ARM Architecture Reference Manual

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved. ARM DDI 0100I

**ARM Architecture Reference Manual**

Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.

**Release Information**

The following changes have been made to this document.

**Change History**

**Date Issue Change**

February 1996 A First edition

July 1997 B Updated and index added

April 1998 C Updated

February 2000 D Updated for ARM architecture v5

June 2000 E Updated for ARM architecture v5TE and corrections to Part B

July 2004 F Updated for ARM architecture v6 (Confidential)

December 2004 G Updated to incorporate corrections to errata

March 2005 H Updated to incorporate corrections to errata

July 2005 I Updated to incorporate corrections to pseudocode and graphics

**Proprietary Notice**

ARM, the ARM Powered logo, Thumb, and StrongARM are registered trademarks of ARM Limited.

The ARM logo, AMBA, Angel, ARMulator, EmbeddedICE, ModelGen, Multi-ICE, PrimeCell, ARM7TDMI, ARM7TDMI-S, ARM9TDMI, ARM9E-S, ETM7, ETM9, TDMI, STRONG, are trademarks of ARM Limited.

All other products or services mentioned herein may be trademarks of their respective owners.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith.

1. Subject to the provisions set out below, ARM hereby grants to you a perpetual, non-exclusive, nontransferable, royalty free, worldwide licence to use this ARM Architecture Reference Manual for the purposes of developing; (i) software applications or operating systems which are targeted to run on microprocessor cores distributed under licence from ARM; (ii) tools which are designed to develop software programs which are targeted to run on microprocessor cores distributed under licence from ARM; (iii) or having developed integrated circuits which incorporate a microprocessor core manufactured under licence from ARM.

2. Except as expressly licensed in Clause 1 you acquire no right, title or interest in the ARM Architecture Reference Manual, or any Intellectual Property therein. In no event shall the licences granted in Clause 1, be construed as granting you expressly or by implication, estoppel or otherwise, licences to any ARM technology other than the ARM Architecture Reference Manual. The licence grant in Clause 1 expressly excludes any rights for you to use or take into use any ARM patents. No right is granted to you under the provisions of Clause 1 to; (i) use the ARM Architecture Reference Manual for the purposes of developing or having developed microprocessor cores or models thereof which are compatible in whole or part with either or both the instructions or programmer's models described in this ARM Architecture Reference

ii *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

Manual; or (ii) develop or have developed models of any microprocessor cores designed by or for ARM; or (iii) distribute in whole or in part this ARM Architecture Reference Manual to third parties, other than to your subcontractors for the purposes of having developed products in accordance with the licence grant in Clause 1 without the express written permission of ARM; or (iv) translate or have translated this ARM Architecture Reference Manual into any other languages.

3.THE ARM ARCHITECTURE REFERENCE MANUAL IS PROVIDED "AS IS" WITH NO WARRANTIES EXPRESS, IMPLIED OR STATUTORY, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF SATISFACTORY QUALITY, NONINFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE.

4. No licence, express, implied or otherwise, is granted to LICENSEE, under the provisions of Clause 1, to use the ARM tradename, in connection with the use of the ARM Architecture Reference Manual or any products based thereon. Nothing in Clause 1 shall be construed as authority for you to make any representations on behalf of ARM in respect of the ARM Architecture Reference Manual or any products based thereon.

Copyright © 1996-1998, 2000, 2004, 2005 ARM limited

110 Fulbourn Road Cambridge, England CB1 9NJ

Restricted Rights Legend: Use, duplication or disclosure by the United States Government is subject to the restrictions set forth in DFARS 252.227-7013 (c)(1)(ii) and FAR 52.227-19

This document is Non-Confidential. The right to use, copy and disclose this document is subject to the licence set out above.

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* iii

iv *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

Contents **ARM Architecture Reference Manual**

**Preface**About this manual ................................................................................ xii Architecture versions and variants ...................................................... xiii Using this manual .............................................................................. xviii Conventions ........................................................................................ xxi Further reading .................................................................................. xxiii Feedback .......................................................................................... xxiv

**Part A CPU Architecture**

**Chapter A1 Introduction to the ARM Architecture**

A1.1 About the ARM architecture ............................................................. A1-2 A1.2 ARM instruction set .......................................................................... A1-6 A1.3 Thumb instruction set ..................................................................... A1-11

**Chapter A2 Programmers’ Model**

A2.1 Data types ........................................................................................ A2-2 A2.2 Processor modes ............................................................................. A2-3 A2.3 Registers .......................................................................................... A2-4 A2.4 General-purpose registers ............................................................... A2-6 A2.5 Program status registers ................................................................ A2-11

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* v

*Contents*

A2.6 Exceptions ..................................................................................... A2-16 A2.7 Endian support ............................................................................... A2-30 A2.8 Unaligned access support .............................................................. A2-38 A2.9 Synchronization primitives ............................................................. A2-44 A2.10 The Jazelle Extension .................................................................... A2-53 A2.11 Saturated integer arithmetic ........................................................... A2-69

**Chapter A3 The ARM Instruction Set**

A3.1 Instruction set encoding ................................................................... A3-2 A3.2 The condition field ............................................................................ A3-3 A3.3 Branch instructions .......................................................................... A3-5 A3.4 Data-processing instructions ............................................................ A3-7 A3.5 Multiply instructions ........................................................................ A3-10 A3.6 Parallel addition and subtraction instructions ................................. A3-14 A3.7 Extend instructions ......................................................................... A3-16 A3.8 Miscellaneous arithmetic instructions ............................................ A3-17 A3.9 Other miscellaneous instructions ................................................... A3-18 A3.10 Status register access instructions ................................................ A3-19 A3.11 Load and store instructions ............................................................ A3-21 A3.12 Load and Store Multiple instructions .............................................. A3-26 A3.13 Semaphore instructions ................................................................. A3-28 A3.14 Exception-generating instructions .................................................. A3-29 A3.15 Coprocessor instructions ............................................................... A3-30 A3.16 Extending the instruction set .......................................................... A3-32

**Chapter A4 ARM Instructions**

A4.1 Alphabetical list of ARM instructions ................................................ A4-2 A4.2 ARM instructions and architecture versions ................................. A4-286

**Chapter A5 ARM Addressing Modes**

A5.1 Addressing Mode 1 - Data-processing operands ............................. A5-2 A5.2 Addressing Mode 2 - Load and Store Word or Unsigned Byte ...... A5-18 A5.3 Addressing Mode 3 - Miscellaneous Loads and Stores ................. A5-33 A5.4 Addressing Mode 4 - Load and Store Multiple ............................... A5-41 A5.5 Addressing Mode 5 - Load and Store Coprocessor ....................... A5-49

**Chapter A6 The Thumb Instruction Set**

A6.1 About the Thumb instruction set ...................................................... A6-2 A6.2 Instruction set encoding ................................................................... A6-4 A6.3 Branch instructions .......................................................................... A6-6 A6.4 Data-processing instructions ............................................................ A6-8 A6.5 Load and Store Register instructions ............................................. A6-15 A6.6 Load and Store Multiple instructions .............................................. A6-18 A6.7 Exception-generating instructions .................................................. A6-20 A6.8 Undefined Instruction space .......................................................... A6-21

vi *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Contents*

**Chapter A7 Thumb Instructions**

A7.1 Alphabetical list of Thumb instructions ............................................. A7-2 A7.2 Thumb instructions and architecture versions .............................. A7-125

**Part B Memory and System Architectures**

**Chapter B1 Introduction to Memory and System Architectures**

B1.1 About the memory system ............................................................... B1-2 B1.2 Memory hierarchy ............................................................................ B1-4 B1.3 L1 cache .......................................................................................... B1-6 B1.4 L2 cache .......................................................................................... B1-7 B1.5 Write buffers ..................................................................................... B1-8 B1.6 Tightly Coupled Memory .................................................................. B1-9 B1.7 Asynchronous exceptions .............................................................. B1-10 B1.8 Semaphores ................................................................................... B1-12

**Chapter B2 Memory Order Model**

B2.1 About the memory order model ........................................................ B2-2 B2.2 Read and write definitions ................................................................ B2-4 B2.3 Memory attributes prior to ARMv6 ................................................... B2-7 B2.4 ARMv6 memory attributes - introduction .......................................... B2-8 B2.5 Ordering requirements for memory accesses ................................ B2-16 B2.6 Memory barriers ............................................................................. B2-18 B2.7 Memory coherency and access issues .......................................... B2-20

**Chapter B3 The System Control Coprocessor**

B3.1 About the System Control coprocessor ............................................ B3-2 B3.2 Registers .......................................................................................... B3-3 B3.3 Register 0: ID codes ........................................................................ B3-7 B3.4 Register 1: Control registers .......................................................... B3-12 B3.5 Registers 2 to 15 ............................................................................ B3-18

**Chapter B4 Virtual Memory System Architecture**

B4.1 About the VMSA .............................................................................. B4-2 B4.2 Memory access sequence ............................................................... B4-4 B4.3 Memory access control .................................................................... B4-8 B4.4 Memory region attributes ............................................................... B4-11 B4.5 Aborts ............................................................................................. B4-14 B4.6 Fault Address and Fault Status registers ....................................... B4-19 B4.7 Hardware page table translation .................................................... B4-23 B4.8 Fine page tables and support of tiny pages ................................... B4-35 B4.9 CP15 registers ............................................................................... B4-39

**Chapter B5 Protected Memory System Architecture**

B5.1 About the PMSA .............................................................................. B5-2

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* vii

*Contents*

B5.2 Memory access sequence ............................................................... B5-4 B5.3 Memory access control .................................................................... B5-8 B5.4 Memory access attributes .............................................................. B5-10 B5.5 Memory aborts (PMSAv6) .............................................................. B5-13 B5.6 Fault Status and Fault Address register support ............................ B5-16 B5.7 CP15 registers ............................................................................... B5-18

**Chapter B6 Caches and Write Buffers**

B6.1 About caches and write buffers ........................................................ B6-2 B6.2 Cache organization .......................................................................... B6-4 B6.3 Types of cache ................................................................................. B6-7 B6.4 L1 cache ........................................................................................ B6-10 B6.5 Considerations for additional levels of cache ................................. B6-12 B6.6 CP15 registers ............................................................................... B6-13

**Chapter B7 Tightly Coupled Memory**

B7.1 About TCM ....................................................................................... B7-2 B7.2 TCM configuration and control ......................................................... B7-3 B7.3 Accesses to TCM and cache ........................................................... B7-7 B7.4 Level 1 (L1) DMA model .................................................................. B7-8 B7.5 L1 DMA control using CP15 Register 11 ......................................... B7-9

**Chapter B8 Fast Context Switch Extension**

B8.1 About the FCSE ............................................................................... B8-2 B8.2 Modified virtual addresses ............................................................... B8-3 B8.3 Enabling the FCSE .......................................................................... B8-5 B8.4 Debug and Trace ............................................................................. B8-6 B8.5 CP15 registers ................................................................................. B8-7

**Part C Vector Floating-point Architecture**

**Chapter C1 Introduction to the Vector Floating-point Architecture**

C1.1 About the Vector Floating-point architecture .................................... C1-2 C1.2 Overview of the VFP architecture .................................................... C1-4 C1.3 Compliance with the IEEE 754 standard ......................................... C1-9 C1.4 IEEE 754 implementation choices ................................................. C1-10

**Chapter C2 VFP Programmer’s Model**

C2.1 Floating-point formats ...................................................................... C2-2 C2.2 Rounding .......................................................................................... C2-9 C2.3 Floating-point exceptions ............................................................... C2-10 C2.4 Flush-to-zero mode ........................................................................ C2-14 C2.5 Default NaN mode ......................................................................... C2-16 C2.6 Floating-point general-purpose registers ....................................... C2-17 C2.7 System registers ............................................................................ C2-21

viii *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Contents*

C2.8 Reset behavior and initialization .................................................... C2-29

**Chapter C3 VFP Instruction Set Overview**

C3.1 Data-processing instructions ............................................................ C3-2 C3.2 Load and Store instructions ........................................................... C3-14 C3.3 Single register transfer instructions ................................................ C3-18 C3.4 Two-register transfer instructions ................................................... C3-22

**Chapter C4 VFP Instructions**

C4.1 Alphabetical list of VFP instructions ................................................. C4-2

**Chapter C5 VFP Addressing Modes**

C5.1 Addressing Mode 1 - Single-precision vectors (non-monadic) ......... C5-2 C5.2 Addressing Mode 2 - Double-precision vectors (non-monadic) ....... C5-8 C5.3 Addressing Mode 3 - Single-precision vectors (monadic) .............. C5-14 C5.4 Addressing Mode 4 - Double-precision vectors (monadic) ............ C5-18 C5.5 Addressing Mode 5 - VFP load/store multiple ................................ C5-22

**Part D Debug Architecture**

**Chapter D1 Introduction to the Debug Architecture**

D1.1 Introduction ...................................................................................... D1-2 D1.2 Trace ................................................................................................ D1-4 D1.3 Debug and ARMv6 ........................................................................... D1-5

**Chapter D2 Debug Events and Exceptions**

D2.1 Introduction ...................................................................................... D2-2 D2.2 Monitor debug-mode ........................................................................ D2-5 D2.3 Halting debug-mode ......................................................................... D2-8 D2.4 External Debug Interface ............................................................... D2-13

**Chapter D3 Coprocessor 14, the Debug Coprocessor**

D3.1 Coprocessor 14 debug registers ...................................................... D3-2 D3.2 Coprocessor 14 debug instructions .................................................. D3-5 D3.3 Debug register reference ................................................................. D3-8 D3.4 Reset values of the CP14 debug registers ..................................... D3-24 D3.5 Access to CP14 debug registers from the external debug interface .........

D3-25

**Glossary**

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ix

*Contents*

x *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

**Preface**

This preface describes the versions of the ARM® architecture and the contents of this manual, then lists the conventions and terminology it uses.

• *About this manual* on page xii

• *Architecture versions and variants* on page xiii

• *Using this manual* on page xviii

• *Conventions* on page xxi

• *Further reading* on page xxiii

• *Feedback* on page xxiv.

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* xi

*Preface*

**About this manual**

The purpose of this manual is to describe the ARM instruction set architecture, including its high code density Thumb® subset, and three of its standard coprocessor extensions:

• The standard System Control coprocessor (coprocessor 15), which is used to control memory system components such as caches, write buffers, Memory Management Units, and Protection Units.

• The *Vector Floating-point* (VFP) architecture, which uses coprocessors 10 and 11 to supply a high-performance floating-point instruction set.

• The debug architecture interface (coprocessor 14), formally added to the architecture in ARM v6 to provide software access to debug features in ARM cores, (for example, breakpoint and watchpoint control).

The 32-bit ARM and 16-bit Thumb instruction sets are described separately in Part A. The precise effects of each instruction are described, including any restrictions on its use. This information is of primary importance to authors of compilers, assemblers, and other programs that generate ARM machine code.

Assembler syntax is given for most of the instructions described in this manual, allowing instructions to be specified in textual form.

However, this manual is not intended as tutorial material for ARM assembler language, nor does it describe ARM assembler language at anything other than a very basic level. To make effective use of ARM assembler language, consult the documentation supplied with the assembler being used.

The memory and system architecture definition is significantly improved in ARM architecture version 6 (the latest version). Prior to this, it usually needs to be supplemented by detailed implementation-specific information from the technical reference manual of the device being used.

xii *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Preface*

**Architecture versions and variants**

The ARM instruction set architecture has evolved significantly since it was first developed, and will continue to be developed in the future. Six major versions of the instruction set have been defined to date, denoted by the version numbers 1 to 6. Of these, the first three versions including the original 26-bit architecture (the 32-bit architecture was introduced at ARMv3) are now OBSOLETE. All bits and encodings that were used for 26-bit features become RESERVED for future expansion by ARM Ltd.

