ARM 810

Preliminary Data Sheet

Document Number: ARM DDI 0081E
Issued: August 1996

ENGLAND
Advanced RISC Machines Limited
90 Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN
UK
Telephone: +44 1223 400400
Facsimile: +44 1223 400410
Email: info@armltd.co.uk

GERMANY
Advanced RISC Machines Limited
Otto-Hahn Str. 13b
85521 Ottobrunn-Riemerling
Munich
Germany
Telephone: +49 89 608 75545
Facsimile: +49 89 608 75599
Email: info@armltd.co.uk

JAPAN
Advanced RISC Machines K.K.
KSP West Bldg, 3F 300D, 3-2-1 Sakado
Takatsu-ku, Kawasaki-shi
Kanagawa
213 Japan
Telephone: +81 44 850 1301
Facsimile: +81 44 850 1308
Email: info@armltd.co.uk

USA
ARM USA Incorporated
Suite 5
985 University Avenue
Los Gatos
CA 95030 USA
Telephone: +1 408 399 5199
Facsimile: +1 408 399 8854
Email: info@arm.com

World Wide Web address: http://www.arm.com

ARM
Advanced RISC Machines

Proprietary Notice

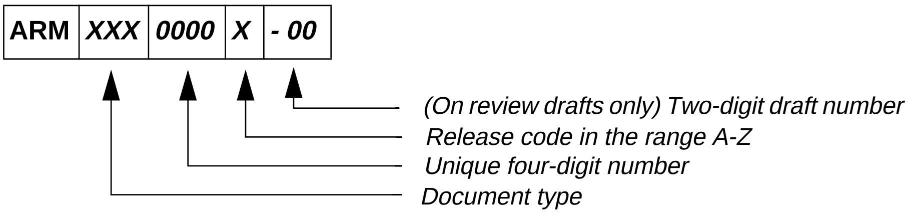
The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties or merchantability,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

This document 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 document, or any error or omission in such information, or any incorrect use of the product.

Key

Document Number

This document has a number which identifies it uniquely. The number is displayed on the front page and at the foot of each subsequent page.


ARM
XXX
0000
X
- 00
(On review drafts only) Two-digit draft number
Release code in the range A-Z
Unique four-digit number
Document type

Document Status

The document’s status is displayed in a banner at the bottom of each page. This describes the document’s confidentiality and its information status.

Confidentiality status is one of:

ARM Confidential	Distributable to ARM staff and NDA signatories only
Named Partner Confidential	Distributable to the above and to the staff of named partner companies only
Partner Confidential	Distributable within ARM and to staff of all partner companies
Open Access	No restriction on distribution

Information status is one of:

Advance	Information on a potential product
Preliminary	Current information on a product under development
Final	Complete information on a developed product

Change Log

Issue	Date	By	Change
A (Draft)	Mar 1995	EH	Created.
B (Draft)		Aug 1995	SFKEdits, in particular Bus Interface.
C	Oct 1995	AP	Clocking additions.
D	Mar 1996	EH	Appendix F removed. Status changed to Open Access.
E	August 1996	KTB	Corrections and amendments

Contents

1 Overview 1-1

1.1 Introduction 1-2
1.2 The ARM810 Architecture 1-2
1.3 The Instruction Set 1-3

2 Signal Description 2-1

2.1 Functional Diagram 2-2
2.2 Signals 2-3

3 Programmer’s Model 3-1

3.1 Introduction 3-2
3.2 ARM810 Configuration 3-2
3.3 ARM8 CPU Core Configuration 3-3
3.4 Operating Modes 3-4
3.5 Registers 3-5
3.6 Exceptions 3-8
3.7 Reset 3-12

4 Instruction Set 4-1

4.1 Summary 4-2
4.2 Reserved Instructions and Usage Restrictions 4-2
4.3 The Condition Field 4-3
4.4 Branch and Branch with Link (B, BL) 4-5
4.5 Data Processing Instructions 4-7
4.6 PSR Transfer (MRS, MSR) 4-17
4.7 Multiply and Multiply-Accumulate (MUL, MLA) 4-23
4.8 Multiply Long and Multiply-Accumulate Long (MULL, MLAL) 4-25
4.9 Single Data Transfer (LDR, STR) 4-27
4.10 Halfword and Signed Data Transfer 4-34

Contents

4.11 Block Data Transfer (LDM, STM) 4-40
4.12 Single Data Swap (SWP) 4-49
4.13 Software Interrupt (SWI) 4-52
4.14 Coprocessor Data Operations (CDP) 4-55
4.15 Coprocessor Data Transfers (LDC, STC) 4-57
4.16 Coprocessor Register Transfers (MRC, MCR) 4-61
4.17 The Instruction Memory Barrier (IMB) Instruction 4-64
4.18 Undefined Instructions 4-67
4.19 Instruction Set Examples 4-68

5 Configuration 5-1

5.1 ARM810 System Control Coprocessor (CP15) Register Map 5-3

6 The Prefetch Unit 6-1

6.1 Overview 6-2
6.2 The Prefetch Buffer 6-2
6.3 Branch Prediction 6-3

7 Instruction and Data Cache (IDC) 7-1

7.1 Introduction 7-2
7.2 Cacheable Bit and Bufferable Bit 7-2
7.3 IDC Operation 7-2
7.4 IDC Validity 7-2
7.5 Read-Lock-Write 7-3
7.6 IDC Enable/Disable and Reset 7-3
7.7 Lock-down Features 7-3

8 Memory Management Unit 8-1

8.1 MMU Program Accessible Registers 8-3
8.2 Address Translation 8-5
8.3 Translation Process 8-6
8.4 Level One Descriptor 8-7
8.5 Page Table Descriptor 8-8
8.6 Section Descriptor 8-9
8.7 Translating Section References 8-10
8.8 Level Two Descriptor 8-11
8.9 Translating Small Page References 8-12
8.10 Translating Large Page References 8-13
8.11 Cacheable and Bufferable Status of Memory Regions 8-14
8.12 MMU Faults and CPU Aborts 8-16
8.13 Fault Address and Fault Status Registers (FAR and FSR) 8-17
8.14 Domain Access Control 8-19
8.15 Fault Checking Sequence 8-20
8.16 External Aborts 8-23
8.17 Interaction of the MMU, IDC and Write Buffer 8-24
8.18 Effect of Reset 8-25

9 Write Buffer 9-1

9.1 Cacheable and Bufferable bits 9-3
9.2 Write Buffer Operation 9-4

10 Coprocessors 10-1
10.1 Overview 10-2

11 ARM810 Clocking 11-1
11.1 The Bus Clock 11-3
11.2 The Processor Clock 11-4
11.3 Generation of the Fast Clock 11-6
11.4 Forced Processor Clock from the Bus Clock 11-9
11.5 Low Power Idle and Sleep 11-10

12 Bus Interface 12-1
12.1 ARM810 Cycle Speed 12-2
12.2 Bus Interface Signals 12-3
12.3 Cycle Types 12-4
12.4 Addressing Signals 12-8
12.5 Memory Request Signals 12-10
12.6 Data Signal Timing 12-11
12.7 ABORT Signal 12-12
12.8 Maximum Sequential Length 12-13
12.9 Read-Lock-Write 12-14
12.10 Use of the nWAIT Pin 12-16
12.11 Use of the APE Pin 12-18
12.12 Bus Interface Sampling Points 12-19
12.13 Word, Halfword, and Byte Operations 12-22
12.14 Memory Access Sequence Summary 12-29
12.15 ARM810 Cycle Type Summary 12-33

13 Boundary-Scan Test Interface 13-1
13.1 Overview 13-3
13.2 Reset 13-4
13.3 Pullup Resistors 13-5
13.4 Instruction Register 13-6
13.5 Public Instructions 13-7
13.6 Test Data Registers 13-10
13.7 Boundary-Scan Interface Signals 13-13

14 ARM810 DC Parameters 14-1
14.1 Absolute Maximum Ratings 14-2
14.2 DC Operating Conditions 14-2
14.3 Input Thresholds 14-3
14.4 DC Characteristics 14-3

15 ARM810 AC Parameters 15-1
15.1 Test Conditions 15-2
15.2 Clocking 15-3
15.3 Main Bus Signals 15-6

16 Physical Details 16-1
16.1 Physical Details 16-2
16.2 Pinout 16-3

Contents

17 Backward Compatibility **17-1**

17.1 Instruction Memory Barrier 17-2
17.2 Undefined Instructions 17-2
17.3 PC Offset 17-2
17.4 Write-Back 17-2
17.5 Misaligned PC Loads and Stores 17-2
17.6 Data Aborts 17-2

A Use of the ARM810 in a 5V TTL System **A-1**

A.1 Using the ARM810 in a 5V TTL System A-2

B Instruction Set Changes **B-1**

B.1 General Compatibility B-2
B.2 Instruction Set Differences B-2

C 26-bit Operations on ARM810 **C-1**

C.1 Introduction C-2
C.2 Instruction Differences C-3
C.3 Performance Differences C-4
C.4 Hardware Compatibility Issues C-4

D Comparisons with 26-bit ARM Processors **D-1**

D.1 Introduction D-2
D.2 The Program Counter and Program Status Register D-2
D.3 Operating Modes D-2
D.4 Instruction Set Changes D-3
D.5 Transferring between 26-bit and 32-bit Modes D-4

E Implementing the Instruction Memory Barrier Instruction **E-1**

E.1 Introduction E-2
E.2 ARM810 IMB Implementation E-2
E.3 Generic IMB Use E-2

1 Overview

This chapter provides an overview of the ARM810.

1.1 Introduction 1-2

1.2 The ARM810 Architecture 1-2

1.3 The Instruction Set 1-3

Overview

1.1 Introduction

The ARM810 is a general purpose 32-bit microprocessor with 8KB unified cache, write buffer and Memory Management Unit (MMU) combined in a single chip. The CPU within ARM810 is the ARM8 core designed to ARM architecture version 4.0. The ARM810 is software compatible with earlier ARM architectures and can be used with ARM Ltd’s support chips.

The architecture of the ARM810 is based on Reduced Instruction Set Computer (RISC) principles, and its instruction set and related decode mechanism are much simpler than those of microprogrammed Complex Instruction Set Computers.

1.2 The ARM810 Architecture

The block diagram for the ARM810 core is shown in ***Figure 1-1: ARM810 block diagram*** on page 1-3. The major blocks of the ARM810 are the ARM8 CPU, Memory Management Unit (MMU), Write Buffer (WB), Coprocessor15 (CP15), 8KB cache and external Bus Interface Unit (BIU).

The ARM8 CPU core combines an instruction Prefetch Unit (PU) with a four stage instruction and data pipeline to make a CPU with a five stage pipeline. This ensures that all parts of the processing and memory systems can operate continuously and the performance of each stage can be maximised allowing the use of very high clock rates. The processor implements static branch prediction in the PU which predicts whether or not a branch will be taken depending on the branch destination address. If the branch instruction is predicted as being taken then further instructions are fetched from the branch destination and the branch is removed. If it is predicted that a branch is not taken then the instruction is removed from the instruction pipeline.

Double-bandwidth reads to the on-chip memory reduces the average Load multiple CPI value by 1.5 when compared to a single bandwidth architecture running the same code. The single register Load and Store instructions require only one cycle for the normal cases. (This assumes cache hit and write buffer not full.)

The processor has an on-chip unified instruction and data cache supporting write-back and write-through operation. Together with the write buffer, this substantially raises the average execution speed and reduces the average amount of memory bandwidth required by the processor when compared to a processor without these features. This allows the external memory to support additional processors or Direct Memory Access (DMA) channels with minimal performance loss and results in low external memory power consumption. In addition the cache also has a lock down capability that can ensure that critical code such as interrupt routines can be locked into the cache to ensure low execution latency.

The MMU supports a conventional two-level page-table structure and a number of extensions which make it ideal for embedded control, UNIX and Object Oriented systems.

The Bus 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. These control signals allow the exploitation of page mode accesses offered by industry standard DRAMs.

The ARM810 is a fully static device designed to minimise power requirements. This makes it ideal for portable applications where both of these features are essential.

1.3 The Instruction Set

The ARM810 uses the ARM architecture version 4.0 instruction set, which comprises eleven basic instruction types:

- Two perform high-speed operations on data in a bank of thirty one 32-bit registers, using the on-chip arithmetic logic unit, shifter and multiplier.
- Three control data transfer between the registers and memory, one optimised for flexibility of addressing, another for rapid context switching and the third for swapping data.
- Three adjust the flow, privilege level and system control parameters of execution.
- Three are dedicated to the control of coprocessors. However ARM810 implements only MCR and MRC, which allow access to the ARM810 system control coprocessor, CP15.

The ARM instruction set is a good target for compilers of many different high-level languages, and is also straightforward to use when programming in assembly language - unlike the instruction sets of some RISC processors which rely on sophisticated compiler technology to manage complicated instruction interdependencies.

All existing code compiled for previous ARM processors will run on ARM810 with only rare exceptions.

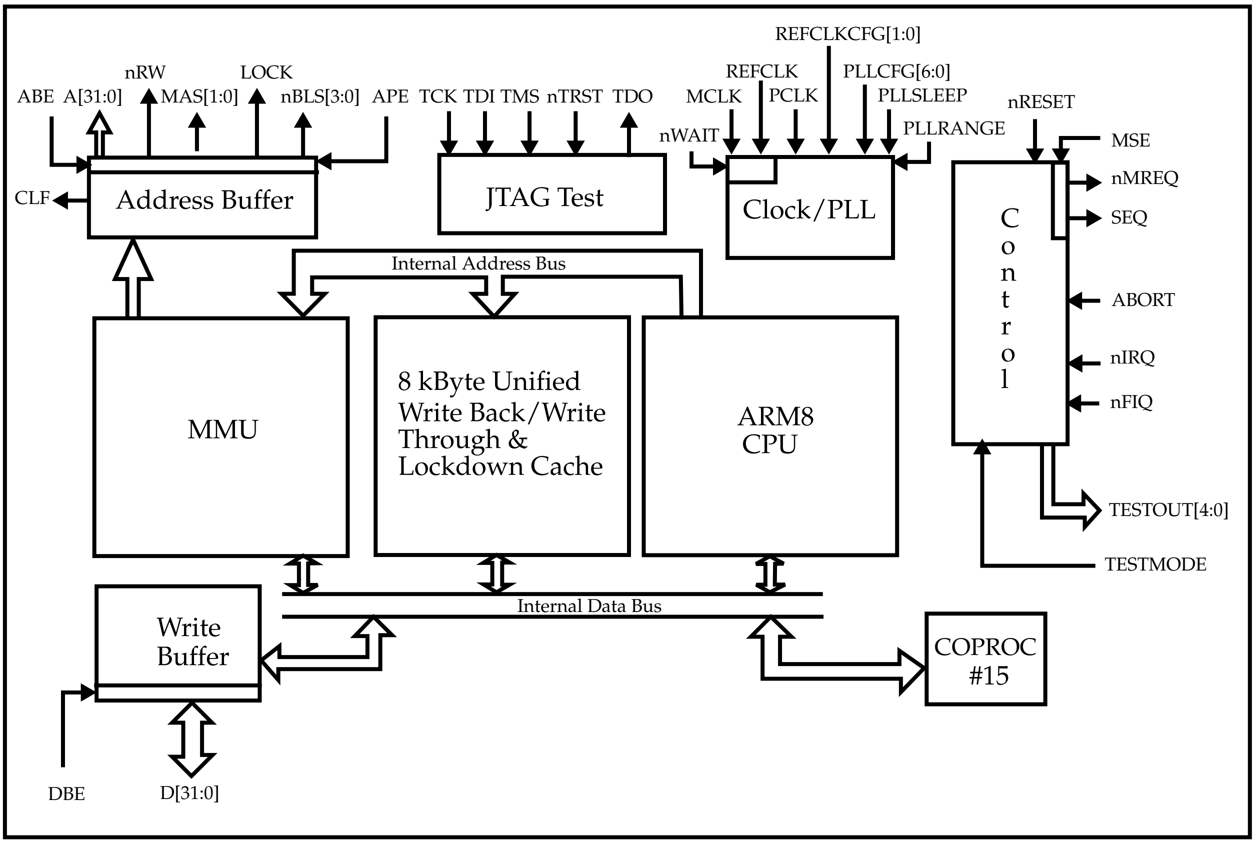


Figure 1-1: ARM810 block diagram

2 Signal Description

This chapter documents the ARM810 signals.

2.1 Functional Diagram 2-2
2.2 Signals 2-3

Signal Description

2.1 Functional Diagram

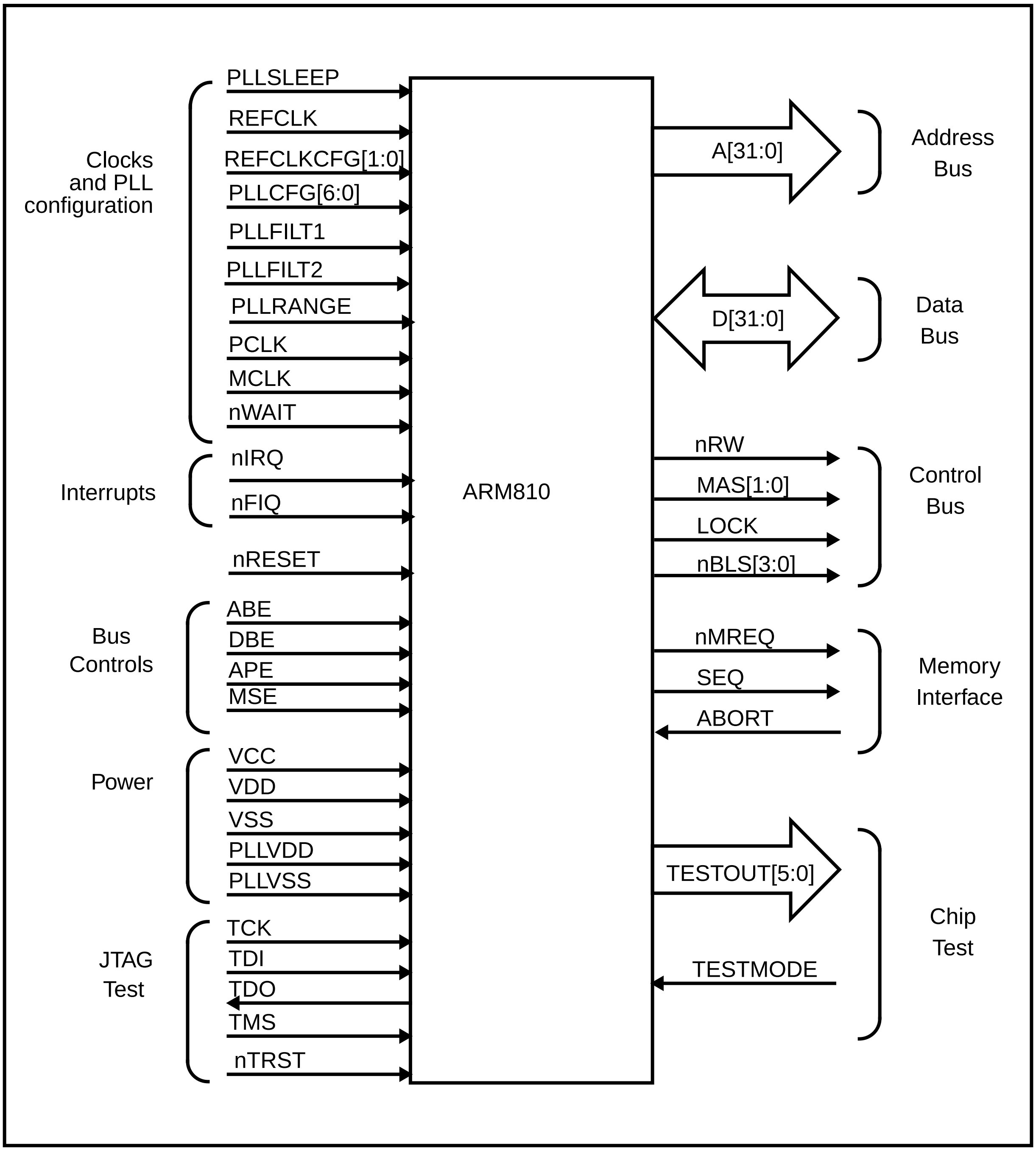


Figure 2-1: ARM810 functional diagram

2.2 Signals

Name	Type	Description:
A[31:0]	OCZ	Address Bus. This bus signals the address requested for memory accesses. Normally it changes during phase 2 of the bus clock. The timing can be changed using **APE**.
ABE	I	Address bus enable. When this input is LOW, the address bus **A[31:0]**, **MAS[1:0]**, **CLF**, **nBLS[3:0]**, **nRW** and **LOCK** are put into a high impedance state (Note 1).
ABORT	I	External abort. Allows the memory system to tell the processor that a requested access has failed. Only monitored when ARM810 is accessing external memory.
APE	I	Address pipeline enable control input. When **APE** is HIGH, address and address-timed outputs are generated with normal pipeliined timing, where a new address is generated in the second phase of the bus clock (**MCLK** HIGH or **PCLK** LOW). Taking **APE** LOW delays these signals by one clock phase so they change in the first phase of the following bus cycle (**MCLK** LOW or **PCLK** HIGH). See the descriptions for **MCLK**/**PCLK** and ***Chapter 11, ARM810 Clocking*** for bus clock information. The address-timed signals are **A[31:0]**, **MAS[1:0]**, **nBLS[3:0**], **CLF**, **LOCK** and **nRW**.
CLF	O	Cache line fill. **CLF** HIGH indicates that the current read cycle is cacheable. **CLF** is always HIGH for writes. This signal may be used to indicate to a second level cache controller that a read is cacheable in the second level cache (if present).
D[31:0]	IOCZ	Data bus. These are bi-directional signal paths used for data transfers between the processor and external memory. For read operations (when **nRW** is LOW), the input data must be valid before the falling edge of **MCLK**. For write operations (when **nRW** is HIGH), the output data will become valid while **MCLK** is LOW. At high clock frequencies the data may not become valid until just after the **MCLK** rising edge.
DBE	I	Data bus enable. When this input is LOW, the data bus, **D[31:0]** is put into a high impedance state (Note 1). The drivers will always be high impedance except during write operations, and **DBE** must be driven HIGH in systems which do not require the data bus for DMA or similar activities.
LOCK	OCZ	Locked operation. **LOCK** is driven HIGH, to signal a “locked” memory access sequence, and the memory manager should wait until **LOCK** goes LOW before allowing another device to access the memory. **LOCK** remains HIGH during the locked memory sequence. Normally it changes during phase 2 of the bus clock. The timing can be changed using **APE**.
MCLK	I	This is a bus clock input. Bus cycles start and end with falling edges of **MCLK**. Hold **PCLK** HIGH to use this clock input. See ***11.1.1 External input clock: MCLK or PCLK*** on page 11-3 for further details. This signal is provided for backwards compatibility with previous processors, see **PCLK** for the preferred bus clock input.

Table 2-1: Signal descriptions

Signal Description

Name	Type	Description:
MSE	I	Memory request/sequential enable. When this input is LOW, the **nMREQ** and **SEQ** outputs are put into a high impedance state (Note 1).
MAS[1:0]	OCZ	Memory Access Size. An output bus used by the processor to indicate the size of the next data transfer to the external memory system as being a byte, half word or full 32 bit word in length. MAS[1:0] is valid for both read and write operations**.** Normally it changes during phase 2 of the bus clock.The timing can be changed using **APE**.
nBLS[3:0]	OCZ	Not Byte Lane Selects. These signify which bytes of the memory are being accessed. For a word access all will be LOW. Normally they change during phase 2 of the bus clock. The timing can be changed using **APE**.
nFIQ	I	Not fast interrupt request. If FIQs are enabled, the processor will respond to a LOW level on this input by taking the FIQ interrupt exception. This is an asynchronous, level-sensitive input to guarantee that the interrupt has been taken.,
nIRQ	I	Not interrupt request. As **nFIQ**, but with lower priority. If IRQs are enabled, the processor will respond to a low level on this signal by taking the IRQ interrupt exception.
nMREQ	OCZ	Not memory request. A pipelined signal that changes while **MCLK** is LOW to indicate whether or not in the following cycle, the processor will be accessing external memory. When **nMREQ** is LOW, the processor will be accessing external memory in the next bus cycle.
nRESET	I	Not reset. This is a level sensitive input which is used to start the processor from a known address. A LOW level will cause the current instruction to terminate abnormally, and the on-chip cache, MMU, and write buffer to be disabled. When **nRESET** is driven HIGH, the processor will re-start from address 0. **nRESET** must remain LOW for at least 5 full fast clock cycles or 5 full bus clock cycles whichever is greater. While **nRESET** is LOW the processor will perform idle cycles and **nWAIT** must be HIGH.
nRW	OCZ	Not read/write. When HIGH this signal indicates a processor write operation; when LOW, a read. Normally it changes during phase 2 of the bus clock. The timing can be changed using **APE**.
nTRST	I	Test interface reset. Note this signal does NOT have an internal pullup resistor. This signal must be pulsed or driven LOW to achieve normal device operation, in addition to the normal device reset (**nRESET**).
nWAIT	I	Not wait. When LOW this allows extra **MCLK** cycles to be inserted in memory accesses. It must change during the LOW phase of the **MCLK** cycle to be extended.

Table 2-1: Signal descriptions (Continued)

Name	Type	Description:
PCLK	I	This is an inverted bus clock input. Bus cycles start and end with rising edges of **PCLK**. Hold **MCLK** LOW to use this clock input. See ***11.1.1 External input clock: MCLK or PCLK*** on page 11-3 for further information. We recommend using this bus clock input for compatibility with the new generations of synchronous memory systems (SSRAM, SDRAM) and future ARM microprocessors. The **MCLK** input is provided for compatibility with earlier ARM processors.
PLLCFG[6:0]	I	Phase locked loop configuration input. Please refer to ***11.3.2 Fast clock from the output of the PLL*** on page 11-7 for further details.
PLLFILT1		Analog filter pin for PLL.
PLLFILT2		Analog filter fast start pin for PLL.
PLLRANGE	IOCZ	In normal operation, an input which selects the **PLL** output frequency range. Please refer to ***11.3.2 Fast clock from the output of the PLL*** on page 11-7 for further details. This pin is also used as an output when the device is in some test modes. The output driver is guaranteed to be high-impedance if the **TESTMODE** pin is LOW.
PLLSLEEP	I	When HIGH, this puts the **PLL** into low power sleep mode. Please refer to ***11.5 Low Power Idle and Sleep*** on page 11-10 for further details.
PLLVDD		VDD supply for analog components in PLL. 1 pin. Should be appropriately isolated from digital noise on supply.
PLLVSS		Ground supply for analog components in PLL. 1 pin.
REFCLK	I	Clock input which is divided by the prescaler to provide the **PLL** reference clock. **REFCLK** can also be configured to a direct source of the internal fast clock, bypassing the **PLL**. Please refer to ***11.3.2 Fast clock from the output of the PLL*** on page 11-7 and ***11.3.3 Fast clock direct (bypassing the PLL)*** on page 11-8 for further details.
REFCLKCFG[1:0]	IOCZ	In normal operation, an input which selects the divide ratio for the **PLL** reference clock prescaler on the **REFCLK** input. Please refer to ***11.3.2 Fast clock from the output of the PLL*** on page 11-7 for further details. These pins are also used as an output when the device is in some test modes. The output drivers are guaranteed to be high-impedance if the **TESTMODE** pin is LOW.
SEQ	OCZ	Sequential address. This signal is the inverse of **nMREQ**, and is provided for compatibility with existing ARM memory systems.
TESTMODE	I	This signal must be tied LOW.
TESTOUT[4:0]	O	This bus should be left unconnected. These outputs will be driven LOW except when device test features are enabled. They will not be tri-stated, except via the JTAG test port.
TCK	I	Test interface reference Clock. This times all the transfers on the JTAG test interface.

Table 2-1: Signal descriptions (Continued)

Name	Type	Description:
TDI	I	Test interface data input. Note this signal does *not* have an internal pullup resistor.
TDO	OCZ	Test interface data output. Note this signal does *not* have an internal pullup resistor.
TMS	I	Test interface mode select. Note this signal does *not* have an internal pullup resistor.
VCC		Pad voltage reference. 1 pin is allocated to VCC. This should be tied to the system power supply, ie. 5V in a TLL system or 3.3V in a 3.3V system. See ***Appendix A, Use of the ARM810 in a 5V TTL System***.
VDD		Positive supply. 15 pins are allocated to **VDD** in the 144 TQFP package.
VSS		Ground supply. 15 pins are allocated to **VSS** in the 144 TQFP package.

Table 2-1: Signal descriptions (Continued)

Notes

1 When output pads are placed in the high impedance state for long periods, care must be taken to ensure that they do not float to an undefined logic level, as this can dissipate power, especially in the pads.

2 The input pads on this device are compatible with both CMOS and TTL signals. The thresholds can be found in ***Chapter 14, ARM810 DC Parameters***.

Key to signal types:

I	Input
OCZ	Output, CMOS levels, tristateable
IOCZ	Input/output tristateable, CMOS levels
ICK	Clock input

3 Programmer’s Model

This chapter describes the programmer’s model.

3.1	Introduction	3-2
3.2	ARM810 Configuration	3-2
3.3	ARM8 CPU Core Configuration	3-3
3.4	Operating Modes	3-4
3.5	Registers	3-5
3.6	Exceptions	3-8
3.7	Reset	3-12

Programmer's Model

3.1 Introduction

The ARM810 supports a variety of operating conditions. Some are controlled by register bits and are known as the register configurations and others are controlled by software which are known as operating modes.

The ARM810 programmers model can be split into two distinct segments. The ARM8 CPU core can be configured to a number of hardware configurations and a number of operating modes. In adidtion the cache, MMU and branch prediction blocks exterior to the ARM8 core can also be configured to operate in different ways by software access to the Coprocessor 15 (CP15) configuration registers. This section concentrates on the programmer's model for the ARM8 CPU core. The details of accessing CP15 for the configuration of the ARM810 blocks external to the ARM8 core can be found in ***Chapter 5, Configuration***.

3.2 ARM810 Configuration

The ARM810 Microprocessor incorporates advanced cache, memory management and branch prediction features.

The ARM810 Cache features are described in detail in ***Chapter 7, Instruction and Data Cache (IDC)***. The Memory Management Unit features are described in ***Chapter 8, Memory Management Unit***. The Write Buffer configuration is decribed in detail in ***Chapter 9, Write Buffer***. The Prefetch Unit features including branch prediction configuration is documented in ***Chapter 6, The Prefetch Unit***.

3.3 ARM8 CPU Core Configuration

The ARM8 CPU core provides 1 register configuration setting which may be changed while the processor is running and which is discussed below. Please refer to ***Chapter 5, Configuration*** for details of how to configure the ARM8.

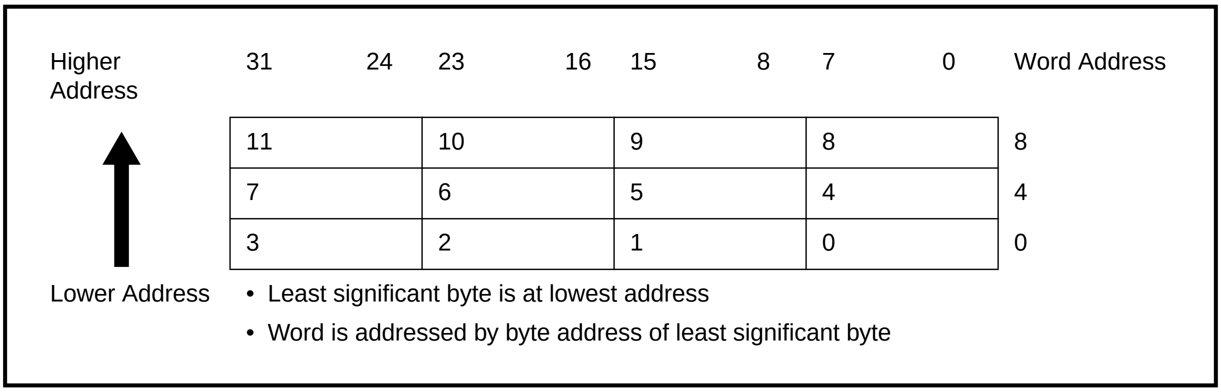
3.3.1 Big and little-endian memory formats (the BIGEND signal)

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. ARM810 can treat words in memory as being stored either in Big Endian or Little Endian format, depending on the state of the **BIGEND** input.

The Load/Store instructions are the only ones affected by the endianness.

Little-endian format

In Little-endian format (**BIGEND** LOW) the lowest numbered byte in a word is considered the least significant byte of the word, and the highest numbered byte the most significant. Byte 0 of the memory system should therefore be connected to data lines 7 through 0.



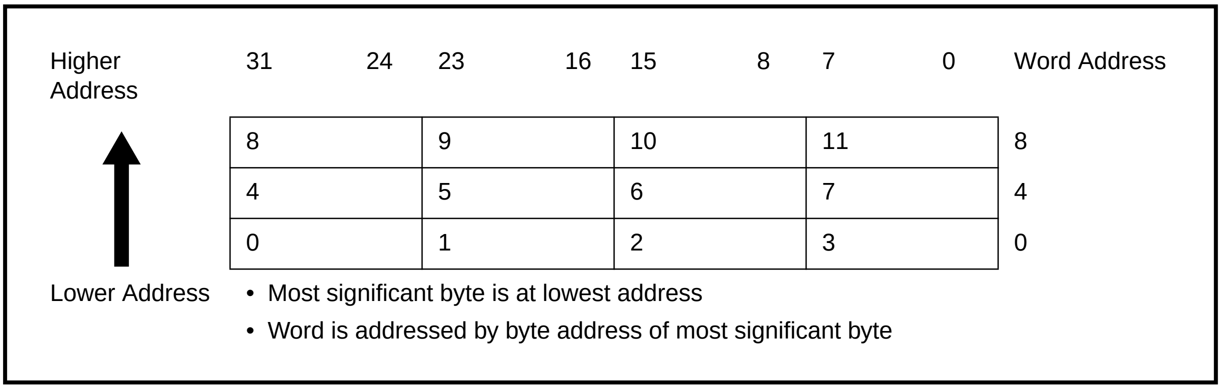
31 24	23 16	15 8	7 0	Word Address
11	10	9	8	8
7	6	5	4	4
3	2	1	0	0

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

Figure 3-1: Little-endian addresses of bytes within words

Big-endian format

In Big-endian format (**BIGEND** HIGH) the most significant byte of a word is stored at the lowest numbered byte and the least significant byte at the highest numbered byte. Byte 0 of the memory system should therefore be connected to data lines 31 through 24.



31 24	23 16	15 8	7 0	Word Address
8	9	10	11	8
4	5	6	7	4
0	1	2	3	0

- Most significant byte is at lowest address
- Word is addressed by byte address of most significant byte

Figure 3-2: Big-endian addresses of bytes within words

3.4 Operating Modes

ARM810 supports *byte* (8-bit), *half-word* (16-bit), and *word* (32-bit) data types. Instructions are exactly one word long, and must be aligned to four-byte boundaries. Data operations, such as ADD, are only performed on word quantities. Load and Store operations are able to transfer bytes, half-words or words.

ARM810 supports seven modes of operation:

Mode	**Description**
User mode (usr)	normal program execution state
FIQ mode (fiq)	used for fast or higher priority interrupt handling
IRQ mode (irq)	used for general-purpose interrupt handling
Supervisor mode (svc)	a protected mode for the operating system
System mode (sys)	a privileged user mode for the operating system
Abort mode (abt)	entered after a data or instruction prefetch abort
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*, are entered in order to service interrupts or exceptions, or to access protected resources.

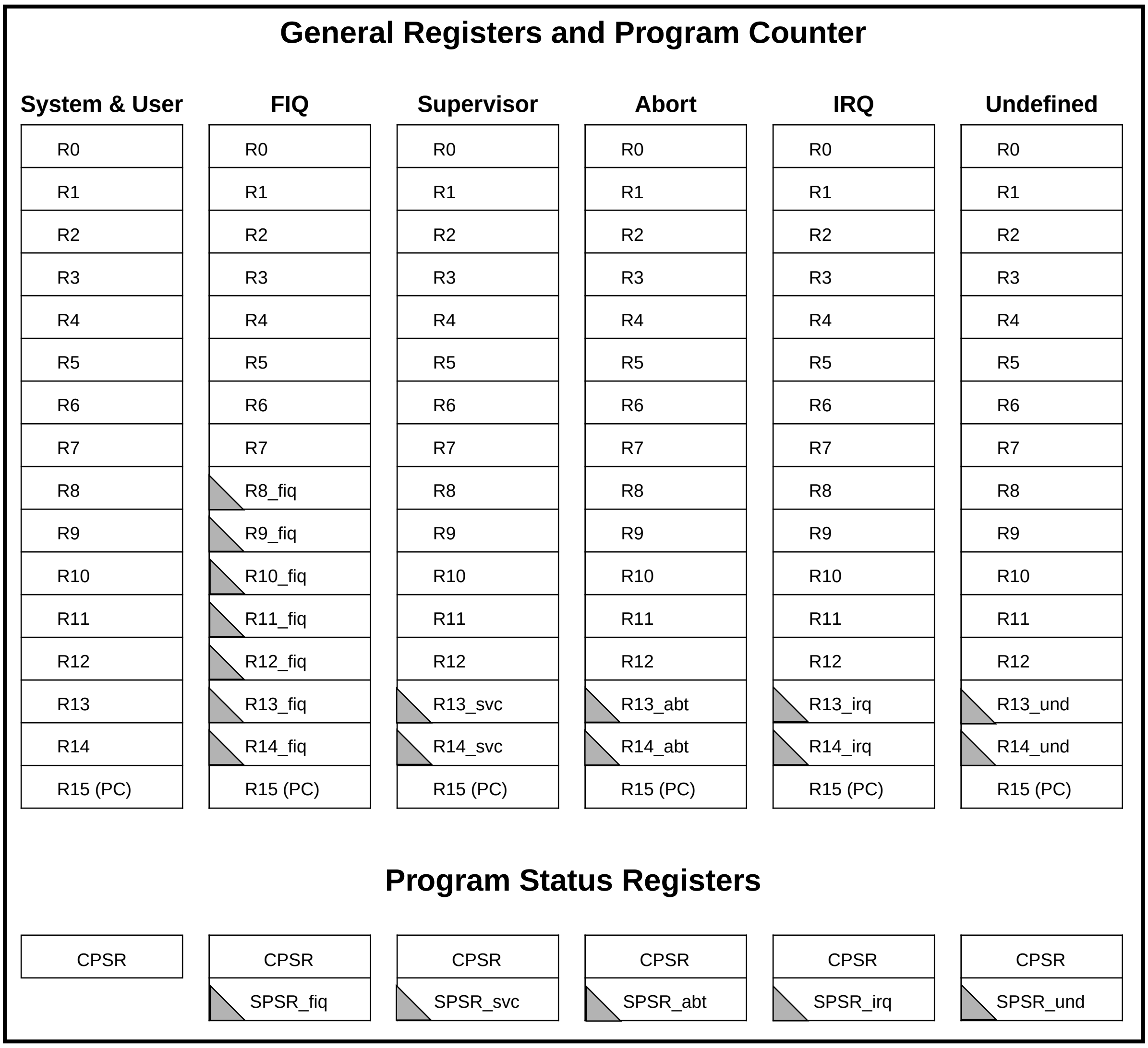
3.5 Registers

ARM8 has a total of 37 registers - 31 general-purpose 32-bit registers and six status registers - but these cannot all be seen at once. The processor mode dictates which registers are available to the programmer. At any one time, 16 general registers and one or two status registers are visible. In privileged (non-User) modes, mode-specific *banked* registers are switched in. ***Figure 3-3: Register organisation*** on page 3-6 shows which registers are available in each processor mode: each of the banked registers is marked with a shaded triangle.

In all modes there are 16 directly accessible registers: R0 to R15. All of these except R15 are general-purpose registers which may be used to hold either data or address values. Register R15 holds the Program Counter (PC). When read, bits [1:0] of R15 are zero and bits [31:2] contain the PC. A seventeenth register, the CPSR (Current Program Status Register), is also accessible. This 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 (LR). This receives a copy of R15 when a Branch and Link (BL) instruction is executed. At all other times it may be treated as a general-purpose register. The corresponding banked registers R14_svc, R14_irq, R14_fiq, R14_abt and R14_und are similarly used to hold the return values of R15 when interrupts and exceptions arise, or when Branch and Link instructions are executed within exception routines.

FIQ mode has seven banked registers mapped to R8-14 (R8_fiq-R14_fiq). Many FIQ handlers will not need to save any registers. User, IRQ, Supervisor, Abort and Undefined each have two banked registers mapped to R13 and R14, allowing each of these modes to have a private stack pointer (SP) and link register (LR).



General Registers and Program Counter

System & User	FIQ	Supervisor	Abort	IRQ	Undefined
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 3-3: Register organisation

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 five SPSRs (Saved Program Status Registers) which are loaded with the CPSR whenever an exception occurs. There is one SPSR for each privileged (non-User) mode, except System mode.

Note No SPSR exists for User or System modes because no exceptions enter these modes. Instructions that attempt to access this SPSR should not be executed in User or System mode.

3.5.1 Program Status Register format

Figure 3-4: Program Status Register (PSR) format shows the format of the Program Status Registers.

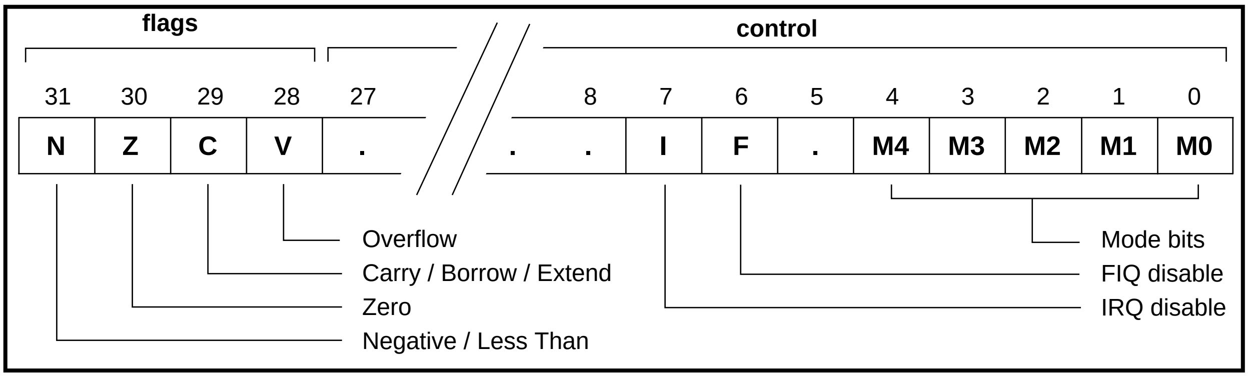


Figure 3-4: Program Status Register (PSR) format

Condition code flags

The N, Z, C and V bits are the *condition code flags*. These may be changed as a result of arithmetic and logical operations in the processor, and may be tested by any instruction to determine whether the instruction is to be executed.

Interrupt disable bits

The I and F bits are the *interrupt disable bits.* When set, these disable the IRQ and FIQ interrupts respectively.

Mode bits

The M4, M3, M2, M1 and M0 bits (M[4:0]) are the *mode bits*. These determine the mode in which the processor operates. The interpretation of the mode bits is shown in ***Table 3-1: The mode bits*** on page 3-8. Not all mode bit combinations define a valid processor mode: you should only use those which are explicitly described.

Control bits

The bottom 8 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. If the processor is operating in a privileged mode, they can also be manipulated by software.

Reserved bits

The remaining bits in the PSRs are *reserved*. When changing a PSR's flag or control bits, you must ensure that these unused bits are not changed by using a read-modify-write scheme. Also, your program should not rely on them containing specific values, since in future processors they may read as one or zero.

M[4:0]	Mode	Accessible Registers	
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
11111	System	PC, R14..R0	CPSR

Table 3-1: The mode bits

3.6 Exceptions

Exceptions arise whenever there is a need for the normal flow of program execution to be broken, so that the processor can be diverted to handle an interrupt from a peripheral, for example. The processor state immediately prior to handling the exception must be preserved, to ensure that the original program can be resumed when the exception routine has completed. It is possible for more than one exception to arise at the same time.

When handling an exception, ARM810 makes 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 the CPSR mode bits are forced to a value which depends on the exception. Where necessary, the interrupt disable flags are set to prevent otherwise unmanageable nestings of exceptions; this is detailed in the following sections.

In the case of a re-entrant interrupt handler, R14 and the SPSR should be saved onto a stack in main memory before the interrupt is re-enabled. 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: see ***3.6.7 Exception priorities*** on page 3-11 for more information.

3.6.1 FIQ

The FIQ (fast interrupt request) exception is externally generated by taking the **nFIQ** input LOW. This input can accept asynchronous transitions because the ARM will always perform the synchronisation. This synchronisation delays the effect of the input transition on the processor execution flow for one cycle.

FIQ is designed to support a fast or high priority interrupt, 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 CPSR’s F flag (but note that this is not possible from User mode). If the F flag is clear, ARM810 checks for a LOW level on the FIQ logic output at the end of each instruction (including cancelled ones), and at the end of any coprocessor busy-wait cycle (allowing the busy-wait state to be interrupted).

On detecting a FIQ, ARM810:

- saves the address of the next instruction to be executed plus 4 in R14_fiq
- saves the CPSR in SPSR_fiq
- forces M[4:0]=10001 (FIQ mode) and sets the F and I bits in the CPSR
- forces the PC to fetch the next instruction from the FIQ vector

To return normally from FIQ, use `SUBS PC,R14_fiq,#4`. This restores both the PC (from R14) and the CPSR (from SPSR_fiq), and resumes execution of the interrupted code.

3.6.2 IRQ

The IRQ (interrupt request) exception is externally generated by taking the **nIRQ** input LOW. This input can accept asynchronous transitions because the ARM will always perform the synchronisation. This synchronisation delays the effect of the input transition on the processor execution flow for one cycle.

IRQ has a lower priority that FIQ and is automatically masked out when a FIQ sequence is entered. The IRQ exception may be disabled by setting the CPSR's I flag (but note that this is not possible from User mode). If the I flag is clear, ARM810 checks for a LOW level on the IRQ logic output at the end of each instruction (including cancelled ones) and at the end of any coprocessor busy-wait cycle (allowing the busy-wait state to be interrupted).

On detecting an IRQ, ARM8:

- saves the address of the next instruction to be executed plus 4 in R14_irq
- saves the CPSR in SPSR_irq
- forces M[4:0]=10010 (IRQ mode) and sets the I bit in the CPSR
- forces the PC to fetch the next instruction from the IRQ vector

To return normally from IRQ, use `SUBS PC,R14_irq,#4`. This restores both the PC (from R14) and the CPSR (from SPSR_irq), and resumes execution of the interrupted code.

3.6.3 Aborts

Not all requests to the Memory System for Data or Instructions will result in a successful completion of the transaction. Such transactions result in an Abort. The rest of this section describes sources and types of aborts, and what happens once they occur.

Abort Sources

Aborts can be generated by Store instructions (STR, STM, SWP (for the write part)), by Data Read instructions (LDR, LDM, SWP (for the read part)) and by Instruction Prefetching. Please refer to ***8.12 MMU Faults and CPU Aborts*** on page 8-16 and ***8.16 External Aborts*** on page 8-23 for further details of Aborts.

Abort types

Aborts are classified as either Prefetch or Data Abort types depending upon the transaction taking place at the time. Each type has its own exception vector to allow branching to the relevant service routine to deal with them. These exception vectors are the Prefetch Abort Vector and the Data Abort Vector and their locations are summarised in ***3.6.6 Exception vector summary*** on page 3-11.

Prefetch Aborts

If the transaction taking place when the abort happened was an Instruction fetch, then a Prefetch Abort is indicated. The instruction is marked as invalid, but the abort exception vector is not taken immediately. Only if the instruction is about to get executed will the Prefetch Abort exception vector be taken.

For Prefetch Aborts, ARM8:

1. Saves the address of the aborted instruction plus 4 into R14_abt
2. Saves the CPSR into SPSR_abt

3. Forces M[4:0] to 10111 (Abort Mode) and sets the I bit in the CPSR
4. Forces the PC to fetch the next instruction from the Prefetch Abort vector.

Returning from a Prefetch Abort: After fixing the reason for the Prefetch Abort, use:

```
SUBS PC,R14_abt,#4
```

This restores both the PC (from R14) and the CPSR (from SPSR_abt), and retries the instruction.

Data aborts

If the transaction taking place when the abort happened was a Data Access (Read or Write), then a Data Abort is indicated, and the action depends upon the instruction type that caused it. In ALL cases, any base register is restored to the value it had before the instruction started whether or not writeback is specified. In addition:

- The LDR instruction does not overwrite the destination register.
- The SWP Instruction is aborted as though it had not executed, although externally the read access may have taken place.
- The LDM Instruction ensures that the PC is not overwritten and will restore the base register such that the instruction can be restarted. All registers up to the aborting one may have been overwritten, but no further ones will be.
- The STM Instruction will ensure that the base register is restored, and any stores up to the aborting one will have already been made - the details depending upon the Memory System itself.

For Data Aborts, ARM8:

1. Saves the address of the instruction which caused the abort plus 8 into R14_abt
2. Saves the CPSR into SPSR_abt
3. Forces M[4:0] to 10111 (Abort Mode) and sets the I bit in the CPSR
4. Forces the PC to fetch the next instruction from the Data Abort vector.

Returning from a Data Abort: After fixing the reason for the Data Abort, use:

```
SUBS PC,R14_abt,#8
```

This restores both the PC (from R14) and the CPSR (from SPSR_abt), and retries the instruction. Note that in the case of LDM, some registers may be re-loaded.

3.6.4 Software interrupt

The software interrupt instruction (SWI) is used for entering Supervisor mode, usually to request a particular supervisor function. When a SWI is executed, ARM810:

- saves the address of the SWI instruction plus 4 in R14_svc
- saves the CPSR in SPSR_svc
- forces M[4:0]=10011 (Supervisor mode) and sets the I bit in the CPSR
- forces the PC to fetch the next instruction from the SWI vector

To return from a SWI, use `MOVS PC,R14_svc`. This restores the PC (from R14) and CPSR (from SPSR_svc), and returns to the instruction following the SWI.

3.6.5 Undefined instruction trap

When the ARM810 decodes an instruction bit-pattern that it cannot process, it takes the undefined instruction trap.

Note Not all non-instruction bit patterns are detected, but such bit patterns will not halt or corrupt the processor and its state.

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

When ARM810 takes the undefined instruction trap, it:

- saves the address of the Undefined instruction plus 4 in R14_und
- saves the CPSR in SPSR_und
- forces M[4:0]=11011 (Undefined mode) and sets the I bit in the CPSR
- forces the PC to fetch the next instruction from the Undefined vector

To return from this trap after servicing or emulating the trapped instruction, use `MOVS PC,R14_und`. This restores the PC (from R14) and the CPSR (from SPSR_und) and returns to the instruction following the undefined instruction.

3.6.6 Exception 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 3-2: Exception vectors

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

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

3.6.7 Exception priorities

When multiple exceptions arise at the same time, a fixed priority system determines the order in which they are handled.

1. Reset (highest priority)
2. Data Abort
3. FIQ
4. IRQ
5. Prefetch Abort
6. Undefined Instruction, Software interrupt (lowest priority)

Not all of the 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 (ie the CPSR’s F flag is clear), ARM810 enters the data abort handler and then immediately proceeds 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.

3.7 Reset

nRESET can be asserted asynchronously. When the **nRESET** signal goes LOW, ARM810 abandons the executing instruction. When **nRESET** goes HIGH again, ARM810:

- overwrites R14_svc and SPSR_svc (by copying the current values of the PC and CPSR into them) with undefined values.
- forces M[4:0]=10011 (Supervisor mode) and sets the I and F bits in the CPSR
- forces the PC to fetch the next instruction from the Reset vector

4 Instruction Set

This chapter details the ARM810 instruction set.

4.1 Summary 4-2
4.2 Reserved Instructions and Usage Restrictions 4-2
4.3 The Condition Field 4-3
4.4 Branch and Branch with Link (B, BL) 4-5
4.5 Data Processing Instructions 4-7
4.6 PSR Transfer (MRS, MSR) 4-17
4.7 Multiply and Multiply-Accumulate (MUL, MLA) 4-23
4.8 Multiply Long and Multiply-Accumulate Long (MULL, MLAL) 4-25
4.9 Single Data Transfer (LDR, STR) 4-27
4.10 Halfword and Signed Data Transfer 4-34
4.11 Block Data Transfer (LDM, STM) 4-40
4.12 Single Data Swap (SWP) 4-49
4.13 Software Interrupt (SWI) 4-52
4.14 Coprocessor Data Operations (CDP) 4-55
4.15 Coprocessor Data Transfers (LDC, STC) 4-57
4.16 Coprocessor Register Transfers (MRC, MCR) 4-61
4.17 The Instruction Memory Barrier (IMB) Instruction 4-64
4.18 Undefined Instructions 4-67
4.19 Instruction Set Examples 4-68

Instruction Set

4.1 Summary

The ARM810 instruction set is summarized below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Cond				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	0	1	U	A	S	RdHi				RdLo				Rn				1	0	0	1	Rm				***Multiply Long***
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	0	0	P	U	0	W	L	Rn				Rd				0	0	0	0	1	S	H	1	Rm				***Halfword Data Transfer: register offset***
Cond				0	0	0	P	U	1	W	L	Rn				Rd				Offset				1	S	H	1	Offset				***Halfword Data Transfer: immediate offset***
Cond				0	1	I	P	U	B	W	L	Rn				Rd				Offset												***Single Data Transfer***
Cond				0	1	1																					1					***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								***Coprocessor Data Transfer***
Cond				1	1	1	0	CP Opc				CRn				CRd				CP#				CP			0	CRm				***Coprocessor Data Operation***
Cond				1	1	1	0	CP Opc			L	CRn				Rd				CP#				CP			1	CRm				***Coprocessor Register Transfer***
Cond				1	1	1	1	Ignored by processor																								***Software Interrupt***
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

Figure 4-1: ARM8 instruction set

Note The instruction cycle times given in this section assume that there is no register interlocking.

4.2 Reserved Instructions and Usage Restrictions

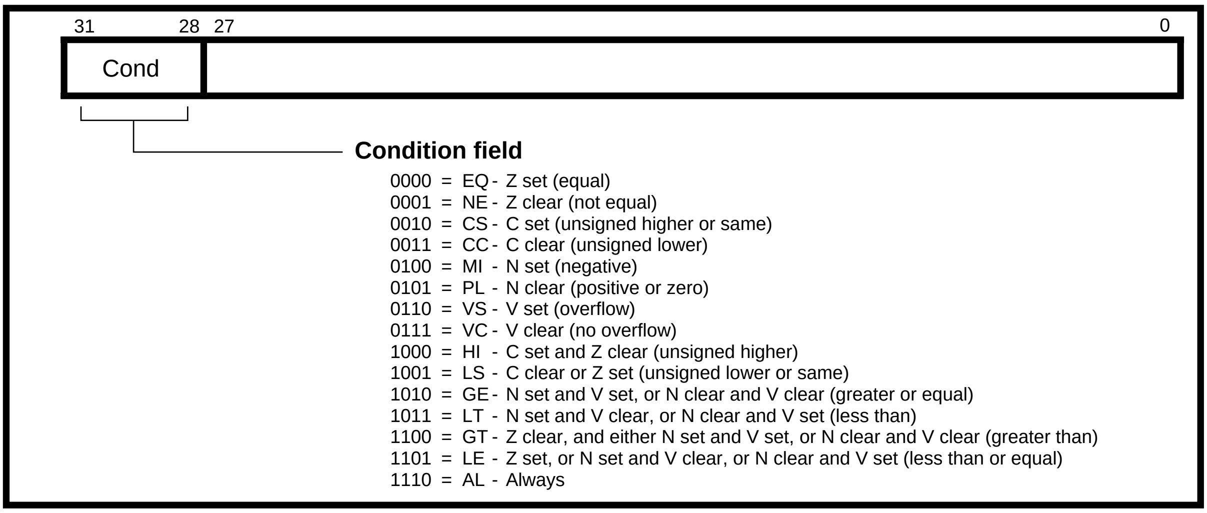
ARM810 enters an Undefined Instruction trap if it encounters an instruction bit pattern that it does not recognize. However, there are some bit patterns which are not defined, but which do not cause the Undefined Instruction trap to be taken. These *reserved* instructions must not be used, as their action may change in future ARM implementations, and may differ from previous ARM implementations.

