Caused by an illegal memory access. Aborts can be caused by the external memory system or the MMU.
Abort model	Describes what happens to the processor state when a Data Abort exception occurs. Different abort models behave differently regarding load/store instructions that specify base register writeback.
Addressing modes	Generally refers to a procedure shared by many different instructions, for generating values used by the instructions. For four of the ARM addressing modes, the values generated are memory addresses (which is the traditional role of an addressing mode). A fifth addressing mode generates values to be used as operands by data-processing instructions.
AL (always)	Specifies that the instruction is executed irrespective of the value of the condition code flags. If no condition code is given with an instruction mnemonic, the AL condition code is used.
ALU	Arithmetic Logic Unit
AND	Performs a bitwise AND.
Arithmetic_Shift_Right	Performs a right shift, repeatedly inserting the original left-most bit (the sign bit) in the vacated bit positions to the left.
ARM instruction	Specifies operation for ARM processor to perform. ARM instructions must be word-aligned.
Assert statements	Used in pseudo-code to indicate that a certain condition has been met.
Assignment	Signified by =.
Banked registers	Register numbers whose physical register is defined by the current processor mode. The banked registers are registers R8 to R14.
Base register	A register specified by a load/store instruction that is used as the base value for the instruction's address calculation. Depending on the instruction and its addressing mode, an offset can be added to or subtracted from the base register value to form the virtual address that is sent to memory.
Base register writeback	When base register used in address calculation has a modified value written to it.
Big-endian memory	A byte or halfword at a word-aligned address is the most significant byte or halfword within the word at that address; a byte at a halfword-aligned address is the most significant byte within the halfword at that address.
Binary numbers	Base 2 numbers (0, 1); binary numbers are preceded by 0b.
Boolean AND	Signified by the `AND` operator.
Boolean OR	Signified by the `OR` operator.
BorrowFrom	Returns 1 if subtraction specified as its parameter caused a borrow (the true result is less than 0, where operands are treated as unsigned integers), and returns 0 in all other cases. This process delivers additional information about a subtraction that occurred earlier in the pseudo-code. Subtraction is not repeated.
Branch prediction	Where an ARM implementation chooses a future execution path to prefetch along (see **Prefetching**). For example, after a branch instruction, the implementation can choose to prefetch either the instruction following the branch, or the instruction at the branch target.
Byte	An 8-bit data item.

Term	Description
Cache	A block of high-speed memory locations whose addresses are changed automatically in response to which memory locations the processor is accessing, and whose purpose is to increase the average speed of a memory access.
Cache contention	When the number of frequently used memory cache lines that use a particular cache set exceeds the set-associativity of the cache. In this case, main memory activity goes up and performance drops.
Cache hit	A memory access that can be processed at high speed because the data it addresses is already in the cache.
Cache line	Is basic unit of storage in a cache. Size is always a power of two (usually 4 or 8 words), and is required to be aligned to a suitable memory boundary. A *memory cache line* is a block of memory locations with the same size and alignment as a cache line. Memory cache lines are sometimes loosely just called cache lines.
Cache line index	A number associated with each cache line in a cache set. Within each cache set, the cache lines are numbered from 0 to (set associativity)–1.
Cache lockdown	Alleviates the delays caused by accessing a cache in a worst-case situation. Cache lockdown allows critical code and data to be loaded into the cache so that the cache lines containing them are not subsequently re-allocated. This process ensures that all subsequent accesses to the code and data concerned are cache hits and therefore complete quickly.
Cache lockdown blocks	One line from each cache set. Cache lockdown is performed in units of a cache lockdown block.
Cache miss	A memory access that cannot be processed at high speed because the data it addresses is not in the cache.
Cache pollution	When used data is loaded into the cache.
Cache pressure	Data not temporal to the current process is loaded in the cache.
Cache sets	Areas of a cache divided up to simplify and shorten the process of determining whether a cache hit occurs. The number of cache sets is always a power of two.