Versions can be qualified with variant letters to specify collections of additional instructions that are included as an architecture extension. Extensions are typically included in the base architecture of the next version number, ARMv5T being the notable exception. Provision is also made to exclude variants by prefixing the variant letter with x, for example the xP variant described below in the summary of version 5 features.**Note** The xM variant which indicates that long multiplies (32 x 32 multiplies with 64-bit results) are not supported, has been withdrawn.

The valid architecture variants are as follows (variant in brackets for legacy reasons only):

ARMv4, ARMv4T, ARMv5T, (ARMv5TExP), ARMv5TE, ARMv5TEJ, and ARMv6

The following architecture variants are now OBSOLETE:

ARMv1, ARMv2, ARMv2a, ARMv3, ARMv3G, ARMv3M, ARMv4xM, ARMv4TxM, ARMv5, ARMv5xM, and ARMv5TxM

Details on OBSOLETE versions are available on request from ARM.

The ARM and Thumb instruction sets are summarized by architecture variant in *ARM instructions and architecture versions* on page A4-286 and *Thumb instructions and architecture versions* on page A7-125 respectively. The key differences introduced since ARMv4 are listed below.

**Version 4 and the introduction of Thumb (T variant)**

The Thumb instruction set is a re-encoded subset of the ARM instruction set. Thumb instructions execute in their own processor state, with the architecture defining the mechanisms required to transition between ARM and Thumb states. The key difference is that Thumb instructions are half the size of ARM instructions (16 bits compared with 32 bits). Greater code density can usually be achieved by using the Thumb instruction set in preference to the ARM instruction set. However, the Thumb instruction set does have some limitations:

• Thumb code usually uses more instructions for a given task, making ARM code best for maximizing performance of time-critical code.

• ARM state and some associated ARM instructions are required for exception handling.

The Thumb instruction set is always used in conjunction with a version of the ARM instruction set.

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* xiii

*Preface*

**New features in Version 5T**

This version extended architecture version 4T as follows:

• Improved efficiency of ARM/Thumb interworking

• Count leading zeros (CLZ, ARM only) and software breakpoint (BKPT, ARM and Thumb) instructions added

• Additional options for coprocessor designers (coprocessor support is ARM only)

• Tighter definition of flag setting on multiplies (ARM and Thumb)

• Introduction of the E variant, adding ARM instructions which enhance performance of an ARM processor on typical digital signal processing (DSP) algorithms:

— Several multiply and multiply-accumulate instructions that act on 16-bit data items. — Addition and subtraction instructions that perform saturated signed arithmetic. Saturated

arithmetic produces the maximum positive or negative value instead of wrapping the result if the calculation overflows the normal integer range. — Load (LDRD), store (STRD) and coprocessor register transfer (MCRR and MRRC) instructions that act

on two words of data.

— A preload data instruction PLD.

• Introduction of the J variant, adding the BXJ instruction and the other provisions required to support the Jazelle® architecture extension.

**Note** Some early implementations of the E variant omitted the LDRD, STRD, MCRR, MRCC and PLD instructions. These are designated as conforming to the ExP variant, and the variant is defined for legacy reasons only.

xiv *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Preface*

**New features in Version 6**

The following ARM instructions are added:

• CPS, SRS and RFE instructions for improved exception handling

• REV, REV16 and REVSH byte reversal instructions

• SETEND for a revised endian (memory) model

• LDREX and STREX exclusive access instructions

• SXTB, SXTH, UXTB, UXTH byte/halfword extend instructions

• A set of Single Instruction Multiple Data (SIMD) media instructions

• Additional forms of multiply instructions with accumulation into a 64-bit result.

The following Thumb instructions are added:

• CPS, CPY (a form of MOV), REV, REV16, REVSH, SETEND, SXTB, SXTH, UXTB, UXTH

Other changes to ARMv6 are as follows:

• The architecture name ARMv6 implies the presence of all preceding features, that is, ARMv5TEJ compliance.

• Revised Virtual and Protected Memory System Architectures.

• Provision of a Tightly Coupled Memory model.

• New hardware support for word and halfword unaligned accesses.

• Formalized adoption of a debug architecture with external and Coprocessor 14 based interfaces.

• Prior to ARMv6, the System Control coprocessor (CP15) described in Chapter B3 was a recommendation only. Support for this coprocessor is now mandated in ARMv6.

• For historical reasons, the rules relating to unaligned values written to the PC are somewhat complex prior to ARMv6. These rules are made simpler and more consistent in ARMv6.

• The *high vectors* extension prior to ARMv6 is an optional (IMPLEMENTATION DEFINED) part of the architecture. This extension becomes obligatory in ARMv6.

• Prior to ARMv6, a processor may use either of two abort models. ARMv6 requires that the *Base Restored Abort Model* (BRAM) is used. The two abort models supported previously were: — The BRAM, in which the base register of any valid load/store instruction that causes a memory

system abort is always restored to its pre-instruction value.

— The *Base Updated Abort Model* (BUAM), in which the base register of any valid load/store instruction that causes a memory system abort will have been modified by the base register writeback (if any) of that instruction.

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* xv

*Preface*

• The restriction that multiplication destination registers should be different from their source registers is removed in ARMv6.

• In ARMv5, the LDM(2) and STM(2) ARM instructions have restrictions on the use of banked registers by the immediately following instruction. These restrictions are removed from ARMv6.

• The rules determining which PSR bits are updated by an MSR instruction are clarified and extended to cover the new PSR bits defined in ARMv6.

• In ARMv5, the Thumb MOV instruction behavior varies according to the registers used (see note). Two changes are made in ARMv6.

— The restriction about the use of low register numbers in the MOV (3) instruction encoding is

removed. — In order to make the new side-effect-free MOV instructions available to the assembler language

programmer without changing the meaning of existing assembler sources, a new assembler syntax CPY Rd,Rn is introduced. This always assembles to the MOV (3) instruction regardless of whether Rd and Rn are high or low registers.

**Note** In ARMv5, the Thumb MOV Rd,Rn instructions have the following properties:

• If both Rd and Rn are low registers, the instruction is the MOV (2) instruction. This instruction sets the N and Z flags according to the value transferred, and sets the C and V flags to 0.

• If either Rd or Rn is a high register, the instruction is the MOV (3) instruction. This instruction leaves the condition flags unchanged.

This situation results in behavior that varies according to the registers used. The MOV(2) side-effects also limit compiler flexibility on use of pseudo-registers in a global register allocator.

**Naming of ARM/Thumb architecture versions**

To name a precise version and variant of the ARM/Thumb architecture, the following strings are concatenated: 1. The string ARMv. 2. The version number of the ARM instruction set. 3. Variant letters of the included variants. 4. In addition, the letter P is used after x to denote the exclusion of several instructions in the

ARMv5TExP variant.

The table *Architecture versions* on page xvii lists the standard names of the current (not obsolete) ARM/Thumb architecture versions described in this manual. These names provide a shorthand way of describing the precise instruction set implemented by an ARM processor. However, this manual normally uses descriptive phrases such as *T variants of architecture version 4 and above* to avoid the use of lists of architecture names.

xvi *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Preface*

All architecture names prior to ARMv4 are now OBSOLETE. The term ***all*** is used throughout this manual to refer to all architecture versions from ARMv4 onwards.

**Architecture versions**

**Name ARM instruction set**

**version**

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* xvii

**Thumb instruction set version Notes**

ARMv4 4 None -

ARMv4T 4 1 -

ARMv5T 5 2 -

ARMv5TExP 5 2 Enhanced DSP

instructions except LDRD, MCRR, MRRC, PLD, and STRD

ARMv5TE 5 2 Enhanced DSP

instructions

ARMv5TEJ 5 2 Addition of BXJ

instruction and Jazelle Extension support over ARMv5TE

ARMv6 6 3 Additional

instructions as listed in Table A4-2 on page A4-286 and Table A7-1 on page A7-125.

*Preface*

**Using this manual**

The information in this manual is organized into four parts, as described below.

**Part A - CPU Architectures**

Part A describes the ARM and Thumb instruction sets, and contains the following chapters:

**Chapter A1** Gives a brief overview of the ARM architecture, and the ARM and Thumb instruction sets.

**Chapter A2** Describes the types of value that ARM instructions operate on, the general-purpose registers that contain those values, and the Program Status Registers. This chapter also describes how ARM processors handle interrupts and other exceptions, endian and unaligned support, information on + synchronization primitives, and the Jazelle® extension.

**Chapter A3** Gives a description of the ARM instruction set, organized by type of instruction.

**Chapter A4** Contains detailed reference material on each ARM instruction, arranged alphabetically by

instruction mnemonic.

**Chapter A5** Contains detailed reference material on the addressing modes used by ARM instructions.

The term *addressing mode* is interpreted broadly in this manual, to mean a procedure shared by many different instructions, for generating values used by the instructions. For four of the addressing modes described in this chapter, the values generated are memory addresses (which is the traditional role of an addressing mode). The remaining addressing mode generates values to be used as operands by data-processing instructions.

**Chapter A6** Gives a description of the Thumb instruction set, organized by type of instruction. This

chapter also contains information about how to switch between the ARM and Thumb instruction sets, and how exceptions that arise during Thumb state execution are handled.

**Chapter A7** Contains detailed reference material on each Thumb instruction, arranged alphabetically by

instruction mnemonic.

xviii *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Preface*

**Part B - Memory and System Architectures**

Part B describes standard memory system features that are normally implemented by the System Control coprocessor (coprocessor 15) in an ARM-based system. It contains the following chapters:

**Chapter B1** Gives a brief overview of this part of the manual.

**Chapter B2** The memory order model.

**Chapter B3** Gives a general description of the System Control coprocessor and its use.

**Chapter B4** Describes the standard ARM memory and system architecture based on the use of a *Virtual*

*Memory System Architecture* (VMSA) based on a *Memory Management Unit* (MMU).

**Chapter B5** Gives a description of the simpler *Protected Memory System Architecture* (PMSA) based on

a *Memory Protection Unit* (MPU).

**Chapter B6** Gives a description of the standard ways to control caches and write buffers in ARM

memory systems. This chapter is relevant both to systems based on an MMU and to systems based on an MPU.

**Chapter B7** Describes the *Tightly Coupled Memory* (TCM) architecture option for level 1 memory.

**Chapter B8** Describes the Fast Context Switch Extension and Context ID support (ARMv6 only).

**Part C - Vector Floating-point Architecture**

Part C describes the *Vector Floating-point* (VFP) architecture. This is a coprocessor extension to the ARM architecture designed for high floating-point performance on typical graphics and DSP algorithms.

**Chapter C1** Gives a brief overview of the VFP architecture and information about its compliance with

the IEEE 754-1985 floating-point arithmetic standard.

**Chapter C2** Describes the floating-point formats supported by the VFP instruction set, the floating-point

general-purpose registers that hold those values, and the VFP system registers.

**Chapter C3** Describes the VFP coprocessor instruction set, organized by type of instruction.

**Chapter C4** Contains detailed reference material on the VFP coprocessor instruction set, organized

alphabetically by instruction mnemonic.

**Chapter C5** Contains detailed reference material on the addressing modes used by VFP instructions.

One of these is a traditional addressing mode, generating addresses for load/store instructions. The remainder specify how the floating-point general-purpose registers and instructions can be used to hold and perform calculations on vectors of floating-point values.

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* xix

*Preface*

**Part D - Debug Architecture**

Part D describes the debug architecture. This is a coprocessor extension to the ARM architecture designed to provide configuration, breakpoint and watchpoint support, and a *Debug Communications Channel* (DCC) to a debug host.

**Chapter D1** Gives a brief introduction to the debug architecture.

**Chapter D2** Describes the key features of the debug architecture.

**Chapter D3** Describes the Coprocessor Debug Register support (cp14) for the debug architecture.

xx *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Preface*

**Conventions**

This manual employs typographic and other conventions intended to improve its ease of use.

**General typographic conventions**

typewriter Is used for assembler syntax descriptions, pseudo-code descriptions of instructions,

and source code examples. In the cases of assembler syntax descriptions and pseudo-code descriptions, see the additional conventions below.

The typewriter font is also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudo-code descriptions of instructions and source code examples.

***italic*** Highlights important notes, introduces special terminology, and denotes internal

cross-references and citations.

**bold** Is used for emphasis in descriptive lists and elsewhere, where appropriate.

**SMALL CAPITALS** Are used for a few terms which have specific technical meanings. Their meanings

can be found in the *Glossary*.

**Pseudo-code descriptions of instructions**

A form of pseudo-code is used to provide precise descriptions of what instructions do. This pseudo-code is written in a typewriter font, and uses the following conventions for clarity and brevity:

• Indentation is used to indicate structure. For example, the range of statements that a for statement loops over, goes from the for statement to the next statement at the same or lower indentation level as the for statement (both ends exclusive).

• Comments are bracketed by /\* and \*/, as in the C language.

• English text is occasionally used outside comments to describe functionality that is hard to describe otherwise.

• All keywords and special functions used in the pseudo-code are described in the *Glossary*.

• Assignment and equality tests are distinguished by using = for an assignment and == for an equality test, as in the C language.

• Instruction fields are referred to by the names shown in the encoding diagram for the instruction. When an instruction field denotes a register, a reference to it means the value in that register, rather than the register number, unless the context demands otherwise. For example, a Rn == 0 test is checking whether the value in the specified register is 0, but a Rd is R15 test is checking whether the specified register is register 15.

• When an instruction uses an addressing mode, the pseudo-code for that addressing mode generates one or more values that are used in the pseudo-code for the instruction. For example, the AND instruction described in *AND* on page A4-8 uses ARM addressing mode 1 (see *Addressing Mode 1 - Data-processing operands* on page A5-2). The pseudo-code for the addressing mode generates two values shifter\_operand and shifter\_carry\_out, which are used by the pseudo-code for the AND instruction.

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* xxi

*Preface*

**Assembler syntax descriptions**

This manual contains numerous syntax descriptions for assembler instructions and for components of assembler instructions. These are shown in a typewriter font, and are as follows:

< > Any item bracketed by < and > is a short description of a type of value to be supplied by the

user in that position. A longer description of the item is normally supplied by subsequent text. Such items often correspond to a similarly named field in an encoding diagram for an instruction. When the correspondence simply requires the binary encoding of an integer value or register number to be substituted into the instruction encoding, it is not described explicitly. For example, if the assembler syntax for an ARM instruction contains an item <Rn> and the instruction encoding diagram contains a 4-bit field named Rn, the number of the register specified in the assembler syntax is encoded in binary in the instruction field.

If the correspondence between the assembler syntax item and the instruction encoding is more complex than simple binary encoding of an integer or register number, the item description indicates how it is encoded.

{ } Any item bracketed by { and } is optional. A description of the item and of how its presence

or absence is encoded in the instruction is normally supplied by subsequent text.

| This indicates an alternative character string. For example, LDM|STM is either LDM or STM.

**spaces** Single spaces are used for clarity, to separate items. When a space is obligatory in the

assembler syntax, two or more consecutive spaces are used.

+/- This indicates an optional + or - sign. If neither is coded, + is assumed.

\* When used in a combination like <immed\_8> \* 4, this describes an immediate value which

must be a specified multiple of a value taken from a numeric range. In this instance, the numeric range is 0 to 255 (the set of values that can be represented as an 8-bit immediate) and the specified multiple is 4, so the value described must be a multiple of 4 in the range 4\*0 = 0 to 4\*255 = 1020.

All other characters must be encoded precisely as they appear in the assembler syntax. Apart from { and }, the special characters described above do not appear in the basic forms of assembler instructions documented in this manual. The { and } characters need to be encoded in a few places as part of a variable item. When this happens, the long description of the variable item indicates how they must be used.

**Note** This manual only attempts to describe the most basic forms of assembler instruction syntax. In practice, assemblers normally recognize a much wider range of instruction syntaxes, as well as various directives to control the assembly process and additional features such as symbolic manipulation and macro expansion. All of these are beyond the scope of this manual.

xxii *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Preface*

**Further reading**

This section lists publications from both ARM Limited and third parties that provide additional information on the ARM family of processors.

