# ARM7DI

## Data Sheet

Document Number: ARM DDI 0027D

Issued: Dec 1994

Copyright Advanced RISC Machines Ltd (ARM) 1994

All rights reserved

### Proprietary Notice

ARM, the ARM Powered logo, BlackICE and ICEbreaker are trademarks of Advanced RISC Machines Ltd.

Neither the whole nor any part of the information contained in, or the product described in, this datasheet may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this datasheet is subject to continuous developments and improvements. All particulars of the product and its use contained in this datasheet are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties or merchantability, or fitness for purpose, are excluded.

This datasheet is intended only to assist the reader in the use of the product. ARM Ltd shall not be liable for any loss or damage arising from the use of any information in this datasheet, or any error or omission in such information, or any incorrect use of the product.

### Change Log

| Issue | Date | By | Change |
|---|---|---|---|
| A | July 1994 | EH | Created. |
| B | Aug 94 | BJH | Updated Instruction Cycle Operations |
| C | Oct 94 | EH | Sources repaired: no material changes to text |
| D | Dec 94 | PB | Edited. |

ARM
Advanced RISC Machines

# Preface

The ARM7DI is a low-power, general purpose 32-bit RISC microprocessor with integrated debug support. It comprises the ARM7D CPU core, and ICEbreaker module and a TAP controller. Its simple, elegant and fully static design is particularly suitable for cost and power sensitive applications.

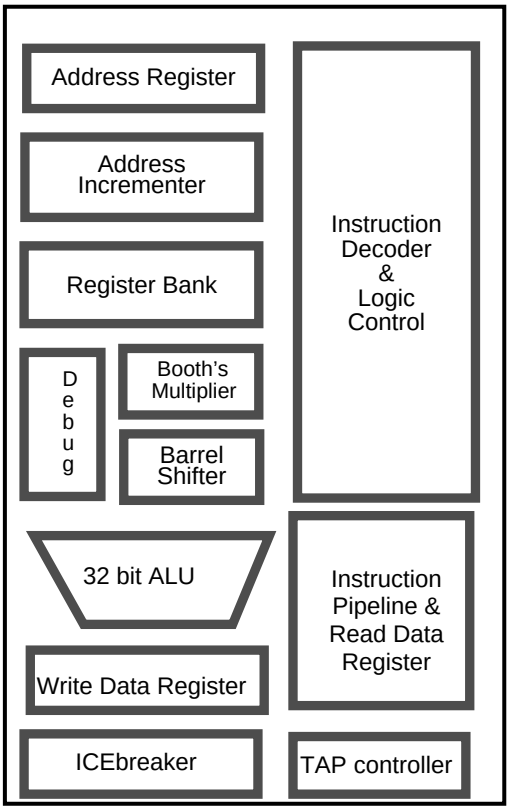
## Enhancements

The ARM7DI is similar to the ARM6 but with the following enhancements:

- *advanced debug (integrated ICE) support for faster time to market*
- *fabrication on a sub-micron process for increased speed and reduced power consumption*
- *3V operation, for very low power consumption, as well as 5V operation for system compatibility*
- *higher clock speedfor faster program execution.*

## Feature Summary

- *32-bit RISC processor (32-bit data & address bus)*
- *Advanced debug*
  fully integrated ICE
- *Big and Little Endian operating modes*
- *High performance RISC*
- *Low power consumption*
- *Fully static operation*
  ideal for power-sensitive applications
- *Fast interrupt response*
  for real-time applications
- *Virtual Memory System Support*
- *Excellent high-level language support*
- *Simple but powerful instruction set*




## Applications

The ARM7DI is ideally suited to those applications requiring RISC performance from a compact, power-efficient processor. These include:

| | |
|---|---|
| **Telecomms** | GSM terminal controller |
| **Datacomms** | Protocol conversion |
| **Portable Computing** | Palmtop computer |
| **Portable InstrumentS** | Handheld data acquisition unit |
| **Automotive** | Engine management unit |
| **Information Systems** | Smart cards |
| **Imaging** | JPEG controller |

# Table of Contents


**1.0 Introduction 5**
- 1.1 ARM7DI Block Diagram 6
- 1.2 ARM7D Core Diagram 7
- 1.3 ARM7DI Functional Diagram 8

**2.0 Signal Description 9**

**3.0 Programmer's Model 15**
- 3.1 Hardware Configuration Signals 15
- 3.2 Operating Mode Selection 16
- 3.3 Registers 17
- 3.4 Exceptions 20
- 3.5 Reset 24

**4.0 Instruction Set 25**
- 4.1 Instruction Set Summary 25
- 4.2 The Condition Field 26
- 4.3 Branch and Branch with link (B, BL) 27
- 4.4 Data processing 29
- 4.5 PSR Transfer (MRS, MSR) 36
- 4.6 Multiply and Multiply-Accumulate (MUL, MLA) 40
- 4.7 Single data transfer (LDR, STR) 42
- 4.8 Block Data Transfer (LDM, STM) 48
- 4.9 Single data swap (SWP) 55
- 4.10 Software interrupt (SWI) 57
- 4.11 Coprocessor data operations (CDP) 59
- 4.12 Coprocessor data transfers (LDC, STC) 61
- 4.13 Coprocessor register transfers (MRC, MCR) 64
- 4.14 Undefined instruction 66
- 4.15 Instruction Set Examples 67

**5.0 Memory Interface 71**
- 5.1 Cycle types 71
- 5.2 Byte addressing 72
- 5.3 Address timing 74
- 5.4 Memory management 74
- 5.5 Locked operations 75
- 5.6 Stretching access times 75
- 5.7 The External Data Bus 76

**6.0 Coprocessor Interface 81**
- 6.1 Interface signals 81
- 6.2 Data transfer cycles 82
- 6.3 Register transfer cycle 82
- 6.4 Privileged instructions 82
- 6.5 Idempotency 83
- 6.6 Undefined instructions 83

**7.0 Debug Interface 85**
- 7.1 Overview 85
- 7.2 Debug Systems 85
- 7.3 Debug Interface Signals 86

7.4 Scan Chains and JTAG Interface 89
7.5 Reset 91
7.6 Pullup Resistors 91
7.7 Instruction Register 92
7.8 Public Instructions 92
7.9 Test Data Registers 94
7.10 ARM7DI Core Clocks 99
7.11 Determining the Core and System State 100
7.12 The PC's Behaviour During Debug 103
7.13 Priorities / Exceptions 105
7.14 Scan Interface Signals 106

**8.0 The ARM7DI ICEBreaker Module 109**
8.1 The Watchpoint Registers 110
8.2 The Debug Control Register 115
8.3 Debug Status Register 116
8.4 Coupling Breakpoints and Watchpoints 119
8.5 Disabling ICEbreaker 120
8.6 ICEbreaker Timing 120
8.7 ICEBreaker Programming Restriction 120

**9.0 Instruction Cycle Operations 121**
9.1 Branch and branch with link 121
9.2 Data Operations 121
9.3 Multiply and multiply accumulate 123
9.4 Load register 123
9.5 Store register 124
9.6 Load multiple registers 125
9.7 Store multiple registers 127
9.8 Data swap 127
9.9 Software interrupt and exception entry 128
9.10 Coprocessor data operation 129
9.11 Coprocessor data transfer (from memory to coprocessor) 129
9.12 Coprocessor data transfer (from coprocessor to memory) 131
9.13 Coprocessor register transfer (Load from coprocessor) 132
9.14 Coprocessor register transfer (Store to coprocessor) 132
9.15 Undefined instructions and coprocessor absent 133
9.16 Unexecuted instructions 134
9.17 Instruction Speed Summary 134

**10.0 DC Parameters 137**
10.1 Absolute Maximum Ratings 137
10.2 DC Operating Conditions: 137

**11.0 AC Parameters 139**

**12.0 Appendix - Backward Compatibility 149**

# 1.0 Introduction

The ARM7DI is part of the Advanced RISC Machines (ARM) family of general purpose 32-bit microprocessors, which offer very low power consumption and price for high performance devices. The architecture is based on Reduced Instruction Set Computer (RISC) principles, and the instruction set and related decode mechanism are much simpler in comparison with microprogrammed Complex Instruction Set Computers. This results in a high instruction throughput and impressive real-time interrupt response from a small and cost-effective chip.

The instruction set comprises eleven basic instruction types:

- Two of these make use of the on-chip arithmetic logic unit, barrel shifter and multiplier to perform high-speed operations on the data in a bank of 31 registers, each 32 bits wide;
- Three classes of instruction control data transfer between memory and the registers, one optimised for flexibility of addressing, another for rapid context switching and the third for swapping data;
- Three instructions control the flow and privilege level of execution; and
- Three types are dedicated to the control of external coprocessors which allow the functionality of the instruction set to be extended off-chip in an open and uniform way.

The ARM instruction set is a good target for compilers of many different high-level languages. Where required for critical code segments, assembly code programming is also straightforward, unlike some RISC processors which depend on sophisticated compiler technology to manage complicated instruction interdependencies.

Pipelining is employed so that all parts of the processing and memory systems can operate continuously. Typically, while one instruction is being executed, its successor is being decoded, and a third instruction is being fetched from memory.

The memory interface has been designed to allow the performance potential to be realised without incurring high costs in the memory system. Speed critical control signals are pipelined to allow system control functions to be implemented in standard low-power logic, and these control signals facilitate the exploitation of the fast local access modes offered by industry standard dynamic RAMs.

ARM7DI has a 32 bit address bus. All ARM processors share the same instruction set, and ARM7DI can be configured to use a 26 bit address bus for backwards compatibility with earlier processors.

ARM7DI is a fully static CMOS implementation of the ARM which allows the clock to be stopped in any part of the cycle with extremely low residual power consumption and no loss of state.

**Notation:**

| | |
|---|---|
| 0x | - marks a Hexadecimal quantity |
| **BOLD** | - external signals are shown in bold capital letters |
| binary | - where it is not clear that a quantity is binary it is followed by the word binary |

## 1.1 ARM7DI Block Diagram

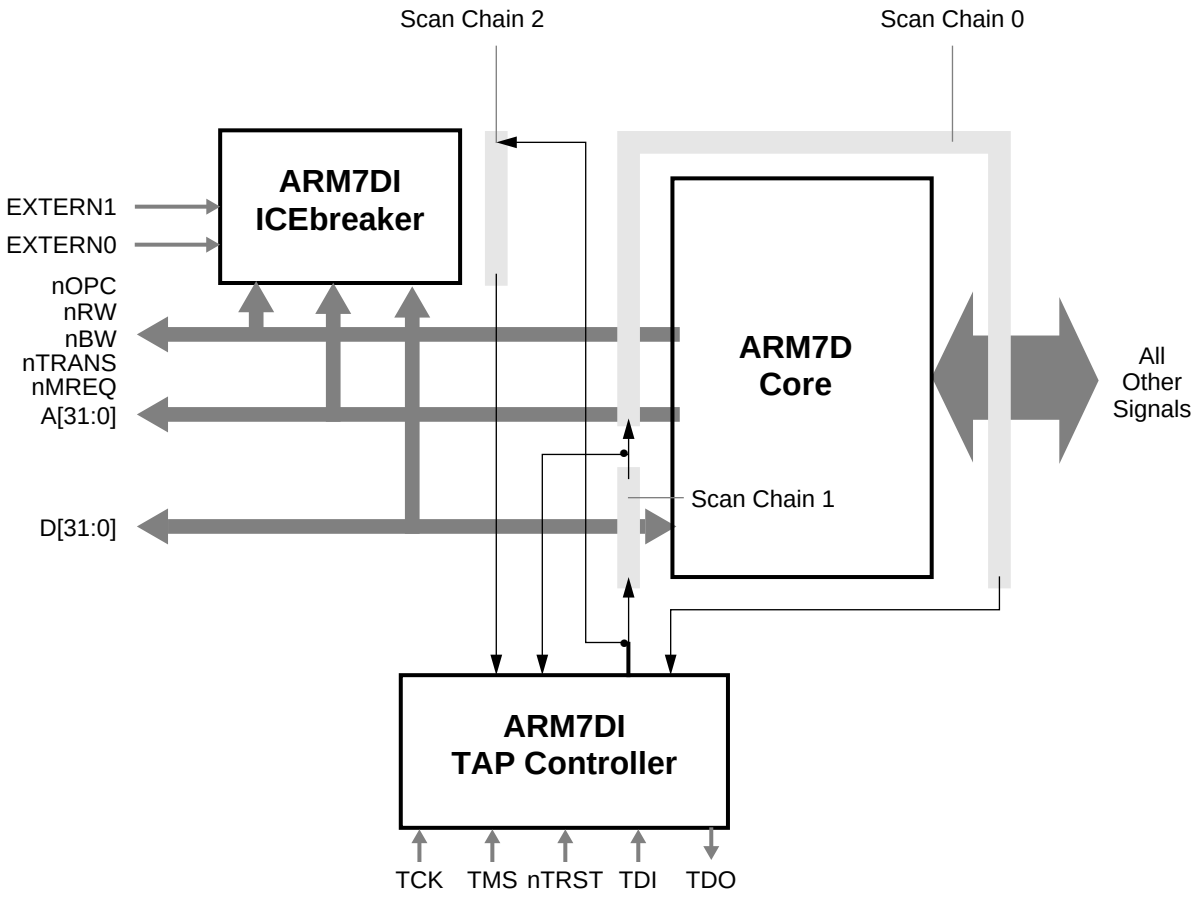



**Figure 1: ARM7DI Block Diagram**

## 1.2 ARM7D Core Diagram

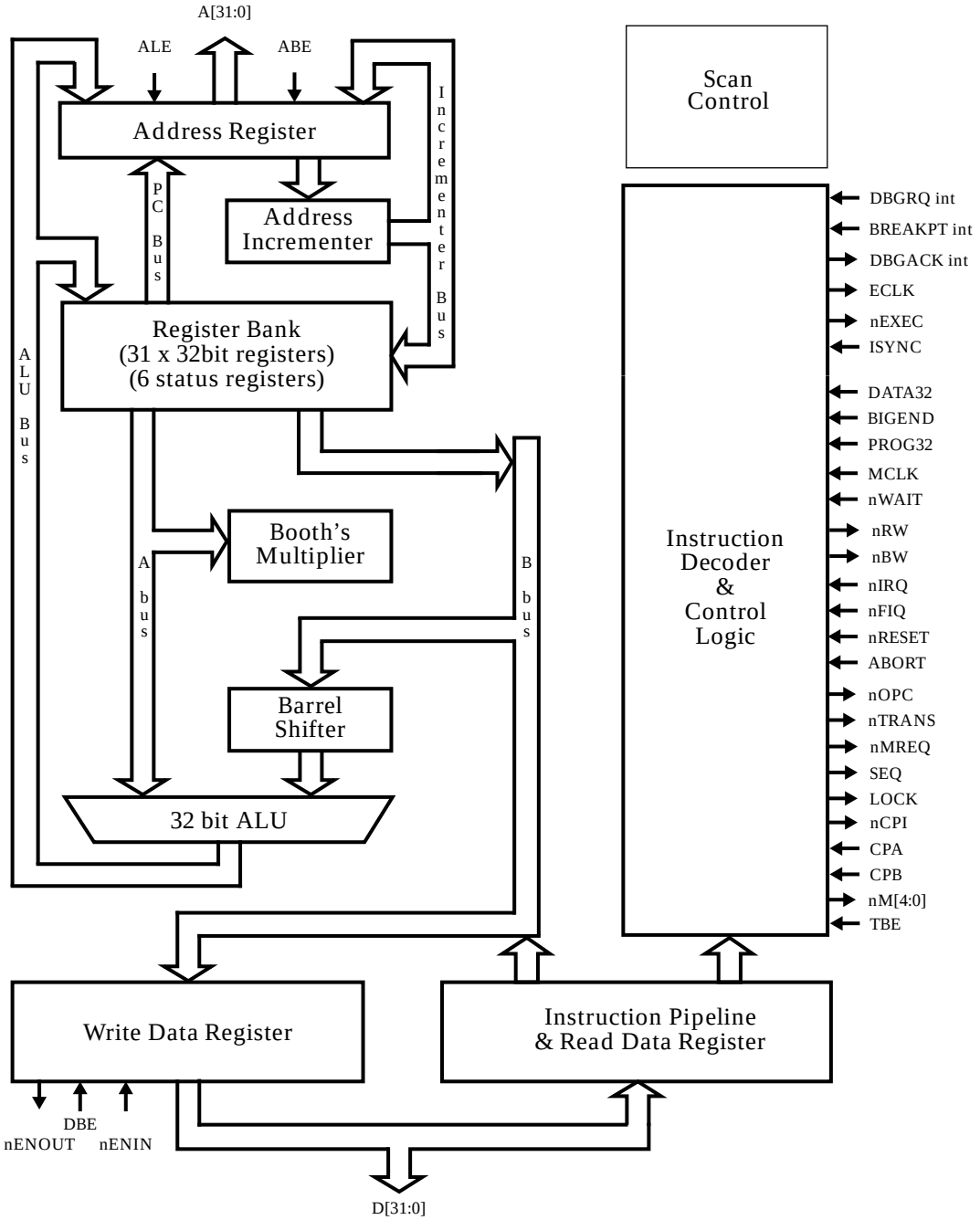



**Figure 2: ARM7D Core Diagram**

## 1.3 ARM7DI Functional Diagram

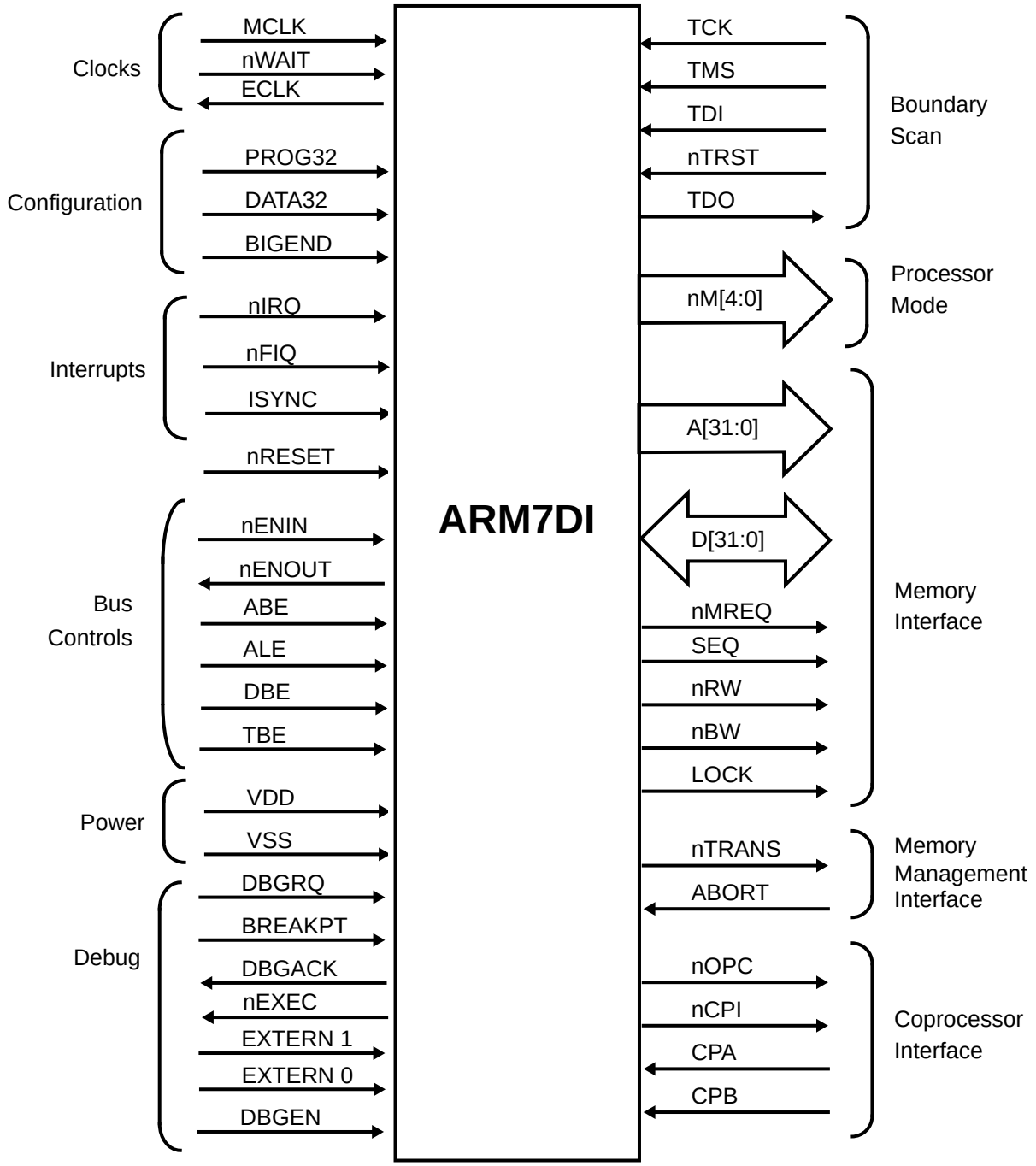



**Figure 3: ARM7DI Functional Diagram**

# 2.0 Signal Description

| Name | Type | Description |
|---|---|---|
| **A[31:0]** | O8 | Addresses. This is the processor address bus. If **ALE** (address latch enable) is HIGH, the addresses become valid during phase 2 of the cycle before the one to which they refer and remain so during phase 1 of the referenced cycle. Their stable period may be controlled by **ALE** as described below. |
| **ABE** | IC | Address bus enable. This is an input signal which, when LOW, puts the address bus drivers into a high impedance state. This signal has a similar effect on the following control signals: **nBW**, **nRW**, **LOCK**, **nOPC** and **nTRANS**. **ABE** must be tied HIGH when there is no system requirement to turn off the address drivers. |
| **ABORT** | IC | Memory Abort. This is an input which allows the memory system to tell the processor that a requested access is not allowed. |
| **ALE** | IC | Address latch enable. This input is used to control transparent latches on the address outputs. Normally the addresses change during phase 2 to the value required during the next cycle, but for direct interfacing to ROMs they are required to be stable to the end of phase 2. Taking **ALE** LOW until the end of phase 2 will ensure that this happens. This signal has a similar effect on the following control signals: **nBW**, **nRW**, **LOCK**, **nOPC** and **nTRANS**. If the system does not require address lines to be held in this way, **ALE** must be tied HIGH. The address latch is static, so **ALE** may be held LOW for long periods to freeze addresses. |
| **BIGEND** | IC | Big Endian configuration. When this signal is HIGH the processor treats bytes in memory as being in Big Endian format. When it is LOW memory is treated as Little Endian. ARM processors which do not have selectable Endianism (ARM2, ARM2aS, ARM3, ARM61) are Little Endian. |
| **BREAKPT** | IC | Breakpoint. This signal allows external hardware to halt the execution of the processor for debug purposes. When HIGH causes the current memory access to be breakpointed. If the memory access is an instruction fetch, ARM7DI will enter debug state if the instruction reaches the execute stage of the ARM7DI pipeline. If the memory access is for data, ARM7DI will enter debug state after the current instruction completes execution.This allows extension of the internal breakpoints provided by the ICEBreaker module. See *Chapter 8.0 The ARM7DI ICEBreaker Module.* |
| **CPA** | IC | Coprocessor absent. A coprocessor which is capable of performing the operation that ARM7DI is requesting (by asserting **nCPI**) should take **CPA** LOW immediately. If **CPA** is HIGH at the end of phase 1 of the cycle in which **nCPI** went LOW, ARM7DI will abort the coprocessor handshake and take the undefined instruction trap. If **CPA** is LOW and remains LOW, ARM7DI will busy-wait until **CPB** is LOW and then complete the coprocessor instruction. |

**Table 1: Signal Description**

| Name | Type | Description |
|---|---|---|
| **CPB** | IC | Coprocessor busy. A coprocessor which is capable of performing the operation which ARM7DI is requesting (by asserting **nCPI**), but cannot commit to starting it immediately, should indicate this by driving **CPB** HIGH. When the coprocessor is ready to start it should take **CPB** LOW. ARM7DI samples **CPB** at the end of phase 1 of each cycle in which **nCPI** is LOW. |
| **D[31:0]** | IC / O8 | Data Bus. These are bidirectional signal paths which are used for data transfers between the processor and external memory. During read cycles (when **nRW** is LOW), the input data must be valid before the end of phase 2 of the transfer cycle. During write cycles (when **nRW** is HIGH), the output data will become valid during phase 1 and remain valid throughout phase 2 of the transfer cycle. |
| **DATA32** | IC | 32 bit Data configuration. When this signal is HIGH the processor can access data in a 32 bit address space using address lines **A[31:0]**. When it is LOW the processor can access data from a 26 bit address space using **A[25:0]**. In this latter configuration the address lines **A[31:26]** are not used. Before changing DATA32, ensure that the processor is not about to access an address greater that 0x3FFFFFF in the next cycle. |
| **DBE** | IC | Internal Data Bus Enable. This is an input signal which, when driven LOW, puts the internal data bus (the bus between the core logic and the pads) into the high impedance state. This is included for test purposes, and should be tied HIGH at all times. |
| **DBGACK** | O4 | Debug acknowledge. When HIGH indicates ARM is in debug state. |
| **DBGEN** | IC | Debug Enable. This input signal allows the debug features of ARM7DI to be disabled. This signal should be driven LOW when debugging is not required. |
| **DBGRQ** | IC | Debug request. This is a level-sensitive input, which when HIGH causes ARM7DI to enter debug state after executing the current instruction. This allows external hardware to force ARM7DI into the debug state, in addition to the debugging features provided by the ICEBreaker block. See *Chapter 8.0 The ARM7DI ICEBreaker Module* for details. |
| **ECLK** | O4 | External clock output. In normal operation, this is simply **MCLK** (optionally stretched with **nWAIT**) exported from the core. When the core is being debugged, this is **DCLK**. This allows external hardware to track when the ARM7DI core is clocked. |
| **EXTERN1** | IC | External input 1. This is an input to the ICEBreaker logic in the ARM7DI which allows breakpoints and / or watchpoints to be dependent on an external condition. |
| **EXTERN0** | IC | External input 0. This is an input to the ICEBreaker logic in the ARM7DI which allows breakpoints and / or watchpoints to be dependent on an external condition. |
| **HIGHZ** | O4 | This signal denotes that the HIGHZ instruction has been loaded into the TAP controller. See *Chapter 7.0 Debug Interface* for details. |

**Table 1: Signal Description (Continued)**

| Name | Type | Description |
|---|---|---|
| **ISYNC** | IC | Synchronous interrupts. When LOW indicates that the **nIRQ** and **nFIQ** inputs are to be synchronised by the ARM core. When HIGH disables this synchronisation for inputs that are already synchronous. |
| **LOCK** | O8 | Locked operation. When **LOCK** is HIGH, the processor is performing a "locked" memory access, and the memory controller must wait until **LOCK** goes LOW before allowing another device to access the memory. **LOCK** changes while **MCLK** is HIGH, and remains HIGH for the duration of the locked memory accesses. It is active only during the data swap (SWP) instruction. The timing of this signal may be modified by the use of **ALE** in a similar way to the address, please refer to the **ALE** description. This signal may also be driven to a high impedance state by driving **ABE** LOW. |
| **MCLK** | IC | Memory clock input. This clock times all ARM7DI memory accesses and internal operations. The clock has two distinct phases - *phase 1* in which **MCLK** is LOW and *phase 2* in which **MCLK** (and **nWAIT**) is HIGH. The clock may be stretched indefinitely in either phase to allow access to slow peripherals or memory. Alternatively, the **nWAIT** input may be used with a free running **MCLK** to achieve the same effect. |
| **nBW** | O8 | Not byte/word. This is an output signal used by the processor to indicate to the external memory system when a data transfer of a byte length is required. The signal is HIGH for word transfers and LOW for byte transfers and is valid for both read and write cycles. The signal will become valid during phase 2 of the cycle before the one in which the transfer will take place. It will remain stable throughout phase 1 of the transfer cycle. The timing of this signal may be modified by the use of **ALE** in a similar way to the address, please refer to the **ALE** description. This signal may also be driven to a high impedance state by driving **ABE** LOW. |
| **nCPI** | O4 | Not Coprocessor instruction. When ARM7DI executes a coprocessor instruction, it will take this output LOW and wait for a response from the coprocessor. The action taken will depend on this response, which the coprocessor signals on the **CPA** and **CPB** inputs. |
| **nENIN** | IC | NOT enable input. This signal may be used in conjunction with **nENOUT** to control the data bus during write cycles. See *Chapter 5.0 Memory Interface.* |
| **nENOUT** | O4 | Not enable output. During a data write cycle, this signal is driven low during phase 1, and remains low for the entire cycle. This may be used to aid arbitration in shared bus applications. See *Chapter 5.0 Memory Interface.* |
| **nEXEC** | O4 | Not executed. When HIGH indicates that the instruction in the execution unit is not being executed, because for example it has failed its condition code check. |
| **nFIQ** | IC | Not fast interrupt request. This is an interrupt request to the processor which causes it to be interrupted if taken LOW when the appropriate enable in the processor is active. The signal is level-sensitive and must be held LOW until a suitable response is received from the processor. **nFIQ** may be synchronous or asynchronous, depending on the state of **ISYNC**. |

**Table 1: Signal Description (Continued)**

| Name | Type | Description |
|---|---|---|
| **nIRQ** | IC | Not interrupt request. As **nFIQ**, but with lower priority. May be taken LOW to interrupt the processor when the appropriate enable is active. **nIRQ** may be synchronous or asynchronous, depending on the state of **ISYNC**. |
| **nM[4:0]** | O4 | Not processor mode. These are output signals which are the inverses of the internal status bits indicating the processor operation mode. |
| **nMREQ** | O4 | Not memory request. This signal, when LOW, indicates that the processor requires memory access during the following cycle. The signal becomes valid during phase 1, remaining valid through phase 2 of the cycle preceding that to which it refers. |
| **nOPC** | O8 | Not op-code fetch. When LOW this signal indicates that the processor is fetching an instruction from memory; when HIGH, data (if present) is being transferred. The signal becomes valid during phase 2 of the previous cycle, remaining valid through phase 1 of the referenced cycle. The timing of this signal may be modified by the use of **ALE** in a similar way to the address, please refer to the **ALE** description. This signal may also be driven to a high impedance state by driving **ABE** LOW. |
| **nRESET** | IC | Not reset. This is a level sensitive input signal which is used to start the processor from a known address. A LOW level will cause the instruction being executed to terminate abnormally. When **nRESET** becomes HIGH for at least one clock cycle, the processor will re-start from address 0. **nRESET** must remain LOW (and **nWAIT** must remain HIGH) for at least two clock cycles. During the LOW period the processor will perform dummy instruction fetches with the address incrementing from the point where reset was activated. The address will overflow to zero if **nRESET** is held beyond the maximum address limit. |
| **nRW** | O8 | Not read / write. When HIGH this signal indicates a processor write cycle; when LOW, a read cycle. It becomes valid during phase 2 of the cycle before that to which it refers, and remains valid to the end of phase 1 of the referenced cycle. The timing of this signal may be modified by the use of **ALE** in a similar way to the address, please refer to the **ALE** description. This signal may also be driven to a high impedance state by driving **ABE** LOW. |
| **nTDOEN** | O4 | Not **TDO** Enable. When LOW, this signal denotes that serial data is being driven out on the **TDO** output. **nTDOEN** would normally be used as an output enable for a **TDO** pin in a packaged part. |
| **nTRANS** | O8 | Not memory translate. When this signal is LOW it indicates that the processor is in user mode. It may be used to tell memory management hardware when translation of the addresses should be turned on, or as an indicator of non-user mode activity. The timing of this signal may be modified by the use of **ALE** in a similar way to the address, please refer to the **ALE** description. This signal may also be driven to a high impedance state by driving **ABE** LOW. |

**Table 1: Signal Description (Continued)**

| Name | Type | Description |
|---|---|---|
| **nTRST** | I | NOT Test Reset. Active-low reset signal for the boundary scan logic. This pin must be pulsed or driven low to achieve normal device operation, in addition to the normal device reset **(nRESET)**. The action of this and the other four boundary scan signals are described in more detail later in this document. |
| **nWAIT** | I | Not wait. When accessing slow peripherals, ARM7DI can be made to wait for an integer number of **MCLK** cycles by driving **nWAIT** LOW. Internally, **nWAIT** is ANDed with **MCLK** and must only change when **MCLK** is LOW. If **nWAIT** is not used it must be tied HIGH. |
| **PROG32** | I | 32 bit Program configuration. When this signal is HIGH the processor can fetch instructions from a 32 bit address space using address lines **A[31:0]**. When it is LOW the processor fetches instructions from a 26 bit address space using **A[25:0]**. In this latter configuration the address lines **A[31:26]** are not used for instruction fetches. Before changing **PROG32**, ensure that the processor is in a 26 bit mode, and is not about to write to an address in the range 0 to 0x1F (inclusive) in the next cycle. |
| **SEQ** | O4 | Sequential address. This output signal will become HIGH when the address of the next memory cycle will be related to that of the last memory access. The new address will either be the same as or 4 greater than the old one.<br><br>The signal becomes valid during phase 1 and remains so through phase 2 of the cycle before the cycle whose address it anticipates. It may be used, in combination with the low-order address lines, to indicate that the next cycle can use a fast memory mode (for example DRAM page mode) and / or to bypass the address translation system. |
| **TBE** | IC | Test Bus Enable. When driven LOW, **TBE** forces the data bus **D[31:0]**, the Address bus **A[31:0]**, plus **LOCK**, **nBW**, **nRW**, **nTRANS** and **nOPC** to high impedance. This is as if both **ABE** and **DBE** had both been driven LOW. However, **TBE** does not have an asssociated scan cell and so allows external signls to be driven high impedance during scan testing. Under normal operating conditions, **TBE** should be held HIGH at all times. |
| **TCK** | IC | Test Clock. |
| **TDI** | IC | Test Data Input. |
| **TDO** | O | Test Data Output. Output from the boundary scan logic. |
| **TMS** | IC | Test Mode Select. |
| **VDD** | P | Power supply. These connections provide power to the device. |
| **VSS** | P | Ground. These connections are the ground reference for all signals. |

**Table 1: Signal Description (Continued)**

**Key to Signal Types:**

**IC** - Input CMOS thresholds
**O4** - Output with INV4 driver
**P** - Power
**O8** - Output with INV8 driver

**Note:**

For a 0.8 µm ARM7DI:
INV4 driver has transistor sizes of p = 29.76 µm / 0.8 µm; N = 16.96 µm / 0.8 µm
INV8 driver has transistor sizes of p = 47.04 µm / 0.8 µm; N = 33.6 µm / 0.8 µm

# 3.0 Programmer's Model

ARM7DI supports a variety of operating configurations. Some are controlled by inputs and are known as the *hardware configurations*. Others may be controlled by software and these are known as *operating modes*.

## 3.1 Hardware Configuration Signals

The ARM7DI processor provides 3 hardware configuration signals which may be changed while the processor is running and which are discussed below.

### 3.1.1 Big and Little Endian

The **BIGEND** input sets whether the ARM7DI treats words in memory as being stored in Big Endian or Little Endian format. Memory is viewed as a linear collection of bytes numbered upwards from zero. Bytes 0 to 3 hold the first stored word, bytes 4 to 7 the second and so on.

In the Little Endian scheme the lowest numbered byte in a word is considered to be the least significant byte of the word and the highest numbered byte is the most significant. Byte 0 of the memory system should be connected to data lines 7 through 0 (**D[7:0]**) in this scheme.

**Little Endian**

| Higher Address | 31 24 | 23 16 | 15 8 | 7 0 | Word Address |
|---|---|---|---|---|---|
| | **11** | **10** | **9** | **8** | 8 |
| | **7** | **6** | **5** | **4** | 4 |
| | **3** | **2** | **1** | **0** | 0 |
| Lower Address | | | | | |

- Least significant byte is at lowest address
- Word is addressed by byte address of least significant byte

**Figure 4: Little Endian addresses of bytes within words**

In the Big Endian scheme the most significant byte of a word is stored at the lowest numbered byte and the least significant byte is stored at the highest numbered byte. Byte 0 of the memory system should therefore be connected to data lines 31 through 24 (**D[31:24]**). Load and store are the only instructions affected by the endian-ness: see *Section 4.7: Single data transfer (LDR, STR)* on page 42 for more details.

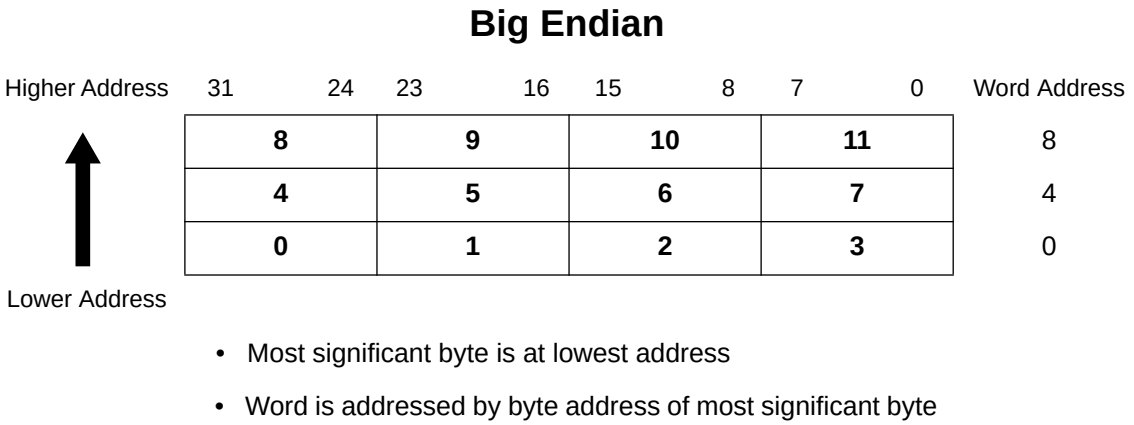



**Figure 5: Big Endian addresses of bytes within words**

### 3.1.2 Configuration Bits for Backward Compatibility

The other two inputs, **PROG32** and **DATA32** are used for backward compatibility with earlier ARM processors (see *12.0: Appendix - Backward Compatibility*) but should normally be set to 1. This configuration extends the address space to 32 bits, introduces major changes in the programmer's model as described below, and provides support for running existing 26 bit programs in the 32 bit environment. This mode is recommended for compatibility with future ARM processors and all new code should be written to use only the 32 bit operating modes.

Because the original ARM instruction set has been modified to accommodate 32 bit operation there are certain additional restrictions which programmers must be aware of. These are indicated in the text by the words *shall* and *shall not*. Reference should also be made to the *ARM Application Notes "Rules for ARM Code Writers"* and *"Notes for ARM Code Writers"*, available from your supplier.

## 3.2 Operating Mode Selection

ARM7DI has a 32 bit data bus and a 32 bit address bus. The processor supports *byte* (8 bit) and *word* (32 bit) data types, where words must be aligned to four byte boundaries. Instructions are exactly one word long, and data operations (eg ADD) are only performed on word quantities. Load and store operations can transfer either bytes or words.

ARM7DI supports six modes of operation:

(1) User mode (usr): the normal program execution state

(2) FIQ mode (fiq): designed to support a data transfer or channel process

(3) IRQ mode (irq): used for general purpose interrupt handling

(4) Supervisor mode (svc): a protected mode for the operating system

(5) Abort mode (abt): entered after a data or instruction prefetch abort

(6) Undefined mode (und): entered when an undefined instruction is executed

Mode changes may be made under software control or may be brought about by external interrupts or exception processing. Most application programs will execute in User mode. The other modes, known as *privileged modes*, will be entered to service interrupts or exceptions or to access protected resources.

## 3.3 Registers

The processor has a total of 37 registers made up of 31 general 32 bit registers and 6 status registers. At any one time 16 general registers (R0 to R15) and one or two status registers are visible to the programmer. The visible registers depend on the processor mode. The other registers, known as the *banked registers*, are switched in to support IRQ, FIQ, Supervisor, Abort and Undefined mode processing. *Figure 6: Register Organisation* shows how the registers are arranged, with the banked registers shaded.

In all modes 16 registers, R0 to R15, are directly accessible. All registers except R15 are general purpose and may be used to hold data or address values. Register R15 holds the Program Counter (PC). When R15 is read, bits [1:0] are zero and bits [31:2] contain the PC. A seventeenth register (the CPSR - Current Program Status Register) is also accessible. It contains condition code flags and the current mode bits and may be thought of as an extension to the PC.

R14 is used as the subroutine link register and receives a copy of R15 when a Branch and Link instruction is executed. It may be treated as a general purpose register at all other times. R14_svc, R14_irq, R14_fiq, R14_abt and R14_und are used similarly to hold the return values of R15 when interrupts and exceptions arise, or when Branch and Link instructions are executed within interrupt or exception routines.

## General Registers and Program Counter Modes

| User32 | FIQ32 | Supervisor32 | Abort32 | IRQ32 | Undefined32 |
|---|---|---|---|---|---|
| R0 | R0 | R0 | R0 | R0 | R0 |
| R1 | R1 | R1 | R1 | R1 | R1 |
| R2 | R2 | R2 | R2 | R2 | R2 |
| R3 | R3 | R3 | R3 | R3 | R3 |
| R4 | R4 | R4 | R4 | R4 | R4 |
| R5 | R5 | R5 | R5 | R5 | R5 |
| R6 | R6 | R6 | R6 | R6 | R6 |
| R7 | R7 | R7 | R7 | R7 | R7 |
| R8 | R8_fiq | R8 | R8 | R8 | R8 |
| R9 | R9_fiq | R9 | R9 | R9 | R9 |
| R10 | R10_fiq | R10 | R10 | R10 | R10 |
| R11 | R11_fiq | R11 | R11 | R11 | R11 |
| R12 | R12_fiq | R12 | R12 | R12 | R12 |
| R13 | R13_fiq | R13_svc | R13_abt | R13_irq | R13_und |
| R14 | R14_fiq | R14_svc | R14_abt | R14_irq | R14_und |
| R15 (PC) | R15 (PC) | R15 (PC) | R15 (PC) | R15 (PC) | R15 (PC) |

## Program Status Registers

| | | | | | |
|---|---|---|---|---|---|
| CPSR | CPSR | CPSR | CPSR | CPSR | CPSR |
| | SPSR_fiq | SPSR_svc | SPSR_abt | SPSR_irq | SPSR_und |

**Figure 6: Register Organisation**

FIQ mode has seven banked registers mapped to R8-14 (R8_fiq-R14_fiq). Many FIQ programs will not need to save any registers. User mode, IRQ mode, Supervisor mode, Abort mode and Undefined mode each have two banked registers mapped to R13 and R14. The two banked registers allow these modes to each have a private stack pointer and link register. Supervisor, IRQ, Abort and Undefined mode programs which require more than these two banked registers are expected to save some or all of the caller's registers (R0 to R12) on their respective stacks. They are then free to use these registers which they will restore before returning to the caller. In addition there are also five SPSRs (Saved Program Status Registers) which are loaded with the CPSR when an exception occurs. There is one SPSR for each privileged mode.