In addition, this datasheet states that some plausible instruction usages must not be used - particular register combinations for example. In all cases where this is so, should the rules be broken, the processor will not halt or become damaged in any way, though its internal state may well be changed.

Please refer to ***4.18 Undefined Instructions*** on page 4-67 for details of which instruction bit patterns fall into the Undefined Instruction trap.

4.3 The Condition Field

All ARM810 instructions are conditionally executed. This 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. ***Figure 4-2: Condition codes*** shows the condition encoding.



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

Figure 4-2: Condition codes

<table>
<tr><th>31 28</th><th>27 26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19 16</th><th>15 12</th><th>11 8</th><th>7 5</th><th>4</th><th>3 0</th><th></th></tr>
<tr><td>Cond</td><td>0 0</td><td>I</td><td colspan="4">Opcode</td><td>S</td><td>Rn</td><td>Rd</td><td colspan="4">Operand 2</td><td><b>Data Processing</b>
<b>PSR Transfer</b></td></tr>
<tr><td>Cond</td><td colspan="5">0 0 0 0 0 0</td><td>A</td><td>S</td><td>Rd</td><td>Rn</td><td>Rs</td><td colspan="2">1 0 0 1</td><td>Rm</td><td><b>Multiply</b></td></tr>
<tr><td>Cond</td><td colspan="4">0 0 0 1 0</td><td>B</td><td colspan="2">0 0</td><td>Rn</td><td>Rd</td><td>0 0 0 0</td><td colspan="2">1 0 0 1</td><td>Rm</td><td><b>Single Data Swap</b></td></tr>
<tr><td>Cond</td><td>0 1</td><td>I</td><td>P</td><td>U</td><td>B</td><td>W</td><td>L</td><td>Rn</td><td>Rd</td><td colspan="4">offset</td><td><b>Single Data Transfer</b></td></tr>
<tr><td>Cond</td><td colspan="2">0 1 1</td><td colspan="9">XXXXXXXXXXXXXXXXXXXX</td><td>1</td><td>XXXX</td><td><b>Undefined</b></td></tr>
<tr><td>Cond</td><td colspan="2">1 0 0</td><td>P</td><td>U</td><td>S</td><td>W</td><td>L</td><td>Rn</td><td colspan="5">Register List</td><td><b>Block Data Transfer</b></td></tr>
<tr><td>Cond</td><td colspan="2">1 0 1</td><td>L</td><td colspan="10">offset</td><td><b>Branch</b></td></tr>
<tr><td>Cond</td><td colspan="2">1 1 0</td><td>P</td><td>U</td><td>N</td><td>W</td><td>L</td><td>Rn</td><td>CRd</td><td>CP#</td><td colspan="3">offset</td><td><b>Coproc Data Transfer</b></td></tr>
<tr><td>Cond</td><td colspan="3">1 1 1 0</td><td colspan="4">CP Opc</td><td>CRn</td><td>CRd</td><td>CP#</td><td>CP</td><td>0</td><td>CRm</td><td><b>Coproc Data Operation</b></td></tr>
<tr><td>Cond</td><td colspan="3">1 1 1 0</td><td colspan="3">CP Opc</td><td>L</td><td>CRn</td><td>Rd</td><td>CP#</td><td>CP</td><td>1</td><td>CRm</td><td><b>Coproc Register Transfer</b></td></tr>
<tr><td>Cond</td><td colspan="3">1 1 1 1</td><td colspan="10">ignored by processor</td><td><b>Software Interrupt</b></td></tr>
</table>

If the *always* (AL) condition is specified in an instruction, the instruction will be executed regardless of the CPSR flags.

Note: A condition field of 1111 is reserved and should not be used. Instructions with such a condition field may be redefined in future variants of the ARM architecture.

The assembler treats the absence of a condition code qualifier as though AL had been specified. If you require a NOP, use `MOV R0,R0`.

The other condition codes have meanings as detailed in ***Figure 4-2: Condition codes***. For example, code 0000 (EQual) causes an instruction to be executed only if the Z flag is set. This corresponds to the case in which a compare (CMP) instruction has found its two operands to be equal. If the two operands are different, the compare will have cleared the Z flag, and the instruction will not be executed.

4.4 Branch and Branch with Link (B, BL)

A Branch instruction is only executed if the specified condition is true: the various conditions are defined at the beginning of this chapter. ***Figure 4-3: Branch instructions*** shows the instruction encoding.

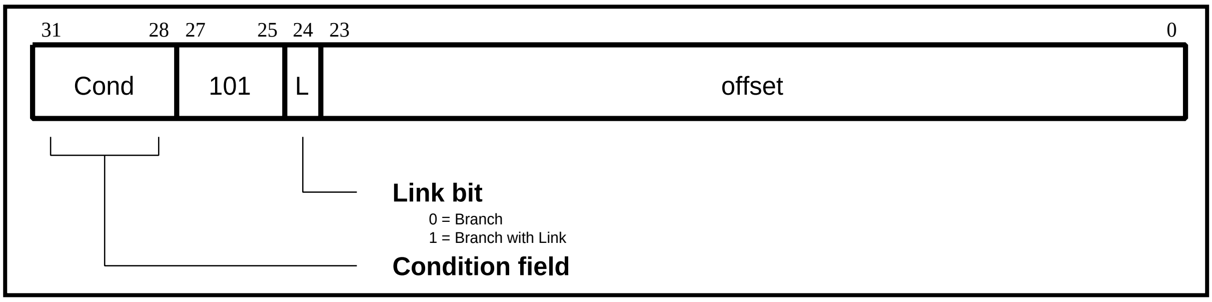


Figure 4-3: Branch instructions

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

Branches beyond +/- 32MB must use an offset or an absolute destination that has been previously loaded into a register. For Branch with Link operations that exceed 32MB, the PC must be saved manually into R14 and the offset added to the PC, or the absolute destination moved to the PC.

4.4.1 The link bit

Branch with Link (BL) writes the old PC into the link register (R14) of the current bank. In the process, 4 is subtracted from the PC value, so that R14 will contain the address of the instruction immediately following the BL instruction. 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.

4.4.2 Branch prediction and removal

The ARM8 Prefetch Unit will attempt to remove a Branch instruction before it reaches the Core. If a Branch is predictable and predicted taken, the Prefetch Unit will start prefetching from the target address, so removing the Branch altogether if predicted correctly. For more information, refer to ***Chapter 6, The Prefetch Unit***.

4.4.3 Instruction cycle times

Note that the cycle times given here are given for the ARM8 processor core, and do not give any information about the additional cycles that may be taken as a result of Cache Misses, MMU Page table walks etc. Future versions of the ARM810 Datasheet will provide such information.Please refer to ***Chapter 12, Bus Interface*** *for timing details of off-chip accesses.*

A Branch (B) or Branch with Link (BL) instruction takes 3 cycles. If optimised by the Prefetch Unit, a Branch will take fewer cycles—possibly 0—and a Branch with Link will take a minimum of 1 cycle if taken, and 0 cycles if not taken.

4.4.4 Assembler syntax

Branch instructions have the following syntax:

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

where

`{L}` requests a Branch with Link.

`{cond}` is one of the two-character mnemonics, shown in ***Figure 4-2: Condition codes*** on page 4-3. The assembler assumes AL (ALways) if no condition is specified.

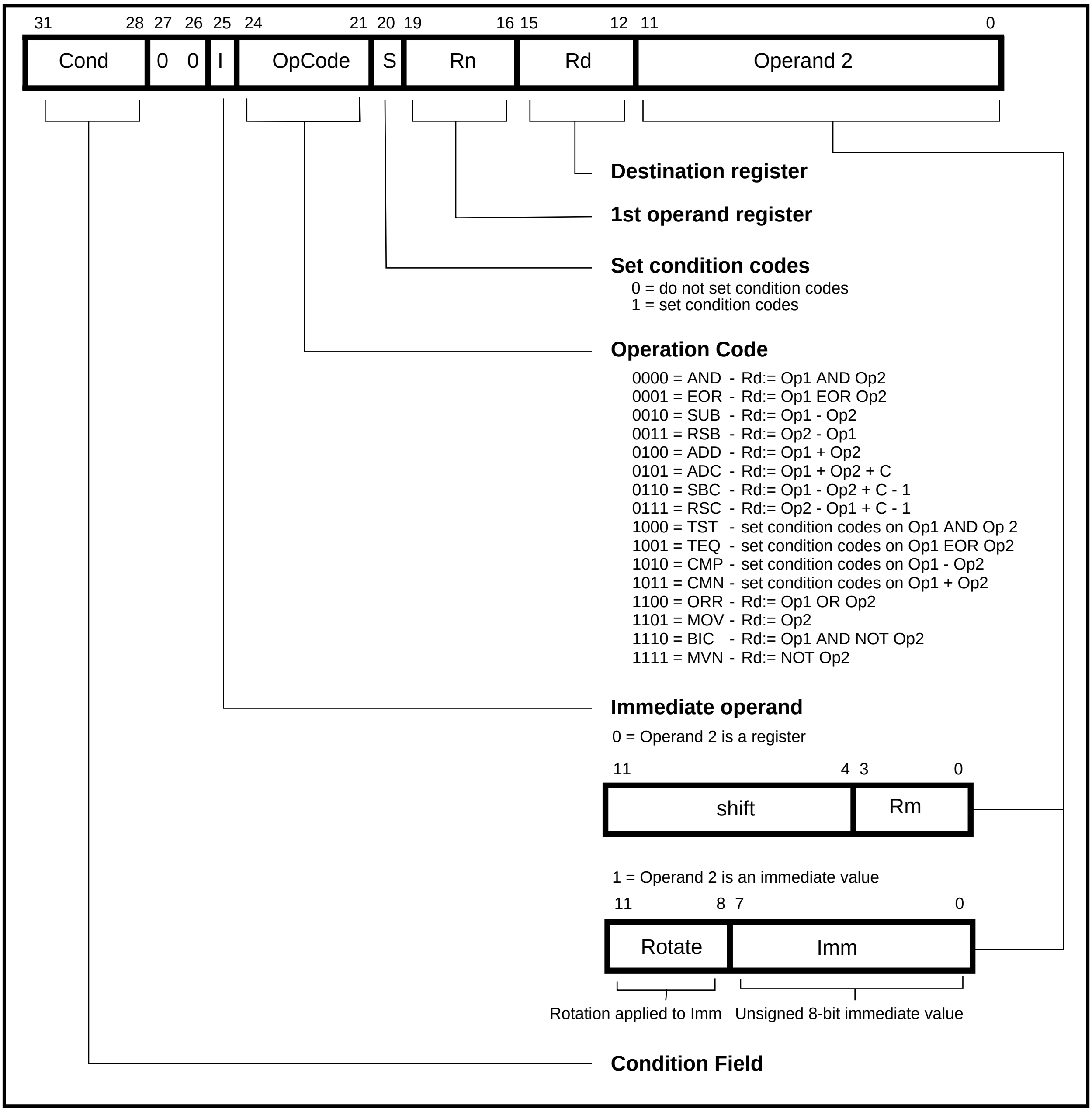
`<expression>` is the destination address. The assembler calculates the offset, taking into account that the PC is 8 ahead of the current instruction.

4.4.5 Examples

```
hereBAL   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
   BEQ    fred      ; if R1 was zero, otherwise continue to next
                    ; instruction

   BL     sub+ROM   ; call subroutine at address computed by
                    ; Assembler

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

4.5 Data Processing Instructions

A data processing instruction is only executed if the specified condition is true: the various conditions are defined at the beginning of this chapter. ***Figure 4-4: Data processing instructions*** shows the instruction encoding.



Destination register

1st operand register

Set condition codes

0 = do not set condition codes
1 = set condition codes

Operation Code

0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1
0100 = ADD - Rd:= Op1 + Op2
0101 = ADC - Rd:= Op1 + Op2 + C
0110 = SBC - Rd:= Op1 - Op2 + C - 1
0111 = RSC - Rd:= Op2 - Op1 + C - 1
1000 = TST - set condition codes on Op1 AND Op 2
1001 = TEQ - set condition codes on Op1 EOR Op2
1010 = CMP - set condition codes on Op1 - Op2
1011 = CMN - set condition codes on Op1 + Op2
1100 = ORR - Rd:= Op1 OR Op2
1101 = MOV - Rd:= Op2
1110 = BIC - Rd:= Op1 AND NOT Op2
1111 = MVN - Rd:= NOT Op2

Immediate operand

0 = Operand 2 is a register

1 = Operand 2 is an immediate value

Rotation applied to Imm
Unsigned 8-bit immediate value

Condition Field

Figure 4-4: Data processing instructions

The instructions in this class produce a result by performing a specified operation on one or two operands, where:

- 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) depending on the value of the instruction’s I bit.

The CPSR flags may be preserved or updated as a result of this instruction, depending on the value of the instruction’s S bit.

Certain operations (TST, TEQ, CMP, CMN) do not write the result to Rd. They are used only to perform tests and to set the CPSR flags on the result, and therefore always have the S bit set.

The data processing instructions and their effects are listed in ***Table 4-1: ARM data processing instructions***.

Assembler mnemonic	OpCode	Action	Note
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	Rd is ignored and should be 0x0000
TEQ	1001	as EOR, but result is not written	Rd is ignored and should be 0x0000
CMP	1010	as SUB, but result is not written	Rd is ignored and should be 0x0000
CMN	1011	as ADD, but result is not written	Rd is ignored and should be 0x0000
ORR	1100	operand1 OR operand2	
MOV	1101	operand2	Rn is ignored and should be 0x0000
BIC	1110	operand1 AND NOT operand2	Bit clear
MVN	1111	NOT operand2	Rn is ignored and should be 0x0000

Table 4-1: ARM data processing instructions

4.5.1 Effects on CPSR flags

Data processing operations are classified as *logical* or *arithmetic*.

Logical operations

The logical operations (AND, EOR, TST, TEQ, ORR, MOV, BIC, MVN) perform the logical action on all the corresponding bits of the operand or operands to produce the result.

If the S bit is set (and Rd is not R15 - see below), they affect the CPSR flags as follows:

N is set to the logical value of bit 31 of the result.

Z is set if and only if the result is all zeros.

C is set to the carry out from the shifter (so is unchanged when no shift operation occurs - see ***4.5.2 Shifts*** and ***4.5.3 Immediate operand rotates*** for the exact details of this).

V is preserved.

Arithmetic operations

The arithmetic operations (SUB, RSB, ADD, ADC, SBC, RSC, CMP, CMN) treat each operand as a 32-bit integer (either unsigned or two's complement signed).

If the S bit is set (and Rd is not R15), they affect the CPSR flags as follows:

N is set to the value of bit 31 of the result. This indicates a negative result if the operands are being treated as 2's complement signed.

Z is set if and only if the result is zero.

C is set to the carry out of bit 31 of the ALU.

V is set if a signed overflow occurs into bit 31 of the result. This can be ignored if the operands are considered as unsigned, but warns of a possible error if they are being treated as 2's complement signed.

4.5.2 Shifts

When the second operand is a shifted register, the instruction’s Shift field controls the operation of the shifter. This 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 either 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 4-5: ARM shift operations***.

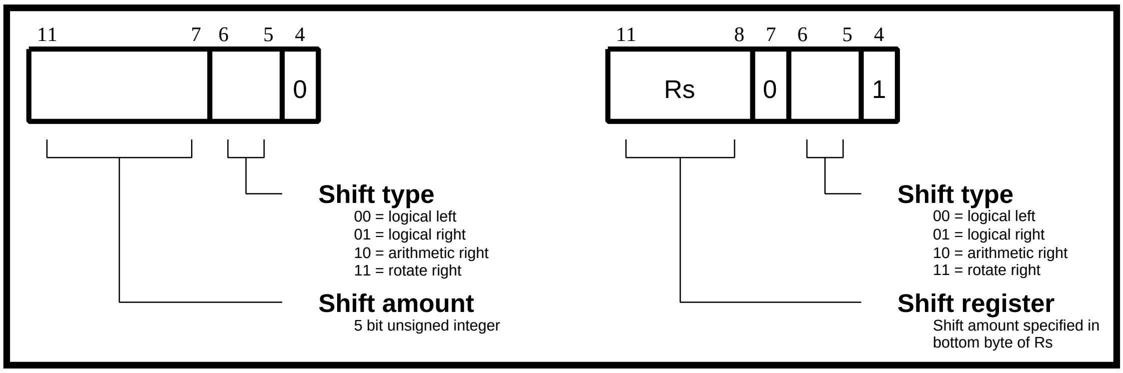


Figure 4-5: ARM shift operations

Instruction-specified shifts

When specified in the instruction, the shift amount 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 to a more significant position by the specified amount. The least significant bits of the result are filled with zeros, and the high bits of Rm that do not map into the result are discarded, with the exception of the least significant discarded bit. This 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 ***Logical operations*** on page 4-9).

As an example, ***Figure 4-6: Logical shift left*** shows the effect of LSL #5.

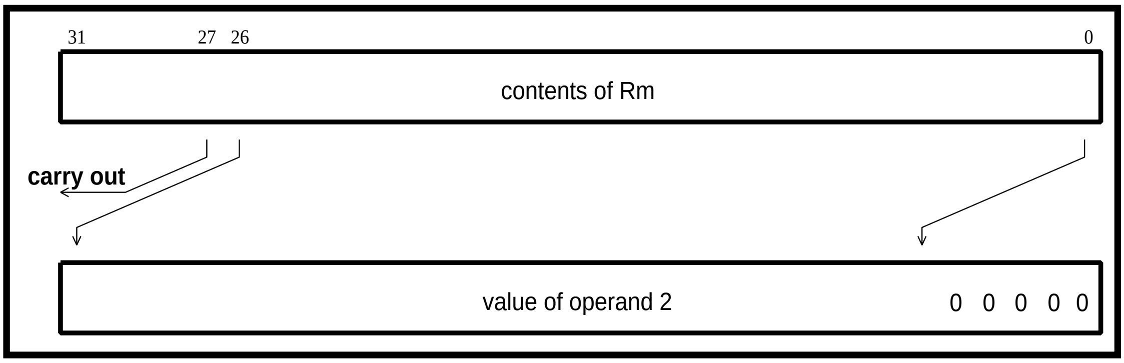


Figure 4-6: Logical shift left

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.

Logical shift right: A logical shift right (LSR) is similar, but the contents of Rm are moved to less significant positions in the result. For example, LSR #5 has the effect shown in ***Figure 4-7: Logical shift right***.

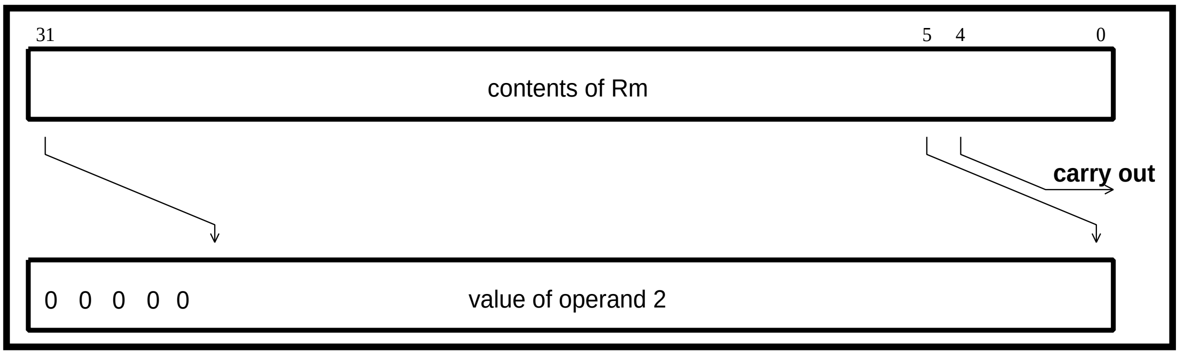


Figure 4-7: Logical shift right

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 converts LSR #0 (as well as ASR #0 and ROR #0) into LSL #0, and allows LSR #32 to be specified.

Arithmetic shift right: An arithmetic shift right (ASR) is similar to a logical shift right, except that the high bits are filled with bit 31 of Rm instead of zeros. This preserves the sign in two's complement notation. ***Figure 4-8: Arithmetic shift right*** on page 4-11 shows the effect of ASR #5.

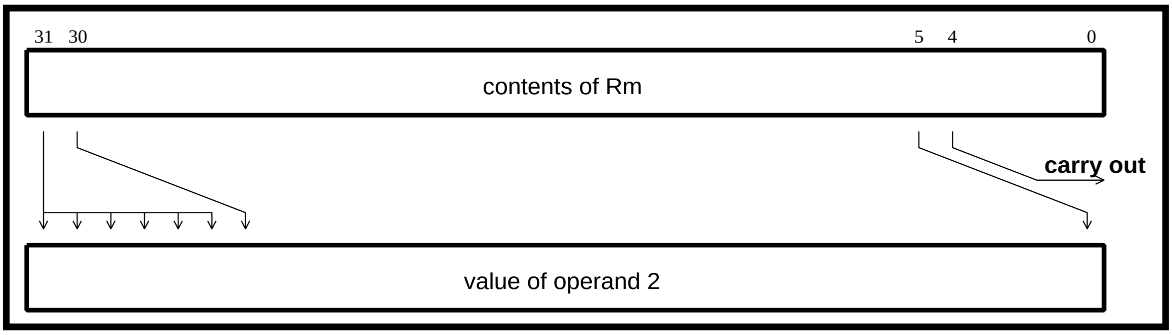


Figure 4-8: Arithmetic shift right

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, depending on the value of bit 31 of Rm.

Rotate right: Rotate right (ROR) operations re-use 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 shift right operations. To illustrate this, the effect of ROR #5 is shown in ***Figure 4-9: Rotate right***.

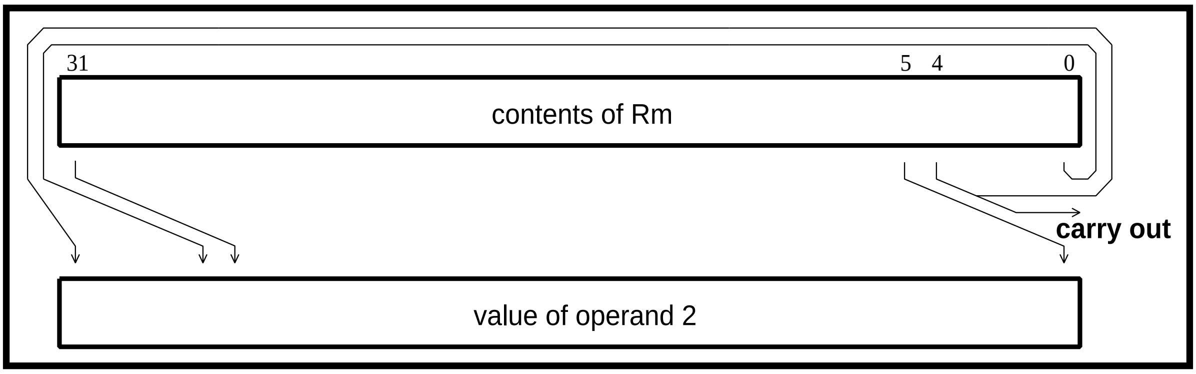


Figure 4-9: Rotate right

The form of the shift field which might be expected to give ROR #0 is used to encode a special function of the 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 4-10: Rotate right extended***.

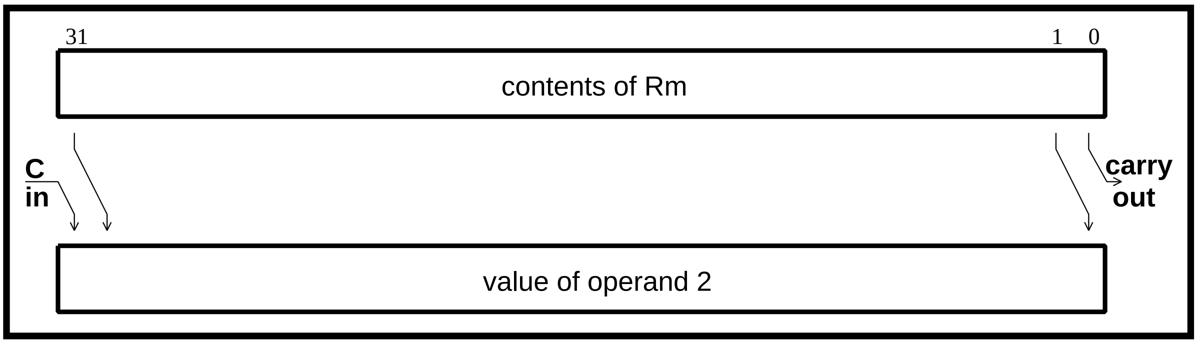


Figure 4-10: Rotate right extended

Register-specified shifts

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

Byte value Description

0 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

1- 31 the shifted result will exactly match that of an instruction specified shift with the same value and shift operation

32 the result will be a logical extension of the shift described above:

- LSL by 32 has result zero, carry out equal to bit 0 of Rm.
- LSL by more than 32 has result zero, carry out zero.
- LSR by 32 has result zero, carry out equal to bit 31 of Rm.
- LSR by more than 32 has result zero, carry out zero.
- ASR by 32 or more has result filled with and carry out equal to bit 31 of Rm.
- ROR by 32 has result equal to Rm, carry out equal to bit 31 of Rm.
- 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

Note Bit 7 of an instruction with a register-controlled shift must be 0: a 1 in this bit will cause the instruction to be something other than a data processing instruction.

4.5.3 Immediate operand rotates

An immediate operand is constructed by taking the 8-bit immediate in the Imm field, zero-extending it to 32 bits, and rotating it by twice the value in the Rotate field. This enables many common constants to be generated, for example all powers of two.

If the value in the Rotate field is zero, the shifter carry out is set to the old value of the CPSR C flag. Otherwise, the shifter carry out is set to bit 31 of the shifter result, just as though an ROR had been performed (see ***Figure 4-9: Rotate right*** on page 4-12).

4.5.4 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 automatically restore both PC and CPSR. This form of instruction must not be used in User mode or System mode.

Note Bits [1:0] of R15 are set to zero when read from, and ignored when written to.

4.5.5 Using R15 as an operand

If R15 (the PC) is used as an operand in a data processing instruction and the shift amount is instruction-specified, the PC value will be the address of the instruction plus 8 bytes.

For any register-controlled shift instructions, neither Rn nor Rm may be R15.

4.5.6 MOV and MVN opcodes

With MOV and MVN opcodes, the Rn field is ignored and should be set to 0000.

4.5.7 TEQ, TST, CMP and CMN opcodes

These instructions do not write the result of their operation but do set flags in the CPSR. An assembler will always set the S flag for these instructions, even if you do not specify this in the mnemonic. The Rd field is ignored and should be set to 0000.

In 32-bit modes, the TEQP form of the instruction used in earlier processors should not be used: the PSR transfer operations (MRS, MSR) must be used instead. Please refer to ***Appendix C, 26-bit Operations on ARM810*** for information on 26-bit mode operation.

Note The S bit (bit 20) of these instructions must be a 1; a 0 in this bit will cause the instruction to be something other than a data processing instruction.

4.5.8 Instruction cycle times

Note that the cycle times given here are given for the ARM8 processor core, and do not give any information about the additional cycles that may be taken as a result of Cache Misses, MMU Page table walks etc. Future versions of the ARM810 Datasheet will provide such information. Please refer to **Chapter 12, Bus Interface** *for timing details of off-chip accesses.*

Data Processing instructions vary in the number of incremental cycles taken, as shown in ***Table 4-2: Instruction cycle times*** on page 4-14.

Description	Cycles
Normal	1
If the opcode is one of ADD, ADC, CMP, CMN, RSB, RSC, SUB, SBC and there is a complex shift (anything other than LSL #0, LSL #1, LSL #2 or LSL #3)	+1
If a register-specified shift is used	+1
With PC written and the S bit is clear	+2
With PC written and the S bit is set	+3

Table 4-2: Instruction cycle times

4.5.9 Assembler syntax

The data processing instructions have the following syntax:

One operand instructions

MOV, MVN

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

Instructions that do not produce a result

CMP, CMN, TEQ, TST

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

Two operand instructions

AND, EOR, SUB, RSB, ADD, ADC, SBC, RSC, ORR, BIC

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

where:

`{cond}` is a two-character condition mnemonic. The assembler assumes AL (ALways) if no condition is specified.

`{S}` if present, specifies that the CPSR flags will be affected (implied for CMP, CMN, TEQ, TST).

`Rd` is an expression evaluating to a valid register number.

`Rn` is an expression evaluating to a valid register number.

`<Op2>` is `Rm{,<shift>}` or `#<expression>,` where `<shift>` is one of:

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

`<shiftname>` can be:

- ASL (ASL is a synonym for LSL)
- LSL
- LSR
- ASR
- ROR

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

If there is a choice of forms (for example as in #0, which can be represented using 0 rotated by 0, 2, 4,...30) the assembler will use a rotation by 0 wherever possible. This affects whether C will be changed in a logical operation with the S bit set - see ***4.5.3 Immediate operand rotates*** on page 4-13. If the rotation is 0, then C won't be modified. If the rotation is non-zero, it will be set to the last rotated bit as shown in ***Figure 4-9: Rotate right*** on page 4-12.

It is also possible to specify the 8-bit immediate and the rotation amount explicitly, by writing `<Op2>` as:

```
#<immediate>,<rotate>
```

where:

`<immediate>` is a number in the range 0-255

`<rotate>` is an even number in the range 0-30

4.5.10 Examples

```
ADDEQR2,R4,R5       ; if the Z flag is set make R2:=R4+R5

TEQSR4,#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

MOVSPC,R14          ; return from exception and restore CPSR
                    ; from SPSR_mode

MOVS R0,#1          ; R0 becomes 1; N and Z flags cleared;
                    ; C and V flags unchanged

MOVS R0,#4,2        ; R0 becomes 1 (4 rotated right by 2);
                    ; N, Z and C flags cleared, V flag unchanged
```

4.6 PSR Transfer (MRS, MSR)

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

These instructions allow access to the CPSR and SPSR registers.

Figure 4-11: MSR (transfer register contents or immediate value to PSR) on page 4-18 and ***Figure 4-12: MRS (transfer PSR contents to a register)*** on page 4-19 show the encodings.

MRS allows the contents of the CPSR or SPSR_<mode> register to be moved to a general register. MSR allows the contents of a general register or an immediate value to be moved to the CPSR or SPSR_<mode> register, with the option of affecting any subset of bytes in the register, including:

- the flag bits only
- the control bits only
- both the flag and control bits

4.6.1 MSR operands

A register operand is any general-purpose register except R15.

An immediate operand is constructed by taking the 8-bit immediate in the Imm field, zero-extending it to 32 bits, and rotating it by twice the value in the Rotate field. This enables many common constants to be generated, for example all powers of two.

4.6.2 Operand restrictions

In User mode, the control bits of the CPSR are protected so that only the condition code flags can be changed. In other (privileged) modes, it is possible to alter the entire CPSR.

The mode at the time of execution determines which of the SPSR registers is accessible: for example, only SPSR_fiq can be accessed when the processor is in FIQ mode.

R15 cannot be specified as the source or destination register.

Note Do not attempt to access an SPSR in User mode or System mode, since no such register exists.

4.6.3 Reserved bits

Only eleven bits of the PSR are defined in ARM810 (N, Z, C, V, I, F and 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 ARM810 programs and future processors, you should observe the following rules:

- Reserved bits must be preserved when changing the value in a PSR.
- Programs must not rely on specific values from reserved bits when checking the PSR status, since in future processors they may read as one or zero.

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

The reserved flag bits (bits 27:24) are an exception to this rule; they may have any values written to them. Any future use of these bits will be compatible with this. In particular, there is no need to use the read-modify-write strategy on these bits.

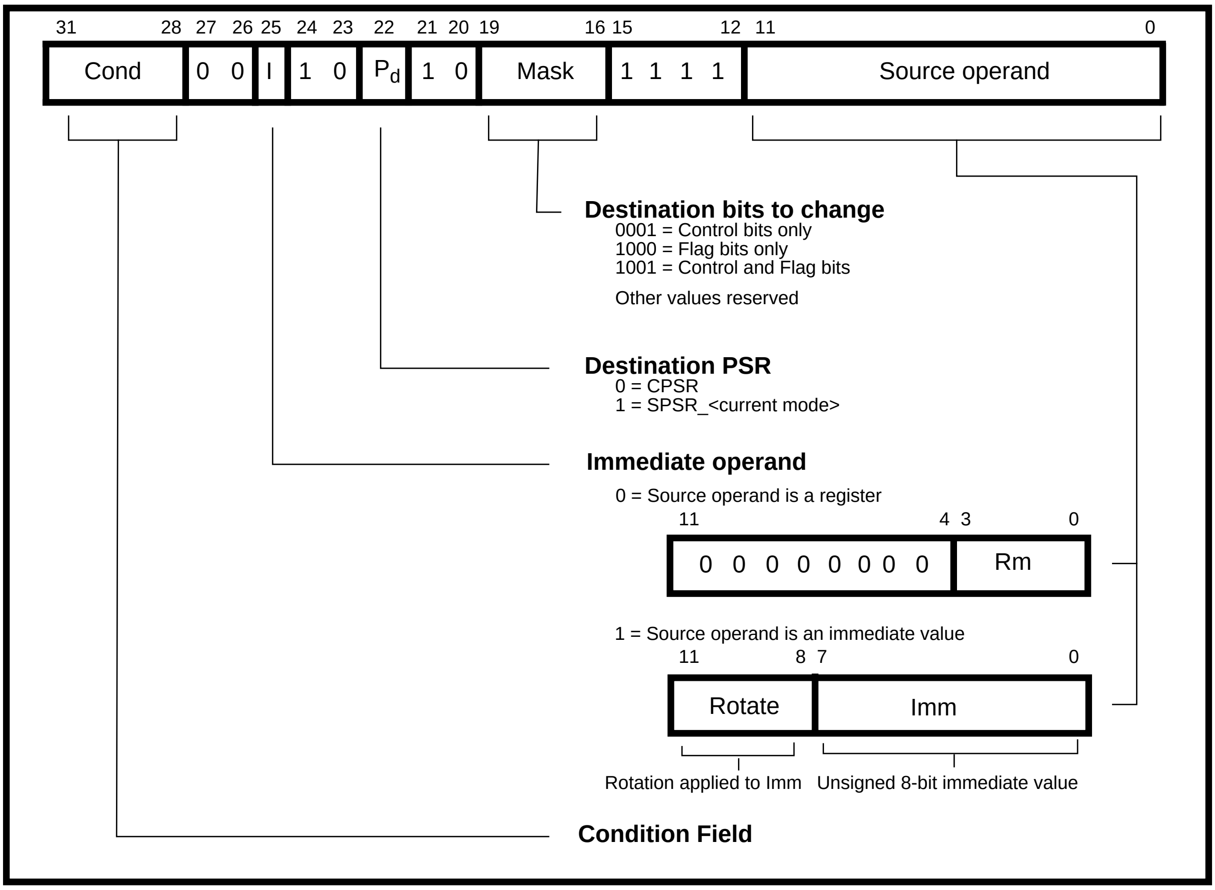


Figure 4-11: MSR (transfer register contents or immediate value to PSR)

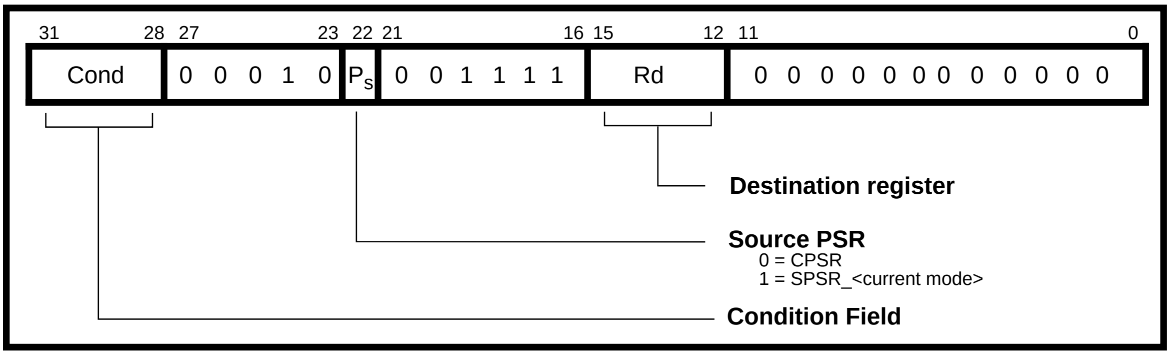


Figure 4-12: MRS (transfer PSR contents to a register)

For example, 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. The following example sets the N, Z, C and V flags:

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

You should not attempt to write an 8-bit immediate value into the whole PSR, since such an operation cannot preserve the reserved bits.

4.6.4 Instruction cycle times

Please note that the cycle times given here are given for the ARM8 processor core, and do not give any information about the additional cycles that may be taken as a result of Cache Misses, MMU Page table walks etc. Future versions of the ARM810 Datasheet will provide such information.

Please refer to ***Chapter 12, Bus Interface*** for timing details of off-chip accesses.

The MRS instruction takes 1 cycle.

The MSR instruction takes 1 cycle when the flag variant is used, or the destination is SPSR_<mode>. In all other cases, MSR takes 3 cycles.

4.6.5 Assembler syntax

The PSR transfer instructions have the following syntax:

Transfer PSR contents to a register

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

Transfer register contents to PSR

```
MSR{cond} <psr>_<fields>,Rm
```

Transfer immediate value to PSR

```
MSR{cond} <psr>_f,#<expression>
```

where:

`{cond}` is a two-character condition mnemonic. The assembler assumes AL (ALways) if no condition is specified.

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

`<psr>` is CPSR or SPSR.

`<fields>` is one of:

_c to set the control field mask bit (bit 0)

_x to set the extension field mask bit (bit 1)

_s to set the status field mask bit (bit 2)

_f to set the flags field mask bit (bit 3)

`#<expression>` is used by the assembler to generate a shifted immediate 8-bit field. If this impossible, the assembler gives an error.

4.6.6 Previous, deprecated MSR assembler syntax

This section describes the old assembler syntax for MSR instructions. These will still work on ARM8, but should be replaced by the new syntax as described in section ***4.6.5 Assembler syntax*** on page 4-20.

Transfer register contents to PSR

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

Transfer immediate value to PSR

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

where:

`{cond}`	is a two-character condition mnemonic. The assembler assumes AL (ALways) if no condition is specified.
`Rd` and `Rm`	are expressions evaluating to a register number other than R15.
`<psrf>`	is one of CPSR, CPSR_all, CPSR_flg, CSPR_ctl, SPSR, SPSR_all, SPSR_flg or SPSR_ctl.
`#<expression>`	is used by the assembler to generate a shifted immediate 8-bit field. If this is impossible, the assembler gives an error.

4.6.7 Examples

User mode

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]
```

System mode

In system mode, 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_ctl,Rm             ; CPSR[7:0]   <- Rm[7:0]

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]
```

Other privileged modes

In other 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_ctl,Rm             ; CPSR[7:0]   <- Rm[7:0]

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_ctl,Rm             ; SPSR_<mode>[7:0]   <- Rm[7:0]

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]
```

4.7 Multiply and Multiply-Accumulate (MUL, MLA)

A multiply instruction is only executed if the specified condition is true. The various conditions are defined at the beginning of this chapter. ***Figure 4-13: Multiply instructions*** shows the instruction encoding.

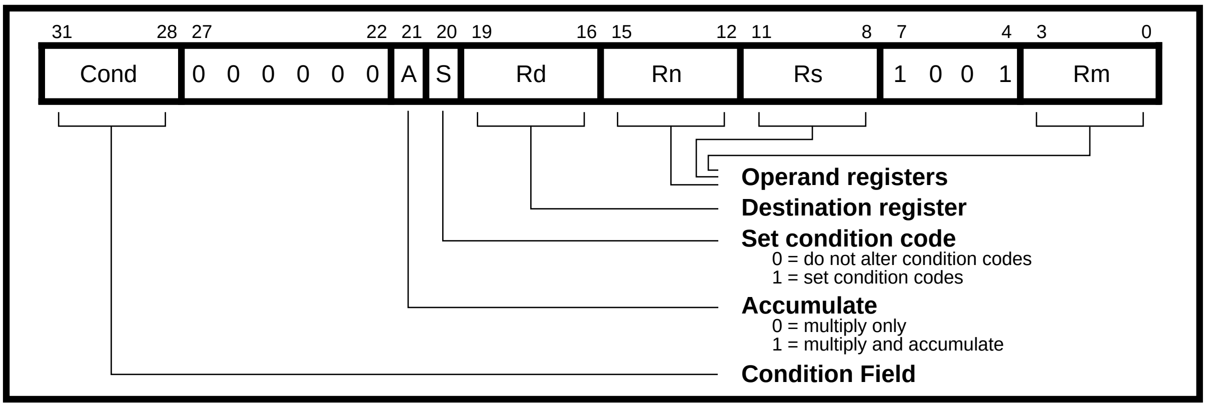


Figure 4-13: Multiply instructions

The multiply and multiply-accumulate instructions perform integer multiplication, optionally accumulating another integer to the product.

Multiply instruction

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

Multiply-accumulate

Multiply-accumulate (MLA) gives Rd:=Rm*Rs+Rn. In some circumstances this can save an explicit ADD instruction.

The result of a signed multiply of 32-bit operands differs from that of an unsigned multiply of 32-bit operands only in the upper 32 bits - the low 32 bits of signed and unsigned results are identical. Since MUL and MLA only produce the low 32 bits of a multiply, they can be used for both signed and unsigned multiplies. Consider the following:

Operand A	Operand B	Result
0xFFFFFFF6	0x00000014	0xFFFFFF38

Signed operands: When the operands are interpreted as signed, A has the value -10 and B has the value 20. The result is -200, which is correctly represented as 0xFFFFFF38.

Unsigned operands: When the operands are interpreted as unsigned, A has the value 4294967286, B has the value 20 and the result is 85899345720, which is represented as 0x13FFFFFF38, the least significant 32 bits of which are 0xFFFFFF38. Again, the representation of the result is correct.

4.7.1 Operand restrictions

- The destination register (Rd) must not be the same as Rm.
- R15 must not be used as Rd, Rm, Rn or Rs.

4.7.2 CPSR flags

Setting the CPSR flags is optional, and is controlled by the S bit. If this is set:

N	is made equal to bit 31 of the result.
Z	is set if and only if the result is zero.
C	is set to a meaningless value.
V	is unaffected.

4.7.3 Instruction cycle times

Please note that the cycle times given here are given for the ARM8 processor core, and do not give any information about the additional cycles that may be taken as a result of Cache Misses, MMU Page table walks etc. Future versions of the ARM810 Datasheet will provide such information.

Please refer to ***Chapter 12, Bus Interface*** for timing details of off-chip accesses.

MUL and MLA take from 3 to 6 cycles to execute, depending upon the early termination, as follows:

Basic cycle count	6 (including any accumulate)
Early termination	-(0 to 3)

4.7.4 Assembler syntax

The multiply instructions have the following syntax:

```
MUL{cond}{S} Rd,Rm,Rs
MLA{cond}{S} Rd,Rm,Rs,Rn
```

where:

`{cond}`	is a two-character condition mnemonic. The assembler assumes AL (ALways) if no condition is specified.
`{S}`	if present, specifies that the CPSR flags will be affected.
`Rd,Rm,Rs,Rn`	are expressions evaluating to a register number other than R15.

4.7.5 Examples

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

4.8 Multiply Long and Multiply-Accumulate Long (MULL, MLAL)

A multiply long instruction is only executed if the specified condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in ***Figure 4-14: Multiply Long instructions***.

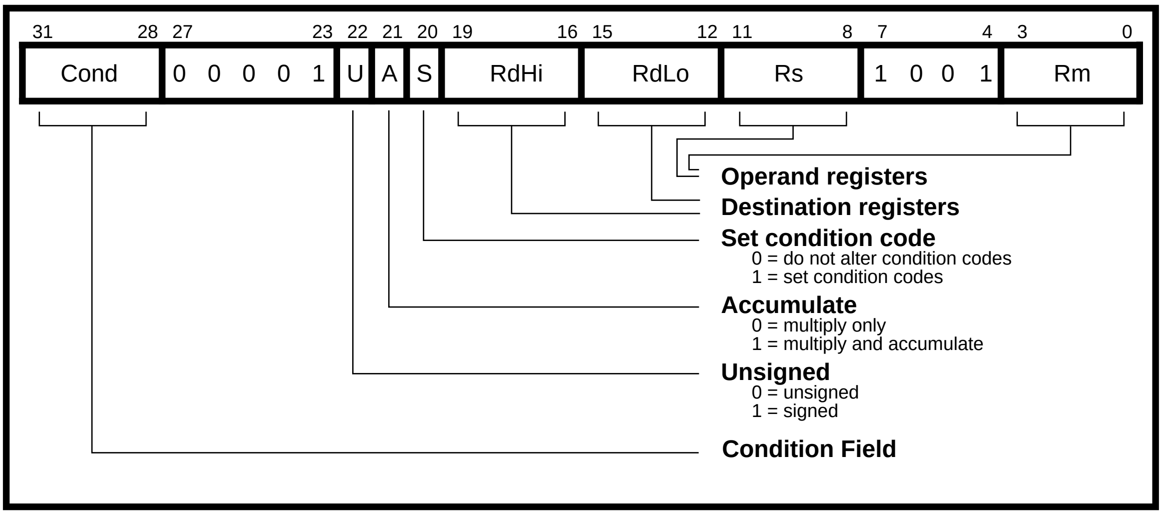


Figure 4-14: Multiply Long instructions

The multiply long instructions perform integer multiplication on two 32-bit operands and produce 64-bit results. Signed and unsigned multiplication each with optional accumulate give rise to four variations.

Multiply (UMULL and SMULL)

UMULL and SMULL take two 32-bit numbers and multiply them to produce a 64-bit result of the form RdHi,RdLo := Rm * Rs. The lower 32 bits of the 64-bit result are written to RdLo, the upper 32 bits of the result are written to RdHi.

Multiply-accumulate (UMLAL and SMLAL)

UMLAL and SMLAL take two 32-bit numbers, multiply them, and add a 64-bit number to produce a 64-bit result of the form RdHi,RdLo := Rm * Rs + RdHi,RdLo. The lower 32 bits of the 64-bit number to add are read from RdLo. The upper 32 bits of the 64-bit number to add are read from RdHi. The lower 32 bits of the 64-bit result are written to RdLo, and the upper 32 bits of the 64-bit result are written to RdHi.

UMULL and UMLAL treat all of their operands as unsigned binary numbers, and write an unsigned 64-bit result. The SMULL and SMLAL instructions treat all of their operands as two’s-complement signed numbers and write a two’s-complement signed 64-bit result.

4.8.1 Operand restrictions

- R15 must not be used as an operand or as a destination register.
- RdHi, RdLo and Rm must all specify different registers.

4.8.2 CPSR Flags

Setting the CPSR flags is optional, and is controlled by the S bit. If this is set:

N	is made equal to bit 63 of the result
Z	is set if and only if all 64 bits of the result are zero
C	is set to a meaningless value
V	is set to a meaningless value

4.8.3 Instruction cycle times

Please note that the cycle times given here are given for the ARM8 processor core, and do not give any information about the additional cycles that may be taken as a result of Cache Misses, MMU Page table walks etc. Future versions of the ARM810 Datasheet will provide such information.

Please refer to ***Chapter 12, Bus Interface*** for timing details of off-chip accesses.

MULL and MLAL take from 4 to 7 cycles to execute, depending upon the early termination, as follows:

Basic cycle count	7 (including any accumulate)
Early termination	-(0 to 3)

4.8.4 Assembler syntax

The multiply long instructions have the following syntax:

Unsigned Multiply Long (32 x 32 = 64)

```
UMULL{cond}{S}    RdLo,RdHi,Rm,Rs
```

Unsigned Multiply and Accumulate Long (32 x 32 + 64 = 64)

```
UMLAL{cond}{S}    RdLo,RdHi,Rm,Rs
```

Signed Multiply Long (32x 32 = 64)

```
SMULL{cond}{S}    RdLo,RdHi,Rm,Rs
```

Signed Multiply and Accumulate Long (32 x 32 + 64 = 64)

```
SMLAL{cond}{S}    RdLo,RdHi,Rm,Rs
```

where

`{cond}`	is a two-character condition mnemonic. The assembler assumes AL (ALways) if no condition is specified.
`{S}`	if present, specifies that the CPSR flags will be affected.
`RdLo,RdHi,Rm,Rs`	are expressions evaluating to a register number other than R15.

```
Examples

UMULL           R1,R4,R2,R3;; R4,R1:=R2*R3

UMLALS          R1,R5,R2,R3;; R5,R1:=R2*R3+R5,R1, also ; ;
                               ; setting condition codes
```

4.9 Single Data Transfer (LDR, STR)

A single data transfer instruction is only executed if the specified condition is true. The various conditions are defined at the beginning of this chapter. ***Figure 4-15: Single data transfer instructions*** shows the instruction encoding.

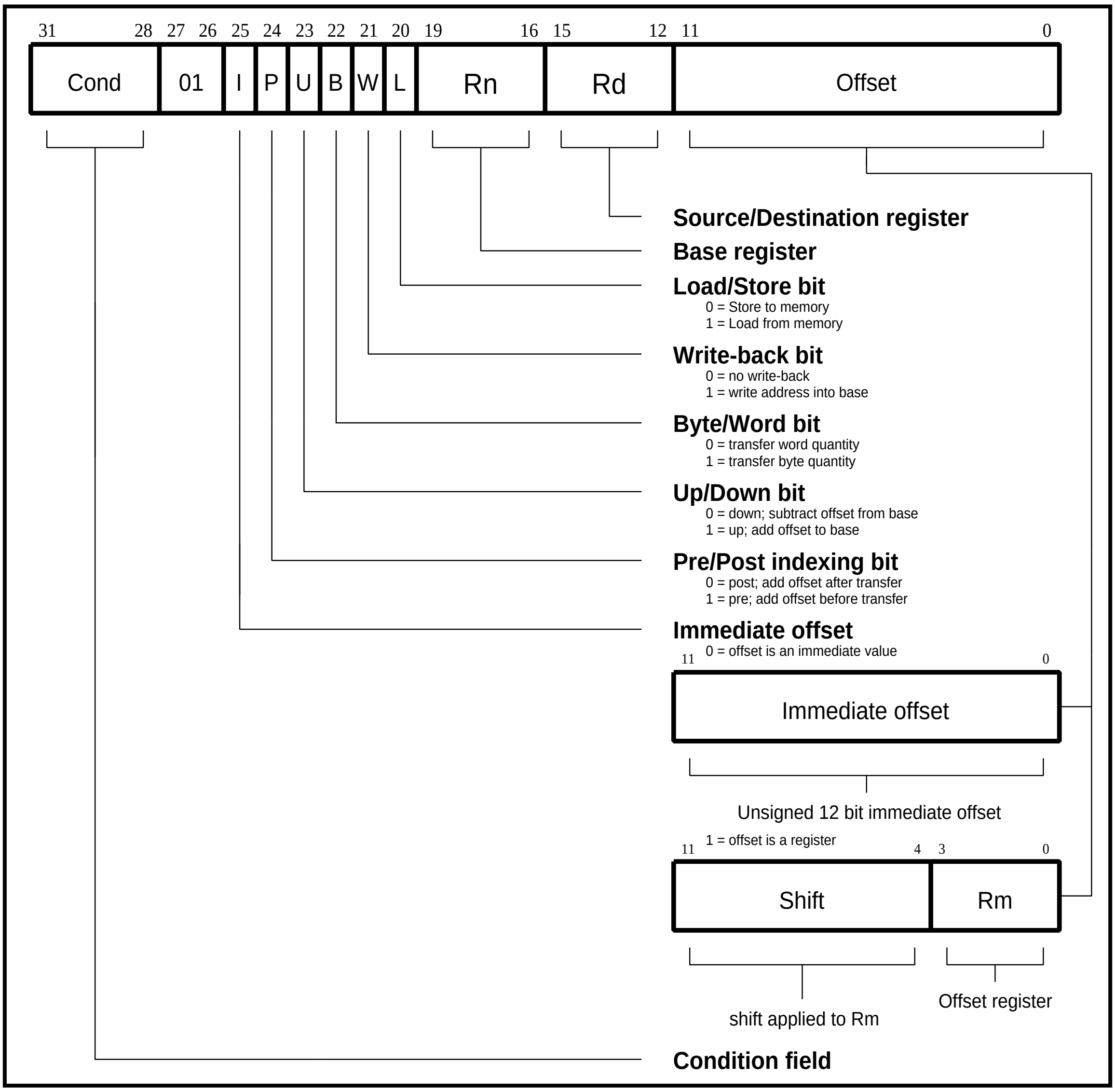


Figure 4-15: Single data transfer instructions

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 or subtracting an offset from a base register. If auto-indexing is required, the result may be written back into the base register.

4.9.1 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, 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 facility.

4.9.2 Shifted register offset

The 8 shift control bits are described in ***4.5.2 Shifts*** on page 4-10. However, register-specified shift amounts are not available in this instruction class.

4.9.3 Bytes and words

This instruction class may be used to transfer a byte (B=1) or a word (B=0) between an ARM810 register and memory.

The action of LDR(B) and STRB instructions is influenced by the BIGEND control signal. The two possible configurations are:

- Little-endian
- Big-endian

Little-endian configuration

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 3-1: Little-endian addresses of bytes within words*** on page 3-3.

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.

Word load (LDR) Any non-word-aligned address will cause the data read to be rotated into the register so that the addressed byte occupies bits 0 to 7. This means that halfwords 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 4-16: Little-endian offset addressing*** on page 4-29.

Note The LDRH and LDRSH insrtuctions provide a more efficient way to load half-words on ARM810. This method of loading half-words should therefore only be used if compatibility with previous ARM processors is required. See ***4.10 Halfword and Signed Data Transfer*** on page 4-34 for further details.

Word store (STR) will normally generate a word-aligned address. The word presented to the data bus is not affected if the address is non-word-aligned, so bit 31 of the register being stored always appears on data bus output 31.