Callee-save registers	Registers that a called procedure must preserve. To preserve a callee-save register, the called procedure would normally either not use the register at all, or store the register to the stack during procedure entry and re-load it from the stack during procedure exit.
Caller-save registers	Registers that a called procedure need not preserve. If the calling procedure requires the values to be preserved, it must store and reload them itself.
CarryFrom	Returns 1 if the addition specified as its parameter caused a carry (true result is bigger than 2^{32}–1, where the operands are treated as unsigned integers), and returns 0 in all other cases. This process delivers additional information about an addition that occurred earlier in the pseudo-code. The addition is not repeated.
case ... endcase statements	Indicate a one-of-many execution option. Indentation indicates the range of statements in each option.
Comments	Non-executable information enclosed in /* */.
Condition field	A four-bit field in an instruction that specifies a condition under which the instruction can execute.
Conditional execution	If the condition code flags indicate that the corresponding condition is true when the instruction starts executing, it executes normally. Otherwise, the instruction does nothing.
ConditionPassed (cond)	Returns TRUE if the state of the N, Z, C and V flags fulfills the condition encoded in the cond argument, and returns FALSE in all other cases.
Control bits	Are the bottom eight bits of a *Program Status Register* (PSR). The control bits change when an exception arises and can be altered by software only when the processor is in a privileged mode.
CPSR	Current Program Status Register
Critical word firs	When a cache line is being transferred, the first word transferred corresponds to the one needed by the processor immediately, as opposed to transferring the data from lowest address first.

Term	Description
CurrentModeHasSPSR()	Returns TRUE: if the current processor mode is not User mode or System mode, Returns FALSE: if the current mode is User mode or System mode.
Data cache	A separate cache used only for processing data loads and stores.
Decode bits	Bits[27:20] and bits[7:4] of an ARM instruction, and are the main bits that determine the type of instruction to be executed.
Digital signal processing	Refers to variety of algorithms used to process signals that have been sampled and converted to digital form. **Saturated arithmetic** is often used in such algorithms.
Direct-mapped cache	A one-way **set-associative cache**. Each cache set consists of a single cache line, so cache look-up needs only to select and check one cache line.
Direct Memory Access	An operation that accesses main memory directly, without the processor performing any accesses to the data concerned.
Domain	A collection of sections, large pages, and small pages of memory, that can have their access permissions switched rapidly by writing to the Domain Access Control Register (CP15 register 3).
Do-not-modify fields (DNM)	A value that must not be altered by software. DNM fields read as UNPREDICTABLE values, and can only be written with the same value read from the same field on the same processor. In ARM documentation, DNM fields are sometimes followed by RAZ or RAO in parentheses as a guideline to implementers as to which way the bits should read for future compatibility; however, programmers must not rely on this behavior.
Double-precision value	Two 32-bit words that must appear consecutively in memory, must both be word-aligned, and which is interpreted as a basic double-precision floating-point number according to the IEEE 754-1985 standard.
Doubleword	A 64-bit data item. Doublewords are normally at least word-aligned in ARM systems.
Doubleword-aligned	An address that is divisible by 8.
DSP	*See* **Digital signal processing**
Elements	Separated by | in a list of possible values for a variable.
Endianness	An aspect of the system memory mapping. See **big-endian** and **little-endian**.
EOR	Performs a bitwise Exclusive OR.
Exception	An event handler. For example, an exception could handle an external interrupt or an undefined instruction.
Exception modes	Privileged modes that are entered when specific exceptions occur.
Exception vector	One of a number of fixed addresses in low memory, or in high memory if high vectors are configured.
External abort	An abort that is generated by the external memory system.
Fault	An abort that is generated by the MMU.
FCSE (Fast Context Switch Extension)	Modifies the behavior of an ARM memory system to allow multiple programs running on the ARM processor to use identical address ranges, while ensuring that the addresses they present to the rest of the memory system differ.
Flat address mapping	Where the physical address for every access is equal to its virtual address.
Flush-to-zero mode	A special processing mode that optimizes the performance of some VFP algorithms by replacing the denormalized operands and intermediate results with zeros, without significantly affecting the accuracy of the final results.