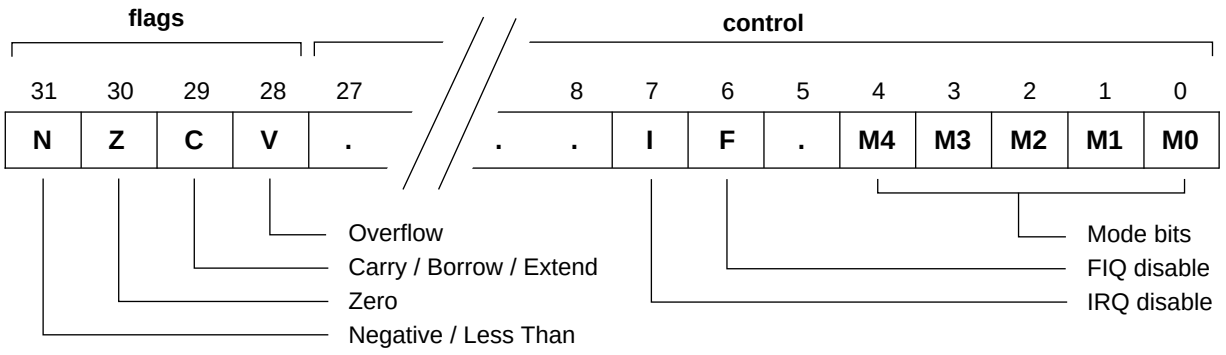



**Figure 7: Format of the Program Status Registers (PSRs)**

The format of the Program Status Registers is shown in *Figure 7: Format of the Program Status Registers (PSRs)*. The N, Z, C and V bits are the *condition code flags*. The condition code flags in the CPSR may be changed as a result of arithmetic and logical operations in the processor and may be tested by all instructions to determine if the instruction is to be executed.

The I and F bits are the *interrupt disable bits*. The I bit disables IRQ interrupts when it is set and the F bit disables FIQ interrupts when it is set. The M0, M1, M2, M3 and M4 bits (M[4:0]) are the *mode bits*, and these determine the mode in which the processor operates. The interpretation of the mode bits is shown in *Table 2: The Mode Bits*. Not all bit combinations define a valid processor mode. Only those explicitly described shall be used. The user should be aware that if any illegal value is programmed into the mode bits, M[4:0], the processor will enter an unrecoverable state. If this occurs, reset should be applied.

The bottom 28 bits of a PSR (incorporating I, F and M[4:0]) are known collectively as the *control bits*. These will change when an exception arises and in addition can be manipulated by software when the processor is in a privileged mode. Unused bits in the PSRs are reserved and their state shall be preserved when changing the flag or control bits. Programs shall not rely on specific values from the reserved bits when checking the PSR status, since they may read as one or zero in future processors.

| M[4:0] | Mode | Accessible register set | |
|---|---|---|---|
| 10000 | User | PC, R14..R0 | CPSR |
| 10001 | FIQ | PC, R14_fiq..R8_fiq, R7..R0 | CPSR, SPSR_fiq |
| 10010 | IRQ | PC, R14_irq..R13_irq, R12..R0 | CPSR, SPSR_irq |
| 10011 | Supervisor | PC, R14_svc..R13_svc, R12..R0 | CPSR, SPSR_svc |
| 10111 | Abort | PC, R14_abt..R13_abt, R12..R0 | CPSR, SPSR_abt |
| 11011 | Undefined | PC, R14_und..R13_und, R12..R0 | CPSR, SPSR_und |

**Table 2: The Mode Bits**

## 3.4 Exceptions

Exceptions arise whenever there is a need for the normal flow of program execution to be broken, so that (for example) the processor can be diverted to handle an interrupt from a peripheral. The processor state just prior to handling the exception must be preserved so that the original program can be resumed when the exception routine has completed. Many exceptions may arise at the same time.

ARM7DI handles exceptions by making use of the banked registers to save state. The old PC and CPSR contents are copied into the appropriate R14 and SPSR and the PC and mode bits in the CPSR bits are forced to a value which depends on the exception. Interrupt disable flags are set where required to prevent otherwise unmanageable nestings of exceptions. In the case of a re-entrant interrupt handler, R14 and the SPSR should be saved onto a stack in main memory before re-enabling the interrupt; when transferring the SPSR register to and from a stack, it is important to transfer the whole 32 bit value, and not just the flag or control fields. When multiple exceptions arise simultaneously, a fixed priority determines the order in which they are handled. This is listed later in *Section 3.4.7: Exception Priorities* on page 23.

### 3.4.1 FIQ

The FIQ (Fast Interrupt reQuest) exception is externally generated by taking the **nFIQ** input LOW. This input can except either synchronous or asynchronous transitions, depending on the state of the **ISYNC** input signal. When **ISYNC** is LOW, **nFIQ** (and **nIRQ**) are considered asynchronous, and a cycle delay for synchronisation is incurred before the interrupt can affect the processor flow. 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 (thus minimising the overhead of context switching). The FIQ exception may be disabled by setting the F flag in the CPSR (but note that this is not possible from User mode). If the F flag is clear, ARM7DI checks for a LOW level on the output of the FIQ synchroniser at the end of each instruction.

When a FIQ is detected, ARM7DI:

(1) Saves the address of the next instruction to be executed plus 4 in R14_fiq; saves CPSR in SPSR_fiq

(2) Forces M[4:0]=10001 (FIQ mode) and sets the F and I bits in the CPSR

(3) Forces the PC to fetch the next instruction from address 0x1C

To return normally from FIQ, use SUBS PC, R14_fiq,#4 which will restore both the PC (from R14) and the CPSR (from SPSR_fiq) and resume execution of the interrupted code.

### 3.4.2 IRQ

The IRQ (Interrupt ReQuest) exception is a normal interrupt caused by a LOW level on the **nIRQ** input. It has a lower priority than FIQ, and is masked out when a FIQ sequence is entered. Its effect may be masked out at any time by setting the I bit in the CPSR (but note that this is not possible from User mode). If the I flag is clear, ARM7DI checks for a LOW level on the output of the IRQ synchroniser at the end of each instruction. Note that **nIRQ** may also have synchronous or asynchronous timing, depending on the state of the **ISYNC** input. When an IRQ is detected, ARM7DI:

(1) Saves the address of the next instruction to be executed plus 4 in R14_irq; saves CPSR in SPSR_irq

(2) Forces M[4:0]=10010 (IRQ mode) and sets the I bit in the CPSR

(3) Forces the PC to fetch the next instruction from address 0x18

To return normally from IRQ, use SUBS PC,R14_irq,#4 which will restore both the PC and the CPSR and resume execution of the interrupted code.

### 3.4.3 Abort

An abort can be signalled by the external **ABORT** input. **ABORT** indicates that the current memory access cannot be completed. For instance, in a virtual memory system the data corresponding to the current address may have been moved out of memory onto a disc, and considerable processor activity may be required to recover the data before the access can be performed successfully. ARM7DI checks for aborts during memory access cycles. When successfully aborted ARM7DI will respond in one of two ways:

(1) If the abort occurred during an instruction prefetch (a *Prefetch Abort*), the prefetched instruction is marked as invalid but the abort exception does not occur immediately. If the instruction is not executed, for example as a result of a branch being taken while it is in the pipeline, no abort will occur. An abort will take place if the instruction reaches the head of the pipeline and is about to be executed.

(2) If the abort occurred during a data access (a *Data Abort*), the action depends on the instruction type.

(a) Single data transfer instructions (LDR, STR) will write back modified base registers and the Abort handler must be aware of this.

(b) The swap instruction (SWP) is aborted as though it had not executed, though externally the read access may take place.

(c) Block data transfer instructions (LDM, STM) complete, and if write-back is set, the base is updated. If the instruction would normally have overwritten the base with data (i.e. LDM with the base in the transfer list), this overwriting is prevented. All register overwriting is prevented after the Abort is indicated, which means in particular that R15 (which is always last to be transferred) is preserved in an aborted LDM instruction.

When either a prefetch or data abort occurs, ARM7DI:

(1) Saves the address of the aborted instruction plus 4 (for prefetch aborts) or 8 (for data aborts) in R14_abt; saves CPSR in SPSR_abt.

(2) Forces M[4:0]=10111 (Abort mode) and sets the I bit in the CPSR.

(3) Forces the PC to fetch the next instruction from either address 0x0C (prefetch abort) or address 0x10 (data abort).

To return after fixing the reason for the abort, use SUBS PC,R14_abt,#4 (for a prefetch abort) or SUBS PC,R14_abt,#8 (for a data abort). This will restore both the PC and the CPSR and retry the aborted instruction.

The abort mechanism allows a *demand paged virtual memory system* to be implemented when suitable memory management software is available. The processor is allowed to generate arbitrary addresses, and when the data at an address is unavailable the MMU signals an abort. The processor traps into system software which must work out the cause of the abort, make the requested data available, and retry the aborted instruction. The application program needs no knowledge of the amount of memory available to it, nor is its state in any way affected by the abort.

### 3.4.4 Software interrupt

The software interrupt instruction (SWI) is used for getting into Supervisor mode, usually to request a particular supervisor function. When a SWI is executed, ARM7DI performs the following:

(1) Saves the address of the SWI instruction plus 4 in R14_svc; saves CPSR in SPSR_svc

(2) Forces M[4:0]=10011 (Supervisor mode) and sets the I bit in the CPSR

(3) Forces the PC to fetch the next instruction from address 0x08

To return from a SWI, use MOVS PC,R14_svc. This will restore the PC and CPSR and return to the instruction following the SWI.

### 3.4.5 Undefined instruction trap

When the ARM7DI comes across an instruction which it cannot handle (see *Chapter 4.0: Instruction Set*), it offers it to any coprocessors which may be present. If a coprocessor can perform this instruction but is busy at that time, ARM7DI will wait until the coprocessor is ready or until an interrupt occurs. If no coprocessor can handle the instruction then ARM7DI will take the undefined instruction trap.

The trap may be used for software emulation of a coprocessor in a system which does not have the coprocessor hardware, or for general purpose instruction set extension by software emulation.

When ARM7DI takes the undefined instruction trap it performs the following:

(1) Saves the address of the Undefined or coprocessor instruction plus 4 in R14_und; saves CPSR in SPSR_und.

(2) Forces M[4:0]=11011 (Undefined mode) and sets the I bit in the CPSR

(3) Forces the PC to fetch the next instruction from address 0x04

To return from this trap after emulating the failed instruction, use MOVS PC,R14_und. This will restore the CPSR and return to the instruction following the undefined instruction.

### 3.4.6 Vector Summary

| Address | Exception | Mode on entry |
|---|---|---|
| 0x00000000 | Reset | Supervisor |
| 0x00000004 | Undefined instruction | Undefined |
| 0x00000008 | Software interrupt | Supervisor |
| 0x0000000C | Abort (prefetch) | Abort |
| 0x00000010 | Abort (data) | Abort |
| 0x00000014 | -- reserved -- | -- |
| 0x00000018 | IRQ | IRQ |
| 0x0000001C | FIQ | FIQ |

**Table 3: Vector Summary**

These are byte addresses, and will normally contain a branch instruction pointing to the relevant routine.

The FIQ routine might reside at 0x1C onwards, and thereby avoid the need for (and execution time of) a branch instruction.

### 3.4.7 Exception Priorities

When multiple exceptions arise at the same time, a fixed priority system determines the order in which they will be handled:

(1) Reset (highest priority)

(2) Data abort

(3) FIQ

(4) IRQ

(5) Prefetch abort

(6) Undefined Instruction, Software interrupt (lowest priority)

Note that not all exceptions can occur at once. Undefined instruction and software interrupt are mutually exclusive since they each correspond to particular (non-overlapping) decodings of the current instruction.

If a data abort occurs at the same time as a FIQ, and FIQs are enabled (i.e. the F flag in the CPSR is clear), ARM7DI will enter the data abort handler and then immediately proceed to the FIQ vector. A normal return from FIQ will cause the data abort handler to resume execution. Placing data abort at a higher priority than FIQ is necessary to ensure that the transfer error does not escape detection; the time for this exception entry should be added to worst case FIQ latency calculations.

### 3.4.8 Interrupt Latencies

The worst case latency for FIQ, assuming that it is enabled, consists of the longest time the request can take to pass through the synchroniser (*Tsyncmax* if asynchronous), plus the time for the longest instruction to complete (*Tldm*, the longest instruction is an LDM which loads all the registers including the PC), plus the time for the data abort entry (*Texc*), plus the time for FIQ entry (*Tfiq*). At the end of this time ARM7DI will be executing the instruction at 0x1C.

*Tsyncmax* is 3 processor cycles, *Tldm* is 20 cycles, *Texc* is 3 cycles, and *Tfiq* is 2 cycles. The total time is therefore 28 processor cycles. This is just over 1.4 microseconds in a system which uses a continuous 20 MHz processor clock. The maximum IRQ latency calculation is similar, but must allow for the fact that FIQ has higher priority and could delay entry into the IRQ handling routine for an arbitrary length of time. The minimum latency for FIQ or IRQ consists of the shortest time the request can take through the synchroniser (*Tsyncmin*) plus *Tfiq*. This is 4 processor cycles.

## 3.5 Reset

When the **nRESET** signal goes LOW, ARM7DI abandons the executing instruction and then continues to fetch instructions from incrementing word addresses.

When **nRESET** goes HIGH again, ARM7DI does the following:

(1) Overwrites R14_svc and SPSR_svc by copying the current values of the PC and CPSR into them. The value of the saved PC and CPSR is not defined.

(2) Forces M[4:0]=10011 (Supervisor mode) and sets the I and F bits in the CPSR.

(3) Forces the PC to fetch the next instruction from address 0x00

# 4.0 Instruction Set

## 4.1 Instruction Set Summary

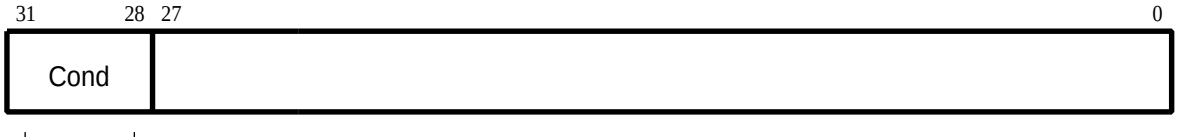
A summary of the ARM7DI instruction set is shown in *Figure 8: Instruction Set Summary*.

Note: some instruction codes are not defined but do not cause the Undefined instruction trap to be taken, for instance a Multiply instruction with bit 6 changed to a 1. These instructions shall not be used, as their action may change in future ARM implementations.

| 31 – 28 | 27 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 – 16 | 15 – 12 | 11 – 8 | 7 – 5 | 4 | 3 – 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Cond | 0 0 | I | Opcode | | | | S | Rn | Rd | Operand 2 | | | | **Data Processing PSR Transfer** |
| Cond | 0 0 0 0 0 0 | | | | | A | S | Rd | Rn | Rs | 1 0 0 1 | | Rm | **Multiply** |
| Cond | 0 0 0 1 0 | | | | B | 0 0 | | Rn | Rd | 0 0 0 0 | 1 0 0 1 | | Rm | **Single Data Swap** |
| Cond | 0 1 | I | P | U | B | W | L | Rn | Rd | offset | | | | **Single Data Transfer** |
| Cond | 0 1 1 | | XXXXXXXXXXXXXXXXXXXX | | | | | | | | | 1 | XXXX | **Undefined** |
| Cond | 1 0 0 | | P | U | S | W | L | Rn | Register List | | | | | **Block Data Transfer** |
| Cond | 1 0 1 | | L | offset | | | | | | | | | | **Branch** |
| Cond | 1 1 0 | | P | U | N | W | L | Rn | CRd | CP# | offset | | | **Coproc Data Transfer** |
| Cond | 1 1 1 0 | | CP Opc | | | | | CRn | CRd | CP# | CP | 0 | CRm | **Coproc Data Operation** |
| Cond | 1 1 1 0 | | CP Opc | | | | L | CRn | Rd | CP# | CP | 1 | CRm | **Coproc Register Transfer** |
| Cond | 1 1 1 1 | | ignored by processor | | | | | | | | | | | **Software Interrupt** |

**Figure 8: Instruction Set Summary**

## 4.2 The Condition Field




**Condition field**

0000 = EQ - Z set (equal)
0001 = NE - Z clear (not equal)
0010 = CS - C set (unsigned higher or same)
0011 = CC - C clear (unsigned lower)
0100 = MI - N set (negative)
0101 = PL - N clear (positive or zero)
0110 = VS - V set (overflow)
0111 = VC - V clear (no overflow)
1000 = HI - C set and Z clear (unsigned higher)
1001 = LS - C clear or Z set (unsigned lower or same)
1010 = GE - N set and V set, or N clear and V clear (greater or equal)
1011 = LT - N set and V clear, or N clear and V set (less than)
1100 = GT - Z clear, and either N set and V set, or N clear and V clear (greater than)
1101 = LE - Z set, or N set and V clear, or N clear and V set (less than or equal)
1110 = AL - always
1111 = NV - never

**Figure 9: Condition Codes**

All ARM7DI instructions are conditionally executed, which means that their execution may or may not take place depending on the values of the N, Z, C and V flags in the CPSR. The condition encoding is shown in *Figure 9: Condition Codes*.

If the *always* (AL) condition is specified, the instruction will be executed irrespective of the flags. The *never* (NV) class of condition codes shall not be used as they will be redefined in future variants of the ARM architecture. If a NOP is required it is suggested that MOV R0,R0 be used. The assembler treats the absence of a condition code as though *always* had been specified.

The other condition codes have meanings as detailed in *Figure 9: Condition Codes*, for instance code 0000 (EQual) causes the instruction to be executed only if the Z flag is set. This would correspond to the case where a compare (CMP) instruction had found the two operands to be equal. If the two operands were different, the compare instruction would have cleared the Z flag and the instruction will not be executed.

## 4.3 Branch and Branch with link (B, BL)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 10: Branch Instructions.*

Branch instructions contain a signed 2's complement 24 bit offset. This is shifted left two bits, sign extended to 32 bits, and added to the PC. The instruction can therefore specify a branch of +/- 32Mbytes. The branch offset must take account of the prefetch operation, which causes the PC to be 2 words (8 bytes) ahead of the current instruction.

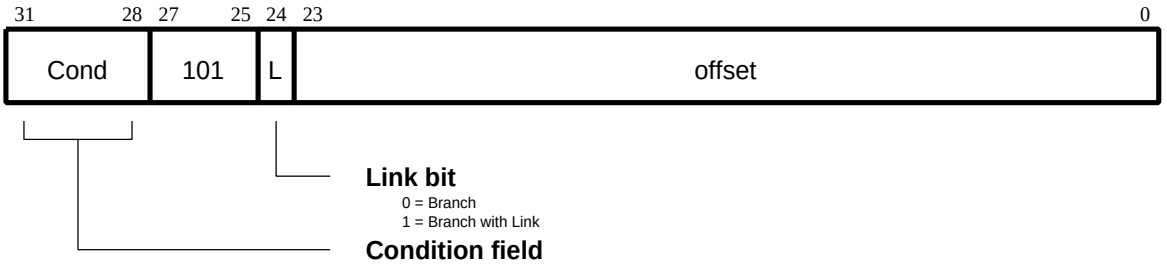



**Figure 10: Branch Instructions**

Branches beyond +/- 32Mbytes must use an offset or absolute destination which has been previously loaded into a register. In this case the PC should be manually saved in R14 if a Branch with Link type operation is required.

### 4.3.1 The link bit

Branch with Link (BL) writes the old PC into the link register (R14) of the current bank. The PC value written into R14 is adjusted to allow for the prefetch, and contains the address of the instruction following the branch and link instruction. Note that the CPSR is not saved with the PC.

To return from a routine called by Branch with Link use MOV PC,R14 if the link register is still valid or LDM Rn!,{..PC} if the link register has been saved onto a stack pointed to by Rn.

### 4.3.2 Instruction Cycle Times

Branch and Branch with Link instructions take 2S + 1N incremental cycles, where S and N are as defined in *Section 5.1: Cycle types* on page 71.

### 4.3.3 Assembler syntax

**B{L}{cond} <expression>**

{L} is used to request the Branch with Link form of the instruction. If absent, R14 will not be affected by the instruction.

{cond} is a two-character mnemonic as shown in *Figure 9: Condition Codes* (EQ, NE, VS etc). If absent then AL (ALways) will be used.

<expression> is the destination. The assembler calculates the offset.

Items in {} are optional. Items in <> must be present.

### 4.3.4 Examples

```
here  BAL      here     ; assembles to 0xEAFFFFFE (note effect of PC offset)
      B        there    ; ALways condition used as default

      CMP      R1,#0    ; compare R1 with zero and branch to fred if R1
      BEQ      fred     ; was zero otherwise continue to next instruction

      BL       sub+ROM  ; call subroutine at computed address

      ADDS     R1,#1    ; add 1 to register 1, setting CPSR flags on the
      BLCC     sub      ; result then call subroutine if the C flag is clear,
                        ; which will be the case unless R1 held 0xFFFFFFFF
```

## 4.4 Data processing

The instruction is only executed if the condition is true, defined at the beginning of this chapter. The instruction encoding is shown in *Figure 11: Data Processing Instructions.*

The instruction produces a result by performing a specified arithmetic or logical operation on one or two operands. The first operand is always a register (Rn). The second operand may be a shifted register (Rm) or a rotated 8 bit immediate value (Imm) according to the value of the I bit in the instruction. The condition codes in the CPSR may be preserved or updated as a result of this instruction, according to the value of the S bit in the instruction. Certain operations (TST, TEQ, CMP, CMN) do not write the result to Rd. They are used only to perform tests and to set the condition codes on the result and always have the S bit set. The instructions and their effects are listed in *Table 4: ARM Data Processing Instructions.*

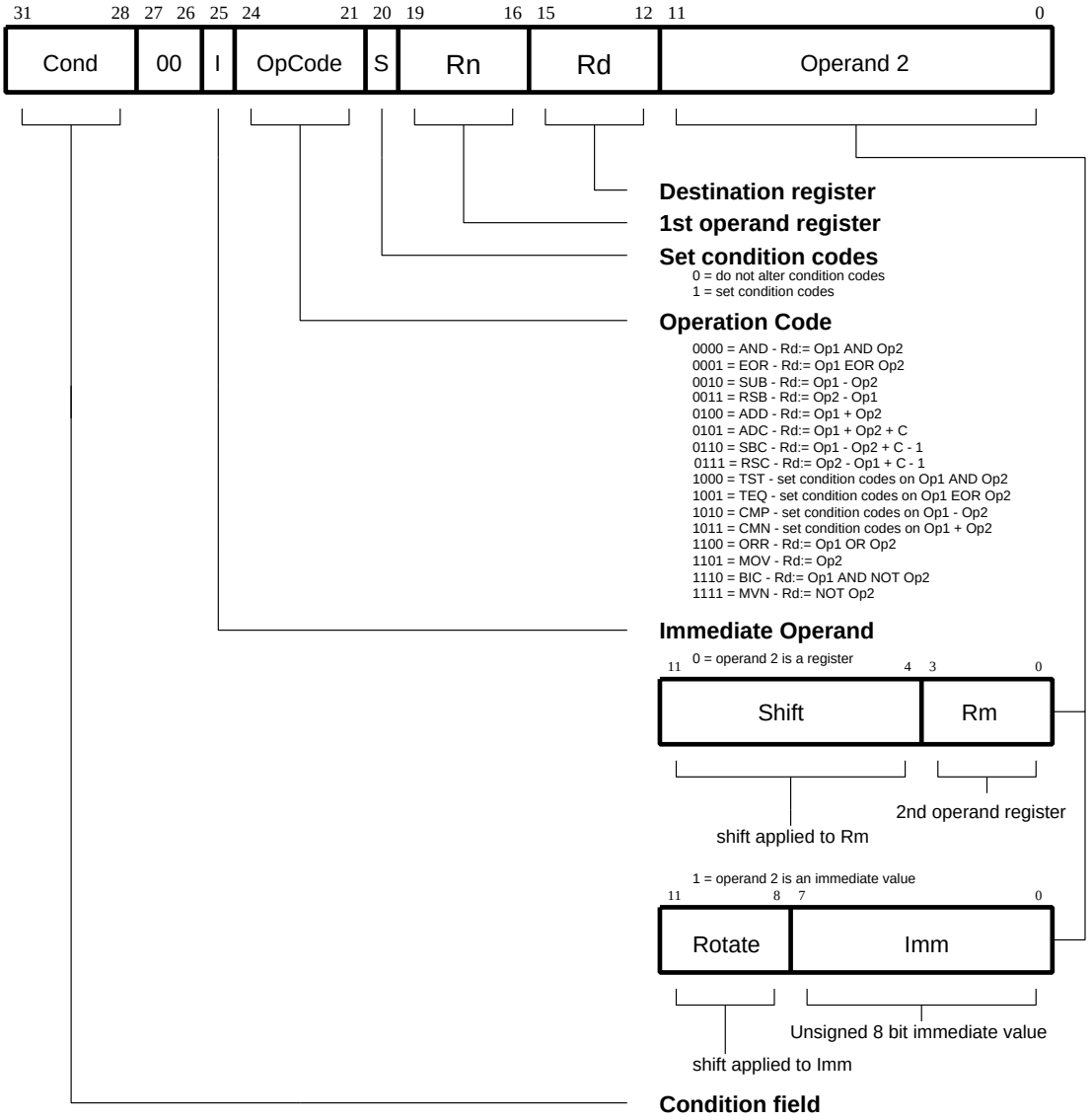



**Figure 11: Data Processing Instructions**

### 4.4.1 CPSR flags

The data processing operations may be classified as logical or arithmetic. The logical operations (AND, EOR, TST, TEQ, ORR, MOV, BIC, MVN) perform the logical action on all corresponding bits of the operand or operands to produce the result. If the S bit is set (and Rd is not R15, see below) the V flag in the CPSR will be unaffected, the C flag will be set to the carry out from the barrel shifter (or preserved when the shift operation is LSL #0), the Z flag will be set if and only if the result is all zeros, and the N flag will be set to the logical value of bit 31 of the result.

| Assembler Mnemonic | OpCode | Action |
|---|---|---|
| AND | 0000 | operand1 AND operand2 |
| EOR | 0001 | operand1 EOR operand2 |
| SUB | 0010 | operand1 - operand2 |
| RSB | 0011 | operand2 - operand1 |
| ADD | 0100 | operand1 + operand2 |
| ADC | 0101 | operand1 + operand2 + carry |
| SBC | 0110 | operand1 - operand2 + carry - 1 |
| RSC | 0111 | operand2 - operand1 + carry - 1 |
| TST | 1000 | as AND, but result is not written |
| TEQ | 1001 | as EOR, but result is not written |
| CMP | 1010 | as SUB, but result is not written |
| CMN | 1011 | as ADD, but result is not written |
| ORR | 1100 | operand1 OR operand2 |
| MOV | 1101 | operand2 (operand1 is ignored) |
| BIC | 1110 | operand1 AND NOT operand2 (Bit clear) |
| MVN | 1111 | NOT operand2 (operand1 is ignored) |

**Table 4: ARM Data Processing Instructions**

The arithmetic operations (SUB, RSB, ADD, ADC, SBC, RSC, CMP, CMN) treat each operand as a 32 bit integer (either unsigned or 2's complement signed, the two are equivalent). If the S bit is set (and Rd is not R15) the V flag in the CPSR will be set if an overflow occurs into bit 31 of the result; this may be ignored if the operands were considered unsigned, but warns of a possible error if the operands were 2's complement signed. The C flag will be set to the carry out of bit 31 of the ALU, the Z flag will be set if and only if the result was zero, and the N flag will be set to the value of bit 31 of the result (indicating a negative result if the operands are considered to be 2's complement signed).

### 4.4.2 Shifts

When the second operand is specified to be a shifted register, the operation of the barrel shifter is controlled by the Shift field in the instruction. This field indicates the type of shift to be performed (logical left or right, arithmetic right or rotate right). The amount by which the register should be shifted may be contained in an immediate field in the instruction, or in the bottom byte of another register (other than R15). The encoding for the different shift types is shown in *Figure 12: ARM Shift Operations.*

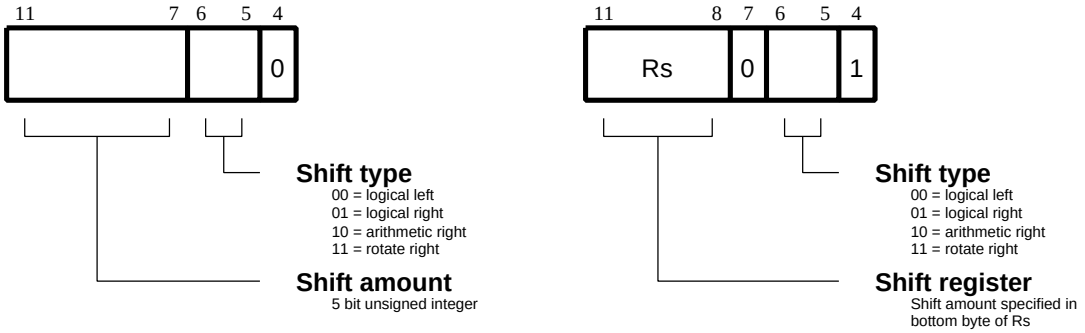



**Figure 12: ARM Shift Operations**

**Instruction specified shift amount**

When the shift amount is specified in the instruction, it is contained in a 5 bit field which may take any value from 0 to 31. A logical shift left (LSL) takes the contents of Rm and moves each bit by the specified amount to a more significant position. The least significant bits of the result are filled with zeros, and the high bits of Rm which do not map into the result are discarded, except that the least significant discarded bit becomes the shifter carry output which may be latched into the C bit of the CPSR when the ALU operation is in the logical class (see above). For example, the effect of LSL #5 is shown in *Figure 13: Logical Shift Left*.

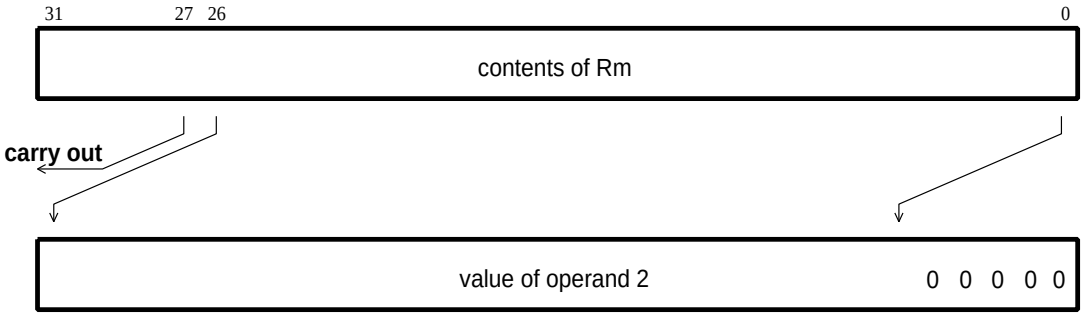



**Figure 13: Logical Shift Left**

Note that LSL #0 is a special case, where the shifter carry out is the old value of the CPSR C flag. The contents of Rm are used directly as the second operand.

A logical shift right (LSR) is similar, but the contents of Rm are moved to less significant positions in the result. LSR #5 has the effect shown in *Figure 14: Logical Shift Right*.

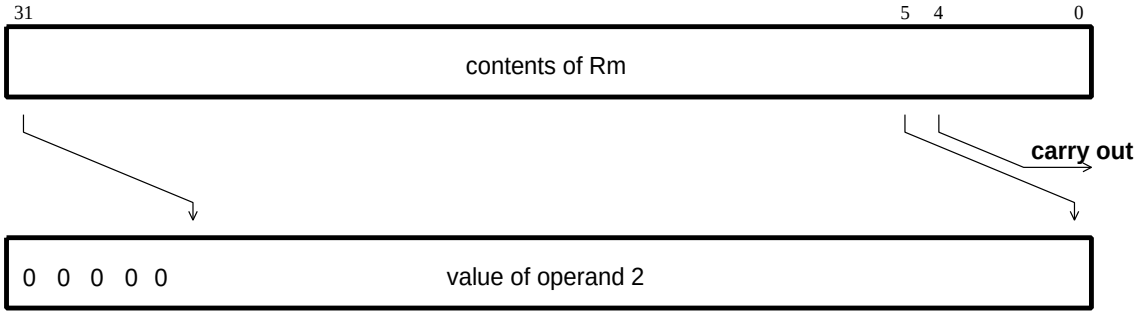



**Figure 14: Logical Shift Right**

The form of the shift field which might be expected to correspond to LSR #0 is used to encode LSR #32, which has a zero result with bit 31 of Rm as the carry output. Logical shift right zero is redundant as it is the same as logical shift left zero, so the assembler will convert LSR #0 (and ASR #0 and ROR #0) into LSL #0, and allow LSR #32 to be specified.

An arithmetic shift right (ASR) is similar to logical shift right, except that the high bits are filled with bit 31 of Rm instead of zeros. This preserves the sign in 2's complement notation. For example, ASR #5 is shown in *Figure 15: Arithmetic Shift Right*.

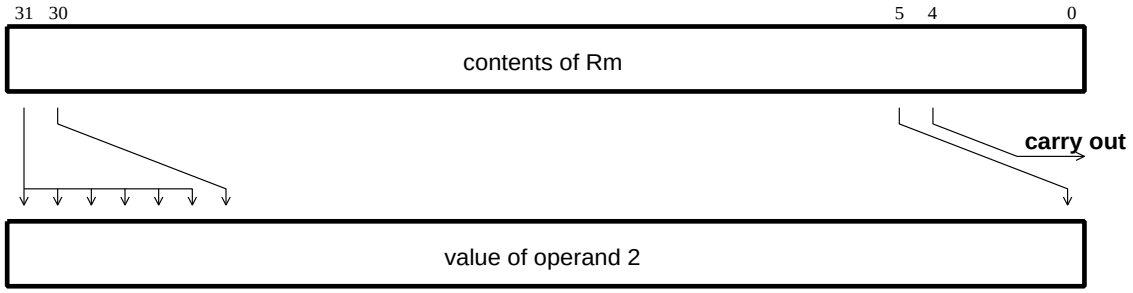



**Figure 15: Arithmetic Shift Right**

The form of the shift field which might be expected to give ASR #0 is used to encode ASR #32. Bit 31 of Rm is again used as the carry output, and each bit of operand 2 is also equal to bit 31 of Rm. The result is therefore all ones or all zeros, according to the value of bit 31 of Rm.

Rotate right (ROR) operations reuse the bits which 'overshoot' in a logical shift right operation by reintroducing them at the high end of the result, in place of the zeros used to fill the high end in logical right operations. For example, ROR #5 is shown in *Figure 16: Rotate Right*.

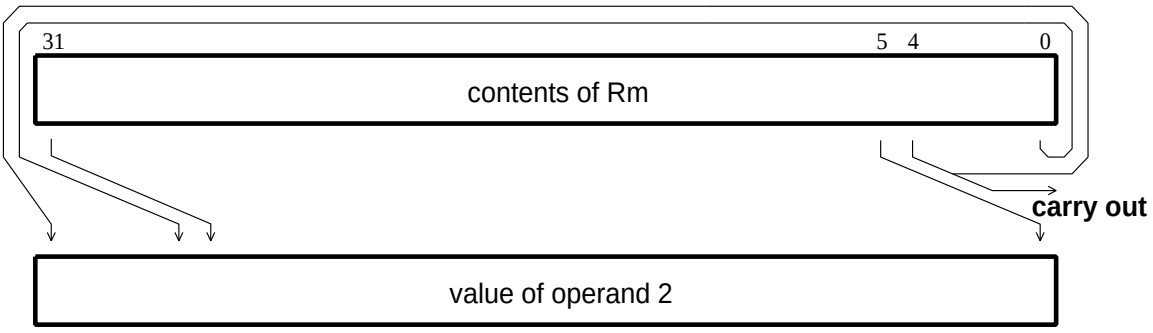



**Figure 16: Rotate Right**

The form of the shift field which might be expected to give ROR #0 is used to encode a special function of the barrel shifter, rotate right extended (RRX). This is a rotate right by one bit position of the 33 bit quantity formed by appending the CPSR C flag to the most significant end of the contents of Rm as shown in *Figure 17: Rotate Right Extended.*

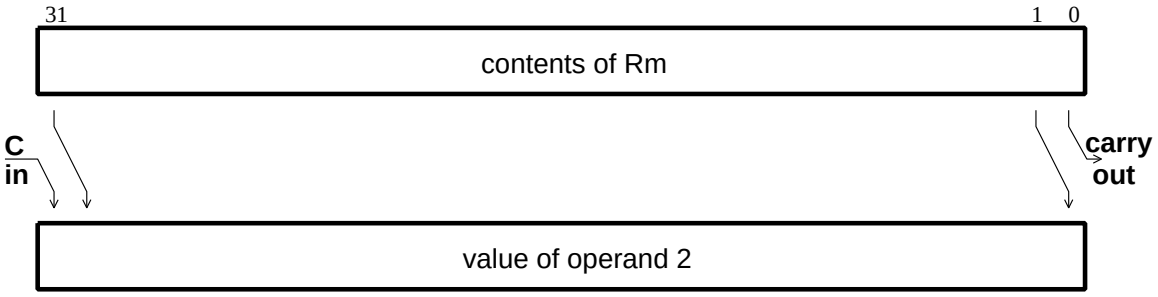



**Figure 17: Rotate Right Extended**

**Register specified shift amount**

Only the least significant byte of the contents of Rs is used to determine the shift amount. Rs can be any general register other than R15.

If this byte is zero, the unchanged contents of Rm will be used as the second operand, and the old value of the CPSR C flag will be passed on as the shifter carry output.

If the byte has a value between 1 and 31, the shifted result will exactly match that of an instruction specified shift with the same value and shift operation.

If the value in the byte is 32 or more, the result will be a logical extension of the shift described above:

(1) LSL by 32 has result zero, carry out equal to bit 0 of Rm.

(2) LSL by more than 32 has result zero, carry out zero.

(3) LSR by 32 has result zero, carry out equal to bit 31 of Rm.

(4) LSR by more than 32 has result zero, carry out zero.

(5) ASR by 32 or more has result filled with and carry out equal to bit 31 of Rm.

(6) ROR by 32 has result equal to Rm, carry out equal to bit 31 of Rm.

(7) ROR by n where n is greater than 32 will give the same result and carry out as ROR by n-32; therefore repeatedly subtract 32 from n until the amount is in the range 1 to 32 and see above.

Note that the zero in bit 7 of an instruction with a register controlled shift is compulsory; a one in this bit will cause the instruction to be a multiply or undefined instruction.

### 4.4.3 Immediate operand rotates

The immediate operand rotate field is a 4 bit unsigned integer which specifies a shift operation on the 8 bit immediate value. This value is zero extended to 32 bits, and then subject to a rotate right by twice the value in the rotate field. This enables many common constants to be generated, for example all powers of 2.

### 4.4.4 Writing to R15

When Rd is a register other than R15, the condition code flags in the CPSR may be updated from the ALU flags as described above.

When Rd is R15 and the S flag in the instruction is not set the result of the operation is placed in R15 and the CPSR is unaffected.

When Rd is R15 and the S flag is set the result of the operation is placed in R15 and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of instruction shall not be used in User mode.

### 4.4.5 Using R15 as an operand

If R15 (the PC) is used as an operand in a data processing instruction the register is used directly.

The PC value will be the address of the instruction, plus 8 or 12 bytes due to instruction prefetching. If the shift amount is specified in the instruction, the PC will be 8 bytes ahead. If a register is used to specify the shift amount the PC will be 12 bytes ahead.

### 4.4.6 TEQ, TST, CMP & CMN opcodes

These instructions do not write the result of their operation but do set flags in the CPSR. An assembler shall always set the S flag for these instructions even if it is not specified in the mnemonic.

The TEQP form of the instruction used in earlier processors shall not be used in the 32 bit modes, the PSR transfer operations should be used instead. If used in these modes, its effect is to move SPSR_<mode> to CPSR if the processor is in a privileged mode and to do nothing if in User mode.

### 4.4.7 Instruction Cycle Times

Data Processing instructions vary in the number of incremental cycles taken as follows:

| | |
|---|---|
| Normal Data Processing | 1S |
| Data Processing with register specified shift | 1S + 1I |

Data Processing with PC written 2S + 1NData Processing with register specified shift and PC written 2S + 1N + 1I

S, N and I are as defined in *Section 5.1: Cycle types* on page 71.

### 4.4.8 Assembler syntax

(1) MOV,MVN - single operand instructions

**<opcode>{cond}{S} Rd,<Op2>**

(2) CMP,CMN,TEQ,TST - instructions which do not produce a result.

**<opcode>{cond} Rn,<Op2>**

(3) AND,EOR,SUB,RSB,ADD,ADC,SBC,RSC,ORR,BIC

**<opcode>{cond}{S} Rd,Rn,<Op2>**

where <Op2> is **Rm{,<shift>}** or,**<#expression>**

{cond} - two-character condition mnemonic, see *Figure 9: Condition Codes*

{S} - set condition codes if S present (implied for CMP, CMN, TEQ, TST).

Rd, Rn and Rm are expressions evaluating to a register number.

If <#expression> is used, the assembler will attempt to generate a shifted immediate 8-bit field to match the expression. If this is impossible, it will give an error.

<shift> is <shiftname> <register> or <shiftname> #expression, or RRX (rotate right one bit with extend).

<shiftname>s are: ASL, LSL, LSR, ASR, ROR. (ASL is a synonym for LSL, they assemble to the same code.)

### 4.4.9 Examples

```
ADDEQ    R2,R4,R5            ; if the Z flag is set make R2:=R4+R5

TEQS     R4,#3               ; test R4 for equality with 3
                             ; (the S is in fact redundant as the
                             ; assembler inserts it automatically)

SUB      R4,R5,R7,LSR R2     ; logical right shift R7 by the number in
                             ; the bottom byte of R2, subtract result
                             ; from R5, and put the answer into R4

MOV      PC,R14              ; return from subroutine

MOVS     PC,R14              ; return from exception and restore CPSR
                               from SPSR_mode
```

## 4.5 PSR Transfer (MRS, MSR)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter.

The MRS and MSR instructions are formed from a subset of the Data Processing operations and are implemented using the TEQ, TST, CMN and CMP instructions without the S flag set. The encoding is shown in *Figure 18: PSR Transfer*.

These instructions allow access to the CPSR and SPSR registers. The MRS instruction allows the contents of the CPSR or SPSR_<mode> to be moved to a general register. The MSR instruction allows the contents of a general register to be moved to the CPSR or SPSR_<mode> register.

The MSR instruction also allows an immediate value or register contents to be transferred to the condition code flags (N,Z,C and V) of CPSR or SPSR_<mode> without affecting the control bits. In this case, the top four bits of the specified register contents or 32 bit immediate value are written to the top four bits of the relevant PSR.

### 4.5.1 Operand restrictions

In User mode, the control bits of the CPSR are protected from change, so only the condition code flags of the CPSR can be changed. In other (privileged) modes the entire CPSR can be changed.

The SPSR register which is accessed depends on the mode at the time of execution. For example, only SPSR_fiq is accessible when the processor is in FIQ mode.

R15 shall not be specified as the source or destination register.

A further restriction is that no attempt shall be made to access an SPSR in User mode, since no such register exists.