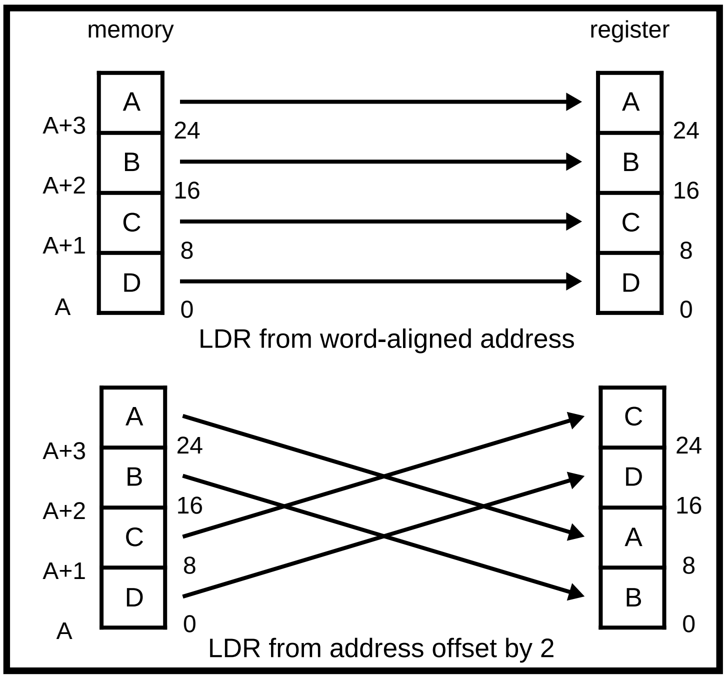


Figure 4-16: Little-endian offset addressing

Big-endian configuration

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 3-2: Big-endian addresses of bytes within words*** on page 3-3.

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.

Word load (LDR) will normally 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 halfwords 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.

Note The LDRH and LDRSH instructions provide a more efficient way to load half-words on ARM810. This method of loading half-words should therefore only be used if compatibility with previous ARM processors is required. See ***4.10 Halfword and Signed Data Transfer*** on page 4-34 for details.

Word store (STR) will normally generate a word-aligned address. The word presented to the data bus is not affected if the address is not word-aligned, so that bit 31 of the register being stored always appears on data bus output 31.

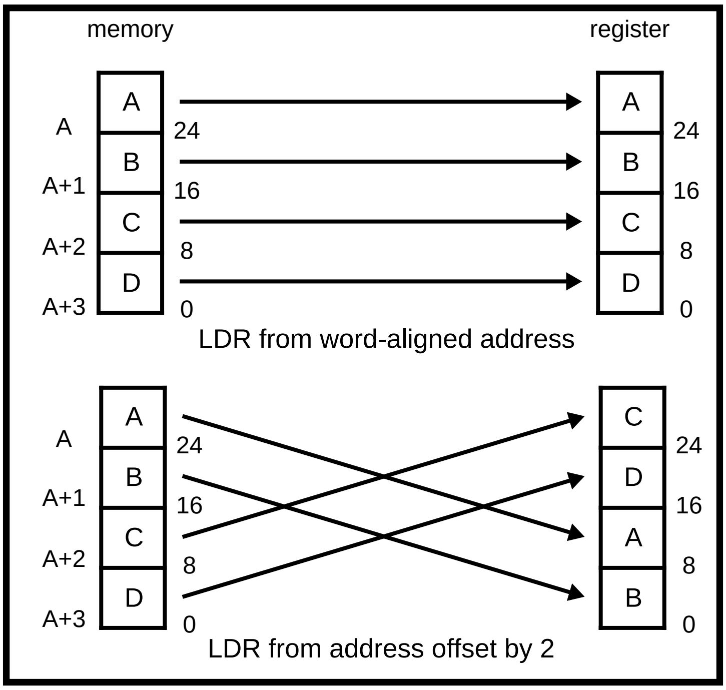


Figure 4-17: Big-endian offset addressing

4.9.4 Use of R15

Do not specify write-back if R15 is the base register (Rn). When using R15 as the base register, it must be remembered that it contains an address 8 bytes on from the address of the current instruction.

Do not specify post-indexing (forcing writeback) to Rn when Rn is R15.

Do not specify R15 as the register offset (Rm).

When R15 is the source register (Rd) of a register store (STR) instruction, the stored value will be the address of the instruction plus 8. Note that this is different from previous ARM processors, which stored the address of the register plus 12.

When R15 is the source register (Rd) of a register store (STR) instruction, or the destination register (Rd) of a register load (LDR) instruction, the byte form of the instruction (LDRB or STRB) must not be used, *and* the address must be word-aligned.

Note Bits [1:0] of R15 are set to zero when read from, and are ignored when written to.

4.9.5 Restrictions on the use of the base register

In the following example, it may sometimes be impossible to calculate the initial value of R0 after an abort in order to restart the instruction:

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

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

When an LDR instruction specifies (or implies) base writeback, register positions Rd and Rn should not be the same register.

4.9.6 Data aborts

Please refer to ***3.6.3 Aborts*** on page 3-9 for details of aborts in general.

In some situations a transfer to or from an address may cause a memory management system to generate an abort.

For example, in a system which uses virtual memory, the required data may be absent from main memory. The memory manager can signal a problem by signalling a Data Abort to the processor, whereupon the Data Abort trap will be taken. It is up to the system software to resolve the cause of the problem, after which the instruction can be restarted and the original program continued.

In all cases, the base register is restored to its original value before the Abort trap is taken. In the case of an LDR or LDRB, the destination register (Rd) will not have been altered.

4.9.7 Instruction cycle times

Please note that the cycle times given here are given for the ARM8 processor core, and do not give any information about the additional cycles that may be taken as a result of Cache Misses, MMU Page table walks etc. Future versions of the ARM810 Datasheet will provide such information.

Please refer to ***Chapter 12, Bus Interface*** for timing details of off-chip accesses.

LDR instructions take 1 cycle:

- +1 cycle if there is a register offset with a shift other than LSL #0, LSL #1, LSL #2 or LSL #3
- +4 cycles for loading the PC

STR instructions take 1 cycle:

- +1 cycle if there is a register offset (regardless of shift type)

4.9.8 Assembler syntax

The single data transfer instructions have the following syntax:

```
<LDR|STR>{cond}{B}{T} Rd,<Addr>
```

where:

`LDR`	loads from memory into a register.
`STR`	stores from a register into memory.
`{cond}`	is a two-character condition mnemonic. If omitted, the assembler assumes ALways.
`{B}`	if present, specifies byte transfer. If omitted, word transfer is used.
`{T}`	if present, sets the W bit 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.
`<Addr>`	is one of:

An `<expression>` specifying an address:

The assembler will attempt to address this location by generating an instruction that uses the PC as a base, along with a corrected immediate offset. This will be a PC relative, pre-indexed address.
If the address is out of range, an error is generated.

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>`

A post-indexed addressing specification:

`[Rn],#<expression>`	offset of `<expression>` bytes
`[Rn],{+/-}Rm{,<shift>}`	offset of +/- contents of index register, shifted by `<shift>`.

Rn and Rm are expressions evaluating to a register number. If Rn is R15, neither post-indexed addressing nor {!} should be specified.

`<shift>` is one of:

`<shiftname> #expression`	
RRX	(rotate right one bit with extend)
`<shiftname>`	is ASL, LSL, LSR, ASR or ROR (ASL is a synonym for LSL)

`{!}` if present, sets the W bit so that the base register is written back.

4.9.9 Examples

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

STR    R1,[R2],R4      ; store R1 at R2. 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 - 7, filling bits 8 - 31 with 0s

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

4.10 Halfword and Signed Data Transfer (LDRH/STRH/LDRSB/LDRSH)

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 4-18: Halfword and signed data transfer with register offset*** and ***Figure 4-19: Halfword and signed data transfer with immediate offset***.

These instructions are used to load or store halfwords of data and also load sign-extended bytes or halfwords 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.

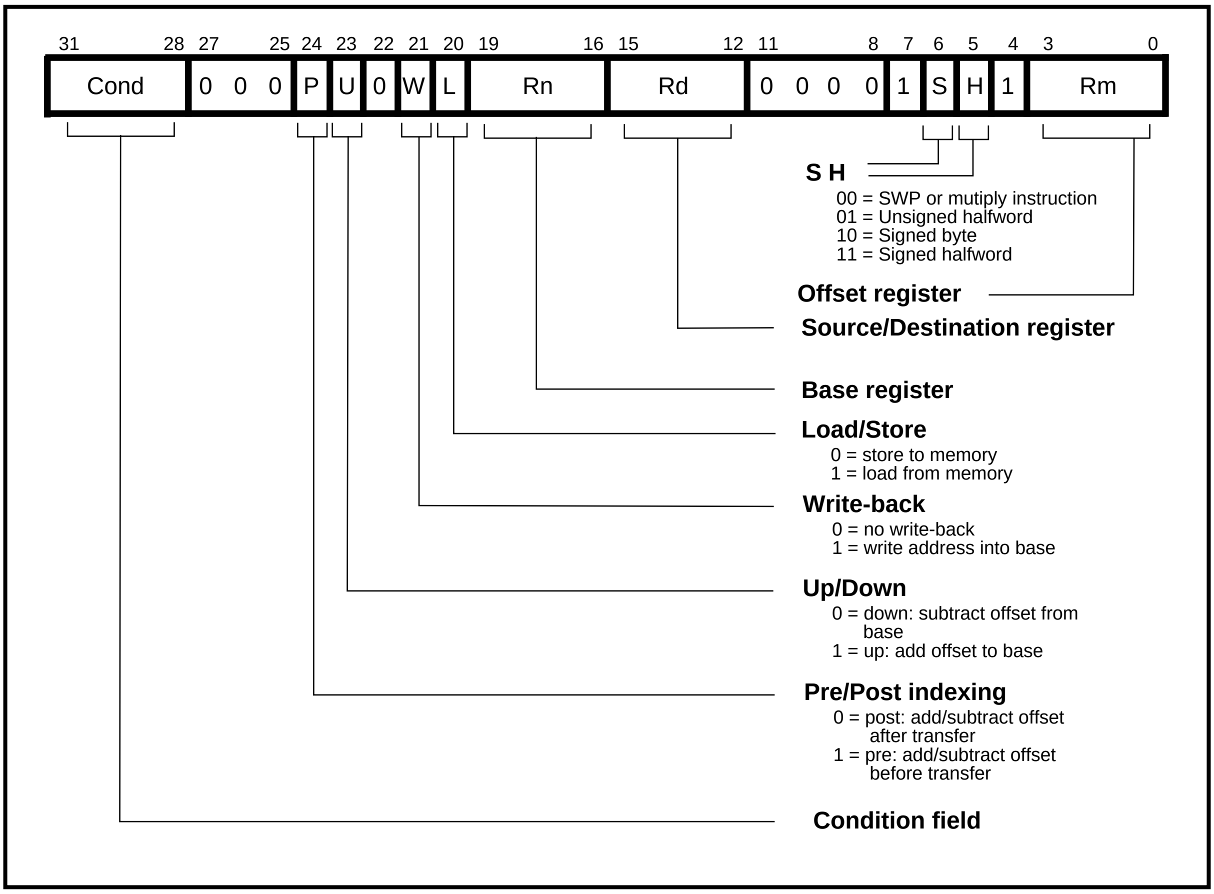


Figure 4-18: Halfword and signed data transfer with register offset

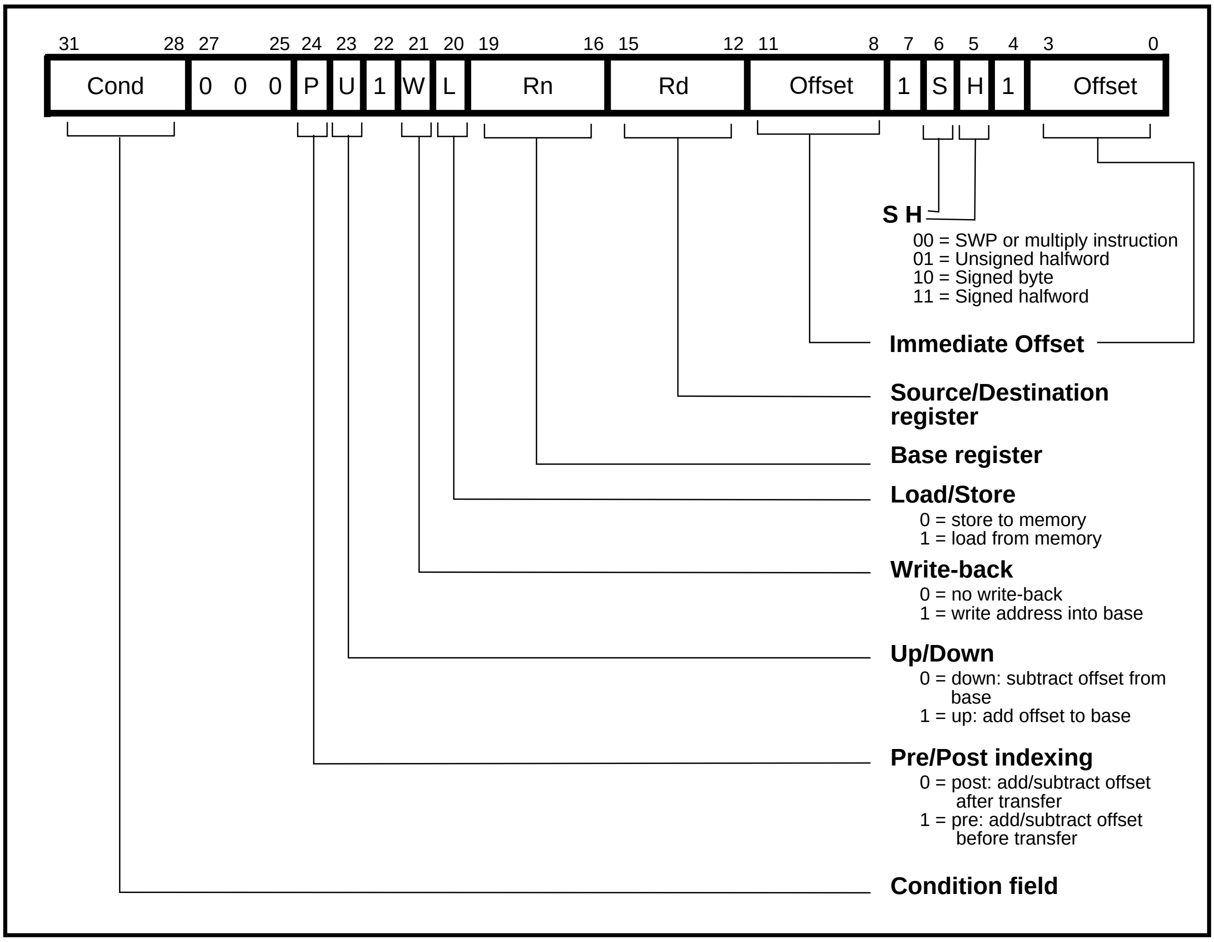


Figure 4-19: Halfword and signed data transfer with immediate offset

4.10.1 Offsets and auto-indexing

The offset from the base may be either an 8-bit unsigned binary immediate value in the instruction, or a second register. In the case of an immediate value, bits 11:8 (xxxx) and bits 3:0 (yyyy) combine to form the offset (xxxxyyyy). 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 register 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 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 if necessary by setting the offset to zero. Therefore post-indexed data transfers always write back the modified base.

The Write-back bit must not be set high (W=1) when post-indexed addressing is selected.

4.10.2 Halfword load and stores

Setting S=0 and H=1 may be used to transfer unsigned halfwords between a register and memory.

The action of LDRH and STRH instructions is influenced by the BIGEND control signal. The two possible configurations are described in the section below.

4.10.3 Signed byte and halfword loads

The S bit controls the loading of sign-extended data. When S=1 the H bit selects between bytes (H=0) and halfwords (H=1). The L bit should not be set LOW (Store) when signed (S=1) operations have been selected.

The LDRSB instruction loads the selected byte into bits 7 to 0 of the destination register and bits 31 to 8 of the destination register are set to the value of bit 7, the sign bit.

The LDRSH instruction loads the selected halfword into bits 15 to 0 of the destination register and bits 31 to 16 of the destination register are set to the value of bit 15, the sign bit.

The action of the LDRSB and LDRSH instructions is influenced by the BIGEND control signal. The two possible configurations are described in the following section.

4.10.4 Endianness and byte/halfword selection

Little-endian configuration

Signed byte load (LDRSB): This load expects data on data bus inputs 7 through to 0 if the supplied address is on a word boundary, on data bus inputs 15 through to 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 the sign bit, the most significant bit of the byte. Please see ***Figure 3-1: Little-endian addresses of bytes within words*** on page 3-3.

Halfword load (LDRSH or LDRH): This load expects data on data bus inputs 15 through to 0 if the supplied address is on a word boundary and on data bus inputs 31 through to 16 if it is on an odd halfword boundary, (A[1]=1).The supplied address should always be on a halfword boundary. If bit 0 of the supplied address is HIGH, an unpredictable value will be loaded. The selected halfword is placed in the bottom 16 bits of the destination register. For unsigned halfwords (LDRH), the top 16 bits of the register are filled with zeros and for signed halfwords (LDRSH) the top 16 bits are filled with the sign bit, the most significant bit of the halfword.

Halfword store (STRH): This store repeats the bottom 16 bits of the source register twice across the data bus outputs 31 through to 0. The external memory system should activate the appropriate halfword subsystem to store the data.

Note The address must be halfword aligned; if bit 0 of the address is HIGH this causes unpredictable behaviour.

Big-endian configuration

Signed byte load (LDRSB): This load (LDRSB) expects data on data bus inputs 31 through to 24 if the supplied address is on a word boundary, on data bus inputs 23 through to 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 the sign bit, the most significant bit of the byte. Please see ***Figure 3-2: Big-endian addresses of bytes within words*** on page 3-3.

Halfword load (LDRSH or LDRH): This load expects data on data bus inputs 31 through to 16 if the supplied address is on a word boundary and on data bus inputs 15 through to 0 if it is on an odd halfword boundary, (A[1]=1). The supplied address should always be on a halfword boundary. If bit 0 of the supplied address is HIGH, an unpredictable value is loaded. The selected halfword is placed in the bottom 16 bits of the destination register. For unsigned halfwords (LDRH), the top 16 bits of the register are filled with zeros and for signed halfwords (LDRSH) the top 16 bits are filled with the sign bit, the most significant bit of the halfword.

Halfword store (STRH): This store repeats the bottom 16 bits of the source register twice across the data bus outputs 31 through to 0. The external memory system should activate the appropriate halfword subsystem to store the data. Note that the address must be halfword aligned, if bit 0 of the address is HIGH this will cause unpredictable behaviour.

4.10.5 Use of R15

Do not specify R15 as:

- the register offset (Rm)
- the destination register (Rd) of a load instruction (LDRH, LDRSH, LDRSB)
- the source register (Rd) of a store instruction (STRH, STRSH, STRSB)

Base register

Do not specify either write-back or post-indexing (which forces write-back) if R15 is specified as the base register (Rn). When using R15 as the base register you must remember that it contains an address 8 bytes on from the address of the current instruction.

4.10.6 Restrictions on the use of the base register

Do not specify post-indexed loads and stores where Rm and Rn are the same register, as they can be impossible to unwind after an abort.

Do not set register positions Rd and Rn to be the same register when a load instruction specifies (or implies) base write-back.

4.10.7 Data aborts

Please refer to ***3.6.3 Aborts*** on page 3-9 for details of aborts in general.

In some situations a transfer to or from an address may cause a memory management system to generate an abort.

For example, in a system which uses virtual memory, the required data may be absent from main memory. The memory manager can signal a problem by signalling a Data Abort to the processor, whereupon the Data Abort trap will be taken. It is up to the system software to resolve the cause of the problem, after which the instruction can be restarted and the original program continued.

In all cases, the base register is restored to its original value before the Abort trap is taken. In the case of an LDRH, LDRSB or LDRSH, the destination register (Rd) will not have been altered.

4.10.8 Instruction cycle times

Please note that the cycle times given here are given for the ARM8 processor core, and do not give any information about the additional cycles that may be taken as a result of Cache Misses, MMU Page table walks etc. Future versions of the ARM810 Datasheet will provide such information.

Please refer to ***Chapter 12, Bus Interface*** for timing details of off-chip accesses.

The cycle times are the same as LDR/STR for *all* cases of (H, SH, SB).

Load instructions take 1 cycle.

Store instructions take 1 cycle.

4.10.9 Assembler syntax

```
<LDR|STR>{cond}<H|SH|SB> Rd,<Addr>
```

`LDR`	load from memory into a register
`STR`	Store from a register into memory
`{cond}`	two-character condition mnemonic. See ***4.3 The Condition Field*** on page 4-3
`H`	Transfer halfword quantity
`SB`	Load sign extended byte (Only valid for LDR)
`SH`	Load sign extended halfword (Only valid for LDR)
`Rd`	is an expression evaluating to a valid register number.
`<Addr>`	is one of:

1. An expression which generates an address:

   ```
   <expression>
   ```

   The assembler will attempt to generate an instruction using the PC as a base and an 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, this generates an error.

2. A pre-indexed addressing specification:

`[Rn]`	offset of zero

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

`[Rn,{+/-}Rm]{!}` offset of +/- contents of index register

3 A post-indexed addressing specification:

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

`[Rn],{+/-}Rm` offset of +/- contents of index register.

Rn and Rm are expressions evaluating to a register number. If Rn is R15, neither post-indexed addressing nor `{!}` should be specified.

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

4.10.10 Examples

```
    LDRH   R1,[R2,-R3]!  ; Load R1 from the contents of the
                         ; halfword address contained in
                         ; R2-R3 (both of which are registers)
                         ; and write back address to R2
    STRH   R3,[R4,#14]   ; Store the halfword in R3 at R14+14
                         ; Don't write back
    LDRSB  R8,[R2],#-223 ; Load R8 with the sign extended
                         ; contents of the byte address
                         ; contained in R2 and write back R2-223
                         ; to R2
    LDRNESH R11,[R0]     ; Conditionally load R11 with the sign
                         ; extended contents of the halfword
                         ; address contained in R0.
HERESTRH  R5,[(PC, # (FRED-HERE-8)]
    .                    ; Generate PC relative offset to
    .                    ; address FRED. Store the halfword
    .                    ; in R5 at address FRED
    .
    .
FRED
```

4.11 Block Data Transfer (LDM, STM)

A block data transfer instruction is only executed if the specified condition is true. The various conditions are defined at the beginning of this chapter. ***Figure 4-20: Block data transfer instructions*** shows the instruction encoding.

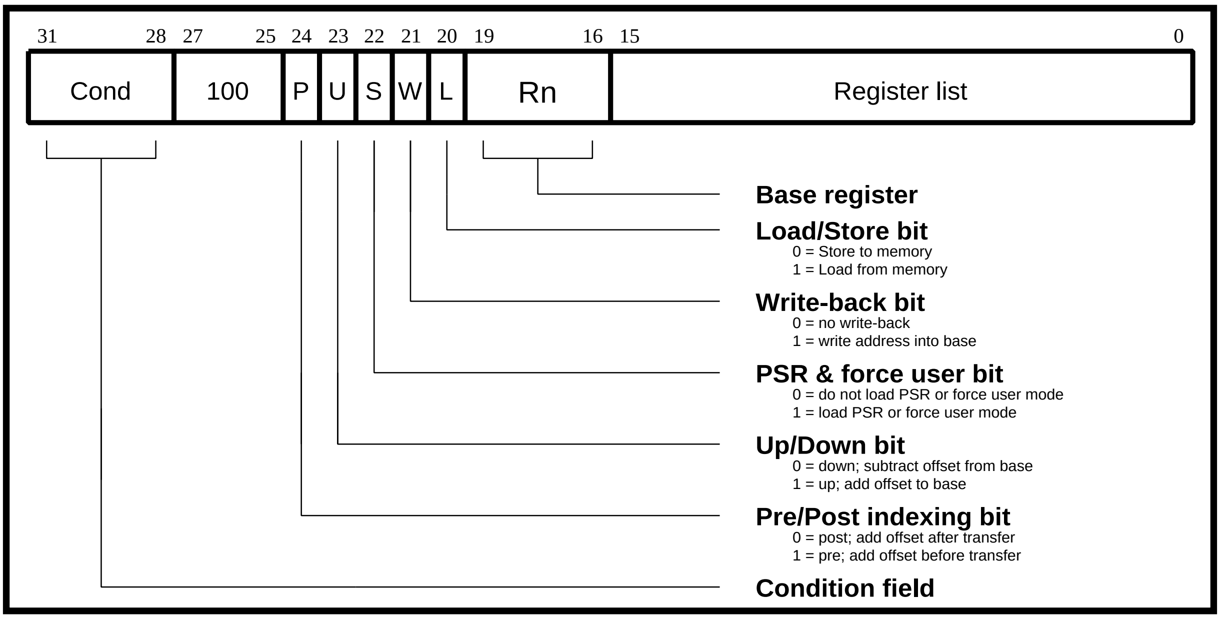

31
28
27
25
24
23
22
21
20
19
16
15
0
Cond
100
P
U
S
W
L
Rn
Register list
Base register
Load/Store bit
0 = Store to memory
1 = Load from memory
Write-back bit
0 = no write-back
1 = write address into base
PSR & force user bit
0 = do not load PSR or force user mode
1 = load PSR or force user mode
Up/Down bit
0 = down; subtract offset from base
1 = up; add offset to base
Pre/Post indexing bit
0 = post; add offset after transfer
1 = pre; add offset before transfer
Condition field

Figure 4-20: 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.

4.11.1 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 must not be empty.

Whenever R15 is stored to memory the stored value is the address of the STM instruction plus 8. Note that this is different from previous ARMs which stored the address of the instruction plus 12 (or 8 if R15 is the only register in the list.)

4.11.2 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 stored such that the lowest register is always at the lowermost address in memory, the highest numbered register is always at the uppermost address, and the others are stored in numerical order between them.

The register transfers will occur in ascending order. 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). The figures beginning on page 4-42 show the sequence of register transfers, the addresses used, and the value of Rn after the instruction has completed.

In all cases, if write-back of the modified base was not 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.

4.11.3 Address alignment

The address should normally be a word-aligned quantity. Non-word-aligned addresses do not affect the instruction: no data rotation occurs (as would happen in LDR.) However, the bottom 2 bits of the address will appear on A[1:0] and might be interpreted by the memory system.

4.11.4 Use of the S bit

When the S bit is set in a LDM/STM instruction, its meaning depends on whether R15 is in the transfer list and also on the type of instruction. The S bit should only be set if the instruction is to execute in a privileged mode other than System mode.

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

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

STM with S bit set (User bank transfer)

The registers to be 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 must not be used when this mechanism is employed.

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

The user bank registers are loaded, rather than those in the bank corresponding to the current mode. This is useful for restoring the user state on process switches. Do not use base write-back when this mechanism is employed. Also, take care not to read from a banked register during the following cycle. (Inserting a NOP after the LDM will ensure safety.)

4.11.5 Use of R15

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

Note Bits [1:0] of R15 are set to zero when read from, and are ignored when written to.

4.11.6 Inclusion of the base in the register list

When write-back is specified during an STM, if the base register is the lowest numbered register in the list, then the original base value is stored. Otherwise the value stored is not specified and should not be used.

4.11.7 Data aborts

Please refer to ***3.6.3 Aborts*** on page 3-9 for details of Aborts in general.

When a Data Abort occurs during LDM or STM instructions, further register transfers are stopped. The base register is always restored to its original value (before the instruction had executed) regardless of whether writeback was specified or not. As such, the instruction can always be restarted without any need to adjust the value of the base register in the Data Abort service routine code.

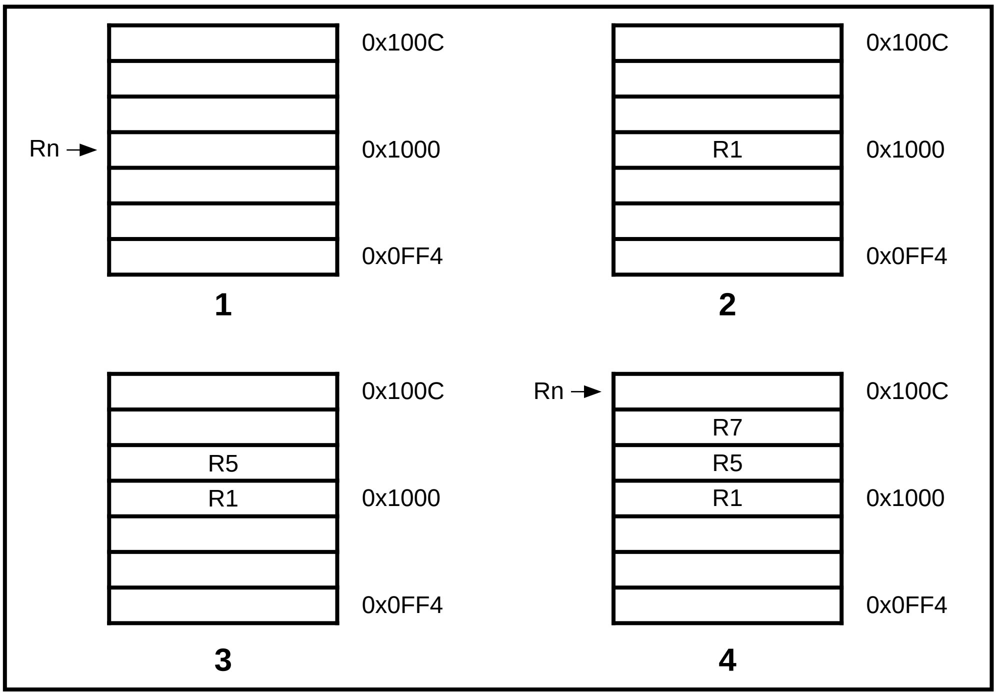


Figure 4-21: Post-increment addressing

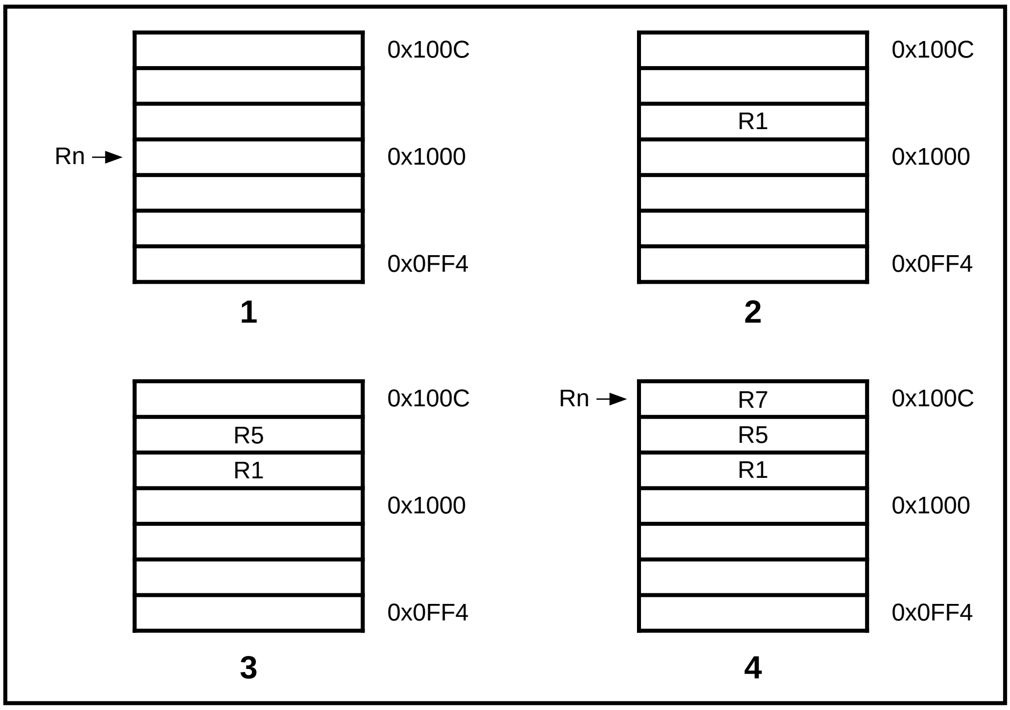


Figure 4-22: Pre-increment addressing

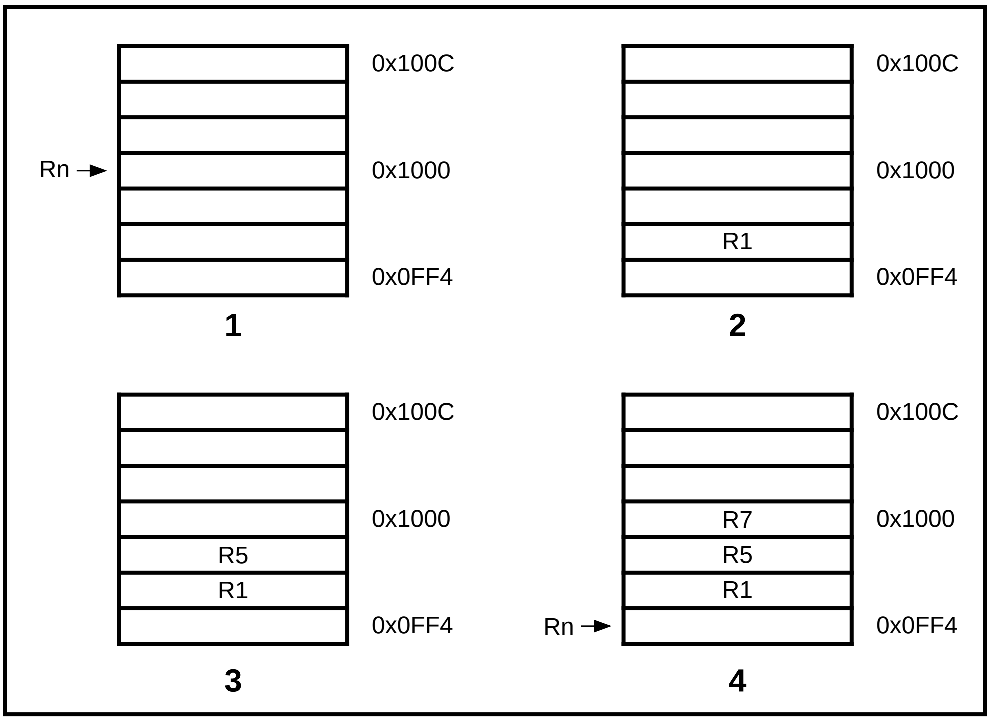


Figure 4-23: Post-decrement addressing

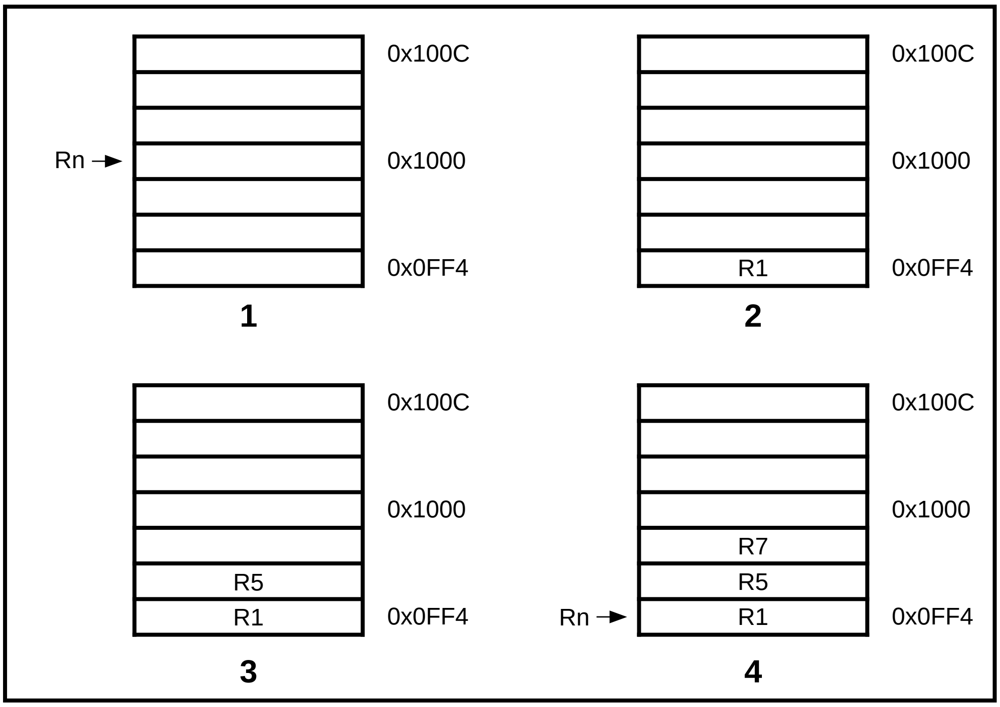


Figure 4-24: Pre-decrement addressing

4.11.8 Instruction cycle times

Note that the cycle times given here are given for the ARM8 processor core, and do not give any information about the additional cycles that may be taken as a result of Cache Misses, MMU Page table walks etc.

Please refer to ***Chapter 12, Bus Interface*** for timing details of off-chip accesses.

The cycle count for LDM instructions depends on the number of ordinary registers being loaded (excluding R15), and whether R15 is being loaded.

The following table shows the basic cycle count for LDM.

Number of Ordinary Registers transferred	Cycles when PC (R15) is not in register list	Cycles when PC (R15) is in register list
0	-	5
1	2	6
2	2	6
3	3	7
4	3	7
5	4	8
6	4	8
7	5	9
8	5	9
9	6	10
10	6	10
11	7	11
12	7	11
13	8	12
14	8	12
15	9	13

Table 4-3: Basic cycle count for LDM

The above assumes that the memory system supports double-bandwidth transfer. If this is not so, then count N cycles for the number of registers being transferred, plus 5 cycles if R15 is loaded, with a minimum of two cycles overall.

A common example of where this might happen in a cached memory system would be when uncacheable memory is being accessed.

Additional cycles may be incurred if the memory system indicates that it is only able to transfer one item of data where two were requested. For example, when accessing the last word in a cache line in a cached memory system.

The following table shows the cycle counts for STM instructions.

Number of Ordinary Registers transferred	Cycles when PC (R15) is not in register list	Cycles when PC (R15) is in register list
0	-	2
1	2	2
2	2	3
3	3	4
4	4	5
5	5	6
6	6	7
7	7	8
8	8	9
9	9	10
10	10	11
11	11	12
12	12	13
13	13	14
14	14	15
15	15	16

Table 4-4: Basic cycle count for STM

Note PC is stored as the address of the current instruction plus 8.

4.11.9 Assembler syntax

The block data transfer instructions have the following syntax:

```
<LDM|STM>{cond}<addressmode> Rn{!},<Rlist>{^}
```

where:

`LDM` loads from memory to registers.

`STM` stores from registers to memory.

`{cond}` is a two-character condition mnemonic. The assembler assumes AL (ALways) if no condition is specified.

`<addressmode>` is one of `<FD|ED|FA|EA|IA|IB|DA|DB>.`
Note that `<addressmode>` is *not* optional.
(See ***Table 4-5: Addressing Mode names*** on page 4-47)

`Rn` is an expression evaluating to a 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, sets the S bit. See ***4.11.4 Use of the S bit*** on page 4-41.

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. These are shown in ***Table 4-5: Addressing Mode names*** on page 4-47.

Key to table:

FD, ED, FA, EA define pre/post indexing and the up/down bit by reference to the form of stack required.

F	Full stack (a pre-index has to be done before storing to the stack)
E	Empty stack
A	Ascending stack (a STM will go up and LDM down)
D	Descending stack (a STM will go down and LDM up)

IA, IB, DA, DB allow control when LDM/STM are not being used for stacks:

IA	Increment After
IB	Increment Before
DA	Decrement After
DB	Decrement Before

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

Table 4-5: Addressing Mode names

Name	Stack	Other	L bit	P bit	U bit
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 4-5: Addressing Mode names

4.11.10 Examples

```
LDMFDSP!,{R0,R1,R2} ; unstack 3 registers
STMIAR0,{R0-R15}    ; save all registers

LDMFDSP!,{R15}      ; unstack R15,CPSR unchanged

LDMFDSP!,{R15}^     ; unstack R15, CPSR <- SPSR_mode
                    ; (allowed only in privileged modes)

STMFDR13,{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:

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

BL  somewhere       ; this nested call will overwrite R14

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

4.12 Single Data Swap (SWP)

A data swap instruction is only executed if the specified condition is true. The various conditions are defined at the beginning of this chapter. ***Figure 4-25: Swap instruction*** shows the instruction encoding.

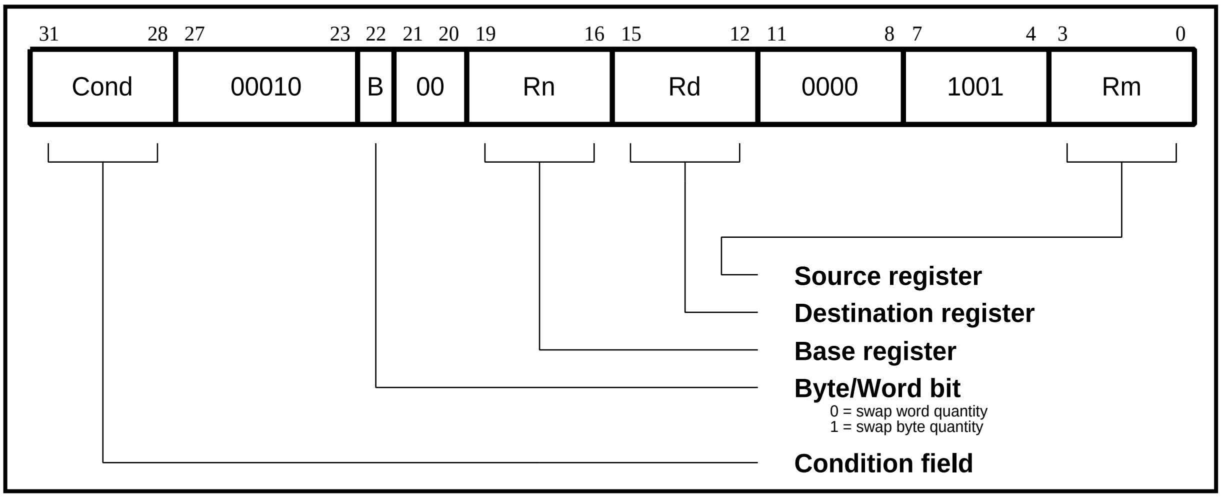


Figure 4-25: Swap instruction

The data swap instruction is used to swap a byte or word quantity between a register and external memory. It 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. It then 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.

Both the read and the write operations result in external accesses to main memory regardless of whether the cache hits or misses. In the case of a cache hit during the write operation, the cache line is updated with the new value and is not marked as dirty.

Swap Read operation:

This performs a single word or byte read that always goes to the external bus, leaving the bus locked for the subsequent write.

Swap Write operation:

This performs a single word or byte write that always goes to the external bus as an unbuffered write. If the write is a cache hit, the cache data is updated and the dirty bit is left unchanged.

The **LOCK** signal on the external interface is used to signal to the external memory manager that the read and write operations of the swap are locked together and should be allowed to complete without interruption; see ***Chapter 12, Bus Interface*** for further details. This operation is important in multi-processor systems, where the swap instruction is the only indivisible operation which may be used to implement semaphores.

4.12.1 Bytes and words

This instruction class may be used to swap a byte (B=1) or a word (B=0) between an ARM810 register and memory. The SWP instruction is implemented as a LDR followed by a STR and the action of these is as described in ***4.9 Single Data Transfer (LDR, STR)*** on page 4-27. In particular, the description of big and little-endian configuration applies to the SWP instruction. Note that there is no halfword SWP.

4.12.2 Use of R15

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

4.12.3 Data aborts

Please refer to ***3.6.3 Aborts*** on page 3-9 for details of Aborts in general.

In some situations, a transfer to or from an address may cause the memory management system to generate an Abort.

If the read operation is aborted, the abort will be returned to ARM8, the write will not take place and the locked indication will be removed from the external bus.

If the read operation succeeds and the write operation is aborted, the abort will be returned to ARM8 and the cache entry will be left with the updated (written) data value. The line will not be invalidated in the cache—this could be done by the abort handler if necessary.

4.12.4 Instruction cycle times

Please note that the cycle times given here are given for the ARM8 processor core, and do not give any information about the additional cycles that may be taken as a result of Cache Misses, MMU Page table walks etc. Future versions of the ARM810 Datasheet will provide such information.

Please refer to ***Chapter 12, Bus Interface*** for timing details of off-chip accesses.

SWP instructions take 2 cycles.

4.12.5 Assembler syntax

The SWP instruction has the following syntax:

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

where:

`{cond}` is a two-character condition mnemonic. The assembler assumes AL (ALways) if no condition is specified.

`{B}` specifies byte transfer. If omitted, word transfer is used.

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

4.12.6 Examples

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

SWPBR2,R3,[R4]      ; load R2 with the byte addressed by R4,
                    ; and store bits 0 to 7 of R3 at R4

SWPEQR0,R0,[R1]     ; conditionally swap the contents of the
                    ; word addressed by R1 with R0
```

4.13 Software Interrupt (SWI)

A SWI instruction is only executed if the specified condition is true. The various conditions are defined at the beginning of this chapter. ***Figure 4-26: Software interrupt instruction*** shows the instruction encoding.

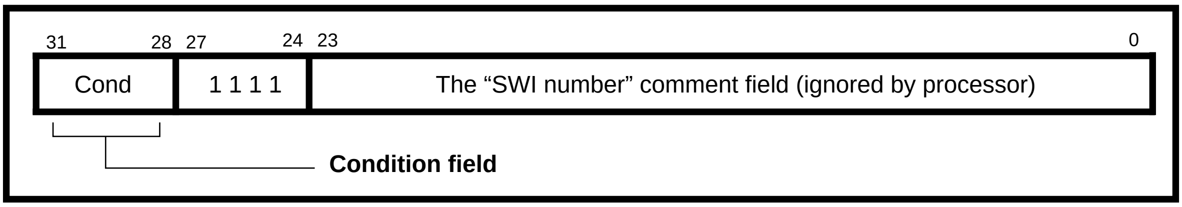


Figure 4-26: Software interrupt instruction

The software interrupt is used to enter Supervisor mode in a controlled manner. It causes the software interrupt trap to be taken, which effects the mode change. The PC is then forced to the SWI vector and the CPSR is saved in SPSR_svc. See ***3.6.4 Software interrupt*** on page 3-10 for more details.

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.

4.13.1 Return from the supervisor

The PC is saved in R14_svc and the CPSR in SPSR_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.

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.

4.13.2 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. This is commonly referred to as the “SWI number”.

4.13.3 Architecturally defined SWIs

The ARM Architecture V4 reserves SWI numbers 0xF00000 to 0xFFFFFF inclusive for current and future Architecturally Defined SWI functions. These SWI numbers should not be used for functions other than those defined by ARM. Please see ***4.17 The Instruction Memory Barrier (IMB) Instruction*** on page 4-64 for examples of two such definitions.

Architecturally defined SWI functions are used to provide a well-defined interface between code which is:

- independent of the ARM processor implementation on which it is running, and
- specific to the ARM processor implementation on which it is running.

The implementation-independent code is provided with a function that is available on all processor implementations via the SWI interface, and which may be accessed by privileged and, where appropriate, non-priviledged (User mode) code.

The Architecturally defined SWI instructions must be implemented in the SWI handler using processor specific code sequences supplied by ARM. Please refer to ***Appendix E, Implementing the Instruction Memory Barrier Instruction*** for details.

4.13.4 Instruction cycle times

Please note that the cycle times given here are given for the ARM8 processor core, and do not give any information about the additional cycles that may be taken as a result of Cache Misses, MMU Page table walks etc. Future versions of the ARM810 Datasheet will provide such information.

Please refer to ***Chapter 12, Bus Interface*** for timing details of off-chip accesses

SWI instructions take 4 cycles to execute.

4.13.5 Assembler syntax

The SWI instruction has the following syntax:

```
SWI{cond} <expression>
```

where:

`{cond}`	is a two-character condition mnemonic. The assembler assumes AL (ALways) if no condition is specified.
`<expression>`	is evaluated and placed in the comment field (which is ignored by ARM810).

4.13.6 Examples

```
SWI ReadC           ; get next character from read stream
SWI WriteI+”k”      ; output a ”k” to the write stream
SWINE0              ; conditionally call supervisor
                    ; with 0 in comment field
```

The above examples assume that suitable supervisor code exists at the SWI vector address, for instance:

```
    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 entry table start address
    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
```

Note ADR is a directive that instructs the assembler to use an ADD or SUB instruction to create the address of a label, so in the above instance

```
    ADR    R2,EntryTable
```

is equivalent to

```
    SUB    R2,R15,#{PC}+8-EntryTable
```

4.14 Coprocessor Data Operations (CDP)

ARM810 will bounce all CDP instructions , forcing them to take the Undefined Instruction trap. The coprocessor instruction may then be emulated.

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 4-27: 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 the ARM810, and it may not wait for the operation to complete. The coprocessor could contain a queue of such instructions awaiting execution, and their execution can overlap other activity, allowing the coprocessor and the ARM810 to perform independent tasks in parallel.

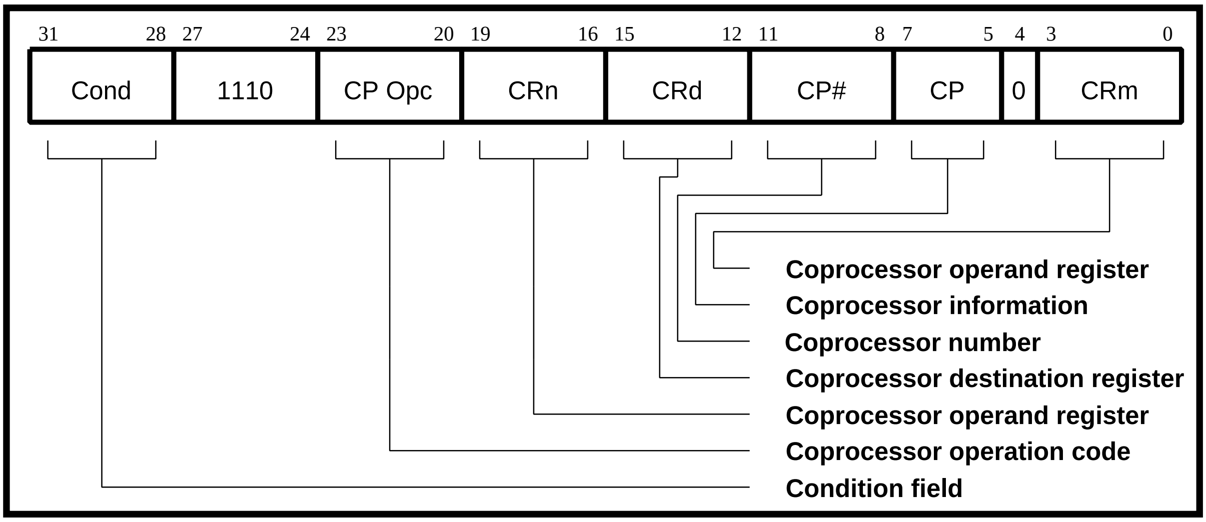


Figure 4-27: Coprocessor data operation instruction

4.14.1 The coprocessor fields

Only bit 4 and bits 24 to 31 are significant to the processor. 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 must 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.

4.14.2 Instruction cycle times

All CDP instructions must be emulated in software: the number of cycles taken will depend on the coprocessor support software.

4.14.3 Assembler syntax

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

where:

`{cond}`	two character condition mnemonic, see ***Figure 4-2: Condition codes*** on page 4-3
`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

4.14.4 Examples

```
CDP p1,10,c1,c2,c3; request coproc 1 to do operation 10
             ; on CR2 and CR3, and put the result in
             ; CR1
CDPEQp2,5,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
```

4.15 Coprocessor Data Transfers (LDC, STC)

ARM810 will bounce all LDC and STC instructions, forcing them to take the Undefined Instruction trap. The coprocessor instruction may then be emulated.

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 4-29: Coprocessor register transfer instructions***.

This class of instruction is used to load (LDC) or store (STC) a subset of a coprocessors’s registers directly to memory. The processor is responsible for supplying the memory address, and the coprocessor supplies or accepts the data and controls the number of words transferred.

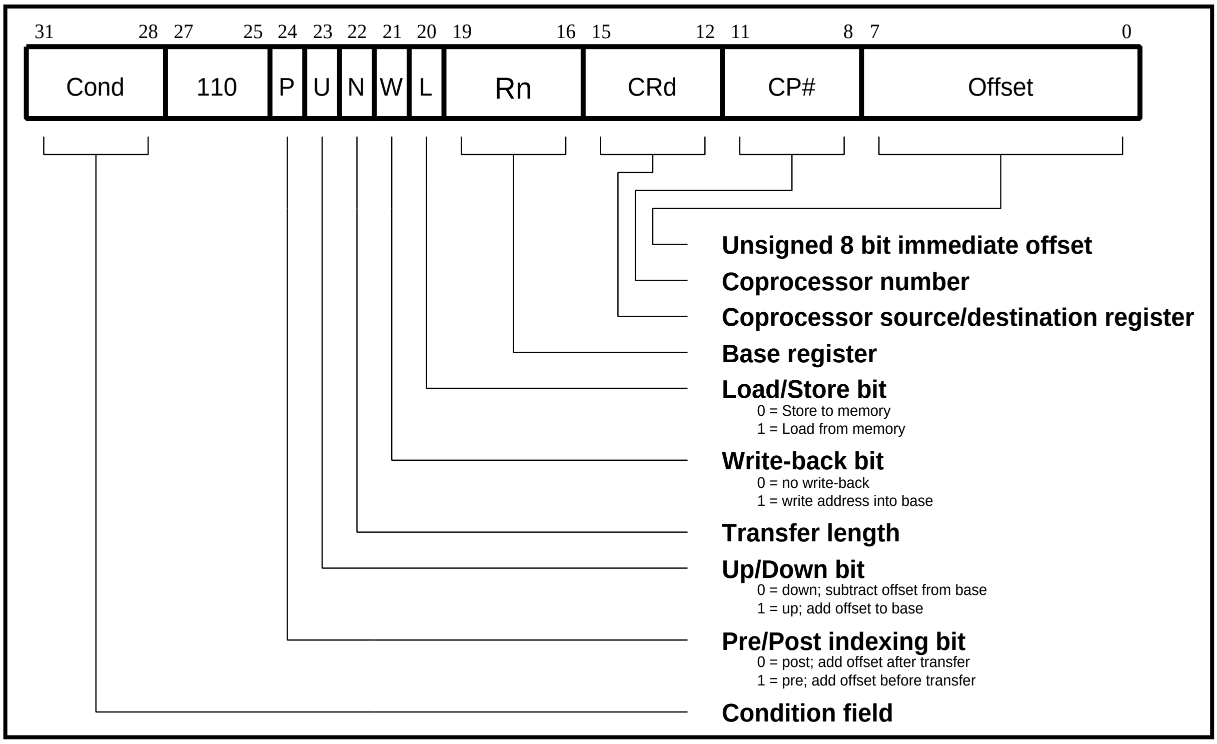


Figure 4-28: Coprocessor data transfer instructions

4.15.1 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.

4.15.2 Addressing modes

The processor 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 for coprocessor data transfers the immediate offsets are 8 bits wide and specify *word* offsets, whereas for single data transfers they are 12 bits wide and specify *byte* offsets.

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. Instructions where P=0 and W=0 are reserved, and must not be used.

4.15.3 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.

4.15.4 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.

4.15.5 Data aborts

If the address is legal but the memory manager generates an abort, the data abort trap is taken. The base register is restored to its original value, and all other processor state are preserved. Any coprocessor emulation is partly responsible for ensuring that the data transfer can restart after the cause of the abort is resolved, and must ensure that any subsequent actions it undertakes can be repeated when the instruction is retried.

4.15.6 Instruction cycle times

All LDC and STC instructions must be emulated in software: the number of cycles taken will depend on the coprocessor support software.