Floating-point Exception Register	A read/write register, two bits of which provide system-level status and control. The remaining bits of this register can communicate exception information between the hardware and software components of the implementation, in an IMPLEMENTATION DEFINED manner.
Floating-point Status and Control Register	A read/write register that provides all user-level status and control of the floating-point system.
Floating-point System ID Register	Rread-only register whose value indicates which VFP implementation is being used.

Term	Description
for ... statements	Indicate a loop over a numeric range. Indentation is used to indicate the range of statements in the loop.
FPEXC	*See* Floating-point Exception Register.
FPSCR	*See* Floating-point Status and Control Register.
FPSID	*See* Floating-point System ID Register.
Fully-associative cache	Single cache set, consists of the entire cache. See also **direct-mapped cache**.
General-purpose register	One of the 32-bit general-purpose integer registers, R0 to R15. Note that R15 holds the Program Counter (and the PSR as well in the 26-bit architectures), and often there are limitations on its use that do not apply to R0 to R14.
Halfword	A 16-bit data item. Halfwords are normally halfword-aligned in ARM systems.
Halfword-aligned	An address that is divisible by 2.
Hexadecimal numbers	Numbers preceded by `0x` and are given in a monospaced font.
High registers	ARM registers 8 to 15 that can be accessed by some Thumb instructions.
High vectors	Alternative locations for exception vectors. The high vector address range is near the top of the address space, rather than at the bottom.
if ... else if ... else statements	Used to signify conditional statements. Indentation indicates the range of statements in each option.
IGNORE fields (IGN)	Must ignore writes.
IMB	*See* Instruction Memory Barrier.
Instruction Memory Barrier	A sequence of operations that can be used in the middle of a **self-modifying code** sequence to make it execute reliably. This sequence often depends on the ARM processor implementation and on the memory system implementation.
Immediate and offset fields	Unsigned unless otherwise stated.
Immediate values	Values encoded directly in instruction and used as numeric data when instruction is executed. Many ARM and Thumb instructions allow small numeric values to be encoded as immediate values within the instruction that operates on them.
IMP	An abbreviation used in diagrams to indicate that the bit or bits concerned have **IMPLEMENTATION DEFINED** behavior.
IMPLEMENTATION DEFINED fields	Behavior that is not architecturally defined, but should be defined and documented by individual implementations.
InAPrivilegedMode()	Returns TRUE if the current processor mode is not User mode, and returns FALSE if the current mode is User mode.
Index register	Register specified in some load/store instructions. Value of this register is used as an offset to be added to or subtracted from the base register value to form the virtual address, which is sent to memory. Some addressing modes optionally allow the index register value to be shifted prior to the addition or subtraction.
Inline literals	Inline literals are constant addresses and other data items held in the same area as the code itself. They are automatically generated by compilers, and can also appear in assembler code.
Instruction cache	A separate cache used only for processing instruction fetches.
Interworking	A method of working that allows branches between ARM and Thumb code.
Little-endian memory	A byte or halfword at a word-aligned address is the least significant byte or halfword within the word at that address; a byte at a halfword-aligned address is the least significant byte within the halfword at that address.
Load/Store architecture	An architecture where data-processing operations only operate on register contents, not directly on memory contents.
Logical_Shift_Left	Performs a left shift, inserting zeros in the vacated bit positions on the right. << is used as a short form for `Logical_Shift_Left`.
Logical_Shift_Right	Performs a right shift, inserting zeros in the vacated bit positions on the left.
Long branch	The use of a load instruction to branch to anywhere in the 4GB address space.
LR (Link Register)	Integer register R14.

Term	Description
Memory[<address>,<size>]	Refers to data item in memory of length `<size>`, at address `<address>`, aligned on a `<size>` byte boundary. The data item is zero-extended to 32 bits. Currently defined sizes are: 1 for bytes 2 for halfwords 4 for words To align on a `<size>` boundary, halfword accesses ignore `<address>`[0] and word accesses ignore `<address>`[1:0].