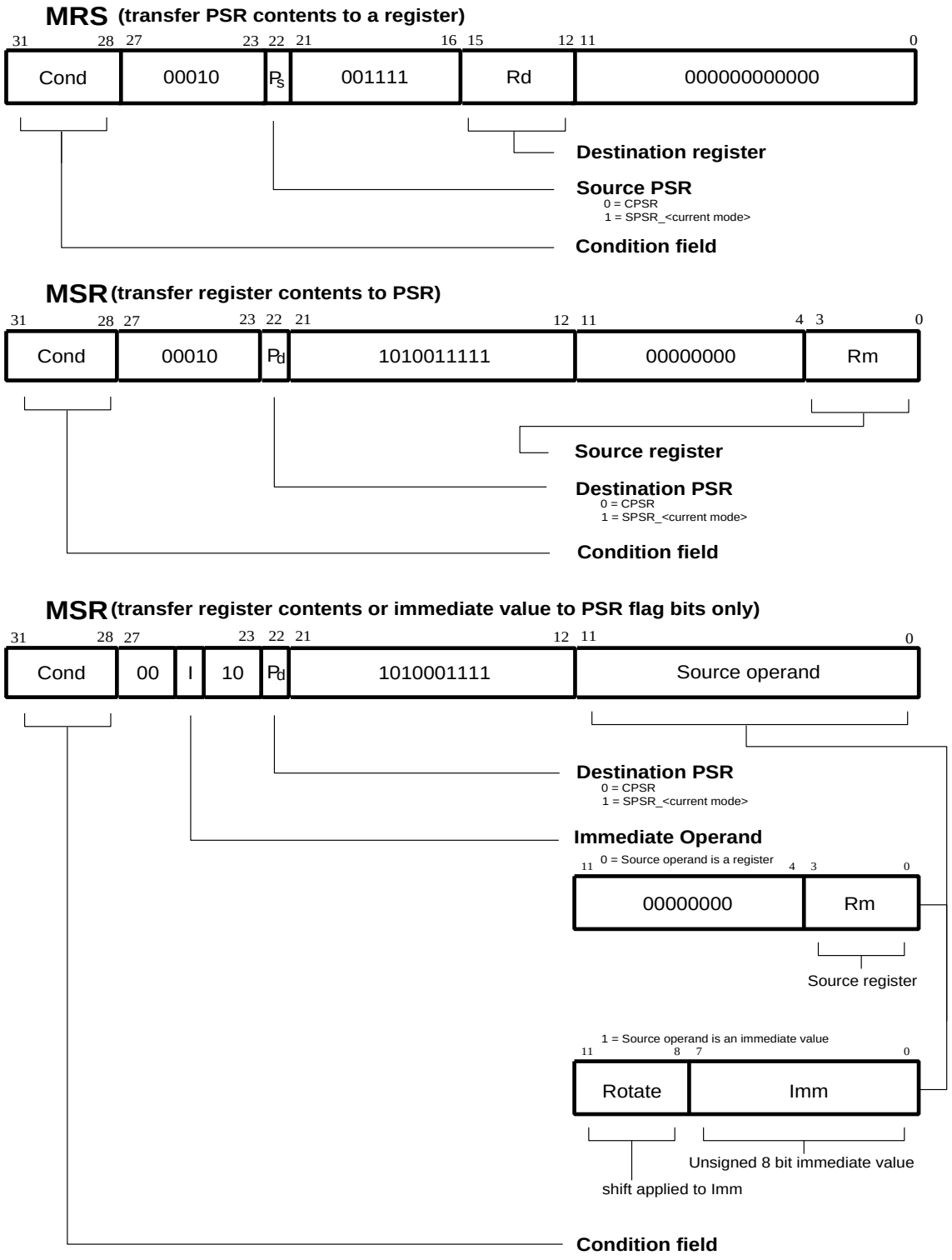



**Figure 18: PSR Transfer**

### 4.5.2 Reserved bits

Only eleven bits of the PSR are defined in ARM7DI (N,Z,C,V,I,F & M[4:0]); the remaining bits (= PSR[27:8,5]) are reserved for use in future versions of the processor. To ensure the maximum compatibility between ARM7DI programs and future processors, the following rules should be observed:

(1) The reserved bits shall be preserved when changing the value in a PSR.

(2) Programs shall not rely on specific values from the reserved bits when checking the PSR status, since they may read as one or zero in future processors.

A read-modify-write strategy should therefore be used when altering the control bits of any PSR register; this involves transferring the appropriate PSR register to a general register using the MRS instruction, changing only the relevant bits and then transferring the modified value back to the PSR register using the MSR instruction.

e.g. The following sequence performs a mode change:

```
MRS     R0,CPSR             ; take a copy of the CPSR
BIC     R0,R0,#0x1F         ; clear the mode bits
ORR     R0,R0,#new_mode     ; select new mode
MSR     CPSR,R0             ; write back the modified CPSR
```

When the aim is simply to change the condition code flags in a PSR, a value can be written directly to the flag bits without disturbing the control bits. e.g. The following instruction sets the N,Z,C & V flags:

```
MSR    CPSR_flg,#0xF0000000     ; set all the flags regardless of
                                ; their previous state (does not
                                ; affect any control bits)
```

No attempt shall be made to write an 8 bit immediate value into the whole PSR since such an operation cannot preserve the reserved bits.

### 4.5.3 Instruction Cycle Times

PSR Transfers take 1S incremental cycles, where S is as defined in *Section 5.1: Cycle types* on page 71.

### 4.5.4 Assembler syntax

(1) MRS - transfer PSR contents to a register

**MRS{cond} Rd,<psr>**

(2) MSR - transfer register contents to PSR

**MSR{cond} <psr>,Rm**

(3) MSR - transfer register contents to PSR flag bits only

**MSR{cond} <psrf>,Rm**

The most significant four bits of the register contents are written to the N,Z,C & V flags respectively.

(4) MSR - transfer immediate value to PSR flag bits only

**MSR{cond} <psrf>,<#expression>**

The expression should symbolise a 32 bit value of which the most significant four bits are written to the N,Z,C & V flags respectively.

{cond} - two-character condition mnemonic, see *Figure 9: Condition Codes*

Rd and Rm are expressions evaluating to a register number other than R15

<psr> is CPSR, CPSR_all, SPSR or SPSR_all. (CPSR and CPSR_all are synonyms as are SPSR and SPSR_all)

<psrf> is CPSR_flg or SPSR_flg

Where <#expression> is used, the assembler will attempt to generate a shifted immediate 8-bit field to match the expression. If this is impossible, it will give an error.

### 4.5.5 Examples

In User mode the instructions behave as follows:

```
MSR      CPSR_all,Rm          ; CPSR[31:28] <- Rm[31:28]
MSR      CPSR_flg,Rm          ; CPSR[31:28] <- Rm[31:28]

MSR      CPSR_flg,#0xA0000000 ; CPSR[31:28] <- 0xA
                              ;   (i.e. set N,C; clear Z,V)

MRS      Rd,CPSR              ; Rd[31:0] <- CPSR[31:0]
```

In privileged modes the instructions behave as follows:

```
MSR      CPSR_all,Rm          ; CPSR[31:0]  <- Rm[31:0]
MSR      CPSR_flg,Rm          ; CPSR[31:28] <- Rm[31:28]

MSR      CPSR_flg,#0x50000000 ; CPSR[31:28] <- 0x5
                              ;   (i.e. set Z,V; clear N,C)

MRS      Rd,CPSR              ; Rd[31:0] <- CPSR[31:0]

MSR      SPSR_all,Rm          ; SPSR_<mode>[31:0]  <- Rm[31:0]
MSR      SPSR_flg,Rm          ; SPSR_<mode>[31:28] <- Rm[31:28]

MSR      SPSR_flg,#0xC0000000 ; SPSR_<mode>[31:28] <- 0xC
                              ;   (i.e. set N,Z; clear C,V)

MRS      Rd,SPSR              ; Rd[31:0] <- SPSR_<mode>[31:0]
```

## 4.6 Multiply and Multiply-Accumulate (MUL, MLA)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 19: Multiply Instructions*.

The multiply and multiply-accumulate instructions use a 2 bit Booth's algorithm to perform integer multiplication. They give the least significant 32 bits of the product of two 32 bit operands, and may be used to synthesize higher precision multiplications.

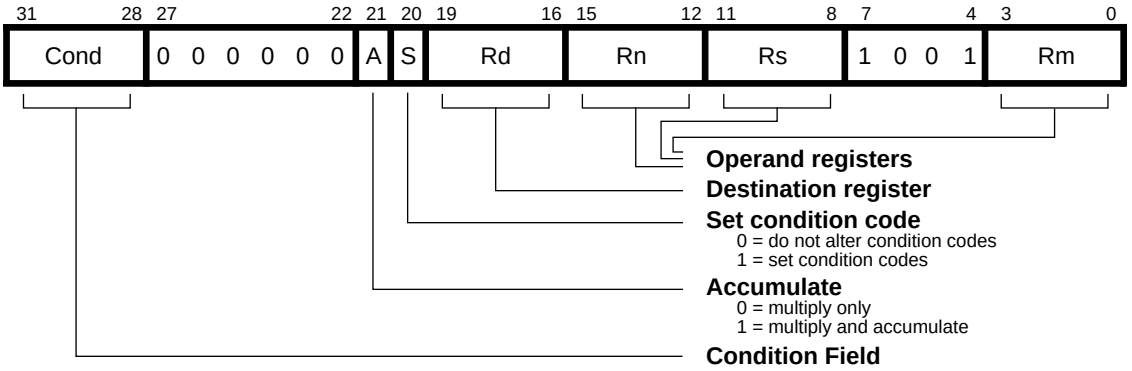



**Figure 19: Multiply Instructions**

The multiply form of the instruction gives Rd:=Rm*Rs. Rn is ignored, and should be set to zero for compatibility with possible future upgrades to the instruction set.

The multiply-accumulate form gives Rd:=Rm*Rs+Rn, which can save an explicit ADD instruction in some circumstances.

Both forms of the instruction work on operands which may be considered as signed (2's complement) or unsigned integers.

### 4.6.1 Operand Restrictions

Due to the way multiplication was implemented, certain combinations of operand registers should be avoided. (The assembler will issue a warning if these restrictions are overlooked.)

The destination register (Rd) should not be the same as the operand register (Rm), as Rd is used to hold intermediate values and Rm is used repeatedly during multiply. A MUL will give a zero result if RM=Rd, and an MLA will give a meaningless result. R15 shall not be used as an operand or as the destination register.

All other register combinations will give correct results, and Rd, Rn and Rs may use the same register when required.

### 4.6.2 CPSR flags

Setting the CPSR flags is optional, and is controlled by the S bit in the instruction. The N (Negative) and Z (Zero) flags are set correctly on the result (N is made equal to bit 31 of the result, and Z is set if and only if the result is zero). The C (Carry) flag is set to a meaningless value and the V (oVerflow) flag is unaffected.

### 4.6.3 Instruction Cycle Times

The Multiply instructions take 1S + mI cycles to execute, where S and I are as defined in *Section 5.1: Cycle types* on page 71.

*m* is the number of cycles required by the multiply algorithm, which is determined by the contents of Rs. Multiplication by any number between *2^(2m-3)* and *2^(2m-1)-1* takes 1S+mI cycles for *1<m>16*. Multiplication by 0 or 1 takes 1S+1I cycles, and multiplication by any number greater than or equal to 2^(29) takes 1S+16I cycles. The maximum time for any multiply is thus 1S+16I cycles.

### 4.6.4 Assembler syntax

**MUL{cond}{S} Rd,Rm,Rs**

**MLA{cond}{S} Rd,Rm,Rs,Rn**

{cond} - two-character condition mnemonic, see *Figure 9: Condition Codes*

{S} - set condition codes if S present

Rd, Rm, Rs and Rn are expressions evaluating to a register number other than R15.

### 4.6.5 Examples

```
MUL      R1,R2,R3            ; R1:=R2*R3
MLAEQS   R1,R2,R3,R4         ; conditionally R1:=R2*R3+R4,
                             ; setting condition codes
```

## 4.7 Single data transfer (LDR, STR)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 20: Single Data Transfer Instructions*.

The single data transfer instructions are used to load or store single bytes or words of data. The memory address used in the transfer is calculated by adding an offset to or subtracting an offset from a base register. The result of this calculation may be written back into the base register if `auto-indexing' is required.

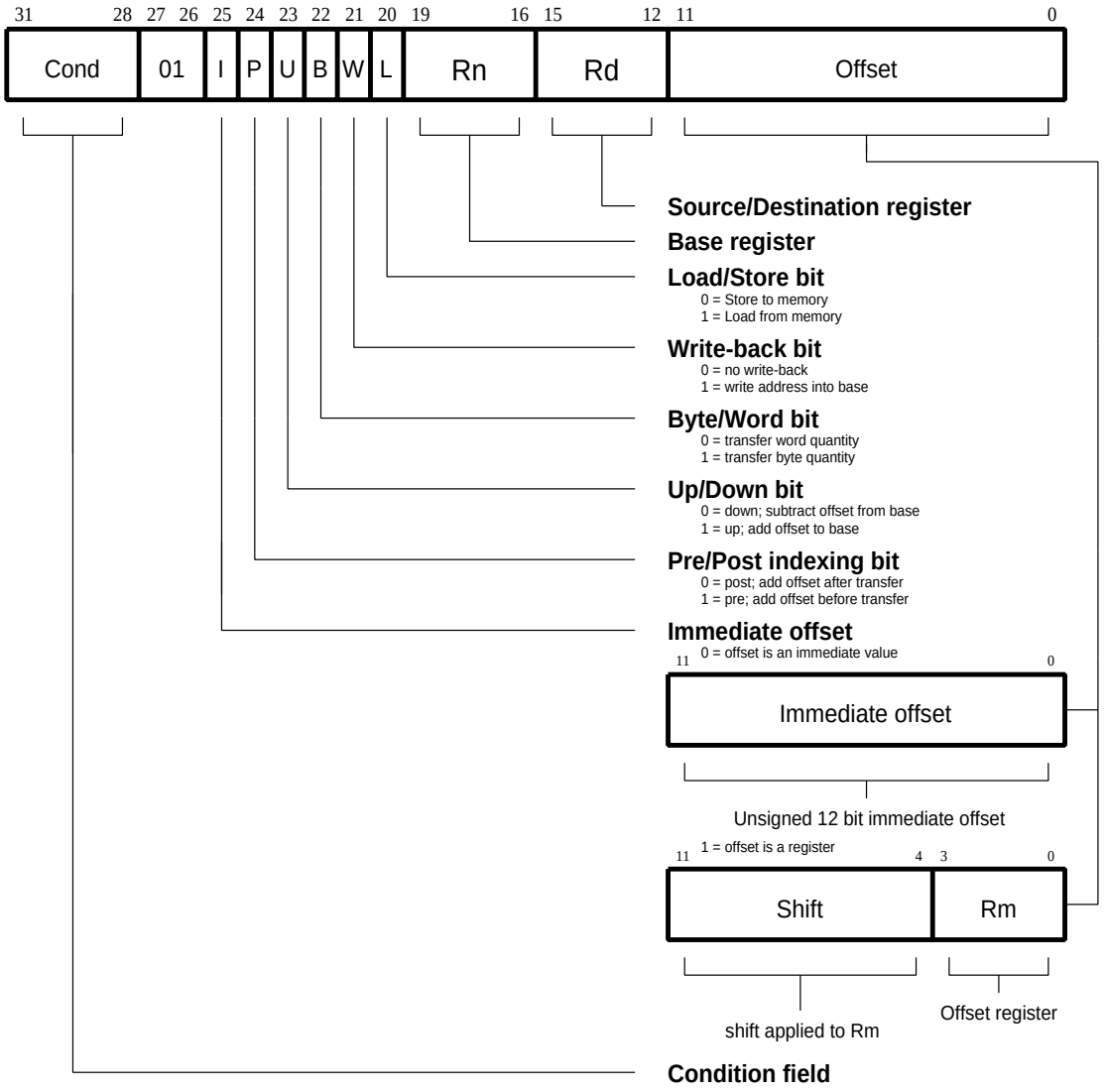



**Figure 20: Single Data Transfer Instructions**

### 4.7.1 Offsets and auto-indexing

The offset from the base may be either a 12 bit unsigned binary immediate value in the instruction, or a second register (possibly shifted in some way). The offset may be added to (U=1) or subtracted from (U=0) the base register Rn. The offset modification may be performed either before (pre-indexed, P=1) or after (post-indexed, P=0) the base is used as the transfer address.

The W bit gives optional auto increment and decrement addressing modes. The modified base value may be written back into the base (W=1), or the old base value may be kept (W=0). In the case of post-indexed addressing, the write back bit is redundant and is always set to zero, since the old base value can be retained by setting the offset to zero. Therefore post-indexed data transfers always write back the modified base. The only use of the W bit in a post-indexed data transfer is in privileged mode code, where setting the W bit forces non-privileged mode for the transfer, allowing the operating system to generate a user address in a system where the memory management hardware makes suitable use of this hardware.

### 4.7.2 Shifted register offset

The 8 shift control bits are described in the data processing instructions section. However, the register specified shift amounts are not available in this instruction class. See *4.4.2 Shifts*.

### 4.7.3 Bytes and words

This instruction class may be used to transfer a byte (B=1) or a word (B=0) between an ARM7DI register and memory.

The action of LDR(B) and STR(B) instructions is influenced by the **BIGEND** control signal. The two possible configurations are described below.

**Little Endian Configuration**

A byte load (LDRB) expects the data on data bus inputs 7 through 0 if the supplied address is on a word boundary, on data bus inputs 15 through 8 if it is a word address plus one byte, and so on. The selected byte is placed in the bottom 8 bits of the destination register, and the remaining bits of the register are filled with zeros. Please see *Figure 4: Little Endian addresses of bytes within words*.

A byte store (STRB) repeats the bottom 8 bits of the source register four times across data bus outputs 31 through 0. The external memory system should activate the appropriate byte subsystem to store the data.

A word load (LDR) will normally use a word aligned address. However, an address offset from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits 0 to 7. This means that half-words accessed at offsets 0 and 2 from the word boundary will be correctly loaded into bits 0 through 15 of the register. Two shift operations are then required to clear or to sign extend the upper 16 bits. This is illustrated in *Figure 21: Little Endian Offset Addressing*.

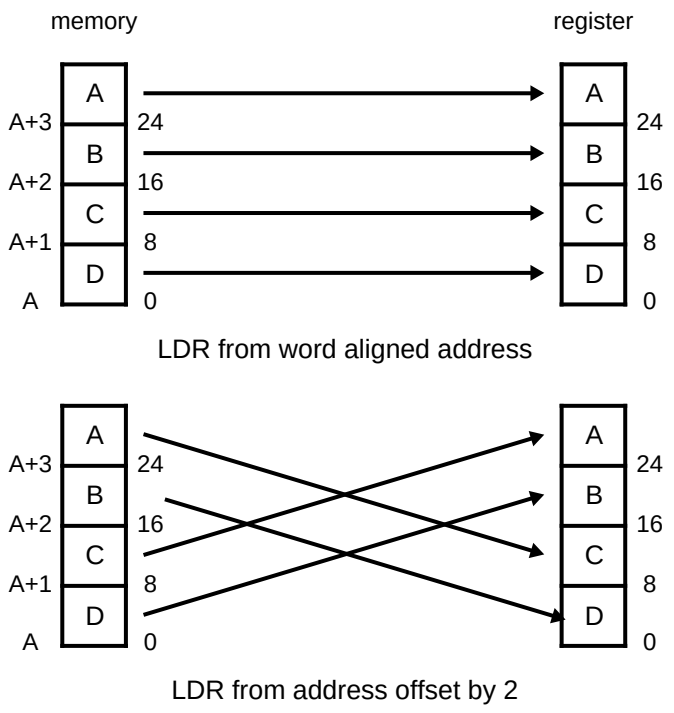



**Figure 21: Little Endian Offset Addressing**

A word store (STR) should generate a word aligned address. The word presented to the data bus is not affected if the address is not word aligned. That is, bit 31 of the register being stored always appears on data bus output 31.

**Big Endian Configuration**

A byte load (LDRB) expects the data on data bus inputs 31 through 24 if the supplied address is on a word boundary, on data bus inputs 23 through 16 if it is a word address plus one byte, and so on. The selected byte is placed in the bottom 8 bits of the destination register and the remaining bits of the register are filled with zeros. Please see *Figure 5: Big Endian addresses of bytes within words.*

A byte store (STRB) repeats the bottom 8 bits of the source register four times across data bus outputs 31 through 0. The external memory system should activate the appropriate byte subsystem to store the data.

A word load (LDR) should generate a word aligned address. An address offset of 0 or 2 from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits 31 through 24. This means that half-words accessed at these offsets will be correctly loaded into bits 16 through 31 of the register. A shift operation is then required to move (and optionally sign extend) the data into the bottom 16 bits. An address offset of 1 or 3 from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits 15 through 8.

A word store (STR) should generate a word aligned address. The word presented to the data bus is not affected if the address is not word aligned. That is, bit 31 of the register being stored always appears on data bus output 31.

### 4.7.4 Use of R15

Write-back shall not be specified if R15 is specified as the base register (Rn). When using R15 as the base register you must remember it contains an address 8 bytes on from the address of the current instruction.

R15 shall not be specified as the register offset (Rm).

When R15 is the source register (Rd) of a register store (STR) instruction, the stored value will be address of the instruction plus 12.

### 4.7.5 Restriction on the use of base register

When configured for late aborts, the following example code is difficult to unwind as the base register, Rn, gets updated before the abort handler starts. Sometimes it may be impossible to calculate the initial value.

For example:

```
LDR   R0,[R1],R1
```

Therefore a post-indexed LDR|STR where Rm is the same register as Rn shall not be used.

### 4.7.6 Data Aborts

A transfer to or from a legal address may cause problems for a memory management system. For instance, in a system which uses virtual memory the required data may be absent from main memory. The memory manager can signal a problem by taking the processor **ABORT** input HIGH whereupon the Data Abort trap will be taken. It is up to the system software to resolve the cause of the problem, then the instruction can be restarted and the original program continued.

### 4.7.7 Instruction Cycle Times

Normal LDR instructions take 1S + 1N + 1I and LDR PC take 2S + 2N +1I incremental cycles, where S,N and I are as defined in *Section 5.1: Cycle types* on page 71.

STR instructions take 2N incremental cycles to execute.

### 4.7.8 Assembler syntax

**<LDR|STR>{cond}{B}{T} Rd,<Address>**

LDR - load from memory into a register

STR - store from a register into memory

{cond} - two-character condition mnemonic, see *Figure 9: Condition Codes*

{B} - if B is present then byte transfer, otherwise word transfer

{T} - if T is present the W bit will be set in a post-indexed instruction, forcing non-privileged mode for the transfer cycle. T is not allowed when a pre-indexed addressing mode is specified or implied.

Rd is an expression evaluating to a valid register number.

<Address> can be:

(i) An expression which generates an address:

**<expression>**

The assembler will attempt to generate an instruction using the PC as a base and a corrected immediate offset to address the location given by evaluating the expression. This will be a PC relative, pre-indexed address. If the address is out of range, an error will be generated.

(ii) A pre-indexed addressing specification:

**[Rn]** offset of zero

**[Rn,<#expression>]{!}** offset of <expression> bytes

**[Rn,{+/-}Rm{,<shift>}]{!}** offset of + / - contents of index register, shifted by <shift>

(iii) A post-indexed addressing specification:

**[Rn],<#expression>** offset of <expression> bytes

**[Rn],{+/-}Rm{,<shift>}** offset of + / - contents of index register, shifted as by <shift>.

Rn and Rm are expressions evaluating to a register number. If Rn is R15 then the assembler will subtract 8 from the offset value to allow for ARM7DI pipelining. In this case base write-back shall not be specified.

<shift> is a general shift operation (see section on data processing instructions) but note that the shift amount may not be specified by a register.

{!} writes back the base register (set the W bit) if ! is present.

### 4.7.9 Examples

```
STR     R1,[R2,R4]!          ; store R1 at R2+R4 (both of which are
                             ; registers) and write back address to R2

STR     R1,[R2],R4           ; store R1 at R2 and write back
                             ; R2+R4 to R2

LDR     R1,[R2,#16]          ; load R1 from contents of R2+16
                             ;  Don't write back

LDR     R1,[R2,R3,LSL#2]     ; load R1 from contents of R2+R3*4
```

```
LDREQB   R1,[R6,#5]          ; conditionally load byte at R6+5 into
                             ;  R1 bits 0 to 7, filling bits 8 to 31
                             ;  with zeros

STR      R1,PLACE            ; generate PC relative offset to address
         •                   ;  PLACE
         •
PLACE
```

## 4.8 Block Data Transfer (LDM, STM)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 22: Block Data Transfer Instructions*.

Block data transfer instructions are used to load (LDM) or store (STM) any subset of the currently visible registers. They support all possible stacking modes, maintaining full or empty stacks which can grow up or down memory, and are very efficient instructions for saving or restoring context, or for moving large blocks of data around main memory.

### 4.8.1 The Register List

The instruction can cause the transfer of any registers in the current bank (and non-user mode programs can also transfer to and from the user bank, see below). The register list is a 16 bit field in the instruction, with each bit corresponding to a register. A 1 in bit 0 of the register field will cause R0 to be transferred, a 0 will cause it not to be transferred; similarly bit 1 controls the transfer of R1, and so on.

Any subset of the registers, or all the registers, may be specified. The only restriction is that the register list should not be empty.

Whenever R15 is stored to memory the stored value is the address of the STM instruction plus 12.

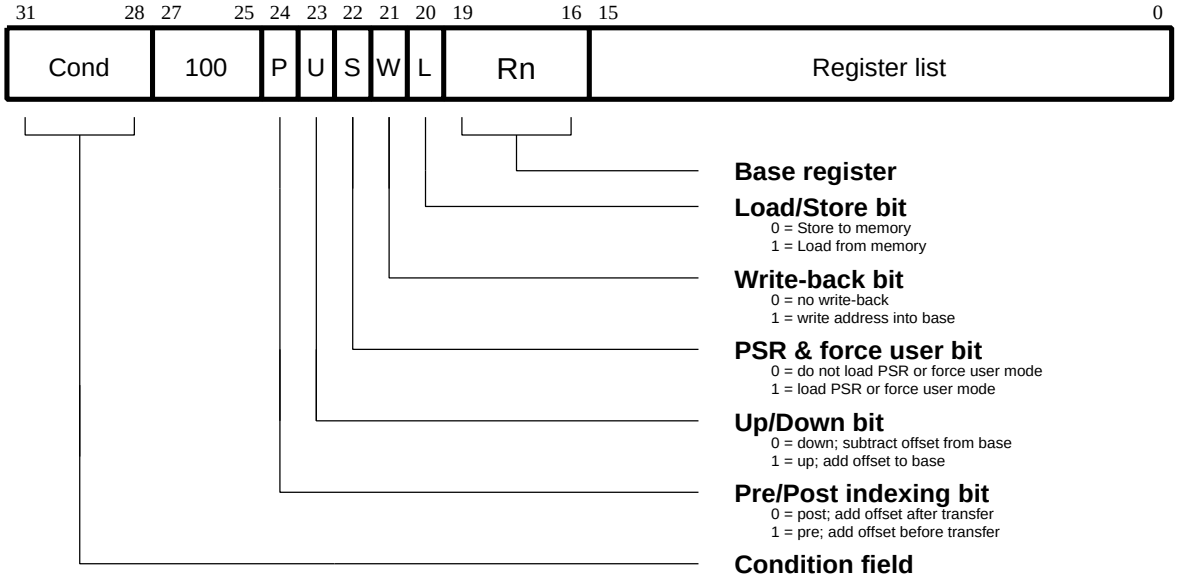



**Figure 22: Block Data Transfer Instructions**

### 4.8.2 Addressing Modes

The transfer addresses are determined by the contents of the base register (Rn), the pre/post bit (P) and the up/down bit (U). The registers are transferred in the order lowest to highest, so R15 (if in the list) will always be transferred last. The lowest register also gets transferred to/from the lowest memory address. By way of illustration, consider the transfer of R1, R5 and R7 in the case where Rn=0x1000 and write back of

the modified base is required (W=1). *Figure 23: Post-increment addressing*, *Figure 24: Pre-increment addressing*, *Figure 25: Post-decrement addressing* and *Figure 26: Pre-decrement addressing* show the sequence of register transfers, the addresses used, and the value of Rn after the instruction has completed.

In all cases, had write back of the modified base not been required (W=0), Rn would have retained its initial value of 0x1000 unless it was also in the transfer list of a load multiple register instruction, when it would have been overwritten with the loaded value.

### 4.8.3 Address Alignment

The address should normally be a word aligned quantity and non-word aligned addresses do not affect the instruction. However, the bottom 2 bits of the address will appear on **A[1:0]** and might be interpreted by the memory system.

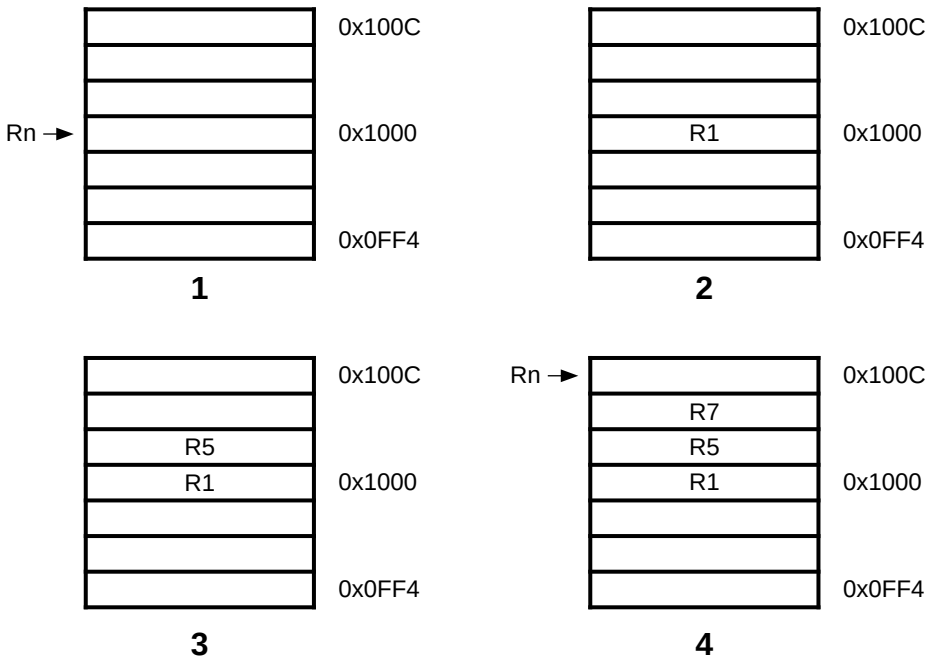



**Figure 23: Post-increment addressing**

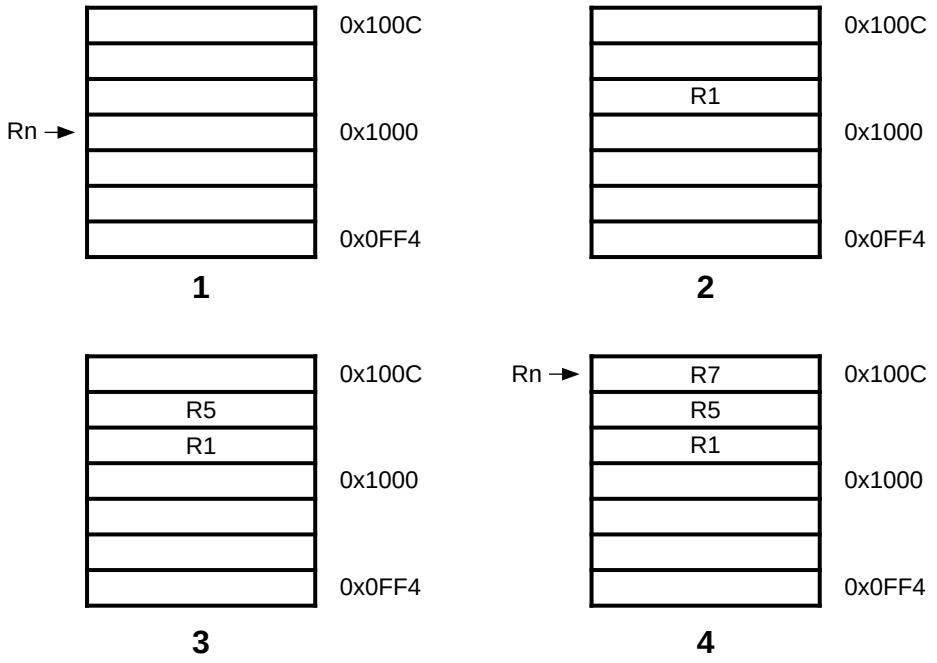



**Figure 24: Pre-increment addressing**

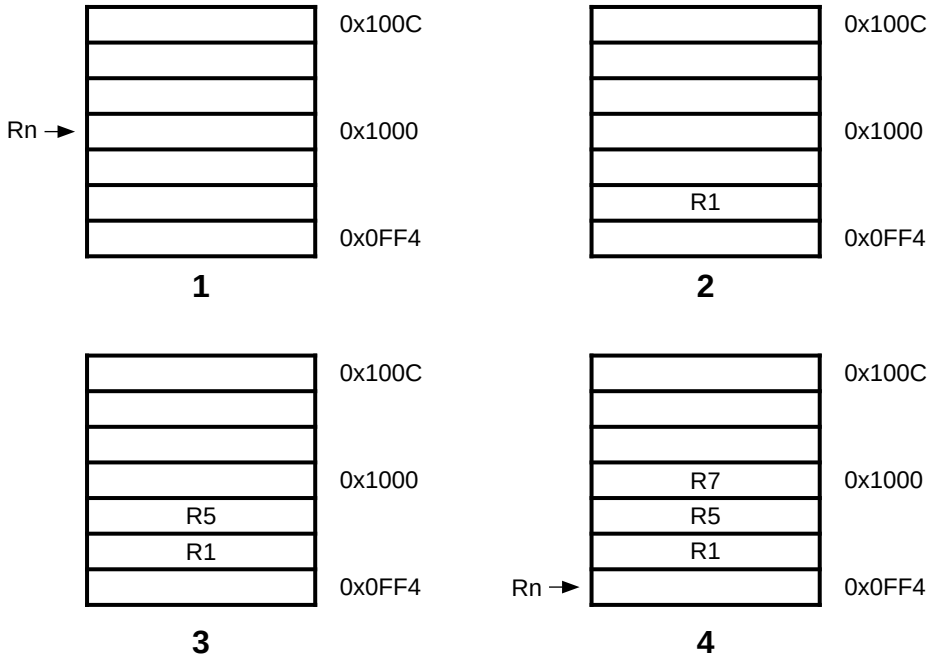



**Figure 25: Post-decrement addressing**

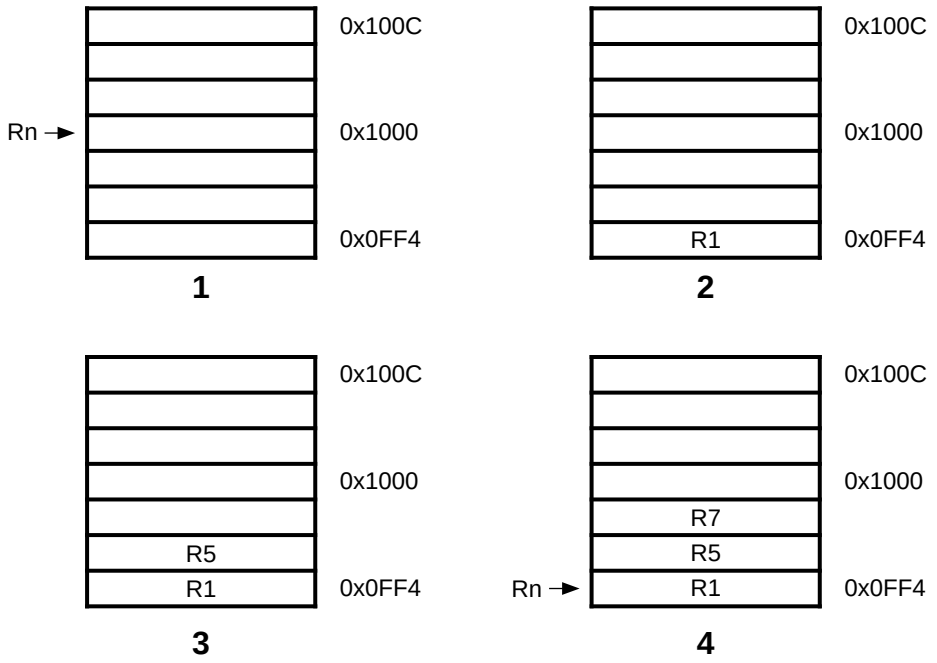



**Figure 26: Pre-decrement addressing**

### 4.8.4 Use of the S bit

When the S bit is set in a LDM / STM instruction its meaning depends on whether or not R15 is in the transfer list and on the type of instruction. The S bit should only be set if the instruction is to execute in a privileged mode.

**LDM with R15 in transfer list and S bit set (Mode changes)**

If the instruction is a LDM then SPSR_<mode> is transferred to CPSR at the same time as R15 is loaded.

**STM with R15 in transfer list and S bit set (User bank transfer)**

The registers transferred are taken from the User bank rather than the bank corresponding to the current mode. This is useful for saving the user state on process switches. Base write-back shall not be used when this mechanism is employed.

**R15 not in list and S bit set (User bank transfer)**

For both LDM and STM instructions, the User bank registers are transferred rather than the register bank corresponding to the current mode. This is useful for saving the user state on process switches. Base write-back shall not be used when this mechanism is employed.

When the instruction is LDM, care must be taken not to read from a banked register during the following cycle (inserting a dummy instruction such as MOV R0, R0 after the LDM will ensure safety).

### 4.8.5 Use of R15 as the base

R15 shall not be used as the base register in any LDM or STM instruction.

### 4.8.6 Inclusion of the base in the register list

When write-back is specified, the base is written back at the end of the second cycle of the instruction. During a STM, the first register is written out at the start of the second cycle. A STM which includes storing the base, with the base as the first register to be stored, will therefore store the unchanged value, whereas with the base second or later in the transfer order, will store the modified value. A LDM will always overwrite the updated base if the base is in the list.

### 4.8.7 Data Aborts

Some legal addresses may be unacceptable to a memory management system, and the memory manager can indicate a problem with an address by taking the **ABORT** signal HIGH. This can happen on any transfer during a multiple register load or store, and must be recoverable if ARM7DI is to be used in a virtual memory system.

#### Aborts during STM instructions

If the abort occurs during a store multiple instruction, ARM7DI takes little action until the instruction completes, whereupon it enters the data abort trap. The memory manager is responsible for preventing erroneous writes to the memory. The only change to the internal state of the processor will be the modification of the base register if write-back was specified, and this must be reversed by software (and the cause of the abort resolved) before the instruction may be retried.

#### Aborts during LDM instructions

When ARM7DI detects a data abort during a load multiple instruction, it modifies the operation of the instruction to ensure that recovery is possible.

(i) Overwriting of registers stops when the abort happens. The aborting load will not take place but earlier ones may have overwritten registers. The PC is always the last register to be written and so will always be preserved.

(ii) The base register is restored, to its modified value if write-back was requested. This ensures recoverability in the case where the base register is also in the transfer list, and may have been overwritten before the abort occurred.

The data abort trap is taken when the load multiple has completed, and the system software must undo any base modification (and resolve the cause of the abort) before restarting the instruction.

### 4.8.8 Instruction Cycle Times

Normal LDM instructions take nS + 1N + 1I and LDM PC takes (n+1)S + 2N + 1I incremental cycles, where S,N and I are as defined in *Section 5.1: Cycle types* on page 71.

STM instructions take (n-1)S + 2N incremental cycles to execute.

*n* is the number of words transferred.

### 4.8.9 Assembler syntax

**<LDM | STM>{cond}<FD | ED | FA | EA | IA | IB | DA | DB> Rn{!},<Rlist>{^}**

{cond} - two character condition mnemonic, see *Figure 9: Condition Codes*

Rn is an expression evaluating to a valid register number

<Rlist> is a list of registers and register ranges enclosed in {} (eg {R0,R2-R7,R10}).

{!} if present requests write-back (W=1), otherwise W=0

{^} if present set S bit to load the CPSR along with the PC, or force transfer of user bank when in privileged mode

**Addressing mode names**

There are different assembler mnemonics for each of the addressing modes, depending on whether the instruction is being used to support stacks or for other purposes. The equivalences between the names and the values of the bits in the instruction are shown in the following table:

| name | stack | other | L bit | P bit | U bit |
|---|---|---|---|---|---|
| pre-increment load | LDMED | LDMIB | 1 | 1 | 1 |
| post-increment load | LDMFD | LDMIA | 1 | 0 | 1 |
| pre-decrement load | LDMEA | LDMDB | 1 | 1 | 0 |
| post-decrement load | LDMFA | LDMDA | 1 | 0 | 0 |
| pre-increment store | STMFA | STMIB | 0 | 1 | 1 |
| post-increment store | STMEA | STMIA | 0 | 0 | 1 |
| pre-decrement store | STMFD | STMDB | 0 | 1 | 0 |
| post-decrement store | STMED | STMDA | 0 | 0 | 0 |

**Table 5: Addressing Mode Names**

FD, ED, FA, EA define pre / post indexing and the up / down bit by reference to the form of stack required. The F and E refer to a “full” or “empty” stack, i.e. whether a pre-index has to be done (full) before storing to the stack. The A and D refer to whether the stack is ascending or descending. If ascending, a STM will go up and LDM down, if descending, vice-versa.

IA, IB, DA, DB allow control when LDM / STM are not being used for stacks and simply mean Increment After, Increment Before, Decrement After, Decrement Before.

### 4.8.10 Examples

```
LDMFD    SP!,{R0,R1,R2}       ; unstack 3 registers

STMIA    R0,{R0-R15}          ; save all registers

LDMFD    SP!,{R15}            ; R15 <- (SP),CPSR unchanged
LDMFD    SP!,{R15}^           ; R15 <- (SP), CPSR <- SPSR_mode (allowed
                              ;  only in privileged modes)
STMFD    R13,{R0-R14}^        ; Save user mode regs on stack (allowed
                              ;  only in privileged modes)
```

These instructions may be used to save state on subroutine entry, and restore it efficiently on return to the calling routine:

```
STMED    SP!,{R0-R3,R14}      ; save R0 to R3 to use as workspace
                              ;  and R14 for returning

BL       somewhere            ; this nested call will overwrite R14

LDMED    SP!,{R0-R3,R15}      ; restore workspace and return
```

## 4.9 Single data swap (SWP)

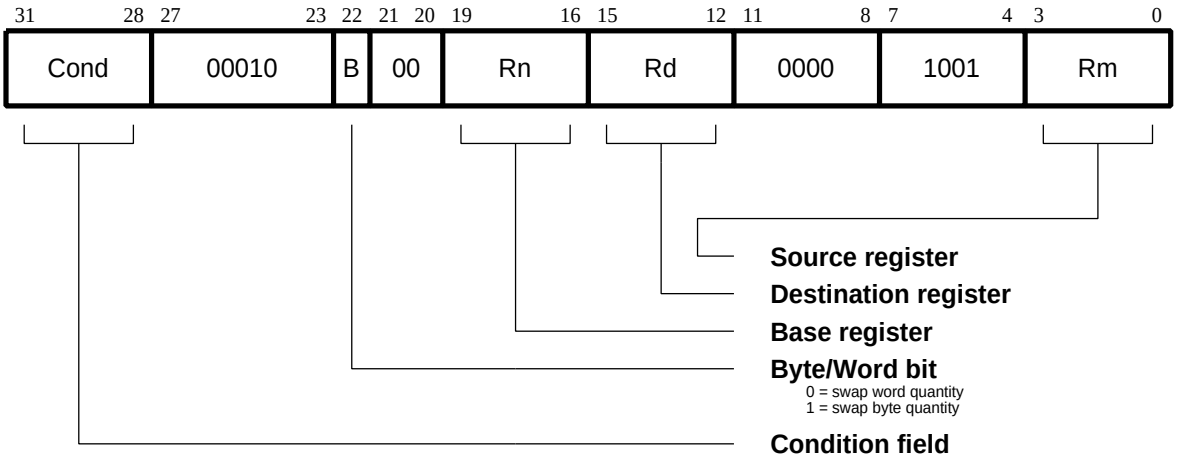



**Figure 27: Swap Instruction**

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 27: Swap Instruction*.