4.15.7 Assembler syntax

```
<LDC|STC>{cond}{L} p#,cd,<Addr>
```

where:

`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 4-2: Condition codes*** on page 4-3.

`p#` the unique number of the required coprocessor

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

`<Addr>` can be:

1 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.

2 A pre-indexed addressing specification:

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

3 A post-indexed addressing specification:

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

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

`Rn` expression evaluating to a valid ARM810 register number

4.15.8 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 Though the address offset is expressed in bytes, the instruction offset field is in words. The assembler will adjust the offset appropriately.

4.16 Coprocessor Register Transfers (MRC, MCR)

ARM810 has only one internal coprocessor; CP15, the system control coprocessor. The MRC and MCR instructions are used to transfer register contents between the core and the coprocessor. Please refer to ***Chapter 5, Configuration*** for details of the register arrangement and operations.

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 4-29: Coprocessor register transfer instructions***.

This class of instruction is used to communicate information directly between ARM810 and a coprocessor. An example of a coprocessor to processor 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 a processor register. A FLOAT of a 32-bit value in a processor register into a floating point value within the coprocessor illustrates the use of a processor register to coprocessor transfer (MCR).

An important use of this instruction is to communicate control information directly from the coprocessor into the processor 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.

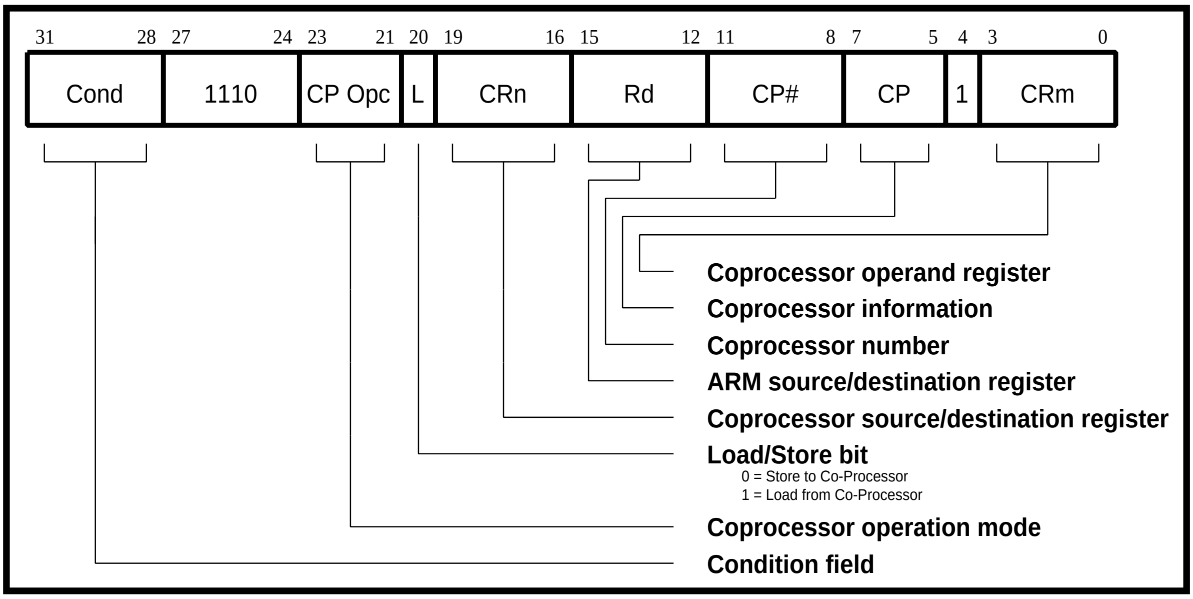

31
28
27
24
23
21
20
19
16
15
12
11
8
7
5
4
3
0
Cond
1110
CP Opc
L
CRn
Rd
CP#
CP
1
CRm
Coprocessor operand register
Coprocessor information
Coprocessor number
ARM source/destination register
Coprocessor source/destination register
Load/Store bit
0 = Store to Co-Processor
1 = Load from Co-Processor
Coprocessor operation mode
Condition field

Figure 4-29: Coprocessor register transfer instructions

4.16.1 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.

4.16.2 Transfers from R15

Do not specify a coprocessor register transfer from ARM810 with R15 as the source register.

4.16.3 Transfers to R15

When a coprocessor register transfer to ARM810 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.

4.16.4 Instruction cycle times

Both the MRC and MCR instructions take 1 cycle to execute, provided that the coprocessor does not “busy-wait” them.

4.16.5 Assembler syntax

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

where:

`MRC`	move from coprocessor to ARM810 register (L=1)
`MCR`	move from ARM810 register to coprocessor (L=0)
`{cond}`	two-character condition mnemonic, see ***Figure 4-2: Condition codes*** on page 4-3
`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 ARM810 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

4.16.6 Examples

```
MRC    p2,5,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    p6,0,R4,c6,c7   ; request coproc 6 to perform operation 0
```

```
                                             ; on R4 and place the result in c6, in a
                                             ; way that may be influenced by c7

                      MRCEQ  p3,9,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
```

4.17 The Instruction Memory Barrier (IMB) Instruction

An Instruction Memory Barrier (IMB) Instruction is used to ensure that correct instruction flow occurs after instruction memory locations are altered in any way - by self-modifying code for example. The recommended implementation of the IMB instructions is via an architecturally defined SWI function (see ***4.13 Software Interrupt (SWI)*** on page 4-52). The instruction encoding for the recommended IMB instruction implementations is shown below:

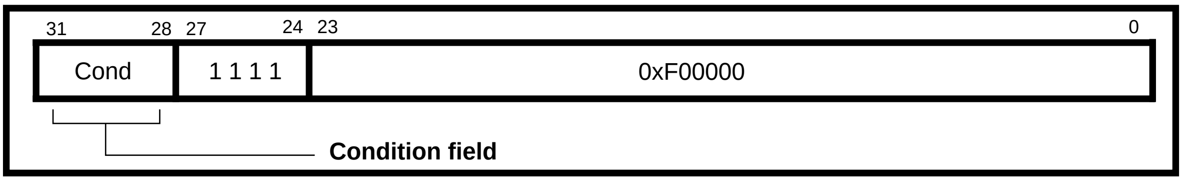


Figure 4-30: IMB instruction

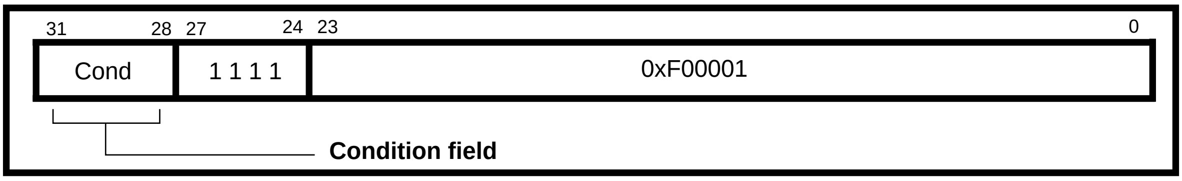


Figure 4-31: IMBRange instruction

IMBRange: Registers R0 and R1 contain the Range of addresses on entry to the SWI. R0 is the lower (inclusive) address and R1 is the upper address (not included in the range).

4.17.1 Use

During the normal operation of ARM8, the Prefetch Unit (PU) reads instructions ahead of the core in order to attempt to remove branches. It does this by predicting whether or not the branches are taken and then prefetching from the predicted address.

If a program changes the contents of memory with the intention of executing the new contents as new instructions, then any prefetched instructions and/or other stored information about instructions in the PU may be out of date because the instructions concerned have been overwritten. Thus the PU holds the wrong instructions; if passed to the execution unit they would cause unintentional behaviour.

In order to prevent such problems, an IMB instruction must be used between changing the contents of memory and executing the new contents to ensure that any stored instructions are flushed from the PU. The choice of IMB Instruction (IMB or IMBRange) depends upon the amount of code changed.

The IMB Instruction flushes all stored information about the instruction stream.

The IMBRange Instruction flushed all stored information about instructions at addresses in the range specified.

Please refer to ***Appendix E, Implementing the Instruction Memory Barrier Instruction*** for further details of the IMB implementation and use.

4.17.2 Assember syntax

```
SWI{cond} IMB          ; Where IMB = 0xF00000

; code that loads R0 and R1 with Range addresses
SWI{cond} IMBRange     ; Where IMBRange = 0xF00001
```

4.17.3 Examples

Loading code from disc

Code that loads a program from a disc, and then branches to the entry point of that program, should execute an IMB instruction between loading the program and trying to execute it.

```
IMB EQU    0xF00000
    .
    .
    ; code that loads program from disc
    .
    .
    SWI    IMB
    .
    .
    MOV    PC, entry_point_of_loaded_program
    .
    .
```

Running BitBlt code

“Compiled BitBlt” routines optimise large copy operations by constructing and executing a copying loop which has been optimised for the exact operation wanted.

When writing such a routine an IMB is needed between the code that constructs the loop and the actual execution of the constructed loop.

```
IMBRange EQU     0xF00001
         .
         .
         ; code that constructs loop code
         ; load R0 with start address of the constructed loop
         ; load R1 with the end address of the constructed loop
         SWI     IMBRange
         ; start of constructed loop code
         .
         .
```

Self-decompressing code

When writing a self-decompressing program, an IMB should be issued after the routine which decompresses the bulk of the code and before the decompressed code starts to be executed.

```
IMB EQU    0xF00000
    .
    .
    ; copy and decompress bulk of code
    SWI    IMB
    ; start of decompressed code
```

4.18 Undefined Instructions

This section shows the instruction bit patterns that will cause the Undefined Instruction trap to be taken if ARM810 attempts to execute them. This vector location is defined in ***3.6.6 Exception vector summary*** on page 3-11. There are a number of such bit pattern classes, and these can be used to cause unimplemented instructions (for example LDC) to be emulated through the Undefined Instruction trap service routine code:

Class A	Undefined instructions in previous ARM processor implementations
Class B	Unallocated MSR/MRS-like instructions
Class C	Unallocated Multiply-like instructions
Class D	Unallocated SWP-like instructions
Class E	Unallocated STRH/LDRH/LDRSH/LDRSB-like instructions

Note *Some or all of Classes B through E may not fall into the Undefined Instruction trap if further implementation restrictions dictate this. ARM reserves the right to make these decisions as necessary.*

Class	Instruction Bit Pattern								Notes
A	Cond	011x	xxxx	xxxx	xxxx	xxxx	xxx1	xxxx	
B	Cond	0001	0xx0	xxxx	xxxx	xxxx	yyy0	xxxx	yyy != 000
	Cond	0001	0xx0	xxxx	xxxx	xxxx	0xx1	xxxx	
	Cond	0011	0x00	xxxx	xxxx	xxxx	xxxx	xxxx	
C	Cond	0000	01xx	xxxx	xxxx	xxxx	1001	xxxx	
D	Cond	0001	yyyy	xxxx	xxxx	xxxx	1001	xxxx	yyyy !=0000 or 0100
E	Cond	0000	xx1x	xxxx	xxxx	xxxx	1yy1	xxxx	yy !=00
	Cond	000x	xxx0	xxxx	xxxx	xxxx	11x1	xxxx	

Table 4-6: Bit patterns for the undefined instruction trap

The Undefined Instruction trap is taken:

- if the condition specified by Cond is met and the instruction bit pattern is in ***Table 4-6: Bit patterns for the undefined instruction trap***

or

- by all coprocessor instructions whose condition is met and which are bounced by any coprocessor. For ARM810, the coprocessor interface must bounce all CDP, LDC and STC instructions

4.18.1 Assembler syntax

At present the assembler has no mnemonics for generating Undefined Instruction classes A through to E.

4.19 Instruction Set Examples

The following examples show ways in which the basic ARM810 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.

4.19.1 Using the conditional instructions

Using conditionals for logical OR

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

can be replaced by :

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

Absolute value

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

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
```

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:="."
```

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.

```
; Unsigned divide of r1 by r0
; Returns quotient in r0, remainder in r1
; Destroys r2, r3
       MOV     r3, #0
       MOVS    r2, r0
       BEQ     |__rt_div0|    ; jump to divide-by-zero
                              ; error handler
; justification stage shifts r2 left 1 bit at a time
; until r2 > (r1/2)
u_loop
       CMP     r2, r1, LSR #1
       MOVLS   r2, r2, LSL #1
       BCC     u_loop
; now division proper can start
u_loop2
       CMP     r1, r2             ; perform divide step
       ADC     r3, r3, r3
       SUBCS   r1, r1, r2
       TEQ     r2, r0             ; all done yet?
       MOVNE   r2, r2, LSR #1
       BNE     u_loop2
       MOV     r0, r3
```

4.19.2 Multiply overflow detection in the ARM810

Overflow in unsigned multiply with a 32 bit result

```
UMULL    Rd,Rt,Rm,Rn    ;4 to 7 cycles
TEQ      Rt,#0          ;+1 cycle and a register
BNE      overflow
```

Overflow in signed multiply with a 32 bit result

```
SMULL    Rd,Rt,Rm,Rn    ;4 to 7 cycles
TEQ      Rt,Rd ASR#31   ;+1 cycle and a register
BNE      overflow
```

Overflow in unsigned multiply accumulate with a 32 bit result

```
UMLAL    Rd,Rt,Rm,Rn    ;4 to 7 cycles
TEQ      Rt,#0          ;+1 cycle and a register
BNE      overflow
```

Overflow in signed multiply accumulate with a 32 bit result

```
SMLAL    Rd,Rt,Rm,Rn    ;4 to 7 cycles
TEQ      Rt,Rd, ASR#31  ;+1 cycle and a register
BNE      overflow
```

Overflow in unsigned multiply accumulate with a 64 bit result

```
SMULL    R1,Rh,Rm,Rn    ;4 to 7 cycles
ADDS     Rl,Rl,Ra1      ;lower accumulate
ADCS     Rh,Rh,Ra2      ;upper accumulate
BCS      overflow       ;2 cycles and 2 registers
```

Overflow in signed multiply accumulate with a 64 bit result

```
UMULL    R1,Rh,Rm,Rn    ;4 to 7 cycles
ADDS     Rl,Rl,Ra1      ;lower accumulate
ADCS     Rh,Rh,Ra2      ;upper accumulate
BVS      overflow       ;2 cycles and 2 registers
```

Note Overflow cannot occur in signed and unsigned multiply with a 64-bit result, so overflow checking is not applicable.

4.19.3 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 of maximal length (ie. 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 (ie. 32 bits).

```
                     ; enter with seed in Ra (32 bits),
                     ; Rb (1 bit in Rb lsb), uses Rc

TST Rb,Rb,LSR#1      ; top bit into carry
MOVSRc,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
```

4.19.4 Multiplication by constant using shifts

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
```

4.19.5 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
LDMIARb,{Rd,Rc}     ; get 64 bits containing answer
AND Rb,Ra,#3        ; correction factor in bytes
MOVSRb,Rb,LSL#3     ; ...now in bits and test if aligned
MOVNERd,Rd,LSR Rb   ; produce bottom of result word
                    ; (if not aligned) for little-endian
                    ; operations (see note below)
RSBNERb,Rb,#32      ; get other shift amount
ORRNERd,Rd,Rc,LSL Rb; combine two halves to get result
                    ; for little-endian operation (see note
                    ; below)
```

Note: for Big-endian operation replace the first “LSR” with “LSL” and the final “LSL” by “LSR”.

5 Configuration

This chapter describes the configuration.

5.1 ARM810 System Control Coprocessor (CP15) Register Map 5-3

Configuration

The operation and configuration of ARM810 is controlled both directly via coprocessor instructions and indirectly via the Memory Management Page tables. The coprocessor instructions manipulate a number of on-chip registers which control the configuration of the Cache, write buffer, MMU and a number of other configuration options.

5.1 ARM810 System Control Coprocessor (CP15) Register Map

5.1.1 CP15 registers

CP15 defines 16 registers. ***Table 5-1: CP15 register summary*** on page 5-4 shows which registers are defined for reading and which for writing. All CP15 register bits which are defined and contain state are set to zero by Reset.

CP15 registers can only be accessed with MRC and MCR instructions in a Privileged mode. The instruction bit pattern of the MCR and MRC instructions is shown below:

31 28	27 24	23 21	20	19 16	15 12	11 8	7 5	4	3 0
Cond	1 1 1 0	opcode_1	L	CRn	Rd	1 1 1 1	opcode_2	1	Crm

Figure 5-1: MRC, MCR bit pattern

CDP, LDC and STC instructions, along with unprivileged MRC and MCR instructions to CP15 will cause the undefined instruction trap to be taken. The CRn field of MRC and MCR instructions specify the coprocessor register to access. The CRm field and opcode_2 field are used to specify a particular action when addressing some registers.

Attempting to read from a register which is not defined for reading, or writing to a register which is not defined for writing will cause the instruction to take the undefined instruction trap. See ***5.1.2 Architectural Compliance of ARM810 CP15*** on page 5-12. In all instructions which access CP15:

- the opcode_1 field SHOULD BE ZERO
- the opcode_2 and CRm fields SHOULD BE ZERO except when accessing registers 7 and 8, when the values specified below should be used to select the desired Cache and TLB operations. Using a value other than those specified below for opcode_2 and CRm when accessing registers 7 and 8, or other than zero when accessing other registers, will cause ARM810 to take the undefined instruction trap. See ***5.1.2 Architectural Compliance of ARM810 CP15*** on page 5-12.

Throughout this section the following terms and abbreviations are used:

UNPREDICTABLE	UNP	If specified for reads: the data returned when reading from this location is unpredictable - it could have any value. If specified for writes: writing to this location will cause unpredictable behaviour or an unpredictable change in device configuration.
UNDEFINED	UND	An instruction that accesses CP15 in the manner indicated will take the undefined instruction trap.
SHOULD BE ZERO	SBZ	When writing to this location, all bits of this field should be 0.

In all cases, reading from, or writing any data values to any CP15 registers, including those fields specified as UNPREDICTABLE or SHOULD BE ZERO will not cause any permanent damage to the ARM810.

Register	Reads	Writes
0	ID Register	UNDEFINED
1	Control	Control
2	Translation Table Base	Translation Table Base
3	Domain Access Control	Domain Access Control
4	UNDEFINED	UNDEFINED
5	Fault Status	Fault Status
6	Fault Address	Fault Address
7	UNDEFINED	Cache operations
8	UNDEFINED	TLB operations
9	Cache Lock-Down	Cache Lock-Down
10	TLB Lock-Down	TLB Lock-Down
11 to 14	UNDEFINED	UNDEFINED
15	Clock and Test Configuration	Clock and Test Configuration

Table 5-1: CP15 register summary

Register 0: ID register

Reading from CP15 register 0 returns the value 0x4101810x. The CRm and opcode_2 fields SHOULD BE ZERO when reading CP15 register 0.

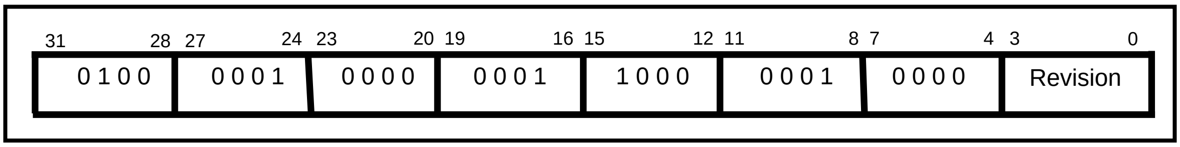


Figure 5-2: ID register read

Writing to CP15 register 0 is UNPREDICTABLE.

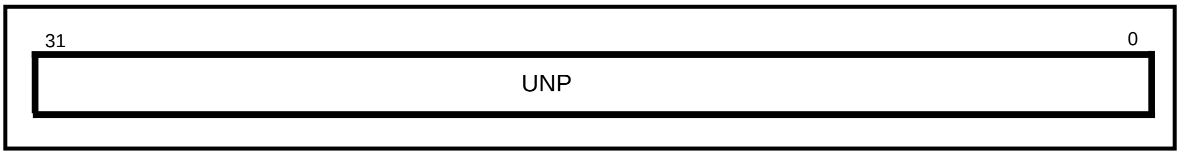


Figure 5-3: ID register write

Register 1: Control register

Reading from CP15 register 1 reads the control bits. The CRm and opcode_2 fields SHOULD BE ZERO when reading CP15 register 1.

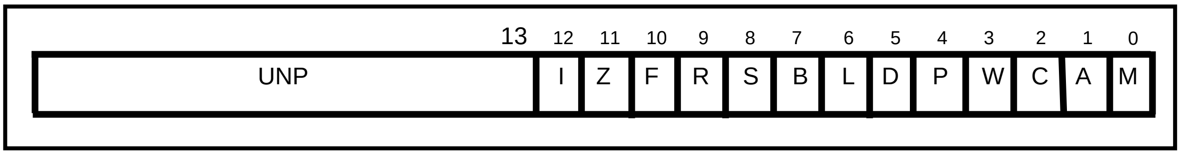


Figure 5-4: Register 1 read

Writing to CP15 register 1 sets the control bits. The CRm and opcode_2 fields SHOULD BE ZERO when writing CP15 register 1.

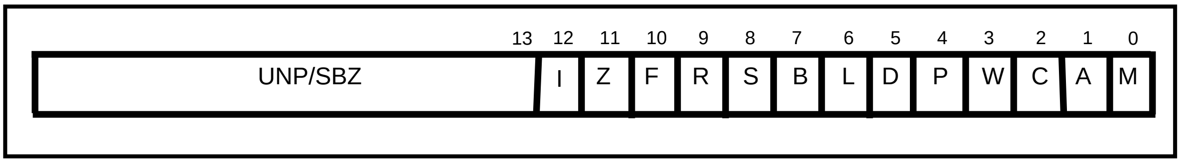


Figure 5-5: Register 1 write

All defined control bits are set to zero on reset. The control bits have the following functions:

M Bit 0	MMU Enable/Disable 0 = Memory Management Unit (MMU) disabled 1 = Memory Management Unit (MMU) enabled
A Bit 1	Alignment Fault Enable/Disable 0 = Address Alignment Fault Checking disabled 1 = Address Alignment Fault Checking enabled
C Bit 2	Cache Enable/Disable 0 = Instruction and/or Data Cache (IDC) disabled 1 = Instruction and/or Data Cache (IDC) enabled
W Bit 3	Write buffer Enable/Disable 0 = Write Buffer disabled 1 = Write Buffer enabled
P Bit 4	When read returns one, and when written is ignored.
D Bit 5	When read returns one, and when written is ignored.
L Bit 6	When read returns one, and when written is ignored.
B Bit 7	Big-endian/Little-endian 0 = Little-endian operation 1 = Big-endian operation
S Bit 8	System protection This bit modifies the MMU protection system.
R Bit 9	ROM protection This bit modifies the MMU protection system.
F Bit 10	When read returns zero. When written SHOULD BE ZERO.

Z Bit 11 Branch Prediction Enable/Disable
0 = Branch Prediction Disabled
1 = Branch Prediction Enabled.

I Bit 12 When read returns zero. When written SHOULD BE ZERO.

Bits 31:13 When read returns an UNPREDICTABLE value, and when written SHOULD BE ZERO, or a value read from these bits on the same processor. Note that using a read-write-modify sequence when modifying this register provides the greatest future compatibility.

Enabling the MMU

Care must be taken if the translated address differs from the untranslated address as the instructions following the enabling of the MMU will have been fetched using no address translation and enabling the MMU may be considered as a branch with delayed execution. A similar situation occurs when the MMU is disabled. The correct code sequence for enabling and disabling the MMU is implementation defined

If the cache and write buffer are enabled when the MMU is not enabled, the results are UNPREDICTABLE.

Register 2: Translation table base register

Reading from CP15 register 2 returns the pointer to the currently active first level translation table in bits[31:14] and an UNPREDICTABLE value in bits[13:0].The CRm and opcode_2 fields SHOULD BE ZERO when reading CP15 register 2.

Writing to CP15 register 2 updates the pointer to the currently active first level translation table from the value in bits[31:14] of the written value. Bits[13:0] SHOULD BE ZERO. The CRm and opcode_2 fields SHOULD BE ZERO when writing CP15 register 2.

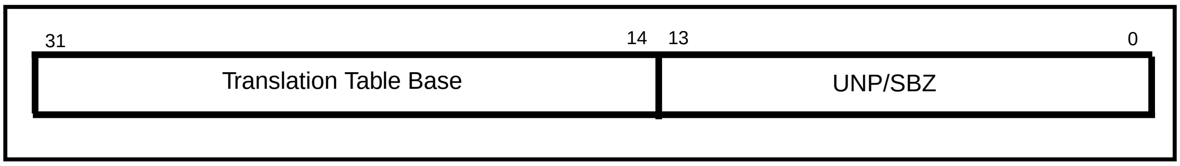


Figure 5-6: Register 2

Register 3: Domain access control register

Reading from CP15 register 3 returns the value of the Domain Access Control Register.

Writing to CP15 register 3 writes the value of Domain Access Control Register.

The Domain Access Control Register consists of sixteen 2-bit fields, each of which defines the access permissions for one of the sixteen Domains (D15-D0).

The CRm and opcode_2 fields SHOULD BE ZERO when reading or writing CP15 register 3.

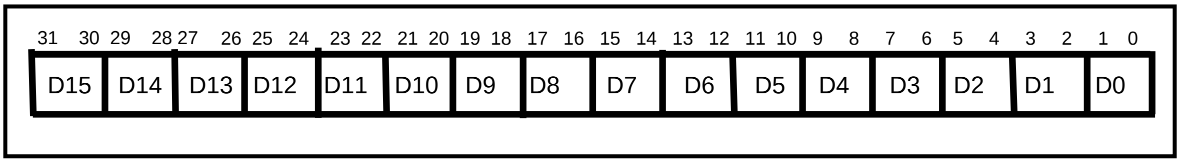


Figure 5-7: Register 3

Register 4: Reserved

Register 4 is reserved. Reading CP15 register 4 is UNDEFINED. Writing CP15 register 4 is UNDEFINED.

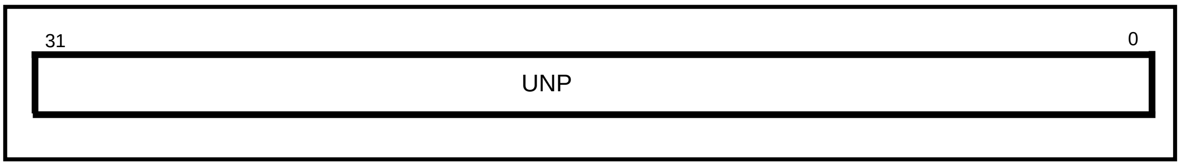


Figure 5-8: Register 4

Register 5: Fault Status Register

Reading CP15 register 5 returns the value of the Fault Status Register (FSR). The FSR contains the source of the last data fault. Note that only the bottom 9 bits are returned. The upper 23 bits are UNPREDICTABLE. The FSR indicates the domain and type of access being attempted when an abort occurred. Bit 8 is always read as zero. Bits 7:4 specify which of the sixteen domains (D15-D0) was being accessed when a fault occurred. Bits 3:1 indicate the type of access being attempted. The encoding of these bits is shown in ***8.13 Fault Address and Fault Status Registers (FAR and FSR)*** on page 8-17. The FSR is only updated for data faults, not for prefetch faults.

Writing CP15 register 5 sets the Fault Status Register to the value of the data written. This is useful for a debugger to restore the value of the FSR. The upper 24 bits written SHOULD BE ZERO. Bit 8 is ignored on writes.

The CRm and opcode_2 fields SHOULD BE ZERO when reading or writing CP15 register 5.

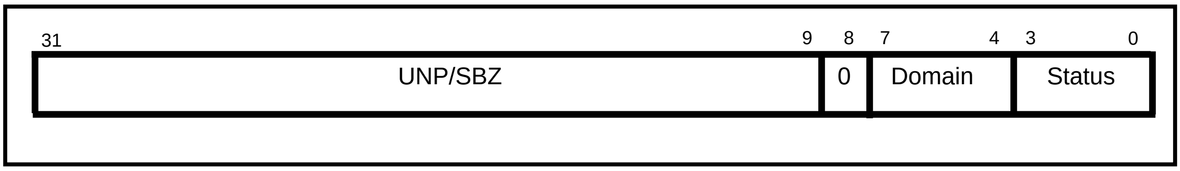


Figure 5-9: Register 5

Register 6: Fault Address Register

Reading CP15 register 6 returns the value of the Fault Address Register (FAR). The FAR holds the virtual address of the access which was attempted when a fault occurred. The FAR is only updated for data faults, not for prefetch faults.

Writing CP15 register 6 sets the Fault Address Register to the value of the data written. This is useful for a debugger to restore the value of the FAR.

The CRm and opcode_2 fields SHOULD BE ZERO when reading or writing CP15 register 6.

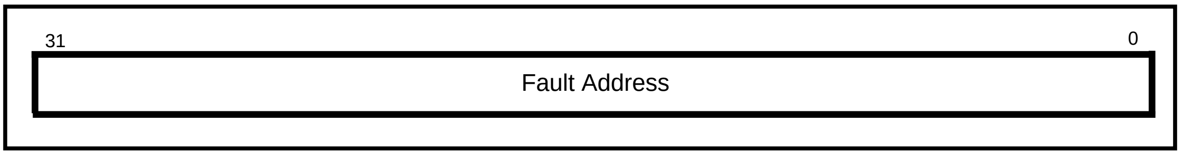


Figure 5-10: Register 6

Register 7: Cache Operations

Writing to CP15 register 7 is used to manage the ARM810 unified instruction and data cache. Four cache operations are defined, and the function to be performed selected by the opcode_2 and CRm fields in the MCR instruction used to write CP15 register 7.

Function	opcode_2 value	CRm value	Data	Instruction
Invalidate ID cache	0b000	0b0111	SBZ	MCR p15, 0, Rd, c7, c7, 0
Invalidate ID single entry	0b001	0b0111	Index, Seg Format	MCR p15, 0, Rd, c7, c7, 1
Clean ID single entry	0b001	0b1011	Index, Seg Format	MCR p15, 0, Rd, c7, c11, 1
Clean and Invalidate ID entry	0b001	0b1111	Index, Seg Format	MCR p15, 0, Rd, c7, c15, 1

Table 5-2: Cache operations

Reading from CP15 register 7 is UNDEFINED.

The “Invalidate ID cache” function invalidates all cache data, including any dirty data (data which has been modified in the cache but not yet written to main memory). Use with caution.

The “Invalidate ID single entry” function invalidates a single cache line, discarding any dirty data (data which has been modified in the cache but not yet written to main memory). Use with caution.

The “Clean ID single entry” function writes the specified cache line to main memory if the line is marked Valid and Dirty, and marks the line as not-Dirty . The Valid bit is unchanged.

The “Clean and Invalidate ID entry” function writes the specified cache line to main memory if the line is marked Valid and Dirty. It always invalidates the line.

The operations which operate upon a single cache line accept the entry's Index and Segment number as the data passed in the MCR instruction in the following format:

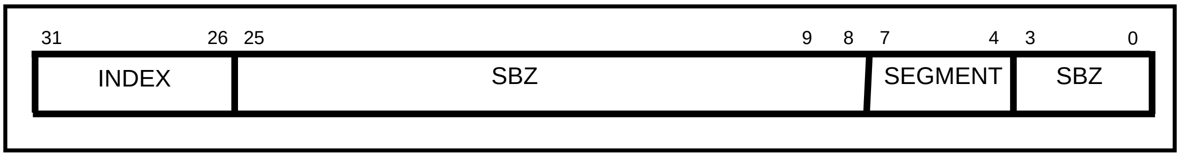


Figure 5-11: Register 7

See ***Chapter 7, Instruction and Data Cache (IDC)*** for discussion of the use of these operations.

Register 8: TLB Operations

Writing to CP15 register 8 is used to control Translation Lookaside Buffers (TLBs). The ARM810 implements a unified instruction and data TLB.

Two TLB operations are defined, and the function to be performed selected by the opcode_2 and CRm fields in the MCR instruction used to write CP15 register 8.

Function	opcode_2 value	CRm value	Data	Instruction
Invalidate TLB	0b000	0b0111	SBZ	MCR p15, 0, Rd, c8, c7, 0
Invalidate TLB single entry	0b001	0b0111	Virtual Address	MCR p15, 0, Rd, c8, c7, 1

Table 5-3: TLB operations

Reading from CP15 register 8 is UNDEFINED.

The “Invalidate TLB” function invalidates all of the unlocked entries in the TLB

The “Invalidate TLB single entry” function invalidates any TLB entry corresponding to the Virtual Address given in Rd, regardless of it’s lock-down state.

Register 9: Cache Lock-Down

Writing CP15 register 9 updates the Cache Lock-Down control register. Bits 30:6 SHOULD BE ZERO when written.

Reading CP15 register 9 returns the value of the Cache Lock-Down control register. Note that only bit 31 and bits 5:0 are returned. Bits 30:6 are UNPREDICTABLE when read.

The Cache Lock-Down control register allows software to load entries into the Cache and lock them in. See ***7.7 Lock-down Features*** on page 7-3.

The CRm and opcode_2 fields SHOULD BE ZERO when reading or writing CP15 register 9.

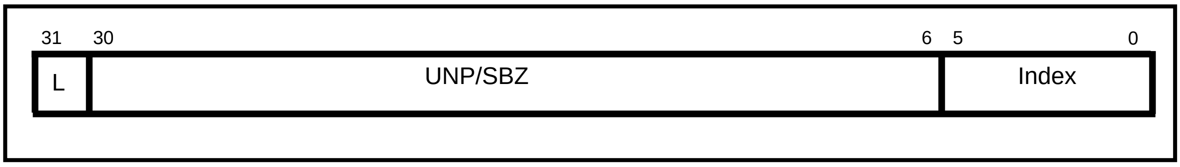


Figure 5-12: Register 9

L Bit 31 Cache Load Entry Mode
0 = Normal operation - Index Field specifies number of lock-down Indexes
1 = Load Entry Mode - Index Field specifies Index number to load into.

Register 10: TLB Lock-Down

Writing CP15 register 10 updates the TLB Lock-Down control register. Bits 30:6 SHOULD BE ZERO when written.

Reading CP15 register 10 returns the value of the TLB Lock-Down control register. Note that only bit 31 and bits 5:0 are returned. Bits 30:6 are UNPREDICTABLE when read.

The TLB Lock-Down control register allows software to load entries into the TLB and lock them in. See ***Appendix F, Cache and TLB Lock-Down Features***.

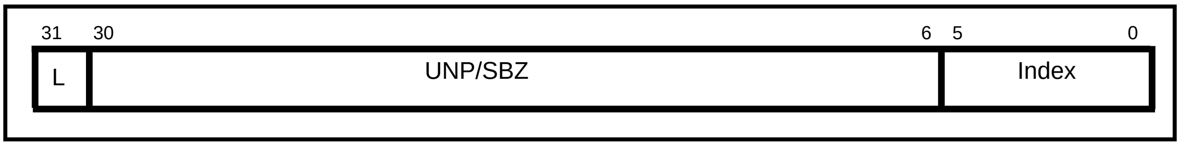


Figure 5-13: Register 10

L Bit 31TLB Load Entry Mode
0 = Normal operation - Index Field specifies number of lock-down
 Indexes.The number of lock-down Indexes must be 0 or 4.
1 = Load Entry Mode - Index Field specifies Index number to load into.

Registers 11 -14: Reserved

Accessing (reading or writing) any of these registers will cause ARM810 to take the undefined instruction trap.

Register 15: Clock and Test Configuration

Register 15 contains clocking configuration bits, test configuration bits, and the PLL Locked status bit. Writing to CP15 register 15 writes to the configuration bits. Writing a 1 to the PLL Locked status bit resets the PLL status bit for subsequent reads - see below for details.

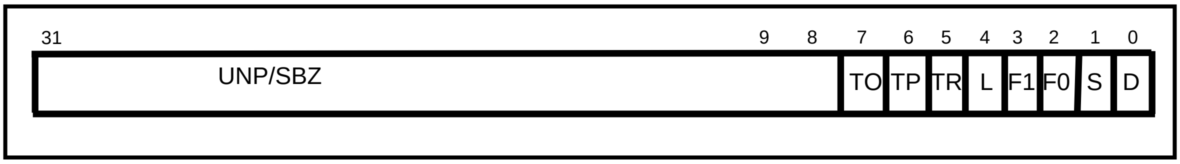


Figure 5-14: Register 15

All defined bits are set to zero on reset. The register bits have the following functions:

D Bit 0 Enable Dynamic Clock Switching

0 Dynamic clock switching is disabled, clock synchroniser will permanently select the bus clock as the source of the processor clock.

1 Dynamic clock switching is enabled, clock synchroniser will dynamically switch between the fast clock and the bus clock as the source of the processor clock as processor access to the Bus Interface is required.

S Bit 1 Synchronous Clock Switching

0 Clock synchroniser operates in asychronous mode. Use this setting if the fast clock and the bus clock *do not* obey the requirements specified in the AC parameters section for synchronous mode operation.

1 Clock synchroniser operates in synchronous mode.
Use this setting if the fast clock and the bus clock *do* obey the requirements specified in the AC parameters section for synchronous mode operation.

F1, F0 Bits 3, 2 Fast clock source configuration

F1 = 0 F0 = 0 bus clock (**MCLK** or **PCLK**) is the fast clock source.

F1 = 0 F0 = 1 **REFCLK** pin is the fast clock source.

F1 = 1 F0 = 0 Reserved. Do not use.

F1 = 1 F0 = 1 PLL output clock is fast clock source.

L Bit 4 PLL Locked indication

When Reading:

L = 1 indicates that the PLL output clock is within a small range of the target frequency.

When Writing:

Writing L = 0 is ignored.

Writing L = 1 resets the PLL Lock Detect circuitry. Following such a reset, reading the register will return L = 0 until the Lock Detect circuit again detects that the PLL output clock is within a small range of the target frequency. This is useful in systems which stop **REFCLK**, or change the frequency applied to the **REFCLK** pin, or change the PLL configuration pins under program control

Note Logic external to ARM810 would be required to implement such features.

TR, TP, and TO (Bits 5, 6, 7 and 8) are configuration bits for test features used in device production test. These bits must all be written as zero for normal device operation.

5.1.2 Architectural Compliance of ARM810 CP15

The ARM810 Coprocessor 15 complies with the definition of the ARMv4 System Control Coprocessor given in the ARM Architecture Reference (ARM DDI 0100) with the following exceptions and clarifications:

- Registers 9, 10, and 15 are not defined in the ARM Architecture Manual and should be considered implementation specific extensions to the CP15 definition.
- The ARM Architecture Reference defines read accesses to registers not defined for reading, and write accesses to registers not defined for writing, as UNPREDICTABLE. ARM810 implements these as UNDEFINED - ie, executing coprocessor instructions which attempt such accesses will cause ARM810 to take the Undefined Instruction Trap.
- The ARM Architecture Reference defines that instructions which access register 7 and 8 and which specify values of opcode_2 or CRm which do not specify an implemented operation should be IGNORED. ARM810 implements these as UNDEFINED.
- The ARM Architecture Reference defines that instructions which access register other than 7 and 8 and which specify values of opcode_2 or CRm other than zero are UNPREDICTABLE. ARM810 implements these as UNDEFINED.

6 The Prefetch Unit

This chapter describes the functions of the prefetch unit.

6.1 Overview 6-2

6.2 The Prefetch Buffer 6-2

6.3 Branch Prediction 6-3

The Prefetch Unit

6.1 Overview

The ARM8 Prefetch Unit (PU) supplies the ARM8 Core with instructions from the memory system. The bus from the memory system to the PU is 32 bits wide but can supply two words every clock cycle. The memory system bandwidth is therefore greater than the bandwidth requirement of the Core. The Prefetch Unit makes use of this fact by buffering instructions in its FIFO and then predicting some of the branches and removing them from the instruction stream to the Core. This reduces the CPI of the Branch instruction, so increasing the processor’s performance.

The Prefetch Unit is responsible for fetching and supplying instructions to the Core, and has its own PC and incrementer to provide the memory system address.

6.2 The Prefetch Buffer

Each 32-bit instruction is buffered together with its (offset) address in a FIFO. The depth of this buffer is 8 instructions. At the far end of the FIFO, the instructions are removed one at a time and presented to the Core.

6.3 Branch Prediction

ARM810 employs static branch prediction. This is based solely on the characteristics of a Branch instruction, and uses no history information. Branch prediction is performed only when the Z bit in CP15 register 1 is set to 1 (see ***Chapter 5, Configuration***).

In ARM processors that have no Prefetch Unit, the target of a Branch is not known until the end of the Execute stage; at which time it is known whether or not the Branch will be taken. The best performance is therefore obtained by predicting all Branches as *not* taken, and filling the pipeline with the instructions that follow the Branch. In this type of Core, an untaken Branch requires 1 cycle and a taken Branch requires 3 cycles.

By adding a Prefetch Buffer, it is possible to detect a Branch *before* it enters the Core. This allows the use of a different prediction scheme - for instance, one which predicts that all *forward* Branches are not taken and all *backward* Branches are taken. This scheme is the one implemented in ARM810 and because it models actual conditional branch behaviour more accurately, it reduces the average branch CPI, thus improving the processor’s performance.

Using ARM8’s Prefetch Unit, around 65% of all Branches are preceded by enough non-Branch cycles to be completely predicted. The Core itself deals with the Branches that the Prefetch Unit does not have time to predict.

6.3.1 Incorrect predictions and correction

Whenever a potentially incorrect prediction is made, information necessary for recovering from the error is stored. This is the fall-through address in the case of a predicted taken Branch, and the Branch’s target address in the case of a predicted not taken Branch.

The Prefetch Unit uses the Core’s condition codes to establish the accuracy of a prediction. If the prediction is found to be in error, the Prefetch Unit begins fetching from the saved alternate address, and cancels any instructions that have been incorrectly passed to the core.

6.3.2 Prediction details

This section describes the conditions under which prediction is made, and the result of the prediction based upon the direction of the branch.

BL is only predicted if it is an unconditional instruction. When predicted, the instruction is effectively changed into a link instruction and a branch instruction. The link part of the instruction is passed to the core as a special MOV instruction, and the branch part is predicted with the same rules as for the prediction of normal B instructions.

The following summarises the prediction scheme:

If any instruction is not predicted, then it is passed straight through to the core without change.

Instructions will not be predicted if any of the following conditions apply:

- Z bit in CP15 register 1 is 0
- Instruction[27:24]=“1011” AND Instruction[31:28]!=“1110” (Conditional BL)
- A prefetch abort occurs when fetching the instruction

- Instruction[31:28]==“1111” (Invalid condition code)
- Instruction[27:25]!=“101” (Non-branch instruction)

otherwise the instruction will be predicted as taken if:

- Instruction[31:28]==“1110” (Always condition code)
- Instruction[24]==“0” AND Instruction[23]==“1” (Backwards branch)

otherwise the instruction will be predicted as not-taken if:

- Instruction[24]==“0” AND Instruction[23]==“0” (Forwards branch)

Consequences of branch prediction and the prefetch buffer

Due to the speculative prefetching of instructions that the Prefetch Unit performs, it is possible for the prefetch buffer to contain incorrect instructions. In such circumstances the prefetch buffer must be flushed, and ARM8 provides a means to do this with the IMB instruction. Please refer to ***4.17 The Instruction Memory Barrier (IMB) Instruction*** on page 4-64 for details of when and how to use the IMB instruction.

6.3.3 Turning off Branch Prediction

Branch prediction is disabled when the Z bit in the control register is 0 (CP15 register 1, bit 11). Clearing the Z bit does not stop speculative prefetching for a branch that has already been predicted. Branch prediction must be disabled and speculative prefetching must have completed before you disable the cache. The following code sequence disables branch prediction, and makes sure that speculative prefetching has completed:

```
Branch_Predict_Off
            MRC    p15,0,R0,c0,c0          ;Clear Control Reg Z bit.
            BIC    R0,R0,#&00000800
            MCR    p15,0,R0,c0,c0
            MSR    CPSR_f, #0xF0000000     ;set carry flag
            BCC    Branch_Predict_Off      ;branch never taken
```

Code to disable the cache should follow this code.

7 Instruction and Data Cache (IDC)

This chapter describes Instruction and Data Cache.

7.1 Introduction 7-2
7.2 Cacheable Bit and Bufferable Bit 7-2
7.3 IDC Operation 7-2
7.4 IDC Validity 7-2
7.5 Read-Lock-Write 7-3
7.6 IDC Enable/Disable and Reset 7-3

Instruction and Data Cache (IDC)

7.1 Introduction

ARM810 contains an 8 Kb mixed instruction and data cache which supports both write-through and write-back (also known as copy-back) operation. The IDC has 512 lines of 16 bytes (4 words), arranged as a 64-way associative, virtually addressed cache. The IDC is always reloaded a line at a time (four words). It may be enabled or disabled via the ARM810 Control Register and is disabled on **nRESET**. The operation of the cache is further controlled by the *Cacheable* (C) and *Bufferable* (B) bits stored in the Memory Management Page Table (see ***Chapter 8, Memory Management Unit***). For this reason, in order to use the IDC, the MMU must be enabled. The two functions may however be enabled simultaneously, with a single write to the Control Register.

7.2 Cacheable Bit and Bufferable Bit

The **Cacheable** bit determines whether data being read may be placed in the IDC and used for subsequent read operations. Typically main memory will be marked as Cacheable to improve system performance, and I/O space as Non-cacheable to stop the data being stored in ARM810's cache. For example if the processor is polling a hardware flag in I/O space, it is important that the processor is forced to read data from the external peripheral, and not a copy of initial data held in the cache. The *Cacheable* bit can be configured for both pages and sections.

When the cacheable bit associated with a memory region is 1, all write acesses to that region are bufferable and the B bit determines whether the region is cached with write-through (B=0) or write-back (B=1) cache operation.

See ***8.11 Cacheable and Bufferable Status of Memory Regions*** on page 8-14.

7.3 IDC Operation

In the ARM810 the cache will be searched regardless of the state of the C bit, only reads that miss the cache will be affected. The only effect of setting the cacheable bit to 0 is to inhibit cache replacement from occuring. If the cache is disabled by clearing bit 2 of the CP15 Control Register, no searching of the cache occurs and all regions are treated as non-cacheable.

7.3.1 Cacheable reads C = 1

A linefetch of 4 words will be performed when a cache miss occurs in a cacheable area of memory and it will be randomly placed in a cache bank.

7.3.2 Uncacheable reads C = 0

An external memory access will be performed and the cache will not be written.

7.4 IDC Validity

The IDC operates with virtual addresses, so care must be taken to ensure that its contents remain consistent with the virtual to physical mappings performed by the Memory Management Unit. If the Memory Mappings are changed, the IDC validity must be ensured.

7.4.1 Doubly mapped space

Since the cache works with virtual addresses, it is assumed that every virtual address maps to a different physical address. If the same physical location is accessed by more than one virtual address, the cache cannot maintain consistency, since each virtual address will have a separate entry in the cache, and only one entry will be updated on a processor write operation. To avoid any cache inconsistencies, both doubly-mapped virtual addresses should be marked as uncacheable.

7.5 Read-Lock-Write

The IDC treats the Read-Locked-Write instruction as a special case. The read phase always forces a read of external memory, regardless of whether the data is contained in the cache. The write phase is treated as a normal write operation (and if the data is already in the cache, the cache will be updated). Externally the two phases are flagged as indivisible by asserting the **LOCK** signal.

7.6 IDC Enable/Disable and Reset

The IDC is automatically disabled and flushed on **nRESET**. Once enabled, cacheable read accesses will cause lines to be placed in the cache.

7.6.1 To enable the IDC

To enable the IDC, make sure that the MMU is enabled first by setting bit 0 in Control Register, then enable the IDC by setting bit 2 in Control Register. The MMU and IDC may be enabled simultaneously with a single control register write.

7.6.2 To disable the IDC

To disable the IDC clear bit 2 in the Control Register and perform a flush by writing to the flush register.

7.7 Lock-down Features

See ***Appendix F, Cache and TLB Lock-Down Features***.

8 Memory Management Unit

This chapter describes the *Memory Management Unit* (*MMU*).

8.1 MMU Program Accessible Registers 8-3
8.2 Address Translation 8-5
8.3 Translation Process 8-6
8.4 Level One Descriptor 8-7
8.5 Page Table Descriptor 8-8
8.6 Section Descriptor 8-9
8.7 Translating Section References 8-10
8.8 Level Two Descriptor 8-11
8.9 Translating Small Page References 8-12
8.10 Translating Large Page References 8-13
8.12 MMU Faults and CPU Aborts 8-16
8.13 Fault Address and Fault Status Registers (FAR and FSR) 8-17
8.14 Domain Access Control 8-19
8.15 Fault Checking Sequence 8-20
8.16 External Aborts 8-23
8.17 Interaction of the MMU, IDC and Write Buffer 8-24
8.18 Effect of Reset 8-25

Memory Management Unit

The Memory Management MMU performs two primary functions: it translates virtual addresses into physical addresses, and it controls memory access permissions. The MMU hardware required to perform these functions consists of a Translation Look-aside Buffer (TLB), access control logic, and translation table walking logic.

The MMU supports memory accesses based on Sections or Pages. Sections are comprised of 1MB blocks of memory. Two different page sizes are supported: Small Pages consist of 4KB blocks of memory and Large Pages consist of 64KB blocks of memory. (Large Pages are supported to allow mapping of a large region of memory while using only a single entry in the TLB). Additional access control mechanisms are extended within Small Pages to 1KB Sub-Pages and within Large Pages to 16KB Sub-Pages.

The MMU also supports the concept of domains - areas of memory that can be defined to possess individual access rights. The Domain Access Control Register is used to specify access rights for up to 16 separate domains.

The TLB caches 64 translated entries. During most memory accesses, the TLB provides the translation information to the access control logic.

If the TLB contains a translated entry for the virtual address, the access control logic determines whether access is permitted. If access is permitted and an off-chip access is required, the MMU outputs the appropriate physical address corresponding to the virtual address. If access is not permitted, the MMU signals the CPU to abort.

If the TLB misses (it does not contain a translated entry for the virtual address), the translation table walk hardware is invoked to retrieve the translation information from a translation table in physical memory. Once retrieved, the translation information is placed into the TLB, possibly overwriting an existing value. The entry to be overwritten is chosen by cycling sequentially through the TLB locations.

When the MMU is turned off (as happens on reset), the virtual address is output directly onto the physical address bus.

8.1 MMU Program Accessible Registers

The following ARM810 System Control Coprocessor (CP15) registers, in conjunction with page table descriptors stored in memory, determine the operation of the MMU.

Register	Number	Bits
Control Register	1	M,A,S,R
Translation Table Base	2	31 .. 14
Domain Access Control	3	31 .. 0
Fault Status	5	8 .. 0
Fault Address	6	31 .. 0
TLB Operations	8	31 .. 0
TLB Lock down Control	10	31 & 5 .. 0

Table 8-1: CP15 register functions

All of these registers except register 8 contain state and can be read using MRC instructions and written using MCR instructions. Registers 5 and 6 are also written by the MMU when a data abort is signaled to record the cause of, and address associated with, an Abort. Writing to Register 8 causes the MMU to perform one of the TLB operations “Invalidate TLB” or “Invalidate TLB Entry”. Register 8 does not contain state and cannot be read.

Depending on the coprocessor instruction used, writing to register 8 with an MCR instruction causes one of the TLB operations “Invalidate TLB” or “Invalidate TLB Entry” to be performed by the MMU. Register 8 does not contain state and cannot be read.

System Control Coprocessor is described in ***Chapter 5, Configuration***. The details of register format and the coprocessor instructions to access them are given there.

A brief description of these registers is provided below. Each register will be discussed in more detail within the section that describes its use.

The **Control Register** contains bits to enable the MMU (M bit), enable Alignment checks (A bit), and to control the access protection scheme (S bit and R bit).

The **Translation Table Base Register** hold the physical address of the base of the translation table maintained in main memory. Note that this base must reside on a 16KB boundary.

The **Domain Access Control Register** consists of sixteen 2-bit fields, each of which defines the access permissions for one of sixteen Domains (D15-D0).

The **Fault Status Register** indicates the cause of an abort and the domain number of the aborted access when a data abort occurs. Bits 7:4 specify which of the sixteen domains (D15-D0) was being accessed when a fault occurred. Bits 3:1 indicate the type of access being attempted. The encoding of these bits is shown in ***Table 8-6: Priority Encoding of Fault Status*** on page 8-17.

The **Fault Address Register** holds the virtual address associated with the access that caused with abort. See ***Table 8-6: Priority Encoding of Fault Status*** on page 8-17 for details of exactly what address is stored for each type of fault.

Writing to the **TLB Operations Register** causes the MMU to perform one of the TLB operations “Invalidate TLB” or “Invalidate TLB Entry” depending on the coprocessor instruction used. For details, see the description of Register 8 in ***Chapter 5, Configuration***.

The **TLB Lock-Down Control Register** allows specific page table entries to be locked into the TLB. Locking entries in the TLB guarantees that accesses to the locked page or section can proceed without incurring the time penalty of a translation table walk. This allows the execution latency for time-critical pieces of code such as interrupt handlers to be minimised. Use of the TLB lock down facilities is described in ***Chapter 7, Instruction and Data Cache (IDC)***.

8.2 Address Translation

The MMU translates virtual addresses generated by the CPU into physical addresses to access external memory, and also derives and checks the access permission. Translation information, which consists of both the address translation data and the access permission data, resides in a translation table located in physical memory. The MMU provides the logic needed to traverse this translation table, obtain the translated address, and check the access permission.

There are three routes by which the address translation (and hence permission check) takes place. The route taken depends on whether the address in question has been marked as a section-mapped access or a page-mapped access; and there are two sizes of page-mapped access (large pages and small pages). However, the translation process always starts out in the same way, as described below, with a Level One fetch. A section-mapped access only requires a Level One fetch, but a page-mapped access also requires a Level Two fetch.

8.3 Translation Process

8.3.1 Translation table base

The translation process is initiated when the on-chip TLB does not contain an entry for the requested virtual address. The Translation Table Base (TTB) Register points to the base of a table in physical memory which contains Section and/or Page descriptors. The 14 low-order bits of the TTB Register are set to zero as illustrated in ***Figure 8-1: Translation table base register***; the table must reside on a 16KB boundary.

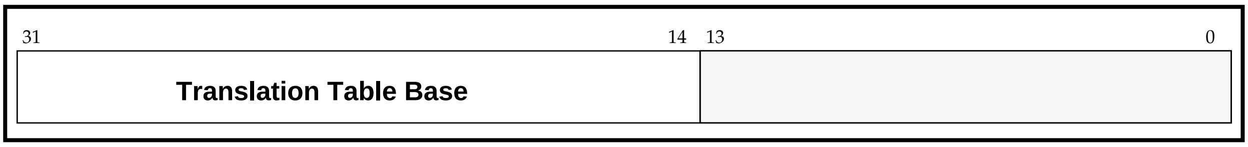


Figure 8-1: Translation table base register

8.3.2 Level one fetch

Bits 31:14 of the Translation Table Base register are concatenated with bits 31:20 of the virtual address to produce a 30-bit address as illustrated in ***Figure 8-2: Accessing the translation table first level descriptors***. This address selects a four-byte translation table entry which is a First Level Descriptor for either a Section or a Page (bit1 of the descriptor returned specifies whether it is for a Section or Page)

.