Memory coherency	Memory coherency is the problem of ensuring that when a memory location is read (either by data read or an instruction fetch), the value actually obtained is always value that was most recently written to the location. This process can be difficult for multiple possible physical locations, such as main memory, a write buffer and/or cache(s).
Memory Management Unit	Allows detailed control of a memory system. Most of the control is provided via **translation tables** held in memory.
Memory-mapped I/O	Uses special memory addresses that supply I/O functions when they are loaded from or stored to.
Modified Virtual Address	The address produced by the FCSE that is sent to the rest of the memory system to be used in place of the normal virtual address.
MMU	*See* Memory Management Unit.
MVA	*See* Modified Virtual Address.
NaN	Not a Number; a type of floating-point value.
NOT	Performs a bitwise complement.
NotFinished(CP_number)	Returns TRUE if the coprocessor signified by the `CP_number` argument has signalled that the current operation is incomplete, and returns FALSE if the operation is complete.
NumberOfSetBitsIn(bitfield)	Performs a population count on (counts the set bits in) the bitfield argument.
Object[from:to]	Indicates bit field extracted from Object, starting at bit “from”, ending with bit “to” (inclusive).
Offset addressing	A memory address is formed by adding or subtracting an offset to or from the base register value.
Optional parts of instructions	Surrounded by `{` and `}`.
OR	Performs a bitwise Inclusive OR.
OverflowFrom	Returns 1 if the addition or subtraction specified as its parameter caused a 32-bit signed overflow. Addition generates an overflow if both operands have the same sign (bit[31]), and the sign of the result is different to the sign of both operands. Subtraction causes an overflow if the operands have different signs, and the first operand and the result have different signs. OverflowFrom delivers additional information about an addition or subtraction that occurred earlier in pseudo-code. The addition or subtraction is not repeated.
PC (Program Counter)	Integer register R15 (or bits[25:2] of R15 on 26-bit architectures).
PCB (Process Control Block)	In software systems that support multiple software processes, the PCB is a data structure associated with each process that holds the process state while it is not executing.
Physical address	Identifies a main memory location.
Predictable subsequent execution	Execution of any instructions that can be reached subsequently by any combination of normal sequential execution and executing branches with statically determined targets. Any instruction that branches to a location that depends on register values (such as `MOV PC,LR`) terminates predictable subsequent execution
Post-indexed addressing	The memory address is the base register value, but an offset is added to or subtracted from the base register value and the result is written back to the base register.

Term	Description
Prefetch scheduling distance	The number of iterations to prefetch data when adding prefetch to a loop that operates on relays; to make it easy to predict where to fetch data.
Prefetching	The process of fetching instructions from memory before the instructions that precede them have finished executing. Prefetching an instruction does not mean that the instruction has to be executed.
Pre-indexed addressing	The memory address is formed in the same way as for **offset addressing**, but the memory address is also written back to the base register.
Privileged mode	Any processor mode other than User mode. Memory systems typically check memory accesses from privileged modes against supervisor access permissions rather than the more restrictive user access permissions. The use of some instructions is also restricted to privileged modes.
Process ID	In the FCSE, this is a 7-bit number that identifies which process block the current process is loaded into.
Protection region	A memory range whose position, size, and other properties are defined by Protection Unit registers.
Protection Unit	A hardware unit whose registers provide simple control of a limited number of protection regions in memory.
PSR	The CPSR or one of the SPSRs (or bits[31:26] and bits[1:0] of register 15 on 26-bit architectures).
Quiet NaN	A NaN that propagates unchanged through most floating-point operations.
Read-allocate cache	A cache in which a cache miss on storing data causes the data to be written to main memory. Cache lines are only allocated to memory locations when data is read/loaded, not when it is written/stored.
Read-As-Zero fields (RAZ)	Appear as zero when read.
Read-Modify-Write fields (RMW)	Read to a general-purpose register, the relevant fields updated in the register, and the register value written back.
RISC	Reduced Instruction Set Computer
Rotate_Right	Performs a right rotate, where each bit that is shifted off the right is inserted on the left.