ARM periodically provides updates and corrections to its documentation. See http://www.arm.com for current errata sheets and addenda, and the ARM Frequently Asked Questions.

**ARM publications**

ARM External Debug Interface Specification.

**External publications**

The following books are referred to in this manual, or provide additional information:

• *IEEE Standard for Shared-Data Formats Optimized for Scalable Coherent Interface (SCI) Processors*, IEEE Std 1596.5-1993, ISBN 1-55937-354-7, IEEE).

• *The JavaTM Virtual Machine Specification* Second Edition, Tim Lindholm and Frank Yellin, published by Addison Wesley (ISBN: 0-201-43294-3)

• JTAG Specification IEEE1149.1

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* xxiii

*Preface*

**Feedback**ARM Limited welcomes feedback on its documentation.

**Feedback on this book**

If you notice any errors or omissions in this book, send email to errata@arm giving:

• the document title

• the document number

• the page number(s) to which your comments apply

• a concise explanation of the problem.

General suggestions for additions and improvements are also welcome.

xxiv *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

Part A **CPU Architecture**

Chapter A1 **Introduction to the ARM Architecture**

This chapter introduces the ARM® architecture and contains the following sections:

• *About the ARM architecture* on page A1-2

• *ARM instruction set* on page A1-6

• *Thumb instruction set* on page A1-11.

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A1-1

*Introduction to the ARM Architecture*

**A1.1 About the ARM architecture**

The ARM architecture has evolved to a point where it supports implementations across a wide spectrum of performance points. Over two billion parts have shipped, establishing it as the dominant architecture across many market segments. The architectural simplicity of ARM processors has traditionally led to very small implementations, and small implementations allow devices with very low power consumption. Implementation size, performance, and very low power consumption remain key attributes in the development of the ARM architecture.

The ARM is a *Reduced Instruction Set Computer* (RISC), as it incorporates these typical RISC architecture features:

• a large uniform register file

• a *load/store* architecture, where data-processing operations only operate on register contents, not directly on memory contents

• simple addressing modes, with all load/store addresses being determined from register contents and instruction fields only

• uniform and fixed-length instruction fields, to simplify instruction decode.

In addition, the ARM architecture provides:

• control over both the *Arithmetic Logic Unit* (ALU) and shifter in most data-processing instructions to maximize the use of an ALU and a shifter

• auto-increment and auto-decrement addressing modes to optimize program loops

• Load and Store Multiple instructions to maximize data throughput

• conditional execution of almost all instructions to maximize execution throughput.

These enhancements to a basic RISC architecture allow ARM processors to achieve a good balance of high performance, small code size, low power consumption, and small silicon area.

A1-2 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Introduction to the ARM Architecture*

**A1.1.1 ARM registers**

ARM has 31 general-purpose 32-bit registers. At any one time, 16 of these registers are visible. The other registers are used to speed up exception processing. All the register specifiers in ARM instructions can address any of the 16 visible registers.

The main bank of 16 registers is used by all unprivileged code. These are the User mode registers. User mode is different from all other modes as it is unprivileged, which means:

• User mode can only switch to another processor mode by generating an exception. The SWI instruction provides this facility from program control.

• Memory systems and coprocessors might allow User mode less access to memory and coprocessor functionality than a privileged mode.

Three of the 16 visible registers have special roles:

**Stack pointer** Software normally uses R13 as a *Stack Pointer* (SP). R13 is used by the PUSH and POP

instructions in T variants, and by the SRS and RFE instructions from ARMv6.

**Link register** Register 14 is the *Link Register* (LR). This register holds the address of the next

instruction after a Branch and Link (BL or BLX) instruction, which is the instruction used to make a subroutine call. It is also used for return address information on entry to exception modes. At all other times, R14 can be used as a general-purpose register.

**Program counter** Register 15 is the *Program Counter* (PC). It can be used in most instructions as

a pointer to the instruction which is two instructions after the instruction being executed. In ARM state, all ARM instructions are four bytes long (one 32-bit word) and are always aligned on a word boundary. This means that the bottom two bits of the PC are always zero, and therefore the PC contains only 30 non-constant bits. Two other processor states are supported by some versions of the architecture. Thumb® state is supported on T variants, and Jazelle® state on J variants. The PC can be halfword (16-bit) and byte aligned respectively in these states.

The remaining 13 registers have no special hardware purpose. Their uses are defined purely by software. For more details on registers, refer to *Registers* on page A2-4.

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A1-3

*Introduction to the ARM Architecture*

**A1.1.2 Exceptions**

ARM supports seven types of exception, and a privileged processing mode for each type. The seven types of exception are:

• reset

• attempted execution of an Undefined instruction

• software interrupt (SWI) instructions, can be used to make a call to an operating system

• Prefetch Abort, an instruction fetch memory abort

• Data Abort, a data access memory abort

• IRQ, normal interrupt

• FIQ, fast interrupt.

When an exception occurs, some of the standard registers are replaced with registers specific to the exception mode. All exception modes have replacement *banked* registers for R13 and R14. The fast interrupt mode has additional banked registers for fast interrupt processing.

When an exception handler is entered, R14 holds the return address for exception processing. This is used to return after the exception is processed and to address the instruction that caused the exception.

Register 13 is banked across exception modes to provide each exception handler with a private stack pointer. The fast interrupt mode also banks registers 8 to 12 so that interrupt processing can begin without the need to save or restore these registers.

There is a sixth privileged processing mode, System mode, which uses the User mode registers. This is used to run tasks that require privileged access to memory and/or coprocessors, without limitations on which exceptions can occur during the task.

In addition to the above, reset shares the same privileged mode as SWIs.

For more details on exceptions, refer to *Exceptions* on page A2-16.

**The exception process**

When an exception occurs, the ARM processor halts execution in a defined manner and begins execution at one of a number of fixed addresses in memory, known as the *exception vectors*. There is a separate vector location for each exception, including reset. Behavior is defined for normal running systems (see section A2.6) and debug events (see Chapter D3 *Coprocessor 14, the Debug Coprocessor*)

An operating system installs a handler on every exception at initialization. Privileged operating system tasks are normally run in System mode to allow exceptions to occur within the operating system without state loss.

A1-4 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Introduction to the ARM Architecture*

**A1.1.3 Status registers**

All processor state other than the general-purpose register contents is held in *status registers*. The current operating processor status is in the *Current Program Status Register* (CPSR). The CPSR holds:

• four condition code flags (Negative, Zero, Carry and oVerflow).

• one sticky (Q) flag (ARMv5 and above only). This encodes whether saturation has occurred in saturated arithmetic instructions, or signed overflow in some specific multiply accumulate instructions.

• four GE (Greater than or Equal) flags (ARMv6 and above only). These encode the following conditions separately for each operation in parallel instructions: — whether the results of signed operations were non-negative — whether unsigned operations produced a carry or a borrow.

• two interrupt disable bits, one for each type of interrupt (two in ARMv5 and below).

• one (A) bit imprecise abort mask (from ARMv6)

• five bits that encode the current processor mode.

• two bits that encode whether ARM instructions, Thumb instructions, or Jazelle opcodes are being executed.

• one bit that controls the endianness of load and store operations (ARMv6 and above only).

Each exception mode also has a *Saved Program Status Register* (SPSR) which holds the CPSR of the task immediately before the exception occurred. The CPSR and the SPSRs are accessed with special instructions.

For more details on status registers, refer to *Program status registers* on page A2-11.

**Table A1-1 Status register summary**

**Field Description Architecture**

N Z C V Condition code flags All

J Jazelle state flag 5TEJ and above

GE[3:0] SIMD condition flags 6

E Endian Load/Store 6

A Imprecise Abort Mask 6

I IRQ Interrupt Mask All

F FIQ Interrupt Mask All

T Thumb state flag 4T and above

Mode[4:0] Processor mode All

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A1-5

*Introduction to the ARM Architecture*

**A1.2 ARM instruction set**

The ARM instruction set can be divided into six broad classes of instruction:

• *Branch instructions*

• *Data-processing instructions* on page A1-7

• *Status register transfer instructions* on page A1-8

• *Load and store instructions* on page A1-8

• *Coprocessor instructions* on page A1-10

• *Exception-generating instructions* on page A1-10.

Most data-processing instructions and one type of coprocessor instruction can update the four condition code flags in the CPSR (Negative, Zero, Carry and oVerflow) according to their result.

Almost all ARM instructions contain a 4-bit *condition* field. One value of this field specifies that the instruction is executed unconditionally.

Fourteen other values specify *conditional execution* of the instruction. 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. The 14 available conditions allow:

• tests for equality and non-equality

• tests for <, <=, >, and >= inequalities, in both signed and unsigned arithmetic

• each condition code flag to be tested individually.

The sixteenth value of the condition field encodes alternative instructions. These do not allow conditional execution. Before ARMv5 these instructions were UNPREDICTABLE.

**A1.2.1 Branch instructions**

As well as allowing many data-processing or load instructions to change control flow by writing the PC, a standard Branch instruction is provided with a 24-bit signed word offset, allowing forward and backward branches of up to 32MB.

There is a Branch and Link (BL) option that also preserves the address of the instruction after the branch in R14, the LR. This provides a subroutine call which can be returned from by copying the LR into the PC.

There are also branch instructions which can switch instruction set, so that execution continues at the branch target using the Thumb instruction set or Jazelle opcodes. Thumb support allows ARM code to call Thumb subroutines, and ARM subroutines to return to a Thumb caller. Similar instructions in the Thumb instruction set allow the corresponding Thumb → ARM switches. An overview of the Thumb instruction set is provided in Chapter A6 *The Thumb Instruction Set*.

The BXJ instruction introduced with the J variant of ARMv5, and present in ARMv6, provides the architected mechanism for entry to Jazelle state, and the associated assertion of the J flag in the CPSR.

A1-6 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Introduction to the ARM Architecture*

**A1.2.2 Data-processing instructions**

The data-processing instructions perform calculations on the general-purpose registers. There are five types of data-processing instructions:

• *Arithmetic/logic instructions*

• *Comparison instructions*

• *Single Instruction Multiple Data (SIMD) instructions*

• *Multiply instructions* on page A1-8

• *Miscellaneous Data Processing instructions* on page A1-8.

**Arithmetic/logic instructions**

The following arithmetic/logic instructions share a common instruction format. These perform an arithmetic or logical operation on up to two source operands, and write the result to a destination register. They can also optionally update the condition code flags, based on the result.

Of the two source operands:

• one is always a register

• the other has two basic forms: — an immediate value — a register value, optionally shifted.

If the operand is a shifted register, the shift amount can be either an immediate value or the value of another register. Five types of shift can be specified. Every arithmetic/logic instruction can therefore perform an arithmetic/logic operation and a shift operation. As a result, ARM does not have dedicated shift instructions.

The *Program Counter* (PC) is a general-purpose register, and therefore arithmetic/logic instructions can write their results directly to the PC. This allows easy implementation of a variety of jump instructions.

**Comparison instructions**

The comparison instructions use the same instruction format as the arithmetic/logic instructions. These perform an arithmetic or logical operation on two source operands, but do not write the result to a register. They always update the condition flags, based on the result.

The source operands of comparison instructions take the same forms as those of arithmetic/logic instructions, including the ability to incorporate a shift operation.

**Single Instruction Multiple Data (SIMD) instructions**

The add and subtract instructions treat each operand as two parallel 16-bit numbers, or four parallel 8-bit numbers. They can be treated as signed or unsigned. The operations can optionally be saturating, wrap around, or the results can be halved to avoid overflow.

These instructions are available in ARMv6.

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A1-7

*Introduction to the ARM Architecture*

**Multiply instructions**

There are several classes of multiply instructions, introduced at different times into the architecture. See *Multiply instructions* on page A3-10 for details.

**Miscellaneous Data Processing instructions**

These include Count Leading Zeros (CLZ) and Unsigned Sum of Absolute Differences with optional Accumulate (USAD8 and USADA8).

**A1.2.3 Status register transfer instructions**

The status register transfer instructions transfer the contents of the CPSR or an SPSR to or from a general-purpose register. Writing to the CPSR can:

• set the values of the condition code flags

• set the values of the interrupt enable bits

• set the processor mode and state

• alter the endianness of Load and Store operations.

**A1.2.4 Load and store instructions**

The following load and store instructions are available:

• *Load and Store Register*

• *Load and Store Multiple registers* on page A1-9

• *Load and Store Register Exclusive* on page A1-9.

There are also swap and swap byte instructions, but their use is deprecated in ARMv6. It is recommended that all software migrates to using the load and store register exclusive instructions.

**Load and Store Register**

Load Register instructions can load a 64-bit doubleword, a 32-bit word, a 16-bit halfword, or an 8-bit byte from memory into a register or registers. Byte and halfword loads can be automatically zero-extended or sign-extended as they are loaded.

Store Register instructions can store a 64-bit doubleword, a 32-bit word, a 16-bit halfword, or an 8-bit byte from a register or registers to memory.

From ARMv6, unaligned loads and stores of words and halfwords are supported, accessing the specified byte addresses. Prior to ARMv6, unaligned 32-bit loads rotated data, all 32-bit stores were aligned, and the other affected instructions UNPREDICTABLE.

A1-8 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Introduction to the ARM Architecture*

Load and Store Register instructions have three primary addressing modes, all of which use a *base register* and an *offset* specified by the instruction:

• In *offset addressing*, the memory address is formed by adding or subtracting an offset to or from the base register value.

• In *pre-indexed addressing*, the memory address is formed in the same way as for offset addressing. As a side effect, the memory address is also written back to the base register.

• In *post-indexed addressing*, the memory address is the base register value. As a side effect, an offset is added to or subtracted from the base register value and the result is written back to the base register.

In each case, the offset can be either an immediate or the value of an *index register*. Register-based offsets can also be scaled with shift operations.

As the PC is a general-purpose register, a 32-bit value can be loaded directly into the PC to perform a jump to any address in the 4GB memory space.

**Load and Store Multiple registers**

Load Multiple (LDM) and Store Multiple (STM) instructions perform a block transfer of any number of the general-purpose registers to or from memory. Four addressing modes are provided:

• pre-increment

• post-increment

• pre-decrement

• post-decrement.

The base address is specified by a register value, which can be optionally updated after the transfer. As the subroutine return address and PC values are in general-purpose registers, very efficient subroutine entry and exit sequences can be constructed with LDM and STM:

• A single STM instruction at subroutine entry can push register contents and the return address onto the stack, updating the stack pointer in the process.

• A single LDM instruction at subroutine exit can restore register contents from the stack, load the PC with the return address, and update the stack pointer.

LDM and STM instructions also allow very efficient code for block copies and similar data movement algorithms.

**Load and Store Register Exclusive**

These instructions support cooperative memory synchronization. They are designed to provide the atomic behavior required for semaphores without locking all system resources between the load and store phases. See *LDREX* on page A4-52 and *STREX* on page A4-202 for details.

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A1-9

*Introduction to the ARM Architecture*

**A1.2.5 Coprocessor instructions**

There are three types of coprocessor instructions:

**Data-processing instructions**

These start a coprocessor-specific internal operation.

**Data transfer instructions**

These transfer coprocessor data to or from memory. The address of the transfer is calculated by the ARM processor.

**Register transfer instructions**

These allow a coprocessor value to be transferred to or from an ARM register, or a pair of ARM registers.

**A1.2.6 Exception-generating instructions**

Two types of instruction are designed to cause specific exceptions to occur.

**Software interrupt instructions**

SWI instructions cause a software interrupt exception to occur. These are normally used to make calls to an operating system, to request an OS-defined service. The exception entry caused by a SWI instruction also changes to a privileged processor mode. This allows an unprivileged task to gain access to privileged functions, but only in ways permitted by the OS.