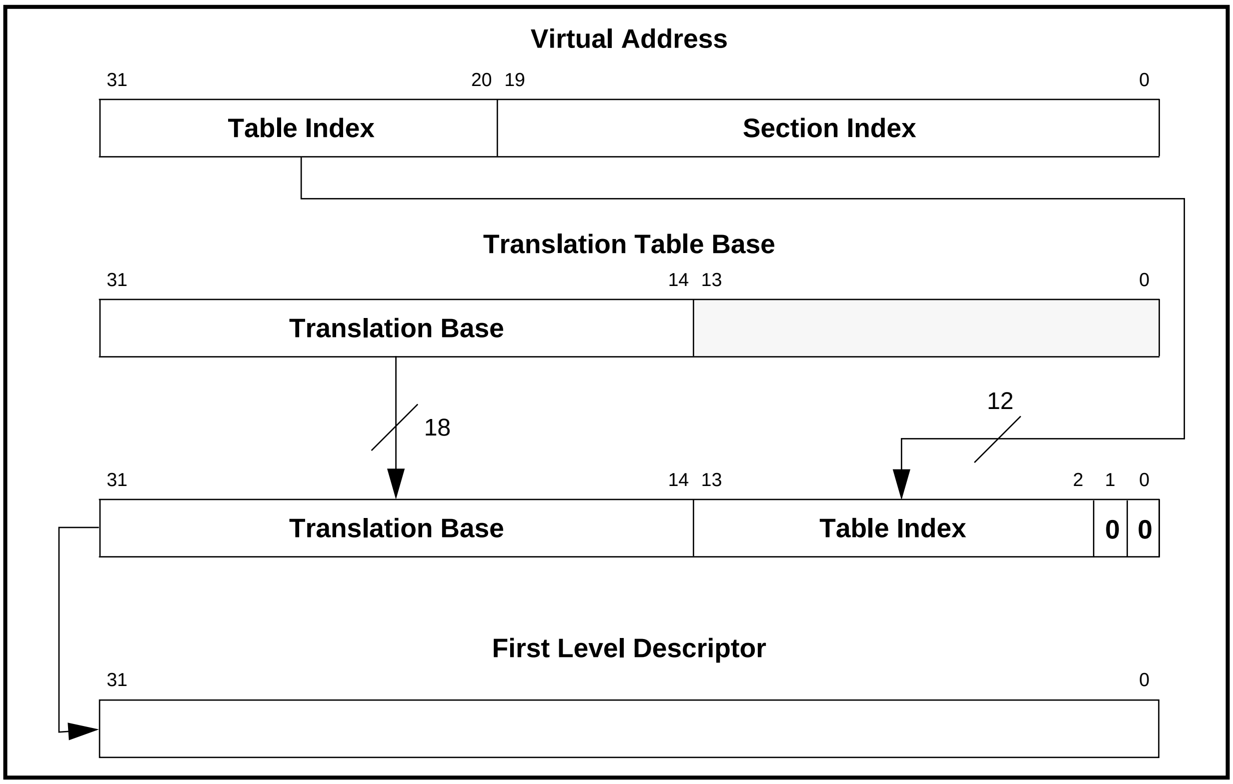


Figure 8-2: Accessing the translation table first level descriptors

8.4 Level One Descriptor

The Level One Descriptor returned is either a Page Table Descriptor or a Section Descriptor, and its format varies accordingly. The following figure illustrates the format of Level One Descriptors.

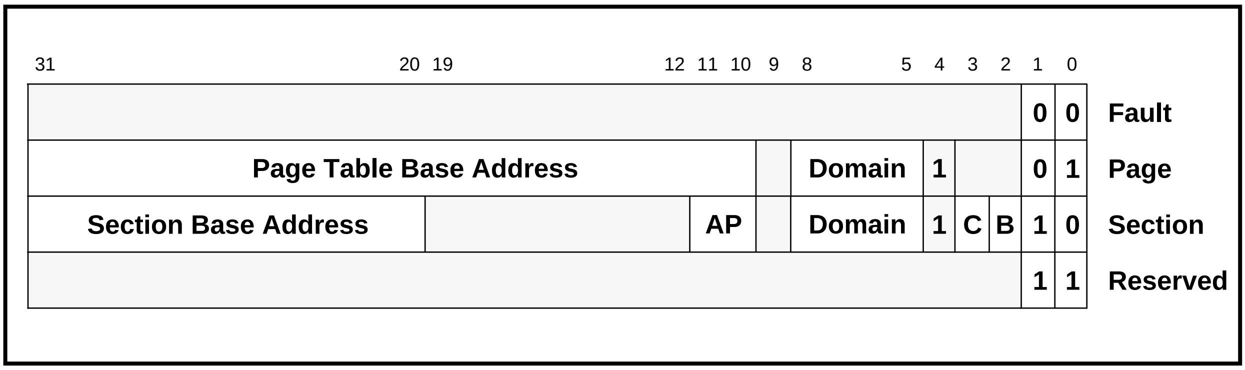


Figure 8-3: Level one descriptors

The two least significant bits indicate the descriptor type and valididty, and are interpreted as shown below..

Value	Meaning	Notes
0 0	Invalid	Generates a Section Translation Fault
0 1	Page	Indicates that this is a Page Descriptor
1 0	Section	Indicates that this is a Section Descriptor
1 1	Reserved	Reserved for future use

Table 8-2: Interpreting level one descriptor bits [1:0]

8.5 Page Table Descriptor

Bits 3:2 are always written as 0.

Bit 4 should be written to 1 for backward compatibility.

Bits 8:5 specify one of the sixteen possible domains (held in the Domain Access Control Register) that contain the primary access controls.

Bits 31:10 form the base for referencing the Page Table Entry. (The page table index for the entry is derived from the virtual address as illustrated in ***Figure 8-6: Small page translation*** on page 8-12).

If a Page Table Descriptor is returned from the Level One fetch, a Level Two fetch is initiated as described below.

8.6 Section Descriptor

Bits 3:2 (C, & B) The C & B bits together indicate whether the area of memory mapped by this section is treated as write-back cacheable, write-through cacheable, non cached buffered or non-cached non-buffered. Reference section 7.1.1 Cacheable and Bufferable Status of Memory Regions.

Bit 4 should be written to 1 for backward compatibility.

Bits 8:5 specify one of the sixteen possible domains (held in the Domain Access Control Register) that contain the primary access controls.

Bits 11:10 (AP) specify the access permissions for this section and are interpreted as shown in ***Table 8-3: Interpreting access permission (AP) Bits*** on page 8-9. Their interpretation is dependent upon the setting of the S and R bits (control register bits 8 and 9). Note that the Domain Access Control specifies the primary access control; the AP bits only have an effect in client mode. Refer to section on access permissions

AP	S	R	Permissions		Notes
			Supervisor	User	
00	0	0	No Access	No Access	Any access generates a permission fault
00	1	0	Read Only	No Access	Supervisor read only permitted
00	0	1	Read Only	Read Only	Any write generates a permission fault
00	1	1	Reserved		
01	x	x	Read/Write	No Access	Access allowed only in Supervisor mode
10	x	x	Read/Write	Read Only	Writes in User mode cause permission fault
11	x	x	Read/Write	Read/Write	All access types permitted in both modes.
xx	1	1	Reserved		

Table 8-3: Interpreting access permission (AP) Bits

Bits 19:12 are always written as 0.

Bits 31:20 form the corresponding bits of the physical address for the 1MByte section.

8.7 Translating Section References

Figure 8-4: Section translation illustrates the complete Section translation sequence. Note that the access permissions contained in the Level One Descriptor must be checked before the physical address is generated. The sequence for checking access permissions is described below.

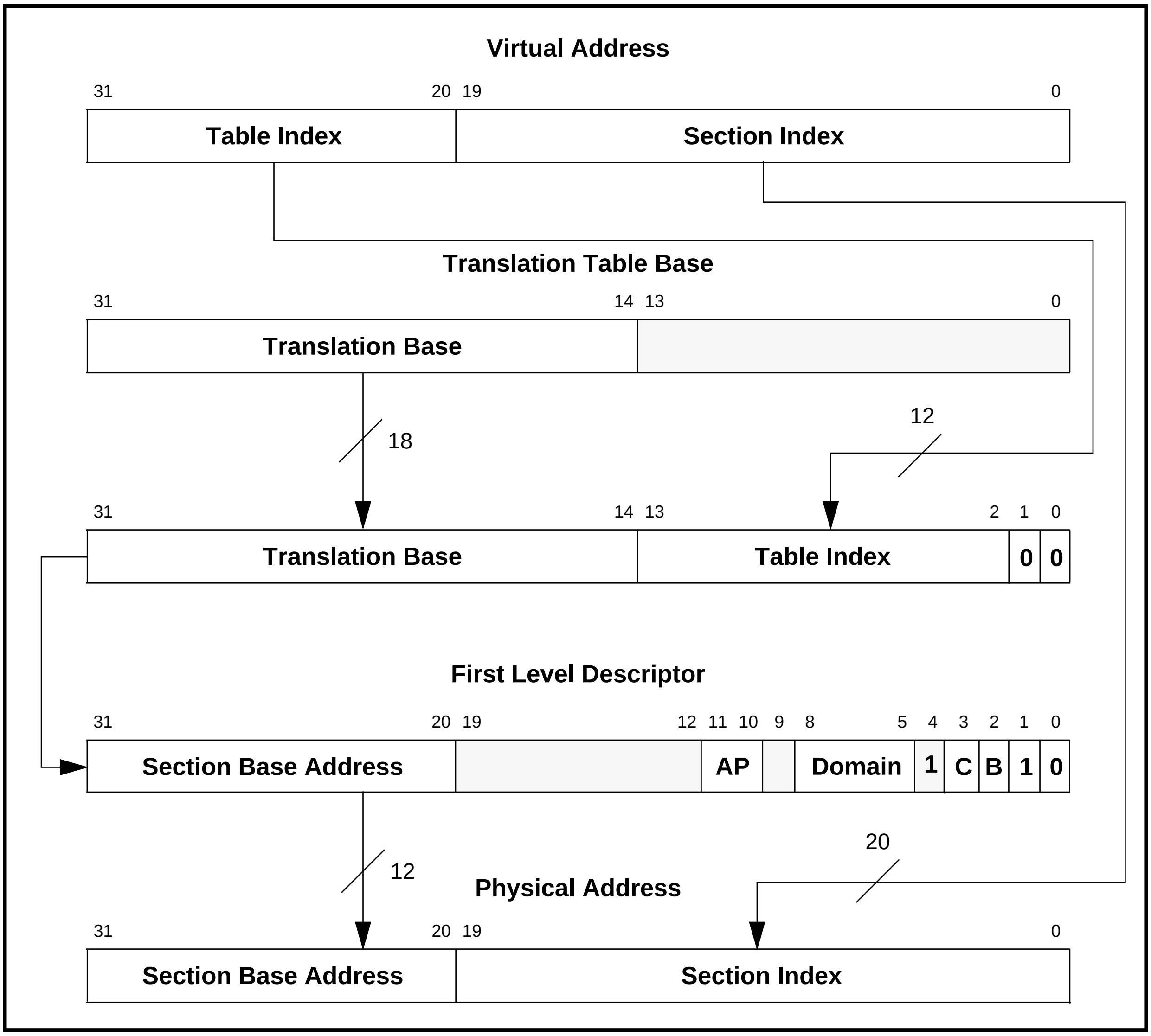


Figure 8-4: Section translation

8.8 Level Two Descriptor

If the Level One fetch returns a Page Table Descriptor, this provides the base address of the page table to be used. The page table is then accessed as described in ***Figure 8-6: Small page translation*** on page 8-12, and a Page Table Entry, or Level Two Descriptor, is returned. This in turn may define either a Small Page or a Large Page access. The figure below shows the format of Level Two Descriptors

.

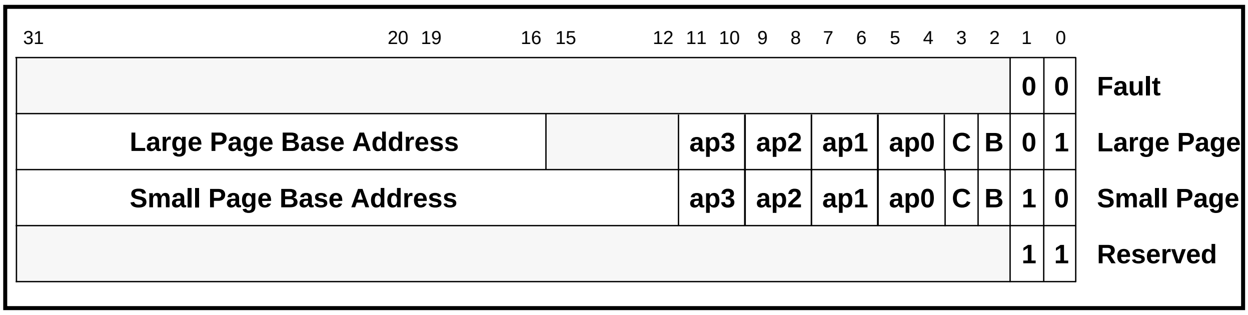


Figure 8-5: Page table entry (level two descriptor)

The two least significant bits indicate the page size and validity, and are interpreted as follows.

Value	Meaning	Notes
0 0	Invalid	Generates a Page Translation Fault
0 1	Large Page	Indicates that this is a 64 KB Page
1 0	Small Page	Indicates that this is a 4 KB Page
1 1	Reserved	Reserved for future use

Table 8-4: Interpreting page table entry Bits 1:0

Bit 3:2 (C : B) - The C & B bits together indicate whether the area of memory mapped by this section is treated as write-back cacheable, write-through cacheable, non cached buffered or non-cached non-buffered. Reference section 7.1.1 Cacheable and Bufferable Status of Memory Regions.

Bits 11:4 specify the access permissions (ap3 - ap0) for the four sub-pages and interpretation of these bits is described earlier in ***Table 8-2: Interpreting level one descriptor bits [1:0]*** on page 8-7.

For large pages, **bits 15:12** are programmed as 0.

Bits 31:12 (small pages) or bits **31:16** (large pages) are used to form the corresponding bits of the physical address - the physical page number. (The page index is derived from the virtual address as illustrated in ***Figure 8-6: Small page translation*** on page 8-12 and ***Figure 8-7: Large page Ttanslation*** on page 8-13).

8.9 Translating Small Page References

Figure 8-6: Small page translation illustrates the complete translation sequence for a 4KB Small Page. Page translation involves one additional step beyond that of a section translation: the Level One descriptor is the Page Table descriptor, and this is used to point to the Level Two descriptor, or Page Table Entry. (Note that the access permissions are now contained in the Level Two descriptor and must be checked before the physical address is generated. The sequence for checking access permissions is described later).

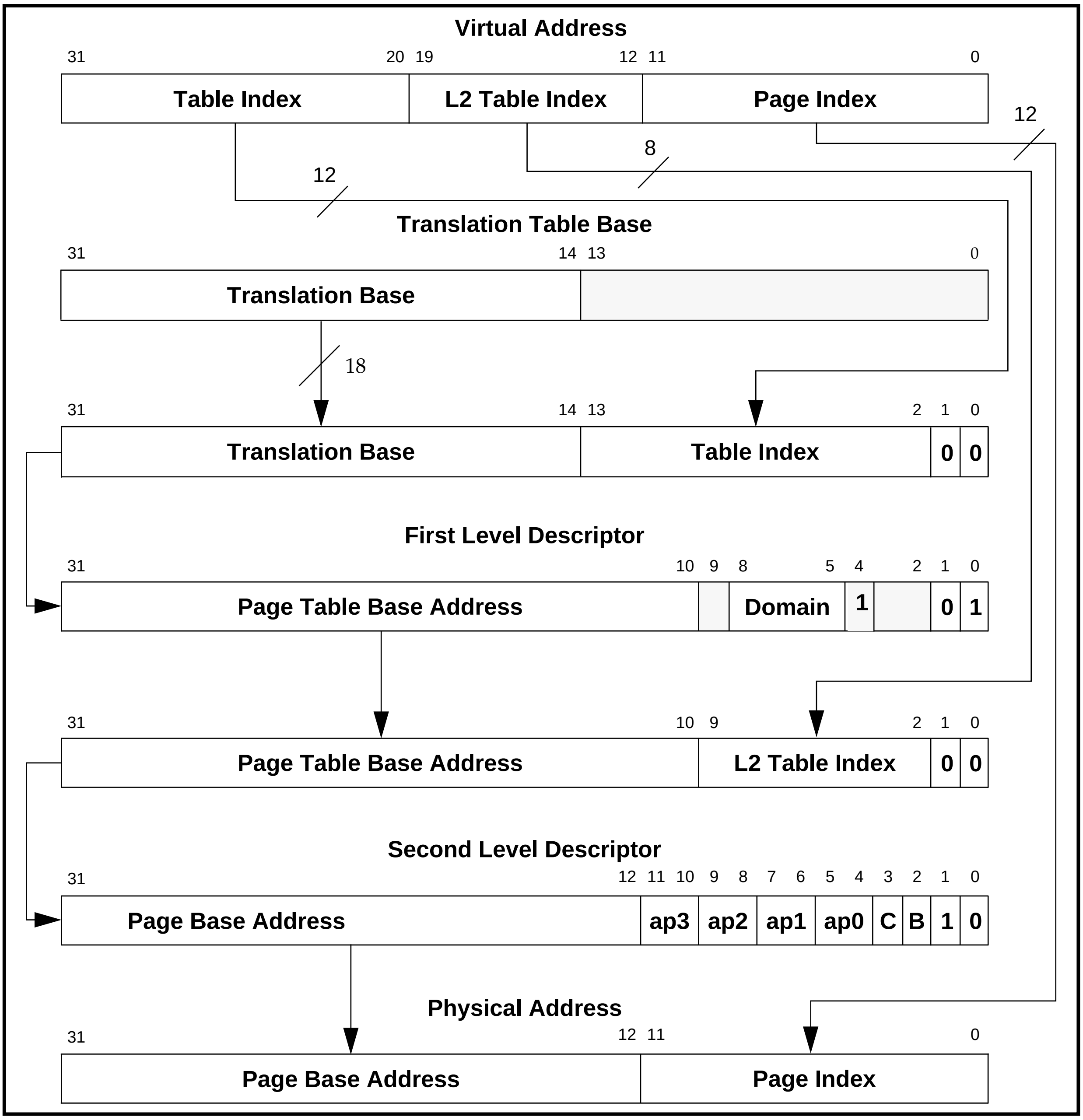


Figure 8-6: Small page translation

8.10 Translating Large Page References

Figure 8-7: Large page Ttanslation illustrates the complete translation sequence for a 64 KB Large Page. Note that since the upper four bits of the Page Index and low-order four bits of the Page Table index overlap, each Page Table Entry for a Large Page must be duplicated 16 times (in consecutive memory locations) in the Page Table.

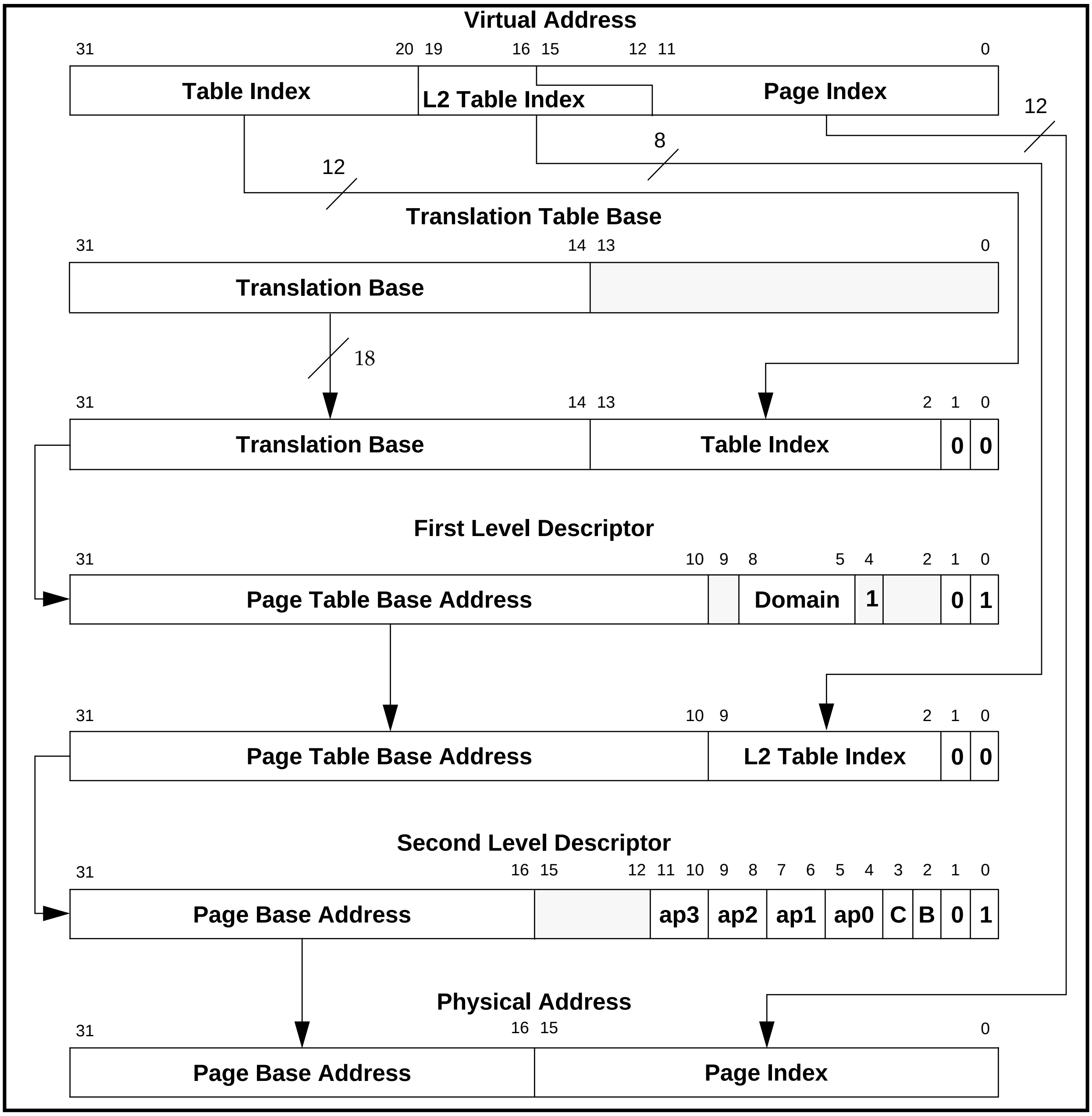


Figure 8-7: Large page Ttanslation

8.11 Cacheable and Bufferable Status of Memory Regions

For first level translation table descriptor for each Section, and the second level translation table descriptor for each Large Page, and each Small Page contain two bits—the C-bit and the B-bit—which specify whether the memory in that Section or Page will be cached or buffered, and whether it will be cached with Write-Through or Write-Back behaviour.†

In addition the cache and write buffer behaviour is controlled by the cache enable bit (C-bit) and write buffer enable bit (W-bit) in the CP15 Control Register.

To differentiate the two C bits, we shall add the subscript “tt” to the translation table bits giving us Ctt and Btt, and the subscript “cr” to the control register bits giving us Ccr and Wcr.

The Cache and Write Buffer Configuration is determined by the values of Ctt, Btt, Ccr, Wcr as shown in ***Table 8-5: Cache and write buffer configuration***.

Note † *Write-Back caches are also known as Copy-Back caches.*
“AND” means bitwise AND function.

Ctt AND Ccr	Btt AND Wcr	Cache, Writebuffer & External Abort Operation
0	0	Non-Cached, Non-Buffered (NCNB) • Reads and Writes are not cached. • Writes are not buffered. • Reads and writes may be externally aborted.*
0	1	Non-Cached Buffered (NCB) • Reads and Writes are not cached. • Writes are buffered. • Reads may be externally aborted. • Writes cannot be externally aborted.
1	0	Cached, Write-Through Mode. (WT) • Reads which hit in the cache read the data from the cache and do not perform an external access. • Reads which miss in the cache cause line fills which may be externally aborted. • All writes go off chip and are buffered. • Writes which hit in the cache update the cache. • Writes cannot be externally aborted.

Table 8-5: Cache and write buffer configuration

Ctt AND Ccr	Btt AND Wcr	Cache, Writebuffer & External Abort Operation
1	1	Cached, Write-Back Mode. (WB) • Reads which hit in the cache read the data from the cache and do not perform an external access. • Reads which miss in the cache cause line fills which may be externally aborted. • Writes which miss in the cache go off-chip and are buffered. • Writes which hit in the cache update the cache and mark the entry as dirty, and do not cause an external access. • Cache write-backs are buffered. • Writes (Cache Write-Misses & Cache Write-Backs) cannot be externally aborted.

Table 8-5: Cache and write buffer configuration (Continued)

Note that the Control Register C bit (Ccr) being zero disables all lookups in the cache, while the Translation table Register C bit (Ctt) being zero only stops new data being loaded into the cache. With Ccr = 1 and Ctt = 0 the cache will still be searched on every access to check whether the cache contains an entry for the data.

8.12 MMU Faults and CPU Aborts

The MMU generates six types of faults:

Alignment Fault
Translation Fault
Domain Fault
Permission Fault
Terminal Fault
Vector Fault

In addition, an external abort may be raised on external data access.

The access control mechanisms of the MMU detect the conditions that produce these faults. If a fault is detected as the result of a memory access, the MMU will abort the access and signal the fault condition to the CPU. The MMU is also capable of retaining status and address information about the abort. The CPU recognises two types of abort: data aborts and prefetch aborts, and these are treated differently by the MMU. See ***8.13 Fault Address and Fault Status Registers (FAR and FSR)***.

If the MMU detects an access violation, it will do so before the external memory access takes place, and it will therefore inhibit the access. External aborts will not necessarily inhibit the external access, as described in the section on external aborts.

8.13 Fault Address and Fault Status Registers (FAR and FSR)

Aborts resulting from data accesses (data aborts) are acted upon by the CPU immediately, and the MMU places an encoded 4 bit value FS[3:0], along with the 4 bit encoded Domain number, in the Fault Status Register (FSR). In addition, the virtual processor address associated with the data abort is latched into the Fault Address Register (FAR). If an access violation simultaneously generates more than one source of abort, they are encoded in the priority given in ***Table 8-6: Priority Encoding of Fault Status*** on page 8-17.

CPU instructions on the other hand are prefetched, so a prefetch abort simply flags the instruction as it enters the instruction pipeline. Only when (and if) the instruction is executed does it cause an abort; an abort is not acted upon if the instruction is not used (i.e. it is branched around). Because instruction prefetch aborts may or may not be acted upon, the MMU status information is not preserved for the resulting CPU abort; for a prefetch abort, the MMU does not update the FSR or FAR.

The sections that follow describe the various access permissions and controls supported by the MMU and detail how these are interpreted to generate faults.

Source		Priority	Domain[3:0]	FAR
		highest priority		
Terminal Exception		0b0010	invalid	VA of start of cache line being written-back
Vector Exception		0b0000	invalid	VA of access causing abort
Alignment		0b00x1	invalid	VA of access causing abort
External Abort on Translation	First level Second level	0b1100 0b1110	invalid valid	VA of access causing abort
Translation	Section Page	0b0101 0b0111	invalid valid	VA of access causing abort
Domain	Section Page	0b1001 0b1011	valid valid	VA of access causing abort
Permission	Section Page	0b1101 0b1111	valid valid	VA of access causing abort
External Abort on linefetch	Section Page	0b0100 0b0110	valid valid	VA of start of cache line being loaded
External Abort on non-linefetch	Section Page	0b1000 0b1010	valid valid	VA of access causing abort
		lowest priority		

Table 8-6: Priority Encoding of Fault Status

Notes

1 Alignment faults may write either 0b0001 or 0b0011 into FS[3:0].

2 Invalid values in Domain[3:0] occur because the fault is raised before a valid domain field has been selected.

3 Any abort masked by the priority encoding may be regenerated by fixing the primary abort and restarting the instruction.

4 The FS[3:0] encoding for Vector Exception breaks from the pattern that FS[0]==0 indicates an external abort.

8.14 Domain Access Control

MMU accesses are primarily controlled via domains. There are 16 domains, and each has a 2-bit field to define it. Two basic kinds of users are supported: Clients and Managers. Clients use a domain; Managers control the behaviour of the domain. The domains are defined in the Domain Access Control Register. ***Figure 8-8: Domain Access Control Register format*** on page 8-19 illustrates how the 32 bits of the register are allocated to define the sixteen 2-bit domains.

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

Figure 8-8: Domain Access Control Register format

Table 8-7: Interpreting access bits in Domain Access Control Register defines how the bits within each domain are interpreted to specify the access permissions.

Value	Meaning	Notes
00	No Access	Any access will generate a Domain Fault.
01	Client	Accesses are checked against the access permission bits in the Section or Page descriptor.
10	Reserved	Reserved. Currently behaves like the no access mode.
11	Manager	Accesses are NOT checked against the access Permission bits so a Permission fault cannot be generated.

Table 8-7: Interpreting access bits in Domain Access Control Register

8.15 Fault Checking Sequence

The sequence by which the MMU checks for access faults is slightly different for Sections and Pages. The figure below illustrates the sequence for both types of accesses. The sections and figures that follow describe the conditions that generate each of the faults.

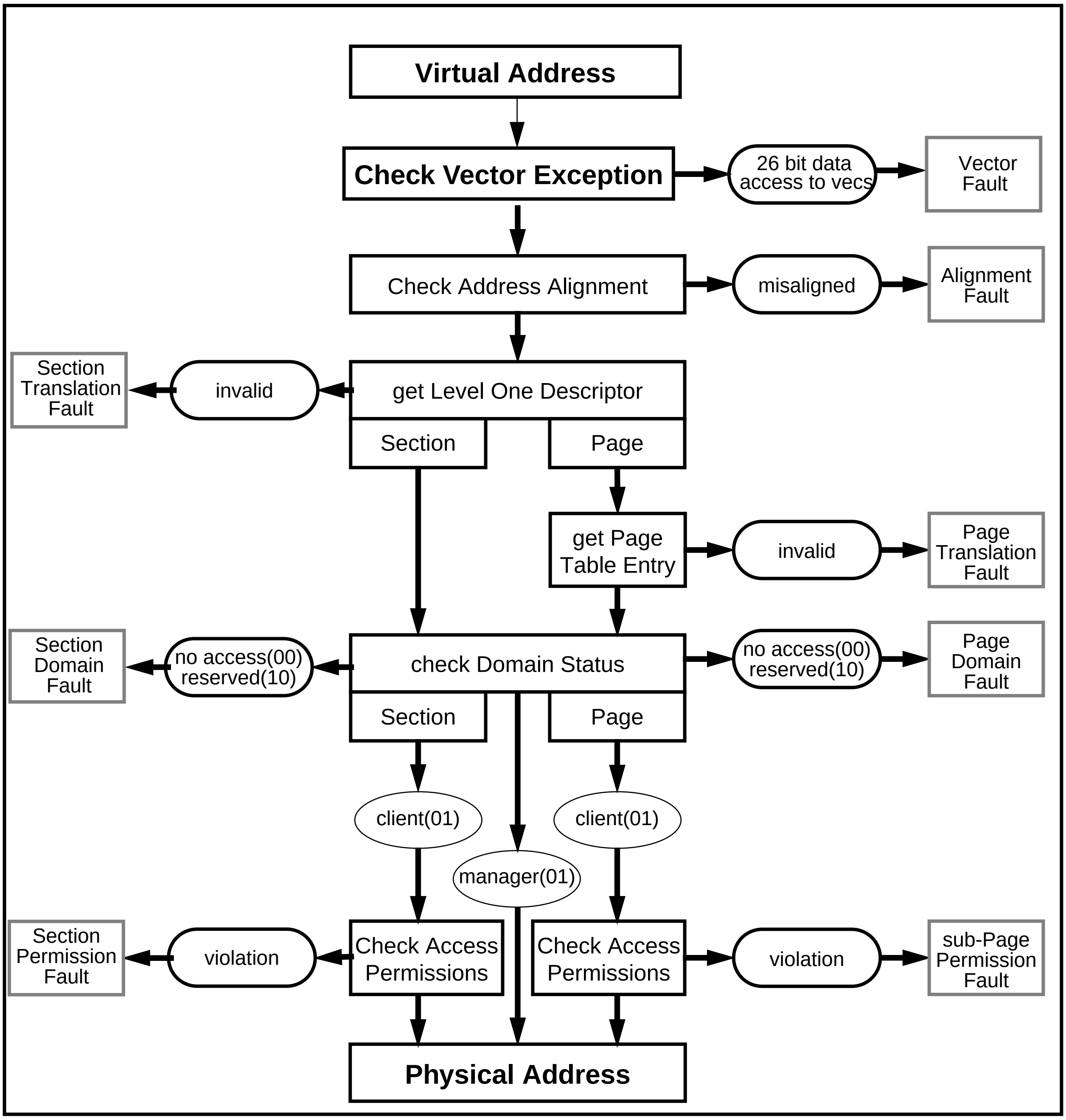


Figure 8-9: Sequence for checking faults

8.15.1 Terminal fault

A terminal fault indicates a system software error in the maintenance of the translation tables in main memory when using the Instruction-Data-Cache in Write-Back mode. It is indicated in theFault Address Register and Fault Status Register to aid debugging system software.

A terminal fault is indicated when a cache-write-back fails to translate the virtual address of the cache line to be written-back into a physical address because the associated translation table walk was aborted by the memory system or returned an invalid Level One or Level Two descriptor [A descriptor is invalid if bits[1:0] have the value “00” or “11”].

System Software must ensure that the cache contains no dirty-data for a page or section before changing the virtual-to-physical mapping of that page or section or disabling the virtual-to-physical mapping of that page or section. A Terminal Fault indicates that system software has failed to do this. When a terminal fault occurs, the data to be written-back from the cache to main memory is irrecoverably lost. A terminal fault is therefore not a reversible fault.

8.15.2 Vector fault

A Vector fault is generated by the MMU if the processor attempts a load or store data access to an address in the range &00000000 and &0000001F inclusive when operating in a 26-bit Mode. Vector faults are never generated for instruction fetches. Vector faults are generated regardless of the setting of the MMU enable bit (M-bit) in the System Control Coprocessor Control Register.

8.15.3 Alignment fault

If Alignment Fault is enabled (bit 1 in Control Register set), the MMU will generate an alignment fault on any data word access the address of which is not word-aligned irrespective of whether the MMU is enabled or not; in other words, if either of virtual address bits [1:0] are not 0. Alignment fault will not be generated on any instruction fetch, nor on any byte access. Note that if the access generates an alignment fault, the access sequence will abort without reference to further permission checks.

8.15.4 Translation fault

There are two types of translation fault: section and page.

1. A Section Translation Fault is generated if the Level One descriptor is marked as invalid. This happens if bits[1:0] of the descriptor are both 0 or both 1.
2. A Page Translation Fault is generated if the Page Table Entry is marked as invalid. This happens if bits[1:0] of the entry are both 0 or both 1.

8.15.5 Domain fault

There are two types of domain fault: section and page. In both cases the Level One descriptor holds the 4-bit Domain field which selects one of the sixteen 2-bit domains in the Domain Access Control Register. The two bits of the specified domain are then checked for access permissions as detailed in ***Table 8-3: Interpreting access permission (AP) Bits*** on page 8-9. In the case of a section, the domain is checked once the Level One descriptor is returned, and in the case of a page, the domain is checked once the Page Table Entry is returned.

If the specified access is either No Access (00) or Reserved (10) then either a Section Domain Fault or Page Domain Fault occurs.

8.15.6 Permission fault

There are two types of permission fault: section and sub-page. Permission fault is checked at the same time as Domain fault. If the 2-bit domain field returns client (01), then the permission access check is invoked as follows:

section:

If the Level One descriptor defines a section-mapped access, then the AP bits of the descriptor define whether or not the access is allowed according to ***Table 8-3: Interpreting access permission (AP) Bits*** on page 8-9. Their interpretation is dependent upon the setting of the S bit (Control Register bit 8). If the access is not allowed, then a Section Permission fault is generated.

sub-page:

If the Level One descriptor defines a page-mapped access, then the Level Two descriptor specifies four access permission fields (ap3..ap0) each corresponding to one quarter of the page. Hence for small pages, ap3 is selected by the top 1KB of the page, and ap0 is selected by the bottom 1KB of the page; for large pages, ap3 is selected by the top 16KB of the page, and ap0 is selected by the bottom 16KB of the page. The selected AP bits are then interpreted in exactly the same way as for a section (see ***Table 8-3: Interpreting access permission (AP) Bits*** on page 8-9), the only difference being that the fault generated is a sub-page permission fault.

8.16 External Aborts

In addition to the MMU-generated aborts, ARM810 has an external abort pin which may be used to flag an error on an external memory access. However, not all accesses can be aborted in this way, so this pin must be used with great care. The following section describes the restrictions.

The following accesses may be aborted and restarted safely. In the case of a read-lock-write sequence in which the read aborts, the write will not happen.

Reads

Unbuffered writes

Level One descriptor fetch

Level Two descriptor fetch

read-lock-write sequence

Cacheable reads (linefetches)

A linefetch may be safely aborted on any word in the transfer. If an abort occurs during the linefetch then the cache line will be invalidated. If the abort happens on a word that has been requested by the ARM8, the instruction will be aborted, otherwise the cache line will be invalidated but program flow will *not* be interrupted. The line is therefore invalidated under all circumstances.

Buffered writes.

Buffered writes cannot be externally aborted. Therefore, the system should be configured such that it does not do buffered writes to areas of memory which are capable of flagging an external abort.

Writes to Cacheable Regions

Writes to cacheable regions and cache write-backs are performed as buffered writes and cannot be externally aborted. The system design should ensure that writes to cacheable regions are not externally aborted.

8.17 Interaction of the MMU, IDC and Write Buffer

The MMU, IDC, WB and Branch prediction may be enabled/disabled independently. However, in order for the write buffer or the cache to be enabled the MMU must also be enabled. Also, Branch prediction must never be enabled when the cache is disabled. There are no hardware interlocks on these restrictions, so invalid combinations will cause undefined results.

MMU	IDC	WB
off	off	off
on	off	off
on	on	off
on	off	on
on	on	on

Table 8-8: Valid MMU, IDC and Write Buffer combinations

The following procedures must be observed.

To enable the MMU:

1. Program the Translation Table Base and Domain Access Control Registers
2. Program Level 1 and Level 2 page tables as required
3. Enable the MMU by setting bit 0 in the Control Register.

Note *Care must be taken if the translated address differs from the untranslated address as severalinstructions following the enabling of the MMU mayhave been fetched using “flat translation” and enabling the MMU may be considered as a branch with delayed execution. A similar situation occurs when the MMU is disabled. Consider the following code sequence:*

```
MOV             R1, #0x1
MCR             15,0,R1,0,0       ; Enable MMU
Fetch Flat
Fetch Flat
Fetch Translated
```

To disable the MMU:

1 Disable Branch prediction, if it is enabled, by using the code sequence given in ***6.3.3 Turning off Branch Prediction***.

2 Disable the WB by clearing bit 3 in the Control Register.

3 Disable the IDC by clearing bit 2 in the Control Register.

4 Disable the MMU by clearing bit 0 in the Control Register.
Note that if the MMU is enabled, then disabled and subsequently re-enabled the contents of the TLB will have been preserved. If these are now invalid, the TLB should be flushed before re-enabling the MMU.

Disabling of all three functions described in steps 2, 3 and 4 may be done simultaneously.

8.18 Effect of Reset

See ***3.7 Reset*** on page 3-12.

9 Write Buffer

This chapter describes the *Write Buffer* (*WB*).

9.1 Cacheable and Bufferable bits 9-3

9.2 Write Buffer Operation 9-4

Write Buffer

The ARM810 write buffer is provided to improve system performance. It can buffer up to 8 words of data, and 4 independent addresses. It may be enabled or disabled via the W bit (bit 3) in the ARM810 Control Register and the buffer is disabled and flushed on reset. The operation of the write buffer is further controlled by the C and B bits which are stored in the Memory Management Page Tables. For this reason, in order to use the write buffer, the MMU must be enabled. The two functions may however be enabled simultaneously, with a single write to the Control Register. For a write to use the write buffer, both the W bit in the Control Register and either the C or B bit in the corresponding page table must be set.

It is not possible to abort buffered writes externally; the abort pin will be ignored. Areas of memory which may generate aborts should be marked as unbufferable in the MMU page tables.

9.1 Cacheable and Bufferable bits

These bits controls whether a write operation may or may not use the write buffer. Typically main memory will be cacheable and bufferable and I/O space unbufferable. The C and B bits can be configured for both pages and sections. This is decribed in section ***8.11 Cacheable and Bufferable Status of Memory Regions*** on page 8-147.

9.2 Write Buffer Operation

9.2.1 Bufferable write

If the write buffer is enabled and the processor performs a write to a bufferable area, the data is placed in the write buffer at **FCLK** (**MCLK** if running with fastbus extension) speeds and the CPU continues execution. The write buffer then performs the external write in parallel. If however the write buffer is full (either because there are already 8 words of data in the buffer, or because there is no slot for the new address) then the processor is stalled until there is sufficient space in the buffer.

9.2.2 Unbufferable writes

If the write buffer is disabled or the CPU performs a write to an unbufferable area, the processor is stalled until the write buffer empties and the unbufferable write completes externally, which may require synchronisation and several external clock cycles.

9.2.3 Read-lock-write

The write phase of a read-lock-write sequence is treated as an Unbuffered write, even if it is marked as buffered.

Note: *A single write requires one address slot and one data slot in the write buffer; a sequential write of n words requires one address slot and n data slots. The total of 8 data slots in the buffer may be used as required. So for instance there could be 3 non-sequential writes and one sequential write of 5 words in the buffer, and the processor could continue as normal: a 5th write or a 6th word in the 4th write would stall the processor until the first write had completed.*

9.2.4 To enable the Write Buffer

To enable the write buffer, ensure the MMU is enabled by setting bit 0 in the Control Register, then enable the write buffer by setting bit 3 in the Control Register. The MMU and write buffer may be enabled simultaneously with a single write to the Control Register.

9.2.5 To disable the Write Buffer

To disable the write buffer, clear bit 3 in the Control Register.

Note *Any writes already in the write buffer will complete normally.*

10 Coprocessors

This chapter describes use of coprocessors with the ARM810.

10.1 Overview 10-2

Coprocessors

10.1 Overview

The ARM810 has no external coprocessor interface, so it is not possible to add external coprocessors to ARM810.

ARM810 has an internal coprocessor, called the System Control Coprocessor designated as coprocessor number 15. The System Control Coprocessor is used to control the configuration of the device, including the endianness setting, enabling of the Cache, MMU, Writebuffer, Branch Prediction, and the control of the Cache and MMU.

The System Control coprocessor is documented in detail in ***Chapter 5, Configuration*** and in the chapters on those parts of the ARM810 it controls: ***Chapter 7, Instruction and Data Cache (IDC)***, ***Chapter 8, Memory Management Unit***,***Chapter 9, Write Buffer***, ***Chapter 6, The Prefetch Unit***.

11 ARM810 Clocking

This chapter describes the bus interface clocking:

11.1 The Bus Clock 11-3

11.2 The Processor Clock 11-4

11.3 Generation of the Fast Clock 11-6

11.4 Forced Processor Clock from the Bus Clock 11-9

11.5 Low Power Idle and Sleep 11-10

ARM810 Clocking

The ARM810 uses two clock signals:

- bus clock
- fast clock

These clocks are derived from external inputs to the processor with configurations defined by external pins and the on-chip programmable registers.

The fast clock can be selected from three sources:

- bus clock
- on-chip PLL
- external reference clock

When the fast clock is sourced from the bus clock, operation is equivalent to ARM710a's Fastbus mode. When the fast clock is sourced from the external reference clock, the operation is equivalent to ARM710a's Standard bus mode.

The following sections explain how these clocks are made and describe their expected usage. In particular, note the addition of a clock multiplier (PLL) in this design.

11.1 The Bus Clock

The external bus clock is used to cycle the external bus interface. This clock is sourced directly from external input pins of the device. See ***Figure 11-1: Generating the external bus interface clock***.

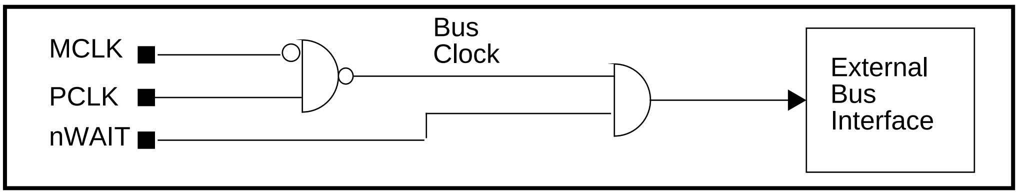


Figure 11-1: Generating the external bus interface clock

The bus clock is gated with **nWait** to provide the external bus clock itself. This allows external bus cycles to be extended if system timing requires it (see ***Figure 12-9: Use of the nWAIT pin to stop ARM810 for 1 MCLK cycle*** on page 12-16 for timing details).

11.1.1 External input clock: MCLK or PCLK

To provide for synchronous memory systems (eg. SDRAM, SSRAM) that use a clock which is essentially an inverted bus clock (returning data on the rising clock edge), you can choose to use the **PCLK** rather than the **MCLK** external input to avoid having to invert the clock externally. If you use **PCLK**, **MCLK** must be tied HIGH. If you use **MCLK**, **PCLK** must be tied LOW. New system designs should use **PCLK** for future compatibility. **MCLK** is provided for backwards compatibility. In future references in this document, the term *bus clock* refers to **MCLK** or **PCLK** depending on which is being used.

11.2 The Processor Clock

The processor clock is used to cycle the internals of the processor, see ***Figure 11-2: Generating the Processor Clock***. The processor clock can be sourced by one of two input clock signals to the synchroniser:

- bus clock
- fast clock

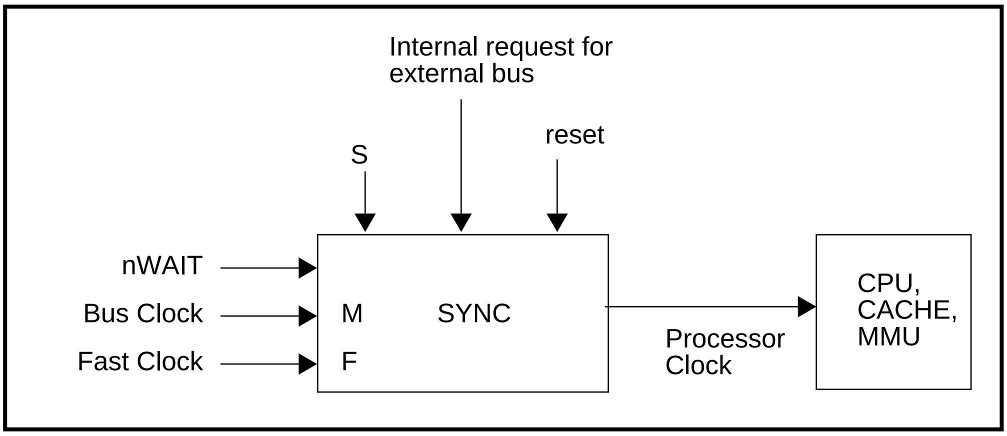


Figure 11-2: Generating the Processor Clock

When the processor is not performing external memory accesses, the fast clock (F) input to the synchroniser is the source for the processor clock (See ***11.3 Generation of the Fast Clock*** on page 11-6 for details of generating the fast clock). When external memory accesses are being made by the processor, the bus clock (M) input to the synchroniser is the source for the processor clock (See ***11.1 The Bus Clock*** on page 11-3 for details of generating the bus clock). Which of the sources to use is determined by the internal request for external bus signal during normal operation (see ***11.4 Forced Processor Clock from the Bus Clock*** on page 11-9 for details during RESET). When changing between F and M inputs, the synchroniser may perform re-synchronisation.

Note When a buffered write is made, the processor clock continues to run from the fast clock source at highest performance.

11.2.1 Synchronous/asynchronous operation

The state of the S bit (from Coprocessor 15, Register 15, bit 1) determines whether any synchronisation occurs between the bus clock (M) and fast clock (F) inputs to the synchroniser when the processor clock is changed from one to the other before and after external memory access cycles.

Synchronous operation

If the S bit is HIGH, there must be a tightly defined relationship between the bus clock and the fast clock (if this relationship is not obeyed, then the S bit should be set LOW). With the S bit HIGH, the Synchroniser will not perform any synchronisation, and the bus clock may only make transitions on the falling edge of the fast clock. Please refer to Section 15.2 for the timing requirements.

Asynchronous Operation

If the S bit is LOW, there is no defined relationship between the bus clock and the fast clock - they are asynchronous. The synchroniser introduces a synchronisation penalty whenever the internal core clock switches between the two input clocks (bus clock (M) and fast clock (F)). This penalty is symmetric, and varies between nothing and a whole period of the clock to which the core is synchronising. For example, when changing from the fast clock to the bus clock, the average synchronisation penalty is half a bus clock period, and when changing from the bus clock to the fast clock, it is half a fast clock period.

11.3 Generation of the Fast Clock

The fast clock input to the synchroniser can be selected from three sources. These are all configured internally using Coprocessor 15, Register 15, bits 2 and 3: F0 and F1. See ***11.5 Low Power Idle and Sleep*** on page 11-10 further details. During RESET, the bus clock is selected as the initial source for the fast clock.

11.3.1 Fast clock from the bus clock (Fastbus mode)

This configuration (F0=0, F1=0) makes the bus clock the source for the fast clock. This guarantees a defined relationship between the fast clock and the bus clock, and so synchronous operation (S=1) can be used for improved performance. This configuration is selected at RESET.

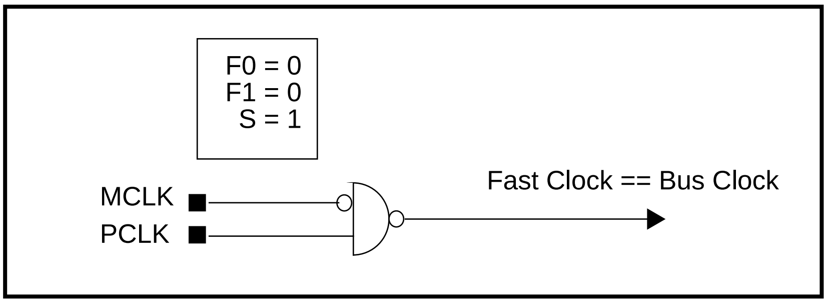


Figure 11-3: Fast clock the same as the bus clock

11.3.2 Fast clock from the output of the PLL

This configuration (F0=1, F1=1) makes the output of the PLL clock multiplier the source for the fast clock (see ***Figure 11-4: Fast clock from the output of the PLL***) When operating in this configuration, the S bit must be set LOW for asynchronous operation (S=0).

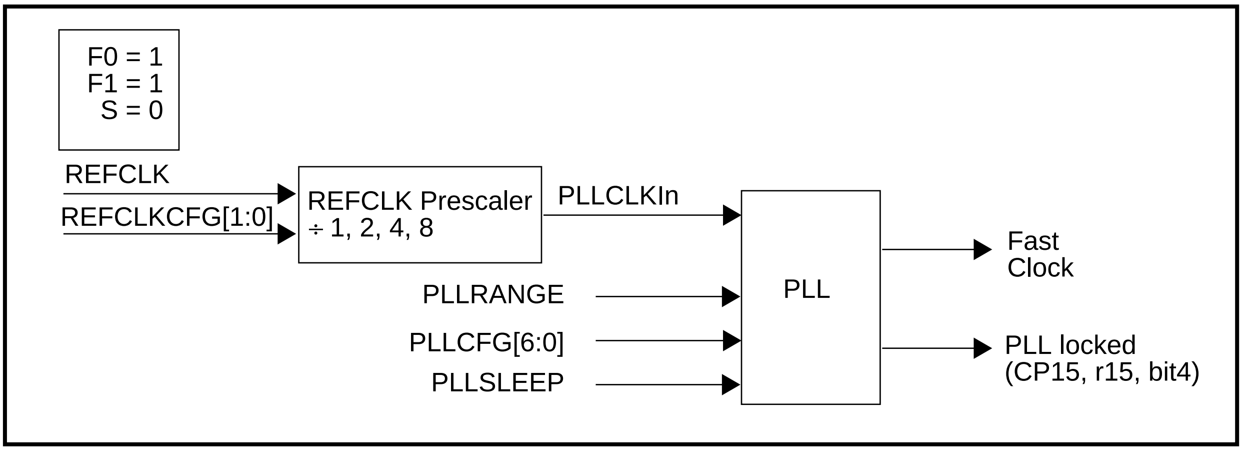


Figure 11-4: Fast clock from the output of the PLL

The PLL and input clock prescaler can be used to produce a fast clock frequency in the range 45MHz to 100MHz (or 22.5MHz to 50MHz if **PLLRANGE** is HIGH) from a **REFCLK** frequency of 1MHz to 80MHz.

The **REFCLK** prescaler divides the **REFCLK** frequency by 1, 2, 4 or 8 to produce **PLLCLKIn** under control of **REFCLKCFG** as shown in ***Table 11-1: Prescaler divide ratios***.

REFCLKCFG		Divide ratio
0	0	1
0	1	2
1	0	4
1	1	8

Table 11-1: Prescaler divide ratios

PLLCLKIn must be in the range 1MHz to 10MHz.

The fast clock output frequency is defined according to the following equation:

$$f_{FastClock} = f_{PLLCLKIn} * M/2$$

where:

$f_{FastClock}$ is the frequency of the fast clock output

$f_{PLLCLKIn}$ is the frequency of **PLLCLKIn**, which is the frequency of **REFCLK** divided by 1, 2, 4 or 8.

M is the value of the **PLLCFG** bus if interpreted as normal unsigned binary reporesentation. M is defined for the range M = 5, 6, 7 ..., 127. Values of M less than 5 are invalid.

The output frequency range of the PLL must reside between certain limits. These limits are determined by the **PLLRANGE**pin shown in ***Table 11-2: Output frequency range***.

PLLRANGE	Min Fast Clock (MHz)	Max Fast Clock (MHz)
LOW	45	100
HIGH	22.5	50

Table 11-2: Output frequency range

11.3.3 Fast clock direct (bypassing the PLL)

This configuration (F0=1, F1=0) provides a means of directly driving the fast clock from an external pin. This configuration may operate synchronously or asynchronously depending on how the reference clock (**REFCLK**) is generated. ***Figure 11-5: Fast clock direct*** shows this configuration.

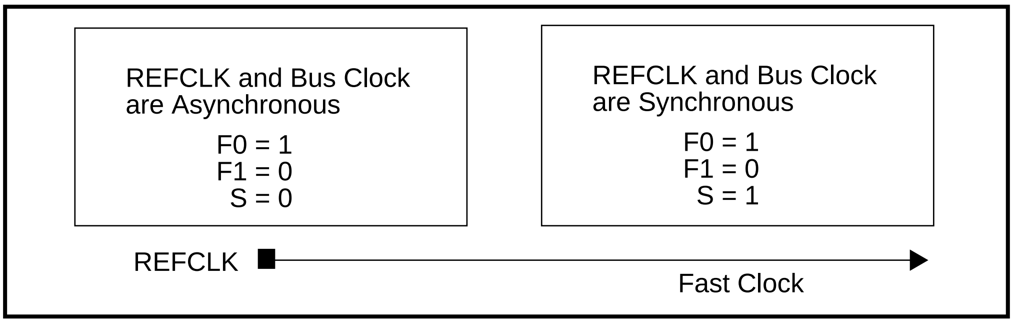


Figure 11-5: Fast clock direct

11.4 Forced Processor Clock from the Bus Clock

Coprocessor 15, Register 15, bit 0 (the **D** bit) is used to override the internal request for external bus signal to the synchroniser (see ***Figure 11-2: Generating the Processor Clock*** on page 11-4) and force the processor clock to be sourced from the bus clock. At RESET, the **D** bit is set LOW, and so the processor clock is sourced from the bus clock until this bit is changed. Once the fast clock source has been configured, and is sufficiently stable, the **D** bit should be set HIGH so the processor runs from the fast clock when not accessing the external bus.