Rounding error	The value of the rounded result of an arithmetic operation minus the exact result of the operation.
Rounding modes	Specify how the exact result of a floating-point operation is rounded to a value which is representable in the destination format.
Round to Nearest (RN) mode	The rounded result is the nearest representable number to the unrounded result.
Round towards Plus Infinity (RP) mode	The rounded result is the nearest representable number that is greater than or equal to the exact result.
Round towards Minus Infinity (RM) mode	The rounded result is the nearest representable number that is less than or equal to the exact result.
Round towards Zero (RZ) mode	Results are rounded to the nearest representable number that is no greater in magnitude than the unrounded result.
Saturated arithmetic	Integer arithmetic in which a result that would be greater than largest representable number is set to the largest representable number, and a result that would be less than the smallest representable number is set to the smallest representable number. Signed saturated arithmetic is often used in DSP algorithms. It contrasts with the normal signed integer arithmetic used in ARM processors, in which overflowing results wrap around from $+2^{31}-1$ to -2^{31} or vice versa.
Security hole	An illegal mechanism that bypasses system protection.
Self-modifying code	Code that writes one or more instructions to memory and then executes them. This type of code cannot be relied on without the use of an **IMB**.
Set-associativity	The number of cache lines in each of the cache sets in a cache. It can be any number ≥ 1, and is not restricted to being a power of two.
Shifter operand	One of the source operands of an ARM data-processing instruction. It is either an immediate value or a register.

Term	Description
Should-Be-One fields (SBO)	Should be written as 1 (or all 1s for bit fields) by software. Values other than 1 produce UNPREDICTABLE results.
Should-Be-One-or-Preserved fields (SBOP)	Should be written as 1 (or all 1s for bit fields) or preserved by writing the same value that has been previously read from the same fields on the same processor.
Should-Be-Zero fields (SBZ)	Should be written as zero (or all 0s for bit fields) by software. Non-zero values produce UNPREDICTABLE results.
Should-Be-Zero-or-Preserved fields (SBZP)	Should be written as zero (or all 0s for bit fields) or preserved by writing the same value that has been previously read from the same fields on the same processor.
Signaling NaNs	Causes an Invalid Operation exception whenever any floating-point operation receives a signaling NaN as an operand. Signaling Nans can be used in debugging to track down some uses of uninitialized variables.
Signed data types	Represent an integer in the range $–2^{N–1}$ to $+2^{N–1}– 1$, using two's complement format.
Signed immediate and offset fields	Fields are encoded in two's-complement notation unless otherwise stated.
SignedDoesSat(x,n)	Returns 0 if x lies inside the range of an n-bit signed integer (that is, if $–2^{(n–1)} \le x \le 2^{(n–1)} – 1$), and 1 otherwise. This operation delivers additional information about a SignedSat(x, n) operation that occurred earlier in the pseudo-code. Any operations used to calculate x or n are not repeated.
SignExtend(arg)	Sign-extends (propagates the sign bit) its argument to 32 bits.
SignedSat(x,n)	Returns x saturated to the range of an n-bit signed integer. That is, it returns: $–2^{(n–1)}$ if $x < –2^{(n–1)}$ x if $–2^{(n–1)} \le x \le 2^{(n–1)} – 1$ $2^{(n–1)} – 1$ if $x > 2^{(n–1)} – 1$.
SIMD	Single-Instruction, Multiple-Data operations
Single-precision value	A 32-bit word, must be word-aligned when held in memory, and is interpreted as a basic single-precision floating-point number according to the IEEE 754-1985 standard.
SP (Stack Pointer)	Integer register R13.
Spatial locality	The observed effect that after a program has accessed a memory location, it is likely to also access nearby memory locations in the near future. Caches with multiword cache lines exploit this effect to improve performance.
SPSR	Is the Saved Program Status register, which is associated with the current processor mode (and is undefined if there is no such Saved Program Status register, as in User mode or System mode).
SWI	Is a software interrupt.