The data swap instruction is used to swap a byte or word quantity between a register and external memory. This instruction is implemented as a memory read followed by a memory write which are "locked" together (the processor cannot be interrupted until both operations have completed, and the memory manager is warned to treat them as inseparable). This class of instruction is particularly useful for implementing software semaphores.

The swap address is determined by the contents of the base register (Rn). The processor first reads the contents of the swap address. Then it writes the contents of the source register (Rm) to the swap address, and stores the old memory contents in the destination register (Rd). The same register may be specified as both the source and destination.

The **LOCK** output goes HIGH for the duration of the read and write operations to signal to the external memory manager that they are locked together, and should be allowed to complete without interruption. This is important in multi-processor systems where the swap instruction is the only indivisible instruction which may be used to implement semaphores; control of the memory must not be removed from a processor while it is performing a locked operation.

### 4.9.1 Bytes and words

This instruction class may be used to swap a byte (B=1) or a word (B=0) between an ARM7DI register and memory. The SWP instruction is implemented as a LDR followed by a STR and the action of these is as described in the section on single data transfers. In particular, the description of Big and Little Endian configuration applies to the SWP instruction.

### 4.9.2 Use of R15

R15 shall not be used as an operand (Rd, Rn or Rs) in a SWP instruction.

### 4.9.3 Data Aborts

If the address used for the swap is unacceptable to a memory management system, the memory manager can flag the problem by driving **ABORT** HIGH. This can happen on either the read or the write cycle (or both), and in either case, the Data Abort trap will be taken. It is up to the system software to resolve the cause of the problem, then the instruction can be restarted and the original program continued.

### 4.9.4 Instruction Cycle Times

Swap instructions take 1S + 2N +1I incremental cycles to execute, where S,N and I are as defined in *Section 5.1: Cycle types* on page 71.

### 4.9.5 Assembler syntax

**<SWP>{cond}{B} Rd,Rm,[Rn]**

{cond} - two-character condition mnemonic, see *Figure 9: Condition Codes*

{B} - if B is present then byte transfer, otherwise word transfer

Rd,Rm,Rn are expressions evaluating to valid register numbers

### 4.9.6 Examples

```
SWP     R0,R1,[R2]          ; load R0 with the word addressed by R2, and
                            ; store R1 at R2

SWPB    R2,R3,[R4]          ; load R2 with the byte addressed by R4, and
                            ; store bits 0 to 7 of R3 at R4

SWPEQ   R0,R0,[R1]          ; conditionally swap the contents of the
                            ; word addressed by R1 with R0
```

## 4.10 Software interrupt (SWI)

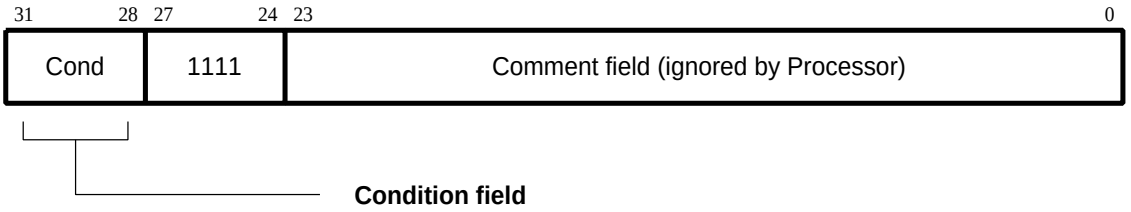



**Figure 28: Software Interrupt Instruction**

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 28: Software Interrupt Instruction*.

The software interrupt instruction is used to enter Supervisor mode in a controlled manner. The instruction causes the software interrupt trap to be taken, which effects the mode change. The PC is then forced to a fixed value (0x08) and the CPSR is saved in SPSR_svc. If the SWI vector address is suitably protected (by external memory management hardware) from modification by the user, a fully protected operating system may be constructed.

### 4.10.1 Return from the supervisor

The PC is saved in R14_svc upon entering the software interrupt trap, with the PC adjusted to point to the word after the SWI instruction. MOVS PC,R14_svc will return to the calling program and restore the CPSR.

Note that the link mechanism is not re-entrant, so if the supervisor code wishes to use software interrupts within itself it must first save a copy of the return address and SPSR.

### 4.10.2 Comment field

The bottom 24 bits of the instruction are ignored by the processor, and may be used to communicate information to the supervisor code. For instance, the supervisor may look at this field and use it to index into an array of entry points for routines which perform the various supervisor functions.

### 4.10.3 Instruction Cycle Times

Software interrupt instructions take 2S + 1N incremental cycles to execute, where S and N are as defined in *Section 5.1: Cycle types* on page 71.

### 4.10.4 Assembler syntax

SWI{cond} <expression>

{cond} - two character condition mnemonic, see *Figure 9: Condition Codes*

<expression> is evaluated and placed in the comment field (which is ignored by ARM7DI).

### 4.10.5 Examples

```
SWI      ReadC                ; get next character from read stream
SWI      WriteI+"k"           ; output a "k" to the write stream
SWINE    0                    ; conditionally call supervisor
                              ; with 0 in comment field
```

The above examples assume that suitable supervisor code exists, for instance:

```
0x08 B Supervisor             ; SWI entry point

EntryTable                    ; addresses of supervisor routines
         DCD ZeroRtn
         DCD ReadCRtn
         DCD WriteIRtn
          . . .

Zero     EQU  0
ReadC    EQU  256
WriteI   EQU  512

Supervisor

; SWI has routine required in bits 8-23 and data (if any) in bits 0-7.
; Assumes R13_svc points to a suitable stack

STMFD    R13,{R0-R2,R14}      ; save work registers and return address
LDR      R0,[R14,#-4]         ; get SWI instruction
BIC      R0,R0,#0xFF000000    ; clear top 8 bits
MOV      R1,R0,LSR#8          ; get routine offset
ADR      R2,EntryTable        ; get start address of entry table
LDR      R15,[R2,R1,LSL#2]    ; branch to appropriate routine

WriteIRtn                     ; enter with character in R0 bits 0-7
    .  .  .  .  .  .
LDMFD    R13,{R0-R2,R15}^     ; restore workspace and return
                              ; restoring processor mode and flags
```

## 4.11 Coprocessor data operations (CDP)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 29: Coprocessor Data Operation Instruction*.
This class of instruction is used to tell a coprocessor to perform some internal operation. No result is communicated back to ARM7DI, and it will not wait for the operation to complete. The coprocessor could contain a queue of such instructions awaiting execution, and their execution can overlap other ARM7DI activity, allowing the coprocessor and to perform independent tasks in parallel.

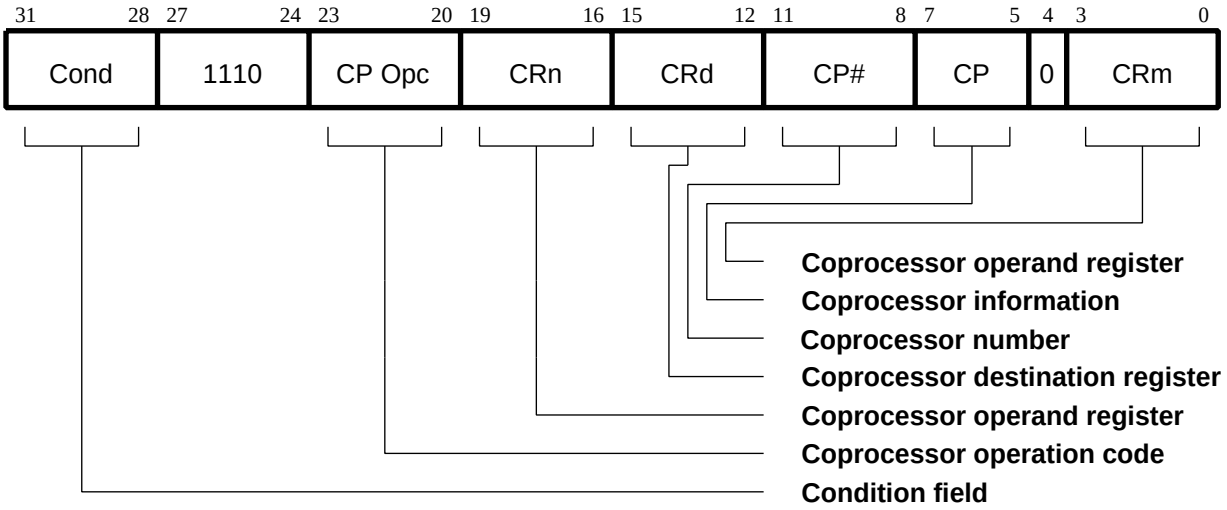



**Figure 29: Coprocessor Data Operation Instruction**

### 4.11.1 The Coprocessor fields

Only bit 4 and bits 24 to 31 are significant toARM7DI . The remaining bits are used by coprocessors. The above field names are used by convention, and particular coprocessors may redefine the use of all fields except CP# as appropriate. The CP# field is used to contain an identifying number (in the range 0 to 15) for each coprocessor, and a coprocessor will ignore any instruction which does not contain its number in the CP# field.

The conventional interpretation of the instruction is that the coprocessor should perform an operation specified in the CP Opc field (and possibly in the CP field) on the contents of CRn and CRm, and place the result in CRd.

### 4.11.2 Instruction Cycle Times

Coprocessor data operations take 1S + bI incremental cycles to execute, where S and I are as defined in *Section 5.1: Cycle types* on page 71.

*b* is the number of cycles spent in the coprocessor busy-wait loop.

### 4.11.3 Assembler syntax

**CDP{cond} p#,<expression1>,cd,cn,cm{,<expression2>}**

{cond} - two character condition mnemonic, see *Figure 9: Condition Codes*

p# - the unique number of the required coprocessor

<expression1> - evaluated to a constant and placed in the CP Opc field

cd, cn and cm evaluate to the valid coprocessor register numbers CRd, CRn and CRm respectively

<expression2> - where present is evaluated to a constant and placed in the CP field

### 4.11.4 Examples

```
CDP      p1,10,c1,c2,c3       ; request coproc 1 to do operation 10
                              ; on CR2 and CR3, and put the result in CR1

CDPEQ    p2,5,c1,c2,c3,2      ; if Z flag is set request coproc 2 to do
                              ; operation 5 (type 2) on CR2 and CR3,
                              ; and put the result in CR1
```

## 4.12 Coprocessor data transfers (LDC, STC)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 30: Coprocessor Data Transfer Instructions*.
This class of instruction is used to load (LDC) or store (STC) a subset of a coprocessors's registers directly to memory. ARM7DI is responsible for supplying the memory address, and the coprocessor supplies or accepts the data and controls the number of words transferred.

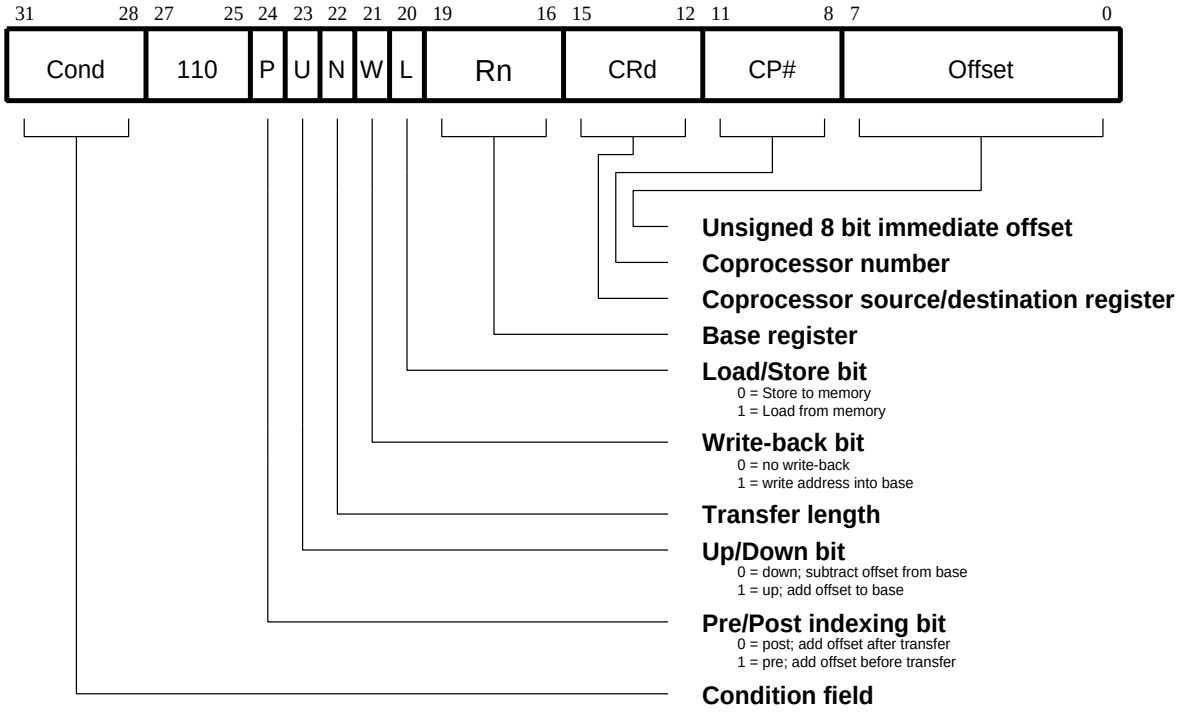



**Figure 30: Coprocessor Data Transfer Instructions**

### 4.12.1 The Coprocessor fields

The CP# field is used to identify the coprocessor which is required to supply or accept the data, and a coprocessor will only respond if its number matches the contents of this field.

The CRd field and the N bit contain information for the coprocessor which may be interpreted in different ways by different coprocessors, but by convention CRd is the register to be transferred (or the first register where more than one is to be transferred), and the N bit is used to choose one of two transfer length options. For instance N=0 could select the transfer of a single register, and N=1 could select the transfer of all the registers for context switching.

### 4.12.2 Addressing modes

ARM7DI is responsible for providing the address used by the memory system for the transfer, and the addressing modes available are a subset of those used in single data transfer instructions. Note, however, that the immediate offsets are 8 bits wide and specify word offsets for coprocessor data transfers, whereas they are 12 bits wide and specify byte offsets for single data transfers.

The 8 bit unsigned immediate offset is shifted left 2 bits and either added to (U=1) or subtracted from (U=0) the base register (Rn); this calculation may be performed either before (P=1) or after (P=0) the base is used as the transfer address. The modified base value may be overwritten back into the base register (if W=1), or the old value of the base may be preserved (W=0). Note that post-indexed addressing modes require explicit setting of the W bit, unlike LDR and STR which always write-back when post-indexed.

The value of the base register, modified by the offset in a pre-indexed instruction, is used as the address for the transfer of the first word. The second word (if more than one is transferred) will go to or come from an address one word (4 bytes) higher than the first transfer, and the address will be incremented by one word for each subsequent transfer.

### 4.12.3 Address Alignment

The base address should normally be a word aligned quantity. The bottom 2 bits of the address will appear on **A[1:0]** and might be interpreted by the memory system.

### 4.12.4 Use of R15

If Rn is R15, the value used will be the address of the instruction plus 8 bytes. Base write-back to R15 shall not be specified.

### 4.12.5 Data aborts

If the address is legal but the memory manager generates an abort, the data trap will be taken. The write-back of the modified base will take place, but all other processor state will be preserved. The coprocessor is partly responsible for ensuring that the data transfer can be restarted after the cause of the abort has been resolved, and must ensure that any subsequent actions it undertakes can be repeated when the instruction is retried.

### 4.12.6 Instruction Cycle Times

All LDC instructions are emulated in software: the number of cycles taken will depend on the coprocessor support software.

Coprocessor data transfer instructions take (n-1)S + 2N + bI incremental cycles to execute, where S, N and I are as defined in *Section 5.1: Cycle types* on page 71.

*n* is the number of words transferred.

*b* is the number of cycles spent in the coprocessor busy-wait loop.

### 4.12.7 Assembler syntax

**<LDC | STC>{cond}{L} p#,cd,<Address>**

LDC - load from memory to coprocessor

STC - store from coprocessor to memory

{L} - when present perform long transfer (N=1), otherwise perform short transfer (N=0)

{cond} - two character condition mnemonic, see *Figure 9: Condition Codes*

p# - the unique number of the required coprocessor

cd is an expression evaluating to a valid coprocessor register number that is placed in the CRd field

<Address> can be:

(i) An expression which generates an address:

<expression>

The assembler will attempt to generate an instruction using the PC as a base and a corrected immediate offset to address the location given by evaluating the expression. This will be a PC relative, pre-indexed address. If the address is out of range, an error will be generated.

(ii) A pre-indexed addressing specification:

**[Rn]** offset of zero

**[Rn,<#expression>]{!}** offset of <expression> bytes

(iii) A post-indexed addressing specification:

**[Rn],<#expression>** offset of <expression> bytes

Rn is an expression evaluating to a valid ARM7DI register number. Note, if Rn is R15 then the assembler will subtract 8 from the offset value to allow for ARM7DI pipelining.

{!} write back the base register (set the W bit) if ! is present

### 4.12.8 Examples

```
LDC      p1,c2,table          ; load c2 of coproc 1 from address table,
                              ; using a PC relative address.
STCEQL   p2,c3,[R5,#24]!      ; conditionally store c3 of coproc 2 into
                              ; an address 24 bytes up from R5, write this
                              ; address back to R5, and use long transfer
                              ; option (probably to store multiple words)
```

Note that though the address offset is expressed in bytes, the instruction offset field is in words. The assembler will adjust the offset appropriately.

## 4.13 Coprocessor register transfers (MRC, MCR)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 31: Coprocessor Register Transfer Instructions*.

This class of instruction is used to communicate information directly between ARM7DI and a coprocessor. An example of a coprocessor to ARM7DI register transfer (MRC) instruction would be a FIX of a floating point value held in a coprocessor, where the floating point number is converted into a 32 bit integer within the coprocessor, and the result is then transferred to ARM7DI register. A FLOAT of a 32 bit value in ARM7DI register into a floating point value within the coprocessor illustrates the use of ARM7DI register to coprocessor transfer (MCR).

An important use of this instruction is to communicate control information directly from the coprocessor into the ARM7DI CPSR flags. As an example, the result of a comparison of two floating point values within a coprocessor can be moved to the CPSR to control the subsequent flow of execution.

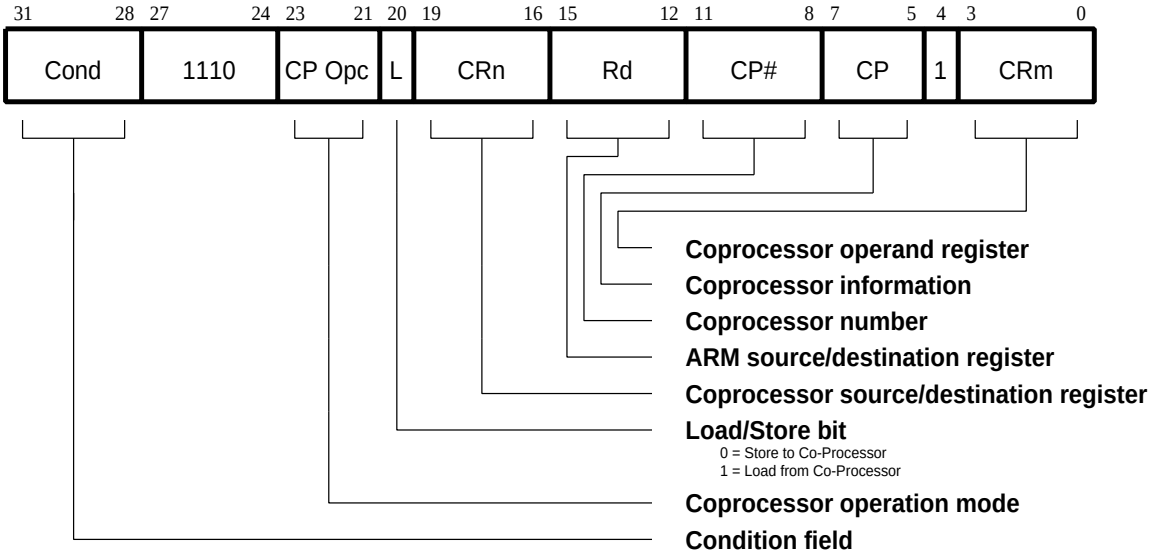



**Figure 31: Coprocessor Register Transfer Instructions**

### 4.13.1 The Coprocessor fields

The CP# field is used, as for all coprocessor instructions, to specify which coprocessor is being called upon.

The CP Opc, CRn, CP and CRm fields are used only by the coprocessor, and the interpretation presented here is derived from convention only. Other interpretations are allowed where the coprocessor functionality is incompatible with this one. The conventional interpretation is that the CP Opc and CP fields specify the operation the coprocessor is required to perform, CRn is the coprocessor register which is the source or destination of the transferred information, and CRm is a second coprocessor register which may be involved in some way which depends on the particular operation specified.

### 4.13.2 Transfers to R15

When a coprocessor register transfer to ARM7DI has R15 as the destination, bits 31, 30, 29 and 28 of the transferred word are copied into the N, Z, C and V flags respectively. The other bits of the transferred word are ignored, and the PC and other CPSR bits are unaffected by the transfer.

### 4.13.3 Transfers from R15

A coprocessor register transfer from ARM7DI with R15 as the source register will store the PC+12.

### 4.13.4 Instruction Cycle Times

MRC instructions take 1S + (b+1)I +1C incremental cycles to execute, where S, I and C are as defined in *Section 5.1: Cycle types* on page 71.

MCR instructions take 1S + bI +1C incremental cycles to execute.

*b* is the number of cycles spent in the coprocessor busy-wait loop.

### 4.13.5 Assembler syntax

**<MCR|MRC>{cond} p#,<expression1>,Rd,cn,cm{,<expression2>}**

MRC - move from coprocessor to ARM7DI register (L=1)

MCR - move from ARM7DI register to coprocessor (L=0)

{cond} - two character condition mnemonic, see *Figure 9: Condition Codes*

p# - the unique number of the required coprocessor

<expression1> - evaluated to a constant and placed in the CP Opc field

Rd is an expression evaluating to a valid ARM7DI register number

cn and cm are expressions evaluating to the valid coprocessor register numbers CRn and CRm respectively

<expression2> - where present is evaluated to a constant and placed in the CP field

### 4.13.6 Examples

```
MRC      2,5,R3,c5,c6          ; request coproc 2 to perform operation 5
                               ;  on c5 and c6, and transfer the (single
                               ;  32 bit word) result back to R3

MCR      6,0,R4,c6             ; request coproc 6 to perform operation 0
                               ;  on R4 and place the result in c6

MRCEQ    3,9,R3,c5,c6,2        ; conditionally request coproc 3 to perform
                               ;  operation 9 (type 2) on c5 and c6, and
                               ;  transfer the result back to R3
```

## 4.14 Undefined instruction

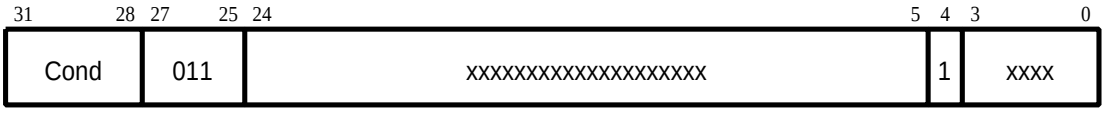



**Figure 32: Undefined Instruction**

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction format is shown in *Figure 32: Undefined Instruction*.

If the condition is true, the undefined instruction trap will be taken.

Note that the undefined instruction mechanism involves offering this instruction to any coprocessors which may be present, and all coprocessors must refuse to accept it by driving **CPA** and **CPB** HIGH.

### 4.14.1 Instruction cycle times

This cycle takes 2S + 1I + 1N,where S, N and I are as defined in *Section 5.1: Cycle types* on page 71.

### 4.14.2 Assembler syntax

At present the assembler has no mnemonics for generating this instruction. If it is adopted in the future for some specified use, suitable mnemonics will be added to the assembler. Until such time, this instruction shall not be used.

## 4.15 Instruction Set Examples

The following examples show ways in which the basic ARM7DI instructions can combine to give efficient code. None of these methods saves a great deal of execution time (although they may save some), mostly they just save code.

### 4.15.1 Using the conditional instructions

(1) using conditionals for logical OR

```
CMP     Rn,#p                ; if Rn=p OR Rm=q THEN GOTO Label
BEQ     Label
CMP     Rm,#q
BEQ     Label
```

can be replaced by

```
CMP     Rn,#p
CMPNE   Rm,#q                ; if condition not satisfied try other test
BEQ     Label
```

(2) absolute value

```
TEQ     Rn,#0                ; test sign
RSBMI   Rn,Rn,#0             ; and 2's complement if necessary
```

(3) multiplication by 4, 5 or 6 (run time)

```
MOV     Rc,Ra,LSL#2          ; multiply by 4
CMP     Rb,#5                ; test value
ADDCS   Rc,Rc,Ra             ; complete multiply by 5
ADDHI   Rc,Rc,Ra             ; complete multiply by 6
```

(4) combining discrete and range tests

```
TEQ     Rc,#127              ; discrete test
CMPNE   Rc,#" "-1            ; range test
MOVLS   Rc,#"."              ; IF   Rc<=" " OR Rc=ASCII(127)
                             ; THEN Rc:="."
```

(5) division and remainder

A number of divide routines for specific applications are provided in source form as part of the ANSI C library provided with the ARM Cross Development Toolkit, available from your supplier. A short general purpose divide routine follows.

```
                                        ; enter with numbers in Ra and Rb
                                        ;
      MOV      Rcnt,#1                  ; bit to control the division
Div1  CMP      Rb,#0x80000000           ; move Rb until greater than Ra
      CMPCC    Rb,Ra
      MOVCC    Rb,Rb,ASL#1
      MOVCC    Rcnt,Rcnt,ASL#1
      BCC      Div1
      MOV      Rc,#0
Div2  CMP      Ra,Rb                    ; test for possible subtraction
      SUBCS    Ra,Ra,Rb                 ; subtract if ok
      ADDCS    Rc,Rc,Rcnt               ; put relevant bit into result
      MOVS     Rcnt,Rcnt,LSR#1          ; shift control bit
      MOVNE    Rb,Rb,LSR#1              ; halve unless finished
      BNE      Div2
                                        ;
                                        ; divide result in Rc
                                        ; remainder in Ra
```

### 4.15.2 Pseudo random binary sequence generator

It is often necessary to generate (pseudo-) random numbers and the most efficient algorithms are based on shift generators with exclusive-OR feedback rather like a cyclic redundancy check generator. Unfortunately the sequence of a 32 bit generator needs more than one feedback tap to be maximal length (i.e. 2^32-1 cycles before repetition), so this example uses a 33 bit register with taps at bits 33 and 20. The basic algorithm is newbit:=bit 33 eor bit 20, shift left the 33 bit number and put in newbit at the bottom; this operation is performed for all the newbits needed (i.e. 32 bits). The entire operation can be done in 5 S cycles:

```
                                        ; enter with seed in Ra (32 bits),
                                          Rb (1 bit in Rb lsb), uses Rc
                                        ;
      TST      Rb,Rb,LSR#1              ; top bit into carry
      MOVS     Rc,Ra,RRX                ; 33 bit rotate right
      ADC      Rb,Rb,Rb                 ; carry into lsb of Rb
      EOR      Rc,Rc,Ra,LSL#12          ; (involved!)
      EOR      Ra,Rc,Rc,LSR#20          ; (similarly involved!)
                                        ;
                                        ; new seed in Ra, Rb as before
```

### 4.15.3 Multiplication by constant using the barrel shifter

(1) Multiplication by 2^n (1,2,4,8,16,32..)

```
      MOV      Ra, Rb, LSL #n
```

(2) Multiplication by 2^n+1 (3,5,9,17..)

```
      ADD      Ra,Ra,Ra,LSL #n
```

(3) Multiplication by 2^n-1 (3,7,15..)

```
RSB       Ra,Ra,Ra,LSL #n
```

(4) Multiplication by 6

```
ADD       Ra,Ra,Ra,LSL #1       ; multiply by 3
MOV       Ra,Ra,LSL#1           ; and then by 2
```

(5) Multiply by 10 and add in extra number

```
ADD       Ra,Ra,Ra,LSL#2        ; multiply by 5
ADD       Ra,Rc,Ra,LSL#1        ; multiply by 2 and add in next digit
```

(6) General recursive method for Rb := Ra*C, C a constant:

(a) If C even, say C = 2^n*D, D odd:

```
D=1:      MOV   Rb,Ra,LSL #n
D<>1:     {Rb := Ra*D}
          MOV      Rb,Rb,LSL #n
```

(b) If C MOD 4 = 1, say C = 2^n*D+1, D odd, n>1:

```
D=1:      ADD   Rb,Ra,Ra,LSL #n
D<>1:     {Rb := Ra*D}
          ADD      Rb,Ra,Rb,LSL #n
```

(c) If C MOD 4 = 3, say C = 2^n*D-1, D odd, n>1:

```
D=1:      RSB   Rb,Ra,Ra,LSL #n
D<>1:     {Rb := Ra*D}
          RSB      Rb,Ra,Rb,LSL #n
```

This is not quite optimal, but close. An example of its non-optimality is multiply by 45 which is done by:

```
RSB       Rb,Ra,Ra,LSL#2        ; multiply by 3
RSB       Rb,Ra,Rb,LSL#2        ; multiply by 4*3-1 = 11
ADD       Rb,Ra,Rb,LSL# 2       ; multiply by 4*11+1 = 45
```

rather than by:

```
ADD       Rb,Ra,Ra,LSL#3        ; multiply by 9
ADD       Rb,Rb,Rb,LSL#2        ; multiply by 5*9 = 45
```

### 4.15.4 Loading a word from an unknown alignment

```
                                ; enter with address in Ra (32 bits)
                                ; uses Rb, Rc; result in Rd.
                                ; Note d must be less than c e.g. 0,1
                                ;
BIC     Rb,Ra,#3                ; get word aligned address
LDMIA   Rb,{Rd,Rc}              ; get 64 bits containing answer
AND     Rb,Ra,#3                ; correction factor in bytes
MOVS    Rb,Rb,LSL#3             ; ...now in bits and test if aligned
MOVNE   Rd,Rd,LSR Rb            ; produce bottom of result word
                                ; (if not aligned)
RSBNE   Rb,Rb,#32               ; get other shift amount
ORRNE   Rd,Rd,Rc,LSL Rb         ; combine two halves to get result
```

### 4.15.5 Loading a halfword (Little Endian)

```
LDR     Ra, [Rb,#2]             ; Get halfword to bits 15:0
MOV     Ra,Ra,LSL #16           ; move to top
MOV     Ra,Ra,LSR #16           ; and back to bottom
                                ; use ASR to get sign extended version
```

### 4.15.6 Loading a halfword (Big Endian)

```
LDR     Ra, [Rb,#2]             ; Get halfword to bits 31:16
MOV     Ra,Ra,LSR #16           ; and back to bottom
                                ; use ASR to get sign extended version
```

# 5.0 Memory Interface

A separate 32 bit address bus specifies the memory location to be used for the transfer, and the **nRW** signal gives the direction of transfer (ARM7DI to memory or memory to ARM7DI). Control signals give additional information about the transfer cycle, and in particular they facilitate the use of DRAM page mode where applicable. Interfaces to static RAM based memories are not ruled out and, in general, they are much simpler than the DRAM interface described here.

## 5.1 Cycle types

All memory transfer cycles can be placed in one of four categories:

(1) Non-sequential cycle. ARM7DI requests a transfer to or from an address which is unrelated to the address used in the preceding cycle.

(2) Sequential cycle. ARM7DI requests a transfer to or from an address which is either the same as the address in the preceding cycle, or is one word after the preceding address.

(3) Internal cycle. ARM7DI does not require a transfer, as it is performing an internal function and no useful prefetching can be performed at the same time.

(4) Coprocessor register transfer. ARM7DI wishes to use the data bus to communicate with a coprocessor, but does not require any action by the memory system.

These four classes are distinguishable to the memory system by inspection of the **nMREQ** and **SEQ** control lines (see *Table 6: Memory Cycle Types*). These control lines are generated during phase 1 of the cycle before the cycle whose characteristics they forecast, and this pipelining of the control information gives the memory system sufficient time to decide whether or not it can use a page mode access.

| nMREQ | SEQ | Cycle type | |
|---|---|---|---|
| 0 | 0 | Non-sequential cycle | (N-cycle) |
| 0 | 1 | Sequential cycle | (S-cycle) |
| 1 | 0 | Internal cycle | (I-cycle) |
| 1 | 1 | Coprocessor register transfer | (C-cycle) |

**Table 6: Memory Cycle Types**

*Figure 33: ARM Memory Cycle Timing* shows the pipelining of the control signals, and suggests how the DRAM address strobes (**nRAS** and **nCAS**) might be timed to use page mode for S-cycles. Note that the N-cycle is longer than the other cycles. This is to allow for the DRAM precharge and row access time, and is not an ARM7DI requirement.

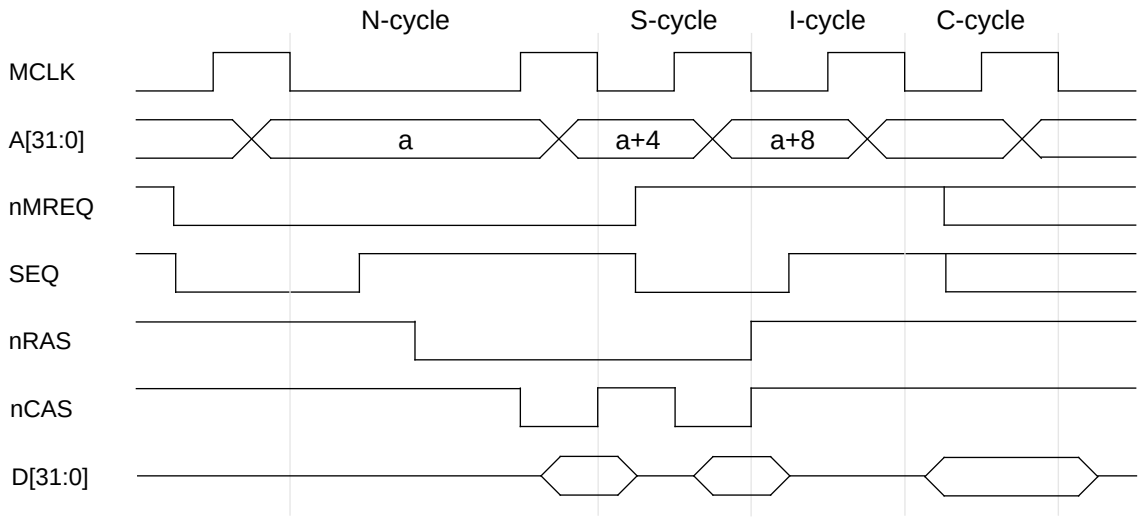



**Figure 33: ARM Memory Cycle Timing**

When an S-cycle follows an N-cycle, the address will always be one word greater than the address used in the N-cycle. This address (marked “a” in the above diagram) should be checked to ensure that it is not the last in the DRAM page before the memory system commits to the S-cycle. If it is at the page end, the S-cycle cannot be performed in page mode and the memory system will have to perform a full access.

The processor clock must be stretched to match the full access. When an S-cycle follows an I- or C-cycle, the address will be the same as that used in the I- or C-cycle. This fact may be used to start the DRAM access during the preceding cycle, which enables the S-cycle to run at page mode speed whilst performing a full DRAM access. This is shown in *Figure 34: Memory Cycle Optimization*.

## 5.2 Byte addressing

The processor address bus gives byte addresses, but instructions are always words (where a word is 4 bytes) and data quantities are usually words. Single data transfers (LDR and STR) can, however, specify that a byte quantity is required. The **nBW** control line is used to request a byte from the memory system; normally it is HIGH, signifying a request for a word quantity, and it goes LOW during phase 2 of the preceding cycle to request a byte transfer.

When the processor is fetching an instruction from memory, the state of the bottom two address lines **A[1:0]** is undefined.

When a byte is requested in a read transfer (LDRB), the memory system can safely ignore that the request is for a byte quantity and present the whole word.

ARM7DI will perform the byte extraction internally. Alternatively, the memory system may activate only the addressed byte of the memory. This may be desirable in order to save power, or to enable the use of a common decoding system for both read and write cycles.

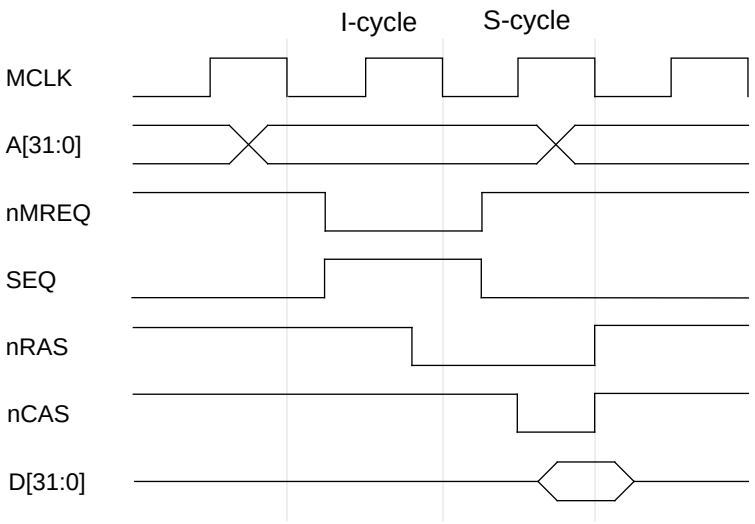



**Figure 34: Memory Cycle Optimization**

If a byte write is requested (STRB), ARM7DI will broadcast the byte value across the data bus, presenting it at each byte location within the word. The memory system must decode **A[1:0]** to enable writing only to the addressed byte.

One way of implementing the byte decode in a DRAM system is to separate the 32 bit wide block of DRAM into four byte wide banks, and generate the column address strobes independently as shown in *Figure 35: Decoding Byte Accesses to Memory*.

When the processor is configured for Little Endian operation byte 0 of the memory system should be connected to data lines 7 through 0 (**D[7:0]**) and strobed by **nCAS0**. **nCAS1** drives the bank connected to data lines 15 though 8, and so on. This has the added advantage of reducing the load on each column strobe driver, which improves the precision of this time critical signal.

In the Big Endian case, byte 0 of the memory system should be connected to data lines 31 through 24.

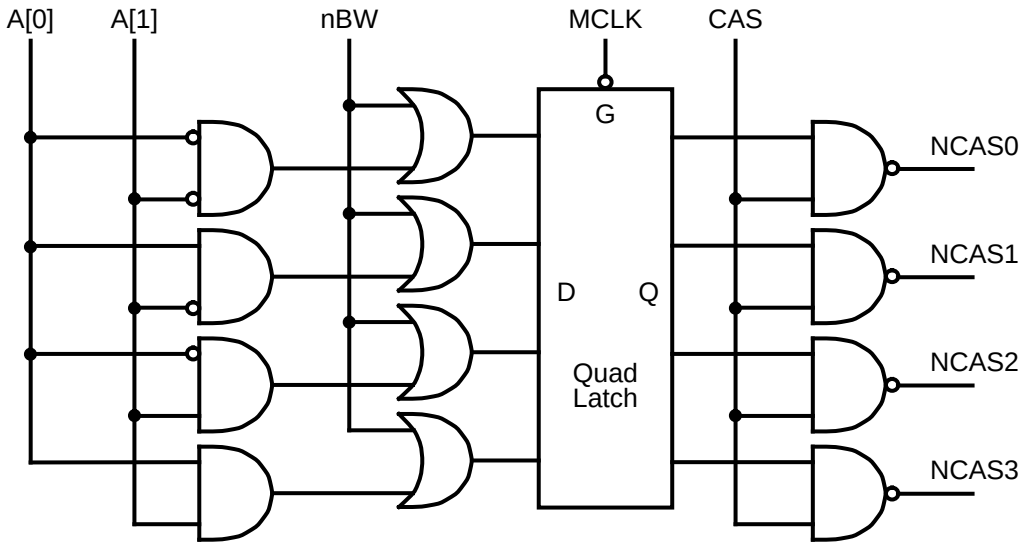



**Figure 35: Decoding Byte Accesses to Memory**

## 5.3 Address timing

Normally the processor address changes during phase 2 to the value which the memory system should use during the following cycle. This gives maximum time for driving the address to large memory arrays, and for address translation where required. Dynamic memories usually latch the address on chip, and if the latch is timed correctly they will work even though the address changes before the access has completed.

Static RAMs and ROMs will not work under such circumstances, as they require the address to be stable until after the access has completed. Therefore, for use with such devices, the address transition must be delayed until after the end of phase 2. An on-chip address latch, controlled by **ALE**, allows the address timing to be modified in this way. In a system with a mixture of static and dynamic memories (which for these purposes means a mixture of devices with and without address latches), the use of **ALE** may change dynamically from one cycle to the next, at the discretion of the memory system.

## 5.4 Memory management

The ARM7DI address bus may be processed by an address translation unit before being presented to the memory, and ARM7DI is capable of running a virtual memory system. The abort input to the processor may be used by the memory manager to inform ARM7DI of page faults. Various other signals enable different page protection levels to be supported:

(1) **nRW** can be used by the memory manager to protect pages from being written to.

(2) **nTRANS** indicates whether the processor is in user or a privileged mode, and may be used to protect system pages from the user, or to support completely separate mappings for the system and the user.

Address translation will normally only be necessary on an N-cycle, and this fact may be exploited to reduce power consumption in the memory manager and avoid the translation delay at other times. The times when translation is necessary can be deduced by keeping track of the cycle types that the processor uses.

If an N-cycle is matched to a full DRAM access, it will be longer than the minimum processor cycle time. Stretching phase 1 rather than phase 2 will give the translation system more time to generate an abort.

## 5.5 Locked operations

ARM7DI includes a data swap (SWP) instruction that allows the contents of a memory location to be swapped with the contents of a processor register. This instruction is implemented as an uninterruptable pair of accesses; the first access reads the contents of the memory, and the second writes the register data to the memory. These accesses must be treated as a contiguous operation by the memory controller to prevent another device from changing the affected memory location before the swap is completed. ARM7DI drives the **LOCK** signal HIGH for the duration of the swap operation to warn the memory controller not to give the memory to another device.

## 5.6 Stretching access times

All memory timing is defined by **MCLK**, and long access times can be accommodated by stretching this clock. It is usual to stretch the LOW period of **MCLK**, as this allows the memory manager to abort the operation if the access is eventually unsuccessful.

Either **MCLK** can be stretched before it is applied to ARM7DI, or the **nWAIT** input can be used together with a free-running **MCLK**. Taking **nWAIT** LOW has the same effect as stretching the LOW period of **MCLK**, and **nWAIT** must only change when **MCLK** is LOW.