11.5 Low Power Idle and Sleep

The **D** bit (see Section 11.4) can be employed to provide a clean transition to allow low-power idle or sleep mode. As the ARM810 is a fully static processor, stopping its clock when it has no work to do provides an ideal way to minimise power consumption and provide a fast start-up when it needs to operate again - all state is just frozen and does not need to be restored.

The easiest means of stopping the processor (and associated system) is to stop the bus clock. To allow the system to be stopped with the processor state at a precisely defined point in program execution, the processor clock must be sourced from the bus clock and the bus clock stopped. This can be achieved by setting the **D** bit LOW (writing 0 to Coprocessor 15, Register 15, bit 0) and then stopping the bus clock externally.

If the fast clock is being generated by the PLL clock multiplier and the PLL is left running while the bus clock is stopped, after restarting the bus clock, the **D** bit can be set HIGH and the processor clock sourced from an already locked fast clock PLL source. This could be implemented in the system for a fast wake-up-from-sleep interrupt response (though more power is consumed if the PLL is running continuously whilst the rest of the system is stopped).

The PLL itself can be placed in Sleep mode (using the PLLSLEEP external input), where it stops running and therefore consuming power. On wake-up, the PLL will take time to lock, and the system must take this into account - more details to be advised in future.

12 Bus Interface

This chapter describes the operation of the bus interface:

12.1 ARM810 Cycle Speed 12-2
12.2 Bus Interface Signals 12-3
12.3 Cycle Types 12-4
12.4 Addressing Signals 12-8
12.5 Memory Request Signals 12-10
12.6 Data Signal Timing 12-11
12.8 Maximum Sequential Length 12-13
12.9 Read-Lock-Write 12-14
12.10 Use of the nWAIT Pin 12-16
12.11 Use of the APE Pin 12-18
12.12 Bus Interface Sampling Points 12-19
12.13 Word, Halfword, and Byte Operations 12-22
12.14 Memory Access Sequence Summary 12-29
12.15 ARM810 Cycle Type Summary 12-33

12.1 ARM810 Cycle Speed

The bus interface is controlled by a *bus clock*, either **MCLK** or **PCLK**, and all timing parameters are referenced with respect to this clock. **MCLK** is the bus clock traditionally used on ARM microprocessors and this chapter is written in terms of **MCLK**. All diagrams show an **MCLK** waveform.

PCLK is an alternative bus clock input pin on ARM810 which allows use of a clock waveform which is equivalent to **MCLK** inverted. This is provided to simplify system design with industry standard synchronous DRAM, synchronous SRAM and programmable logic devices which are designed assuming a bus cycle starts and ends with a rising clock edge. If using the **PCLK** input, invert all references to **MCLK**, for example **MCLK** rising is equivalent to **PCLK** falling, **MCLK** HIGH is equivalent to **PCLK** LOW, etc. See ***Chapter 11, ARM810 Clocking*** for more details.

The speed of memory accesses may be controlled, for example to provide more time for a slow memory device or peripheral, in two ways:

- The LOW or HIGH phases, or both phases of the bus clock may be stretched.
- **nWAIT** may be used to insert entire bus clock cycles into the access. When LOW this signal holds ARM810 in the first phase of the bus cycle by gating out **MCLK** HIGH (or **PCLK** LOW). See ***12.10 Use of the nWAIT Pin*** on page 12-16.

When the ARM810 fast clock is being derived from the bus clock, we recommend that **nWAIT** is used to extend memory acesses rather than stretching the bus clock directly. See ***Chapter 11, ARM810 Clocking***.

12.2 Bus Interface Signals

The signals in the Bus interface can be grouped into 3 categories:

Addressing signals:

A[31:0]
nRW
MAS[1:0]
LOCK
nBLS[3:0]]
CLF

Memory Request signals:

nMREQ
SEQ

Data sampled signals:

D[31:0]

Abort signal:

ABORT

Each of these groups shares a common timing relationship to the bus interface cycles. The ARM bus interface addressing signals and memory request signals are pipelined ahead of the data. **nMREQ** and **SEQ** are pipelined by a whole bus cycle, and the address timed signals by 1/2 a cycle. The timing of the address timed signal can be altered by the **APE** pin.

Note *Unless otherwise specified, all diagrams in this chapter show the ARM810 operating with the* **APE** *pin held HIGH.*

12.3 Cycle Types

The ARM810 bus interface can perform 2 types of cycle, *idle* cycles and *memory* cycles. These cycles are differentiated by the pipelined signals **nMREQ** and **SEQ**. Conventionally cycles are considered to start from the falling edge of **MCLK**, and this is how they are shown in all diagrams.

The Addressing and Memory Request signals are pipelined ahead of the Data. Addressing by a phase (1/2 a cycle), and **nMREQ** and **SEQ** by a cycle. This advance information allows the implementation of efficient memory systems. **SEQ** is the inverse of **nMREQ** and is provided for backwards compatibility with earlier memory controllers.

A simplified single word memory access is shown in ***Figure 12-1: Simplified single cycle access***. The Access starts with the Address being broadcast. This can be used for decoding, but the access is not committed until **nMREQ** (Not Memory Request) goes LOW the following phase. This indicates that the next cycle will be a *memory* cycle. In the example, **nMREQ** returns HIGH after a single cycle, indicating that there will be a single *memory* cycle, followed by an *idle* cycle. The Data is transferred on the falling edge of **MCLK** at the end of the *memory* cycle.

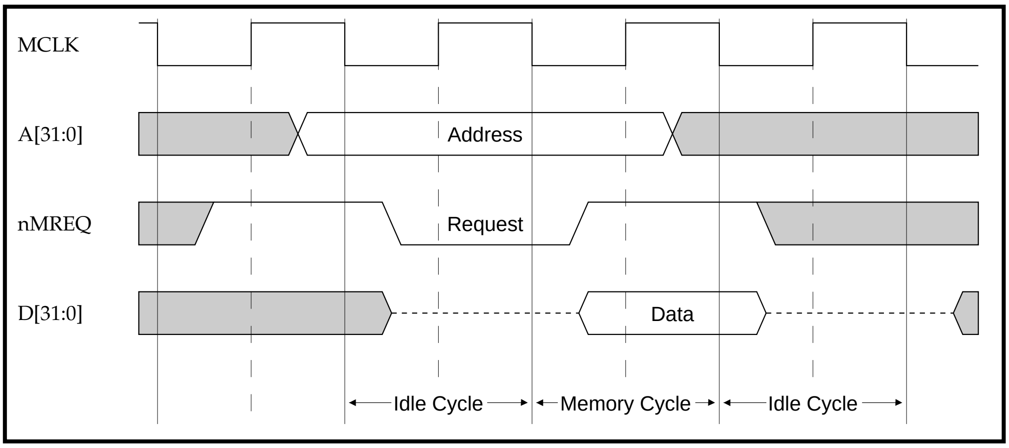


Figure 12-1: Simplified single cycle access

So a memory access consists of an *idle* cycle, with a valid address, followed by a *memory* cycle with the same address. The initial *idle* cycle allows the memory controller more time to decode the address.

The ARM810 can perform sequential bursts of accesses. These consist of an *idle* cycle and a *memory* cycle, as shown previously, followed by further *memory* cycles to incrementing word addresses (ie. a, a+4, a+8 etc.). See***Figure 12-2: Simplified sequential access*** on page 12-5. After the initial *idle* cycle, the address is pipelined by 1/2 a bus cycle from the data.

Note *Unless otherwise stated all of the diagrams in this section depict operation with* **APE** *held HIGH. The operation of* **APE** *is described in* ***12.11 Use of the APE Pin*** *on page 12-18.*

nMREQ and **SEQ** are pipelined by a bus cycle from the data. If **nWAIT** is being used to stretch cycles, then **nMREQ** and **SEQ** will no longer refer to the next **MCLK** cycle, but the next bus cycle. See ***12.10 Use of the nWAIT Pin*** on page 12-16.

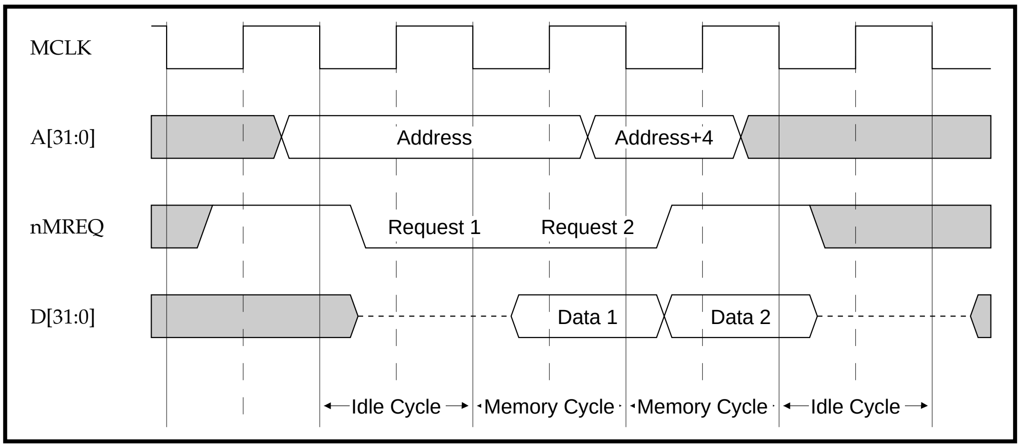


Figure 12-2: Simplified sequential access

Sequential bursts can only occur on word accesses, and will always be in the same direction, ie. Read (**nRW** LOW) or Write (**nRW** HIGH).

A memory controller should always qualify the use of the address with **nMREQ**. There are certain circumstances in which a new address can be broadcast on the address bus, but **nMREQ** will not go LOW to signal a memory access. This will only happen when an internal (MMU generated) abort occurs.

A single cycle memory access is shown in more detail in ***Figure 12-3: Single word read or write***. The timing parameters used throughout this section are defined in ***Chapter 15, ARM810 AC Parameters***.

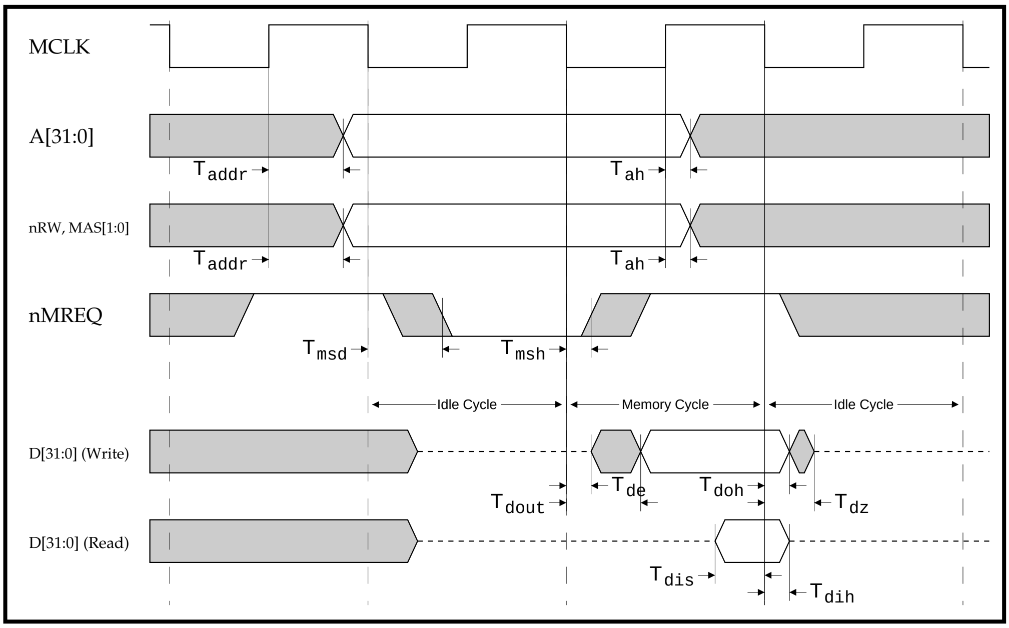


Figure 12-3: Single word read or write

After a non-sequential access as shown in ***Figure 12-3: Single word read or write*** on page 12-5 the interface can perform sequential *memory* cycles. See***Figure 12-4: Two word sequential read or write*** on page 12-6.

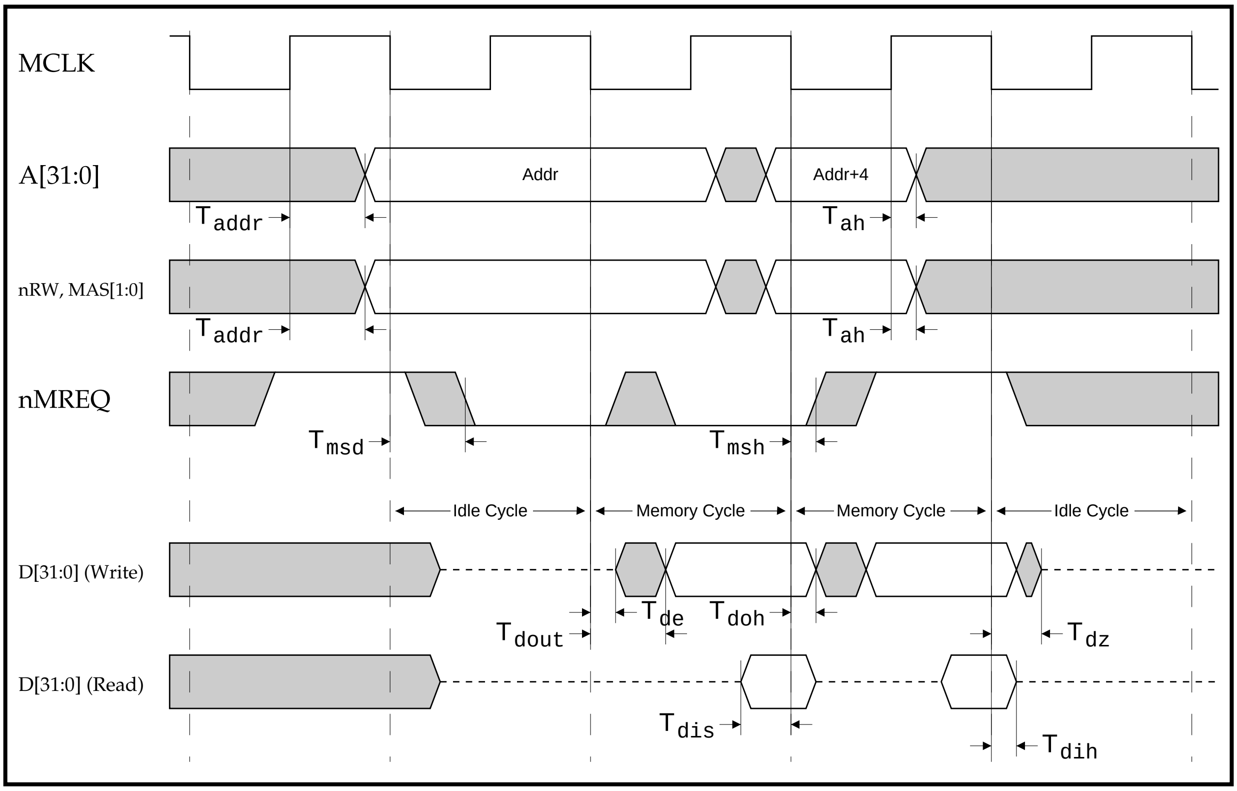


Figure 12-4: Two word sequential read or write

The minimum interval between bus accesses can occur after a buffered write. In this case there may only be a single *idle* cycle between two *memory* cycles to non-sequential addresses. This means that the address for the second access is broadcast on **A[31:0]** during the HIGH phase of the final *memory* cycle of the buffered write. See ***Figure 12-5: Minimum interval between bus accesses*** on page 12-7

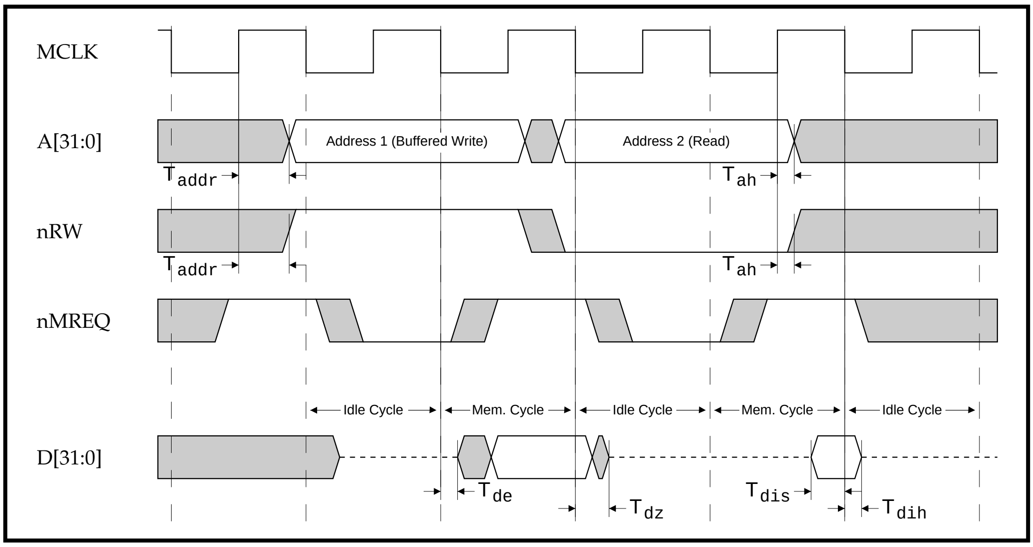


Figure 12-5: Minimum interval between bus accesses

This is the closest case of back to back cycles on the bus, and the memory controller should be designed to handle this case. In high speed systems one solution is to use **nWAIT** to increase the decode and access time available for the second access. The case shown is that of a write followed by a read. It could also have been a write followed by a non-sequential write.

A further result is that memory and peripheral strobes should not be direct decodes of the address bus. This could result in them changing during the last cycle of a write burst. Either **APE** should be used to modify the address timing, see ***12.11 Use of the APE Pin*** on page 12-18, or the memory controller must ensure that the address used is held until after the end of the cycle.

Where to sample the signals on the ARM810 bus interface is discussed in ***12.12 Bus Interface Sampling Points*** on page 12-19.

12.4 Addressing Signals

The timing of the addressing signals can be modified using the **APE** pin. If this pin is HIGH the addressing signals will be timed from the rising edge of the memory clock **MCLK.**

This in considered to be the standard timing of the interface, and is shown in the diagrams unless otherwise specified.

Memory accesses may be read or write, and are differentiated by the signal **nRW**. **nRW** may not change during a sequential access, so if a read from address A is followed immediately by a write to address (A+4), then the write to address (A+4) would be performed on the bus as a non-sequential access.

Likewise, any memory access may be of a word, halfword, or a byte quantity. These are differentiated by the signal **MAS[1:0]**. Again, **MAS[1:0]** may not change during sequential accesses. It is not possible to perform sequential byte or halfword accesses.

MAS[1:0] is encoded as follows:

MAS bit 1	bit 0	Access size
0	0	byte
0	1	halfword
1	0	word
1	1	reserved, not used

Table 12-1: MAS encoding

In order to reduce system power consumption, at the end of an access the addressing signals will be left with their current values until the next access occurs.

After a buffered write there may be only a single *idle* cycle between the two *memory* cycles. In this case the next non-sequential address will be broadcast in the last cycle of the previous access. This is the worst case for address decoding, see ***Figure 12-5: Minimum interval between bus accesses*** on page 12-7.

When operating the device with the **APE** pin LOW, the addresses are latched until the LOW phase of **MCLK**. See ***Figure 12-6: Single word read or write with delayed addressing*** on page 12-9. This is discussed further in .***12.11 Use of the APE Pin*** on page 12-18.

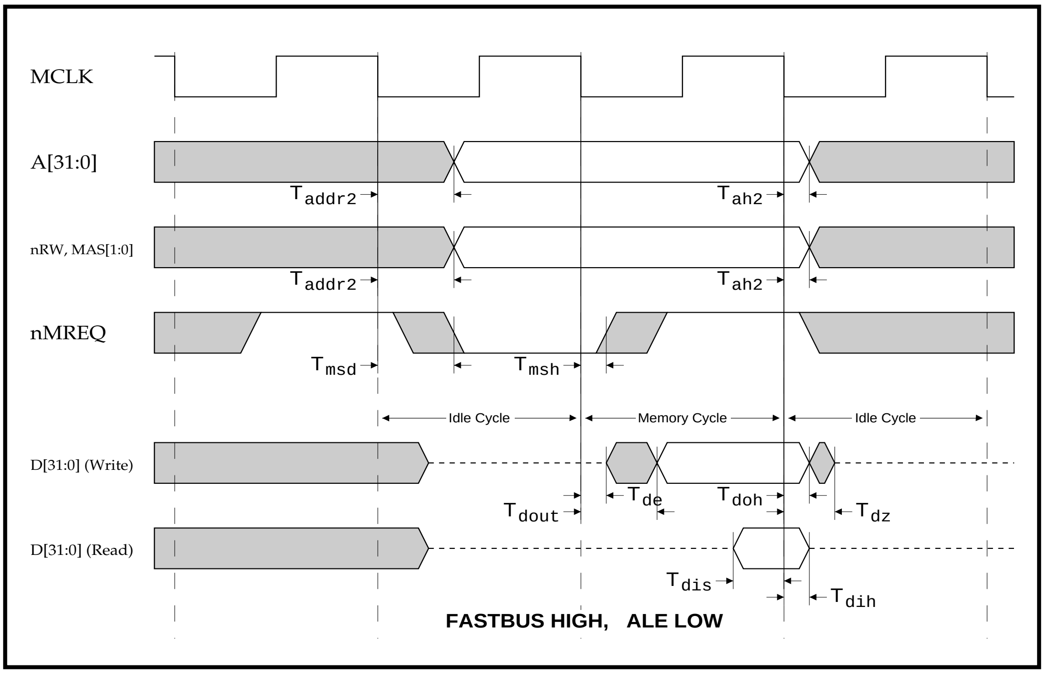


Figure 12-6: Single word read or write with delayed addressing

12.5 Memory Request Signals

The memory request signals, **nMREQ** and **SEQ**, are pipelined by 1 bus cycle, and refer to the next bus cycle. A LOW value on **nMREQ** indicates that next cycle on the ARM810 bus interface will be a *memory* cycle. Conversely, a HIGH value on **nMREQ** indicates that the next bus cycle will be an *idle* cycle.

Care must be taken when de-pipelining these signals if **nWAIT** is being used, as they always refer to the following bus cycle, rather than the following **MCLK** cycle. **nWAIT** will stretch the bus cycle by an integer number of **MCLK** cycles. See ***12.10 Use of the nWAIT Pin*** on page 12-16.

The signal **SEQ** is the inverse of **nMREQ**, and is provided for backwards compatibility with earlier memory controllers. **SEQ** may be left unconnected in new designs.

12.6 Data Signal Timing

12.6.1 D[31:0]

During a read access the data is sampled on the falling edge of **MCLK** at the end of the *memory* cycle. The setup and hold timings are given in ***Chapter 15, ARM810 AC Parameters***.

During a write access, the data on **D[31:0]** is timed off the falling edge of **MCLK** at the start of the *memory* cycle. If **nWAIT** is being used to stretch this cycle, the data will be valid from the falling edge of **MCLK** at the end of the previous cycle, when **nWAIT** was HIGH. See ***12.10 Use of the nWAIT Pin*** on page 12-16.

In a low power system it is important to ensure that the databus is not allowed to float to an undefined level. This will cause power to be dissipated in the inputs of devices connected to the bus. This is particularly important when a system is put into a low power sleep mode. We recommend that one set of databus drivers in the system are left enabled during sleep to hold the bus at a defined level.

12.7 ABORT Signal

The **ABORT** signal is sampled in the middle of each *memory* cycle on the rising edge of MCLK, on both read and write accesses.The effect of **ABORT** on the operation of the ARM810 is discussed in ***Chapter 3, Programmer’s Model***, and in ***8.16 External Aborts*** on page 8-23

An **ABORT** can be flagged on any *memory* cycle, however it will be ignored on buffered writes, which cannot be aborted. Cache write-backs from regions of memory marked as write-back cacheable are performed as buffered-writes and also cannot be aborted.

Due to the pipelined nature of the bus interface, if an access in the middle of a sequential burst is aborted, the bus interface has already requested the next access in the burst, and ARM810 will end the sequential branch instruction after this next access is completed, unless the access is a linefill. Linefills always read four words, regardless of which accesses are aborted.

The effect of **ABORT** during linefetches is slightly different to that during other access. During a linefetch the ARM810 will fetch four words of data regardless of which words of data were requested by the ARM core, the rest of the words are fetched speculatively. If **ABORT** is asserted on a word which was requested by the ARM core, the abort will function normally. If the abort is signalled on a word which was not requested by the ARM core, the access will not be aborted, and program flow will not be interrupted. Regardless of which word was aborted, the line of data will not be placed in the cache as it is assumed to contain invalid data.

12.8 Maximum Sequential Length

The ARM810 may perform sequential memory accesses whenever the cycle is of the same type (i.e. read/write) as the previous cycle, and the addresses are consecutive. However, sequential accesses are interrupted on a 256-word boundary. This is to allow the MMU to check the translation protection as the address crosses a sub-page boundary. If a sequential access is performed over a 256-word boundary, the access to word 256 is turned into a non-sequential access, and further accesses continue sequentially as before.

This also simplifies the design of the memory controller. Provided that peripherals and areas of memory are aligned to 256-word boundaries sequential bursts will always be local to one peripheral or memory device. This means that all accesses to a device will always start with a non-sequential access.

A DRAM controller can take advantage of the fact that sequential cycles will always be within a DRAM page, provided the DRAM page covers an address range of at least 256 words.

12.9 Read-Lock-Write

The read-lock-write sequence is generated by a SWP instruction. On the bus it consists of a read access followed by a write access to the same address. This sequence is differentiated by the **LOCK** signal. **LOCK** has addressing signal timing and is controlled similarly by **APE**. If **APE** is HIGH, **LOCK** will go HIGH in the HIGH phase of **MCLK** at the start of the read access. It will always go LOW at the end of the write access.

The **LOCK** signal indicates that the two accesses should be treated as an atomic unit. A memory controller should ensure that no other bus activity is allowed to happen in between the accesses while **LOCK** is asserted. When the ARM810 has started a read-lock-write sequence it cannot be interrupted until it has completed.

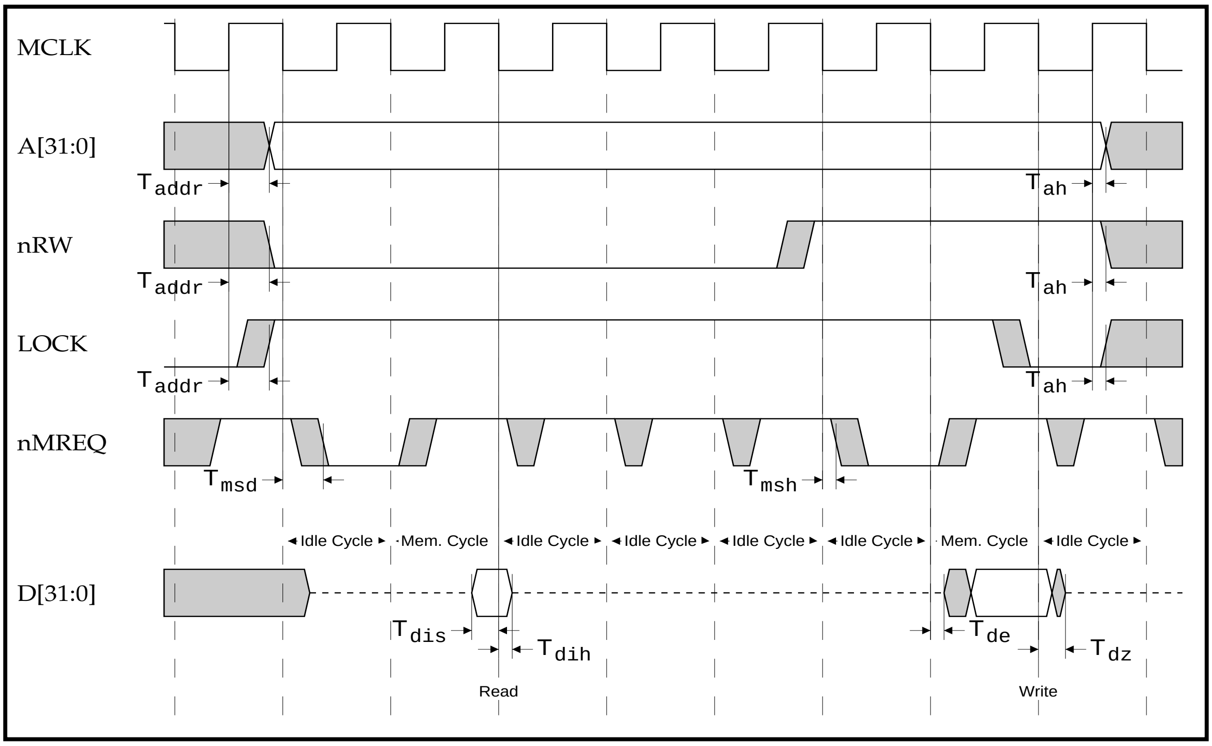


Figure 12-7: Read-locked-write

The read cycle will always be performed as a single, non-sequential, external read cycle, regardless of the contents of the cache. The write will be forced to be unbuffered, so that it can be aborted if necessary. The cache will be updated on the write.

Either of the accesses of the read locked write sequence can be aborted. The bus behaviour when the read access is aborted is shown in ***Figure 12-8: Read-locked-write with read aborted***

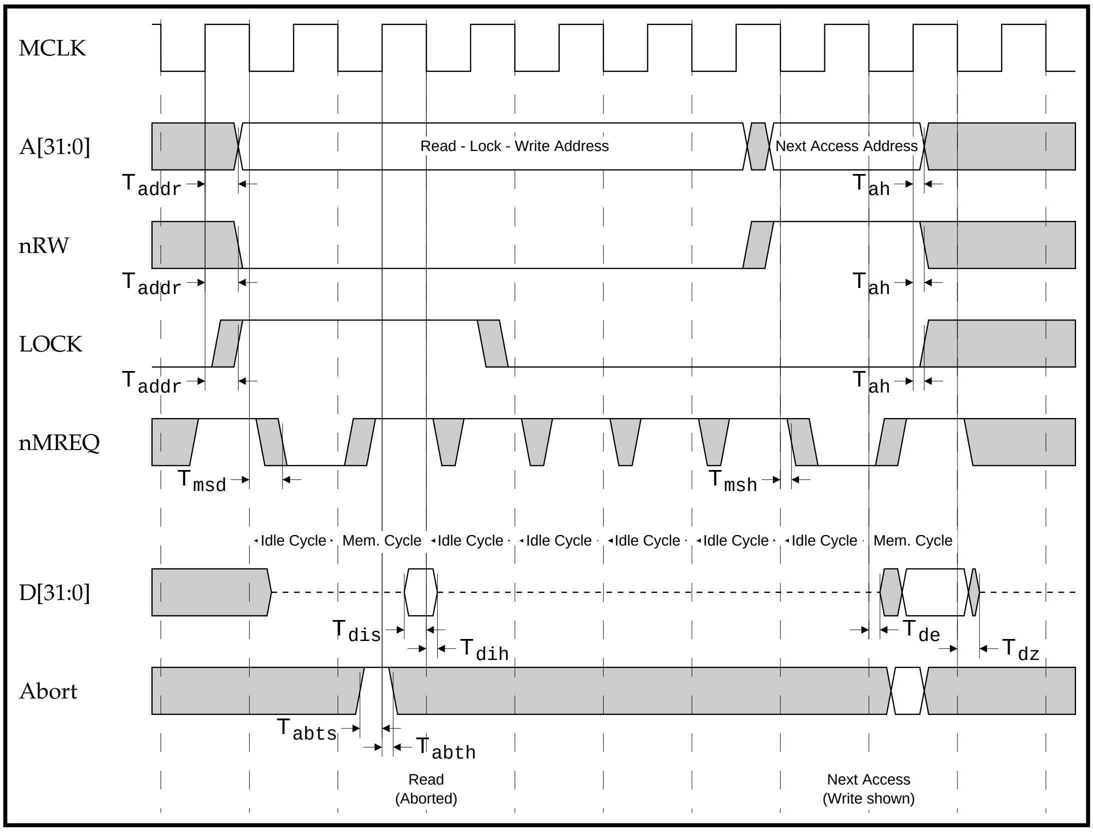


Figure 12-8: Read-locked-write with read aborted

12.10 Use of the nWAIT Pin

The **nWAIT** pin can be used to extend memory accesses in whole cycle increments. **nWAIT** may only change during the LOW phase of **MCLK** and when low gates out **MCLK** HIGH phases. **nWAIT** will not prevent changes in **nMREQ**, **SEQ** and a write on **D[31:0]** during the phase in which it is taken LOW. Changes in these signals will then be prevented until the **MCLK** HIGH phase after **nWAIT** was taken HIGH. All other outputs cannot change from the time **nWAIT** goes LOW until the next **MCLK** HIGH phase after **nWAIT** returns HIGH.

The address timing is dependant on **nWAIT** as follows:

- If **APE** is LOW, **nWAIT** will not prevent changes on **A[31:0]** during the phase in which it was taken LOW. **A[31:0]** will be prevented from changing until the **MCLK** LOW phase after **nWAIT** becomes HIGH again.
- If **APE** is HIGH, **A[31:0]** will be prevented from changing from the time **nWAIT** goes LOW until the next **MCLK** HIGH phase after **nWAIT** returns HIGH.

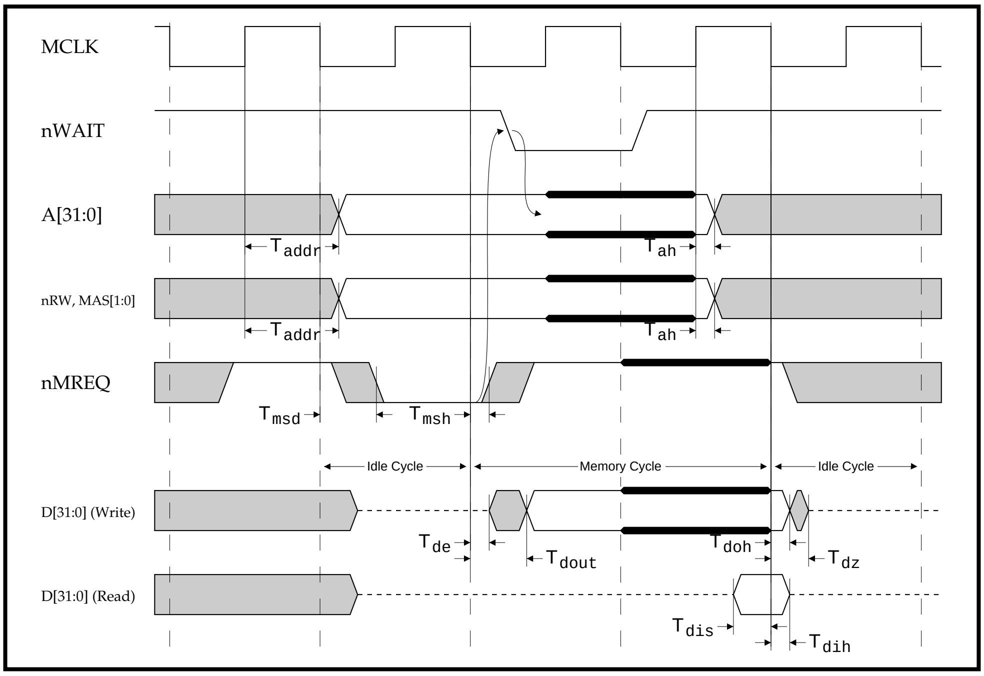


Figure 12-9: Use of the* nWAIT *pin to stop ARM810 for 1 MCLK cycle

The heavy bars indicate the cycle for which signals will be stable as a result of asserting **nWAIT**, assuming APE is HIGH.

The signals **nMREQ** and **SEQ** are pipelined by one bus cycle. This pipelining should be taken into account when these signals are being decoded. The value of **nMREQ** indicates whether the next bus cycle is a data cycle or an Idle Cycle. As bus cycles are stretched by **nWAIT,** the boundary between bus cycles is determined by the falling edge of **MCLK** when **nWAIT** is HIGH. A useful rule of thumb is to sample the value of **nMREQ** only when **nWAIT** is HIGH.

When **nWAIT** is used to stretch a *memory* cycle, **nMREQ** will return HIGH during the first phase of the *memory* cycle if a single word access is occurring. In this case it is important that the memory controller does not interpret the HIGH value on **nMREQ** as indicating that an *idle* cycle is signalled when in fact it is a stretched *memory* cycle. See ***Figure 12-9: Use of the nWAIT pin to stop ARM810 for 1 MCLK cycle*** on page 12-16

12.11 Use of the APE Pin

APE is used to modify the address timing. It is a static configuration pin which should be tied HIGH or LOW.

- If **APE** is HIGH the address timing seen on **A[31:0], nBLS[3:0],nRW,nBW** and **LOCK** will be the standard pipelined address timing, with the addresses changing during the HIGH phase of **MCLK**.
- If **APE** is LOW the address timing is modified, and the address changes during the following LOW phase of **MCLK**. See ***Chapter 15, ARM810 AC Parameters***.

12.12 Bus Interface Sampling Points

The following two sections describe the recommended sampling points for bus interface signals, the first section when operating at or near the maximum frequency, and the second section when operating at a lower frequency. Recommended sampling points are denoted by the heavy bars (transfer bars) on signals in the figures, and the earliest recommended sampling point is also given in the tables.

The signals **nMREQ** and **SEQ** are pipelined with respect to the bus interface. This pipelining should be taken into account when these signals are being decoded. The value of **nMREQ** indicates whether the next bus cycle is a data cycle or an idle cycle. As bus cycles are stretched by **nWAIT** the boundary between bus cycles is determined by the falling edge of **MCLK** when **nWAIT** is HIGH. A useful rule of Thumb is to sample the value of **nMREQ** only when nWAIT is HIGH. This is shown by the transfer bars in ***Figure 12-10: Sampling points at maximum frequency*** and ***Figure 12-11: Sampling points at reduced frequency***.

The **MCLK** frequencies at which these differing methodologies should be used will depend on the device parameters. Please consult the AC timings to determine which sampling points should be used. These can be obtained from your Semiconductor supplier.

12.12.1 Fast Operation

If the ARM810 is being operated at, or near, its maximum operating frequency the output delays on the bus interface mean that the signals must be sampled as late as is possible.

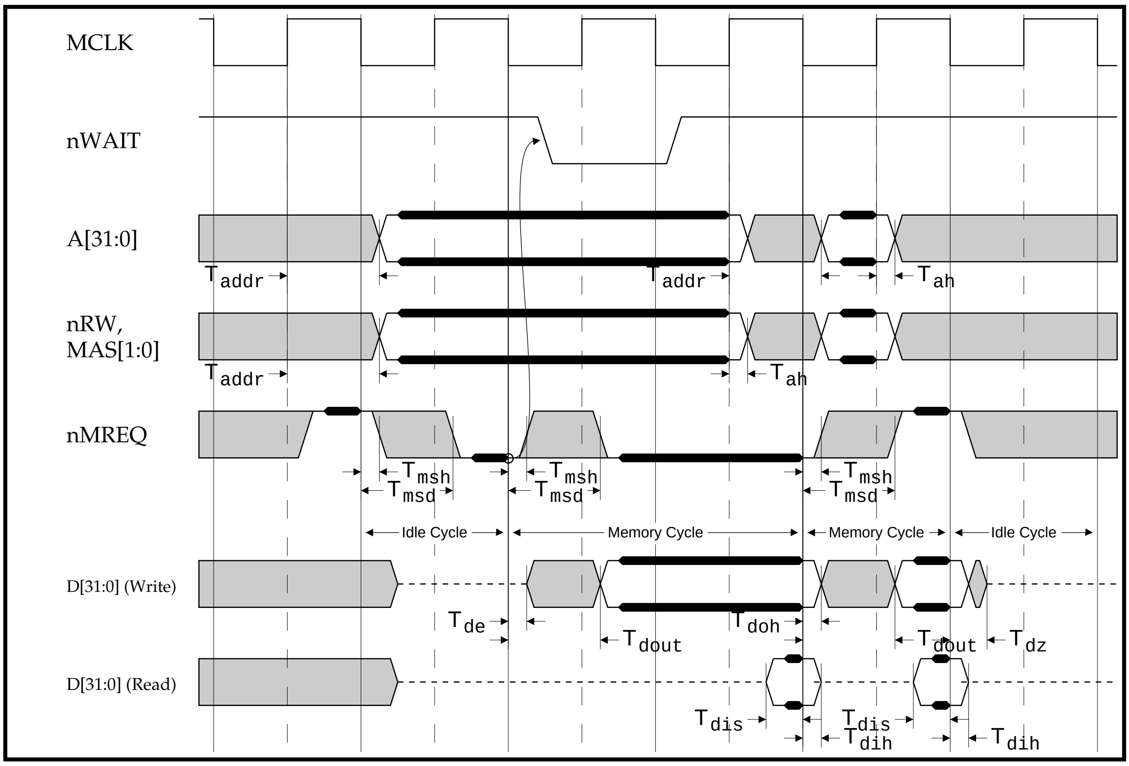


Figure 12-10: Sampling points at maximum frequency

Signal	Earliest Recommended Sample Point
A[31:0]	Set-up to **MCLK** RISING
nBLS[3:0],nRW, MAS[1:0], LOCK	Set-up to **MCLK** RISING
nMREQ, SEQ	Set-up to **MCLK** FALLING
D[31:0] (Write)	Set-up to **MCLK** FALLING

Table 12-2: Sampling points at maximum frequency

Sampling the signals at these points will result in the most robust system design, which will scale to faster clock speeds. However, it does reduce the time available to the memory controller.

12.12.2Reduced frequency operation

When operating the bus interface at a reduced frequency it is possible to sample the bus interface signals at earlier points in the cycle. This allows the memory system to make more efficient use of the cycles.

It is strongly recommended that **nWAIT** is derived from **nMREQ** as shown in the diagram. Trying to generate **nWAIT** in the previous cycle is liable to result in a critical path which will limit the maximum frequency of operation of the design unnecessarily..

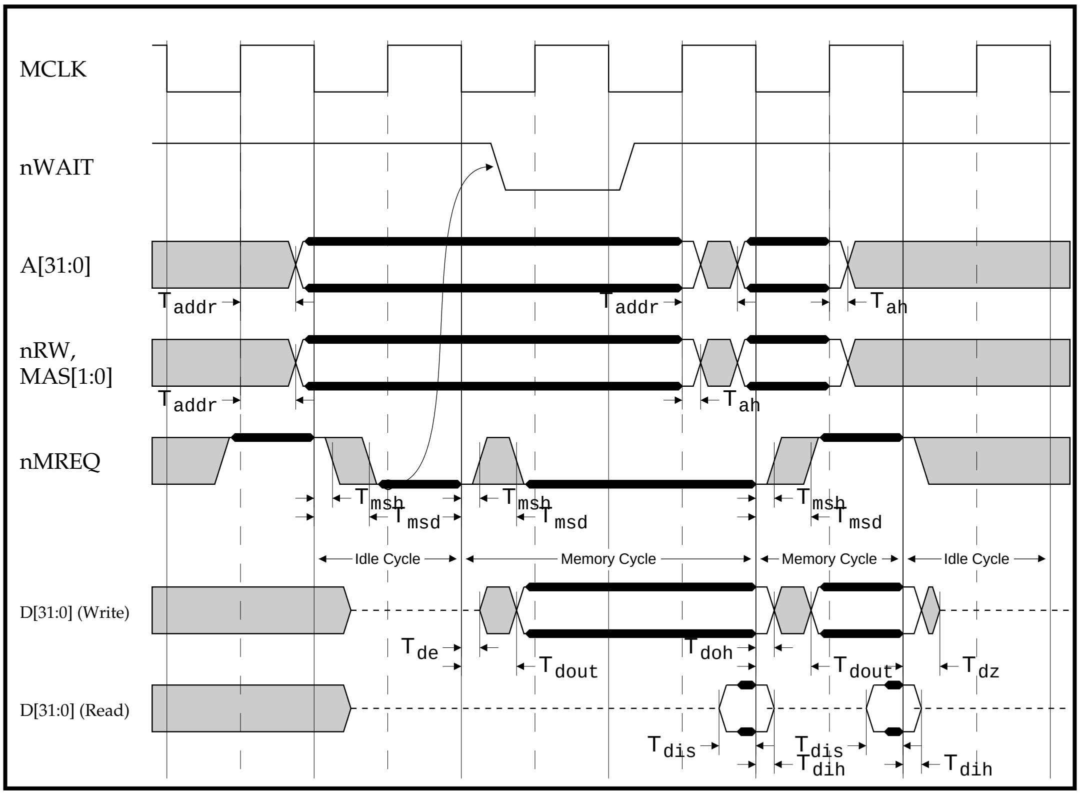


Figure 12-11: Sampling points at reduced frequency

Signal	Earliest Recommended Sample Point
A[31:0]	**MCLK** FALLING
nBLS[3:0],nRW, MAS[1:0], LOCK	**MCLK** FALLING
nMREQ, SEQ	**MCLK** RISING
D[31:0] (Write)	**MCLK** RISING

Table 12-3: Sampling points at reduced frequency

12.13 Word, Halfword, and Byte Operations

The ARM810 can perform word, halfword and byte operations on the bus interface. The memory system can decode which bytes of the memory system to access using one of two methods

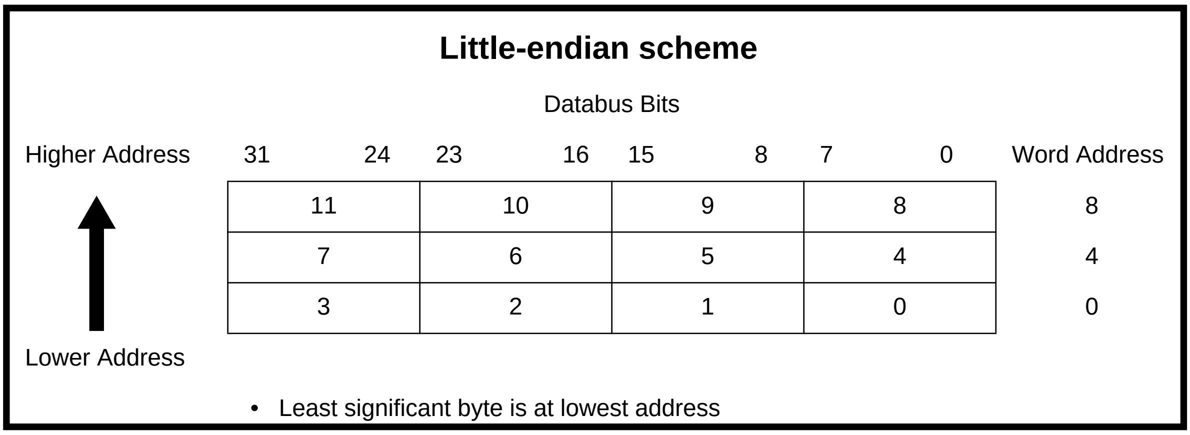
- Decoding the access width and byte address information contained in MAS[1:0] and A[1:0]. ***12.13.1 Decoding Byte Activity using MAS[1:0] and A[1:0]*** describes this method of memory system operation. This method provides limited compatibilty with memory systems designed for earlier ARM processors - see ***12.13.3 Backwards compatibility with earlier ARM Processors*** on page 12-27 for details.
- Using the Byte Lane Selects (nBLS[3:0]). These outputs indicate directly which bytes of the memory system should be activated for the access. ***12.13.2 Decoding Byte Activity using nBLS[3:0]*** on page 12-25 describes this method of memory system operation. We recommend use of this decode method for all new memory system designs.

12.13.1 Decoding Byte Activity using MAS[1:0] and A[1:0]

Big-endian / little-endian operation

The ARM810 treats words in memory as being stored in big-endian or little-endian format depending on the value of the bigend bit in the control register.

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.

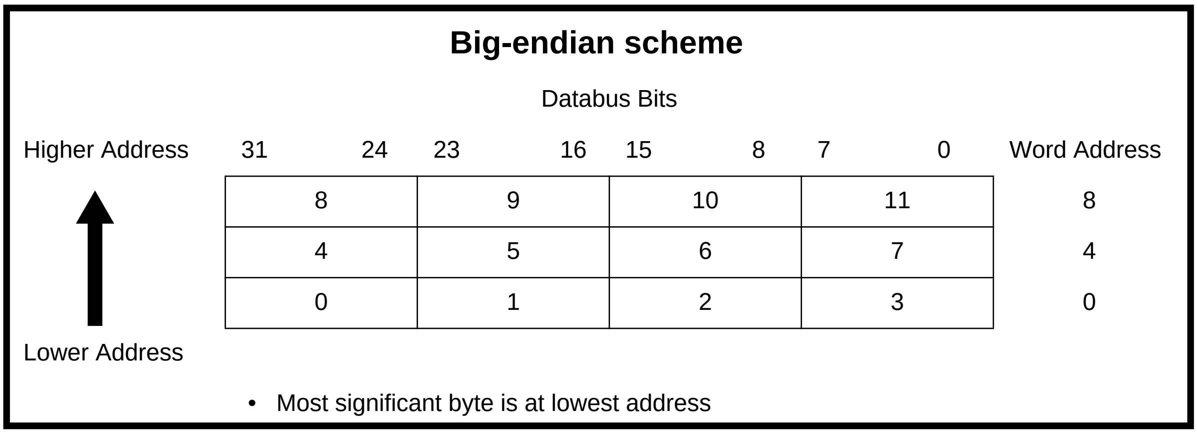


31 24	23 16	15 8	7 0	Word Address
11	10	9	8	8
7	6	5	4	4
3	2	1	0	0

- Least significant byte is at lowest address

Figure 12-12: Little-endian addresses of bytes within word

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 endianness: see ***4.9 Single Data Transfer (LDR, STR)*** on page 4-27 for more details.



Big-endian scheme

Databus Bits

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

- Most significant byte is at lowest address

Figure 12-14: Big-endian addresses of bytes within words

Decoding MAS[1:0] and A[31:0]

This Byte Activity decode method is recommended if upgrading a memory controller designed for an earlier ARM processor or if compatibility with earlier ARM Processors is a concern. If neither of these conditions apply, we recommend the Byte Activity Decode method described in ***12.13.1 Decoding Byte Activity using MAS[1:0] and A[1:0]*** on page 12-22

For all accesses **A[31:2]** provides the address of the 32-bit word the access will read or write. **A[1:0]** and **MAS[1:0]** indicate which bytes within that word must be written or read.

The external memory system must decode these signals differently for big-endian and little-endian systems. It is recommended that systems which wish to support dynamic switching of endianness should decode Byte Activity using the **nBLS[3:0]** signals described in section ***12.13.2 Decoding Byte Activity using nBLS[3:0]*** on page 12-25

When performing word writes, the ARM810 presents the 32 bit word to the memory system on the data bus **D[31:0]**, and the memory system must enable a write into all four bytes of the memory system.

When performing halfword writes the ARM810 ensures the halfword to be written appears on the data bus in the place where it is required by a 32-bit wide memory system. The ARM810 achieves this by duplicating the halfword to be written twice across the 32 bit data bus, on **D[31:16]** and **D[15:0]**, so that the memory system need only enable a write into the correct two bytes of the memory system as shown in the tables below. Memory systems narrower than 32-bits may make use of this duplication to simplify multiplexing of the data bus to the narrower memory.

When performing byte writes the ARM810 ensures the byte to be written appears on the data bus in the place where it is required by a 32-bit wide memory system. The ARM810 achieves this by duplicating the byte to be written four times across the 32 bit data bus on **D[31:24]**, **D[23:16**], **D[15:8]**, **D[7:0]**, so that the memory system need only enable a write into the correct byte of the memory system as shown in the tables below. Memory systems narrower than 32-bits may make use of this duplication to simplify multiplexing of the data bus to the narrower memory.

When performing word reads, the ARM810 reads the 32 bit word provided by the memory system on **D[31:0]**.

When performing halfword reads, the ARM810 reads the 16-bit halfword from the place where it would be provided by a 32-bit wide memory system. The ARM810 achieves this by reading 16 bits from the two bytes indicated in the tables below and then internally rotating and zero-extending or sign-extending the data to be in the correct position in the register at the completion of load-halfword instruction.

When performing byte reads, the ARM810 reads the byte from the place where it would be provided by a 32-bit wide memory system. The ARM810 achieves this by reading the byte indicated as active in the tables below and then internally rotating and zero-extending or sign-extending the data to be in the correct position in the register at the completion of the byte-load instruction.

Because ARM810 duplicates bytes and halfwords before presenting to the data bus for writes, and internally rotates and then sign-extends or zero-extends bytes and halfwords after reading from the data bus for reads, a 32-bit wide memory system never needs to shift, rotate, zero-extend or sign-extend data. The memory system control logic only needs to enable the appropriate bytes of the memory as shown in the tables below.

Note: The memory system design should ensure that during byte and halfword reads all bits of the data bus are driven to valid levels. Floating inputs levels on any bits of the data bus may result in increased power consumption in the ARM810 input pads. One way of ensuring this is to enable all bytes of the memory system for byte and halfword reads—the ARM810 will ignore the extra data on the data bus.

MAS[1:0] Indicates	MAS[1]	MAS[0]	A[1]	A[0]	Memory Read/Write the Byte on			
					D[31:24]	D[23:16]	D[15:8]	D[7:0]
Word	1	0	X	X	Yes	Yes	Yes	Yes
Halfword	0	1	0	X	No	No	Yes	Yes
			1	X	Yes	Yes	No	No
Byte	0	0	0	0	No	No	No	Yes
			0	1	No	No	Yes	No
			1	0	No	Yes	No	No
			1	1	Yes	No	No	No
Reserved	1	1	X	X	Yes	Yes	Yes	Yes

Table 12-4: Decoding Byte Activity for little-endian system.

Notes *X means “don’t care”.*

MAS[1:0] *= 11 is Reserved for future use, it is never used by ARM810.*

The Byte Activity Decode indicated is recommended for compatibility with future ARM Microprocessors.

MAS[1:0] Indicates	MAS[1]	MAS[0]	A[1]	A[0]	Memory Read/Write the Byte on			
					D[31:24]	D[23:16]	D[15:8]	D[7:0]
Word	1	0	X	X	Yes	Yes	Yes	Yes
Halfword	0	1	0	X	Yes	Yes	No	No
			1	X	No	No	Yes	Yes
Byte	0	0	0	0	Yes	No	No	No
			0	1	No	Yes	No	No
			1	0	No	No	Yes	No
			1	1	No	No	No	Yes
Reserved	1	1	X	X	Yes	Yes	Yes	Yes

Table 12-5: Decoding Byte Activity for big-endian system.

Notes *X means “don't care”.*

MAS[1:0] *= 11 is Reserved for future use, it is never used by ARM810.*

The Byte Activity Decode indicated is recommended for compatibility with future ARM Microprocessors.