**Software breakpoint instructions**

BKPT instructions cause an abort exception to occur. If suitable debugger software is installed on the abort vector, an abort exception generated in this fashion is treated as a breakpoint. If debug hardware is present in the system, it can instead treat a BKPT instruction directly as a breakpoint, preventing the abort exception from occurring.

In addition to the above, the following types of instruction cause an Undefined Instruction exception to occur:

• coprocessor instructions which are not recognized by any hardware coprocessor

• most instruction words that have not yet been allocated a meaning as an ARM instruction.

In each case, this exception is normally used either to generate a suitable error or to initiate software emulation of the instruction.

A1-10 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Introduction to the ARM Architecture*

**A1.3 Thumb instruction set**

The Thumb instruction set is a subset of the ARM instruction set, with each instruction encoded in 16 bits instead of 32 bits. For details see Chapter A6 *The Thumb Instruction Set*.

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A1-11

*Introduction to the ARM Architecture*

A1-12 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

Chapter A2 **Programmers’ Model**

This chapter introduces the ARM® Programmers’ Model. It contains the following sections:

• *Data types* on page A2-2

• *Processor modes* on page A2-3

• *Registers* on page A2-4

• *General-purpose registers* on page A2-6

• *Program status registers* on page A2-11

• *Exceptions* on page A2-16

• *Endian support* on page A2-30

• *Unaligned access support* on page A2-38

• *Synchronization primitives* on page A2-44

• *The Jazelle Extension* on page A2-53

• *Saturated integer arithmetic* on page A2-69.

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A2-1

*Programmers’ Model*

**A2.1 Data types**

ARM processors support the following data types:

**Byte** 8 bits

**Halfword** 16 bits

**Word** 32 bits

**Note**

• Support for halfwords was introduced in version 4.

• ARMv6 has introduced unaligned data support for words and halfwords. See *Unaligned access support* on page A2-38 for more information.

• When any of these types is described as *unsigned*, the N-bit data value represents a non-negative integer in the range 0 to +2N-1, using normal binary format.

• When any of these types is described as *signed*, the N-bit data value represents an integer in the range -2N-1 to +2N-1-1, using two's complement format.

• Most data operations, for example ADD, are performed on word quantities. Long multiplies support 64-bit results with or without accumulation. ARMv5TE introduced some halfword multiply operations. ARMv6 introduced a variety of Single Instruction Multiple Data (SIMD) instructions operating on two halfwords or four bytes in parallel.

• Load and store operations can transfer bytes, halfwords, or words to and from memory, automatically zero-extending or sign-extending bytes or halfwords as they are loaded. Load and store operations that transfer two or more words to and from memory are also provided.

• ARM instructions are exactly one word and are aligned on a four-byte boundary. Thumb® instructions are exactly one halfword and are aligned on a two-byte boundary. Jazelle® opcodes are a variable number of bytes in length and can appear at any byte alignment.

A2-2 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Programmers’ Model*

**A2.2 Processor modes**

The ARM architecture supports the seven processor modes shown in Table A2-1.

**Table A2-1 ARM processor modes**

**Processor mode Mode number Description**

User usr 0b10000 Normal program execution mode

FIQ fiq 0b10001 Supports a high-speed data transfer or channel process

IRQ irq 0b10010 Used for general-purpose interrupt handling

Supervisor svc 0b10011 A protected mode for the operating system

Abort abt 0b10111 Implements virtual memory and/or memory protection

Undefined und 0b11011 Supports software emulation of hardware coprocessors

System sys 0b11111 Runs privileged operating system tasks (ARMv4 and

above)

Mode changes can be made under software control, or can be caused by external interrupts or exception processing.

Most application programs execute in User mode. When the processor is in User mode, the program being executed is unable to access some protected system resources or to change mode, other than by causing an exception to occur (see *Exceptions* on page A2-16). This allows a suitably-written operating system to control the use of system resources.

The modes other than User mode are known as *privileged modes*. They have full access to system resources and can change mode freely. Five of them are known as *exception modes*:

• FIQ

• IRQ

• Supervisor

• Abort

• Undefined.

These are entered when specific exceptions occur. Each of them has some additional registers to avoid corrupting User mode state when the exception occurs (see *Registers* on page A2-4 for details).

The remaining mode is System mode, which is not entered by any exception and has exactly the same registers available as User mode. However, it is a privileged mode and is therefore not subject to the User mode restrictions. It is intended for use by operating system tasks that need access to system resources, but wish to avoid using the additional registers associated with the exception modes. Avoiding such use ensures that the task state is not corrupted by the occurrence of any exception.

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A2-3

*Programmers’ Model*

**A2.3 Registers**

The ARM processor has a total of 37 registers:

• Thirty-one general-purpose registers, including a program counter. These registers are 32 bits wide and are described in *General-purpose registers* on page A2-6.

• Six status registers. These registers are also 32 bits wide, but only some of the 32 bits are allocated or need to be implemented. The subset depends on the architecture variant supported. These are described in *Program status registers* on page A2-11.

Registers are arranged in partially overlapping banks, with the current processor mode controlling which bank is available, as shown in Figure A2-1 on page A2-5. At any time, 15 general-purpose registers (R0 to R14), one or two status registers, and the program counter are visible. Each column of Figure A2-1 on page A2-5 shows which general-purpose and status registers are visible in the indicated processor mode.

A2-4 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Programmers’ Model*

**Modes**

**Privileged modes**

**Exception modes**

**User**

**System**

**Supervisor**

**Supervisor**

**Abort**

**Abort**

**Abort**

**Undefined**

**Undefined**

**Undefined**

**Undefined**

**Interrupt**

**Interrupt**

**Interrupt**

**Interrupt**

**Interrupt**

**Fast interrupt**

**Fast interrupt**

**Fast interrupt**

**Fast interrupt**

**Fast interrupt**

**Fast interrupt**

R0R0R0R0R0R0R0R1R1R1R1R1R1R1R2R2R2R2R2R2R2R3R3R3R3R3R3R3R4R4R4R4R4R4R4R5R5R5R5R5R5R5R6R6R6R6R6R6R6R7R7R7R7R7R7R7

R8R8R8R8R8R8R8\_fiq

R9R9R9R9R9R9R9\_fiq

R10

R10

R10

R10

R10

R10

R10

R10

R10

R10

R10

R10

R10

R10

R10

R10

R10\_fiq

R10\_fiq

R10\_fiq

R10\_fiq

R10\_fiq

R10\_fiq

R11

R11

R11

R11

R11

R11

R11

R11

R11

R11

R11

R11

R11

R11

R11

R11

R11\_fiq

R11\_fiq

R11\_fiq

R11\_fiq

R11\_fiq

R11\_fiq

R12

R12

R12

R12

R12

R12

R12

R12

R12

R12

R12

R12

R12

R12

R12

R12

R12\_fiq

R12\_fiq

R12\_fiq

R12\_fiq

R12\_fiq

R12\_fiq

R13

R13

R13\_svc

R13\_svc

R13\_abt

R13\_abt

R13\_abt

R13\_und

R13\_und

R13\_und

R13\_und

R13\_irq

R13\_irq

R13\_irq

R13\_irq

R13\_irq

R13\_fiq

R13\_fiq

R13\_fiq

R13\_fiq

R13\_fiq

R13\_fiq

R14

R14

R14\_svc

R14\_svc

R14\_abt

R14\_abt

R14\_abt

R14\_und

R14\_und

R14\_und

R14\_und

R14\_irq

R14\_irq

R14\_irq

R14\_irq

R14\_irq

R14\_fiq

R14\_fiq

R14\_fiq

R14\_fiq

R14\_fiq

R14\_fiq

PC

PC

PC

PC

PC

PC

PC

PC

PC

PC

PC

PC

PC

PC

PC

PC

PC

PC

PC

PC

PC

PC

CPSR

CPSR

CPSR

CPSR

CPSR

CPSR

CPSR

CPSR

CPSR

CPSR

CPSR

CPSR

CPSR

CPSR

CPSR

CPSR

CPSR

CPSR

CPSR

CPSR

CPSR

CPSR

SPSR\_svc

SPSR\_abt

SPSR\_und

SPSR\_und

SPSR\_irq

SPSR\_irq

SPSR\_irq

SPSR\_fiq

SPSR\_fiq

SPSR\_fiq

SPSR\_fiq

*indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode*

**Figure A2-1 Register organization**

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A2-5

*Programmers’ Model*

**A2.4 General-purpose registers**

The general-purpose registers R0 to R15 can be split into three groups. These groups differ in the way they are banked and in their special-purpose uses:

• *The unbanked registers, R0 to R7*

• *The banked registers, R8 to R14*

• Register 15, the PC, is described in *Register 15 and the program counter* on page A2-9.

**A2.4.1 The unbanked registers, R0 to R7**

Registers R0 to R7 are *unbanked registers*. This means that each of them refers to the same 32-bit physical register in all processor modes. They are completely general-purpose registers, with no special uses implied by the architecture, and can be used wherever an instruction allows a general-purpose register to be specified.

**A2.4.2 The banked registers, R8 to R14**

Registers R8 to R14 are *banked registers*. The physical register referred to by each of them depends on the current processor mode. Where a particular physical register is intended, without depending on the current processor mode, a more specific name (as described below) is used. Almost all instructions allow the banked registers to be used wherever a general-purpose register is allowed.

**Note** There are a few exceptions to this rule for processors pre-ARMv6, and they are noted in the individual instruction descriptions. Where a restriction exists on the use of banked registers, it always applies to all of R8 to R14. For example, R8 to R12 are subject to such restrictions even in systems in which FIQ mode is never used and so only one physical version of the register is ever in use.

Registers R8 to R12 have two banked physical registers each. One is used in all processor modes other than FIQ mode, and the other is used in FIQ mode. Where it is necessary to be specific about which version is being referred to, the first group of physical registers are referred to as R8\_usr to R12\_usr and the second group as R8\_fiq to R12\_fiq.

Registers R8 to R12 do not have any dedicated special purposes in the architecture. However, for interrupts that are simple enough to be processed using registers R8 to R14 only, the existence of separate FIQ mode versions of these registers allows very fast interrupt processing.

Registers R13 and R14 have six banked physical registers each. One is used in User and System modes, and each of the remaining five is used in one of the five exception modes. Where it is necessary to be specific about which version is being referred to, you use names of the form:

R13\_<mode> R14\_<mode>

where <mode> is the appropriate one of usr, svc (for Supervisor mode), abt, und, irq and fiq.

A2-6 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Programmers’ Model*

Register R13 is normally used as a stack pointer and is also known as the SP. The SRS instruction, introduced in ARMv6, is the only ARM instruction that uses R13 in a special-case manner. There are other such instructions in the Thumb instruction set, as described in Chapter A6 *The Thumb Instruction Set*.

Each exception mode has its own banked version of R13. Suitable uses for these banked versions of R13 depend on the architecture version:

• In architecture versions earlier than ARMv6, each banked version of R13 will normally be initialized to point to a stack dedicated to that exception mode. On entry, the exception handler typically stores the values of other registers that it wants to use on this stack. By reloading these values into the register when it returns, the exception handler can ensure that it does not corrupt the state of the program that was being executed when the exception occurred.

If fewer exception-handling stacks are desired in a system than this implies, it is possible instead to initialize the banked version of R13 for an exception mode to point to a small area of memory that is used for temporary storage while transferring to another exception mode and its stack. For example, suppose that there is a requirement for an IRQ handler to use the Supervisor mode stack to store SPSR\_irq, R0 to R3, R12, R14\_irq, and then to execute in Supervisor mode with IRQs enabled. This can be achieved by initializing R13\_irq to point to a four-word temporary storage area, and using the following code sequence on entry to the handler:

STMIA R13, (R0-R3) ; Put R0-R3 into temporary storage MRS R0, SPSR ; Move banked SPSR and R12-R14 into MOV R1, R12 ; unbanked registers MOV R2, R13 MOV R3, R14 MRS R12, CPSR ; Use read/modify/write sequence BIC R12, R12, #0x1F ; on CPSR to switch to Supervisor ORR R12, R12, #0x13 ; mode MSR CPSR\_c, R12 STMFD R13!, (R1,R3) ; Push original {R12, R14\_irq}, then STR R0, [R13,#-20]! ; SPSR\_irq with a gap for R0-R3 LDMIA R2, {R0-R3} ; Reload R0-R3 from temporary storage BIC R12, R12, #0x80 ; Modify and write CPSR again to MSR CPSR\_c, R12 ; re-enable IRQs STMIB R13, {R0-R3} ; Store R0-R3 in the gap left on the

; stack for them

• In ARMv6 and above, it is recommended that the OS designer should decide how many exception-handling stacks are required in the system, and select a suitable processor mode in which to handle the exceptions that use each stack. For example, one exception-handling stack might be required to be locked into real memory and be used for aborts and high-priority interrupts, while another could use virtual memory and be used for SWIs, Undefined instructions and low-priority interrupts. Suitable processor modes in this example might be Abort mode and Supervisor mode respectively.

The banked version of R13 for each of the selected modes is then initialized to point to the corresponding stack, and the other banked versions of R13 are normally not used. Each exception handler starts with an SRS instruction to store the exception return information to the appropriate stack, followed (if necessary) by a CPS instruction to switch to the appropriate mode and possibly

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A2-7

*Programmers’ Model*

re-enable interrupts, after which other registers can be saved on that stack. So in the above example, an Undefined Instruction handler that wants to re-enable interrupts immediately would start with the following two instructions:

SRSFD #svc\_mode! CPSIE i, #svc\_mode The handler can then operate entirely in Supervisor mode, using the virtual memory stack pointed to by R13\_svc.

Register R14 (also known as the *Link Register* or LR) has two special functions in the architecture:

• In each mode, the mode's own version of R14 is used to hold subroutine return addresses. When a subroutine call is performed by a BL or BLX instruction, R14 is set to the subroutine return address. The subroutine return is performed by copying R14 back to the program counter. This is typically done in one of the two following ways:

— Execute a BX LR instruction.

**Note** An MOV PC,LR instruction will perform the same function as BX LR if the code to which it returns uses the current instruction set, but will not return correctly from an ARM subroutine called by Thumb code, or from a Thumb subroutine called by ARM code. The use of MOV PC,LR instructions for subroutine return is therefore deprecated.

— On subroutine entry, store R14 to the stack with an instruction of the form:

STMFD SP!,{<registers>,LR} and use a matching instruction to return: LDMFD SP!,{<registers>,PC}

• When an exception occurs, the appropriate exception mode's version of R14 is set to the exception return address (offset by a small constant for some exceptions). The exception return is performed in a similar way to a subroutine return, but using slightly different instructions to ensure full restoration of the state of the program that was being executed when the exception occurred. See *Exceptions* on page A2-16 for more details.

Register R14 can be treated as a general-purpose register at all other times.

**Note** When nested exceptions are possible, the two special-purpose uses might conflict. For example, if an IRQ interrupt occurs when a program is being executed in User mode, none of the User mode registers are necessarily corrupted. But if an interrupt handler running in IRQ mode re-enables IRQ interrupts and a nested IRQ interrupt occurs, any value the outer interrupt handler is holding in R14\_irq at the time is overwritten by the return address of the nested interrupt.

System programmers need to be careful about such interactions. The usual way to deal with them is to ensure that the appropriate version of R14 does not hold anything significant at times when nested exceptions can occur. When this is hard to do in a straightforward way, it is usually best to change to another

A2-8 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Programmers’ Model*

processor mode during entry to the exception handler, before re-enabling interrupts or otherwise allowing nested exceptions to occur. (In ARMv4 and above, System mode is often the best mode to use for this purpose.)

**A2.4.3 Register 15 and the program counter**

Register R15 (R15) is often used in place of the other general-purpose registers to produce various special-case effects. These are instruction-specific and so are described in the individual instruction descriptions.

There are also many instruction-specific restrictions on the use of R15. these are also noted in the individual instruction descriptions. Usually, the instruction is UNPREDICTABLE if R15 is used in a manner that breaks these restrictions.