ARM7DI does not contain any dynamic logic which relies upon regular clocking to maintain its internal state. Therefore there is no limit upon the maximum period for which **MCLK** may be stretched, or **nWAIT** held LOW.

## 5.7 The External Data Bus

ARM7DI has a single, bidirectional data bus. Most of the time, the ARM reads from memory and so this bus is configured to input. During write cycles however, the ARM7DI must output data. During phase 2 of the previous cycle, the signal **nRW** is driven HIGH to indicate a write cycle. During the actual cycle, **nENOUT** is driven LOW to indicate that the ARM7DI is driving **D[31:0]** as an output. *Figure 36: Data Write Bus Cycle* shows this bus timing (**DBE** has been tied HIGH in this example). *Figure 37: ARM7DI Data Bus Control Circuit* shows the circuit which exists in ARM7DI for controlling exactly when the external bus is driven out.

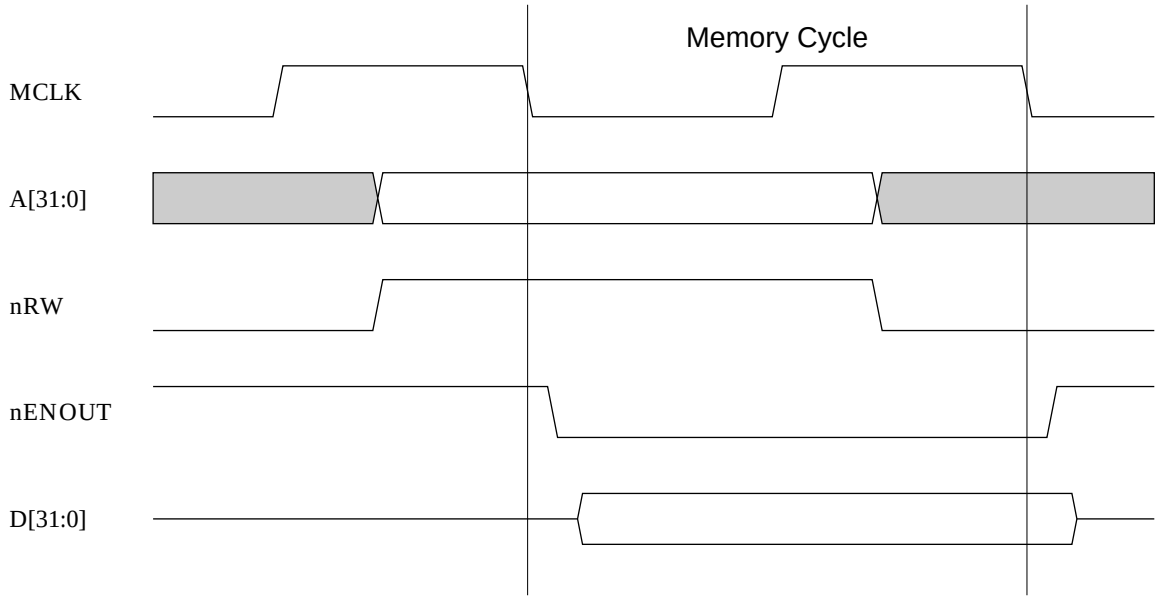



**Figure 36: Data Write Bus Cycle**

The ARM7DI macrocell has an additional bus control signal, **nENIN,** which allows the external system to manually tristate the bus. In the simplest systems, **nENIN** can be tied LOW and **nENOUT** can be ignored. However, in many applications when the external data bus is a shared resource, greater control may be required. In this situation, **nENIN** can be used to delay when the external bus is driven. Note that for backwards compatibility, **DBE** is also included. At the macrocell level,**DBE** and **nENIN** have almost identical functionality and in most applications one can be tied off.

*Section 5.7.1 Example System: The ARM70D* describes how ARM7DI may be interfaced to an external data bus, using ARM70D as an example.

ARM7DI has another output control signal called **TBE**. This signal is normally only used during test and must be tied HIGH when not in use. When driven LOW, **TBE** forces all three-stateable outputs to high impedance. It is as if both **DBE** and **ABE** have been driven LOW, causing the data bus, the address bus, and all other signals normally controlled by **ABE** to become high impedance. Note, however, that there is no scan cell on **TBE**. Thus, **TBE** is completely independent of scan data and may be used to put the outputs into a high impedance state while scan testing takes place.

*Table 7: Output Enable Control Summary* shows the tri-state control of ARM7DI's outputs. Those signals which do not have an asterix in the **ABE**, **DBE** or **TBE** column cannot be driven to the high impedance state:

| ARM7DI output | ABE | DBE | TBE |
|---|---|---|---|
| **A[31:0]** | * | | * |
| **D[31:0]** | | * | * |
| **nBW** | * | | * |
| **nRW** | * | | * |
| **LOCK** | * | | * |
| **nOPC** | * | | * |
| **nTRANS** | * | | * |
| **DBGACK** | | | |
| **ECLK** | | | |
| **nCPI** | | | |
| **nENOUT** | | | |
| **nEXEC** | | | |
| **nM[4:0]** | | | |
| **nMREQ** | | | |
| **SDOUTMS** | | | |
| **SDOUTDATA** | | | |
| **SEQ** | | | |

**Table 7: Output Enable Control Summary**

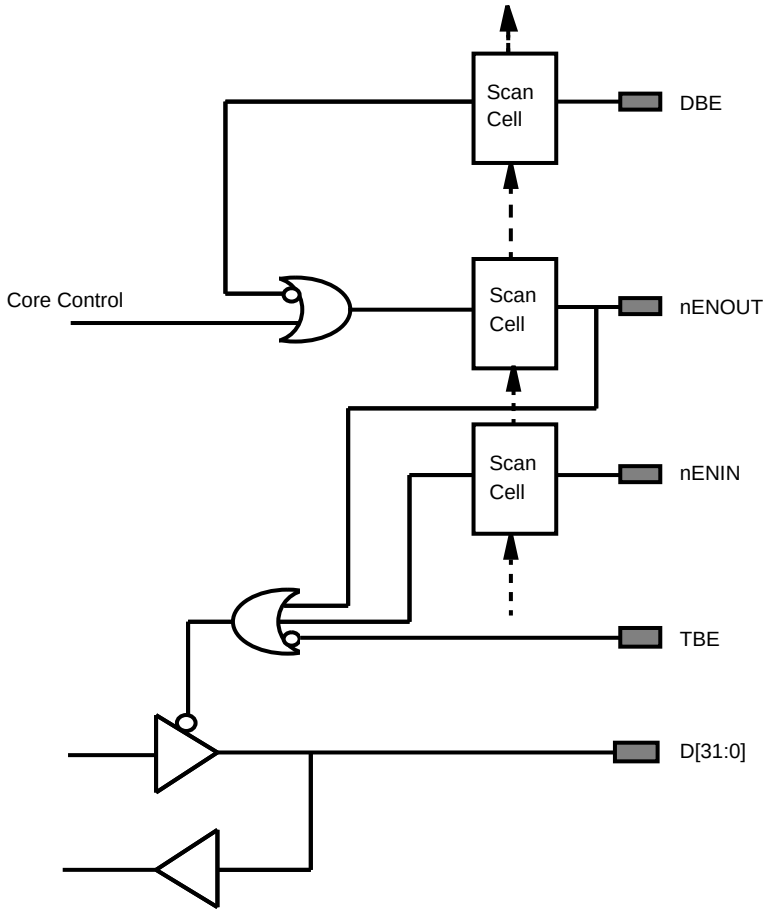



**Figure 37: ARM7DI Data Bus Control Circuit**

### 5.7.1 Example System: The ARM70D

Connecting ARM7DI's data bus, **D[31:0]** to an external shared bus requires some simple additional logic. This will vary from application to application. As an example, the following describes how the ARM7D macrocell was connected to the bi-directional data bus pads of the ARM70D.

In this application, care must be taken to prevent bus clash on **D[31:0]** when the data bus drive changes direction. The timing of **nENIN**, and the pad control signals must be arranged so that when the core starts to drive out, the pad drive onto **D[31:0]** switches off before the core starts to drive. Similarly, when the bus switches back to input, the core must stop driving before the pad switches on.

All this can be achieved using a simple non-overlapping clock generator. The actual circuit implemented in the ARM70D is shown in *Figure 38: The ARM70D Data Bus Circuit*. Note that at the core level, **TBE** and **DBE** are tied HIGH (inactive). This is because in a packaged part, there is no need to ever manually force the internal buses into a high impedance state. Note also that at the pad level, the signal **EDBE** is factored into the bus control logic. This allows the external memory controller to arbitrate the bus and asynchronously disable ARM70D if required.

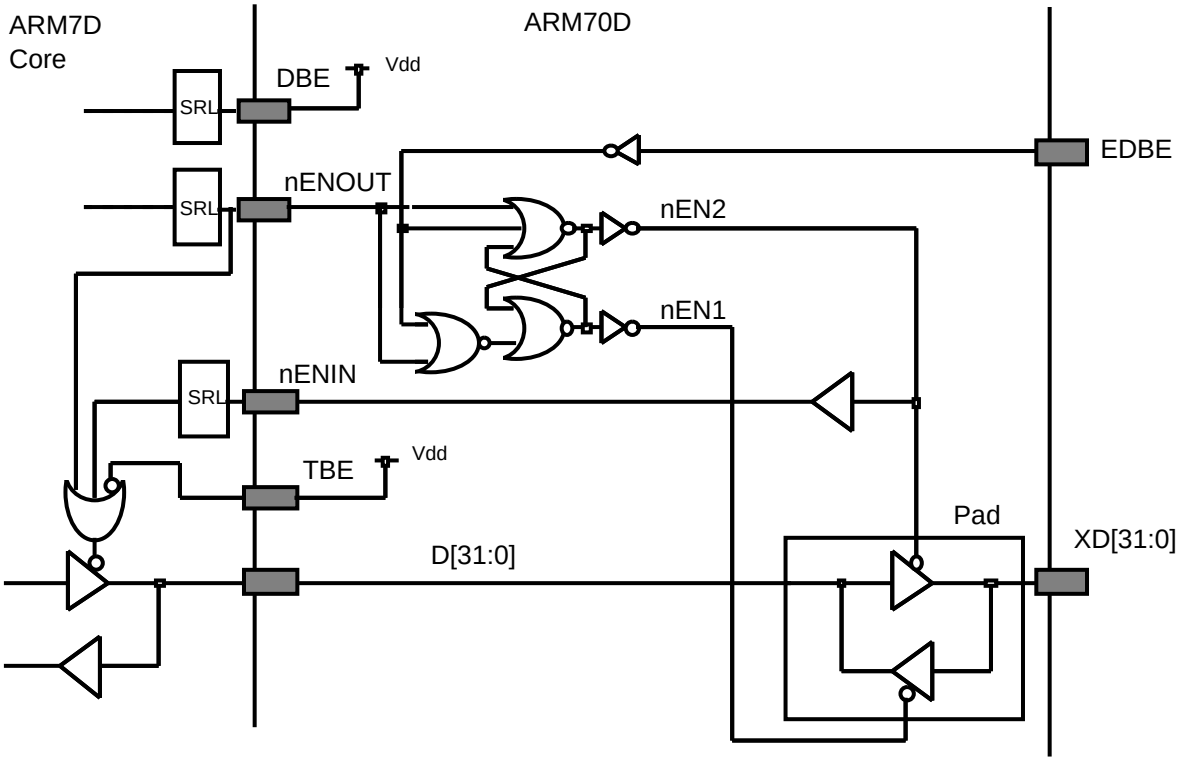



**Figure 38: The ARM70D Data Bus Circuit**

*Figure 39: Data Bus Control Signal Timing* shows how the various control signals interact. Under normal conditions, when the data bus is configured as input, **nENOUT** is HIGH, **nEN1** is LOW, and **nEN2**/**nENIN** is HIGH. Thus the pads drive **XD[31:0]** onto **D[31:0]**.

When a write cycle occurs, **nRW** is driven HIGH to indicate a write during phase 2 of the previous cycle, (ie, with the address). During phase 1 of the actual cycle, **nENOUT** is driven LOW to indicate that ARM7DI is about to drive the bus. The falling edge of this signal makes **nEN1** go HIGH, which disables the input half pad from driving **D[31:0]**. This in turn makes **nEN2** go LOW, which enables the output half of the pad so that the ARM70D is now driving the external data bus, **XD[31:0]**. **nEN2** is then buffered and driven back into the core on **nENIN,** so that finally the ARM7DI macrocell drives **D[31:0]**. The delay between all the signals ensures that there is no clash on the data bus as it changes direction from input to output.

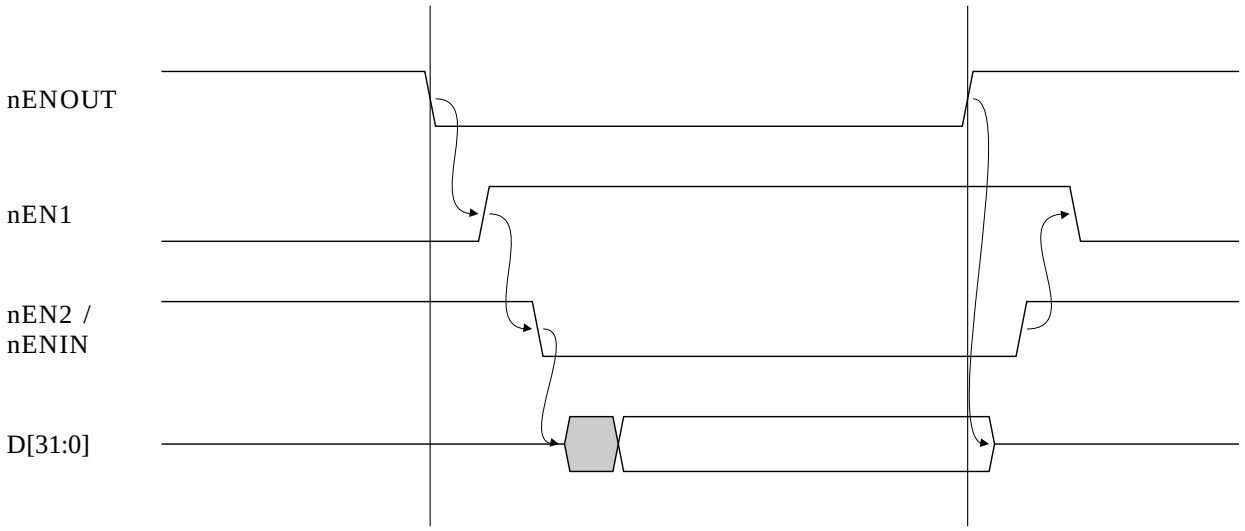



**Figure 39: Data Bus Control Signal Timing**

When the bus turns around to the other direction at the end of the cycle, the various control signals switch the other way. Again, the non-overlap ensures that there is never a bus clash. This time, **nENOUT** is driven HIGH to denote that ARM7DI no longer needs to drive the bus and the core's output is immediately switched off. This causes **nEN2** to disable the output half of the pad which in turn causes **nEN1** to switch on the input half. Thus, the bus is back to its original input configuration.

Note that the data out time of ARM7DI is not directly determined by **nENOUT** and **nENIN**, and so delaying exactly when the bus is driven will not affect the propagation delay. Please refer to *Chapter 11.0 AC Parameters* for timing details.

# 6.0 Coprocessor Interface

The functionality of the ARM7DI instruction set may be extended by the addition of up to 16 external coprocessors. When the coprocessor is not present, instructions intended for it will trap, and suitable software may be installed to emulate its functions. Adding the coprocessor will then increase the system performance in a software compatible way. Note that some coprocessor numbers have already been assigned. Contact ARM Ltd for up to date information.

## 6.1 Interface signals

Three dedicated signals control the coprocessor interface, **nCPI**, **CPA** and **CPB**. The **CPA** and **CPB** inputs should be driven HIGH except when they are being used for handshaking.

### 6.1.1 Coprocessor present/absent

ARM7DI takes **nCPI** LOW whenever it starts to execute a coprocessor (or undefined) instruction. (This will not happen if the instruction fails to be executed because of the condition codes.) Each coprocessor will have a copy of the instruction, and can inspect the CP# field to see which coprocessor it is for. Every coprocessor in a system must have a unique number and if that number matches the contents of the CP# field the coprocessor should drive the **CPA** (coprocessor absent) line LOW. If no coprocessor has a number which matches the CP# field, **CPA** and **CPB** will remain HIGH, and ARM7DI will take the undefined instruction trap. Otherwise ARM7DI observes the **CPA** line going LOW, and waits until the coprocessor is not busy.

### 6.1.2 Busy-waiting

If **CPA** goes LOW, ARM7DI will watch the **CPB** (coprocessor busy) line. Only the coprocessor which is driving **CPA** LOW is allowed to drive **CPB** LOW, and it should do so when it is ready to complete the instruction. ARM7DI will busy-wait while **CPB** is HIGH, unless an enabled interrupt occurs, in which case it will break off from the coprocessor handshake to process the interrupt. Normally ARM7DI will return from processing the interrupt to retry the coprocessor instruction.

When **CPB** goes LOW, the instruction continues to completion. This will involve data transfers taking place between the coprocessor and either ARM7DI or memory, except in the case of coprocessor data operations which complete immediately the coprocessor ceases to be busy.

All three interface signals are sampled by both ARM7DI and the coprocessor(s) on the rising edge of **MCLK**. If all three are LOW, the instruction is committed to execution, and if transfers are involved they will start on the next cycle. If **nCPI** has gone HIGH after being LOW, and before the instruction is committed, ARM7DI has broken off from the busy-wait state to service an interrupt. The instruction may be restarted later, but other coprocessor instructions may come sooner, and the instruction should be discarded.

### 6.1.3 Pipeline following

In order to respond correctly when a coprocessor instruction arises, each coprocessor must have a copy of the instruction. All ARM7DI instructions are fetched from memory via the main data bus, and coprocessors are connected to this bus, so they can keep copies of all instructions as they go into the ARM7DI pipeline. The **nOPC** signal indicates when an instruction fetch is taking place, and **MCLK** gives the timing of the transfer, so these may be used together to load an instruction pipeline within the coprocessor.

## 6.2 Data transfer cycles

Once the coprocessor has gone not-busy in a data transfer instruction, it must supply or accept data at the ARM7DI bus rate (defined by **MCLK**). It can deduce the direction of transfer by inspection of the L bit in the instruction, but must only drive the bus when permitted to by **DBE** being HIGH. The coprocessor is responsible for determining the number of words to be transferred; ARM7DI will continue to increment the address by one word per transfer until the coprocessor tells it to stop. The termination condition is indicated by the coprocessor driving **CPA** and **CPB** HIGH.

There is no limit in principle to the number of words which one coprocessor data transfer can move, but by convention no coprocessor should allow more than 16 words in one instruction. More than this would worsen the worst case ARM7DI interrupt latency, as the instruction is not interruptible once the transfers have commenced. At 16 words, this instruction is comparable with a block transfer of 16 registers, and therefore does not affect the worst case latency.

## 6.3 Register transfer cycle

The coprocessor register transfer cycle is the one case when ARM7DI requires the data bus without requiring the memory to be active. The memory system is informed that the bus is required by ARM7DI taking both **nMREQ** and **SEQ** HIGH. When the bus is free, **DBE** should be taken HIGH to allow ARM7DI or the coprocessor to drive the bus, and an **MCLK** cycle times the transfer.

## 6.4 Privileged instructions

The coprocessor may restrict certain instructions for use in privileged modes only. To do this, the coprocessor will have to track the **nTRANS** output.

As an example of the use of this facility, consider the case of a floating point coprocessor (FPU) in a multi-tasking system. The operating system could save all the floating point registers on every task switch, but this is inefficient in a typical system where only one or two tasks will use floating point operations. Instead, there could be a privileged instruction which turns the FPU on or off. When a task switch happens, the operating system can turn the FPU off without saving its registers. If the new task attempts an FPU operation, the FPU will appear to be absent, causing an undefined instruction trap. The operating system will then realise that the new task requires the FPU, so it will re-enable it and save FPU registers. The task can then use the FPU as normal. If, however, the new task never attempts an FPU operation (as will be the case for most tasks), the state saving overhead will have been avoided.

## 6.5 Idempotency

A consequence of the implementation of the coprocessor interface, with the interruptible busy-wait state, is that all instructions may be interrupted at any point up to the time when the coprocessor goes not-busy. If so interrupted, the instruction will normally be restarted from the beginning after the interrupt has been processed. It is therefore essential that any action taken by the coprocessor before it goes not-busy must be idempotent, ie must be repeatable with identical results.

For example, consider a FIX operation in a floating point coprocessor which returns the integer result to an ARM7DI register. The coprocessor must stay busy while it performs the floating point to fixed point conversion, as ARM7DI will expect to receive the integer value on the cycle immediately following that where it goes not-busy. The coprocessor must therefore preserve the original floating point value and not corrupt it during the conversion, because it will be required again if an interrupt arises during the busy period.

The coprocessor data operation class of instruction is not generally subject to idempotency considerations, as the processing activity can take place after the coprocessor goes not-busy. There is no need for ARM7DI to be held up until the result is generated, because the result is confined to stay within the coprocessor.

## 6.6 Undefined instructions

Undefined instructions are treated by ARM7DI as coprocessor instructions. All coprocessors must be absent (ie **CPA** and **CPB** must be HIGH) when an undefined instruction is presented. ARM7DI will then take the undefined instruction trap. Note that the coprocessor need only look at bit 27 of the instruction to differentiate undefined instructions (which all have 0 in bit 27) from coprocessor instructions (which all have 1 in bit 27).

# 7.0 Debug Interface

The debug interface is based on the IEEE Std. 1149.1- 1990, “Standard Test Access Port and Boundary-Scan Architecture” (please refer to this standard for an explanation of the terms used in this section and for a description of the TAP controller states.)

## 7.1 Overview

ARM7DI contains hardware support for advanced debugging features. This is intended to ease the user's development of application software, operating systems, and the hardware itself.

Simplistically, ARM7DI's debug extensions allow the core to be stopped on a given instruction fetch (breakpoint), or data access (watchpoint), or asynchronously by a debug-request. When this happens, ARM7DI is said to be in *debug state*. At this point, the core's internal state and the system's external state may be examined. Once examination is complete, the core and system state may be restored and program execution may resume.

ARM7DI is forced into debug state either by a request on one of the external debug interface signals, or by an internal functional unit known as *ICEBreaker*. Once in debug state, the core isolates itself from the memory system. The core can then be examined while all other system activity continues as normal.

ARM7DI's internal state is examined via a JTAG-style serial interface. This allows instructions to be serially inserted into the core's pipeline without using the external data bus. Thus, when in debug state, a store-multiple (STM) could be inserted into the instruction pipeline and this would dump the contents of ARM7DI's registers. This data can be serially shifted out without affecting the rest of the system.

## 7.2 Debug Systems

The ARM7DI forms one component of a debug system that interfaces from the high-level debugging performed by the user to the low-level interface supported by ARM7DI. Such a system will typically have three parts:

1. *The Debug Host*

   This is a computer - for example a PC - running a software debugger such as ARMSD. It allows the user to issue high level commands such as “set breakpoint at location XX”, or “examine the contents of memory from 0x0 to 0x100”.

2. *The Protocol Converter*

   The Debug Host will be connected to the ARM7DI development system via an interface such as RS232. The messages broadcast over this connection must be converted to the interface signals of the ARM7DI, and this function is performed by the protocol converter.

3. *ARM7DI*

   ARM7DI is the lowest level of the system, containing hardware extensions to ease debugging. The debug extensions allow the user to stall the core from program execution, examine its internal state and the state of the memory system, and then resume program execution.

These are shown in *Figure 40: Typical debug system*.

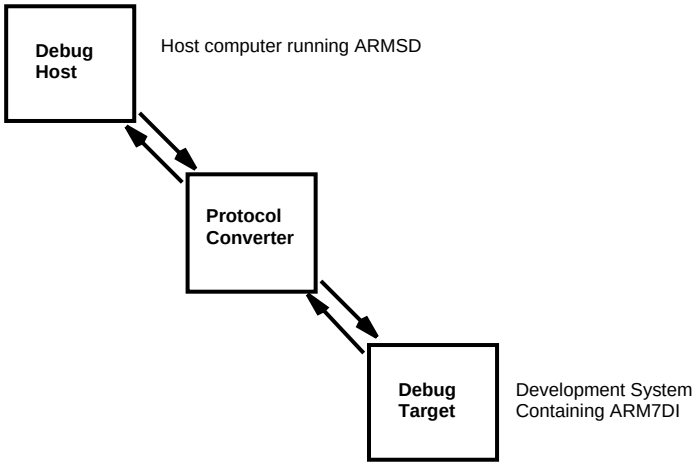



**Figure 40: Typical debug system**

The anatomy of ARM7DI is shown in *Figure 43: ARM7DI Scan Chain Arrangement*. The major blocks are:

a) *ARM7DM*

This is the CPU core, with hardware support for debug.

b) *ICEBreaker*

This is a set of registers and comparators used to generate debug exceptions (eg breakpoints). This unit is described in full later in this chapter.

c) *TAP controller*

This controls the action of the scan chains via a JTAG serial interface.

The Debug Host and the Protocol Converter are system dependent. The rest of this chapter is given over to the hardware debug extensions of ARM7DI.

## 7.3 Debug Interface Signals

There are three primary external signals associated with the debug interface:

- **BREAKPT** and **DBGRQ**
  with which the system requests ARM7DI to enter debug state.
- **DBGACK**
  which ARM7DI uses to flag back to the system that it is in debug state.

### 7.3.1 Entry into Debug State

ARM7DI is forced into debug state after a breakpoint, watchpoint or debug-request has occurred.

Conditions under which a breakpoint or watchpoint occur can be programmed using ICEBreaker. Alternatively, external logic can monitor the address and data bus, and flag breakpoints and watchpoints via the **BREAKPT** pin.

The timing is the same for externally generated breakpoints and watchpoints. Data must always be valid around the falling edge of **MCLK**. If this data is an instruction to be breakpointed, then the **BREAKPT** signal must be HIGH around the next rising edge of **MCLK**. Similarly, if the data is for a load or store, then this can be marked as watchpointed by asserting **BREAKPT** around the next rising edge of **MCLK**.

When a breakpoint or watchpoint is generated, there may be a delay before ARM7DI enters debug state. When it does, the signal **DBGACK** is asserted in the HIGH phase of **MCLK**. The timing for an externally generated breakpoint is shown in *Figure 41: Debug State Entry*.

***Entry into debug state on breakpoint***

After an instruction has been breakpointed, the core does not enter debug state immediately. Instructions are marked as being breakpointed as they enter ARM7DI's instruction pipeline. Thus ARM7DI only enters debug state when (and if) the instruction reaches the pipeline's execute stage.

A breakpointed instruction may not cause ARM7DI to enter debug state for one of two reasons:

- *a branch precedes the breakpointed instruction*.

  When the branch is executed, the instruction pipeline is flushed and the breakpoint is cancelled.

- *an exception has occurred.*

  Again, the instruction pipeline is flushed and the breakpoint is cancelled. However, the normal way to exit from an exception is to branch back to the instruction that would have executed next. This involves refilling the pipeline, and so the breakpoint can be re-flagged.

When a breakpointed conditional instruction reaches the execute stage of the pipeline, the breakpoint is *always* taken and ARM7DI enters debug state, regardless of whether the condition was met.

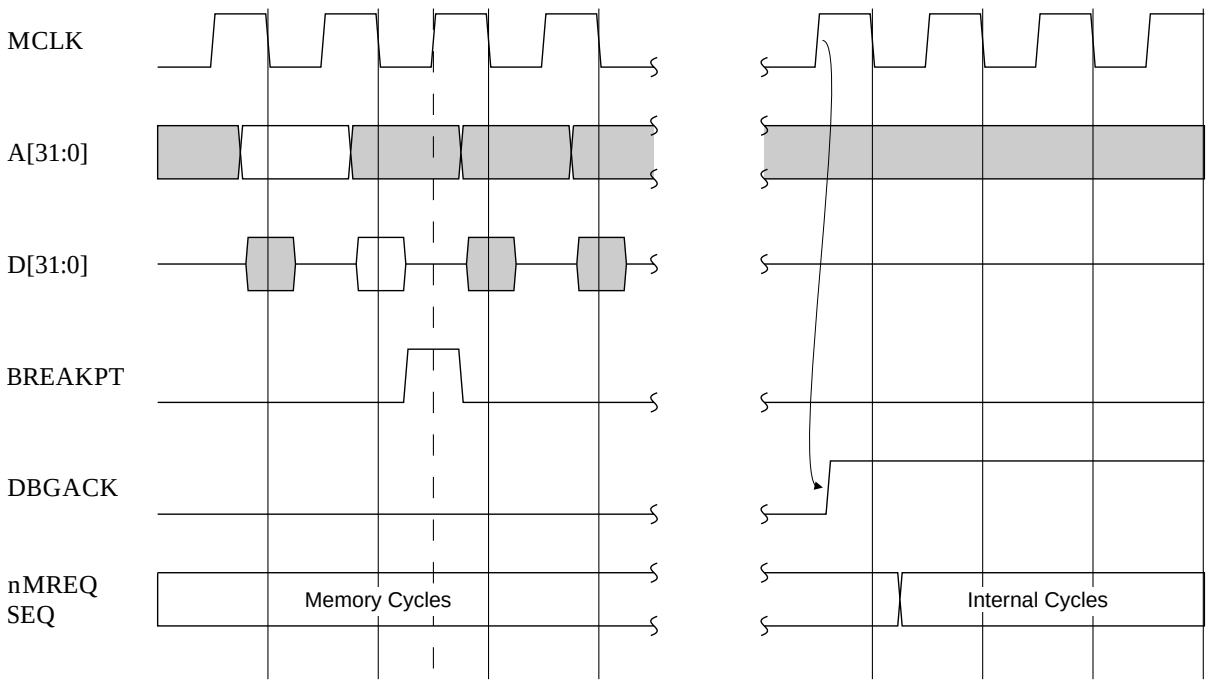



**Figure 41: Debug State Entry**

Breakpointed instructions *do not* get executed: instead, ARM7DI enters debug state. Thus, when the internal state is examined, the state *before* the breakpointed instruction is seen. Once examination is complete, the breakpoint should be removed and program execution restarted from the previously breakpointed instruction.

***Entry into debug state on watchpoint***

Watchpoints occur on data accesses. A watchpoint is always taken, but the core may not enter debug state immediately. In all cases, the current instruction will complete. If this is a multi-word load or store (LDM or STM), then many cycles may elapse before the watchpoint is taken.

Watchpoints can be thought of as being similar to data aborts. The difference is however that if a data abort occurs, then although the instruction completes, all subsequent changes to ARM7DI's state are prevented. This allows the cause of the abort to be cured by the abort handler, and the instruction re-executed. This is not so in the case of a watchpoint. Here, the instruction completes and all changes to the core's state occur (ie load data is written into the destination registers, and base write-back occurs). Thus the instruction does not need to be restarted

Watchpoints are *always* taken. If an exception is pending when a watchpoint occurs, then the core enters debug state in the mode of that exception.

***Entry into debug state on debug-request***

ARM7DI may also be forced into debug state on debug request. This can be done either through ICEBreaker programming (see *Chapter 8.0: The ARM7DI ICEBreaker Module* on page 109), or by the assertion of the **DBGRQ** pin. This pin is an asynchronous input and is thus synchronised by logic inside ARM7DI before it takes effect. Following synchronisation, the core will normally enter debug state at the end of the current instruction. However, if the current instruction is a busy-waiting access to a coprocessor, the instruction terminates and ARM7DI enters debug state immediately (this is similar to the action of **nIRQ** and **nFIQ**).

***Action of ARM7DI in debug state***

Once ARM7DI is in debug state, **nMREQ** and **SEQ** are forced to indicate internal cycles. This allows the rest of the memory system to ignore ARM7DI and functions as normal. Since the rest of the system continues operation, ARM7DI must be forced to ignore aborts and interrupts.

When instructions are executed in debug state, ARM7DI outputs (except **nMREQ** and **SEQ**) will change asynchronously to the memory system. For example, every time a new instruction is scanned into the pipeline, the address bus will change. Although this is asynchronous it should not affect the system, since **nMREQ** and **SEQ** are forced to indicate internal cycles regardless of what the rest of ARM7DI is doing. The memory controller must be designed to ensure that this asynchronous behaviour does not affect the rest of the system.

## 7.4 Scan Chains and JTAG Interface

### 7.4.1 Overview

There are three JTAG style scan chains inside ARM7DI. These allow testing, debugging and ICEBreaker programming. The scan chains are controlled from a JTAG style TAP (Test Access Port) controller. For further details of the JTAG specification, please refer to the IEEE Standard 1149.1 - 1990 "*Standard Test Access Port and Boundary-Scan Architecture*". Note that the scan cells are not fully JTAG compliant. The following sections describe the limitations on their use.

The three scan paths are referred to as scan chain 0,1 and 2: these are shown in *Figure 43: ARM7DI Scan Chain Arrangement*.

***Scan chain 0***

Scan chain 0 allows access to the entire periphery of the ARM7DI core, including the data bus. The scan chain functions allow inter-device testing (EXTEST) and serial testing of the core (INTEST).

The order of the scan chain (from **SDIN** to **SDOUTMS**) is: data bus bits 0 through 31, the control signals, followed by the address bus bits 31 through 0. See *Table 9: Scan Signals and Pins* on page 108 for a detailed listing of the scan order.

***Scan chain 1***

Scan chain 1 is a subset of the signals that are accessible through scan chain 0. Access to the core's data bus **D[31:0]**, and the **BREAKPT** signal is available serially. There are 33 bits in this scan chain, the order being (from serial data in to out): data bus bits 0 through 31, followed by **BREAKPT**.

***Scan Chain 2***

This scan chain simply allows access to the ICEBreaker registers. Refer to *Chapter 8.0: The ARM7DI ICEBreaker Module* on page 109 for details.

### 7.4.2 The JTAG State Machine

The process of serial test and debug is best explained in conjunction with the JTAG state machine. *Figure 42: Test Access Port (TAP) Controller State Transitions* shows the state transitions that occur in the TAP controller.

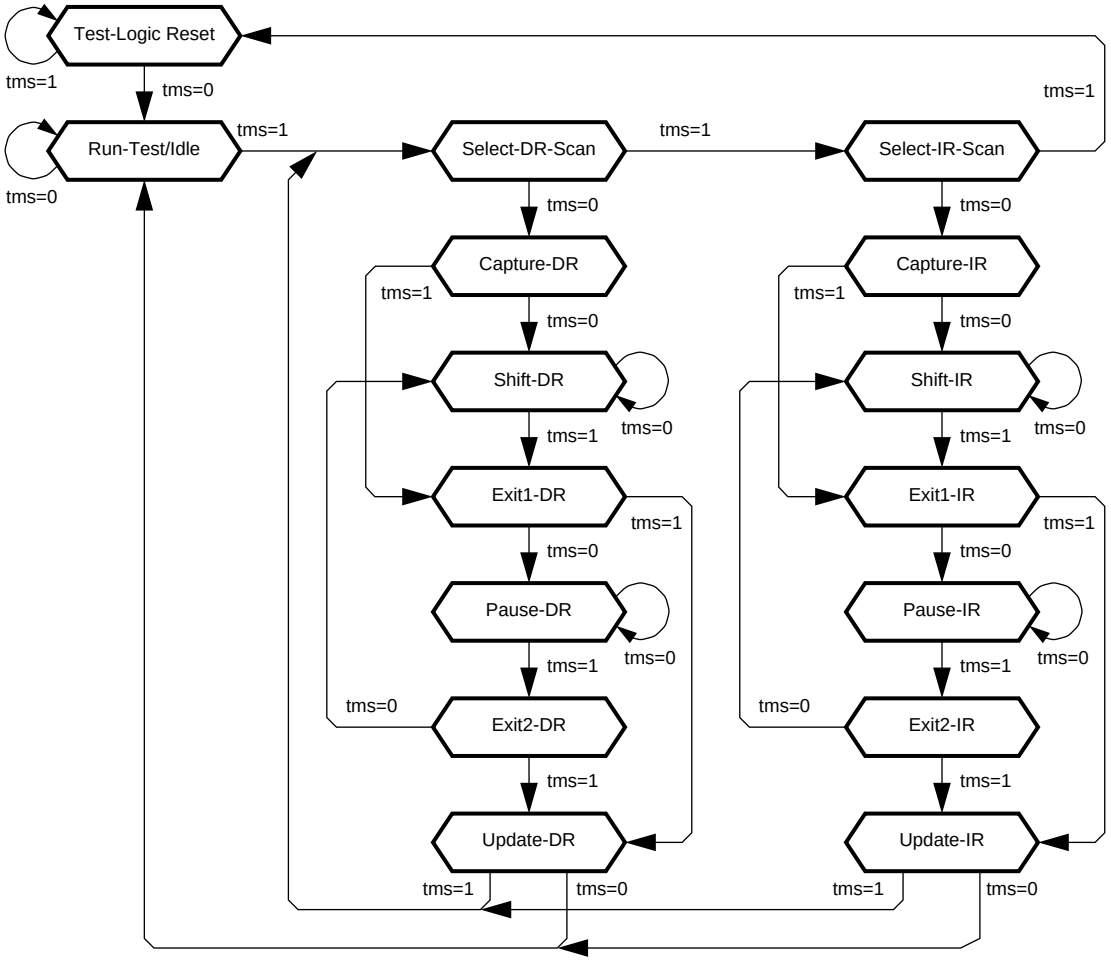



**Figure 42: Test Access Port (TAP) Controller State Transitions**

## 7.5 Reset

The boundary-scan interface includes a state-machine controller (the TAP controller). In order to force the TAP controller into the correct state after power-up of the device, a reset pulse must be applied to the **nTRST** signal. If the boundary scan interface is to be used, then **nTRST** must be driven LOW, and then HIGH again. If the boundary scan interface is not to be used, then the **nTRST** input may be tied permanently LOW. Note that a clock on **TCK** is not necessary to reset the device.

The action of reset is as follows:

System mode is selected (i.e. the boundary scan chain cells do *not* intercept any of the signals passing between the external system and the core).

IDcode mode is selected. If the TAP controller is put into the Shift-DR state and **TCK** is pulsed, the contents of the ID register will be clocked out of **TDO**.

## 7.6 Pullup Resistors

The IEEE 1149.1 standard effectively requires that **TDI** and **TMS** should have internal pullup resistors. In order to minimise static current draw, these resistors are *not* fitted to ARM7DI. Accordingly, the 4 inputs to the test interface (the above 3 signals plus **TCK**) must all be driven to good logic levels to achieve normal circuit operation.

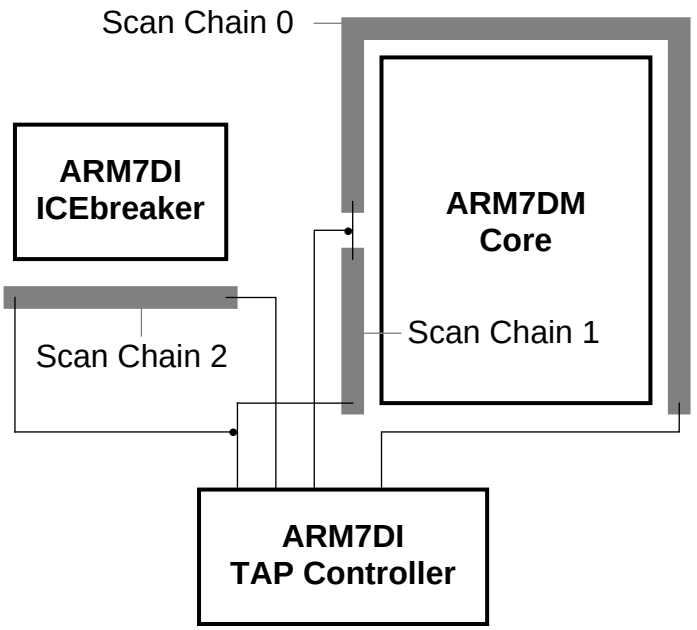



**Figure 43: ARM7DI Scan Chain Arrangement**

## 7.7 Instruction Register

The instruction register is 4 bits in length.

There is no parity bit. The fixed value loaded into the instruction register during the CAPTURE-IR controller state is: 0001.

## 7.8 Public Instructions

The following public instructions are supported:

| Instruction | Binary Code |
|---|---|
| EXTEST | 0000 |
| SCAN_N | 0010 |
| INTEST | 1100 |
| IDCODE | 1110 |
| BYPASS | 1111 |
| CLAMP | 0101 |
| HIGHZ | 0111 |
| CLAMPZ | 1001 |
| SAMPLE/PRELOAD | 0011 |

In the descriptions that follow, **TDI** and **TMS** are sampled on the rising edge of **TCK** and all output transitions on **TDO** occur as a result of the falling edge of **TCK**.

### 7.8.1 EXTEST (0000)

The selected scan chain is placed in test mode by the EXTEST instruction.

The EXTEST instruction connects the selected scan chain between **TDI** and **TDO**.

When the instruction register is loaded with the EXTEST instruction, all the scan cells are placed in their test mode of operation.

In the CAPTURE-DR state, inputs from the system logic and outputs from the output scan cells to the system are captured by the scan cells. In the SHIFT-DR state, the previously captured test data is shifted out of the scan chain via **TDO**, while new test data is shifted in via the **TDI** input. Note that this data is applied immediately to the system logic and system pins.

### 7.8.2 SCAN_N (0010)

This instruction connects the Scan Path Select Register between **TDI** and **TDO**. During the CAPTURE-DR state, the fixed value 1000 is loaded into the register. During the SHIFT-DR state, the ID number of the desired scan path is shifted into the scan path select register. In the UPDATE-DR state, the scan register of the selected scan chain is connected between **TDI** and **TDO**, and remains connected until a subsequent SCAN_N instruction is issued. On reset, scan chain 0 is selected by default. The scan path select register is 4 bits long in this implementation, although no finite length is specified.

### 7.8.3 INTEST (1100)

The selected scan chain is placed in test mode by the INTEST instruction.

The INTEST instruction connects the selected scan chain between **TDI** and **TDO**.

When the instruction register is loaded with the INTEST instruction, all the scan cells are placed in their test mode of operation.

In the CAPTURE-DR state, the value of the data applied from the core logic to the output scan cells, and the value of the data applied from the system logic to the input scan cells is captured.

In the SHIFT-DR state, the previously captured test data is shifted out of the scan chain via the **TDO** pin, while new test data is shifted in via the **TDI** pin.

Single-step operation is possible using the INTEST instruction.

### 7.8.4 IDCODE (1110)

The IDCODE instruction connects the device identification register (or ID register) between **TDI** and **TDO**. The ID register is a 32-bit register that allows the manufacturer, part number and version of a component to be determined through the TAP.

When the instruction register is loaded with the IDCODE instruction, all the scan cells are placed in their normal (system) mode of operation.

In the CAPTURE-DR state, the device identification code (specified at the end of this section) is captured by the ID register. In the SHIFT-DR state, the previously captured device identification code is shifted out of the ID register via the **TDO** pin, while data is shifted in via the **TDI** pin into the ID register. In the UPDATE-DR state, the ID register is unaffected.

### 7.8.5 BYPASS (1111)

The BYPASS instruction connects a 1 bit shift register (the BYPASS register) between **TDI** and **TDO**.

When the BYPASS instruction is loaded into the instruction register, all the scan cells are placed in their normal (system) mode of operation. This instruction has no effect on the system pins.