12.13.2Decoding Byte Activity using nBLS[3:0]

This is the recommended Byte Activity decode method for all new memory controller designs if compatibility with earlier ARM Processors is not a concern.

For all accesses **A[31:2]** provides the address of the 32-bit word the access will read or write. The Byte Lane Selects **nBLS[3:0**] are active low signals which indicate directly which bytes within that word must be written or read.

nBLS[3:0] have the same timing as address, and are related to **MAS[1:0]**, **A[1:0]**, and the internal endianness setting (CP15 Control Register “B” bit) as follows:

CP15 Control Register B Bit	MAS[1:0]	A[1:0]	nBLS
0 (Little-endian)	1 0 (Word)	X X	0000
0 0	0 1 (Halfword) 0 1	0 X 1 X	1100 0011
0 0 0 0	0 0 (Byte) 0 0 0 0 0 0	0 0 0 1 1 0 1 1	1110 1101 1011 0111
1 (Big-endian)	1 0 (Word)	X X	0000
1 1	0 1 (Halfword) 0 1	0 X 1 X	0011 1100
1 1 1 1	0 0 (Byte) 0 0 0 0 0 0	0 0 0 1 1 0 1 1	0111 1011 1101 1110

Table 12-6: **nBLS[3:0]** ***as a function of B,*** **MAS[1:0]** ***and*** **A[1:0]**

Note *X means “don't care”.*

As the internal endianness configuration has been used in the generation of the **nBLS** outputs, memory systems using **nBLS** to control memory system Byte Activity can switch endianness dynamically with no added external logic.

The **nBLS[3:0**] should be used to enable the bytes of the memory system as follows

Signal	When Low, enable read or write of byte connected to data bus bits
nBLS[0]	**D[7:0]**
nBLS[1]	**D[15:8]**
nBLS[2]	**D[23:16]**
nBLS[3]	**D[31:24]**

Table 12-7: **nBLS[3:0]** ***and Bytes of memory system***

ARM810 will only output the patterns of **nBLS[3:0]** shown in ***Table 12-6: nBLS[3:0] as a function of B, MAS[1:0] and A[1:0]*** and it duplicates bytes and halfwords across the data bus on writes. Memory system designs for use with ARM810 can depend on this behaviour to reduce the amount of bus multiplexing required when connecting ARM810 to memory systems narrower than 32-bits. However, if compatibility is desired with future ARM Microprocessors the memory system should be designed to operate with arbitrary patterns appearing on **nBLS[3:0**], and depend only on the byte of data appearing on the data bus pins associated with the appropriate nBLS signal (see ***Table 12-7: nBLS[3:0] and Bytes of memory system***.

For write operations, the ARM810 will ensure that the data is available on the correct byte or bytes of the data bus as indicated by **nBLS[3:0**].

For read operations, the ARM810 will accept the data on the byte or bytes of the data bus indicated by **nBLS[3:0]**. Data returned on bytes of the data bus for which the associated **nBLS** signal is 1 will be ignored. [However see note below re. valid signal levels].

Together the ARM810 behaviour for reads and writes means that a 32 bit wide memory system never needs to shift, rotate, zero-extend or sign-extend data. The memory system control logic only needs to enable the appropriate bytes of the memory as indicated by **nBLS[3:0]**.

Note: The ARM710a microprocessor also has output pins **nBLS[3:0]**. The ARM710a does not include endianness information in the **nBLS[3:0]** outputs, and so does not support dynamic switching of endianness without some additional logic in the memory system. For further details of use of the ARM710a **nBLS[3:0]** outputs see the ARM710a Datasheet.

Note: During reads it should be ensured that all bits of the data bus are driven to a defined level. Floating inputs levels on any bits of the data bus may result in increased power consumption in the ARM810 input pads.

12.13.3Backwards compatibility with earlier ARM Processors

Halfword Instructions

ARM processor implementations prior to ARM Architecture v4 including ARM710, ARM610, ARM7, and ARM6 do not have any halfword load and store instructions and they do not perform halfword load or store accesses on their bus interfaces—they perform only byte and word accesses.

To use ARM810 with a memory controller designed for an earlier ARM processor and which was designed to use the signals nBW and A[1:0] to decode byte activity for bus accesses:

- Connect **MAS[1]** the memory controller’s **nBW** input.
- Leave **MAS[0]** disconnected.
- Connect **A[1:0]** as before.

This will allow the ARM810 to perform byte and word accesses correctly. Halfword bus operations will not work correctly, so halfword instructions cannot be used in this configuration - note however that as halfword instructions did not exist in earlier ARM processors, existing application code will not contain any of these instructions and so will operate correctly with this configuration. New software developments for a hardware platform using a memory system configured in this way must not use halfword instructions.

Byte Lane Strobes

The ARM710a microprocessor also has output pins **nBLS[3:0]**. The ARM710a does not include endianness information in the **nBLS[3:0]** outputs, and so does not support dynamic switching of endianness without some additional logic in the memory system. For further details of use of the ARM710a **nBLS[3:0]** outputs see the ARM710a Datasheet.

12.14Memory Access Sequence Summary

ARM810 performs many different bus access sequences, and all are constructed out of combinations of non-sequential and sequential accesses. There may be any number of idle cycles between two other memory accesses. If a memory access is followed by an idle period on the bus (as opposed to another non-sequential access), then the address, and the signal **nRW** and **nBW** will remain at their previous value in order to avoid unnecessary bus transitions.

The accesses performed by an ARM810 are:

Unbuffered WriteLevel 1 translation fetch

Uncached ReadLevel 2 translation fetch

Buffered WriteRead-Lock-Write sequence

Linefetch

See also ***12.15 ARM810 Cycle Type Summary*** on page 12-33.

12.14.1Unbuffered writes / uncacheable reads

These are the most basic access types. Apart from the difference between read and write, they are the same. Each may consist of a single (LDR/STR) or multiple (LDM/STM) access. A multiple access consists of a non-sequential access followed by a sequential access. These cycles always reflect the type (ie. read/write, byte/word/halfword) of the instruction requesting the cycle.

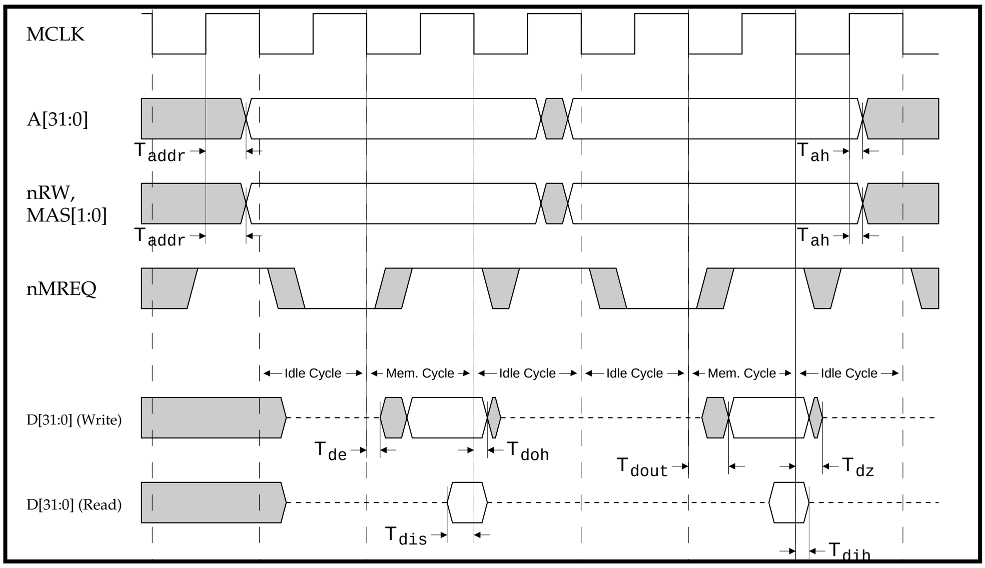


Figure 12-15: Two single word non-sequential unbuffered accesses

12.14.2Buffered write

The external bus cycle of a buffered write is identical to and indistinguishable from the bus cycle of an unbuffered write. However there may only be a single idle cycle between a buffered write, and the next access on the bus. These cycles always reflect the type (byte/word/halfword) of the instruction requesting the cycle. Note that if several write accesses are stored concurrently within the write buffer, then each burst will start with a non-sequential access, followed by subsequent sequential cycles.

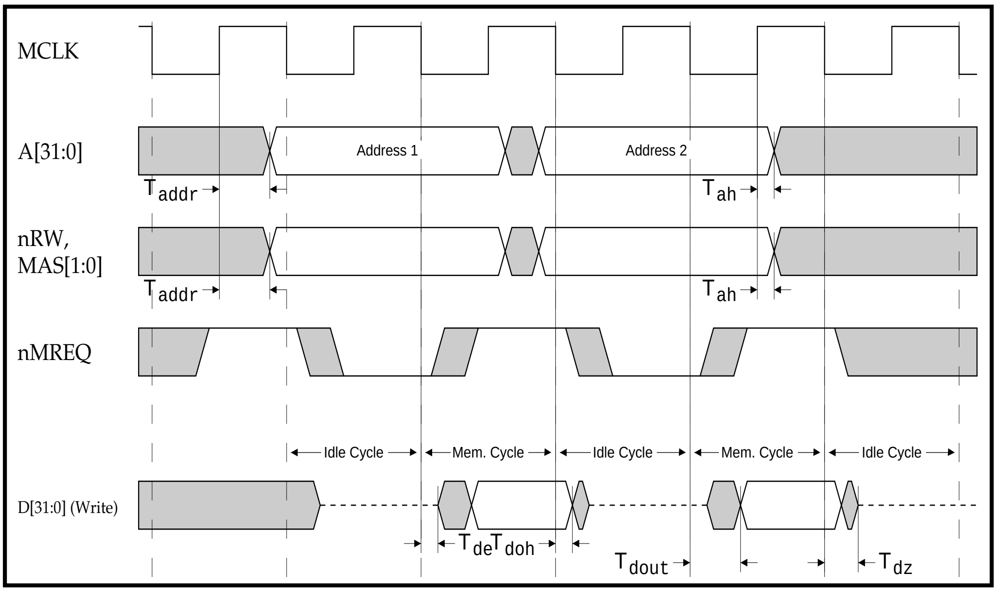


Figure 12-16: Two single word non-sequential buffered writes

Note that in the case of a pair of buffered writes, there may only be a single idle cycle between the two accesses.

12.14.3Linefetch

This access appears on the bus as a non-sequential access followed by three sequential accesses. Note that linefetch accesses always start on a 4-word boundary, and are always word accesses. Even if the instruction which caused the linefetch was a byte load instruction (eg. LDRB), the linefetch accesses will be word accesses on the bus. ***Figure 12-17: Linefetch*** shows a linefetch.

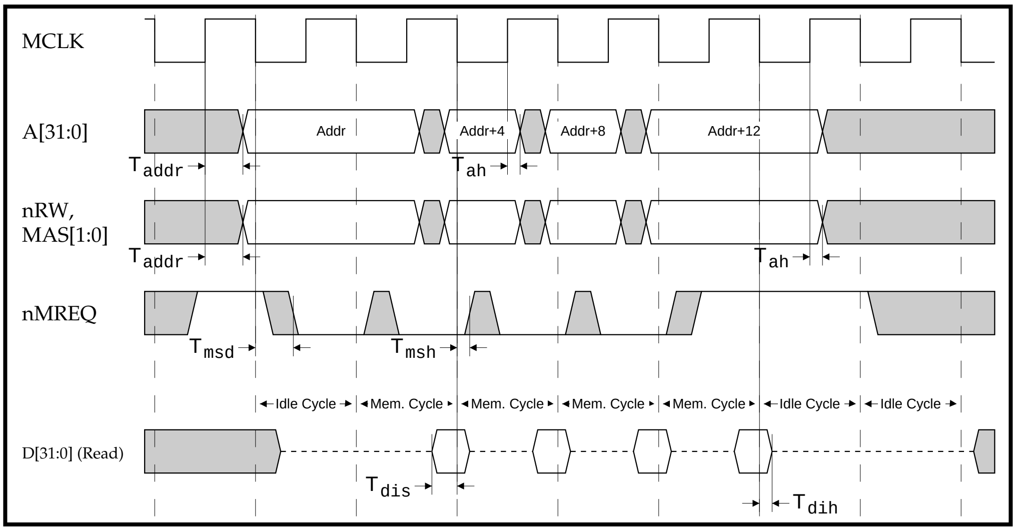


Figure 12-17: Linefetch

A linefetch may be safely aborted on any word in the transfer. If an abort occurs on any word during the linefetch, the line will not be placed in the Cache, as it is assumed to be invalid. If the abort occurs on a word that has been requested by the ARM core, the core will be aborted, otherwise the cache line will be invalidated but program flow will *not* be interrupted.

12.14.4Translation fetches

These accesses are required to obtain the translation data for an access. There are two types, level 1 and level 2. A level 1 access is required for a section-mapped memory location, and a level 2 access is required for a page mapped memory location. A Level 2 access is always preceded by a level 1 access. Note that these translation fetches are often immediately followed by a data access. In fact the translation fetch held up the data access because the translation was not contained in the Translation Lookaside Buffer (TLB). Translation fetches are always read word accesses. So if a byte or write (or both) access was not possible because the address was not contained in the TLB, the access would be preceded by the translation fetch(es) which would always be word read accesses.

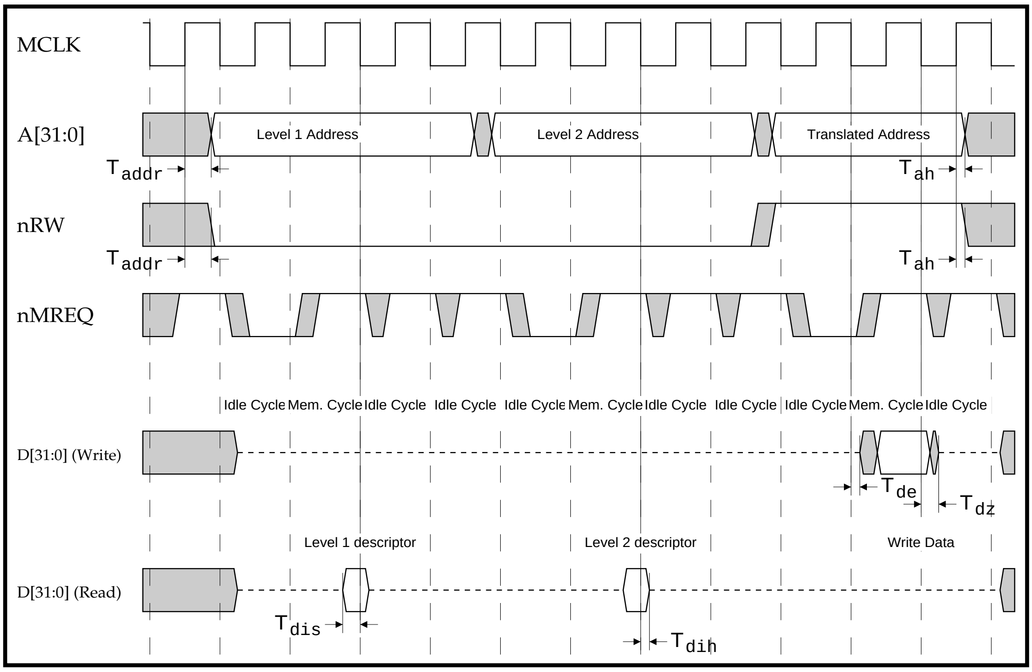


Figure 12-18: Translation table-walking sequence for page

The translation fetch diagrams show a page table walk caused by a write access that missed the TLB. The diagrams show the relationship of the page table walk and the access. The access could have equally well been a read.

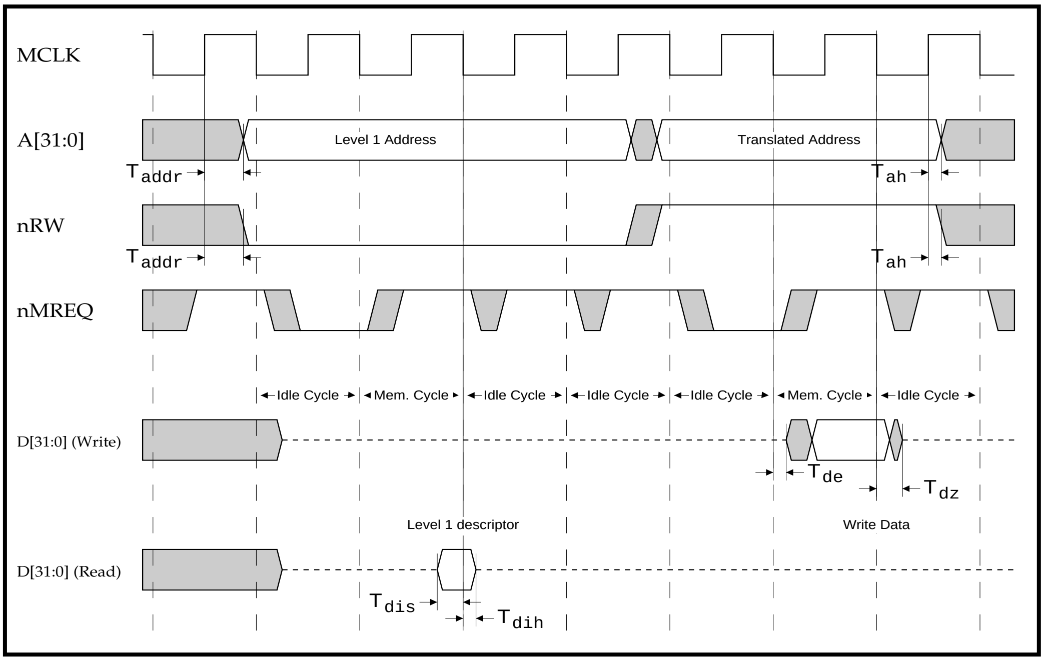


Figure 12-19: Translation table-walking sequence for section

12.15 ARM810 Cycle Type Summary

	Operation	nRW	A[31:0]	nMREQ	D[31:0]
	Idle	old	old	i	
	Linefetch	read	a	idle	
		read	a	memory	
		read	a+4	memory	data
		read	a+8	memory	data
		read	a+12	memory	data
		read	a+12	idle	data
Uncacheable Read / Unbuffered Write	Start	r/w	a	idle	
		r/w	a	memory	
					data
	Repeat	r/w	a+n	memory	
					data
	End	r/w	old	idle	
Buffered Write	Start	write	a	idle	
		write	a	memory	
					data
	Repeat	write	a+n	memory	
					data
Read-Lock-Write	Read phase	read	aL	idle	
		read	aL	memory	
		read	aL	idle	data
	Write phase	write	aL	idle	
		write	aL	idle	
		write	aL	idle	
		write	aL	memory	
		write	aL	idle	data
	Write phase after aborted read	read	a	idle	
		read	a	idle	
		read	a	idle	
Section Translation Fetch	Start	read	l1a	idle	
		read	l1a	memory	

Table 12-8: Cycle type summary

	Operation	nRW	A[31:0]	nMREQ	D[31:0]
		read	l1a	idle	data
Page Translation Fetch	Start	read	l1a	idle	
		read	l1a	memory	
		read	l1a	idle	data
		read	l1a	idle	
		read	l2a	idle	
		read	l2a	memory	
		read	l2a	idle	data

Table 12-8: Cycle type summary

Key to cycle type summary:

read	Read (**nRW** LOW)
r/w	applies equally to Read and Write
write	Write (**nRW** HIGH)
old	signal remains at previous value
a	first Address
a+n	next sequential address
aL	Read-Lock-Write Address
l1a	Level 1 translation Table address
l2a	Level 2 translation Table address
idle	Idle cycle (**nMREQ** HIGH)
memory	Memory cycle (**nMREQ** LOW)
data	valid data on data bus

Each line in ***Table 12-8: Cycle type summary*** on page 12-33 shows the state of the bus interface during a single **MCLK** cycle. It illustrates the pipelining of **nMREQ** and the address. Each operation type section shows the sequence of cycles which make up that type of access, with each line down the diagram showing successive clock cycles.

The uncached read / unbuffered write is shown in three sections. The start and end are always present, with the repeat section repeated as many times as required when a multiple access is being performed.

Buffered Writes are also of variable length and consist of the start section plus as many consecutive repeat sections as are necessary.

A swap instruction consists of the read phase, followed by one of the two possible write phases.

Activity on the memory interface is the succession of these access sequences.

13 Boundary-Scan Test Interface

This chapter describes the boundary-scan interface.

13.1 Overview 13-3
13.2 Reset 13-4
13.3 Pullup Resistors 13-5
13.4 Instruction Register 13-6
13.5 Public Instructions 13-7
13.6 Test Data Registers 13-10
13.7 Boundary-Scan Interface Signals 13-13

Boundary-Scan Test Interface

The boundary-scan interface conforms to the IEEE Std. 1149.1- 1990, Standard Test Access Port and Boundary-Scan Architecture (refer to this standard for an explanation of the terms used in this section and for a description of the TAP controller states).

13.1 Overview

The boundary-scan interface provides a means of testing the core of the device when it is fitted to a circuit board, and a means of driving and sampling all the external pins of the device irrespective of the core state. This latter function permits testing of both the device's electrical connections to the circuit board, and (in conjunction with other devices on the circuit board having a similar interface) testing the integrity of the circuit board connections between devices. The interface intercepts all external connections within the device, and each such “cell” is then connected together to form a serial register (the boundary scan register). The whole interface is controlled via 5 dedicated pins: **TDI**, **TMS**, **TCK**, **nTRST** and **TDO**. ***Figure 13-1: Test Access Port (TAP) Controller State Transitions*** shows the state transitions that occur in the TAP controller.

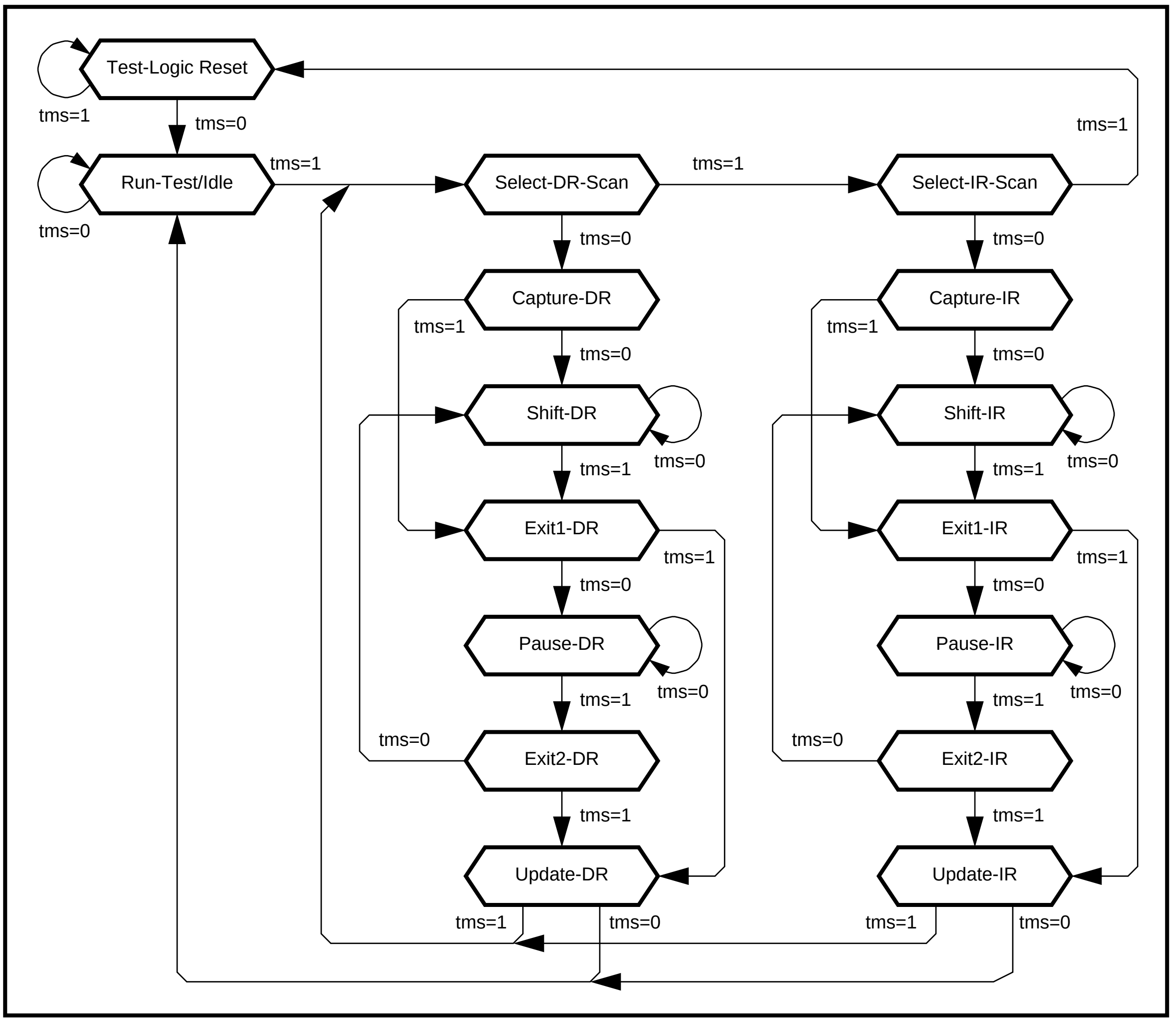


Figure 13-1: Test Access Port (TAP) Controller State Transitions

13.2 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** pin. 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** pin may be tied permanently LOW. Note that a clock on **TCK** is not necessary to reset the device.

The action of reset (either a pulse or a DC level) is as follows:

System mode is selected (i.e. the boundary scan chain does *not* intercept any of the signals passing between the pads and the core).

IDcode mode is selected. If **TCK** is pulsed, the contents of the ID register will be clocked out of **TDO**.

13.3 Pullup Resistors

The IEEE 1149.1 standard effectively requires that **TDI**, **nTRST** and **TMS** should have internal pullup resistors. In order to minimise static current draw, these resistors are *not* fitted to ARM810. 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.

13.4 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.

13.5 Public Instructions

The following public instructions are supported:

Instruction	**Binary Code**
EXTEST	0000
SAMPLE/PRELOAD	0011
CLAMP	0101
HIGHZ	0111
CLAMPZ	1001
IDCODE	1110
BYPASS	1111

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**.

13.5.1 EXTEST (0000)

The BS (boundary-scan) register is placed in test mode by the EXTEST instruction.

The EXTEST instruction connects the BS register between **TDI** and **TDO**.

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

In the CAPTURE-DR state, inputs from the system pins and outputs from the boundary-scan output cells to the system pins are captured by the boundary-scan cells. In the SHIFT-DR state, the previously captured test data is shifted out of the BS register via the **TDO** pin, whilst new test data is shifted in via the **TDI** pin to the BS register parallel input latch. In the UPDATE-DR state, the new test data is transferred into the BS register parallel output latch. Note that this data is applied immediately to the system logic and system pins. The first EXTEST vector should be clocked into the boundary-scan register, using the SAMPLE/PRELOAD instruction, prior to selecting INTEST to ensure that known data is applied to the system logic.

13.5.2 SAMPLE/PRELOAD (0011)

The BS (boundary-scan) register is placed in normal (system) mode by the SAMPLE/PRELOAD instruction.

The SAMPLE/PRELOAD instruction connects the BS register between **TDI** and **TDO**.

When the instruction register is loaded with the SAMPLE/PRELOAD instruction, all the boundary-scan cells are placed in their normal system mode of operation.

In the CAPTURE-DR state, a snapshot of the signals at the boundary-scan cells is taken on the rising edge of **TCK**. Normal system operation is unaffected. In the SHIFT-DR state, the sampled test data is shifted out of the BS register via the **TDO** pin, whilst new data is shifted in via the **TDI** pin to preload the BS register parallel input latch. In the UPDATE-DR state, the preloaded data is transferred into the BS register parallel output latch. Note that this data is not applied to the system logic or system pins while the SAMPLE/PRELOAD instruction is active. This instruction should be used to preload the boundary-scan register with known data prior to selecting the INTEST or EXTEST instructions (see the table below for appropriate guard values to be used for each boundary-scan cell).

13.5.3 CLAMP (0101)

The CLAMP 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 output signals is defined by the values previously loaded into the boundary-scan register. A guarding pattern (specified for this device at the end of this section) should be pre-loaded into the boundary-scan register using the SAMPLE/PRELOAD instruction prior to selecting the CLAMP instruction.

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.

13.5.4 HIGHZ (0111)

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

When the HIGHZ instruction is loaded into the instruction register, all outputs are placed in an inactive drive state.

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.

13.5.5 CLAMPZ (1001)

The CLAMPZ 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 outputs are placed in an inactive drive state, but the data supplied to the disabled output drivers is derived from the boundary-scan cells. The purpose of this instruction is to ensure, during production testing, that each output driver can be disabled when its data input is either a 0 or a 1.

A guarding pattern (specified for this device at the end of this section) should be pre-loaded into the boundary-scan register using the SAMPLE/PRELOAD instruction prior to selecting the CLAMPZ instruction.

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.

Single-step operation is possible using the INTEST instruction.

13.5.6 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 boundary-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, whilst data is shifted in via the **TDI** pin into the ID register. In the UPDATE-DR state, the ID register is unaffected.

13.5.7 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 boundary-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.

13.6 Test Data Registers

Figure 13-2: Boundary-scan block diagram illustrates the structure of the boundary scan logic.

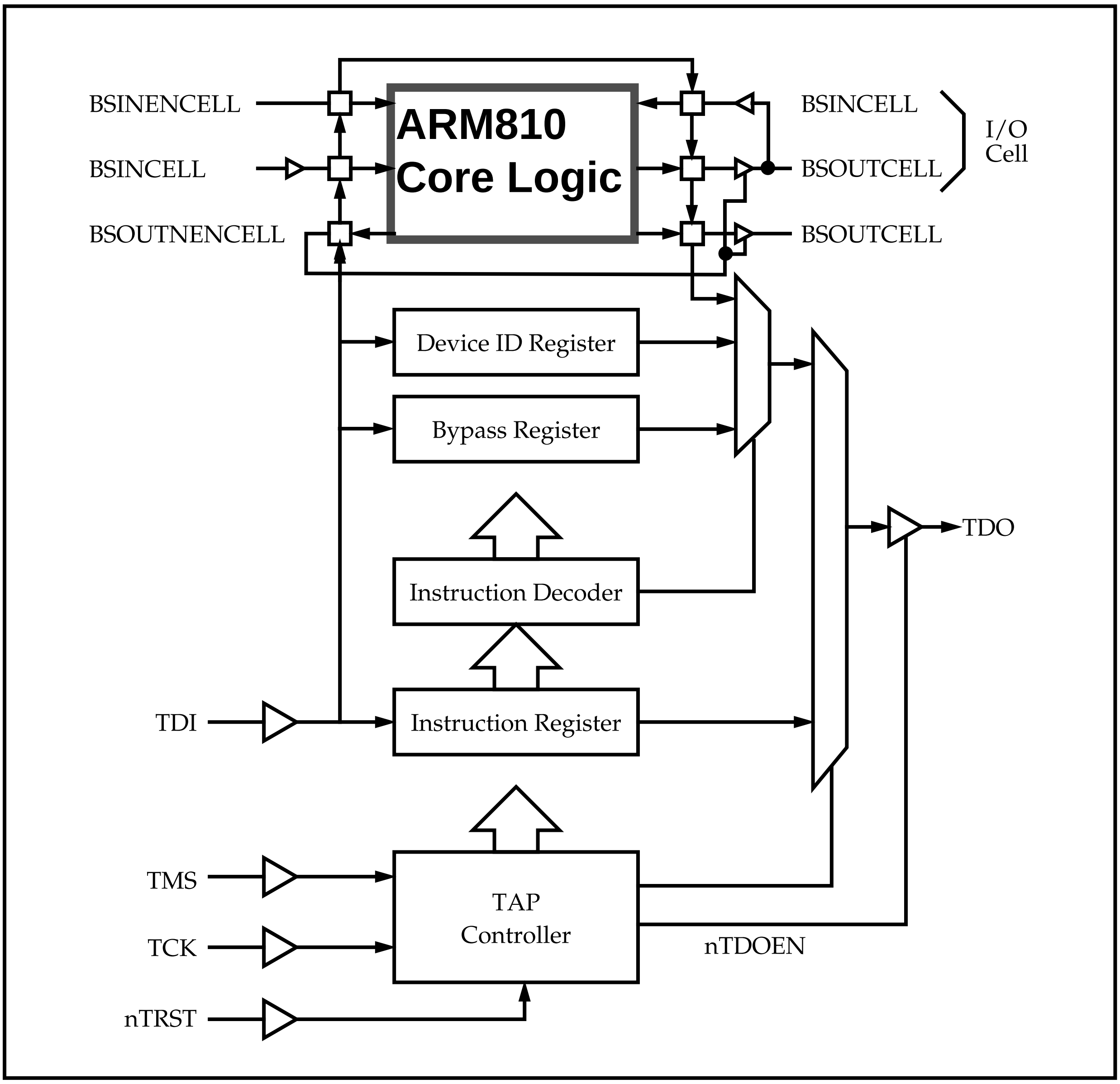


Figure 13-2: Boundary-scan block diagram

13.6.1 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 boundary-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.

13.6.2 ARM810 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:

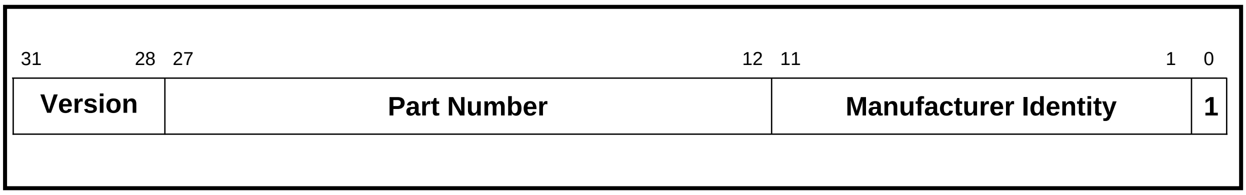


Figure 13-3: ID register format

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.

13.6.3 ARM810 boundary scan (BS) register

Purpose: The BS register consists of a serially connected set of cells around the periphery of the device, at the interface between the core logic and the system input/output pads. This register can be used to isolate the core logic from the pins and then apply tests to the core logic, or conversely to isolate the pins from the core logic and then drive or monitor the system pins.

Operating modes: The BS register is selected as the register to be connected between **TDI** and **TDO** only during the SAMPLE/PRELOAD, EXTEST and INTEST instructions. Values in the BS register are used, but are not changed, during the CLAMP and CLAMPZ instructions.

In the normal (system) mode of operation, straight-through connections between the core logic and pins are maintained and normal system operation is unaffected.

In TEST mode (ie when either EXTEST or INTEST is the currently selected instruction), values can be applied to the core logic or output pins independently of the actual values on the input pins and core logic outputs respectively. On the ARM810 all of the boundary scan cells include an update register and thus all of the pins can be controlled in the above manner. Additional boundary-scan cells are interposed in the scan chain in order to control the enabling of tristateable buses.

The correspondence between boundary-scan cells and system pins, system direction controls and system output enables is as shown in ***Table 13-2: Boundary scan chain description*** on page 13-15. The cells are listed in the order in which they are connected in the boundary-scan register, starting with the cell closest to **TDI**. All boundary-scan register cells at input pins can apply tests to the on-chip core logic.

The EXTEST guard values specified in ***Table 13-2: Boundary scan chain description*** on page 13-15 should be clocked into the boundary-scan register (using the SAMPLE/PRELOAD instruction) before the EXTEST instruction is selected, to ensure that known data is applied to the core logic during the test. The INTEST guard values shown in the table below should be clocked into the boundary-scan register (using the SAMPLE/PRELOAD instruction) before the INTEST instruction is selected to ensure that all outputs are disabled. These guard values should also be used when new EXTEST or INTEST vectors are clocked into the boundary-scan register.

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

13.6.4 Output enable boundary-scan cells

The boundary-scan register cells **Nendout**, Nabe, Ntbe, and Nmse control the output drivers of tristate outputs as shown in the table below. In the case of OUTEN0 enable cells (**Nendout**, Ntbe), loading a 1 into the cell will place the associated drivers into the tristate state, while in the case of type INEN1 enable cells (Nabe, Nmse), loading a 0 into the cell will tristate the associated drivers.

To put all ARM810 tristate outputs into their high impedance state, a logic 1 should be clocked into the output enable boundary-scan cells **Nendout** and Ntbe, and a logic 0 should be clocked into Nabe and Nmse. Alternatively, the HIGHZ instruction can be used.

For example, if the on-chip core logic causes the drivers controlled by **Nendout** to be tristate, (ie by driving the signal **nENDOUT** HIGH), then a 1 will be observed on this cell if the SAMPLE/PRELOAD or INTEST instructions are active.

13.7 Boundary-Scan Interface Signals

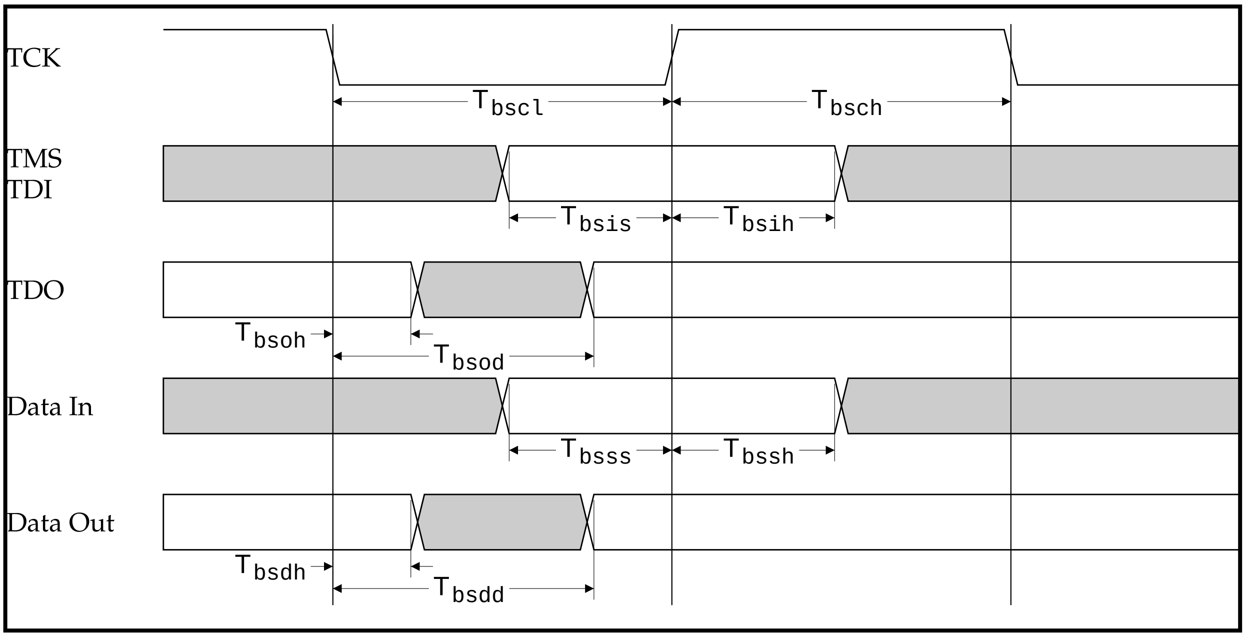


Figure 13-4: Boundary-scan general timing

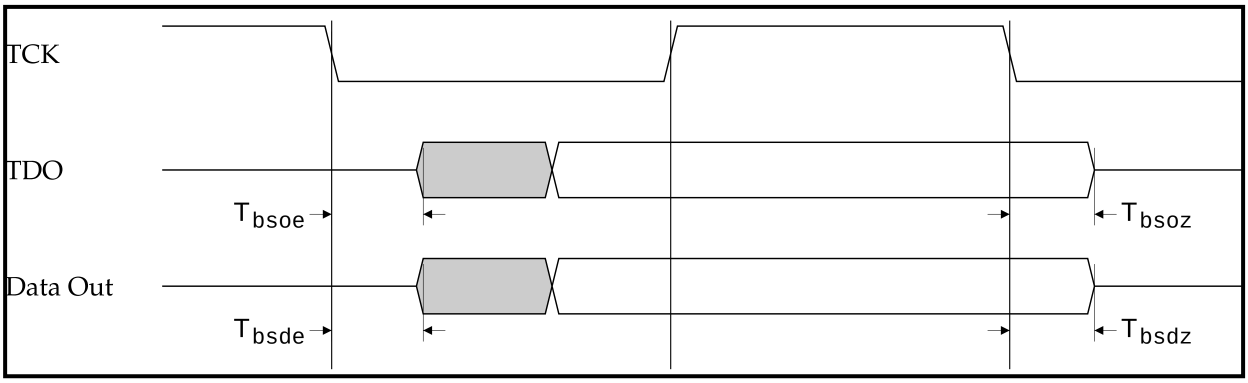


Figure 13-5: Boundary-scan tri-state timing

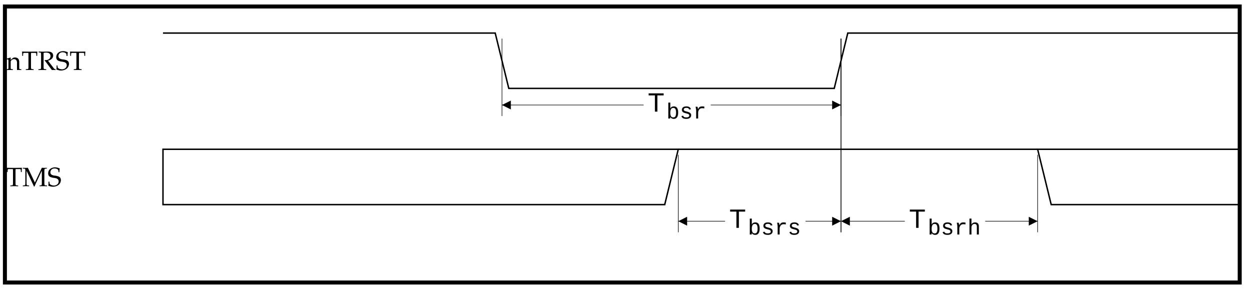


Figure 13-6: Boundary-scan reset timing

Symbol	Parameter	Min	Typ	Max	Units	Notes
Tbscl	**TCK** low period	50			ns	9
Tbsch	**TCK** high period	50			ns	9
Tbsis	**TDI**,**TMS** setup to [TCr]	10			ns	
Tbsih	**TDI**,**TMS** hold from [TCr]	10			ns	
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				ns	1,3
Tbsde	data output enable time	5			ns	5,6
Tbsdz	data output disable time			40	ns	5,7
Tbsr	Reset period	30			ns	
Tbsrs	tms setup to [TRr]	10			ns	9
Tbsrh	tms hold from [TRr]	10			ns	9

Table 13-1: ARM810 boundary-scan interface timing

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 SAMPLE/PRELOAD, 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 boundary-scan logic is used to enable the output drivers.

7 Data output disable time applies when the boundary scan is used to disable the output drivers.

8 **TMS** must be held high as **nTRST** is taken high at the end of the boundary-scan reset sequence.

9 **TCK** may be stopped indefinitely in either the low or high phase.

As the signal list of the ARM810 is still preliminary so is the boundary scan order .

Cell No. from tdi	**Cell Name**	**Pin**	**Type**	**Output Enable**
1	bsNencon	(TnOEnCon)	OUTNEN	
2	bsNrw	nRW	OUT	TnOEnCon
3	bsbls	nBLS[3]	OUT	TnOEnCon
4	bsbls	nBLS[2]	OUT	TnOEnCon
5	bsbls	nBLS[1]	OUT	TnOEnCon
6	bsbls	nBLS[0]	OUT	TnOEnCon
7	bsdata	D[31]	IN	
8	bsdata	D[31]	OUT	TnOEnD
9	bsdata	D[30]	IN	
10	bsdata	D[30]	OUT	TnOEnD
11	bsdata	D[29]	IN	
12	bsdata	D[29]	OUT	TnOEnD
13	bsdata	D[28]	IN	
14	bsdata	D[28]	OUT	TnOEnD
15	bsdata	D[27]	IN	
16	bsdata	D[27]	OUT	TnOEnD
17	bsdata	D[26]	IN	
18	bsdata	D[26]	OUT	TnOEnD

Table 13-2: Boundary scan chain description

Cell No. from tdi	Cell Name	Pin	Type	Output Enable
19	bsdata	D[25]	IN	
20	bsdata	D[25]	OUT	TnOEnD
21	bsdata	D[24]	IN	
22	bsdata	D[24]	OUT	TnOEnD
23	bsdata	D[23]	IN	
24	bsdata	D[23]	OUT	TnOEnD
25	bsdata	D[22]	IN	
26	bsdata	D[22]	OUT	TnOEnD
27	bsdata	D[21]	IN	
28	bsdata	D[21]	OUT	TnOEnD
29	bsdata	D[20]	IN	
30	bsdata	D[20]	OUT	TnOEnD
31	bsdata	D[19]	IN	
32	bsdata	D[19]	OUT	TnOEnD
33	bsdata	D[18]	IN	
34	bsdata	D[18]	OUT	TnOEnD
35	bsdata	D[17]	IN	
36	bsdata	D[17]	OUT	TnOEnD
37	bsdata	D[16]	IN	
38	bsdata	D[16]	OUT	TnOEnD
39	bsdata	D[15]	IN	
40	bsdata	D[15]	OUT	TnOEnD
41	bsdata	D[14]	IN	
42	bsdata	D[14]	OUT	TnOEnD
43	bsdata	D[13]	IN	
44	bsdata	D[13]	OUT	TnOEnD
45	bsdata	D[12]	IN	
46	bsdata	D[12]	OUT	TnOEnD

Table 13-2: Boundary scan chain description (Continued)

Cell No. from tdi	**Cell Name**	**Pin**	**Type**	**Output Enable**
47	bsdata	D[11]	IN	
48	bsdata	D[11]	OUT	TnOEnD
49	bsdata	D[10]	IN	
50	bsdata	D[10]	OUT	TnOEnD
51	bsdata	D[09]	IN	
52	bsdata	D[09]	OUT	TnOEnD
53	bsdata	D[08]	IN	
54	bsdata	D[08]	OUT	TnOEnD
55	bsdata	D[07]	IN	
56	bsdata	D[07]	OUT	TnOEnD
57	bsdata	D[06]	IN	
58	bsdata	D[06]	OUT	TnOEnD
59	bsdata	D[05]	IN	
60	bsdata	D[05]	OUT	TnOEnD
61	bsdata	D[04]	IN	
62	bsdata	D[04]	OUT	TnOEnD
63	bsdata	D[03]	IN	
64	bsdata	D[03]	OUT	TnOEnD
65	bsdata	D[02]	IN	
66	bsdata	D[02]	OUT	TnOEnD
67	bsdata	D[01]	IN	
68	bsdata	D[01]	OUT	TnOEnD
69	bsdata	D[00]	IN	
70	bsdata	D[00]	OUT	TnOEnD
71	bsNendata	(TnOEnD)	OUTNEN	
72	bsdbe	DBE	OUTENIN	
73	bsmclk	MClk	IN	
74	bspclk	PClk	IN	

Table 13-2: Boundary scan chain description (Continued)

Cell No. from tdi	Cell Name	Pin	Type	Output Enable
75	bsrefclk	REFCLK	IN	
76	bsnwait	nWAIT	IN	
77	bspllrange	PLLRANGE	IN	
78	bspllrange	PLLRANGE	OUT	TnOEnRCLKCFG
79	bspllsleep	PLLSLEEP	IN	
80	bsrefclkcfg	REFCLKCFG[1]	IN	
81	bsrefclkcfg	REFCLKCFG[1]	OUT	TnOEnRCLKCFG
82	bsrefclkcfg	REFCLKCFG[0]	IN	
83	bsrefclkcfg	REFCLKCFG[0]	OUT	TnOEnRCLKCFG
84	bsNenrccfg	(TnOEnRCLKCFG)	OUTNEN	
85	bsmreq	nMREQ	OUT	TnOEnMSE
86	bsseq	SEQ	OUT	TnOEnMSE
87	bsNenmse	(TnOEnMSE)	OUTNEN	
88	bsmse	MSE	OUTENIN	
89	bspllcfg	PLLCFG[6]	IN	
90	bspllcfg	PLLCFG[5]	IN	
91	bspllcfg	PLLCFG[4]	IN	
92	bspllcfg	PLLCFG[3]	IN	
93	bspllcfg	PLLCFG[2]	IN	
94	bspllcfg	PLLCFG[1]	IN	
95	bspllcfg	PLLCFG[0]	IN	
96	bsabort	ABORT	IN	
97	bsNfiq	nFIQ	IN	
98	bsNreset	nRESET	IN	
99	bsNirq	nIRQ	IN	
100	bstestmode	TESTMODE	IN	
101	bstestout	TESTOUT[4]	OUT	TnOEnTESTOUT
102	bstestout	TESTOUT[3]	OUT	TnOEnTESTOUT
103	bstestout	TESTOUT[2]	OUT	TnOEnTESTOUT
104	bstestout	TESTOUT[1]	OUT	TnOEnTESTOUT

Table 13-2: Boundary scan chain description (Continued)

Cell No. from tdi	Cell Name	Pin	Type	Output Enable
105	bstestout	TESTOUT[0]	OUT	TnOEnTESTOUT
106	bsNentestout	(TnOEnTESTOUT)	OUTNEN	
107	bsape	APE	IN	
108	bsabe	ABE	OUTENIN	
109	bsNenabe	(TnOEnA)	OUTNEN	
110	bsa	A[31]	IN	
111	bsa	A[31]	OUT	TnOEnA
112	bsa	A[30]	IN	
113	bsa	A[30]	OUT	TnOEnA
114	bsa	A[29]	IN	
115	bsa	A[29]	OUT	TnOEnA
116	bsa	A[28]	IN	
117	bsa	A[28]	OUT	TnOEnA
118	bsa	A[27]	IN	
119	bsa	A[27]	OUT	TnOEnA
120	bsa	A[26]	IN	
121	bsa	A[26]	OUT	TnOEnA
122	bsa	A[25]	IN	
123	bsa	A[25]	OUT	TnOEnA
124	bsa	A[24]	IN	
125	bsa	A[24]	OUT	TnOEnA
126	bsa	A[23]	IN	
127	bsa	A[23]	OUT	TnOEnA
128	bsa	A[22]	IN	
129	bsa	A[22]	OUT	TnOEnA
130	bsa	A[21]	IN	
131	bsa	A[21]	OUT	TnOEnA
132	bsa	A[20]	IN	
133	bsa	A[20]	OUT	TnOEnA
134	bsa	A[19]	IN	

Table 13-2: Boundary scan chain description (Continued)

Cell No. from tdi	Cell Name	Pin	Type	Output Enable
135	bsa	A[19]	OUT	TnOEnA
136	bsa	A[18]	IN	
137	bsa	A[18]	OUT	TnOEnA
138	bsa	A[17]	IN	
139	bsa	A[17]	OUT	TnOEnA
140	bsa	A[16]	IN	
141	bsa	A[16]	OUT	TnOEnA
142	bsa	A[15]	IN	
143	bsa	A[15]	OUT	TnOEnA
144	bsa	A[14]	IN	
145	bsa	A[14]	OUT	TnOEnA
146	bsa	A[13]	IN	
147	bsa	A[13]	OUT	TnOEnA
148	bsa	A[12]	IN	
149	bsa	A[12]	OUT	TnOEnA
150	bsa	A[11]	IN	
151	bsa	A[11]	OUT	TnOEnA
152	bsa	A[10]	IN	
153	bsa	A[10]	OUT	TnOEnA
154	bsa	A[09]	IN	
155	bsa	A[09]	OUT	TnOEnA
156	bsa	A[08]	IN	
157	bsa	A[08]	OUT	TnOEnA
158	bsa	A[07]	IN	
159	bsa	A[07]	OUT	TnOEnA
160	bsa	A[06]	IN	
161	bsa	A[06]	OUT	TnOEnA
162	bsa	A[05]	IN	
163	bsa	A[05]	OUT	TnOEnA
164	bsa	A[04]	IN	

Table 13-2: Boundary scan chain description (Continued)

Cell No. from tdi	Cell Name	Pin	Type	Output Enable
165	bsa	A[04]	OUT	TnOEnA
166	bsa	A[03]	IN	
167	bsa	A[03]	OUT	TnOEnA
168	bsa	A[02]	IN	
169	bsa	A[02]	OUT	TnOEnA
170	bsa	A[01]	IN	
171	bsa	A[01]	OUT	TnOEnA
172	bsa	A[00]	IN	
173	bsa	A[00]	OUT	TnOEnA
174	bslock	LOCK	OUT	TnOEnCon
175	bsclf	CLF	OUT	TnOEnCon
176	bsmas	MAS[1]	OUT	TnOEnCon
177	bsmas	MAS[0]	OUT	TnOEnCon

Table 13-2: Boundary scan chain description (Continued)

Key

IN	Input pad.
OUT	Output pad.
OUTNEN	An internal signal which is Active low output enable controlling output or bidirectional pads. Internal signal name is given in Pin column in braces, eg “(TnOEncon)”.
OUTENIN	Active high input pin which controls pad output enables. In some cases gated with other internal signals, eg, DBE high only enables D[31:0] output pads when ARM810 is performing a write operation on the external bus. DBE is ignored at other times.

14 ARM810 DC Parameters

This chapter describes the DC Parameters.The information in this chapter is provided as a guide only. Refer to your semiconductor vendor for definitive DC parameters.

14.1	Absolute Maximum Ratings	14-2
14.2	DC Operating Conditions	14-2
14.3	Input Thresholds	14-3
14.4	DC Characteristics	14-3

ARM810 DC Parameters

14.1 Absolute Maximum Ratings

Symbol	Parameter	Min	Max	Units	Note
VDD	Supply voltage	VSS-0.3	VSS+4.0	V	1
VCC	Pad voltage reference	VSS-0.3	VSS+5.5	V	1
Vip	Voltage applied to any pin	VSS-0.3	VCC+0.3	V	1
Ts	Storage temperature	-40	125	deg C	1

Table 14-1: ARM810 DC maximum ratings

Note: 1 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.

14.2 DC Operating Conditions

Symbol	Parameter	Min	Typ	Max	Units	Notes
VDD	Supply voltage	3.0	3.3	3.6	V	
Vohc	OCZ and IOCZ output HIGH voltage	2.4		VDD	V	1,2,3
Volc	OCZ and IOCZ output LOW voltage	0.0		0.4	V	1,2,3
Ta	Ambient operating temperature	0		70	C	

Table 14-2: ARM810 DC operating conditions

Notes:

1 Voltages measured with respect to VSS.

2 OCZ - Output, CMOS levels, tri-stateable
IOCZ - Input/Output, CMOS levels, tri-stateable

3 Measured with 2mA load on output

14.3 Input Thresholds

The following table gives input thresholds.

Symbol	Parameter	Min	Max	Units	Notes
Vihc	Input HIGH voltage	2.3	VDD	V	1
Vilc	Input LOW voltage	0.0	1.0	V	1

Table 14-3: Input thresholds

Notes: 1 Voltages measured with respect to VSS.