If an instruction description neither describes a special-case effect when R15 is used nor places restrictions on its use, R15 is used to read or write the *Program Counter* (PC), as described in:

• *Reading the program counter*

• *Writing the program counter* on page A2-10.

**Reading the program counter**

When an instruction reads the PC, the value read depends on which instruction set it comes from:

• For an ARM instruction, the value read is the address of the instruction plus 8 bytes. Bits [1:0] of this value are always zero, because ARM instructions are always word-aligned.

• For a Thumb instruction, the value read is the address of the instruction plus 4 bytes. Bit [0] of this value is always zero, because Thumb instructions are always halfword-aligned.

This way of reading the PC is primarily used for quick, position-independent addressing of nearby instructions and data, including position-independent branching within a program.

An exception to the above rule occurs when an ARM STR or STM instruction stores R15. Such instructions can store either the address of the instruction plus 8 bytes, like other instructions that read R15, or the address of the instruction plus 12 bytes. Whether the offset of 8 or the offset of 12 is used is IMPLEMENTATION DEFINED. An implementation must use the same offset for all ARM STR and STM instructions that store R15. It cannot use 8 for some of them and 12 for others.

Because of this exception, it is usually best to avoid the use of STR and STM instructions that store R15. If this is difficult, use a suitable instruction sequence in the program to ascertain which offset the implementation uses. For example, if R0 points to an available word of memory, then the following instructions put the offset of the implementation in R0:

SUB R1, PC, #4 ; R1 = address of following STR instruction STR PC, [R0] ; Store address of STR instruction + offset, LDR R0, [R0] ; then reload it SUB R0, R0, R1 ; Calculate the offset as the difference

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A2-9

*Programmers’ Model*

**Note** The rules about how R15 is read apply only to reads by instructions. In particular, they do not necessarily describe the values placed on a hardware address bus during instruction fetches. Like all other details of hardware interfaces, such values are IMPLEMENTATION DEFINED.

**Writing the program counter**

When an instruction writes the PC, the normal result is that the value written to the PC is treated as an instruction address and a branch occurs to that address.

Since ARM instructions are required to be word-aligned, values they write to the PC are normally expected to have bits[1:0] == 0b00. Similarly, Thumb instructions are required to be halfword-aligned and so values they write to the PC are normally expected to have bit[0] == 0.

The precise rules depend on the current instruction set state and the architecture version:

• In T variants of ARMv4 and above, including all variants of ARMv6 and above, bit[0] of a value written to R15 in Thumb state is ignored unless the instruction description says otherwise. If bit[0] of the PC is implemented (which depends on whether and how the Jazelle Extension is implemented), then zero must be written to it regardless of the value written to bit[0] of R15.

• In ARMv6 and above, bits[1:0] of a value written to R15 in ARM state are ignored unless the instruction description says otherwise. Bit[1] of the PC must be written as zero regardless of the value written to bit[1] of R15. If bit[0] of the PC is implemented (which depends on how the Jazelle Extension is implemented), then zero must be written to it.

• In all variants of ARMv4 and ARMv5, bits[1:0] of a value written to R15 in ARM state must be 0b00. If they are not, the results are UNPREDICTABLE.

Several instructions have their own rules for interpreting values written to R15. For example, BX and other instructions designed to transfer between ARM and Thumb states use bit[0] of the value to select whether to execute the code at the destination address in ARM state or Thumb state. Special rules of this type are described on the individual instruction pages, and override the general rules in this section.

A2-10 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Programmers’ Model*

**A2.5 Program status registers**

The *Current Program Status Register* (CPSR) is accessible in all processor modes. It contains condition code flags, interrupt disable bits, the current processor mode, and other status and control information. Each exception mode also has a *Saved Program Status Register* (SPSR), that is used to preserve the value of the CPSR when the associated exception occurs.

**Note** User mode and System mode do not have an SPSR, because they are not exception modes. All instructions that read or write the SPSR are UNPREDICTABLE when executed in User mode or System mode.

The format of the CPSR and the SPSRs is shown below.

31 30 29 28 27 26 25 24 23 20 19 16 15 10 9 8 7 6 5 4 0

N Z C V Q Res J RESERVED GE[3:0] RESERVED E A I F T M[4:0]

**A2.5.1 Types of PSR bits**

PSR bits fall into four categories, depending on the way in which they can be updated:

**Reserved bits** Reserved for future expansion. Implementations must read these bits as 0 and ignore

writes to them. For maximum compatibility with future extensions to the architecture, they must be written with values read from the same bits.

**User-writable bits** Can be written from any mode. The N, Z, C, V, Q, GE[3:0], and E bits are

user-writable.

**Privileged bits** Can be written from any privileged mode. Writes to privileged bits in User mode are

ignored. The A, I, F, and M[4:0] bits are privileged.

**Execution state bits** Can be written from any privileged mode. Writes to execution state bits in User

mode are ignored. The J and T bits are execution state bits, and are always zero in ARM state. Privileged MSR instructions that write to the CPSR execution state bits must write zeros to them, in order to avoid changing them. If ones are written to either or both of them, the resulting behavior is UNPREDICTABLE. This restriction applies only to the CPSR execution state bits, not the SPSR execution state bits.

**A2.5.2 The condition code flags**

The N, Z, C, and V (Negative, Zero, Carry and oVerflow) bits are collectively known as the *condition code flags*, often referred to as *flags*. The condition code flags in the CPSR can be tested by most instructions to determine whether the instruction is to be executed.

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A2-11

*Programmers’ Model*

The condition code flags are usually modified by:

• Execution of a comparison instruction (CMN, CMP, TEQ or TST).

• Execution of some other arithmetic, logical or move instruction, where the destination register of the instruction is not R15. Most of these instructions have both a flag-preserving and a flag-setting variant, with the latter being selected by adding an S qualifier to the instruction mnemonic. Some of these instructions only have a flag-preserving version. This is noted in the individual instruction descriptions.

In either case, the new condition code flags (after the instruction has been executed) usually mean:

**N** Is set to bit 31 of the result of the instruction. If this result is regarded as a two's complement

signed integer, then N = 1 if the result is negative and N = 0 if it is positive or zero.

**Z** Is set to 1 if the result of the instruction is zero (this often indicates an *equal* result from a

comparison), and to 0 otherwise.

**C** Is set in one of four ways:

• For an addition, including the comparison instruction CMN, C is set to 1 if the addition produced a carry (that is, an unsigned overflow), and to 0 otherwise.

• For a subtraction, including the comparison instruction CMP, C is set to 0 if the subtraction produced a borrow (that is, an unsigned underflow), and to 1 otherwise.

• For non-addition/subtractions that incorporate a shift operation, C is set to the last bit shifted out of the value by the shifter.

• For other non-addition/subtractions, C is normally left unchanged (but see the individual instruction descriptions for any special cases).

**V** Is set in one of two ways:

• For an addition or subtraction, V is set to 1 if signed overflow occurred, regarding the operands and result as two's complement signed integers.

• For non-addition/subtractions, V is normally left unchanged (but see the individual instruction descriptions for any special cases).

The flags can be modified in these additional ways:

• Execution of an MSR instruction, as part of its function of writing a new value to the CPSR or SPSR.

• Execution of MRC instructions with destination register R15. The purpose of such instructions is to transfer coprocessor-generated condition code flag values to the ARM processor.

• Execution of some variants of the LDM instruction. These variants copy the SPSR to the CPSR, and their main intended use is for returning from exceptions.

• Execution of an RFE instruction in a privileged mode that loads a new value into the CPSR from memory.

• Execution of flag-setting variants of arithmetic and logical instructions whose destination register is R15. These also copy the SPSR to the CPSR, and are intended for returning from exceptions.

A2-12 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Programmers’ Model*

**A2.5.3 The Q flag**

In E variants of ARMv5 and above, bit[27] of the CPSR is known as the Q flag and is used to indicate whether overflow and/or saturation has occurred in some DSP-oriented instructions. Similarly, bit[27] of each SPSR is a Q flag, and is used to preserve and restore the CPSR Q flag if an exception occurs. See *Saturated integer arithmetic* on page A2-69 for more information.

In architecture versions prior to ARMv5, and in non-E variants of ARMv5, bit[27] of the CPSR and SPSRs must be treated as a reserved bit, as described in *Types of PSR bits* on page A2-11.

**A2.5.4 The GE[3:0] bits**

In ARMv6, the SIMD instructions use bits[19:16] as *Greater than or Equal* (GE) flags for individual bytes or halfwords of the result. You can use these flags to control a later SEL instruction, see *SEL* on page A4-127 for more details.

Instructions that operate on halfwords:

• set or clear GE[3:2] together, based on the result of the top halfword calculation

• set or clear GE[1:0] together, based on the result of the bottom halfword calculation.

Instructions that operate on bytes:

• set or clear GE[3] according to the result of the top byte calculation

• set or clear GE[2] according to the result of the second byte calculation

• set or clear GE[1] according to the result of the third byte calculation

• set or clear GE[0] according to the result of the bottom byte calculation.

Each bit is set (otherwise cleared) if the results of the corresponding calculation are as follows:

• for unsigned byte addition, if the result is greater than or equal to 28

• for unsigned halfword addition, if the result is greater than or equal to 216

• for unsigned subtraction, if the result is greater than or equal to zero

• for signed arithmetic, if the result is greater than or equal to zero.

In architecture versions prior to ARMv6, bits[19:16] of the CPSR and SPSRs must be treated as a reserved bit, as described in *Types of PSR bits* on page A2-11.

**A2.5.5 The E bit**

From ARMv6, bit[9] controls load and store endianness for data handling. See *Instructions to change CPSR E bit* on page A2-36. This bit is ignored by instruction fetches.

In architecture versions prior to ARMv6, bit[9] of the CPSR and SPSRs must be treated as a reserved bit, as described in *Types of PSR bits* on page A2-11.

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A2-13

*Programmers’ Model*

**A2.5.6 The interrupt disable bits**

A, I, and F are the interrupt disable bits:

**A bit** Disables imprecise data aborts when it is set. This is available only in ARMv6 and above.

In earlier versions, bit[8] of CPSR and SPSRs must be treated as a reserved bit, as described in *Types of PSR bits* on page A2-11.

**I bit** Disables IRQ interrupts when it is set.

**F bit** Disables FIQ interrupts when it is set.

**A2.5.7 The mode bits**

M[4:0] are the mode bits. These determine the mode in which the processor operates. Their interpretation is shown in Table A2-2.

Not all combinations of the mode bits define a valid processor mode. Only those combinations explicitly described can be used. If any other value is programmed into the mode bits M[4:0], the result is UNPREDICTABLE.

A2-14 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I **Table A2-2 The mode bits**

**M[4:0] Mode Accessible registers**

0b10000 User PC, R14 to R0, CPSR

0b10001 FIQ PC, R14\_fiq to R8\_fiq, R7 to R0, CPSR, SPSR\_fiq

0b10010 IRQ PC, R14\_irq, R13\_irq, R12 to R0, CPSR, SPSR\_irq

0b10011 Supervisor PC, R14\_svc, R13\_svc, R12 to R0, CPSR, SPSR\_svc

0b10111 Abort PC, R14\_abt, R13\_abt, R12 to R0, CPSR, SPSR\_abt

0b11011 Undefined PC, R14\_und, R13\_und, R12 to R0, CPSR, SPSR\_und

0b11111 System PC, R14 to R0, CPSR (ARMv4 and above)

*Programmers’ Model*

**A2.5.8 The T and J bits**

The T and J bits select the current instruction set, as shown in Table A2-3.

**Table A2-3 The T and J bits**

**J T Instruction set**

0 0 ARM

0 1 Thumb

1 0 Jazelle

1 1 RESERVED

The T bit exists on t variants of ARMv4, and on all variants of ARMv5 and above. on non-T variants of ARMv4, the T bit must be treated as a reserved bit, as described in *Types of PSR bits* on page A2-11.

The Thumb instruction set is implemented on T variants of ARMv4 and ARMv5, and on all variants of ARMv6 and above. instructions that switch between ARM and Thumb state execution can be used freely on implementation of these architectures.

The Thumb instruction set is not implemented on non-T variants of ARMv5. If the Thumb instruction set is selected by setting T ==1 on these architecture variants, the next instruction executed will cause an Undefined Instruction exception (see *Undefined Instruction exception* on page A2-19). Instructions that switch between ARM and Thumb state execution can be used on implementation of these architecture variants, but only function correctly as long as the program remains in ARM state. If the program attempts to switch to Thumb state, the first instruction executed after that switch causes an Undefined Instruction exception. Entry into that exception then switches back to ARM state. The exception handler can detect that this was the cause of the exception from the fact that the T bit of SPSR\_und is set.

The J bit exists on ARMv5TEJ and on all variants of ARMv6 and above. On variants of ARMv4 and ARMv5, other than ARMv5TEJ, the J bit must be treated as a reserved bit, as described in *Types of PSR bits* on page A2-11.

Hardware acceleration for Jazelle opcode execution can be implemented on ARMv5TEJ and on ARMv6 and above. On these architecture variants, the BXJ instruction is used to switch from ARM state into Jazelle state when the hardware accelerator is present and enabled. If the hardware accelerator is disabled, or not present, the BXJ instruction behaves as a BX instruction, and the J bit remains clear. For more details, see *The Jazelle Extension* on page A2-53.

**A2.5.9 Other bits**

Other bits in the program status registers are reserved for future expansion. In general, programmers must take care to write code in such a way that these bits are never modified. Failure to do this might result in code that has unexpected side effects on future versions of the architecture. See *Types of PSR bits* on page A2-11, and the usage notes for the MSR instruction on page A4-76 for more details.

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A2-15

*Programmers’ Model*

**A2.6 Exceptions**

Exceptions are generated by internal and external sources to cause the processor to handle an event, such as an externally generated interrupt or an attempt to execute an Undefined instruction. The processor state just before handling the exception is normally preserved so that the original program can be resumed when the exception routine has completed. More than one exception can arise at the same time.

The ARM architecture supports seven types of exception. Table A2-4 lists the types of exception and the processor mode that is used to process each type. When an exception occurs, execution is forced from a fixed memory address corresponding to the type of exception. These fixed addresses are called the *exception vectors*.

**Note** The normal vector at address 0x00000014 and the high vector at address 0xFFFF0014 are reserved for future expansion.

**Table A2-4 Exception processing modes**

**Exception type Mode VEa Normal**

**address**

A2-16 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

**High vector address**

Reset Supervisor 0x00000000 0xFFFF0000

Undefined instructions Undefined 0x00000004 0xFFFF0004

Software interrupt (SWI) Supervisor 0x00000008 0xFFFF0008

Prefetch Abort (instruction fetch memory abort) Abort 0x0000000C 0xFFFF000C

Data Abort (data access memory abort) Abort 0x00000010 0xFFFF0010

IRQ (interrupt) IRQ 0 0x00000018 0xFFFF0018

1 IMPLEMENTATION DEFINED

FIQ (fast interrupt) FIQ 0 0x0000001C 0xFFFF001C

1 IMPLEMENTATION DEFINED

a. VE = vectored interrupt enable (CP15 control); RAZ when not implemented.

*Programmers’ Model*

When an exception occurs, the banked versions of R14 and the SPSR for the exception mode are used to save state as follows:

R14\_<exception\_mode> = return link SPSR\_<exception\_mode> = CPSR CPSR[4:0] = exception mode number CPSR[5] = 0 /\* Execute in ARM state \*/ if <exception\_mode> == Reset or FIQ then

CPSR[6] = 1 /\* Disable fast interrupts \*/ /\* else CPSR[6] is unchanged \*/ CPSR[7] = 1 /\* Disable normal interrupts \*/ if <exception\_mode> != UNDEF or SWI then

CPSR[8] = 1 /\* Disable imprecise aborts (v6 only) \*/ /\* else CPSR[8] is unchanged \*/ CPSR[9] = CP15\_reg1\_EEbit /\* Endianness on exception entry \*/ PC = exception vector address

To return after handling the exception, the SPSR is moved into the CPSR, and R14 is moved to the PC. This can be done atomically in two ways:

• using a data-processing instruction with the S bit set, and the PC as the destination

• using the Load Multiple with Restore CPSR instruction, as described in *LDM (3)* on page A4-40.