Status registers	*See* **CPSR** and **SPSR**.
Tag bits	Bits[31:L+S]) of a virtual address, where L and S are the logarithms base 2 of the cache line length and the number of cache sets respectively. A cache hit occurs if the tag bits of the virtual address supplied by the ARM processor match the tag bits associated with a valid line in the selected cache set.
Temporal locality	The observed effect that after a program has accesses a memory location, it is likely to access the same memory location again in the near future. Caches exploit this effect to improve performance.
Test for equality	Signified by ==.
Thumb instruction	A halfword that specifies an operation for an ARM processor in Thumb state to perform. Thumb instructions must be halfword-aligned.
TLB	*See* **Translation Lookaside Buffer**.
TLB lockdown	A way to prevent specific translation table walk results being accessed. This method ensures that accesses to the associated memory areas never cause a translation table walk.

Term	Description
Translation Lookaside Buffer	A memory structure containing the results of translation table walks; helps reduce the average cost of a memory access. Usually, there is a TLB for each memory interface of the ARM implementation.
Translation tables	Tables held in memory. They define the properties of memory areas of various sizes from 1KB to 1MB.
Translation table walk	The process of doing a full translation table lookup. It is performed automatically by hardware.
Trap enable bits	Determines whether trapped or untrapped exception handling is selected. If trapped exception handling is selected, the way it is carried out is IMPLEMENTATION DEFINED.
Unaffected items	Items not changed by a particular operation.
Unaligned memory accesses	Memory accesses that are not appropriately word-aligned or halfword-aligned.
Unbanked registers	General-purpose registers that refer to the same 32-bit physical register in all processor modes. Unbanked registers are registers R0 to R7.
UNDEFINED	Indicates an instruction that generates an undefined instruction trap.
Unified cache	A cache used for both processing instruction fetches and processing data loads and stores.
Unindexed addressing	Indicates addressing in which the base register value is used directly as the virtual address to send to memory, without adding or subtracting an offset. In most types of addressing mode, unindexed addressing is performed by using offset addressing with an immediate offset of 0. ARM Addressing Mode 5 (used for LDC and STC instructions) has an explicit unindexed addressing mode that allows the offset field in the instruction to be used to specify additional coprocessor options.
UNPREDICTABLE	Means the result of an instruction cannot be relied upon. UNPREDICTABLE instructions or results must not represent security holes. UNPREDICTABLE instructions must not halt or hang the processor, or any parts of the system.
UNPREDICTABLE fields (UNP)	UNPREDICTABLE fields do not contain valid data, and a value can vary from moment to moment, instruction to instruction, and implementation to implementation.
Unsigned data types	Represent a non-negative integer in the range 0 to $+2^N–1$, using normal binary format.
Variable parts of instructions	Surrounded by < and >.
VFP	*See* Vector Floating-point Architecture.
Vector Floating-point Architecture	A coprocessor extension to the ARM architecture that provides single-precision and double-precision floating-point arithmetic.
VFP emulator	An implementation that consists of software only, with all floating-point arithmetic being emulated by ARM routines.
Virtual address	An address generated by an ARM processor.
while .... statements	Used to indicate a loop. Indentation indicates the range of statements in the loop.
Word	A 32-bit data item. Words are normally word-aligned in ARM systems.
Word-aligned	An address that is address is divisible by 4.
Write-allocate cache	A cache in which a cache miss on storing data causes a cache line to be allocated and main memory contents to be read into it, followed by writing the stored data into the cache line.
Write-back Cache	Is a cache in which when a cache hit occurs on a store access, the data is only written to the cache. Data in the cache can therefore be more up-to-date than data in main memory. Any such data is written back to main memory when the cache line is cleaned or re-allocated. Another common term for a write-back cache is a *copy-back cache*.
Write-through cache	Is a cache in which when a cache hit occurs on a store access, the data is written both to the cache and to main memory. This is normally done via a write buffer, to avoid slowing down the processor.
Write buffer	Block of high-speed memory whose purpose is to optimize main memory stores.

Index