In the CAPTURE-DR state, a logic 0 is captured by the bypass register. In the SHIFT-DR state, test data is shifted into the bypass register via **TDI** and out via **TDO** after a delay of one **TCK** cycle. Note that the first bit shifted out will be a zero. The bypass register is not affected in the UPDATE-DR state. Note that all unused instruction codes default to the BYPASS instruction.

### 7.8.6 CLAMP (0101)

This instruction connects a 1 bit shift register (the BYPASS register) between **TDI** and **TDO**.

When the CLAMP instruction is loaded into the instruction register, the state of all the output signals is defined by the values previously loaded into the currently loaded scan chain. Note that this instruction should only be used when scan chain 0 is the currently selected scan chain.

In the CAPTURE-DR state, a logic 0 is captured by the bypass register. In the SHIFT-DR state, test data is shifted into the bypass register via **TDI** and out via **TDO** after a delay of one **TCK** cycle. Note that the first bit shifted out will be a zero. The bypass register is not affected in the UPDATE-DR state.

### 7.8.7 HIGHZ (0111)

This instruction connects a 1 bit shift register (the BYPASS register) between **TDI** and **TDO**.

When the HIGHZ instruction is loaded into the instruction register, the Address bus, **A[31:0]**, the data bus, **D[31:0**], plus **nRW**, **nBW**, **nOPC**, **LOCK** and **nTRANS** are all driven to the high impedance state. This is as if the signal **TBE** had been driven low.

In the CAPTURE-DR state, a logic 0 is captured by the bypass register. In the SHIFT-DR state, test data is shifted into the bypass register via **TDI** and out via **TDO** after a delay of one **TCK** cycle. Note that the first bit shifted out will be a zero. The bypass register is not affected in the UPDATE-DR state.

### 7.8.8 CLAMPZ (1001)

This instruction connects a 1 bit shift register (the BYPASS register) between **TDI** and **TDO**.

When the CLAMPZ instruction is loaded into the instruction register, all the 3-state outputs (as described above) are placed in their inactive state, but the data supplied to the outputs is derived from the scan cells. The purpose of this instruction is to ensure that, during production test, each output can be disabled when its data value is either a logic 0 or a logic 1.

In the CAPTURE-DR state, a logic 0 is captured by the bypass register. In the SHIFT-DR state, test data is shifted into the bypass register via **TDI** and out via **TDO** after a delay of one **TCK** cycle. Note that the first bit shifted out will be a zero. The bypass register is not affected in the UPDATE-DR state.

### 7.8.9 SAMPLE/PRELOAD (0011)

This instruction is included for production test only, and should never be used.

## 7.9 Test Data Registers

There are 6 test data registers which may be connected between **TDI** and **TDO**. They are: Bypass Register, ID Code Register, Scan Chain Select Register, Scan chain 0, 1 or 2. These are now described in detail.

### 7.9.1 Bypass Register

Purpose: This is a single bit register which can be selected as the path between **TDI** and **TDO** to allow the device to be bypassed during scan testing.

Length: 1 bit

Operating Mode: When the BYPASS instruction is the current instruction in the instruction register, serial data is transferred from **TDI** to **TDO** in the SHIFT-DR state with a delay of one **TCK** cycle.

There is no parallel output from the bypass register.

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

### 7.9.2 ARM7DI Device Identification (ID) Code Register

Purpose: This register is used to read the 32-bit device identification code. No programmable supplementary identification code is provided.

Length: 32 bits

The format of the ID register is as follows:

| 31 – 28 | 27 – 12 | 11 – 1 | 0 |
|---|---|---|---|
| **Version** | **Part Number** | **Manufacturer Identity** | **1** |

Please contact your supplier for the correct Device Identification Code.

Operating Mode: When the IDCODE instruction is current, the ID register is selected as the serial path between **TDI** and **TDO**.

There is no parallel output from the ID register.

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

### 7.9.3 Instruction Register

Purpose: This is a 4 bit register which is used to change the current TAP instruction.

Length: 4 bits

Operating mode: When in the SHIFT-IR state, the instruction register is selected as the serial path between **TDI** and **TDO**.

During the CAPTURE-IR state, the value 0001 binary is loaded into this register. This is shifted out during SHIFT-IR (lsb first), while a new instruction is shifted in (lsb first). During the UPDATE-IR state, the value in the instruction register becomes the current instruction. On reset, BYPASS becomes the current instruction.

### 7.9.4 Scan Chain Select Register

Purpose: This is a 4 bit register which is used to change the current active scan chain.

Length: 4 bits

Operating mode: After SCAN_N has been selected as the current instruction, when in the SHIFT-DR state, the Scan Chain Select Register is selected as the serial path between **TDI** and **TDO**.

During the CAPTURE-DR state, the value 1000 binary is loaded into this register. This is shifted out during SHIFT-DR (lsb first), while a new value is shifted in (lsb first). During the UPDATE-DR state, the value in the register selects a scan chain to become the currently active scan chain. All further instructions such as INTEST then apply to that scan chain.

The currently selected scan chain only changes when a SCAN_N instruction is executed, or a reset occurs. On reset, scan chain 0 is selected as the active scan chain.

### 7.9.5 Scan Chains 0,1 and 2

These allow serial access to the core logic, and to ICEBreaker for programming purposes. They are described in detail below.

#### 7.9.5.1 Scan Chain 0 and 1

Purpose: To allow access to the ARM7DM CPU core for test and debug.

Length: Scan chain 0: 97 bits, Scan Chain 1: 33 bits.

Each scan chain cell is fairly simple. Each consists of a serial register and a multiplexer. The scan cells perform two basic functions, *capture* and *shift*.

For input cells, the capture stage involves copying the value of the system input to the core into the serial register. During shift, this value is output serially. The value applied to the core from an input cell is either the system input or the contents of the serial register, and this is controlled by the multiplexer.

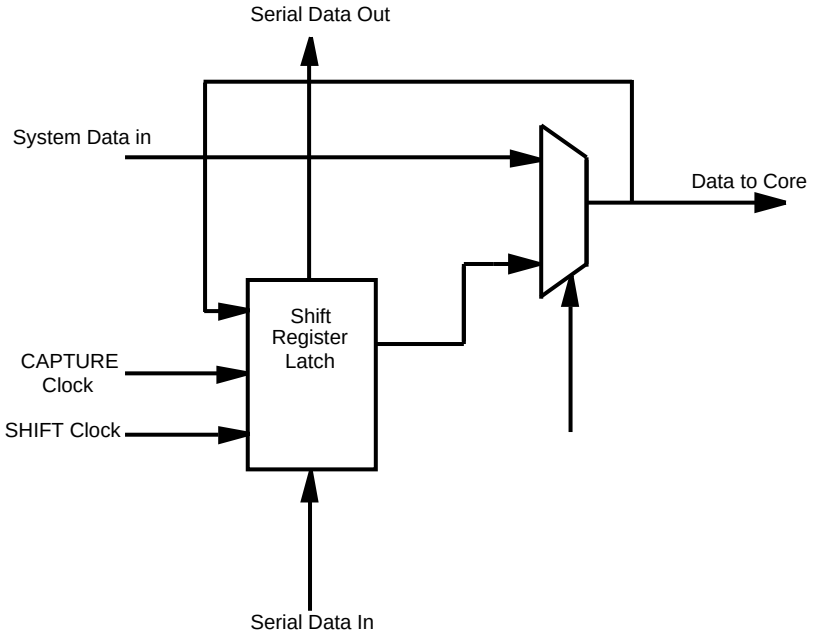



**Figure 44: Input Scan Cell**

For output cells, capture involves placing the value of a core's output into the serial register. During shift, this value is serially output as before. The value applied to the system from an output cell is either the core output, or the contents of the serial register.

All the control signals for the scan cells are generated internally by the TAP controller. The action of the TAP controller is determined by the current instruction, and the state of the TAP state machine. This is described below.

There are three basic modes of operation of the scan chains, INTEST, EXTEST and SYSTEM, and these are selected by the various TAP controller instructions. In SYSTEM mode, the scan cells are idle. System data is applied to inputs, and core outputs are applied to the system. In INTEST mode, the core is internally tested. The data serially scanned in is applied to the core, and the resulting outputs are captured in the output cells and scanned out. In EXTEST mode, data is scanned onto the core's outputs and applied to the external system. System input data is captured in the input cells and then shifted out.

Note that the scan cells are not fully JTAG compliant in that they do not have an *Update* stage. Therefore, while data is being moved around the scan chain, the contents of the scan cell is not isolated from the output. Thus the output from the scan cell to the core or to the external system could change on every scan clock.

This does not affect ARM7DI since its internal state does not change until it is clocked. The rest of the system needs to be aware though that every output could change asynchronously as data is moved around the scan chain. External logic must ensure that this does not harm the rest of the system.

***Scan chain 0***

Scan chain 0 is intended primarily for inter-device testing (EXTEST), and testing the core (INTEST). Scan chain 0 is selected via the SCAN_N TAP Controller instruction.

INTEST allows serial testing of the core. The TAP Controller must be placed in INTEST mode after scan chain 0 has been selected. During CAPTURE-DR, the current outputs from the core's logic are captured in the output cells. During SHIFT-DR, this captured data is shifted out while a new serial test pattern is scanned in, thus applying known stimuli to the inputs. During RUN-TEST/IDLE, the core is clocked. Normally, the TAP controller should only spend 1 cycle in RUN-TEST/IDLE (see later section for details of the core's clocks during test and debug). The whole operation may then be repeated.

EXTEST allows inter-device testing, useful for verifying the connections between devices on a circuit board. The TAP Controller must be placed in EXTEST mode after scan chain 0 has been selected. During CAPTURE-DR, the current inputs to the core's logic from the system are captured in the input cells. During SHIFT-DR, this captured data is shifted out while a new serial test pattern is scanned in, thus applying known values on the core's outputs. During UPDATE-DR, the value shifted into the data bus **D[31:0]** scan cells appears on the outputs. For all other outputs, the value appears as the data is shifted round. Note, during RUN-TEST/IDLE, the core is not clocked. The operation may then be repeated.

***Scan chain 1***

The primary use for scan chain 1 is for debugging, although it can be used for EXTEST on the data bus. Scan chain 1 is selected via the SCAN_N TAP Controller instruction. Debugging is similar to INTEST, and the procedure described above for scan chain 0 should be followed.

Note that this scan chain is 33 bits long - 32 bits for the data value, plus the scan cell on the **BREAKPT** core input. This 33rd bit serves 4 purposes. Firstly, under normal INTEST test conditions, this allows a known value to be scanned into the **BREAKPT** input. Secondly, during EXTEST test conditions, the value applied to the **BREAKPT** input from the system can be captured. Thirdly, while debugging, the value placed in the 33rd bit determines whether ARM7DI synchronises back to system speed before executing the instruction. See*Section 7.12.5: System Speed Access* for further details. Finally, after ARM7DI has entered debug state, the first time this bit is captured and scanned out, its value tells the debugger whether the core entered debug state due to a breakpoint (bit 33 LOW), or a watchpoint (bit 33 HIGH).

### 7.9.5.2 Scan Chain 2

Purpose: Scan chain 2 is provided to allow ICEBreaker's registers to be accessed. The order of the scan chain, from TDI to TDO is: read/write, register address bits 4 to 0, followed by data value bits 31 to 0.

Length: 38 bits.

To access this serial register, scan chain 2 must first be selected via the SCAN_N TAP controller instruction. The TAP controller must then be place in INTEST mode. No action is taken during CAPTURE-DR. During SHIFT-DR, a data value is shifted into the serial register. Bits 32 to 36 specify the address of the ICEBreaker register to be accessed. During UPDATE-DR, this register is either read or written depending on the value of bit 37 (0 = read). Refer to *Chapter 8.0: The ARM7DI ICEBreaker Module* on page 109 for further details.

## 7.10 ARM7DI Core Clocks

ARM7DI has two clocks, the memory clock, **MCLK**, and an internally **TCK** generated clock, **DCLK**. During normal operation, the core is clocked by **MCLK**, and internal logic holds **DCLK** LOW. When ARM7DI is in the debug state, the core is clocked by **DCLK** under control of the TAP state machine, and **MCLK** may free run. The selected clock is output on the signal **ECLK** for use by the external system. Note that when the CPU core is being debugged and is running from **DCLK**, **nWAIT** has no effect.

There are two cases in which the clocks switch, during debugging and during testing.

### 7.10.1 Clock Switch During Debug

When ARM7DI enters debug state, it must switch from **MCLK** to **DCLK**. This is handled automatically by logic in the ARM7DI. On entry to debug state, ARM7DI asserts **DBGACK** in the HIGH phase of **MCLK**. The switch between the two clocks occurs on the next falling edge of **MCLK**. This is shown in *Figure 45: Clock Switching on Entry to Debug State.*

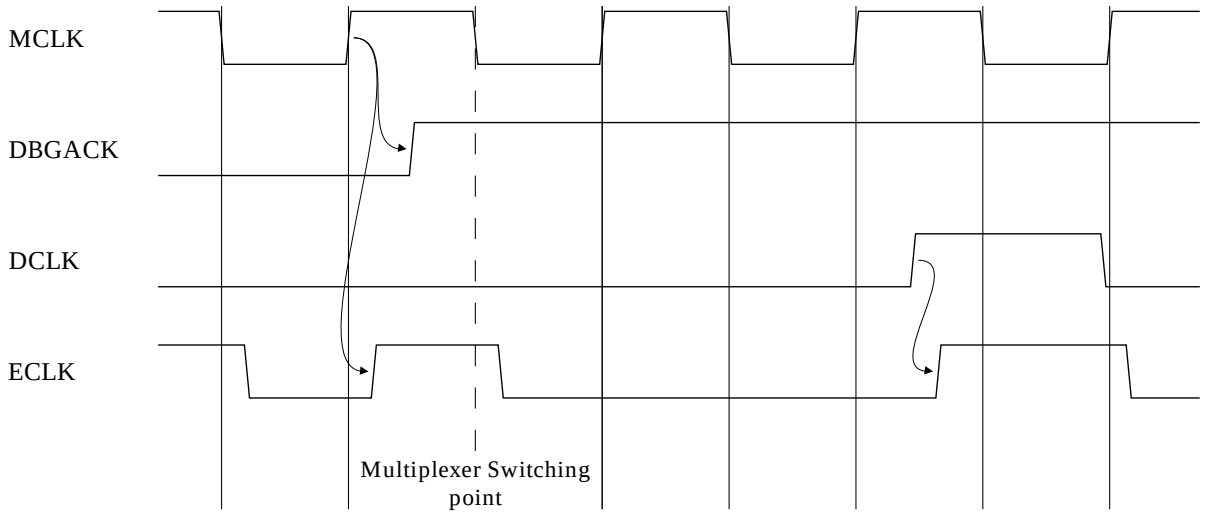



**Figure 45: Clock Switching on Entry to Debug State**

ARM7DI is forced to use **DCLK** as the primary clock until debugging is complete. On exit from debug, the core must be allowed to synchronise back to **MCLK**. This must be done in the following sequence. The final instruction of the debug sequence must be shifted into the data bus scan chain and clocked in by asserting **DCLK**. At this point, BYPASS must be clocked into the TAP instruction register. ARM7DI will now automatically resynchronise back to **MCLK** and start fetching instructions from memory at **MCLK** speed. Please refer also to *Section 7.11.3: Exit from Debug State.*

### 7.10.2 Clock Switch During Test

When under serial test conditions - ie when test patterns are being applied to the ARM7DM core through the JTAG interface - ARM7DI must be clocked using **DCLK**. Entry into test is less automatic than debug and some care must be taken.

On the way into test, **MCLK** must be held LOW. The TAP controller can now be used to serially test ARM7DI. If scan chain 0 and INTEST are selected, then **DCLK** is generated while the state machine is in the RUN-TEST/IDLE state. During EXTEST, **DCLK** is not generated.

On exit from test, BYPASS must be selected as the TAP controller instruction. When this is done, MCLK can be allowed to resume. After INTEST testing, care should be taken to ensure that the core is in a sensible state before switching back. The safest way to do this is to either select BYPASS and then cause a system reset, or to insert MOV PC, #0 into the instruction pipeline before switching back.

## 7.11 Determining the Core and System State

When ARM7DI is in debug state, the core and system's state may be examined. This is done by forcing load and store multiples into the instruction pipeline.

### 7.11.1 Determining the Core's State

Typically, the first instruction to be executed would be:

```
STM R0, {R0-R15}
```

This causes the contents of the registers to be made visible on the data bus. (Note that the use of R0 here is arbitrary, and for illustration only). These values can then be sampled and shifted out.

After determining the values in the current bank of registers, it may be desirable to access the banked registers. This can only be done by changing mode. Normally, a mode change may only occur if the core is already in a privileged mode. However, while in debug state, a mode change from any mode into any other mode may occur. Note that the debugger must restore the original mode before exiting debug state.

For example, assume that the debugger had been asked to return the state of the USER mode and FIQ mode registers, and debug state was entered in supervisor mode. The instruction sequence could be:

```
STM R0, {R0-R15}        ; Save current registers
MRS R0, CPSR
STR R0, R0              ; Save CPSR to determine current mode
BIC R0, 0x1F            ; Clear mode bits
ORR R0, 0x10            ; Select user mode
MSR CPSR, R0            ; Enter USER mode
STM R0, {R13,R14}       ; Save register not previously visible
ORR R0, 0x01            ; Select FIQ mode
MSR CPSR, R0            ; Enter FIQ mode
STM R0, {R8-R14}        ; Save banked FIQ registers
```

All these instructions are said to execute at *debug speed*. Debug speed is much slower than system speed since between each core clock, 33 scan clocks occur in order to shift in an instruction, or shift out data. Executing instructions more slowly than usual is fine for accessing the core's state since ARM7DI is fully static. However, this same method cannot be used for determining the state of the rest of the system.

### 7.11.2 Determining System State

In order to meet the dynamic timing requirements of the memory system, any attempt to access system state must occur synchronously to it. Thus, ARM7DI must be forced to synchronise back to system speed. This is controlled by the 33rd bit of scan chain 1.

Any instruction may be placed in scan chain 1 with bit 33 (the **BREAKPT** bit) LOW. This instruction will then be executed at debug speed. To execute an instruction at system speed, the instruction prior to it must be scanned into scan chain 1 with bit 33 set HIGH.

After the system speed instruction has been scanned into the data bus and clocked into the pipeline, the BYPASS instruction must be loaded into the TAP controller. This will cause ARM7DI to automatically synchronise back to **MCLK** (the system clock), execute the instruction at system speed, and then re-enter debug state and switch itself back to the internally generated **DCLK**. When the instruction has completed, **DBGACK** will be HIGH and the core will have switched back to **DCLK**. At this point, INTEST can be selected in the TAP controller, and debugging can resume.

In order to determine that a system speed instruction has completed, the debugger must look at both **DBGACK** and **nMREQ**. In order to access memory, ARM7DI drives **nMREQ** LOW after it has synchronised back to system speed. This transition is used by the memory controller to arbitrate whether ARM7DI can have the bus in the next cycle. If the bus is not available, then ARM7DI may have its clock stalled indefinitely. Therefore, the only way to tell that the memory access has completed, is to examine the state of both **nMREQ** and **DBGACK**. When both are HIGH, the access has completed. Usually, the debugger would be using ICEBreaker to control debugging, and by reading ICEBreaker's status register, the state of **nMREQ** and **DBGACK** can be determined. Refer to *Chapter 8.0: The ARM7DI ICEBreaker Module* on page 109 for more details.

By the use of system speed load multiples and debug speed store multiples, the state of the system's memory can be fed back to the debug host.

There are restrictions on which instructions may have the 33rd bit set. The only valid instructions on which to set this bit are loads, stores, load multiple and store multiple. See also *Section 7.11.3: Exit from Debug State*. When ARM7DI returns to debug state after a system speed access, bit 33 of scan chain 1 is set HIGH. This gives the debugger information about why the core entered debug state the first time this scan chain is read.

### 7.11.3 Exit from Debug State

Leaving debug state involves restoring ARM7DI's internal state, causing a branch to the next instruction to be executed, and synchronising back to **MCLK**. After restoring internal state, a branch instruction must be loaded into the pipeline. See *Section 7.12: The PC's Behaviour During Debug* for details on calculating the branch.

Bit 33 of scan chain 1 is used to force ARM7DI to resynchronise back to **MCLK**. The penultimate instruction of the debug sequence is scanned in with bit 33 set HIGH. The final instruction of the debug sequence is the branch, and this is scanned in with bit 33 LOW. The core is then clocked to load the branch into the pipeline. Now, the BYPASS instruction is selected in the TAP controller. This allows the scan chain to go back to system mode and allows resynchronisation to occur. ARM7DI then resumes normal operation, fetching instructions from memory.

The function of **DBGACK** is to tell the rest of the system when ARM7DI is in debug state. This can be used to inhibit peripherals such as watchdog timers which have real time characteristics. Also, **DBGACK** can be used to mask out memory accesses which are caused by the debugging process. For example, when ARM7DI enters debug state after a breakpoint, the instruction pipeline contains the breakpointed instruction plus two other instructions which have been prefetched. On entry to debug state, the pipeline is flushed. Therefore, on exit from debug state, the pipeline must be refilled to its previous state. Thus, because of the debugging process, more memory accesses occur than would normally be expected. Any system peripheral which may be sensitive to the number of memory accesses can be inhibited through the use of **DBGACK**.

For example, imagine a fictitious peripheral that simply counts the number of memory cycles. This device should return the same answer after a program has been run both with and without debugging. *Figure 46: Debug Exit Sequence* shows the behaviour of ARM7DI on exit from the debug state.

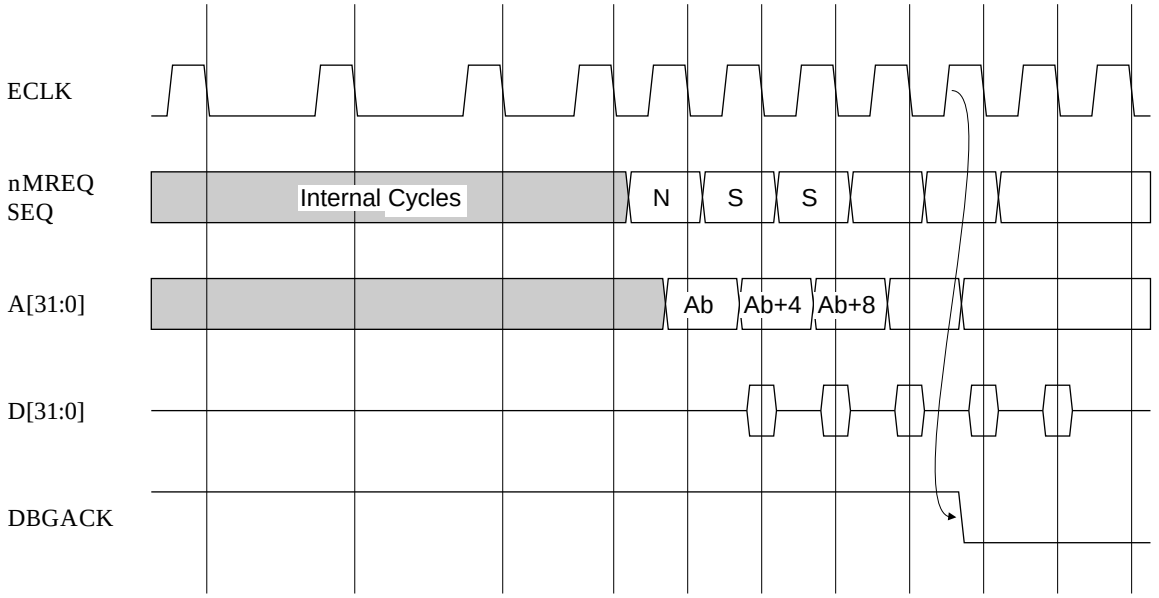



**Figure 46: Debug Exit Sequence**

It can be seen from *Figure 41: Debug State Entry* that the final memory access occurs in the cycle *after* **DBGACK** goes HIGH, and this is the point at which the cycle counter should be disabled. *Figure 46: Debug Exit Sequence* shows that the first memory access that the cycle counter has not seen before occurs in the cycle after **DBGACK** goes LOW, and so this is the point at which the counter should be re-enabled.

Note that when a system speed access from debug state occurs, ARM7DI temporarily drops out of debug state, and so **DBGACK** can go LOW. If there are peripherals which are sensitive to the number of memory accesses, then they must be led to believe that ARM7DI is still in debug state. By programming the ICEBreaker control register, the value on **DBGACK** can be forced to be HIGH. See *Chapter 8.0: The ARM7DI ICEBreaker Module* on page 109 for more details.

## 7.12 The PC's Behaviour During Debug

In order that ARM7DI may be forced to branch back to the place at which program flow was interrupted by debug, the debugger must keep track of what happens to the PC. There are five cases: breakpoint, watchpoint, watchpoint when another exception occurs, debug request and system speed access.

### 7.12.1 Breakpoint

Entry to the debug state from a breakpoint advances the PC by 4 addresses, or 16 bytes. Each instruction executed in debug state advances the PC by 1 address, or 4 bytes. The normal way to exit from debug state after a breakpoint is to remove the breakpoint, and branch back to the previously breakpointed address.

For example, if ARM7DI entered debug state from a breakpoint set on a given address and 2 debug speed instructions were executed, then a branch of -7 addresses must occur (4 for debug entry, +2 for the instructions, +1 for the final branch). The following sequence shows the data scanned into scan chain 1. This is msb first, and so the first digit is the value placed in the **BREAKPT** bit, followed by the instruction data.

```
0 E0802000        ; ADD R2, R0, R0
1 E1826001        ; ORR R6, R2, R1
0 EAFFFFF9        ; B -7 (2's complement)
```

Note that once in debug state, a minimum of two instructions must be executed before the branch, although these may both be NOPs (MOV R0, R0). For small branches, the final branch could be replaced with a subtract with the PC as the destination (SUB PC, PC, #28 in the above example).

### 7.12.2 Watchpoints

Returning to program execution after entering debug state from a watchpoint is done in the same way as the procedure described above. Debug entry adds 4 addresses to the PC, and every instruction adds 1 address. The difference is that since the instruction that caused the watchpoint has executed, the program returns to the next instruction.

### 7.12.3 Watchpoint with another Exception

If a watchpointed access simultaneously causes a data abort, then ARM7DI will enter debug state in abort mode. Entry into debug is held off until the core has changed into abort mode, and fetched the instruction from the abort vector.

A similar sequence is followed when an interrupt - or any other exception - occurs during a watchpointed memory access. ARM7DI will enter debug state in the exception's mode, and so the debugger must check to see whether this happened. The debugger can deduce whether an exception occurred by looking at the current and previous mode (in the CPSR and SPSR), and the value of the PC. If an exception did take place, then the user should be given the choice of whether to service the exception before debugging.

Exiting debug state if an exception occurred is slightly different from the other cases. Here, entry to debug state causes the PC to be incremented by 3 addresses rather than 4, and this must be taken into account in the return branch calculation. For example, suppose that an abort occurred on a watchpointed access and 10 instructions had been executed to determine this. The following sequence could be used to return to program execution.

```
0 E1A00000        ; MOV R0, R0
1 E1A00000        ; MOV R0, R0
0 EAFFFFF0        ; B -16
```

This will force a branch back to the abort vector, causing the instruction at that location to be refetched and executed. Note that after the abort service routine, the instruction which caused the abort and watchpoint will be reexecuted. This will cause the watchpoint to be generated and hence ARM7DI will enter debug state again.

### 7.12.4 Debug Request

Entry into debug state via a debug request is similar to a breakpoint. However, unlike a breakpoint, the last instruction will have completed execution and so must not be refetched on exit from debug state. Therefore, it can be thought that entry to debug state adds 3 addresses to the PC, and every instruction executed in debug state adds 1.

For example, suppose that the user has invoked a debug request, and decides to return to program execution straight away. The following sequence could be used:

```
0 E1A00000        ; MOV R0, R0
1 E1A00000        ; MOV R0, R0
0 EAFFFFFA        ; B -6
```

This restores the PC, and restarts the program from the next instruction.

### 7.12.5 System Speed Access

If a system speed access is performed during debug state, the value of the PC is increased by 3 addresses. Since system speed instructions access the memory system, it is possible for aborts to take place. If an abort occurs during a system speed memory access, ARM7DI enters abort mode before returning to debug state.

This is similar to an aborted watchpoint except that the problem is much harder to fix, because the abort was not caused by an instruction in the main program, and the PC does not point to the instruction which caused the abort. An abort handler usually looks at the PC to determine the instruction which caused the abort, and hence the abort address. In this case, the value of the PC is invalid, but the debugger should know what location was being accessed. Thus the debugger can be written to help the abort handler fix the memory system.

### 7.12.6 Summary of Return Address Calculations

The calculation of the branch return address can be summarised as follows:

a) For normal breakpoint and watchpoint, the branch is:

```
- (4 + N + 3S)
```

b) For entry through debug request (**DBGRQ**), or watchpoint with exception, the branch is:

```
- (3 + N + 3S)
```

where N is the number of debug speed instructions executed (including the final branch), and S is the number of system speed instructions executed.

## 7.13 Priorities / Exceptions

Because the normal program flow is broken when a breakpoint or a debug request occurs, debug can be thought of as being another type of exception. Some of the interaction with other exceptions has been described above. This section summarises the priorities.

### 7.13.1 Breakpoint with Prefetch Abort

When a breakpointed instruction fetch causes a prefetch abort, the abort is taken and the breakpoint is disregarded. Normally, prefetch aborts occur when, for example, an access is made to a virtual address which does not physically exist, and the returned data is therefore invalid. In such a case the operating system's normal action will be to swap in the page of memory and return to the previously invalid address. This time, when the instruction is fetched, and providing the breakpoint is activated (it may be data dependent), ARM7DI will enter debug state.

Thus the prefetch abort takes higher priority than the breakpoint.

### 7.13.2 Interrupts

When ARM7DI enters debug state, interrupts are automatically disabled. If interrupts are disabled during debug, ARM7DI will never be forced into an interrupt mode.

If an interrupt was pending during the instruction prior to entering debug state, then ARM7DI will enter debug state in the mode of the interrupt. Thus, on entry to debug state, the debugger cannot assume that ARM7DI will be in the expected mode of the user's program. It must check the PC, the CPSR and the SPSR to fully determine the reason for the exception.

Thus, debug takes higher priority than the interrupt, although ARM7DI *remembers* that an interrupt has occurred.

### 7.13.3 Data Aborts

As described above, when a data abort occurs on a watchpointed access, ARM7DI enters debug state in abort mode. Thus the watchpoint has higher priority than the abort, although, as in the case of interrupt, ARM7DI remembers that the abort happened.

## 7.14 Scan Interface Signals

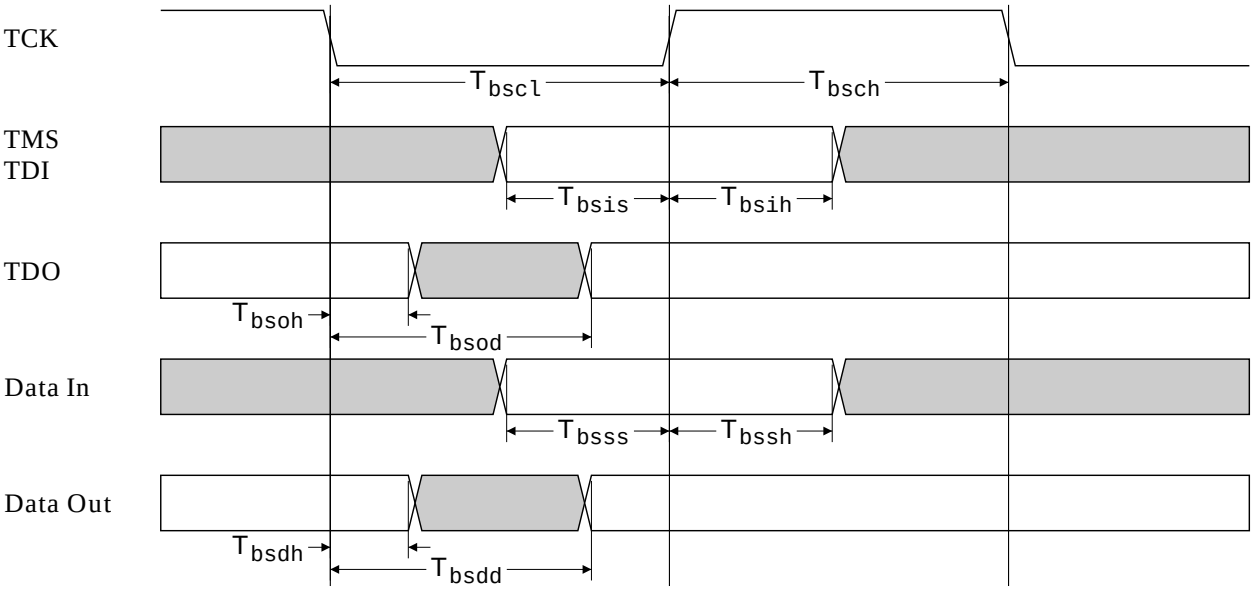



**Figure 47: Scan General Timing**

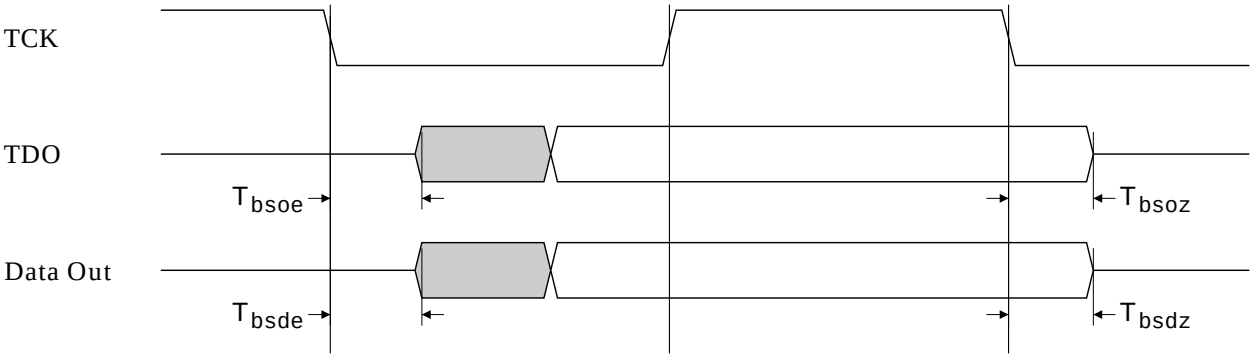



**Figure 48: Scan 3-state Timing**

| Symbol | Parameter | Min | Typ | Max | Units | Notes |
|---|---|---|---|---|---|---|
| Tbscl | **TCK** low period | TBD | | | ns | |
| Tbsch | **TCK** high period | TBD | | | ns | |
| Tbsis | **TDI**,**TMS** setup to [TCr] | TBD | | | ns | |
| Tbsih | **TDI**,**TMS** hold from [TCr] | TBD | | | ns | |

**Table 8: ARM7DI Scan Interface Timing**

| Symbol | Parameter | Min | Typ | Max | Units | Notes |
|---|---|---|---|---|---|---|
| Tbsoh | **TDO** hold time | 5 | | | ns | 1 |
| Tbsod | TCf to **TDO** valid | | | 40 | ns | 1 |
| Tbsss | I/O signal setup to [TCr] | 5 | | | ns | 4 |
| Tbssh | I/O signal hold from [TCr] | 20 | | | ns | 4 |
| Tbsdh | data output hold time | 5 | | | ns | 5 |
| Tbsdd | TCf to data output valid | | | 40 | ns | |
| Tbsoe | **TDO** enable time | 5 | | | ns | 1,2 |
| Tbsoz | **TDO** disable time | | | 40 | ns | 1,3 |
| Tbsde | data output enable time | 5 | | | ns | 5,6 |
| Tbsdz | data output disable time | | | 40 | ns | 5,7 |

**Table 8: ARM7DI Scan Interface Timing**

*Please note that the above figures are provisional.*

Notes:

1. Assumes a 25pF load on **TDO**. Output timing derates at 0.072ns/pF of extra load applied.
2. **TDO** enable time applies when the TAP controller enters the Shift-DR or Shift-IR states.
3. **TDO** disable time applies when the TAP controller leaves the Shift-DR or Shift-IR states.
4. For correct data latching, the I/O signals (from the core and the pads) must be setup and held with respect to the rising edge of **TCK** in the CAPTURE-DR state of the INTEST and EXTEST instructions.
5. Assumes that the data outputs are loaded with the AC test loads (see AC parameter specification).
6. Data output enable time applies when the scan logic is used to enable the output drivers.
7. Data output disable time applies when the scan logic is used to disable the output drivers.

| No | Signal | Type | No | Signal | Type | No | Signal | Type | No | Signal | Type |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | D[0] | I/O | 26 | D[25] | I/O | 51 | nOPC | O | 76 | A[21] | O |
| 2 | D[1] | I/O | 27 | D[26] | I/O | 52 | IFEN | I | 77 | A[20] | O |
| 3 | D[2] | I/O | 28 | D[27] | I/O | 53 | nCPI | O | 78 | A[19] | O |
| 4 | D[3] | I/O | 29 | D[28] | I/O | 54 | nMREQ | O | 79 | A[18] | O |
| 5 | D[4] | I/O | 30 | D[29] | I/O | 55 | SEQ | O | 80 | A[17] | O |
| 6 | D[5] | I/O | 31 | D[30] | I/O | 56 | nTRANS | O | 81 | A[16] | O |
| 7 | D[6] | I/O | 32 | D[31] | I/O | 57 | CPB | I | 82 | A[15] | O |
| 8 | D[7] | I/O | 33 | BREAKPT | I | 58 | nM[4] | O | 83 | A[14] | O |
| 9 | D[8] | I/O | 34 | NENIN | I | 59 | nM[3] | O | 84 | A[13] | O |
| 10 | D[9] | I/O | 35 | NENOUT | O | 60 | nM[2] | O | 85 | A[12] | O |
| 11 | D[10] | I/O | 36 | LOCK | O | 61 | nM[1] | O | 86 | A[11] | O |
| 12 | D[11] | I/O | 37 | BIGEND | I | 62 | nM[0] | O | 87 | A[10] | O |
| 13 | D[12] | I/O | 38 | DBE | I | 63 | nEXEC | O | 88 | A[9] | O |
| 14 | D[13] | I/O | 39 | nBW | O | 64 | ALE | I | 89 | A[8] | O |
| 15 | D[14] | I/O | 40 | DBGACK | O | 65 | ABE | I | 90 | A[7] | O |
| 16 | D[15] | I/O | 41 | nFIQ | I | 66 | A[31] | O | 91 | A[6] | O |
| 17 | D[16] | I/O | 42 | nIRQ | I | 67 | A[30] | O | 92 | A[5] | O |
| 18 | D[17] | I/O | 43 | nRESET | I | 68 | A[29] | O | 93 | A[4] | O |
| 19 | D[18] | I/O | 44 | DATA32 | I | 69 | A[28] | O | 94 | A[3] | O |
| 20 | D[19] | I/O | 45 | ISYNC | I | 70 | A[27] | O | 95 | A[2] | O |
| 21 | D[20] | I/O | 46 | DBGRQ | I | 71 | A[26] | O | 96 | A[1] | O |
| 22 | D[21] | I/O | 47 | nRW | O | 72 | A[25] | O | 97 | A[0] | O |
| 23 | D[22] | I/O | 48 | ABORT | I | 73 | A[24] | O | | | |
| 24 | D[23] | I/O | 49 | CPA | I | 74 | A[23] | O | | | |
| 25 | D[24] | I/O | 50 | PROG32 | I | 75 | A[22] | O | | | |

**Table 9: Scan Signals and Pins**

Key: I - Input
O - Output
I/O - Input / Output

# 8.0 The ARM7DI ICEBreaker Module

The ARM7DI-ICEbreaker module - hereafter referred to simply as *ICEbreaker* - provides integrated on-chip debug support for the ARM7D CPU core. ICEbreaker consists of two real time watchpoint registers, together with a control and status register. One or both of the watchpoint registers can be programmed to halt execution of instructions by the ARM7D CPU core via its **BREAKPT** signal. Execution is halted when a match occurs between the values programmed into ICEbreaker and the values currently appearing on the address bus, data bus and various control signals. Any bit can be masked so that its value does not affect the comparison.

*Figure 49: ARM7DI Block Diagram* shows the relationship between ARM7D, ICEbreaker and the TAP controller. Only those ARM7DI signals that are pertinent to ICEbreaker are shown.

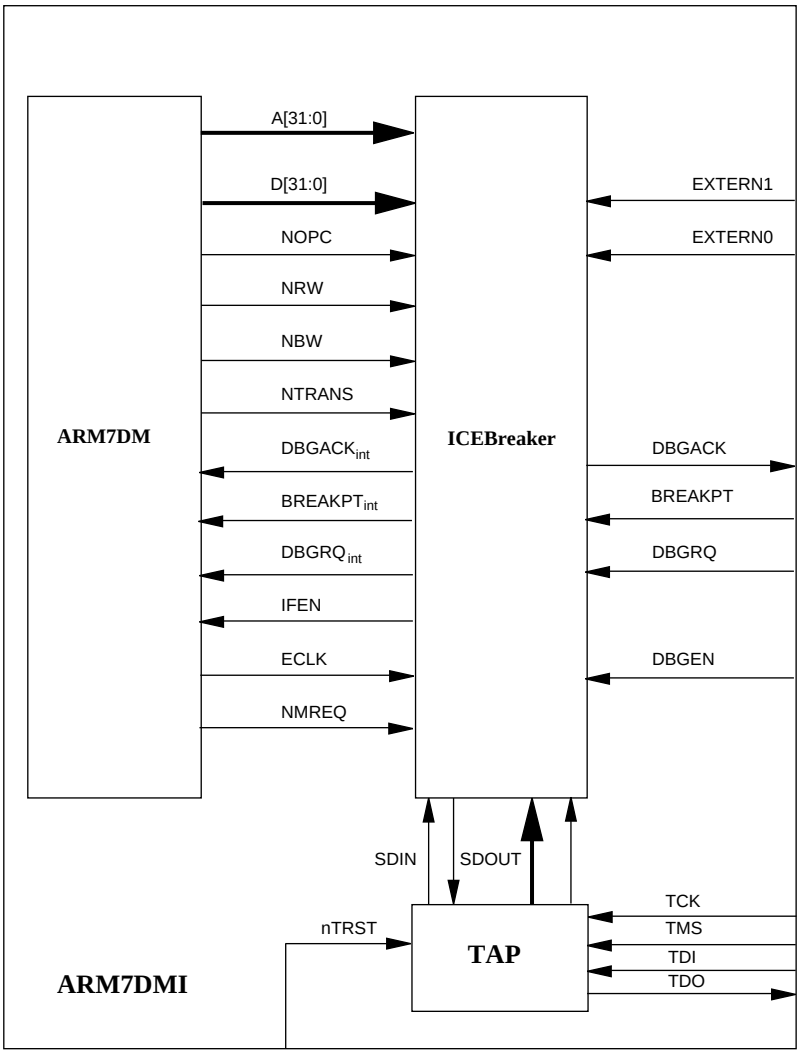



**Figure 49: ARM7DI Block Diagram**

Either watchpoint register can be configured as a watchpoint (data access) or a breakpoint (instruction fetch) by suitable programming of the value and mask registers. A particular feature of ICEbreaker is that both watchpoints and breakpoints can be data-dependent.

Independent Debug Control and Debug Status registers are used to provide overall control of ICEbreaker's operation. ICEbreaker is programmed in a serial fashion using the ARM7DI TAP controller.