In addition, in ARMv6, the RFE instruction (see *RFE* on page A4-113) can be used to load the CPSR and PC from memory, so atomically returning from an exception to a PC and CPSR that was previously saved in memory.

Collectively these mechanisms define all of the mechanisms which perform a return from exception.

The following sections show what happens automatically when the exception occurs, and also show the recommended data-processing instruction to use to return from each exception. This instruction is always a MOVS or SUBS instruction with the PC as its destination.

**Note** When the recommended data-processing instruction is a SUBS and a Load Multiple with Restore CPSR instruction is used to return from the exception handler, the subtraction must still be performed. This is usually done at the start of the exception handler, before the return link is stored to memory.

For example, an interrupt handler that wishes to store its return link on the stack might use instructions of the following form at its entry point:

SUB R14, R14, #4 STMFD SP!, {<other\_registers>, R14}

and return using the instruction:

LDMFD SP!, {<other\_registers>, PC}^

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A2-17

*Programmers’ Model*

**A2.6.1 ARMv6 extensions to the exception model**

In ARMv6 and above, the exception model is extended as follows:

• An imprecise data abort mechanism that allows some types of data abort to be treated asynchronously. The resulting exceptions behave like interrupts, except that they use Abort mode and its banked registers. This mechanism includes a mask bit (the A bit) in the PSRs, in order to ensure that imprecise data aborts do not occur while another abort is being handled. The mechanism is described in *Imprecise data aborts* on page A2-23.

• Support for vectored interrupts controlled by the VE bit in the system control coprocessor (see *Vectored interrupt support* on page A2-26). It is IMPLEMENTATION DEFINED whether support for this mechanism is included in earlier versions of the architecture.

• Support for a low interrupt latency configuration controlled by the FI bit in the system control coprocessor (see *Low interrupt latency configuration* on page A2-27). It is IMPLEMENTATION DEFINED whether support for this mechanism is included in earlier versions of the architecture.

• Three new instructions (CPS, SRS, RFE) to improve nested stack handling of different exceptions in a common mode. CPS can also be used to efficiently enable or disable the interrupt and imprecise abort masks, either within a mode, or while transitioning from a privileged mode to any other mode. See *New instructions to improve exception handling* on page A2-28 for a brief description.

**A2.6.2 Reset**

When the Reset input is asserted on the processor, the ARM processor immediately stops execution of the current instruction. When Reset is de-asserted, the following actions are performed:

R14\_svc = UNPREDICTABLE value SPSR\_svc = UNPREDICTABLE value CPSR[4:0] = 0b10011 /\* Enter Supervisor mode \*/ CPSR[5] = 0 /\* Execute in ARM state \*/ CPSR[6] = 1 /\* Disable fast interrupts \*/ CPSR[7] = 1 /\* Disable normal interrupts \*/ CPSR[8] = 1 /\* Disable Imprecise Aborts (v6 only) \*/ CPSR[9] = CP15\_reg1\_EEbit /\* Endianness on exception entry \*/ if high vectors configured then

PC = 0xFFFF0000 elsePC = 0x00000000

After Reset, the ARM processor begins execution at address 0x00000000 or 0xFFFF0000 in Supervisor mode with interrupts disabled.

**Note** There is no architecturally defined way of returning from a Reset.

A2-18 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Programmers’ Model*

**A2.6.3 Undefined Instruction exception**

If the ARM processor executes a coprocessor instruction, it waits for any external coprocessor to acknowledge that it can execute the instruction. If no coprocessor responds, an Undefined Instruction exception occurs.

If an attempt is made to execute an instruction that is UNDEFINED, an Undefined Instruction exception occurs (see *Extending the instruction set* on page A3-32).

The Undefined Instruction exception can be used for software emulation of a coprocessor in a system that does not have the physical coprocessor (hardware), or for general-purpose instruction set extension by software emulation.

When an Undefined Instruction exception occurs, the following actions are performed:

R14\_und = address of next instruction after the Undefined instruction SPSR\_und = CPSR CPSR[4:0] = 0b11011 /\* Enter Undefined Instruction mode \*/ CPSR[5] = 0 /\* Execute in ARM state \*/ /\* CPSR[6] is unchanged \*/ CPSR[7] = 1 /\* Disable normal interrupts \*/

/\* CPSR[8] is unchanged \*/ CPSR[9] = CP15\_reg1\_EEbit /\* Endianness on exception entry \*/ if high vectors configured then

PC = 0xFFFF0004 elsePC = 0x00000004

To return after emulating the Undefined instruction use:

MOVS PC,R14

This restores the PC (from R14\_und) and CPSR (from SPSR\_und) and returns to the instruction following the Undefined instruction.

In some coprocessor designs, an internal exceptional condition caused by one coprocessor instruction is signaled *imprecisely* by refusing to respond to a later coprocessor instruction. In these circumstances, the Undefined Instruction handler takes whatever action is necessary to clear the exceptional condition, then returns to the second coprocessor instruction. To do this use:

SUBS PC,R14,#4

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A2-19

*Programmers’ Model*

**A2.6.4 Software Interrupt exception**

The Software Interrupt instruction (SWI) enters Supervisor mode to request a particular supervisor (operating system) function. When a SWI is executed, the following actions are performed:

R14\_svc = address of next instruction after the SWI instruction SPSR\_svc = CPSR CPSR[4:0] = 0b10011 /\* Enter Supervisor mode \*/ CPSR[5] = 0 /\* Execute in ARM state \*/ /\* CPSR[6] is unchanged \*/ CPSR[7] = 1 /\* Disable normal interrupts \*/

/\* CPSR[8] is unchanged \*/ CPSR[9] = CP15\_reg1\_EEbit /\* Endianness on exception entry \*/ if high vectors configured then

PC = 0xFFFF0008 elsePC = 0x00000008

To return after performing the SWI operation, use the following instruction to restore the PC (from R14\_svc) and CPSR (from SPSR\_svc) and return to the instruction following the SWI:

MOVS PC,R14

**A2.6.5 Prefetch Abort (instruction fetch memory abort)**

A memory abort is signaled by the memory system. Activating an abort in response to an instruction fetch marks the fetched instruction as invalid. A Prefetch Abort exception is generated if the processor tries to execute the invalid instruction. If the instruction is not executed (for example, as a result of a branch being taken while it is in the pipeline), no Prefetch Abort occurs.

In ARMv5 and above, a Prefetch Abort exception can also be generated as the result of executing a BKPT instruction. For details, see *BKPT* on page A4-14 (ARM instruction) and *BKPT* on page A7-24 (Thumb instruction).

When an attempt is made to execute an aborted instruction, the following actions are performed:

R14\_abt = address of the aborted instruction + 4 SPSR\_abt = CPSR CPSR[4:0] = 0b10111 /\* Enter Abort mode \*/ CPSR[5] = 0 /\* Execute in ARM state \*/ /\* CPSR[6] is unchanged \*/ CPSR[7] = 1 /\* Disable normal interrupts \*/ CPSR[8] = 1 /\* Disable Imprecise Data Aborts (v6 only) \*/ CPSR[9] = CP15\_reg1\_EEbit /\* Endianness on exception entry \*/ if high vectors configured then

PC = 0xFFFF000C elsePC = 0x0000000C

A2-20 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Programmers’ Model*

To return after fixing the reason for the abort, use:

SUBS PC,R14,#4

This restores both the PC (from R14\_abt) and CPSR (from SPSR\_abt), and returns to the aborted instruction.

**A2.6.6 Data Abort (data access memory abort)**

A memory abort is signaled by the memory system. Activating an abort in response to a data access (load or store) marks the data as invalid. A Data Abort exception occurs before any following instructions or exceptions have altered the state of the CPU. The following actions are performed:

R14\_abt = address of the aborted instruction + 8 SPSR\_abt = CPSR CPSR[4:0] = 0b10111 /\* Enter Abort mode \*/ CPSR[5] = 0 /\* Execute in ARM state \*/ /\* CPSR[6] is unchanged \*/ CPSR[7] = 1 /\* Disable normal interrupts \*/ CPSR[8] = 1 /\* Disable Imprecise Data Aborts (v6 only) \*/ CPSR[9] = CP15\_reg1\_EEbit /\* Endianness on exception entry \*/ if high vectors configured then

PC = 0xFFFF0010 elsePC = 0x00000010

To return after fixing the reason for the abort use:

SUBS PC,R14,#8

This restores both the PC (from R14\_abt) and CPSR (from SPSR\_abt), and returns to re-execute the aborted instruction.

If the aborted instruction does not need to be re-executed use:

SUBS PC,R14,#4

**Effects of data-aborted instructions**

Instructions that access data memory can modify memory by storing one or more values. If a Data Abort occurs in such an instruction, the value of each memory location that the instruction stores to is:

• unchanged if the memory system does not permit write access to the memory location

• UNPREDICTABLE otherwise.

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A2-21

*Programmers’ Model*

Instructions that access data memory can modify registers in the following ways:

• By loading values into one or more of the general-purpose registers, that can include the PC.

• By specifying *base register write-back*, in which the base register used in the address calculation has a modified value written to it. All instructions that allow this to be specified have UNPREDICTABLE results if base register write-back is specified and the base register is the PC, so only general-purpose registers other than the PC can legitimately be modified in this way.

• By loading values into coprocessor registers.

• By modifying the CPSR.

If a Data Abort occurs, the values left in these registers are determined by the following rules:

1. The PC value on entry to the Data Abort handler is 0x00000010 or 0xFFFF0010, and the R14\_abt value is determined from the address of the aborted instruction. Neither is affected in any way by the results of any PC load specified by the instruction.

2. If base register write-back is not specified, the base register value is unchanged. This applies even if the instruction loaded its own base register and the memory access to load the base register occurred earlier than the aborting access.

For example, suppose the instruction is:

LDMIA R0,{R0,R1,R2} and the implementation loads the new R0 value, then the new R1 value and finally the new R2 value. If a Data Abort occurs on any of the accesses, the value in the base register R0 of the instruction is unchanged. This applies even if it was the load of R1 or R2 that aborted, rather than the load of R0.

3. If base register write-back is specified, the value left in the base register is determined by the *abort*

*model* of the implementation, as described in *Abort models* on page A2-23.

4. If the instruction only loads one general-purpose register, the value in that register is unchanged.

5. If the instruction loads more than one general-purpose register, UNPREDICTABLE values are left in

destination registers that are neither the PC nor the base register of the instruction.

6. If the instruction loads coprocessor registers, UNPREDICTABLE values are left in the destination

coprocessor registers, unless otherwise specified in the instruction set description of the specific coprocessor.

7. CPSR bits not defined as updated on exception entry maintain their current value.

A2-22 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Programmers’ Model*

**Abort models**

The abort model used by an ARM implementation is IMPLEMENTATION DEFINED, and is one of the following:

**Base Restored Abort Model**

If a precise Data Abort occurs in an instruction that specifies base register write-back, the value in the base register is unchanged. This is the only abort model permitted in ARMv6 and above.

**Base Updated Abort Model**

If a precise Data Abort occurs in an instruction that specifies base register write-back, the base register write-back still occurs. This model is prohibited in ARMv6 and above.

In either case, the abort model applies uniformly across all instructions. An implementation does not use the Base Restored Abort Model for some instructions and the Base Updated Abort Model for others.

**A2.6.7 Imprecise data aborts**

An imprecise data abort, caused, for example, by an external error on a write that has been held in a Write Buffer, is asynchronous to the execution of the causing instruction and might in reality occur many cycles after the instruction that caused the memory access has retired. For this reason, the imprecise data abort might occur at a time that the processor is in abort mode because of a precise abort, or might have live state in abort mode, but be handling an interrupt.

To avoid the loss of the Abort mode state (R14 and SPSR\_abt) in these cases, that would lead to the processor entering an unrecoverable state, the existence of a pending imprecise data abort must be held by the system until such time as the abort mode can safely be entered.

From ARMv6, a mask is added into the CPSR (CPSR[8]) to control when an imprecise abort cannot be accepted. This bit is referred to as the A bit. The imprecise data abort causes a Data Abort to be taken when imprecise data aborts are not masked. When imprecise data aborts are masked, the implementation is responsible for holding the presence of a pending imprecise abort until the mask is cleared and the abort is taken. It is IMPLEMENTATION DEFINED whether more than one imprecise abort can be pended.

The A bit is set automatically on taking a Prefetch Abort, a Data Abort, an IRQ or FIQ interrupt, and on reset.

The A bit can only be changed from a privileged mode.

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A2-23

*Programmers’ Model*

**A2.6.8 Interrupt request (IRQ) exception**

The IRQ exception is generated externally by asserting the IRQ input on the processor. It has a lower priority than FIQ (see Table A2-1 on page A2-25), and is masked out when an FIQ sequence is entered.

Interrupts are disabled when the I bit in the CPSR is set. If the I bit is clear, ARM checks for an IRQ at instruction boundaries.

**Note** The I bit can only be changed from a privileged mode.

When an IRQ is detected, the following actions are performed:

R14\_irq = address of next instruction to be executed + 4 SPSR\_irq = CPSR CPSR[4:0] = 0b10010 /\* Enter IRQ mode \*/ CPSR[5] = 0 /\* Execute in ARM state \*/ /\* CPSR[6] is unchanged \*/ CPSR[7] = 1 /\* Disable normal interrupts \*/ CPSR[8] = 1 /\* Disable Imprecise Data Aborts (v6 only) \*/ CPSR[9] = CP15\_reg1\_EEbit /\* Endianness on exception entry \*/ if VE==0 then

if high vectors configured then

PC = 0xFFFF0018 elsePC = 0x00000018 elsePC = IMPLEMENTATION DEFINED /\* see page A2-26 \*/

To return after servicing the interrupt, use:

SUBS PC,R14,#4

This restores both the PC (from R14\_irq) and CPSR (from SPSR\_irq), and resumes execution of the interrupted code.

**A2.6.9 Fast interrupt request (FIQ) exception**

The FIQ exception is generated externally by asserting the FIQ input on the processor. FIQ is designed to support a data transfer or channel process, and has sufficient private registers to remove the need for register saving in such applications, therefore minimizing the overhead of context switching.

Fast interrupts are disabled when the F bit in the CPSR is set. If the F bit is clear, ARM checks for an FIQ at instruction boundaries.

**Note** The F bit can only be changed from a privileged mode.

When an FIQ is detected, the following actions are performed:

A2-24 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Programmers’ Model*

R14\_fiq = address of next instruction to be executed + 4 SPSR\_fiq = CPSR CPSR[4:0] = 0b10001 /\* Enter FIQ mode \*/ CPSR[5] = 0 /\* Execute in ARM state \*/ CPSR[6] = 1 /\* Disable fast interrupts \*/ CPSR[7] = 1 /\* Disable normal interrupts \*/ CPSR[8] = 1 /\* Disable Imprecise Data Aborts (v6 only) \*/ CPSR[9] = CP15\_reg1\_EEbit /\* Endianness on exception entry \*/ if VE==0 then

if high vectors configured then

PC = 0xFFFF001C elsePC = 0x0000001C elsePC = IMPLEMENTATION DEFINED /\* see page A2-26 \*/

To return after servicing the interrupt, use:

SUBS PC, R14,#4

This restores both the PC (from R14\_fiq) and CPSR (from SPSR\_fiq), and resumes execution of the interrupted code.

The FIQ vector is deliberately the last vector to allow the FIQ exception-handler software to be placed directly at address 0x0000001C or 0xFFFF001C, without requiring a branch instruction from the vector.