## 8.1 The Watchpoint Registers

ICEbreaker consists of two watchpoint registers, known as *Watchpoint 0* and *Watchpoint 1*. Each watchpoint register consists of 6 independently programmable registers. These are:

- 32-bit Address Value register
- 32-bit Address Mask register
- 32-bit Data Value register
- 32-bit Data Mask register
- 8-bit Control Value register
- 7-bit Control Mask register

In addition, there are two global registers which provide overall control and status:

- 3-bit Debug Control register
- 4-bit Debug Status register

Each register is programmed by scanning appropriate data into the ICEbreaker scan chain (scan chain 2). The scan chain register consists of a 38-bit shift register comprising a 32-bit data field, a 5-bit address field and a read / write bit - see *Figure 50: ICEBreaker Block Diagram*.

Programming a register is achieved by scanning the data to be written into the 32-bit data field, the address of the register into the 5-bit address field and a '1' into the read / write bit.

Reading a register is achieved by scanning the address of the register into the 5-bit address field and a '0' into the read / write bit. The 32-bit data field is ignored in the case of reads.

Reading or writing of the register actually occurs when the TAP controller enters the UPDATE-DR state (see *Section 7.4: Scan Chains and JTAG Interface* on page 89).

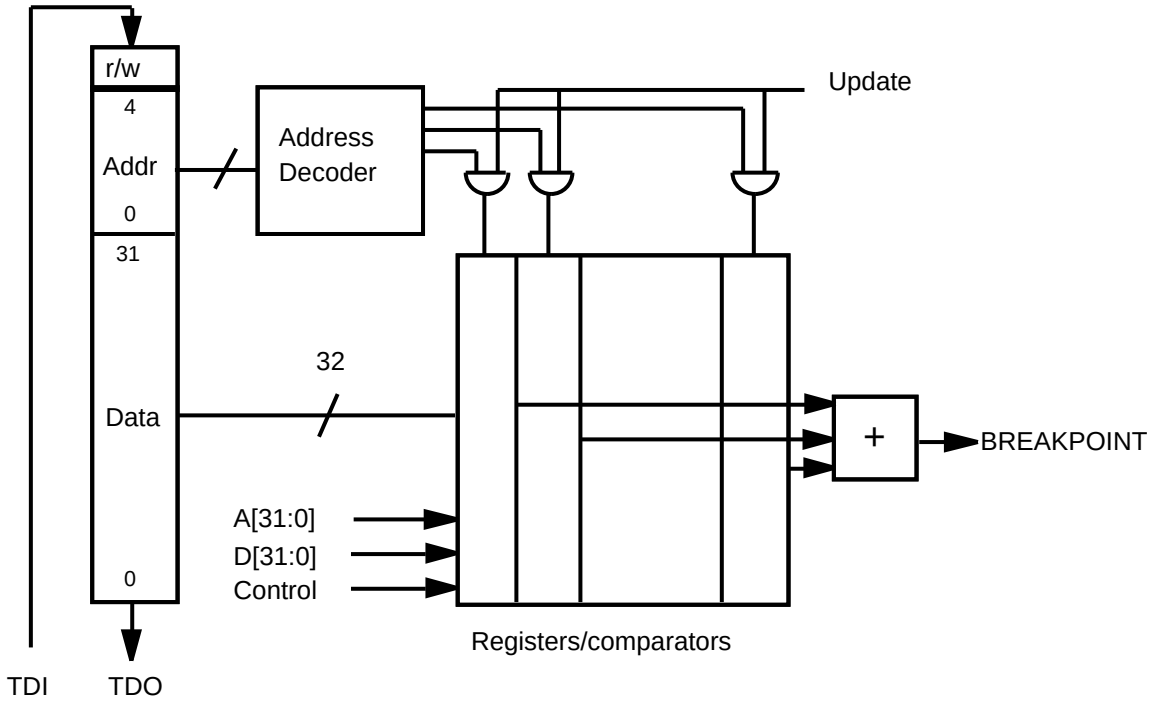



**Figure 50: ICEBreaker Block Diagram**

The registers are mapped as shown in *Table 10: Function and Mapping of ICEBreaker Registers*.

| Address | Width | Function |
|---|---|---|
| 00000 | 3 | Debug Control Register |
| 00001 | 4 | Debug Status Register |
| 01000 | 32 | Watchpoint 0 Address Value Register |
| 01001 | 32 | Watchpoint 0 Address Mask Register |
| 01010 | 32 | Watchpoint 0 Data Value Register |
| 01011 | 32 | Watchpoint 0 Data Mask Register |
| 01100 | 8 | Watchpoint 0 Control Value Register |
| 01101 | 7 | Watchpoint 0 Control Mask Register |
| 10000 | 32 | Watchpoint 1 Address Value Register |
| 10001 | 32 | Watchpoint 1 Address Mask Register |
| 10010 | 32 | Watchpoint 1 Data Value Register |
| 10011 | 32 | Watchpoint 1 Data Mask Register |
| 10100 | 8 | Watchpoint 1 Control Value Register |
| 10101 | 7 | Watchpoint 1 Control Mask Register |

**Table 10: Function and Mapping of ICEBreaker Registers**

The location of bits in the Control Value and Mask registers is shown in *Figure 51: ICEBreaker Watchpoint Control Value and Mask Registers*.

The control value and mask registers are mapped identically in the lower 7 bits. Bit 7 of the control value register is the **ENABLE** bit, which cannot be masked.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| ENABLE | RANGE | CHAIN | EXTERN | nTRANS | nOPC | nBW | nRW |

**Figure 51: ICEBreaker Watchpoint Control Value and Mask Registers**

**nRW :** This compares against the not read / write signal from the ARM7D core in order to detect the direction of bus activity. **nRW**=0 for a read cycle and **nRW**=1 for a write cycle.

**nBW** : This compares against the not byte / word signal from the ARM7D core in order to detect the size of bus activity. **nBW**=0 for a byte access and **nBW**=1 for a word access.

**nOPC** : This is used to detect whether the current cycle is an instruction fetch (**nOPC**=0) or a data access (**nOPC**=1).

**nTRANS** : This compares against the not translate signal from the ARM7DI core in order to distinguish between User mode (**nTRANS**=0) and non-User mode (**nTRANS**=1) accesses.

**EXTERN** : This is an external input to ICEbreaker which allows the watchpoint to be dependent upon some external condition. The **EXTERN** input for Watchpoint 0 is labelled **EXTERN0** and the **EXTERN** input for Watchpoint 1 is labelled **EXTERN1**.

**CHAIN** : This can be connected to the chain output of another watchpoint register in order to implement, for example, debugger requests of the form 'breakpoint on address YYY only when in process XXX'. In the ARM7DI-ICEbreaker, the **CHAINOUT** output of watchpoint register 1 is connected to the **CHAIN** input of watchpoint register 0. The **CHAINOUT** output is derived from a latch; the address / control field comparator drives the write enable for the latch and the input to the latch is the value of the data field comparator. The **CHAINOUT** latch is cleared when the Control Value register is written or when **NTRST** is LOW.

**RANGE** : This can be connected to the range output of another watchpoint register. In the ARM7DI-ICEbreaker, the **RANGEOUT** output of watchpoint register 1 is connected to the **RANGE** input of watchpoint register 0. This allows the two watchpoints to be coupled for detecting conditions that occur simultaneously, eg. for range-checking.

**ENABLE** : If a watchpoint match occurs, the **BREAKPT** signal will only be asserted when the **ENABLE** bit is set. This bit only exists in the value register: it cannot be masked.

For each of the bits 6:0 in the value register, there is a corresponding bit in the mask register. This allows the dependency on particular signals to be removed. For example, if a watchpoint is required on a particular memory location, but the data value is irrelevant, then the mask register can be programmed to make the entire data bus field 'don't care'. Note that the mask is an XNOR mask rather than a conventional AND mask: by setting the mask bit to a '1', the comparator for that bit position will always match, irrespective of the value register or the input value.

Setting the mask bit to '0' means that the comparator will only match if the input value matches the value programmed into the value register.

### 8.1.1 Programming the Watchpoint Registers

An ICEbreaker watchpoint register can be configured as a breakpoint (for instruction fetches) or as a watchpoint (for data accesses). Breakpoints can be further classified as hardware breakpoints or software breakpoints. Hardware breakpoints typically monitor the address value and can be set in any code, even in code that is in ROM and self-modifying code. Conversely, software breakpoints monitor a particular bit

pattern being fetched from any address; one ICEbreaker watchpoint register can thus be used to support any number of software breakpoints. Software breakpoints can normally only be set in RAM because an instruction has to be replaced by the special bit pattern chosen to cause a software breakpoint.

### 8.1.2 Programming Hardware Breakpoints

The watchpoint register can be used to cause hardware breakpoints (i.e. on instruction fetches) by programming it in the following way:

- Program the address value register with the address of the instruction to be breakpointed
- Program the address mask register with '11' in bits [1:0] (since A[1:0] are don't care values on instruction fetches) and with ’0’ in all other bits.
- Program the data value register only if a data-dependent breakpoint is required; i.e. only if the actual instruction code fetched must be matched as well as the address.
- If the data value is a don't care, program the data mask register to all '1's, otherwise program it to all '0's.
- Program the control value register with **nOPC**=0, all other bits are don't care
- Program the control mask register with **nOPC**=0, all other bits to '1'
- If the distinction between user and non-user mode instruction fetches is to be made, then the **nTRANS** value and mask register bits must also be programmed accordingly.
- The **EXTERN**, **RANGE** and **CHAIN** bits may also be programmed if desired.

### 8.1.3 Programming Software Breakpoints

The watchpoint register can be used to cause software breakpoints (i.e. on instruction fetches of a particular bit pattern) by programming it in the following way:

- The address value register need not be programmed.
- Program the address mask register with all '1's so that the address is a don't care.
- Program the data value register with the particular bit pattern that has been chosen to represent a software breakpoint.
- Program the data mask register to all '0's
- Program the control value register with **nOPC**=0, all other bits are don't care
- Program the control mask register with **nOPC**=0, all other bits to '1'
- If the distinction between user and non-user mode instruction fetches is to be made, then the **nTRANS** value and mask register bits must also be programmed accordingly.
- The **EXTERN**, **RANGE** and **CHAIN** bits may also be programmed if desired.

Then to set a software breakpoint, the instruction at the desired address should be read and stored away. Then the special bit pattern that represents a software breakpoint should be written at that address. This process may be repeated as many times as necessary. To clear a software breakpoint, the previously stored away instruction should be restored.

### 8.1.4 Programming Watchpoints

The watchpoint register can be used to cause watchpoints (i.e. on data accesses) by programming it in the following way:

- Program the address value register with the address of the data access to be watchpointed.
- Program the address mask register with all 0's
- Program the data value register only if a data-dependent watchpoint is required; i.e. only if the actual data value that is read or written must be matched as well as the address.
- If the data value is a don't care, program the data mask register to all '1's, otherwise program it to all '0's.
- Program the control value register with **nOPC**=1, **nRW**=0 for a read or **nRW**=1 for a write, **nBW**=0 for a byte access or **nBW**=1 for a word access
- Program the control mask register with **nOPC**=0, **nRW**=0, **nBW**=0, all other bits to '1'. Note that **nRW** or **nBW** may be set to '1' if both reads and writes or both byte and word accesses respectively are to be watchpointed.
- If the distinction between user and non-user mode data accesses is to be made, then the **nTRANS** value and mask register bits must also be programmed accordingly.
- The **EXTERN**, **RANGE** and **CHAIN** bits may also be programmed if desired.

Note that the above are just examples of how to program the watchpoint register to generate breakpoints and watchpoints; many other ways of programming the registers are possible. For instance, by setting one or more of the address mask bits, simple range breakpoints can be provided.

## 8.2 The Debug Control Register

The Debug Control Register is 3 bits wide. If the register is accessed for a write (with the read / write bit HIGH), then the control bits are written. If the register is accessed for a read (with the read / write bit LOW), then the control bits are read. The function and mapping of the Debug Control Register is shown below:

| Address | Width | Function |
|---|---|---|
| 00000 | 3 | Debug Control |

**Table 11: Debug Control Register - Function and Mapping**

The function of each bit in this register is as follows:

Bit 2: **INTDIS**
Bit 1: **DBGRQ**
Bit 0: **DBGACK**

Bits 1 and 0 allow the values on **DBGRQ** and **DBGACK** to be forced. In the case of **DBGRQ**, the value held in bit 1 of the Debug Control register is logically OR'ed with the external **DBGRQ** before being applied to the core. In the case of **DBGACK**, the value of **DBGACK** from the ARM7D core is OR'ed with the value held in bit 0 to generate the external value of **DBGACK** seen at the periphery of ARM7DI. This allows the debug system to signal to the rest of the system that the ARM7D is still being debugged even when system-speed accesses are being performed (in which case the internal **DBGACK** signal from the ARM7D core will be LOW).

If Bit 2 (**INTDIS**) is asserted, then the interrupt enable signal (**IFEN**) of the ARM7D core is forced LOW. Thus all interrupts (IRQ and FIQ) are disabled during debugging (**DBGACK**=1) or if the **INTDIS** bit is asserted. The **IFEN** signal is driven according to the following table:

| DBGACK (from core) | INTDIS | IFEN |
|---|---|---|
| 0 | 0 | 1 |
| 1 | x | 0 |
| x | 1 | 0 |

**Table 12: IFEN signal control**

## 8.3 Debug Status Register

The Debug Status Register is 4 bits wide. If the register is accessed for a write (with the read / write bit high), then the status bits are written. If the register is accessed for a read (with the read / write bit LOW), then the status bits are read. The function and mapping of the Debug Status Register is shown below:

| Address | Width | Function |
|---|---|---|
| 00001 | 4 | Debug Status |

**Table 13: Debug Status Register - Function and Mapping**

The function of each bit in this register is as follows:

Bit 3 : **nMREQ**
Bit 2 : **IFEN**
Bit 1 : **DBGRQ**
Bit 0 : **DBGACK**

Bits 1 and 0 allow the values on the synchronised versions of **DBGRQ** and **DBGACK** to be read.

Bit 2 allows the state of the interrupt enable pin (**IFEN**) of the ARM7D core to be read. Since the capture clock for the scan chain may be asynchronous to the processor clock, the **DBGACK** output from the core is synchronised before being used to generate the **IFEN** status bit.

Bit 3 allows the state of the **nMREQ** signal from the core (synchronised to **TCK**) to be read. This allows the debugger to determine that a memory access from the debug state has completed.

The structure of **DBGRQ**, **DBGACK** and **INTDIS** is shown below in *Figure 52: Structure of nMREQ, DBGACK, DBGRQ and INTDIS bits in ICEBreaker*.

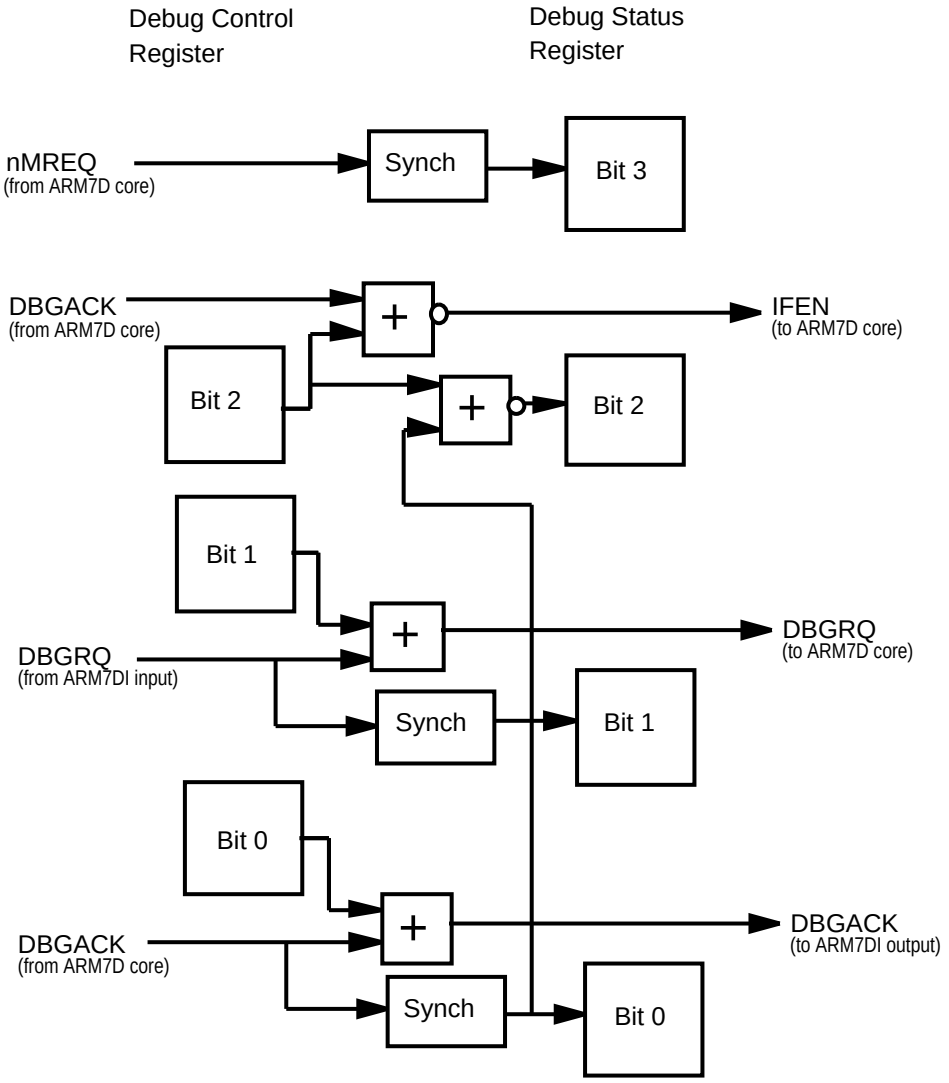



**Figure 52: Structure of nMREQ, DBGACK, DBGRQ and INTDIS bits in ICEBreaker**

## 8.4 Coupling Breakpoints and Watchpoints

Watchpoint registers 1 and 0 can be coupled together via the **CHAIN** and **RANGE** inputs. The use of **CHAIN** enables watchpoint 0 to be triggered only if watchpoint 1 has previously matched. The use of **RANGE** enables simple range checking to be performed by combining the outputs of both watchpoints.

| | | |
|---|---|---|
| Let | $A_v$[31:0] | be the value in the Address Value Register |
| | $A_m$[31:0] | be the value in the Address Mask Register |
| | A[31:0] | be the Address Bus from the ARM7D |
| | $D_v$[31:0] | be the value in the Data Value Register |
| | $D_m$[31:0] | be the value in the Data Mask Register |
| | D[31:0] | be the Data Bus from the ARM7D |
| | $C_v$[7:0] | be the value in the Control Value Register |
| | $C_m$[6:0] | be the value in the Control Mask Register |
| | C[8:0] | be the combined Control Bus from the ARM7D, other watchpoint registers and the **EXTERN** signal. |

Then the **CHAINOUT** signal is derived as follows:

WHEN (({$A_v$[31:0],$C_v$[3:0]} XNOR {A[31:0],C[3:0]}) OR {$A_m$[31:0],$C_m$[3:0]} == 0xFFFFFFFFF)

CHAINOUT = ((({$D_v$[31:0],$C_v$[6:4]} XNOR {D[31:0],C[6:4]}) OR {$D_m$[31:0],$C_m$[6:4]}) == 0x7FFFFFFFF)

The **CHAINOUT** output of watchpoint register 1 provides the **CHAIN** input to watchpoint register 0. This allows for quite complicated configurations of breakpoints and watchpoints.

Take for example the request by a debugger to breakpoint on the instruction at location YYY when running process XXX in a multiprocess system.

If the current process ID is stored in memory, then the above function can be implemented with a watchpoint and breakpoint chained together. The watchpoint address is set to a known memory location containing the current process ID, the watchpoint data is set to the required process ID and the **ENABLE** bit is set to “off”.

The address comparator output of the watchpoint is used to drive the write enable for the **CHAINOUT** latch, the input to the latch being the output of the data comparator from the same watchpoint. The output of the latch drives the **CHAIN** input of the breakpoint comparator. The address YYY is stored in the breakpoint register and when the **CHAIN** input is asserted, and the breakpoint address matches, the breakpoint triggers correctly.

The **RANGEOUT** signal is derived as follows:

RANGEOUT = ((({$A_v$[31:0],$C_v$[3:0]} XNOR {A[31:0],C[3:0]}) OR {$A_m$[31:0],$C_m$[3:0]}) == 0xFFFFFFFFF) AND ((({$D_v$[31:0],$C_v$[6:4]} XNOR {D[31:0],C[6:4]}) OR {$D_m$[31:0],$C_m$[6:4]}) == 0x7FFFFFFFF)

The **RANGEOUT** output of watchpoint register 1 provides the **RANGE** input to watchpoint register 0. This allows two breakpoints to be coupled together to form *range breakpoints*. Note that selectable ranges are restricted to being powers of 2. This is best illustrated by an example.

If a breakpoint is to occur when the address is in the first 256 bytes of memory, but not in the first 32 bytes, the watchpoint registers should be programmed as follows:

- Watchpoint 1 is programmed with an address value of 0x00000000 and an address mask of 0x0000001F. The **ENABLE** bit is cleared. All other Watchpoint 1 registers are programmed as normal for a breakpoint. An address within the first 32 bytes will cause the **RANGE** output to go HIGH but the breakpoint will not be triggered.
- Watchpoint 0 is programmed with an address value of 0x00000000 and an address mask of 0x000000FF. The **ENABLE** bit is set and the **RANGE** bit programmed to match a '0'. All other Watchpoint 0 registers are programmed as normal for a breakpoint.

If Watchpoint 0 matches but Watchpoint 1 does not (ie the **RANGE** input to Watchpoint 0 is '0'), then the breakpoint will be triggered.

## 8.5 Disabling ICEbreaker

ICEbreaker may be disabled by wiring the **DBGEN** input LOW.

When **DBGEN** is LOW, **BREAKPT** and **DBGRQ** to the ARM7D core are forced LOW, **DBGACK** from the ARM7DI is also forced LOW and the **IFEN** input to the ARM7D core is forced HIGH, enabling interrupts to be detected by ARM7D.

When **DBGEN** is LOW, ICEbreaker is also put into a low-power mode.

## 8.6 ICEbreaker Timing

The **EXTERN1** and **EXTERN0** inputs are sampled by ICEbreaker on the falling edge of **ECLK**. Sufficient set-up and hold time must therefore be allowed for these signals.

## 8.7 ICEBreaker Programming Restriction

The ICEBreaker watchpoint registers should only be programmed when the clock to the ARM7D core is stopped. This can be achieved by putting the core into the debug state.

The reason for this restriction is that if the core continues to run at **ECLK** rates when ICEbreaker is being programmed at **TCK** rates, then it is possible for the **BREAKPT** signal to be asserted asynchronously to the core.

This restriction does not apply if **MCLK** and **TCK** are driven from the same clock, or if it is known that the breakpoint or watchpoint condition can only occur some time after ICEbreaker has been programmed.

Note that this restriction does not apply in any event to the Debug Control or Status Registers.

# 9.0 Instruction Cycle Operations

In the following tables **nMREQ** and **SEQ** (which are pipelined up to one cycle ahead of the cycle to which they apply) are shown in the cycle in which they appear, so they predict the type of the next cycle. The address, **nBW**, **nRW**, and **nOPC** (which appear up to half a cycle ahead) are shown in the cycle to which they apply.

## 9.1 Branch and branch with link

A branch instruction calculates the branch destination in the first cycle, whilst performing a prefetch from the current PC. This prefetch is done in all cases, since by the time the decision to take the branch has been reached it is already too late to prevent the prefetch.

During the second cycle a fetch is performed from the branch destination, and the return address is stored in register 14 if the link bit is set.

The third cycle performs a fetch from the destination + 4, refilling the instruction pipeline, and if the branch is with link R14 is modified (4 is subtracted from it) to simplify return from SUB PC,R14,#4 to MOV PC,R14. This makes the STM..{R14} LDM..{PC} type of subroutine work correctly. The cycle timings are shown below in *Table 14: Branch Instruction Cycle Operations*

| Cycle | Address | nBW | nRW | Data | nMREQ | SEQ | nOPC |
|---|---|---|---|---|---|---|---|
| 1 | pc+8 | 1 | 0 | (pc + 8) | 0 | 0 | 0 |
| 2 | alu | 1 | 0 | (alu) | 0 | 1 | 0 |
| 3 | alu+4 | 1 | 0 | (alu + 4) | 0 | 1 | 0 |
| | alu+8 | | | | | | |

**Table 14: Branch Instruction Cycle Operations**

*pc* is the address of the branch instruction

*alu* is an address calculated by ARM7DI

*(alu)* are the contents of that address, etc

## 9.2 Data Operations

A data operation executes in a single datapath cycle except where the shift is determined by the contents of a register. A register is read onto the A bus, and a second register or the immediate field onto the B bus. The ALU combines the A bus source and the shifted B bus source according to the operation specified in the instruction, and the result (when required) is written to the destination register. (Compares and tests do not produce results, only the ALU status flags are affected.)

An instruction prefetch occurs at the same time as the above operation, and the program counter is incremented.

When the shift length is specified by a register, an additional datapath cycle occurs before the above operation to copy the bottom 8 bits of that register into a holding latch in the barrel shifter. The instruction prefetch will occur during this first cycle, and the operation cycle will be internal (ie will not request memory). This internal cycle can be merged with the following sequential access by the memory manager as the address remains stable through both cycles.

The PC may be one or more of the register operands. When it is the destination external bus activity may be affected. If the result is written to the PC, the contents of the instruction pipeline are invalidated, and the address for the next instruction prefetch is taken from the ALU rather than the address incrementer. The instruction pipeline is refilled before any further execution takes place, and during this time exceptions are locked out.

PSR Transfer operations exhibit the same timing characteristics as the data operations except that the PC is never used as a source or destination register. The cycle timings are shown below *Table 15: Data Operation Instruction Cycle Operations*.

| | Cycle | Address | nBW | nRW | Data | nMREQ | SEQ | nOPC |
|---|---|---|---|---|---|---|---|---|
| normal | 1 | pc+8 | 1 | 0 | (pc+8) | 0 | 1 | 0 |
| | | pc+12 | | | | | | |
| | | | | | | | | |
| dest=pc | 1 | pc+8 | 1 | 0 | (pc+8) | 0 | 0 | 0 |
| | 2 | alu | 1 | 0 | (alu) | 0 | 1 | 0 |
| | 3 | alu+4 | 1 | 0 | (alu+4) | 0 | 1 | 0 |
| | | alu+8 | | | | | | |
| | | | | | | | | |
| shift(Rs) | 1 | pc+8 | 1 | 0 | (pc+8) | 1 | 0 | 0 |
| | 2 | pc+12 | 1 | 0 | - | 0 | 1 | 1 |
| | | pc+12 | | | | | | |
| | | | | | | | | |
| shift(Rs) | 1 | pc+8 | 1 | 0 | (pc+8) | 1 | 0 | 0 |
| dest=pc | 2 | pc+12 | 1 | 0 | - | 0 | 0 | 1 |
| | 3 | alu | 1 | 0 | (alu) | 0 | 1 | 0 |
| | 4 | alu+4 | 1 | 0 | (alu+4) | 0 | 1 | 0 |
| | | alu+8 | | | | | | |

**Table 15: Data Operation Instruction Cycle Operations**

## 9.3 Multiply and multiply accumulate

The multiply instructions make use of special hardware which implements a 2 bit Booth's algorithm with early termination. During the first cycle the accumulate Register is brought to the ALU, which either transmits it or produces zero (depending on the instruction being MLA or MUL) to initialise the destination register. During the same cycle, the multiplier (Rs) is loaded into the Booth's shifter via the A bus.

The datapath then cycles, adding the multiplicand (Rm) to, subtracting it from, or just transmitting, the result register. The multiplicand is shifted in the Nth cycle by 2N or 2N+1 bits, under control of the Booth's logic. The multiplier is shifted right 2 bits per cycle, and when it is zero the instruction terminates (possibly after an additional cycle to clear a pending borrow).

All cycles except the first are internal. The cycle timings are shown below in *Table 16: Multiply Instruction Cycle Operations*.

| | Cycle | Address | nBW | nRW | Data | nMREQ | SEQ | nOPC |
|---|---|---|---|---|---|---|---|---|
| (Rs)=0,1 | 1 | pc+8 | 1 | 0 | (pc+8) | 1 | 0 | 0 |
| | 2 | pc+12 | 1 | 0 | - | 0 | 1 | 1 |
| | | pc+12 | | | (pc+8) | | | |
| | | | | | | | | |
| (Rs)>1 | 1 | pc+8 | 1 | 0 | (pc+8) | 1 | 0 | 0 |
| | 2 | pc+12 | 1 | 0 | - | 1 | 0 | 1 |
| | • | pc+12 | 1 | 0 | - | 1 | 0 | 1 |
| | m | pc+12 | 1 | 0 | - | 1 | 0 | 1 |
| | m+1 | pc+12 | 1 | 0 | - | 0 | 1 | 1 |
| | | pc+12 | | | | | | |

**Table 16: Multiply Instruction Cycle Operations**

*m* is the number of cycles required by the Booth's algorithm; see the section on instruction speeds.

## 9.4 Load register

The first cycle of a load register instruction performs the address calculation. The data is fetched from memory during the second cycle, and the base register modification is performed during this cycle (if required). During the third cycle the data is transferred to the destination register, and external memory is unused. This third cycle may normally be merged with the following prefetch to form one memory N-cycle. The cycle timings are shown below in *Table 17: Load Register Instruction Cycle Operations*.

Either the base or the destination (or both) may be the PC, and the prefetch sequence will be changed if the PC is affected by the instruction.

The data fetch may abort, and in this case the destination modification is prevented.

| | Cycle | Address | nBW | nRW | Data | nMREQ | SEQ | nOPC |
|---|---|---|---|---|---|---|---|---|
| normal | 1 | pc+8 | 1 | 0 | (pc+8) | 0 | 0 | 0 |
| | 2 | alu | b/w | 0 | (alu) | 1 | 0 | 1 |
| | 3 | pc+12 | 1 | 0 | - | 0 | 1 | 1 |
| | | pc+12 | | | | | | |
| | | | | | | | | |
| dest=pc | 1 | pc+8 | 1 | 0 | (pc+8) | 0 | 0 | 0 |
| | 2 | alu | b/w | 0 | pc’ | 1 | 0 | 1 |
| | 3 | pc+12 | 1 | 0 | - | 0 | 0 | 1 |
| | 4 | pc’ | 1 | 0 | (pc’) | 0 | 1 | 0 |
| | 5 | pc’+4 | 1 | 0 | (pc’+4) | 0 | 1 | 0 |
| | | pc’+8 | | | | | | |

**Table 17: Load Register Instruction Cycle Operations**

## 9.5 Store register

The first cycle of a store register is similar to the first cycle of load register. During the second cycle the base modification is performed, and at the same time the data is written to memory. There is no third cycle. The cycle timings are shown below in *Table 18: Store Register Instruction Cycle Operations*.

| Cycle | Address | nBW | nRW | Data | nMREQ | SEQ | nOPC |
|---|---|---|---|---|---|---|---|
| 1 | pc+8 | 1 | 0 | (pc+8) | 0 | 0 | 0 |
| 2 | alu | b/w | 1 | Rd | 0 | 0 | 1 |
| | pc+12 | | | | | | |

**Table 18: Store Register Instruction Cycle Operations**

## 9.6 Load multiple registers

The first cycle of LDM is used to calculate the address of the first word to be transferred, whilst performing a prefetch from memory. The second cycle fetches the first word, and performs the base modification. During the third cycle, the first word is moved to the appropriate destination register while the second word is fetched from memory, and the modified base is latched internally in case it is needed to patch up after an abort. The third cycle is repeated for subsequent fetches until the last data word has been accessed, then the final (internal) cycle moves the last word to its destination register. The cycle timings are shown in *Table 19: Load Multiple Registers Instruction Cycle Operations.*

The last cycle may be merged with the next instruction prefetch to form a single memory N-cycle.

If an abort occurs, the instruction continues to completion, but all register writing after the abort is prevented. The final cycle is altered to restore the modified base register (which may have been overwritten by the load activity before the abort occurred).

When the PC is in the list of registers to be loaded the current instruction pipeline must be invalidated.

Note that the PC is always the last register to be loaded, so an abort at any point will prevent the PC from being overwritten.

| | Cycle | Address | nBW | nRW | Data | nMREQ | SEQ | nOPC |
|---|---|---|---|---|---|---|---|---|
| 1 register | 1 | pc+8 | 1 | 0 | (pc+8) | 0 | 0 | 0 |
| | 2 | alu | 1 | 0 | (alu) | 1 | 0 | 1 |
| | 3 | pc+12 | 1 | 0 | - | 0 | 1 | 1 |
| | | pc+12 | | | | | | |
| | | | | | | | | |
| 1 register | 1 | pc+8 | 1 | 0 | (pc+8) | 0 | 0 | 0 |
| dest=pc | 2 | alu | 1 | 0 | pc’ | 1 | 0 | 1 |
| | 3 | pc+12 | 1 | 0 | - | 0 | 0 | 1 |
| | 4 | pc’ | 1 | 0 | (pc’) | 0 | 1 | 0 |
| | 5 | pc’+4 | 1 | 0 | (pc’+4) | 0 | 1 | 0 |
| | | pc’+8 | | | | | | |
| | | | | | | | | |
| n registers | 1 | pc+8 | 1 | 0 | (pc+8) | 0 | 0 | 0 |
| (n>1) | 2 | alu | 1 | 0 | (alu) | 0 | 1 | 1 |
| | • | alu+• | 1 | 0 | (alu+•) | 0 | 1 | 1 |
| | n | alu+• | 1 | 0 | (alu+•) | 0 | 1 | 1 |
| | n+1 | alu+• | 1 | 0 | (alu+•) | 1 | 0 | 1 |
| | n+2 | pc+12 | 1 | 0 | - | 0 | 1 | 1 |
| | | pc+12 | | | | | | |
| | | | | | | | | |
| n registers | 1 | pc+8 | 1 | 0 | (pc+8) | 0 | 0 | 0 |
| (n>1) | 2 | alu | 1 | 0 | (alu) | 0 | 1 | 1 |
| incl pc | • | alu+• | 1 | 0 | (alu+•) | 0 | 1 | 1 |
| | n | alu+• | 1 | 0 | (alu+•) | 0 | 1 | 1 |
| | n+1 | alu+• | 1 | 0 | pc’ | 1 | 0 | 1 |
| | n+2 | pc+12 | 1 | 0 | - | 0 | 0 | 1 |
| | n+3 | pc’ | 1 | 0 | (pc’) | 0 | 1 | 0 |
| | n+4 | pc’+4 | 1 | 0 | (pc’+4) | 0 | 1 | 0 |
| | | pc’+8 | | | | | | |

**Table 19: Load Multiple Registers Instruction Cycle Operations**

## 9.7 Store multiple registers

Store multiple proceeds very much as load multiple, without the final cycle. The restart problem is much more straightforward here, as there is no wholesale overwriting of registers to contend with. The cycle timings are shown in *Table 20: Store Multiple Registers Instruction Cycle Operations*.

| | Cycle | Address | nBW | nRW | Data | nMREQ | SEQ | nOPC |
|---|---|---|---|---|---|---|---|---|
| 1 register | 1 | pc+8 | 1 | 0 | (pc+8) | 0 | 0 | 0 |
| | 2 | alu | 1 | 1 | Ra | 0 | 0 | 1 |
| | | pc+12 | | | | | | |
| | | | | | | | | |
| n registers | 1 | pc+8 | 1 | 0 | (pc+8) | 0 | 0 | 0 |
| (n>1) | 2 | alu | 1 | 1 | Ra | 0 | 1 | 1 |
| | • | alu+• | 1 | 1 | R• | 0 | 1 | 1 |
| | n | alu+• | 1 | 1 | R• | 0 | 1 | 1 |
| | n+1 | alu+• | 1 | 1 | R• | 0 | 0 | 1 |
| | | pc+12 | | | | | | |

**Table 20: Store Multiple Registers Instruction Cycle Operations**

## 9.8 Data swap

This is similar to the load and store register instructions, but the actual swap takes place in cycles 2 and 3. In the second cycle, the data is fetched from external memory. In the third cycle, the contents of the source register are written out to the external memory. The data read in cycle 2 is written into the destination register during the fourth cycle. The cycle timings are shown below in *Table 21: Data Swap Instruction Cycle Operations*.

The **LOCK** output of ARM7DI is driven HIGH for the duration of the swap operation (cycles 2 & 3) to indicate that both cycles should be allowed to complete without interruption.

The data swapped may be a byte or word quantity (b / w).

The swap operation may be aborted in either the read or write cycle, and in both cases the destination register will not be affected.

| Cycle | Address | nBW | nRW | Data | nMREQ | SEQ | nOPC | LOCK |
|---|---|---|---|---|---|---|---|---|
| 1 | pc+8 | 1 | 0 | (pc+8) | 0 | 0 | 0 | 0 |
| 2 | Rn | b/w | 0 | (Rn) | 0 | 0 | 1 | 1 |
| 3 | Rn | b/w | 1 | Rm | 1 | 0 | 1 | 1 |
| 4 | pc+12 | 1 | 0 | - | 0 | 1 | 1 | 0 |
| | pc+12 | | | | | | | |

**Table 21: Data Swap Instruction Cycle Operations**

## 9.9 Software interrupt and exception entry

Exceptions (and software interrupts) force the PC to a particular value and refill the instruction pipeline from there. During the first cycle the forced address is constructed, and a mode change may take place. The return address is moved to R14 and the CPSR to SPSR_svc.

During the second cycle the return address is modified to facilitate return, though this modification is less useful than in the case of branch with link.

The third cycle is required only to complete the refilling of the instruction pipeline. The cycle timings are shown below in *Table 22: Software Interrupt Instruction Cycle Operations*.

| Cycle | Address | nBW | nRW | Data | nMREQ | SEQ | nOPC | nTRANS | Mode |
|---|---|---|---|---|---|---|---|---|---|
| 1 | pc+8 | 1 | 0 | (pc+8) | 0 | 0 | 0 | C | old mode |
| 2 | Xn | 1 | 0 | (Xn) | 0 | 1 | 0 | 1 | exception mode |
| 3 | Xn+4 | 1 | 0 | (Xn+4) | 0 | 1 | 0 | 1 | exception mode |
| | Xn+8 | | | | | | | | |

**Table 22: Software Interrupt Instruction Cycle Operations**

where C represents the current mode-dependent value.

For software interrupts, *pc* is the address of the SWI instruction. For interrupts and reset, *pc* is the address of the instruction following the last one to be executed before entering the exception. For prefetch abort, *pc* is the address of the aborting instruction. For data abort, *pc* is the address of the instruction following the one which attempted the aborted data transfer. *Xn* is the appropriate trap address.

## 9.10 Coprocessor data operation

A coprocessor data operation is a request from ARM7DI for the coprocessor to initiate some action. The action need not be completed for some time, but the coprocessor must commit to doing it before driving **CPB** LOW.

If the coprocessor can never do the requested task, it should leave **CPA** and **CPB** HIGH. If it can do the task, but can't commit right now, it should drive **CPA** LOW but leave **CPB** HIGH until it can commit. ARM7DI will busy-wait until **CPB** goes LOW. The cycle timings are shown in *Table 23: Coprocessor Data Operation Instruction Cycle Operations*.

| | Cycle | Address | nBW | nRW | Data | nMREQ | SEQ | nOPC | nCPI | CPA | CPB |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ready | 1 | pc+8 | 1 | 0 | (pc+8) | 0 | 0 | 0 | 0 | 0 | 0 |
| | | pc+12 | | | | | | | | | |
| | | | | | | | | | | | |
| not ready | 1 | pc+8 | 1 | 0 | (pc+8) | 1 | 0 | 0 | 0 | 0 | 1 |
| | 2 | pc+8 | 1 | 0 | - | 1 | 0 | 1 | 0 | 0 | 1 |
| | • | pc+8 | 1 | 0 | - | 1 | 0 | 1 | 0 | 0 | 1 |
| | n | pc+8 | 1 | 0 | - | 0 | 0 | 1 | 0 | 0 | 0 |
| | | pc+12 | | | | | | | | | |

**Table 23: Coprocessor Data Operation Instruction Cycle Operations**

## 9.11 Coprocessor data transfer (from memory to coprocessor)

Here the coprocessor should commit to the transfer only when it is ready to accept the data. When **CPB** goes LOW, ARM7DI will produce addresses and expect the coprocessor to take the data at sequential cycle rates. The coprocessor is responsible for determining the number of words to be transferred, and indicates the last transfer cycle by driving **CPA** and **CPB** HIGH.