**A2.6.10 Exception priorities**

Table A2-1 shows the exception priorities:

**Table A2-1 Exception priorities**

**Priority Exception**

Highest 1 Reset

2 Data Abort (including data TLB miss)

3 FIQ

4 IRQ

5 Imprecise Abort (external abort) - ARMv6

6 Prefetch Abort (including prefetch TLB miss)

Lowest 7 Undefined instruction

SWI

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A2-25

*Programmers’ Model*

Undefined instruction and software interrupt cannot occur at the same time, because they each correspond to particular (non-overlapping) decodings of the current instruction. Both must be lower priority than Prefetch Abort, because a Prefetch Abort indicates that no valid instruction was fetched.

The priority of a Data Abort exception is higher than FIQ, which ensures that the Data Abort handler is entered before the FIQ handler is entered (so that the Data Abort is resolved after the FIQ handler has completed).

**A2.6.11 High vectors**

High vectors were introduced into some implementations of ARMv4 and are required in ARMv6 implementations. High vectors allow the exception vector locations to be moved from their normal address range 0x00000000-0x0000001C at the bottom of the 32-bit address space, to an alternative address range 0xFFFF0000-0xFFFF001C near the top of the address space. These alternative locations are known as the *high vectors*.

Prior to ARMv6, it is IMPLEMENTATION DEFINED whether the high vectors are supported. When they are, a hardware configuration input selects whether the normal vectors or the high vectors are to be used from reset.

The ARM instruction set does not contain any instructions that can directly change whether normal or high vectors are configured. However, if the standard System Control coprocessor is attached to an ARM processor that supports the high vectors, bit[13] of coprocessor 15 register 1 can be used to switch between using the normal vectors and the high vectors (see *Register 1: Control registers* on page B3-12).

**A2.6.12 Vectored interrupt support**

Historically, the IRQ and FIQ exception vectors are affected by whether high vectors are enabled, and are otherwise fixed. The result is that interrupt handlers typically have to start with an instruction sequence to determine the cause of the interrupt and branch to a routine to handle it. Support of vectored interrupts allows an interrupt controller to prioritize interrupts, and provide the required interrupt handler address directly to the core. The vectored interrupt behavior is explicitly enabled by the setting of a bit, the VE bit, in the system coprocessor CP15 register 1. See *Register 1: Control registers* on page B3-12. For backwards compatibility, the vectored interrupt mechanism is disabled on reset. The details of the hardware to support vectored interrupts is IMPLEMENTATION DEFINED.

A vectored interrupt controller (VIC) can reduce effective interrupt latency considerably, by eliminating the need for an interrupt handler to identify the source of an interrupt and acknowledge it before re-enabling the interrupts. Furthermore, if the VIC and core implement an appropriate handshake as the interrupt handler routine is entered, the VIC can automatically mask out the interrupt source associated with that handler and any lower priority sources. This allows the interrupts concerned to be re-enabled by the processor core as soon as their return information (that is, R14 and SPSR values) have been saved, reducing the period during which higher priority interrupts are disabled.

A2-26 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Programmers’ Model*

**A2.6.13 Low interrupt latency configuration**

The FI bit (bit[21]) in the system control register (CP15 register 1) enables the interrupt latency configuration logic in an implementation. See *Register 1: Control registers* on page B3-12. The purpose of this configuration is to reduce the interrupt latency of the processor. The exact mechanisms that are used to perform this are IMPLEMENTATION DEFINED.

In order to ensure that a change between normal and low interrupt latency configurations is synchronized correctly, the FI bit must only be changed in IMPLEMENTATION DEFINED circumstances. It is recommended that software systems should only change the FI bit shortly after reset, while interrupts are disabled.

When interrupt latency is reduced, this may result in reduced performance overall. Examples of the mechanisms which may be used are disabling Hit-Under-Miss functionality within a core, and the abandoning of restartable external accesses, allowing the core to react to a pending interrupt faster than would otherwise be the case. Low interrupt latency configuration may have IMPLEMENTATION DEFINED effects in the memory system or elsewhere outside the processor core. It is legal for the interrupt to be seen as being taken before a store to a restartable memory location, but for the memory to have been updated when in low interrupt latency configuration.

In low interrupt latency configuration, software must only use multi-word load/store instructions in ways that are fully restartable. This allows (but does not require) implementations to make multi-word instructions interruptible when in low interrupt latency configuration. The multi-access instructions to which this rule currently applies are:

**ARM** LDC, all forms of LDM, LDRD, STC, all forms of STM, STRD

**Thumb** LDMIA, PUSH, POP, STMIA

**Note** If the instruction is interrupted before it is complete, the result may be that one or more of the words are accessed twice. Idempotent memory (multiple reads or writes of the same information exhibit identical system results) is a requirement of system correctness.

In ARMv6, memory with the normal attribute is guaranteed to behave this way, however, memory marked as Device or Strongly Ordered is not (for example, a FIFO). It is IMPLEMENTATION DEFINED whether multi-word accesses are supported for Device and Strongly Ordered memory types in the low interrupt latency configuration.

A similar situation exists with regard to multi-word load/store instructions that access memory locations that can abort in a recoverable way, since an abort on one of the words accessed may cause a previously-accessed word to be accessed twice – once before the abort, and a second time after the abort handler has returned. The requirement in this case is either that all side-effects are idempotent, or that the abort must either occur on the first word accessed or not at all.

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A2-27

*Programmers’ Model*

**A2.6.14 New instructions to improve exception handling**

ARMv6 adds an instruction to simplify changes of processor mode and the disabling and enabling of interrupts. New instructions are also added to reduce the processing cost of handling exceptions in a different mode to the exception entry mode, by removing any need to use the original mode’s stack. Two examples are:

• IRQ routines may wish to execute in System or Supervisor mode, so that they can both re-enable IRQs and use BL instructions. This is not possible in IRQ mode, because a nested IRQ could corrupt the BL’s return link at any time. Using the new instructions, the system can store the return state (R14 link register and SPSR\_irq) to the System/User or Supervisor mode stack, switch to System or Supervisor mode and re-enable IRQs efficiently, without making any use of R13\_irq or the IRQ stack.

• FIQ mode is designed for efficient use by a single owner, using R8\_fiq – R13\_fiq as global variables. In addition, unlike IRQs, FIQs are not disabled by other exceptions (apart from reset), making them the preferred type for real time interrupts, when other exceptions are being used routinely, such as virtual memory or instruction emulation. IRQs may be disabled for unacceptably long periods of time while these needs are being serviced.

However, if more than one real-time interrupt source is required, there is a conflict of interest. The new mechanism allows multiple FIQ sources and minimizes the period with FIQs disabled, greatly reducing the interrupt latency penalty. The FIQ mode registers can be allocated to the highest priority FIQ as a single owner.

**SRS – Store Return State**

This instruction stores R14\_<current\_mode> and SPSR\_<current\_mode> to sequential addresses, using the banked version of R13 for a specified mode to supply the base address (and to be written back to if base register writeback is specified). This allows an exception handler to store its return state on a stack other than the one automatically selected by its exception entry sequence.

The addressing mode used is a version of ARM addressing mode 4 (see *Addressing Mode 4 - Load and Store Multiple* on page A5-41), modified so as to assume a {R14,SPSR} register list rather than using a list specified by a bit mask in the instruction. This allows the SRS instruction to access stacks in a manner compatible with the normal use of STM instructions for stack accesses. See *SRS* on page A4-174 for the instruction details.

**RFE – Return From Exception**

This instruction loads the PC and CPSR from sequential addresses. This is used to return from an exception which has had its return state saved using the SRS instruction, and again uses a version of ARM addressing mode 4, modified this time to assume a {PC,CPSR} register list. See *RFE* on page A4-113 for the instruction details.

A2-28 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Programmers’ Model*

**CPS – Change Processor State**

This instruction provides new values for the CPSR interrupt masks, mode bits, or both, and is designed to shorten and speed up the read/modify/write instruction sequence used in earlier architecture variants to perform such tasks. Together with the SRS instruction, it allows an exception handler to save its return information on the stack of another mode and then switch to that other mode, without modifying the stack belonging to the original mode or any registers other than the stack pointer of the new mode.

The instruction also streamlines interrupt mask handling and mode switches in other code, and in particular allows short, efficient, atomic code sequences in a uniprocessor system by disabling interrupts at their start and re-enabling interrupts at their end. See *CPS* on page A4-29 for the instruction details.

A CPS Thumb instruction that allows mask updates within the current mode is also provided, see section *CPS* on page A7-39.

**Note** The Thumb instruction cannot change the mode due to instruction space usage constraints.

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A2-29

*Programmers’ Model*

**A2.7 Endian support**

This section discusses memory and memory-mapped I/O, with regard to the assumptions ARM processor implementations make about endianness.

ARMv6 introduces several architectural extensions to support mixed-endian access in hardware:

• Byte reverse instructions that operate on general-purpose register contents to support word, and signed and unsigned halfword data quantities.

• Separate instruction and data endianness, with instructions fixed as little-endian format, naturally aligned, but with legacy support for 32-bit word-invariant binary images/ROM.

• A PSR Endian control flag, the E bit, which dictates the byte order used for the entire load and store instruction space when data is loaded into, and stored back out of the register file. In previous architectures this PSR bit was specified as 0 and is never set in legacy code written to conform to architectures prior to ARMv6.

• ARM and Thumb instructions to set and clear the E bit explicitly.

• A byte-invariant addressing scheme to support fine-grain big-endian and little-endian shared data structures, to conform to the *IEEE Standard for Shared-Data Formats Optimized for Scalable Coherent Interface (SCI) Processors*, IEEE Std 1596.5-1993 (ISBN 1-55937-354-7, IEEE).

• Bus interface endianness is IMPLEMENTATION DEFINED. However, it must support byte lane controls for unaligned word and halfword data access.

**A2.7.1 Address space**

The ARM architecture uses a single, flat address space of 232 8-bit bytes. Byte addresses are treated as unsigned numbers, running from 0 to 232 - 1.

This address space is regarded as consisting of 230 32-bit words, each of whose addresses is word-aligned, which means that the address is divisible by 4. The word whose word-aligned address is A consists of the four bytes with addresses A, A+1, A+2 and A+3.

In ARMv4 and above, the address space is also regarded as consisting of 231 16-bit halfwords, each of whose addresses is halfword-aligned (divisible by 2). The halfword whose halfword-aligned address is A consists of the two bytes with addresses A and A+1.

In ARMv5E and above, the address space supports 64-bit doubleword operations. Doubleword operations can be considered as two-word load/store operations, each word addressed as follows:

• A, A+1, A+2, and A+3 for the first word

• A+4, A+5, A+6, and A+7 for the second word.

Prior to ARMv6, word-aligned doubleword operations are UNPREDICTABLE with doubleword-aligned addresses always supported. ARMv6 mandates support of both modulo4 and modulo8 alignment of doublewords, and introduces support for unaligned word and halfword data accesses, all controlled through the standard System Control coprocessor.

A2-30 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Programmers’ Model*

Jazelle state (see *The T and J bits* on page A2-15) introduced with ARM architecture variant v5J supports byte addressing.

Address calculations are normally performed using ordinary integer instructions. This means that they normally wrap around if they overflow or underflow the address space. This means that the result of the calculation is reduced modulo 232.

Normal sequential execution of instructions effectively calculates:

(address\_of\_current\_instruction) + 4

after each instruction to determine which instruction to execute next. If this calculation overflows the top of the address space, the result is UNPREDICTABLE. In other words, programs should not rely on sequential execution of the instruction at address 0x00000000 after the instruction at address 0xFFFFFFFC.

The above only applies to instructions that are executed, including those which fail their condition code check. Most ARM implementations prefetch instructions ahead of the currently-executing instruction. If this prefetching overflows the top of the address space, it does not cause the implementation's behavior to become UNPREDICTABLE until and unless the prefetched instructions are actually executed.

LDC, LDM, LDRD, POP, PUSH, STC, STRD, and STM instructions access a sequence of words at increasing memory addresses, effectively incrementing a memory address by 4 for each load or store. If this calculation overflows the top of the address space, the result is UNPREDICTABLE. In other words, programs should not use these instructions in such a way that they access the word at address 0x00000000 sequentially after the word at address 0xFFFFFFFC.

Any unaligned load or store whose calculated address is such that it would access the byte at 0xFFFFFFFF and the byte at address 0x00000000 as part of the instruction is UNPREDICTABLE.

**A2.7.2 Endianness - an overview**

The rules in *Address space* on page A2-30 require that for a word-aligned address A:

• the word at address A consists of the bytes at addresses A, A+1, A+2 and A+3

• the halfword at address A consists of the bytes at addresses A and A+1

• the halfword at address A+2 consists of the bytes at addresses A+2 and A+3.

• the word at address A therefore consists of the halfwords at addresses A and A+2.

However, this does not totally specify the mappings between words, halfwords, and bytes.

A memory system uses one of the two following mapping schemes. This choice is known as the endianness of the memory system.

In a *little-endian* memory system:

• 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.

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A2-31

*Programmers’ Model*

In a *big-endian* memory system:

• 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.

For a word-aligned address A, Table A2-2 and Table A2-3 show how the word at address A, the halfwords at addresses A and A+2, and the bytes at addresses A, A+1, A+2 and A+3 map on to each other for each endianness.

**Table A2-2 Big-endian memory system**

31 24 23 16 15 8 7 0

Word at Address A

Halfword at Address A Halfword at Address A+2

Byte at Address A Byte at Address A+1 Byte at Address A+2 Byte at Address A+3

**Table A2-3 Little-endian memory system**

31 24 23 16 15 8 7 0

Word at Address A

Halfword at Address A+2 Halfword at Address A

Byte at Address A+3 Byte at Address A+2 Byte at Address A+1 Byte at Address A

On memory systems wider than 32 bits, the ARM architecture has traditionally supported a word-invariant memory model, meaning that a word aligned address will fetch the same data in both big endian and little endian systems. This is illustrated for a 64-bit data path in Table A2-4 and Table A2-5 on page A2-33.

**Table A2-4 Big-endian word invariant case**

63 32 31 0

Word at Address A+4 Word at Address A

Halfword at Address A+4

Halfword at Address A+6

Halfword at Address A

Halfword at Address A

Halfword at Address A+2

Halfword at Address A+2

Halfword at Address A+2

A2-32 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Programmers’ Model*

**Table A2-5 Little-endian word invariant case**

63 32 31 0

Word at Address A+4 Word at Address A

Halfword at

Halfword at Address A+6

Address A+4

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A2-33

Halfword at Address A+2

Halfword at Address A

**New provisions in ARMv6**

ARMv6 has introduced new configurations known as mixed endian support. These use a byte-invariant address model, affecting the order that bytes are transferred to and from ARM registers. Byte invariance means that the address of a byte in memory is the same irrespective of whether that byte is being accessed in a big endian or little endian manner.

Byte, halfword, and word accesses access the same one, two or four bytes in memory for both big and little endian configuration. Double word and multiple word accesses in the ARM architecture are treated as a series of word accesses from incrementing word addresses, and hence each word also returns the same bytes of information in these cases too.

**Note** When an implementation is configured in mixed endian mode, this only affects data accesses and how they are loaded/stored to/from the register file. Instruction fetches always assume a little endian byte order model.

• When configured for big endian load/store, the lowest address provides the most significant byte of the requested word or halfword. For LDRD/STRD this is the most significant byte of the first word accessed.

• When configured for little endian load/store, the lowest address provides the least significant byte of the requested word or halfword. For LDRD/STRD this is the least significant byte of the first word accessed.

The convention adopted in this book is to identify the different endian models as follows:

• the word invariant big endian model is known as BE-32

• the byte invariant big endian model is referred to as BE-8

• little endian data is identical in both models and referred to as LE.