ARM7DI spends the first cycle (and any busy-wait cycles) generating the transfer address, and performs the write-back of the address base during the transfer cycles. The cycle timings are shown in *Table 24: Coprocessor Data Transfer Instruction Cycle Operations*.

| | Cycle | Address | nBW | nRW | Data | nMREQ | SEQ | nOPC | nCPI | CPA | CPB |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 register | 1 | pc+8 | 1 | 0 | (pc+8) | 0 | 0 | 0 | 0 | 0 | 0 |
| ready | 2 | alu | 1 | 0 | (alu) | 0 | 0 | 1 | 1 | 1 | 1 |
| | | pc+12 | | | | | | | | | |
| | | | | | | | | | | | |
| 1 register | 1 | pc+8 | 1 | 0 | (pc+8) | 1 | 0 | 0 | 0 | 0 | 1 |
| not ready | 2 | pc+8 | 1 | 0 | - | 1 | 0 | 1 | 0 | 0 | 1 |
| | • | pc+8 | 1 | 0 | - | 1 | 0 | 1 | 0 | 0 | 1 |
| | n | pc+8 | 1 | 0 | - | 0 | 0 | 1 | 0 | 0 | 0 |
| | n+1 | alu | 1 | 0 | (alu) | 0 | 0 | 1 | 1 | 1 | 1 |
| | | pc+12 | | | | | | | | | |
| | | | | | | | | | | | |
| n registers | 1 | pc+8 | 1 | 0 | (pc+8) | 0 | 0 | 0 | 0 | 0 | 0 |
| (n>1) | 2 | alu | 1 | 0 | (alu) | 0 | 1 | 1 | 1 | 0 | 0 |
| ready | • | alu+• | 1 | 0 | (alu+•) | 0 | 1 | 1 | 1 | 0 | 0 |
| | n | alu+• | 1 | 0 | (alu+•) | 0 | 1 | 1 | 1 | 0 | 0 |
| | n+1 | alu+• | 1 | 0 | (alu+•) | 0 | 0 | 1 | 1 | 1 | 1 |
| | | pc+12 | | | | | | | | | |
| | | | | | | | | | | | |
| m registers | 1 | pc+8 | 1 | 0 | (pc+8) | 1 | 0 | 0 | 0 | 0 | 1 |
| (m>1) | 2 | pc+8 | 1 | 0 | - | 1 | 0 | 1 | 0 | 0 | 1 |
| not ready | • | pc+8 | 1 | 0 | - | 1 | 0 | 1 | 0 | 0 | 1 |
| | n | pc+8 | 1 | 0 | - | 0 | 0 | 1 | 0 | 0 | 0 |
| | n+1 | alu | 1 | 0 | (alu) | 0 | 1 | 1 | 1 | 0 | 0 |
| | • | alu+• | 1 | 0 | (alu+•) | 0 | 1 | 1 | 1 | 0 | 0 |
| | n+m | alu+• | 1 | 0 | (alu+•) | 0 | 1 | 1 | 1 | 0 | 0 |
| | n+m+1 | alu+• | 1 | 0 | (alu+•) | 0 | 0 | 1 | 1 | 1 | 1 |
| | | pc+12 | | | | | | | | | |

**Table 24: Coprocessor Data Transfer Instruction Cycle Operations**

## 9.12 Coprocessor data transfer (from coprocessor to memory)

The ARM7DI controls these instructions exactly as for memory to coprocessor transfers, with the one exception that the nRW line is inverted during the transfer cycle. The cycle timings are show in *Table 25: Coprocessor Data Transfer Instruction Cycle Operations*.

| | Cycle | Address | nBW | nRW | Data | nMREQ | SEQ | nOPC | nCPI | CPA | CPB |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 register | 1 | pc+8 | 1 | 0 | (pc+8) | 0 | 0 | 0 | 0 | 0 | 0 |
| ready | 2 | alu | 1 | 1 | CPdata | 0 | 0 | 1 | 1 | 1 | 1 |
| | | pc+12 | | | | | | | | | |
| 1 register | 1 | pc+8 | 1 | 0 | (pc+8) | 1 | 0 | 0 | 0 | 0 | 1 |
| not ready | 2 | pc+8 | 1 | 0 | - | 1 | 0 | 1 | 0 | 0 | 1 |
| | • | pc+8 | 1 | 0 | - | 1 | 0 | 1 | 0 | 0 | 1 |
| | n | pc+8 | 1 | 0 | - | 0 | 0 | 1 | 0 | 0 | 0 |
| | n+1 | alu | 1 | 1 | CPdata | 0 | 0 | 1 | 1 | 1 | 1 |
| | | pc+12 | | | | | | | | | |
| n registers | 1 | pc+8 | 1 | 0 | (pc+8) | 0 | 0 | 0 | 0 | 0 | 0 |
| (n>1) | 2 | alu | 1 | 1 | CPdata | 0 | 1 | 1 | 1 | 0 | 0 |
| ready | • | alu+• | 1 | 1 | CPdata | 0 | 1 | 1 | 1 | 0 | 0 |
| | n | alu+• | 1 | 1 | CPdata | 0 | 1 | 1 | 1 | 0 | 0 |
| | n+1 | alu+• | 1 | 1 | CPdata | 0 | 0 | 1 | 1 | 1 | 1 |
| | | pc+12 | | | | | | | | | |
| m registers | 1 | pc+8 | 1 | 0 | (pc+8) | 1 | 0 | 0 | 0 | 0 | 1 |
| (m>1) | 2 | pc+8 | 1 | 0 | - | 1 | 0 | 1 | 0 | 0 | 1 |
| not ready | • | pc+8 | 1 | 0 | - | 1 | 0 | 1 | 0 | 0 | 1 |
| | n | pc+8 | 1 | 0 | - | 0 | 0 | 1 | 0 | 0 | 0 |
| | n+1 | alu | 1 | 1 | CPdata | 0 | 1 | 1 | 1 | 0 | 0 |
| | • | alu+• | 1 | 1 | CPdata | 0 | 1 | 1 | 1 | 0 | 0 |
| | n+m | alu+• | 1 | 1 | CPdata | 0 | 1 | 1 | 1 | 0 | 0 |
| | n+m+1 | alu+• | 1 | 1 | CPdata | 0 | 0 | 1 | 1 | 1 | 1 |
| | | pc+12 | | | | | | | | | |

**Table 25: Coprocessor Data Transfer Instruction Cycle Operations**

## 9.13 Coprocessor register transfer (Load from coprocessor)

Here the busy-wait cycles are much as above, but the transfer is limited to one data word, and ARM7DI puts the word into the destination register in the third cycle. The third cycle may be merged with the following prefetch cycle into one memory N-cycle as with all ARM7DI register load instructions. The cycle timings are shown in *Table 26: Coprocessor register transfer (Load from coprocessor).*

| | Cycle | Address | nBW | nRW | Data | nMREQ | SEQ | nOPC | nCPI | CPA | CPB |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ready | 1 | pc+8 | 1 | 0 | (pc+8) | 1 | 1 | 0 | 0 | 0 | 0 |
| | 2 | pc+12 | 1 | 0 | CPdata | 1 | 0 | 1 | 1 | 1 | 1 |
| | 3 | pc+12 | 1 | 0 | - | 0 | 1 | 1 | 1 | - | - |
| | | pc+12 | | | | | | | | | |
| | | | | | | | | | | | |
| not ready | 1 | pc+8 | 1 | 0 | (pc+8) | 1 | 0 | 0 | 0 | 0 | 1 |
| | 2 | pc+8 | 1 | 0 | CPdata | 1 | 0 | 1 | 0 | 0 | 1 |
| | • | pc+8 | 1 | 0 | - | 1 | 0 | 1 | 0 | 0 | 1 |
| | n | pc+8 | 1 | 0 | - | 1 | 1 | 1 | 0 | 0 | 0 |
| | n+1 | pc+12 | 1 | 0 | CPdata | 1 | 0 | 1 | 1 | 1 | 1 |
| | n+2 | pc+12 | 1 | 0 | - | 0 | 1 | 1 | 1 | - | - |
| | | pc+12 | | | | | | | | | |

**Table 26: Coprocessor register transfer (Load from coprocessor)**

## 9.14 Coprocessor register transfer (Store to coprocessor)

As for the load from coprocessor, except that the last cycle is omitted. The cycle timings are shown below in *Table 27: Coprocessor register transfer (Store to coprocessor).*

| | Cycle | Address | nBW | nRW | Data | nMREQ | SEQ | nOPC | nCPI | CPA | CPB |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ready | 1 | pc+8 | 1 | 0 | (pc+8) | 1 | 1 | 0 | 0 | 0 | 0 |
| | 2 | pc+12 | 1 | 1 | Rd | 0 | 0 | 1 | 1 | 1 | 1 |
| | | pc+12 | | | | | | | | | |
| | | | | | | | | | | | |
| not ready | 1 | pc+8 | 1 | 0 | (pc+8) | 1 | 0 | 0 | 0 | 0 | 1 |
| | 2 | pc+8 | 1 | 0 | - | 1 | 0 | 1 | 0 | 0 | 1 |
| | • | pc+8 | 1 | 0 | - | 1 | 0 | 1 | 0 | 0 | 1 |
| | n | pc+8 | 1 | 0 | - | 1 | 1 | 1 | 0 | 0 | 0 |
| | n+1 | pc+12 | 1 | 1 | Rd | 0 | 0 | 1 | 1 | 1 | 1 |
| | | pc+12 | | | | | | | | | |

**Table 27: Coprocessor register transfer (Store to coprocessor)**

## 9.15 Undefined instructions and coprocessor absent

When a coprocessor detects a coprocessor instruction which it cannot perform, and this must include all undefined instructions, it must not drive **CPA** or **CPB** LOW. These will remain HIGH, causing the undefined instruction trap to be taken. Cycle timings are shown in *Table 28: Undefined Instruction Cycle Operations*.

| Cycle | Address | nBW | nRW | Data | nMREQ | SEQ | nOPC | nCPI | CPA | CPB | nTRANS | Mode |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | pc+8 | 1 | 0 | (pc+8) | 1 | 0 | 0 | 0 | 1 | 1 | Old | Old |
| 2 | pc+8 | 1 | 0 | - | 0 | 0 | 0 | 1 | 1 | 1 | Old | Old |
| 3 | Xn | 1 | 0 | (Xn) | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 00100 |
| 4 | Xn+4 | 1 | 0 | (Xn+4) | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 00100 |
| | Xn+8 | | | | | | | | | | | |

**Table 28: Undefined Instruction Cycle Operations**

## 9.16 Unexecuted instructions

Any instruction whose condition code is not met will fail to execute. It will add one cycle to the execution time of the code segment in which it is embedded (see *Table 29: Unexecuted Instruction Cycle Operations*).

| Cycle | Address | nBW | nRW | Data | nMREQ | SEQ | nOPC |
|---|---|---|---|---|---|---|---|
| 1 | pc+8 | 1 | 0 | (pc+8) | 0 | 1 | 0 |
| | pc+12 | | | | | | |

**Table 29: Unexecuted Instruction Cycle Operations**

## 9.17 Instruction Speed Summary

Due to the pipelined architecture of the CPU, instructions overlap considerably. In a typical cycle one instruction may be using the data path while the next is being decoded and the one after that is being fetched. For this reason the following table presents the incremental number of cycles required by an instruction, rather than the total number of cycles for which the instruction uses part of the processor. Elapsed time (in cycles) for a routine may be calculated from these figures which are shown in *Table 30: ARM Instruction Speed Summary*. These figures assume that the instruction is actually executed. Unexecuted instructions take one cycle.

| Instruction | Cycle count | Additional |
|---|---|---|
| Data Processing | 1S | + 1I for SHIFT(Rs)<br>+ 1S + 1N if R15 written |
| MSR, MRS | 1S | |
| LDR | 1S + 1N + 1I | + 1S + 1N if R15 loaded |
| STR | 2N | |
| LDM | nS + 1N + 1I | + 1S + 1N if R15 loaded |
| STM | (n-1)S + 2N | |
| SWP | 1S + 2N + 1I | |
| B,BL | 2S + 1N | |
| SWI, trap | 2S + 1N | |
| MUL,MLA | 1S + mI | |
| CDP | 1S + bI | |
| LDC,STC | (n-1)S + 2N + bI | |
| MCR | 1N + bI + 1C | |
| MRC | 1S + (b+1)I + 1C | |

**Table 30: ARM Instruction Speed Summary**

*n* is the number of words transferred.

*m* is the number of cycles required by the multiply algorithm, which is determined by the contents of Rs. Multiplication by any number between *2^(2m-3)* and *2^(2m-1)-1* takes 1S+mI *m* cycles for *1<m>16*. Multiplication by 0 or 1 takes 1S+1I cycles, and multiplication by any number greater than or equal to 2^(29) takes 1S+16I cycles. The maximum time for any multiply is thus 1S+16I cycles.

b is the number of cycles spent in the coprocessor busy-wait loop.

If the condition is not met all the instructions take one S-cycle. The cycle types N, S, I, and C are defined in *Chapter 5.0 Memory Interface.*

# 10.0 DC Parameters

***Subject to Change***

## 10.1 Absolute Maximum Ratings

| Symbol | Parameter | Min | Max | Units | Note |
|---|---|---|---|---|---|
| VDD | Supply voltage | VSS-0.3 | VSS+7.0 | V | 1 |
| Vin | Input voltage applied to any pin | VSS-0.3 | VDD+0.3 | V | 1 |
| Ts | Storage temperature | -40 | 125 | deg C | 1 |

**Table 31: ARM7DI DC Maximum Ratings**

Note:

These are stress ratings only. Exceeding the absolute maximum ratings may permanently damage the device. Operating the device at absolute maximum ratings for extended periods may affect device reliability.

## 10.2 DC Operating Conditions:

| Symbol | Parameter | Min | Typ | Max | Units | Notes |
|---|---|---|---|---|---|---|
| VDD | Supply voltage | 2.7 | 3.0 - 5.0 | 5.5 | V | |
| Vihc | IC input HIGH voltage | .8xVDD | | VDD | V | 1,2 |
| Vilc | IC input LOW voltage | 0.0 | | .2xVDD | V | 1,2 |
| Ta | Ambient operating temperature | 0 | | 70 | C | |

**Table 32: ARM7DI DC Operating Conditions**

Notes:

1. Voltages measured with respect to VSS.
2. IC CMOS-level inputs

# 11.0 AC Parameters

***** Important Note - Provisional Figures *****

The timing parameters given here are preliminary data and subject to change.

The AC timing diagrams presented in this section assume that the outputs of the ARM7DI have been loaded with the capacitive loads shown in the `Test Load' column of *Table 33: AC Test Loads*. These loads have been chosen as typical of the type of system in which ARM7DI might be employed.

The output drivers of the ARM7DI are CMOS inverters which exhibit a propagation delay that increases linearly with the increase in load capacitance. An “Output derating” figure is given for each output driver, showing the approximate rate of increase of output time with increasing load capacitance.

| Output Signal | Test Load (pF) | Output derating (ns/pF) |
|---|---|---|
| D[31:0] | TBD | TBD |
| A[31:0] | TBD | TBD |
| LOCK | TBD | TBD |
| nCPI | TBD | TBD |
| nMREQ | TBD | TBD |
| SEQ | TBD | TBD |
| nRW | TBD | TBD |
| nBW | TBD | TBD |
| nOPC | TBD | TBD |
| nTRANS | TBD | TBD |
| TDO | TBD | TBD |

**Table 33: AC Test Loads**

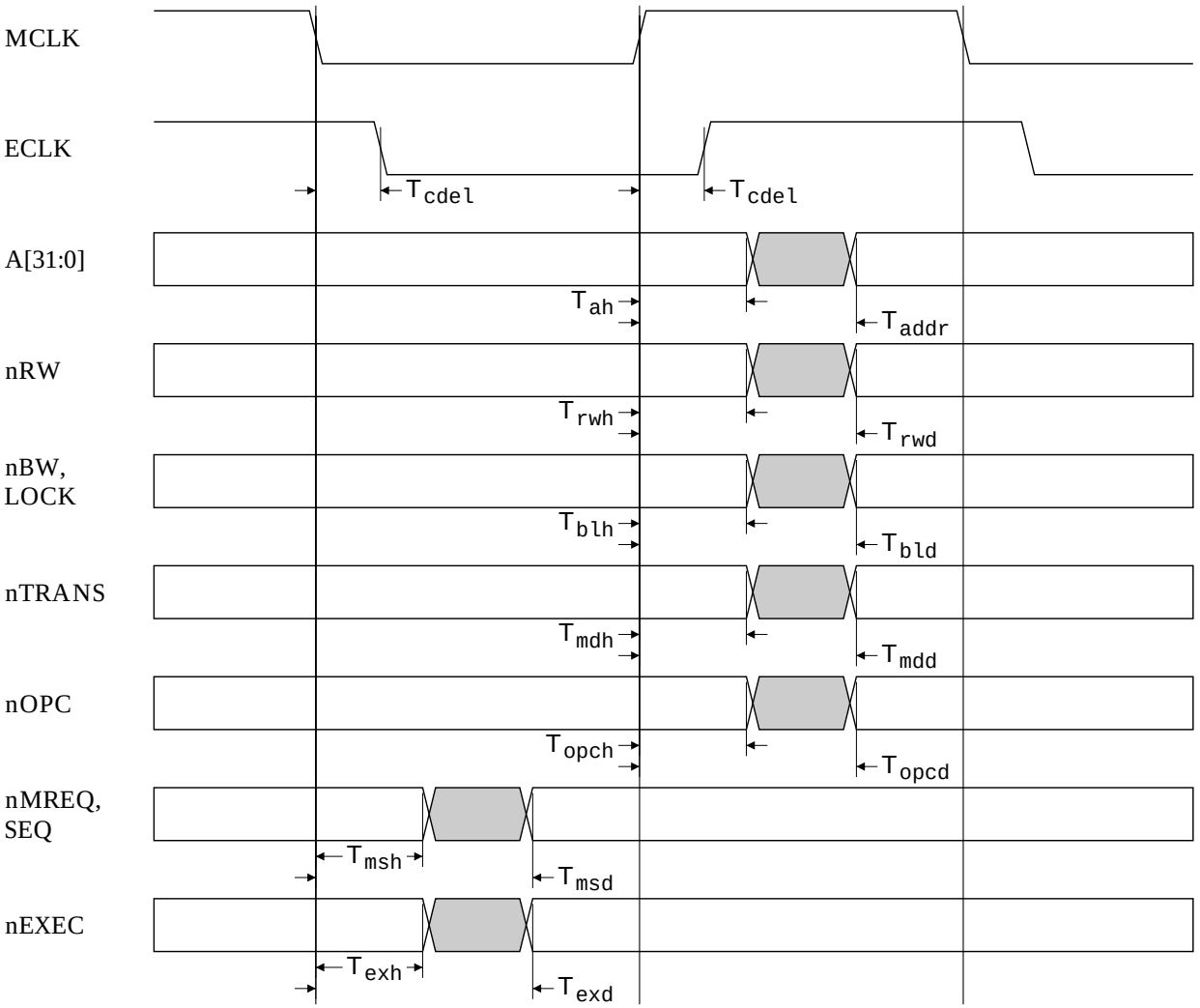



**Figure 53: General Timings**

Note: **nWAIT** and **ABE** are both HIGH during the cycle shown. Tcdel is the delay (on either edge) from **MCLK** changing to **ECLK** changing.

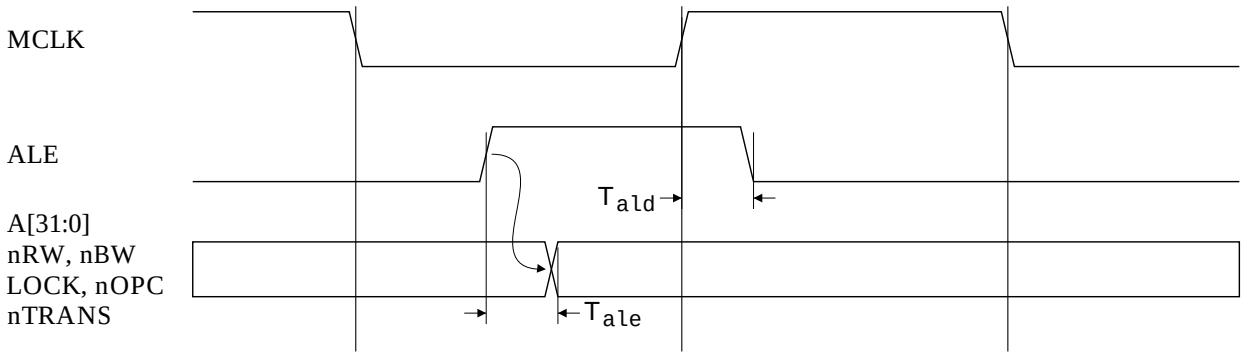



**Figure 54: Address Timing**

Note: Tald is the time by which **ALE** must be driven LOW in order to latch the current address in phase 2. If **ALE** is driven low after Tald, then a new address will be latched.

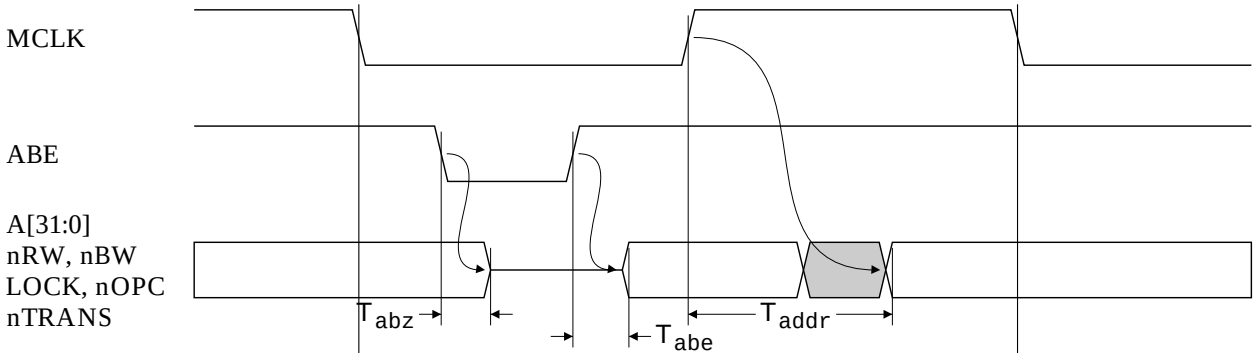



**Figure 55: Address Control**

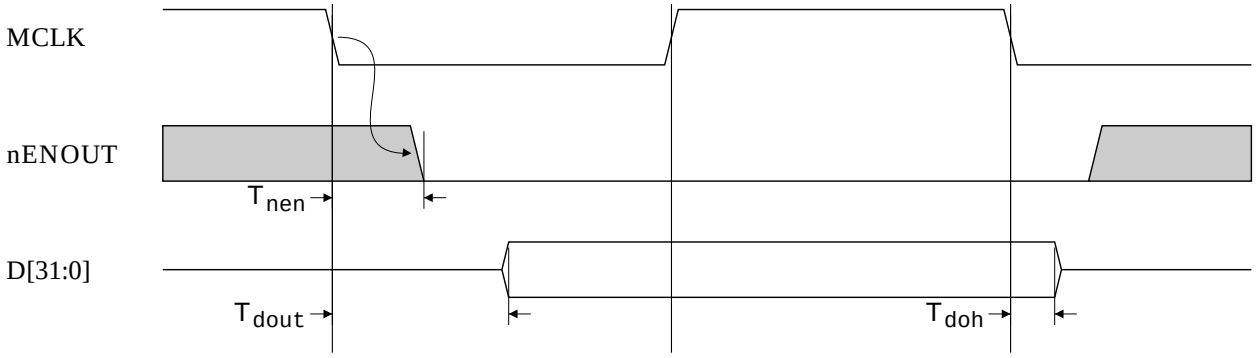



**Figure 56: Data Write Cycle**

Note: **DBE** is HIGH and **nENIN** is LOW during the cycle shown.

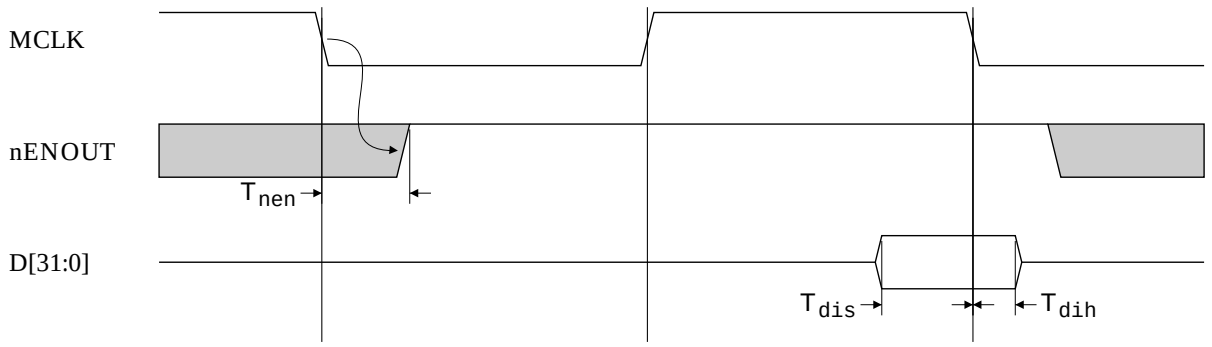



**Figure 57: Data Read Cycle**

Note: **DBE** is HIGH and **nENIN** is LOW during the cycle shown.

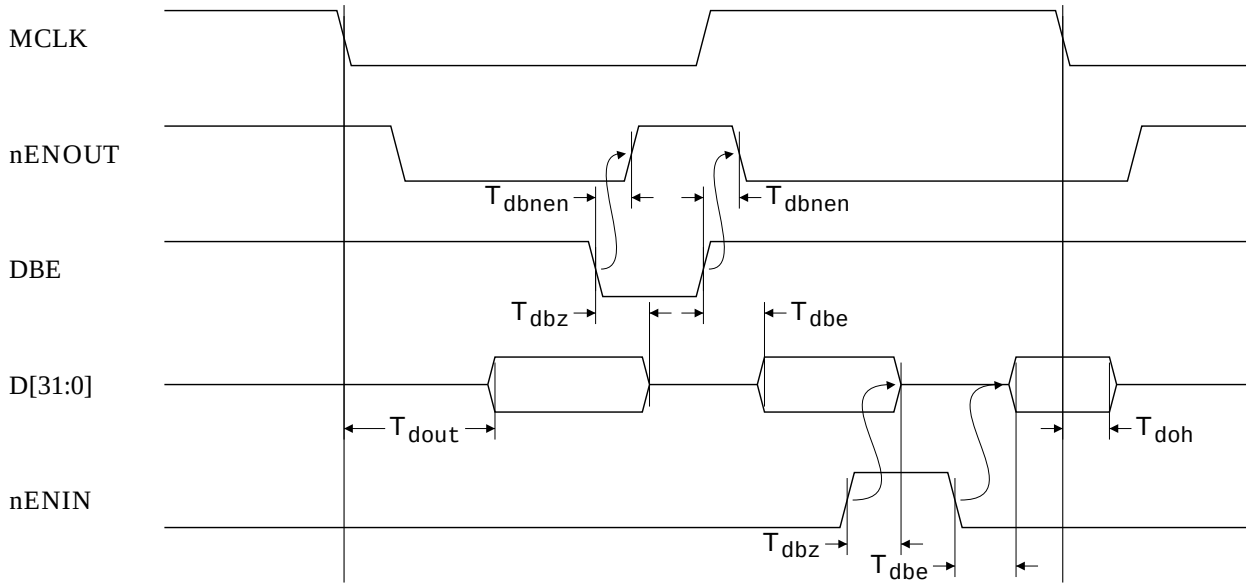



**Figure 58: Data Bus Control**

Note: The cycle shown is a data write cycle since **nENOUT** was driven LOW during phase 1. Here, **DBE** has first been used to modify the behaviour of the data bus, and then **nENIN**.

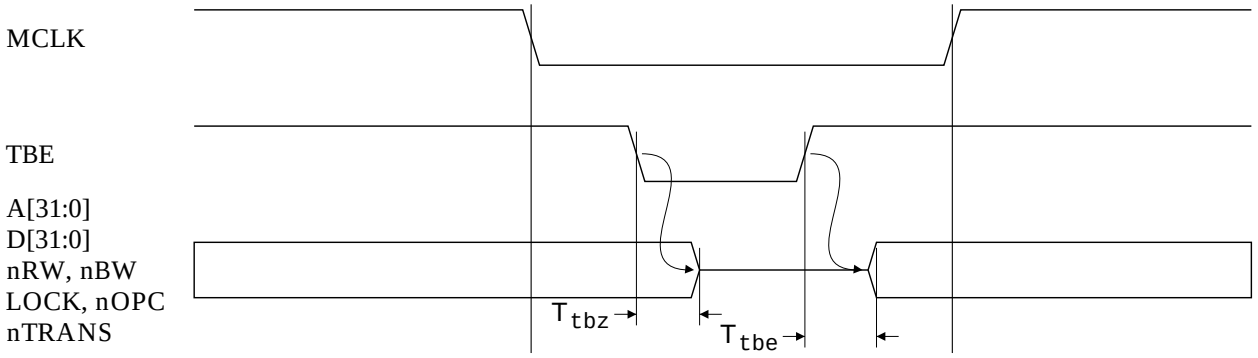



**Figure 59: Output 3-state Time**

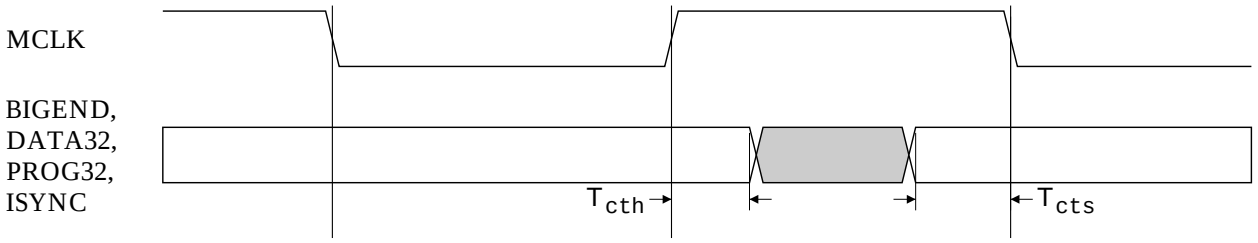



**Figure 60: Configuration Pin Timing**

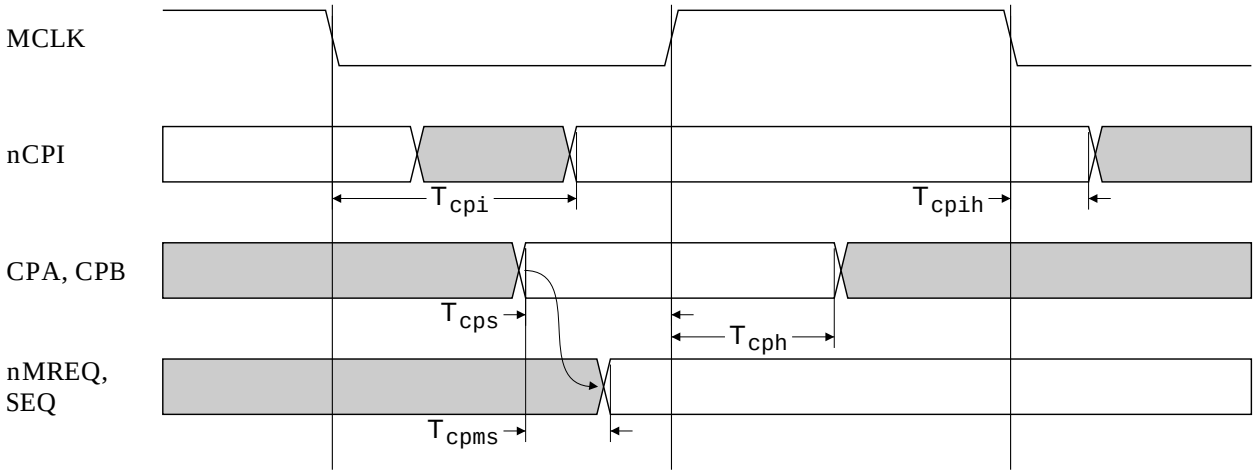



**Figure 61: Coprocessor Timing**

Note: Normally, **nMREQ** and **SEQ** become valid Tmsd after the falling edge of **MCLK**. In this cycle the ARM has been busy-waiting, waiting for a coprocessor to complete the instruction. If **CPA** and **CPB** change during phase 1, the timing of **nMREQ** and **SEQ** will depend on Tcpms. Most systems should be able to generate **CPA** and **CPB** during the previous phase 2, and so the timing of **nMREQ** and **SEQ** will always be Tmsd.

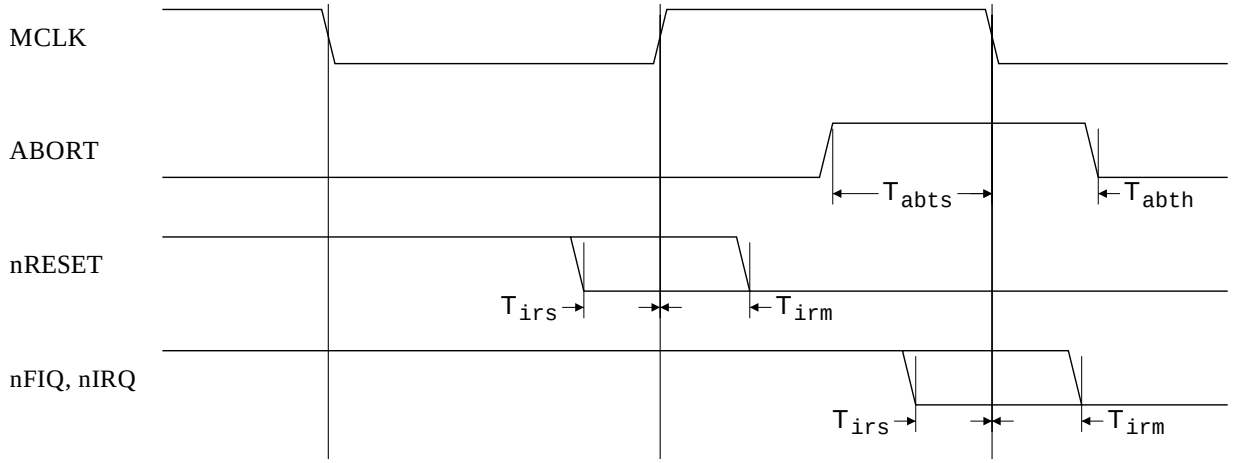



**Figure 62: Exception Timing**

Note: Tirs guarantees recognition of the interrupt (or reset) source by the corresponding clock edge. Tirm guarantees non-recognition by that clock edge. These inputs may be applied fully asynchronously where the exact cycle of recognition is unimportant.

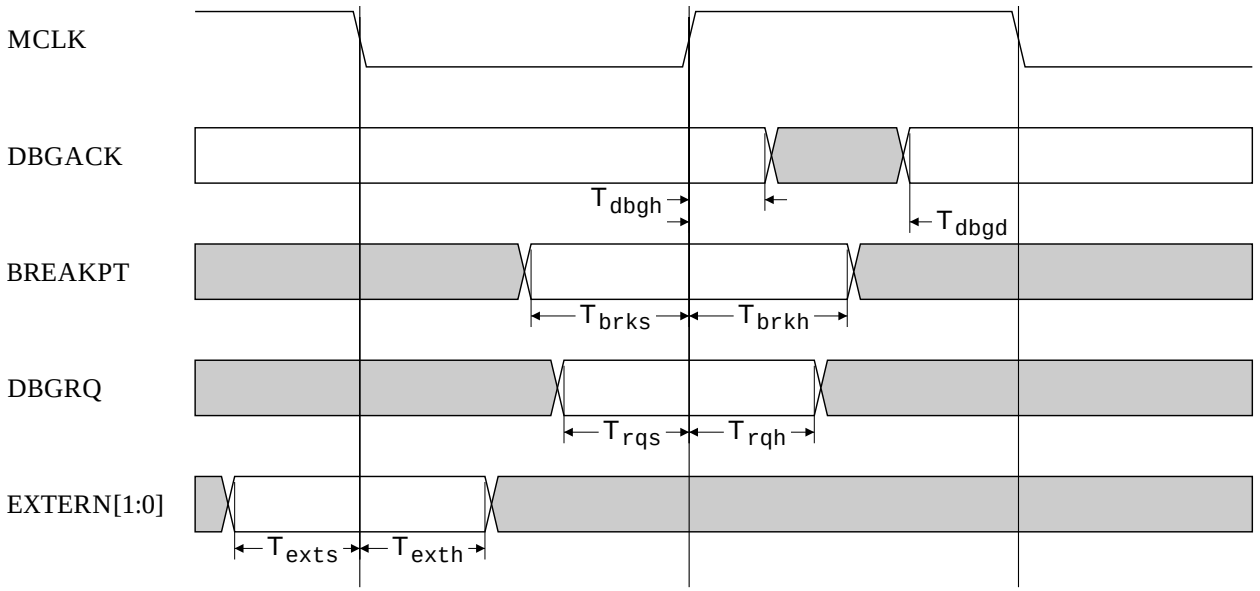



**Figure 63: Debug Timing**

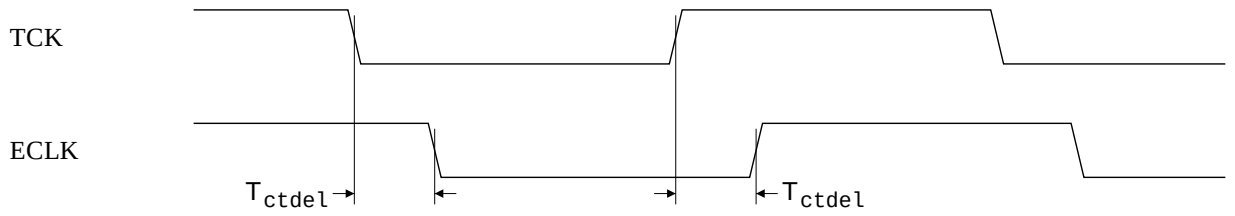



**Figure 64: TCK-ECLK relationship**

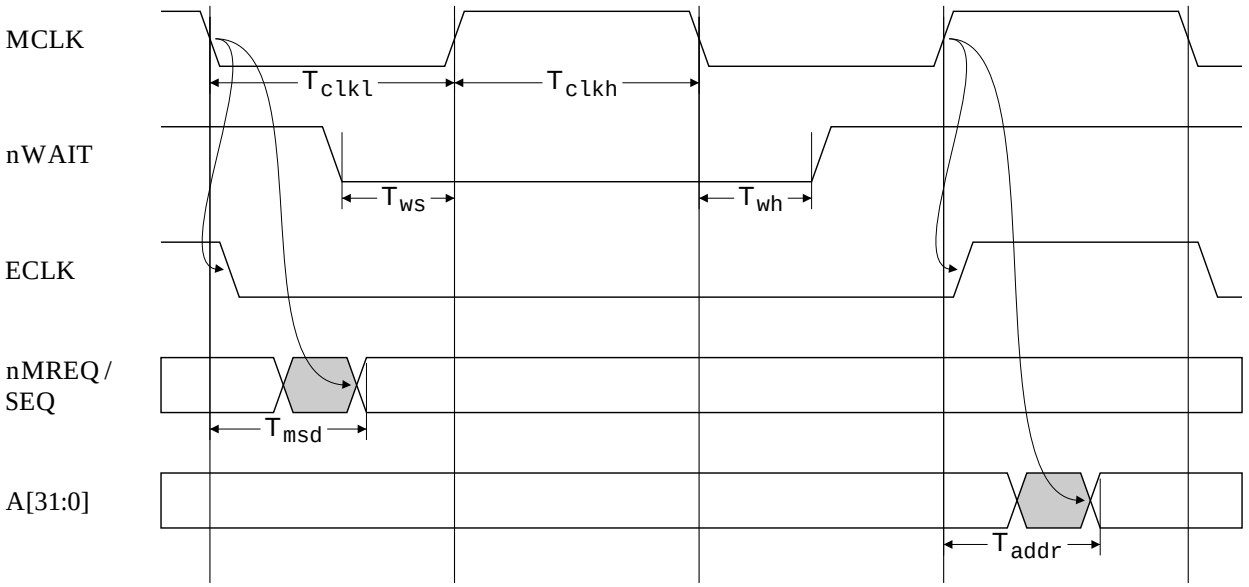



**Figure 65: MCLK Timing**

Note: The ARM core is not clocked by the HIGH phase of **MCLK** enveloped by **nWAIT**. Thus, during the cycles shown, **nMREQ** and **SEQ** change once, during the first LOW phase of **MCLK**, and **A[31:0]** change once, during the second HIGH phase of **MCLK**. For reference, ph2 is shown. This is the internal clock from which the core times all its activity. This signal is included to show how the high phase of the external **MCLK** has been removed from the internal core clock.

### 11.0.1 Notes on AC Parameters

All figures are provisional, and assume that 0.8 micron CMOS technology is used to fabricate the ARM7DI.

| Symbol | Parameter | Min | Max |
|---|---|---|---|
| Tckl | clock LOW time | 21 | |
| Tckh | clock HIGH time | 21 | |
| Tws | nWAIT setup to CKr | 3 | |
| Twh | nWAIT hold from CKf | 3 | |
| Tale | address latch open | | 4 |
| Tald | address latch time | | 4 |
| Taddr | CKr to address valid | | 12 |
| Tah | address hold time | 5 | |
| Tabe | address bus enable time | | 10 |
| Tabz | address bus disable time | | 15 |
| Tdout | data out delay | | 17 |
| Tdoh | data out hold | 5 | |
| Tdis | data in setup | 0 | |
| Tdih | data in hold | 5 | |
| Tnen | nENOUT delay time from MCLK falling | | 3 |
| Tdbe | Data bus enable time from DBE rising | | 10 |
| Tdbz | Data bus disable time from DBE falling | | 15 |
| Tdbnen | nENOUT changing from DBE changing | | 6 |
| Ttbz | Address and Data bus disable time from TBE falling | | 15 |
| Ttbe | Address and Data bus enable time from TBE rising | | 10 |
| Trwd | CKr to nRW valid | | 21 |
| Trwh | nRW hold time | 5 | |
| Tmsd | CKf to nMREQ & SEQ | | 23 |
| Tmsh | nMREQ & SEQ hold time | 5 | |
| Tbld | CKr to nBW & LOCK | | 21 |
| Tblh | nBW & LOCK hold | 5 | |
| Tmdd | CKr to nTRANS/nM[4:0] | | 21 |
| Tmdh | nTRANS/nM[4:0] | 5 | |
| Topcd | CKr to nOPC valid | | 11 |
| Topch | nOPC hold time | 5 | |
| Tcps | CPA, CPB setup | 7 | |
| Tcph | CPA,CPB hold time | 2 | |
| Tcpms | CPA, CPB to nMREQ, SEQ | | 15 |
| Tcpi | CKf to nCPI delay | | 11 |
| Tcpih | nCPI hold time | 5 | |
| Tcts | Config setup time | 10 | |
| Tcth | Config hold time | 5 | |
| Tabts | ABORT set up time to MCLK falling | 10 | |
| Tabth | ABORT hold time from MCLK falling | 5 | |
| Tirs | Interrupt set up time to MCLK falling for guaranteed recognition | 6 | |
| Tirm | Interrupt guaranteed non-recognition time | | 10 |

**Table 34: Provisional AC Parameters (units of nS)**

| Symbol | Parameter | Min | Max |
|---|---|---|---|
| Texd | MCLK falling to nEXEC valid | | 21 |
| Texh | nEXEC hold time from MCLK falling | 5 | |
| Tbrks | Set up time of BREAKPT to MCLK rising | 10 | |
| Tbrkh | Hold time of BREAKPT from MCLK rising | 5 | |
| Tdbgd | MCLK rising to DBGACK valid | | 5 |
| Tdbgh | DGBACK hold time from MCLK rising | 3 | |
| Trqs | DBGRQ set up time to MCLK falling for guaranteed recognition | 6 | |
| Trqh | DBGRQ guaranteed non-recognition time | 10 | |
| Tcdel | MCLK to ECLK delay | | 4 |
| Tctdel | TCK to ECLK delay | | 4 |
| Texts | EXTERN[1:0] set up time to MCLK falling | 5 | |
| Texth | EXTERN[1:0] hold time from MCLK falling | 5 | |

**Table 34: Provisional AC Parameters (units of nS)**

# 12.0 Appendix - Backward Compatibility

Two inputs, **PROG32** and **DATA32**, allow one of three processor configurations to be selected as follows:

(1) **26 bit program and data space** - (**PROG32** LOW, **DATA32** LOW). This configuration forces ARM7DI to operate like the earlier ARM processors with 26 bit address space. The programmer's model for these processors applies, but the new instructions to access the CPSR and SPSR registers operate as detailed elsewhere in this document. In this configuration it is impossible to select a 32 bit operating mode, and all exceptions (including address exceptions) enter the exception handler in the appropriate 26 bit mode.

(2) **26 bit program space and 32 bit data space** - (**PROG32** LOW, **DATA32** HIGH). This is the same as the 26 bit program and data space configuration, but with address exceptions disabled to allow data transfer operations to access the full 32 bit address space.

(3) **32 bit program and data space** - (**PROG32** HIGH, **DATA32** HIGH). This configuration extends the address space to 32 bits, introduces major changes in the programmer's model as described below and provides support for running existing 26 bit programs in the 32 bit environment.

The fourth processor configuration which is possible (26 bit data space and 32 bit program space) should not be selected.

When configured for 26 bit program space, ARM7DI is limited to operating in one of four modes known as the 26 bit modes. These modes correspond to the modes of the earlier ARM processors and are known as:

User26

FIQ26

IRQ26 and

Supervisor26.

These are the normal operating modes in this configuration and the 26 bit modes are only provided for backwards compatibility to allow execution of programs originally written for earlier ARM processors.

The differences between ARM7DI and the earlier ARM processors are documented in an *ARM Application Note 11 - "Differences between ARM6 and earlier ARM Processors"*