*Programmers’ Model*

**A2.7.3 Endian configuration and control**

Prior to ARMv6, a single bit (B bit) provides endian control. It is IMPLEMENTATION DEFINED whether implementations of ARMv5 and below support little-endian memory systems, big-endian memory systems, or both. If a standard System Control coprocessor is attached to an ARM implementation supporting the B bit, this configuration input can be changed by writing to bit[7] of register 1 of the System Control coprocessor (see *Register 1: Control registers* on page B3-12). An implementation may preset the B bit on reset. If an ARM processor configures for little-endian operation on reset, and it is attached to a big-endian memory system, one of the first things the reset handler must do is switch the configured endianness to big-endian, using an instruction sequence like:

MRC p15, 0, r0, c1, c0 ; r0 := CP15 register 1 ORR r0, r0, #0x80 ; Set bit[7] in r0 MCR p15, 0, r0, c1, c0 ; CP15 register 1 := r0

This must be done before there is any possibility of a byte or halfword data access occurring, or instruction execution in Thumb or Jazelle state.

ARMv6 supports big-endian, little-endian, and byte-invariant hybrid systems. LE and BE-8 formats must be supported. Support of BE-32 is IMPLEMENTATION DEFINED.

Features are provided in the System Control coprocessor and CPSR/SPSR to support hybrid operation. The System Control Coprocessor register (CP15 register 1) and CPSR bits used are:

• Bit[1] - A bit - used to enable alignment checking. Always reset to zero (alignment checking OFF).

• Bit[7] - B bit - OPTIONAL, retained for backwards compatibility

• Bit[22] - the U bit - enables ARMv6 unaligned data support, and used with Bit[1] - the A bit - to determine alignment checking behavior.

• Bit [25] - the EE bit - Exception Endian bit.

• CPSR/SPSR[9] - the E bit - load/store endian control.

The behavior of the memory system with respect to the U and A bits is summarized in Table A2-6.

**Table A2-6**

**U A Description**

0 0 Legacy (32-bit word invariant only)

0 1 Modulo 8 alignment checking: LDRD/STRD (8 and 32-bit invariant

memory models)

1 0 Unaligned access support (8-bit byte invariant data accesses only)

1 1 Modulo 4 alignment checking: LDRD/STRD (8-bit and 32-bit invariant

memory models)

A2-34 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Programmers’ Model*

The EE-bit value is used to overwrite the CPSR\_E bit on exception entry and for page table lookups. These are asynchronous events with respect to normal control of the CPSR E bit.

A 2-bit configuration (CFGEND[1:0]) replaces the BigEndinit configuration pin to provide hardware system configuration on reset. CFGEND[1] maps to the U bit, while CFGEND[0] sets either the B bit or EE bit and CPSR\_E on reset.

Table A2-7 defines the CFGEND[1:0] encoding and associated configurations.

Where an implementation does not include configuration pins, the U bit and A bit shall clear on reset.

The usage model for the U bit and A bit with respect to the B bit and E bit is summarized in Table A2-8. Where BE-32 is not supported, the B bit must read as zero, and all entries indicated by B==1 are RESERVED. Interaction of these control bits with data alignment is discussed in *Unaligned access support* on page A2-38.

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A2-35 **Table A2-7**

**CFGEND[1:0] Coprocessor 15 System Control Register (register 1) CPSR/SPSR**

EE bit[25] U bit[22] A bit[1] B bit[7] E bit

00 0 0 0 0 0

01a0 0 0 1 0

10 0 1 0 0 0

11 1 1 0 0 1

a. This configuration is RESERVED in implementations which do not support BE-32. In this case, the B bit

must read as zero (RAZ).

**Table A2-8 Endian and Alignment Control Bit Usage Summary**

**U A B E Instruction**

**Endianness**

**Data Endianness**

**Unaligned Behavior Description**

0 0 0 0 LE LE Rotated LDR Legacy LE / programmed BE

configuration

0 0 0 1 - - - RESERVED (no E bit in legacy code)

0 0 1 0 BE-32 BE-32 Rotated LDR Legacy BE (32-bit word-invariant)

0 0 1 1 - - - RESERVED (no E bit in legacy code)

0 1 0 0 LE LE Data Abort modulo 8 LDRD/STRD doubleword

alignment checking. LE Data

*Programmers’ Model*

**Table A2-8 Endian and Alignment Control Bit Usage Summary (continued)**

**U A B E Instruction**

**Endianness**

**Data**

**Unaligned Endianness**

**Behavior Description**

0 1 0 1 LE BE-8 Data Abort modulo 8 LDRD/STRD doubleword

alignment checking. BE Data

0 1 1 0 BE-32 BE-32 Data Abort modulo 8 LDRD/STRD doubleword

alignment checking, legacy BE

0 1 1 1 - - - RESERVED

1 0 0 0 LE LE Unaligned LE instructions, LE mixed-endian data,

unaligned access permitted

1 0 0 1 LE BE-8 Unaligned LE instructions, BE mixed-endian data,

unaligned access permitted

1 0 1 x - - - RESERVED

1 1 0 0 LE LE Data Abort modulo 4 alignment checking, LE Data

1 1 0 1 LE BE-8 Data Abort modulo 4 alignment checking, BE data

1 1 1 0 BE-32 BE-32 Data Abort modulo 4 alignment checking, legacy BE

1 1 1 1 - - - RESERVED

BE-32 and BE-8 are as defined in *Endianness - an overview* on page A2-31. Data aborts cause an alignment error to be reported in the Fault Status Register in the system coprocessor.

**Note** The U, A and B bits are System Control Coprocessor bits, while the E bit is a CPSR/SPSR flag.

The behavior of SETEND instructions (or any other instruction that modifies the CPSR) is UNPREDICTABLE when setting the E bit would result in a RESERVED state.

**A2.7.4 Instructions to change CPSR E bit**

ARM and Thumb instructions are provided to set and clear the E bit efficiently: **SETEND BE** Set the CPSR E bit. **SETEND LE** Reset the CPSR E bit.

These are unconditional instructions. See ARM *SETEND* on page A4-129 and Thumb *SETEND* on page A7-95.

A2-36 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Programmers’ Model*

**A2.7.5 Instructions to reverse bytes in a general-purpose register**

When an application or device driver has to interface to memory-mapped peripheral registers or shared-memory DMA structures that are not the same endianness as that of the internal data structures, or the endianness of the Operating System, an efficient way of being able to explicitly transform the endianness of the data is required.

ARMv6 ARM and Thumb instruction sets provide this functionality:

• Reverse word (four bytes) register, for transforming big and little-endian 32-bit representations. See ARM *REV* on page A4-109 and Thumb *REV* on page A7-88.

• Reverse halfword and sign-extend, for transforming signed 16-bit representations. See ARM *REVSH* on page A4-111 and Thumb *REVSH* on page A7-90.

• Reverse packed halfwords in a register for transforming big- and little-endian 16-bit representations. See ARM *REV16* on page A4-110 and Thumb *REV16* on page A7-89.

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A2-37

*Programmers’ Model*

**A2.8 Unaligned access support**

The ARM architecture traditionally expects all memory accesses to be suitably aligned. In particular, the address used for a halfword access should normally be halfword-aligned, the address used for a word access should normally be word-aligned.

Prior to ARMv6, doubleword (LDRD/STRD) accesses to memory, where the address is not doubleword-aligned, are UNPREDICTABLE. Also, data accesses to non-aligned word and halfword data are treated as aligned from the memory interface perspective. That is:

• the address is treated as truncated, with address bits[1:0] treated as zero for word accesses, and address bit[0] treated as zero for halfword accesses.

• load single word ARM instructions are architecturally defined to rotate right the word-aligned data transferred by a non word-aligned address one, two or three bytes depending on the value of the two least significant address bits.

• alignment checking is defined for implementations supporting a System Control coprocessor using the A bit in CP15 register 1. When this bit is set, a Data Abort indicating an alignment fault is reported for unaligned accesses.

ARMv6 introduces unaligned word and halfword load and store data access support. When this is enabled, the processor uses one or more memory accesses to generate the required transfer of adjacent bytes transparently to the programmer, apart from a potential access time penalty where the transaction crosses an IMPLEMENTATION DEFINED cache-line, bus-width or page boundary condition. Doubleword accesses must be word-aligned in this configuration.

**A2.8.1 Unaligned instruction fetches**

All instruction fetches must be aligned. Specifically they must be:

• word aligned in ARM state

• halfword aligned in Thumb state.

Writing an unaligned address to R15 is UNPREDICTABLE, except in the specific cases where the instructions are associated with a Thumb to ARM state transition, bit[1] providing a valid address bit on transition to Thumb state, and bit[0] indicating whether a transition needs to occur. The BX instruction in ARM state (see *BX* on page A4-20) and POP instruction in Thumb state (see *POP* on page A7-82) are examples of instructions providing state transition support.

The general rules for reading and writing the program counter are defined in *Register 15 and the program counter* on page A2-9.

A2-38 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Programmers’ Model*

**A2.8.2 Unaligned data access in ARMv6 systems**

ARMv6 uses the U bit (CP15 register 1 bit[22]) and A bit (CP15 register 1 bit[1]), to provide a configuration supporting the following unaligned memory accesses:

• Unaligned halfword accesses for LDRH, LDRSH and STRH.

• Unaligned word accesses for LDR, LDRT, STR and STRT.

The U bit and A bit are also used to configure endian support as described in *Endian configuration and control* on page A2-34. All other multi-byte load and store accesses shall be word aligned.

Instructions must always be aligned (and in little endian format):

• ARM instructions must be word-aligned

• Thumb instructions must be halfword-aligned.

In addition, an ARMv6 system shall reset to the CFGEND[1:0] condition as described in Table A2-7 on page A2-35.

For ARMv6, Table A2-10 on page A2-40 defines when an alignment fault must occur for an access, and when the behavior of an access is architecturally UNPREDICTABLE. It also gives details of precisely which memory locations are returned for valid accesses.

The access type descriptions used in this section are determined from the load/store instructions as described in Table A2-9:

The following terminology is used to describe the memory locations accessed:

**Byte[X]** Means the byte whose address is X in the current endianness model. The correspondence

between the endianness models is that Byte[A] in the LE endianness model, Byte[A] in the BE-8 endianness model, and Byte[A EOR 3] in the BE-32 endianness model are the same actual byte of memory.

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A2-39 **Table A2-9**

**Access Type ARM instructions Thumb instructions**

Byte LDRB LDRBT LDRSB STRB STRBT SWPB (either access) LDRB LDRSB STRB

Halfword LDRH LDRSH STRH LDRH LDRSH STRH

WLoad LDR LDRT SWP (load access, if U == 0) LDR

WStore STR STRT SWP (store access, if U == 0) STR

WSync LDREX STREX SWP (either access, if U == 1) -

Two-word LDRD STRD -

Multi-word LDC LDM RFE SRS STC STM LDMIA POP PUSH STMIA

*Programmers’ Model*

**Halfword[X]** Means the halfword consisting of the bytes whose addresses are X and X+1 in the current

endianness model, combined to form a halfword in little-endian order in the LE endianness model or in big-endian order in the BE-8 or BE-32 endianness model.

**Word[X]** Means the word consisting of the bytes whose addresses are X, X+1, X+2, and X+3 in the

current endianness model, combined to form a word in little-endian order in the LE endianness model or in big-endian order in the BE-8 or BE-32 endianness model.

**Note** It is a consequence of these definitions that if X is word-aligned, Word[X] consists of the same four bytes of actual memory in the same order in the LE and BE-32 endianness models.

**Align[X]** Means (X AND 0xFFFFFFFC) - that is, X with its least significant two bits forced to zero to make

it word-aligned.

**Note** There is no difference between Addr and Align(Addr) on lines for which Addr[1:0] == 0b00 anyway. This can be exploited by implementations to simplify the control of when the least significant bits are forced to zero.

For the Two-word and Multi-word access types, the Memory accessed column only specifies the lowest word accessed. Subsequent words have addresses constructed by successively incrementing the address of the lowest word by 4, and are constructed using the same endianness model as the lowest word.

**Table A2-10 Data Access Behavior in ARMv6 Systems**

**U A Addr[2:0] Access**

**Types Behavior Memory**

**accessed Notes**

0 0 LEGACY, NO

ALIGNMENT FAULTING

0 0 xxx Byte Normal Byte[Addr] -

0 0 xx0 Halfword Normal Halfword[Addr] -

0 0 xx1 Halfword UNPREDICTABLE - -

0 0 xxx WLoad Normal Word[Align(Addr)] Loaded data rotated right by

8 \* Addr[1:0] bits

0 0 xxx WStore Normal Word[Align(Addr)] Operation unaffected by

Addr[1:0]

0 0 x00 WSync Normal Word[Addr] -

A2-40 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I

*Programmers’ Model*

**Table A2-10 Data Access Behavior in ARMv6 Systems (continued)**

**U A Addr[2:0] Access**

**Types Behavior Memory**

**accessed Notes**

0 0 xx1, x1x WSync UNPREDICTABLE - -

0 0 xxx Multi-word Normal Word[Align(Addr)] Operation unaffected by

Addr[1:0]

0 0 000 Two-word Normal Word[Addr] -

0 0 xx1, x1x,

1xx

ARM DDI 0100I *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* A2-41

Two-word UNPREDICTABLE - -

1 0 NEW ARMv6

UNALIGNED SUPPORT

1 0 xxx Byte Normal Byte[Addr] -

1 0 xxx Halfword Normal Halfword[Addr] -

1 0 xxx WLoad WStore

Normal Word[Addr] -

1 0 x00 WSync

Multi-word Two-word

Normal Word[Addr] -

1 0 xx1, x1x WSync

Multi-word Two-word

Alignment Fault - -

x 1 FULL ALIGNMENT

FAULTING

x 1 xxx Byte Normal Byte[Addr] -

x 1 xx0 Halfword Normal Halfword[Addr] -

x 1 xx1 Halfword Alignment Fault - -

x 1 x00 WLoad WStore WSync Multi-word

Normal Word[Addr] -

x 1 xx1, x1x WLoad WStore WSync Multi-word

Alignment Fault - -

*Programmers’ Model*

**Table A2-10 Data Access Behavior in ARMv6 Systems (continued)**

**U A Addr[2:0] Access**

**Types Behavior Memory**

**accessed Notes**

x 1 000 Two-word Normal Word[Addr] -

0 1 100 Two-word Alignment Fault - -

1 1 100 Two-word Normal Word[Addr] -

x 1 xx1, x1x Two-word Alignment Fault - -

**Other reasons for unaligned accesses to be UNPREDICTABLE**

The following exceptions to the behavior described in Table A2-10 on page A2-40 apply, causing the resultant unaligned accesses to be UNPREDICTABLE:

• An LDR instruction that loads the PC, has Addr[1:0] != 0b00, and is specified in the table as having Normal behavior instead has UNPREDICTABLE behavior.

**Note** The reason this applies only to LDR is that most other load instructions are UNPREDICTABLE regardless of alignment if the PC is specified as their destination register. The exceptions are LDM, RFE and Thumb POP. If Addr[1:0] != 0b00 for these instructions, the effective address of the transfer has its two least significant bits forced to 0 if A == 0 and U ==0, and otherwise the behavior specified in the table is either UNPREDICTABLE or Alignment Fault regardless of the destination register.

• Any WLoad, WStore, WSync, Two-word or Multi-word instruction that accesses memory with the Strongly Ordered or Device memory attribute, has Addr[1:0] != 0b00, and is specified in the table as having Normal behavior instead has UNPREDICTABLE behavior.

• Any Halfword instruction that accesses memory with the Strongly Ordered or Device memory attribute, has Addr[0] != 0, and is specified in the table as having Normal behavior instead has UNPREDICTABLE behavior.

If any of these reasons applies, it overrides the behavior specified in the table.

**Note** These reasons never cause Alignment Fault behavior to be overridden.

ARM implementations are not required to ensure that the low-order address bits that make an access unaligned are cleared from the address they send to memory. They can instead send the address as calculated by the load/store instruction unchanged to memory, and require the memory system to ignore address[0] for a halfword access and address[1:0] for a word access.

A2-42 *Copyright © 1996-1998, 2000, 2004, 2005 ARM Limited. All rights reserved.* ARM DDI 0100I