14.4 DC Characteristics

Symbol	Parameter	Nom	Units	Note
IDD	Static Supply current	20	µA	
Isc	Output short circuit current	100	mA	
Ilu	DC latch-up current	>500	mA	
Iin	IC input leakage current	1	µA	
Ioh	Output HIGH current (Vout = VDD-0.4V)	2	mA	
Iol	Output LOW current (Vout = VSS+0.4V)	7	mA	
Cin	Input capacitance	7	pF	
ESD	HMB model ESD	4	KV	1

Table 14-4: ARM810 DC characteristics

Notes: 1 ESD - 2 KV minimum

Refer to ***Appendix A, Use of the ARM810 in a 5V TTL System*** for more information.

15 ARM810 AC Parameters

This chapter describes the AC Parameters.The information in this chapter is provided as a guide only. Refer to your semiconductor vendor for definitive DC parameters.

15.1 Test Conditions 15-2

15.2 Clocking 15-3

15.3 Main Bus Signals 15-6

ARM810 AC Parameters

15.1 Test Conditions

The AC timing parameters presented in this section assume that the outputs of ARM810 have been loaded with the capacitive loads shown in the Test Load column of the table below; these loads have been chosen as typical of the system in which ARM810 might be employed. The output pads of ARM810 are CMOS drivers which exhibit a propagation delay that increases linearly with the increase in load capacitance. An "Output derating" figure is given for each output pad, showing the approximate rate of increase of output time with increasing load capacitance.

Output Signal	Test Load (pF)	Output Derating (ns/pF)	
		Rise	Fall
A[31:0]	50	0.04	0.06
nBLS	50	0.04	0.06
CLF	50	0.04	0.06
D[31:0]	50	0.04	0.06
nR/W	50	0.04	0.06
nB/W	50	0.04	0.06
LOCK	50	0.04	0.06
MAS[1:0]	50	0.04	0.06
nMREQ	50	0.04	0.06
SEQ	50	0.04	0.06

Table 15-1: ARM810 AC test conditions

15.2 Clocking

Fast Clock from Bus Clock

Symbol	Parameter	Min	Max	Unit	Note
Tmclkl_fb	MCLK LOW time	10		ns	1,2
Tmclkh_fb	MCLK HIGH time	10		ns	1,2
Tmclkc_fb	MCLK cycle time	20		ns	1,2
Tpclkh_fb	PCLK LOW time	10		ns	1,2
Tpclkl_fb	PCLK HIGH time	10		ns	1,2
Tpclkc_fb	PCLK cycle time	20		ns	1,2

Table 15-2: Timing, fast clock from bus clock

Fast Clock from Output of PLL

Symbol	Parameter	Min	Max	Unit	Note
Tpllrefclkl	REFCLK low time	10		ns	2,3
Tpllrefclkh	REFCLK high time	10		ns	2,3
Fpllrefclk	REFCLK Frequency	1	80	MHz	2,3
Fpllclkin	PLL Clock Input Freq	1	10	MHz	2,3
Fpllclkout	PLL Clock Output Freq	25	66	MHz	2,3,4

Table 15-3: Timing, fast clock from output of PLL

Fast Clock direct from REFCLK

Symbol	Parameter	Min	Max	Unit	Note
Trefclkl	REFCLK low time	7.5		ns	2,4
Trefclkh	REFCLK high time	7.5		ns	2,4
Trefclkc	REFCLK cycle time	15		ns	2,4

Table 15-4: Timing, fast clock direct from REFCLK

ARM810 AC Parameters

Bus Clock

Bus clock timing for all modes except “Fast Clock from Bus Clock”.

Symbol	Parameter	Min	Max	Unit	Note
Tmclkl	MCLK low time	10		ns	1,2
Tmclkh	MCLK high time	10		ns	1,2
Tmclkc	MCLK cycle time	20		ns	1,2
Tpclkl	PCLK low time	10		ns	1,2
Tpclkh	PCLK high time	10		ns	1,2
Tpclkc	PCLK cycle time	20		ns	1,2

Table 15-5: Timing, bus clocks

Relationship between REFCLK and Bus Clock in Synchronous Mode.

This timing relationship must be maintained between the external clock pins when ARM810 is being used in Synchronous clocking mode. Note that this requirement only arises when both the fast clock and the bus clock are being provided directly from two separate external pins. It does not apply when the PLL is being used to generate the fast clock—in that case Asynchronous clocking mode must be used. It does not apply when the bus clock is selected as the source of the fast clock—in that case there is only 1 external clock pin.

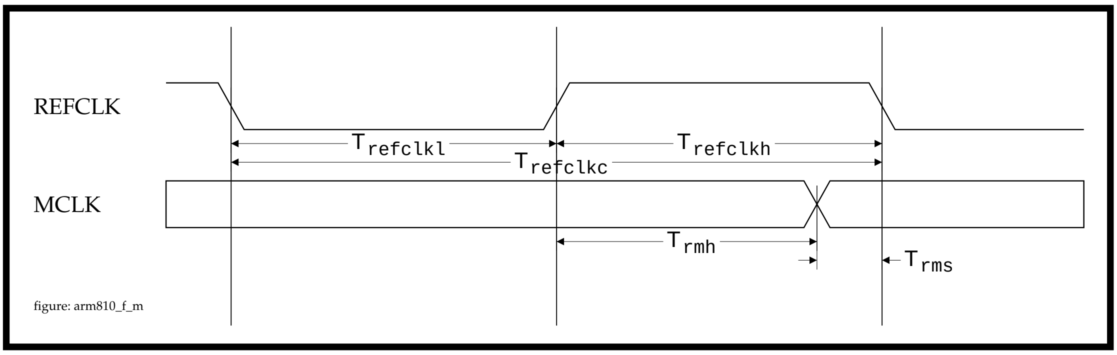


Figure 15-1: Synchronous mode using MCLK

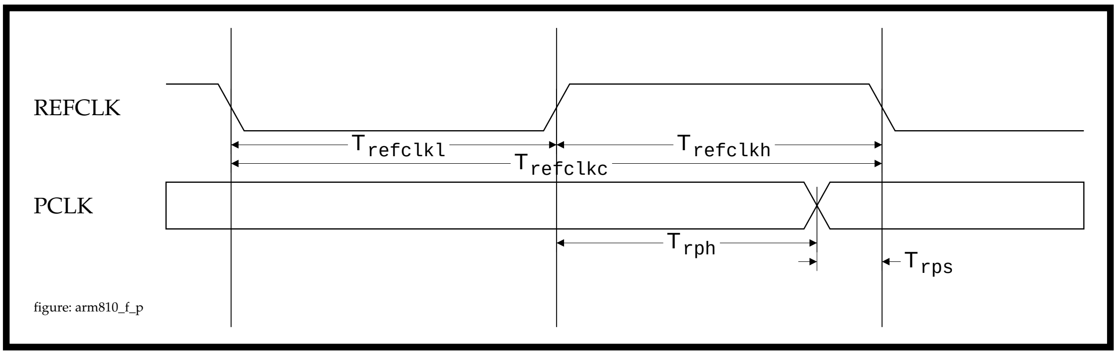


Figure 15-2: Synchronous mode using PCLK

Symbol	Parameter	min	max	unit	note
Trmh	REFCLK - MCLK hold time	4		ns	1,2
Trms	MCLK - REFCLK setup time	4		ns	1,2
Trph	REFCLK - PCLK hold time	4		ns	1,2
Trps	PCLK - REFCLK setup time	4		ns	1,2

Table 15-6: Synchronous mode clock relationship

15.3 Main Bus Signals

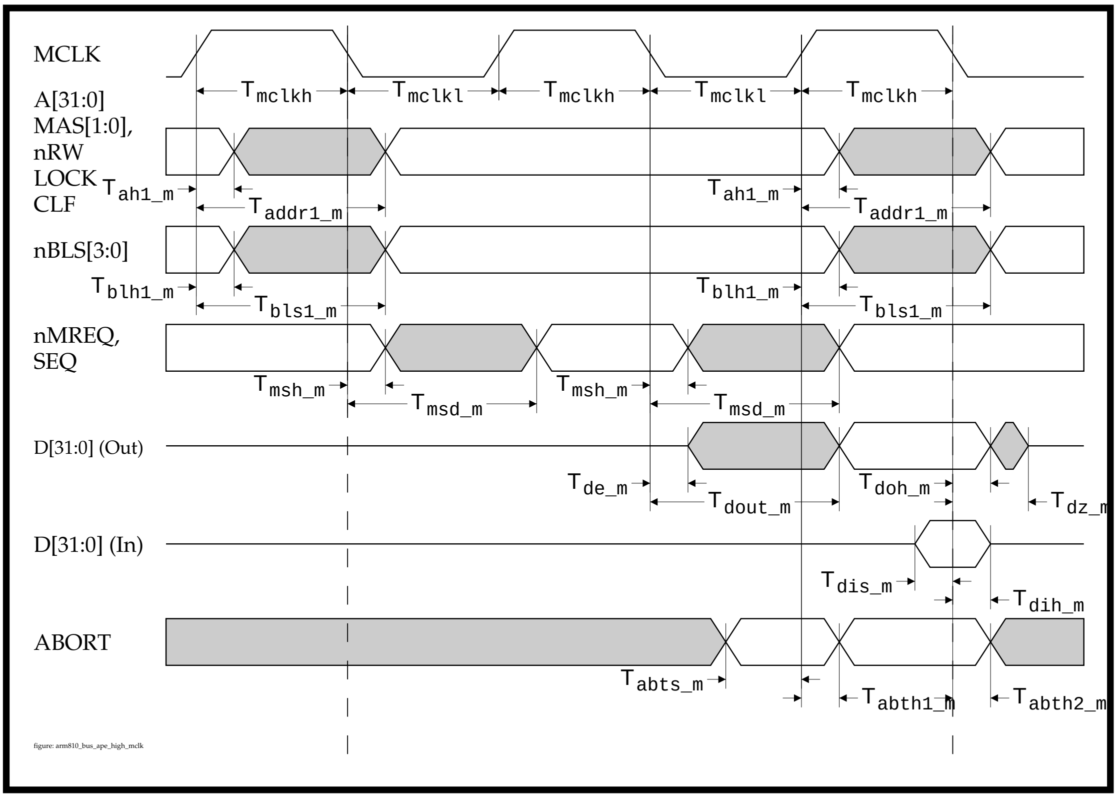


Figure 15-3: ARM810 bus timing using MCLK with APE HIGH

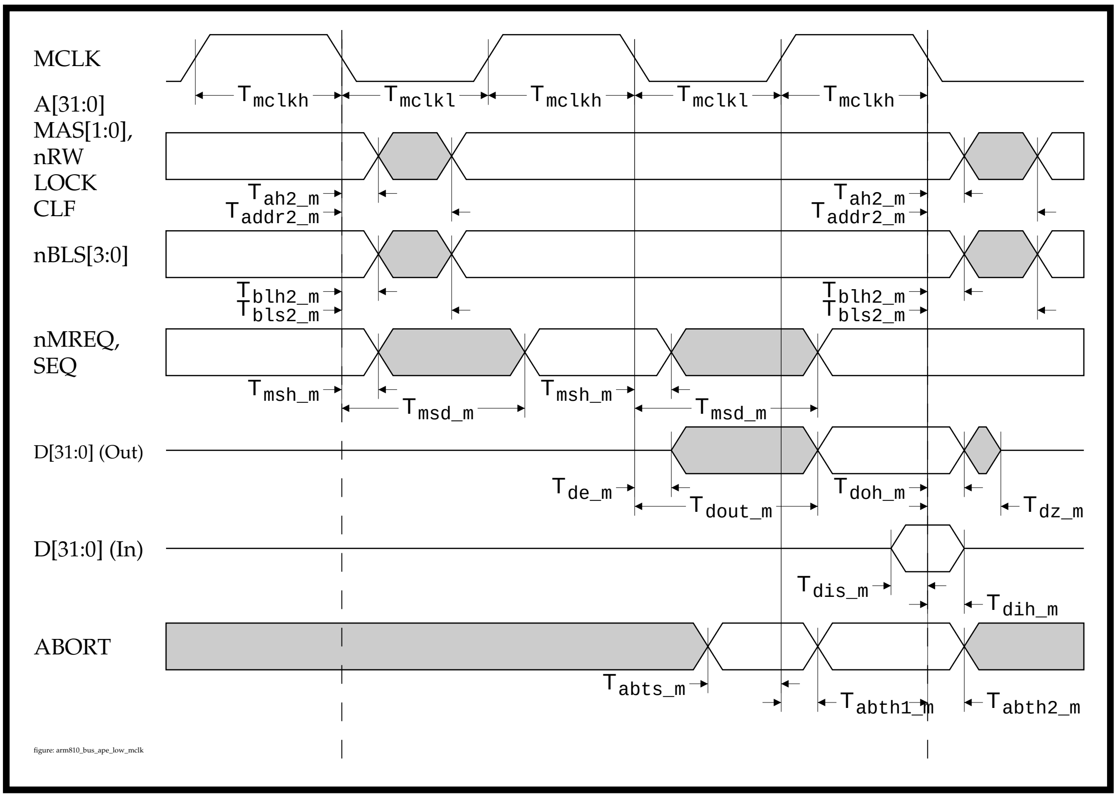


Figure 15-4: ARM810 bus timing using MCLK with APE LOW

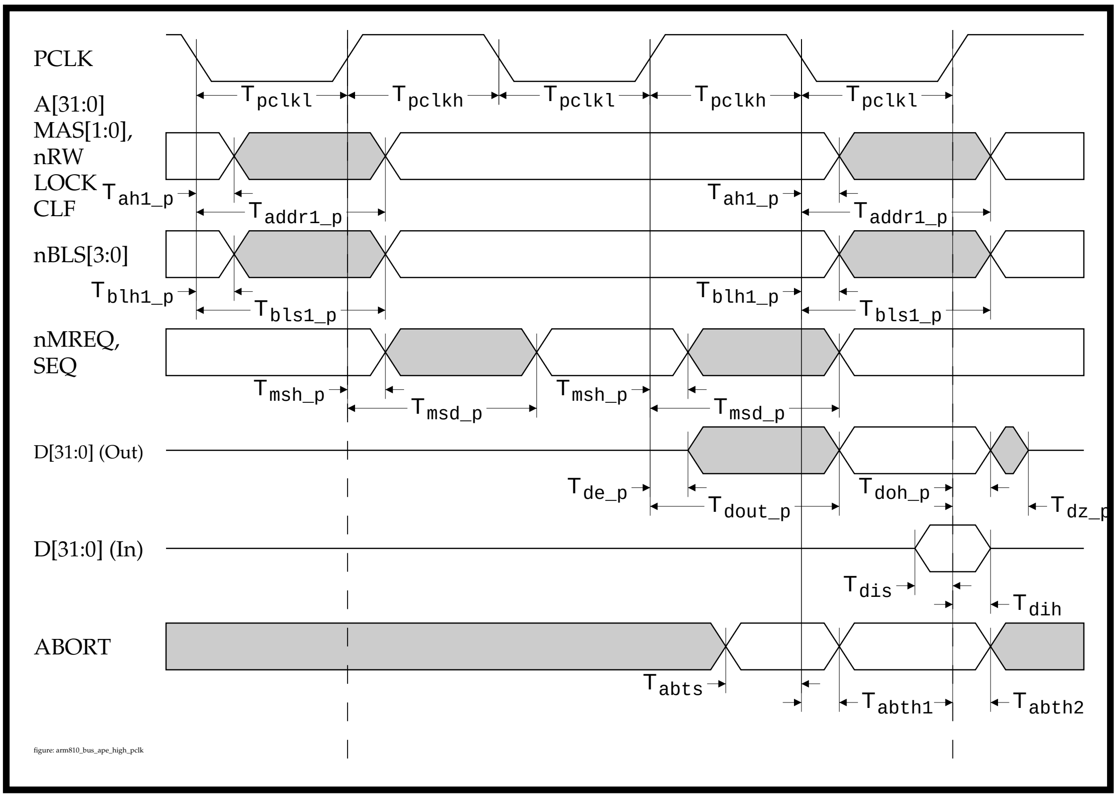


Figure 15-5: ARM810 bus timing using PCLK with APE HIGH

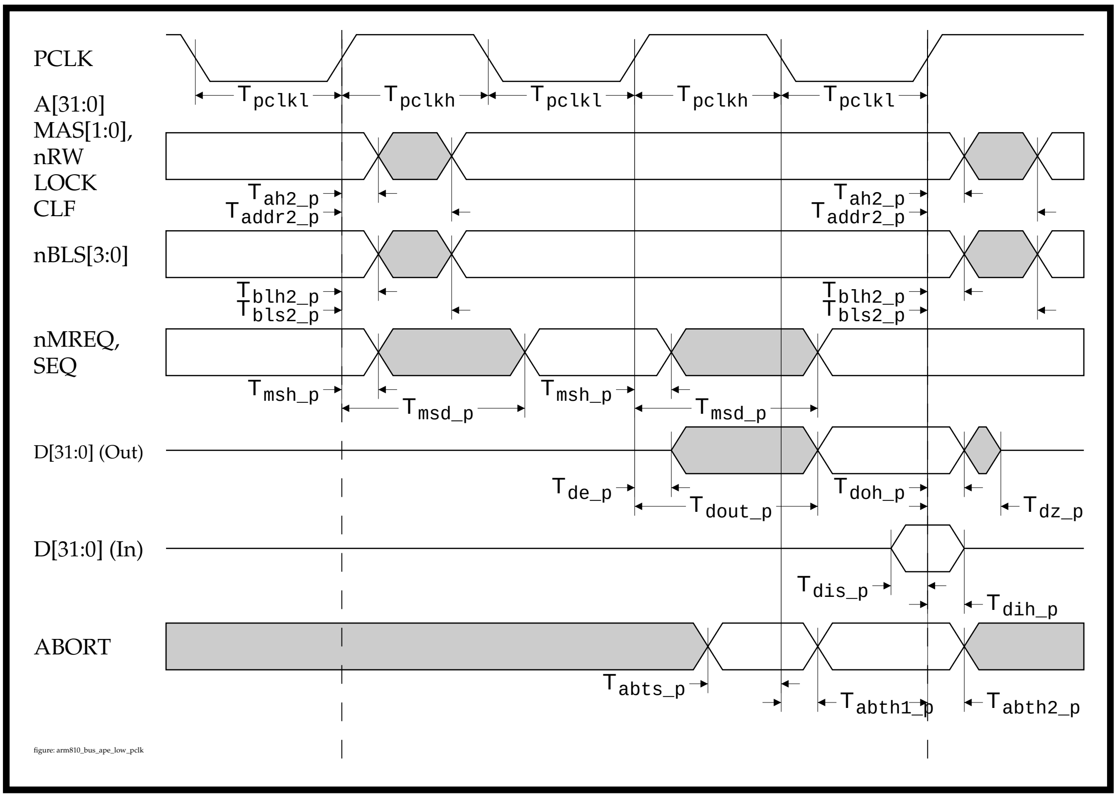


Figure 15-6: ARM810 bus timing using PCLK with APE LOW

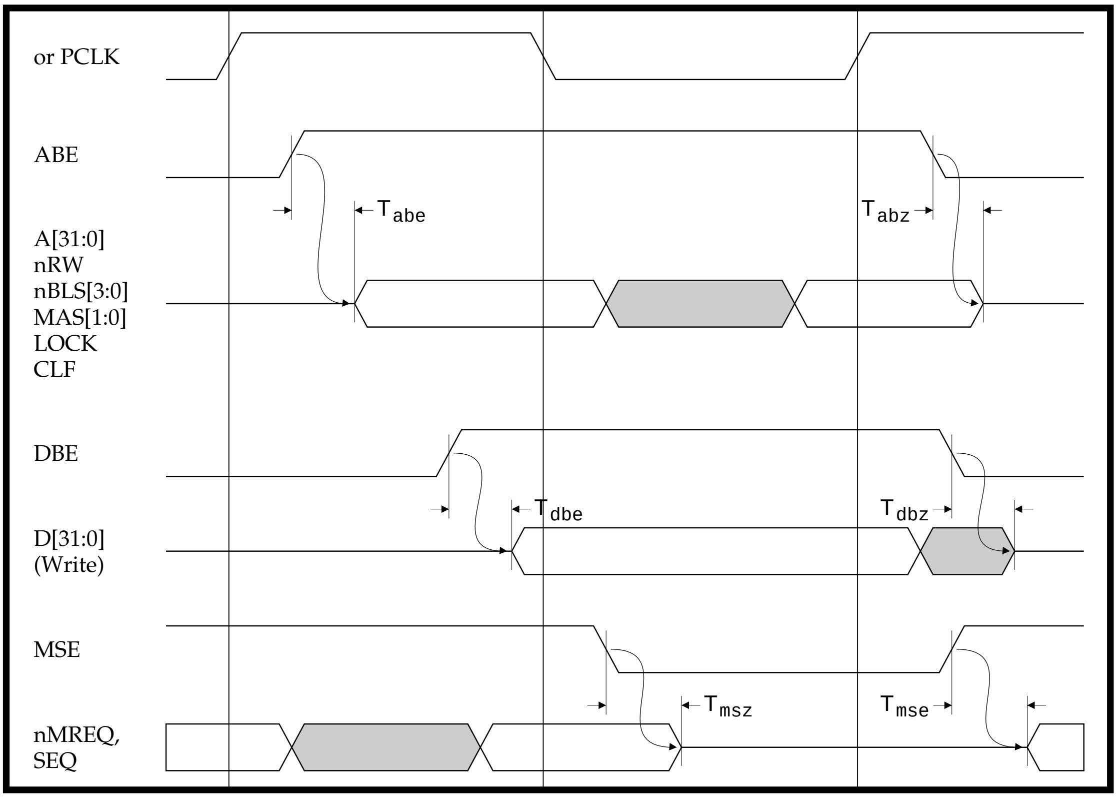


Figure 15-7: ARM810 bus enable timing

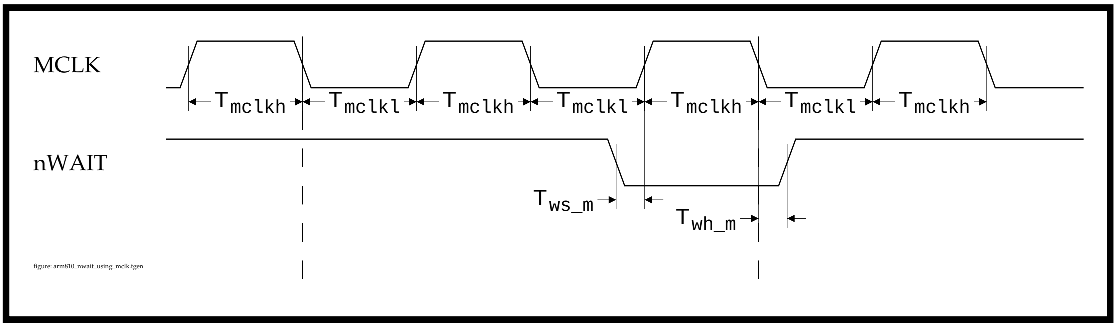


Figure 15-8: ARM810 nWAIT timing using MCLK

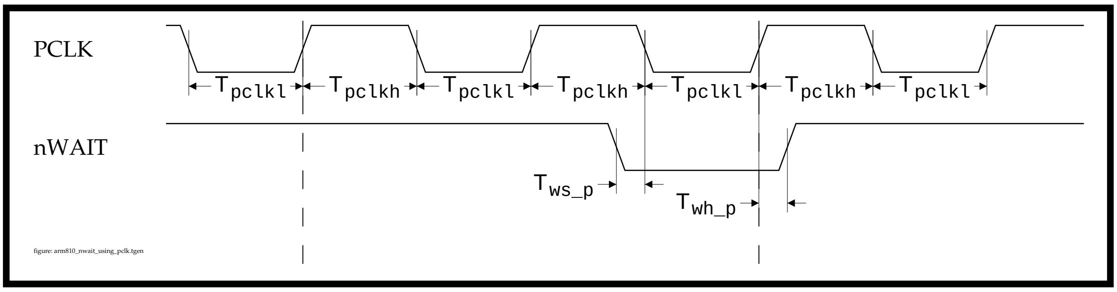


Figure 15-9: ARM810 nWAIT timing using PCLK

Symbol	Parameter	Min	Max	Unit	Note
Tws_m	nWAIT setup to MCLK	0		ns	
Twh_m	WAIT hold from MCLK	2		ns	
Tabe_m	ABE to address bus enable		12	ns	5
Tabz_m	ABE address bus disable		24	ns	
Taddr1_m	MCLK to addr. delay ALE High		12	ns	5
Taddr2_m	MCLK to addr. delay ALE Low		12	ns	5
Tah1_m	address hold time ALE High	5		ns	5
Tah2_m	address hold time ALE Low	5		ns	5
Tbls1_m	MCLK to nBLS[3:0] delay ALE High		12	ns	5
Tbls2_m	MCLK to nBLS[3:0] delay ALE Low		12	ns	5
Tblh1_m	MCLK to nBLS[3:0] hold, ALE High	5		ns	5
Tblh2_m	MCLK to nBLS[3:0] hold, ALE Low	5		ns	5
Tdbe_m	DBE to data enable		12	ns	5
Tde_m	MCLK to data enable	6		ns	5
Tdbz_m	DBE to data disable		20	ns	
Tdz_m	MCLK to data disable	6	20	ns	
Tdout_m	MCLK to data out delay		24	ns	5
Tdoh_m	MCLK to data out hold	5			5
Tdis_m	data in to MCLK setup	2			
Tdih_m	MCLK to data in hold	5			
Tabts_m	abort to MCLK setup	8			
Tabth1_m	MCLK to abort hold	2			6
Tabth2_m	MCLK to about hold	2			6
Tmse_m	nMREQ and SEQ enable		10		
Tmsz_m	nMREQ and SEQ disable		20		
Tmsd_m	nMREQ and SEQ delay		12		
Tmsh_m	nMREQ and SEQ hold	5			

Table 15-7: ARM810 bus timing using MCLK

Symbol	Parameter	Min	Max	Unit	Note
Tws_p	nWAIT setup to PCLK	0		ns	
Twh_p	WAIT hold from PCLK	2		ns	
Tabe_p	ABE to address bus enable		12	ns	5
Tabz_p	ABE address bus disable		24	ns	
Taddr1_p	PCLK to addr. delay,ALE High		12	ns	5
Taddr2_p	PCLK to addr. delay,ALE Low		12	ns	5
Tah1_p	address hold time,ALE High	5		ns	5
Tah2_p	address hold time,ALE Low	5		ns	5
Tbls1_p	PCLK to nBLS[3:0] delay,ALE High		12	ns	5
Tbls2_p	PCLK to nBLS[3:0] delay,ALE Low		12	ns	5
Tblh1_p	PCLK to nBLS[3:0] hold, ALE High	5		ns	5
Tblh2_p	PCLK to nBLS[3:0] hold, ALE Low	5		ns	5
Tdbe_p	DBE to data enable		12	ns	5
Tde_p	PCLK to data enable	6		ns	5
Tdbz_p	DBE to data disable		20	ns	
Tdz_p	PCLK to data disable	6	20	ns	
Tdout_p	PCLK to data out delay		24	ns	5
Tdoh_p	PCLK to data out hold	5			5
Tdis_p	data in to PCLK setup	2			
Tdih_p	PCLK to data in hold	5			
Tabts_p	abort to PCLK setup	8			
Tabth1_p	PCLK to abort hold	2			6
Tabth2_p	PCLK to about hold	5			6
Tmse_p	nPREQ and SEQ enable		10		
Tmsz_p	nMREQ and SEQ disable		20		
Tmsd_p	nMREQ and SEQ delay		12		
Tmsh_p	nMREQ and SEQ hold	5			

Table 15-8: ARM810 bus timing using PCLK

Notes:

1 Only one of MCLK or PCLK is used. The other is tied as described in ***Chapter 11, ARM810 Clocking***.

2 MCLK, PCLK, and REFCLK timings are measured at 50% of Vdd.

3 The PLL must be configured so that all of these parameters are within allowed limits. See ***11.3.2 Fast clock from the output of the PLL*** on page 11-7.

4 In all clocking modes the Fast clock frequency must be greater than or equal to the Bus clock frequency.

5 The timings of these buses are measured at 50% of Vdd.

6 Tabth1 is required by this device. To ensure compatibility with other ARM processors, you are advised to make your designs meet Tabth2. Tabth2 is not tested on this device, and is given as a recommendation only.

16 Physical Details

This chapter describes the physical details of the ARM810. The information in this chapter is provided as a guide only. Refer to your semiconductor vendor for definitive physical details.

16.1 Physical Details 16-2
16.2 Pinout 16-3

16.1 Physical Details

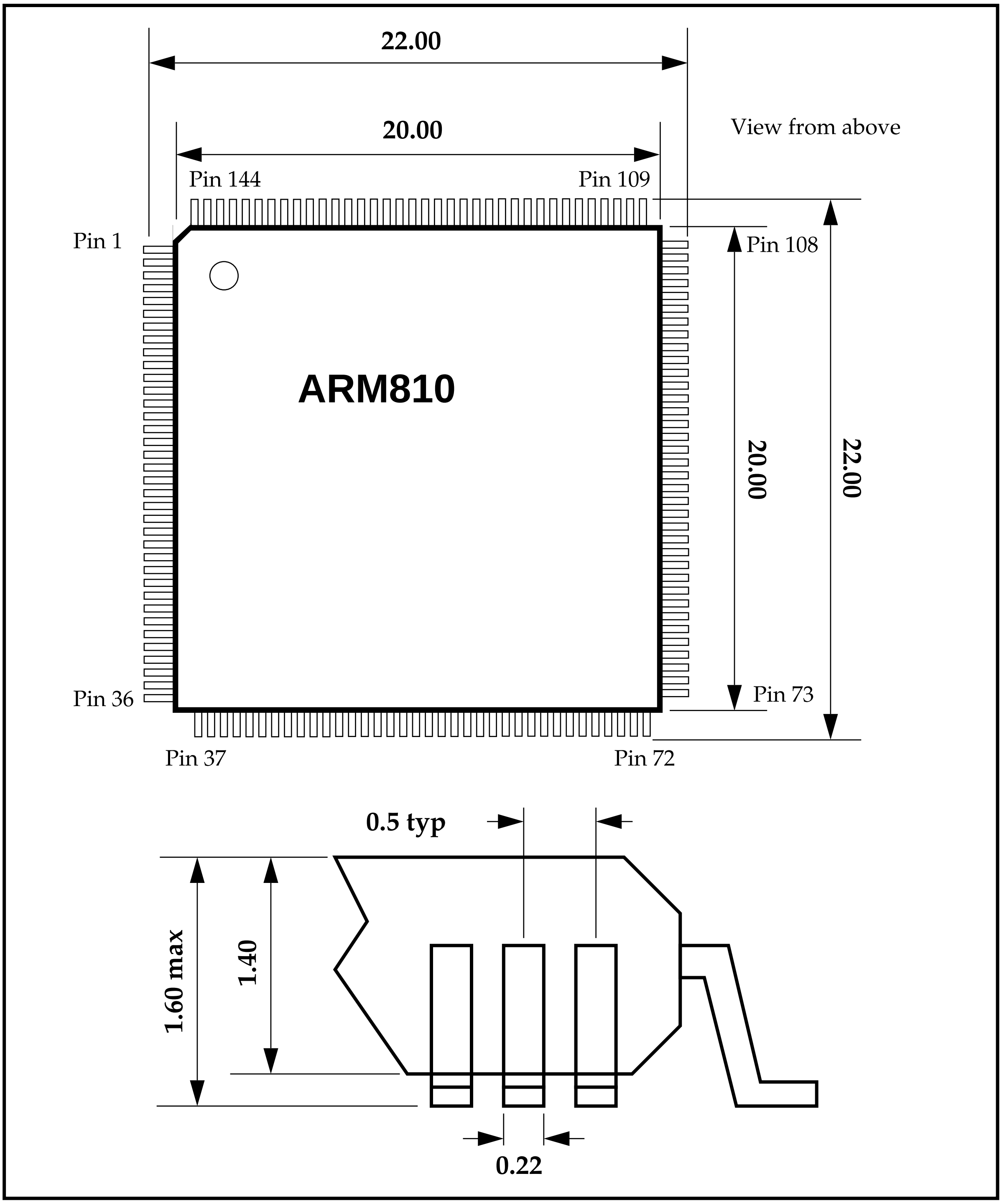


Figure 16-1: Typical ARM810 144 Pin TQFP mechanical dimensions in mm

16.2 Pinout

Pin	Signal
1	MSE
2	SEQ
3	NMREQ
4	REFCLKCFG[0]
5	REFCLKCFG[1]
6	Vdd_core
7	PLLSLEEP
8	Vss_core
9	PLLRANGE
10	PLLVDD
11	PLLFILT2
12	PLLFILT1
13	PLLGND
14	NWAIT
15	REFCLK
16	Vdd_pad
17	PCLK
18	MCLK
19	Vss_pad
20	DBE
21	D[0]
22	D[1]
23	D[2]
24	D[3]
25	D[4]
26	D[5]
27	D[6]
28	Vdd_pad
29	D[7]

Pin	Signal
30	Vss_pad
31	D[8]
32	D[9]
33	D[10]
34	D[11]
35	D[12]
36	D[13]
37	D[14]
38	D[15]
39	D[16]
40	Vdd_pad
41	D[17]
42	Vss_pad
43	D[18]
44	D[19]
45	Vdd_core
46	D[20]
47	Vss_core
48	D[21]
49	D[22]
50	D[23]
51	D[24]
52	Vdd_pad
53	D[25]
54	Vss_pad
55	D[26]
56	D[27]
57	D[28]
58	D[29]

Pin	Signal
59	D[30]
60	D[31]
61	TDO
62	TCK
63	TMS
64	nTRST
65	TDI
66	Vdd_pad
67	NBLS[0]
68	Vss_pad
69	NBLS[1]
70	NBLS[2]
71	NBLS[3]
72	NRW
73	MAS[0]
74	MAS[1]
75	CLF
76	LOCK
77	A[0]
78	A[1]
79	Vdd_pad
80	A[2]
81	Vss_pad
82	A[3]
83	A[4]
84	Vdd_core
85	A[5]
86	Vss_core
87	A[6]

Pin	Signal
88	A[7]
89	A[8]
90	Vdd_pad
91	A[9]
92	Vss_pad
93	A[10]
94	A[11]
95	A[12]
96	Vdd_core
97	A[13]
98	Vss_core
99	A[14]
100	A[15]
101	A[16]
102	Vdd_pad
103	A[17]
104	Vss_pad
105	A[18]
106	A[19]
107	A[20]
108	A[21]
109	A[22]
110	A[23]
111	A[24]
112	A[25]
113	A[26]
114	Vdd_pad
115	A[27]
116	Vss_pad
117	A[28]
118	A[29]
119	A[30]
120	A[31]
121	ABE
122	APE
123	Vcc
124	TESTOUT[0]
125	TESTOUT[1]
126	Vdd_core
127	TESTOUT[2]
128	Vss_core
129	TESTOUT[3]
130	TESTOUT[4]
131	TESTMODE
132	NIRQ
133	Vdd_pad
134	NRESET
135	Vss_pad
136	NFIQ
137	ABORT
138	PLLCFG[0]
139	PLLCFG[1]
140	PLLCFG[2]
141	PLLCFG[3]
142	PLLCFG[4]
143	PLLCFG[5]
144	PLLCFG[6]

17 Backward Compatibility

This chapter summarises the changes in ARM810 when compared to previous ARM processors.

ARM810 will be able to run binary code targetted to earlier processors with only a few exceptions. The following changes from previous implementations of the ARM should be noted. See ***Appendix B, Instruction Set Changes*** for a more detailed discussion.

17.1 Instruction Memory Barrier 17-2
17.2 Undefined Instructions 17-2
17.3 PC Offset 17-2
17.4 Write-Back 17-2
17.5 Misaligned PC Loads and Stores 17-2
17.6 Data Aborts 17-2

Backward Compatibility

17.1 Instruction Memory Barrier

The requirement for an Instruction Memory Barrier (IMB) instruction means that if code changes the instruction stream and then tries to execute it without an intervening IMB, the consequences will be unpredictable. For example, there must be an IMB instruction between loading code into memory and executing it. See ***4.17 The Instruction Memory Barrier (IMB) Instruction*** on page 4-64 for more information.

17.2 Undefined Instructions

In ARM810, most unallocated instruction bit patterns in the instruction set space enter the Undefined Instruction trap. See ***4.18 Undefined Instructions*** on page 4-67 for further information.

17.3 PC Offset

Rare ARM7 instructions which store a R15 value to memory as the address of the instruction plus an offset of 12 will now either use an offset of 8 instead or will no longer be valid on ARM8. The following summarises their behaviour on ARM810:

- STR instructions with Rd = R15 store the address of the instruction plus 8
- STM instructions with R15 in the list of registers to be stored store the address of the instruction plus 8
- Data processing instructions with a register-specified shift and at least one of Rm and Rn equal to R15 are no longer valid
- MCR instructions with R15 as the source register are no longer valid

17.4 Write-Back

Loading a register with write-back to it will have UNPREDICTABLE effects.

The rules governing whether stores with write-back to the stored register store the register's old or new value differ from those of ARM7. See ***4.11.6 Inclusion of the base in the register list*** on page 4-42 for further details.

17.5 Misaligned PC Loads and Stores

Misaligned loads or stores of the PC have UNPREDICTABLE effects.

17.6 Data Aborts

In all cases where a data abort occurs, any base register is restored to its original value (before the instruction started), regardless of whether writeback is specified or not.

A Use of the ARM810 in a 5V TTL System

This appendix describes how to use the ARM810 in a 5V TTL system.

A.1 Using the ARM810 in a 5V TTL System A-2

Use of the ARM810 in a 5V TTL System

A.1 Using the ARM810 in a 5V TTL System

The ARM810 can be used in a 5V TTL level system. For this application a separate 3.3V supply, connected to VDD, is required. VREF should be connected to the 5V system power supply as shown in ***Figure A-1: System power connection***.

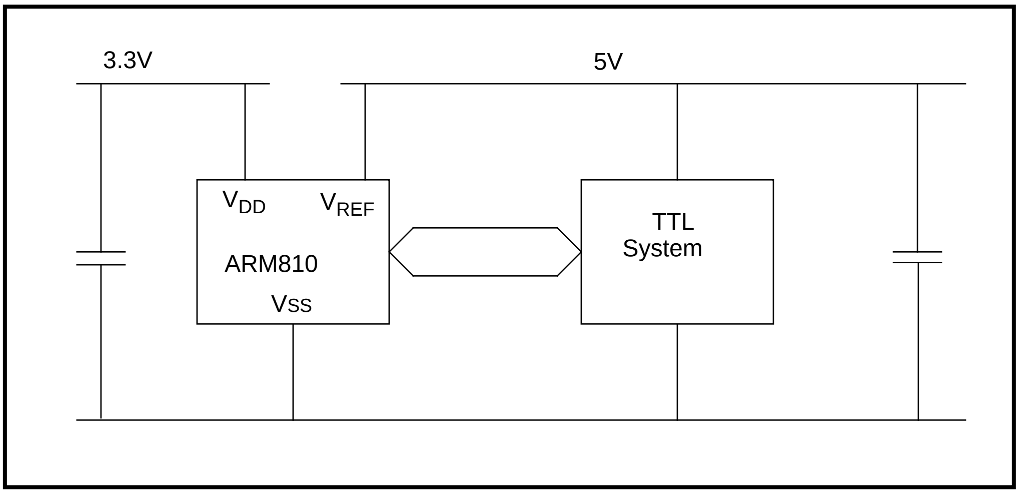


Figure A-1: System power connection

In this system, the ARM810 input and output levels are directly TTL compatible. See ***Figure A-2: ARM810 inputs driven by TTL outputs*** and ***Figure A-3: ARM810 outputs driving TTL inputs*** on page A-3.

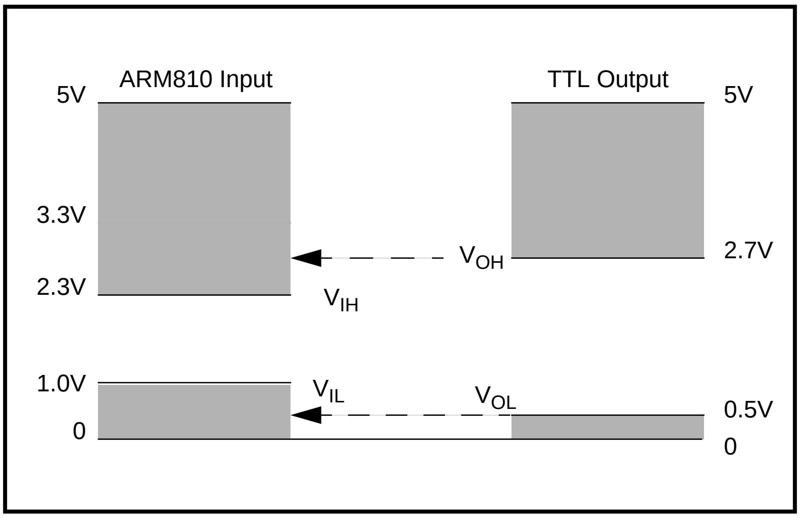


Figure A-2: ARM810 inputs driven by TTL outputs

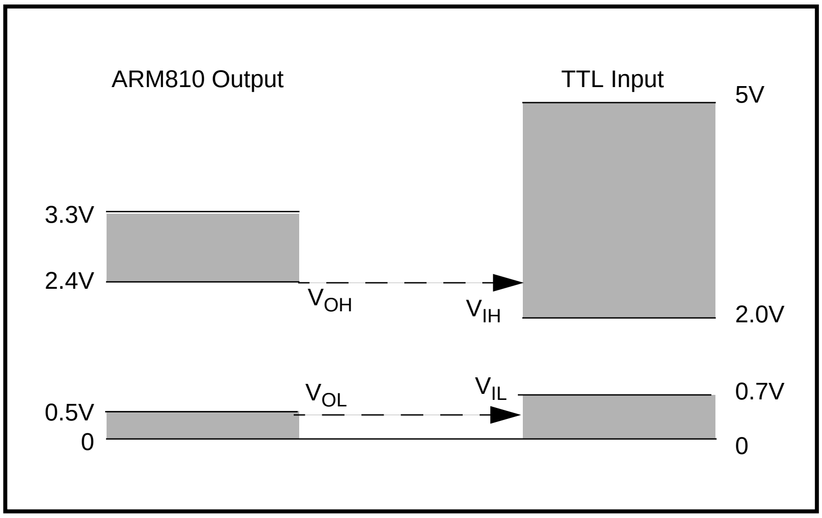


Figure A-3: ARM810 outputs driving TTL inputs

B Instruction Set Changes

This Appendix gives an overview of changes to the instruction set when compared to ARM7.

B.1 General Compatibility B-2

B.2 Instruction Set Differences B-2

Instruction Set Changes

B.1 General Compatibility

Existing code will run subject to all of the restrictions described in all ARM Datasheets up to ARM810, in addition to those in this data sheet.

As previous code does not have the IMB instructions, some code may not be compatible in certain circumstances.

For example:

- Code that constructs a routine in memory and then branches to it will be incompatible unless branch prediction has been turned off, or a calculated branch was used to get to it.
- Code that constructs a routine in memory, and then falls through to it sequentially will be incompatible unless the fall-through code instruction has been constructed at least 12 instructions in advance of its execution.

B.2 Instruction Set Differences

This section describes the instruction set additions and changes that have been made for ARM810.

B.2.1 New features

An Instruction Memory Barrier (IMB) instruction

This tells the ARM to flush any stored information about the instruction stream, and must be issued between modifying an instruction area and executing it.

Please refer to ***4.17 The Instruction Memory Barrier (IMB) Instruction*** on page 4-64 and ***Appendix E, Implementing the Instruction Memory Barrier Instruction*** for details.

Half-word and signed byte support

This has been added to the instruction set. Please refer to ***4.10 Halfword and Signed Data Transfer*** on page 4-34 for details.

B.2.2 Existing instructions

STM instructions with base writeback and the base register in the register list

This concerns the order of writeback and reading the value to be stored:

- If the base register is the lowest numbered register in the list, then the original base value is stored.
- Otherwise, the stored value is undefined at present.

LDM instructions with base writeback and the base register in the register list

This has no function, since the written-back register value is overwritten by the loaded value.

The behaviour is now architecturally undefined.

LDR instructions with writeback which load the base register

The behaviour is already architecturally undefined; see Application Note A002.

LDRB PC

The behaviour is already architecturally undefined; see Application Note A002.

LDR PC from a misaligned address

The behaviour is now architecturally undefined.

STRB PC

The behaviour is architecturally undefined; see Application Note A002.

STR PC to a misaligned address

The behaviour is now architecturally undefined.

STR PC

These were expected to store the address of the instruction plus 12, not the normal address of instruction plus 8.

These now store the address of instruction plus 8.

STM ...,PC}

These were expected to store the address of the instruction plus 12, not the normal address of instruction plus 8.

These now store the address of instruction plus 8.

Data processing instructions that do a register-controlled shift and have either or both of the main operand registers equal to the PC.

The behaviour is now architecturally undefined.

MCR instructions (coprocessor register transfers from ARM to coprocessor) with the PC as the source register.

The behaviour is now architecturally undefined.

C

26-bit Operations on ARM810

This appendix describes the 26-bit operations on ARM810.

C.1	Introduction	C-2
C.2	Instruction Differences	C-3
C.3	Performance Differences	C-4
C.4	Hardware Compatibility Issues	C-4

26-bit Operations on ARM810

C.1 Introduction

To maintain compatibility with earlier ARM processors, it is possible to execute code in 26-bit operating modes **usr26**, **fiq26**, **irq26** and **svc26**. Details of how to do this have already been written for earlier ARM processors, and these have been included here for your information.

This appendix summarises how 26-bit binary code will be able to run on the ARM810 processor. The details below show the instruction and performance differences when ARM810 is operated in 26-bit modes. The last section describes the hardware changes that affect 26-bit operation.

Use of 26-bit modes for any reason other than executing existing 26-bit code is strongly discouraged, as this will no longer be supported in ARM processors after the ARM810. It is also worth noting that ARM810's performance in 26-bit modes may be poorer than in 32-bit modes.

C.2 Instruction Differences

When ARM810 is executing in a 26-bit mode, the top 6 bits of the PC are forced to be zero at all times. The following restrictions must be obeyed to avoid problems due to the Prefetch Unit having prefetched an unknown distance beyond the current instruction:

- Do not enter any 26-bit mode when at an address outside the 26-bit address space.
- Do not execute code sequentially from address 0x03FFFFFC to address 0x00000000 in 26-bit code.

An additional requirement for 32-bit and 26-bit operations is that if a system contains code that is intended for execution in both 26-bit and 32-bit modes, an IMB instruction must accompany any change from any 26-bit mode to any 32-bit mode, and vice versa. It is therefore advisable to keep code intended for 26-bit modes and code intended for 32-bit modes completely separate.

26-bit operation removes some of the instruction constraints placed on 32-bit code. 26-bit code must obey the constraints laid out for 32-bit code with the following exceptions:

1 CMN, CMP, TEQ, TST
A second form of these instructions becomes available, which is encoded in the instruction by setting the Rd field to “1111” and in the assembler syntax by using:

```
<opcode>{cond}P
```

in place of the normal

```
<opcode>{cond}
```

In all modes, the normal setting of the CPSR flags is suppressed for the new form of the instruction. Instead, the normal arithmetic result is calculated (Op1+Op2, Op1-Op2, Op1 EOR Op2 and Op1 AND Op2 for CMN, CMP, TEQ and TST respectively) and used to set selected CPSR bits. In user mode, the N, Z, C and V bits of the CPSR are set to bits 31 to 28 of the arithmetic result; in non-user modes, the N, Z, C, V, I, F, M1 and M0 bits of the CPSR are set to bits 31 to 26, 1 and 0 of the arithmetic result.

The CMNP, CMPP, TEQP and TSTP instructions take a base of 3 cycles to execute, along with the extra cycles listed in ***4.5.8 Instruction cycle times*** on page 4-14 for complex and register-specified shifts..

2 Data processing instructions with destination register R15 and the S bit set
These become valid in User mode, and their behaviour in all modes is altered.

In all modes, the normal setting of the CPSR flags from the current mode’s SPSR is suppressed. In user mode, the N, Z, C and V bits of the CPSR are set to bits 31 to 28 of the arithmetic result; in non-user modes, the N, Z, C, V, I, F, M1 and M0 bits of the CPSR are set to bits 31 to 26, 1 and 0 of the arithmetic result.

3 LDM with R15 in Register list, and the S bit set
This becomes valid in User mode, and its behaviour in all modes is altered.

In all modes, the normal setting of the CPSR flags from the current mode's SPSR is suppressed. In user mode, the N, Z, C and V bits of the CPSR are set to bits 31 to 28 of the value loaded for R15; in non-user modes, the N, Z, C, V, I, F, M1 and M0 bits of the CPSR are set to bits 31 to 26, 1 and 0 of the value loaded for R15.

4 Address Exceptions

The address exceptions which occur on true 26-bit ARM processors cannot occur on ARM810. If required, these should now be generated externally to the ARM810 as aborts, along with an abort handler routine which recognises the address exception.

Note: Some unusual coding cases may present problems: for example, LDMs and STMs wrapping around from the top of 26-bit memory space to the bottom. It is thought that such cases are not in common use, and so should not present any difficulties.

C.3 Performance Differences

This information is provisional at this release of the data sheet. Implementation details may affect performance.

There is no cycle count performance degradation for operating in 26-bit mode; the cycle counts are the same as those for 32-bit mode operations. However, there may be degradation due to the additional software overheads in getting to and from 32-bit-mode-only operations.

C.4 Hardware Compatibility Issues

This section describes the ways in which the ARM810 will differ from previous ARM processors, as far as its hardware is concerned, for 26-bit compatibility.

This section is up-to-date, but is not necessarily complete.

C.4.1 Configuration

ARM810 will not have the two configuration bits, **DATA32** and **PROG32** that could be found on previous ARM610 and ARM710 processors.

As such, the processor's normal mode of operation is in full 32-bit modes: as if both of these bits were configured in their active HIGH state. Aborts on Read and Write of the Exception Vectors can be done by the Memory Manager, thus stimulating the original hardware configurations in software.

D

Comparisons with 26-bit ARM Processors

This appendix describes the differences between 32-bit ARM processors and earlier 26-bit ARM processors:

- 32-bit ARM processors are the ARM6 family, and all later processors including the ARM7 family, the ARM8 family and StrongARM.
- 26-bit ARM processors are ARM2, ARM3 and ARM2aS.

This information is included here for completeness as it provides further details about the differences between 26-bit and 32-bit codes to that described in ***Appendix C, 26-bit Operations on ARM810***. The information in the rest of the datasheet supersedes this appendix.

D.1 Introduction D-2

D.2 The Program Counter and Program Status Register D-2

D.3 Operating Modes D-2

D.4 Instruction Set Changes D-3

D.5 Transferring between 26-bit and 32-bit Modes D-4

Comparisons with 26-bit ARM Processors

D.1 Introduction

The ARM6 family, and all later processors including the ARM7 family, the ARM8 family and StrongARM, are ARM processors that have 32-bit program counters. Earlier ARMs (ARM2, ARM3 and ARM2aS) had a 26-bit program counter (PC). This appendix describes the major differences between the two types of processor.

D.2 The Program Counter and Program Status Register

The introduction of the larger program counter has meant that the flags and control bits of R15 (the combined PC and PSR) have been moved to a separate register. The extra space in the new register (the CPSR, Current Program Status Register) allows for more control bits. A further 3 mode bits have been added to allow for a larger number of operating modes.

The removal of the PSR to a separate register also means that it is no longer possible to save these flags automatically in R14 when a Branch with Link (BL) instruction is executed, or when an exception occurs. Program analysis has shown that the saving of these flags is only required in 3% of subroutine calls, so there is only a slight overhead in explicitly saving them on a stack when necessary. To cope with the requirement of saving them when an exception occurs, 5 further registers have been provided to hold a copy of the CPSR at the time of the exception. These registers are the Saved Program Status Registers (SPSRs). There is one SPSR for each of the modes that the processor may enter as a result of the various types of exception.

The expansion of the PC to 32 bits also means that the Branch instruction, being limited to +/-32 Mbytes, can no longer specify a branch to the entire program space. Branches greater than +/-32 Mbytes can be made with other instructions, but the equivalent of the Branch with Link instruction will require a separate instruction to save the PC in R14.

D.3 Operating Modes

There are a total of 10 operating modes in two overlapping sets. Four modes—**User26**, **IRQ26**, **FIQ26** and **Supervisor26**—allow the processor to behave like earlier ARM processors with a 26-bit PC. These correspond to the four operating modes of the ARM2 and ARM3 processors. A further four operating modes correspond to these, but with the processor running with the full 32-bit PC: these are **User32**, **IRQ32**, **FIQ32** and **Supervisor32**.

The final two modes are **Undefined32** and **Abort32**, and are entered when the Undefined instruction and Abort exceptions occur. They have been added to remove restrictions on Supervisor mode programs which exist with the ARM2 and ARM3 processors. The two sets of User, FIQ, IRQ and Supervisor modes each share a set of banked registers to allow them to maintain some private state at all times. The Abort and Undefined modes also have a pair of banked registers each for the same purpose.

D.4 Instruction Set Changes

The instruction set is changed in two major areas: new instructions have been introduced and restrictions have been placed on existing ones.

D.4.1 New Instructions

The new instructions allow access to the CPSR and SPSR registers. They are formed by using opcodes from the Data Processing group of instructions that were previously unused. Specifically, these are the TST, TEQ, CMP and CMN instructions with the S flag clear. They are now known as MSR to move data into the CPSR and SPSR registers, and MRS to move from the CPSR and SPSR to a general register. The data moved to CPSR and SPSR can be either the contents of a general register or an immediate value.

D.4.2 Instruction Set Limitations

When configured for 32-bit program and data space, 32-bit processors support operation in 26-bit modes for compatibility with ARM processors that have a 26-bit address space. The 26-bit modes are **User26**, **FIQ26**, **IRQ26** and **Supervisor26**. When a 26-bit mode is selected, the programmer’s model reverts to that of existing 26 bit ARMs (ARM2, ARM3, ARM2aS). The behaviour is that of the ARM2aS macrocell with the following alterations:

- Address exceptions are *never* generated. The OS may simulate the behaviour of address exception by using external logic such as a memory management unit to generate an abort if the 64 Mbyte range is exceeded, and converting that abort into an “address exception” trap for the application.

Note Address exceptions are still possible when the processor is configured for 26-bit program and data space.

- The new instructions to transfer data between general registers and the program status registers remain operative. The new instructions can be used by the operating system to return a 32-bit operating mode after calling a binary containing code written for a 26-bit ARM.
- All exceptions (including Undefined Instruction and Software Interrupt) return the processor to a 32-bit mode, so the operating system must be modified to handle them.
- 32-bit processors include hardware which prevents the write operation and generates a data abort if the processor attempts to write to a location between &00000000 and &0000001F inclusive (the exception vectors) when operating in 26-bit mode. This allows the operating system to intercept all changes to the exception vectors and redirect the vector to some veneer code. The veneer code should place the processor in a 26-bit mode before calling the 26-bit exception handler.

In all other respects, 32-bit processors behave like a 26-bit ARM when operating in 26-bit mode. The relevant bits of the CPSR appear to be incorporated back into R15 to form the PC/CPSR with the I and F bits in bits 27 and 26. The instruction set behaves like that of the ARM2aS macrocell with the addition of the MRS and MSR instructions.

D.5 Transferring between 26-bit and 32-bit Modes

A program executing in a privileged 32-bit mode can enter a 26-bit mode by executing an MSR instruction which alters the mode bits to one of the values shown below:

M[4:0]	Mode	Accessible register set
00000	usr26	PC/PSR, R14..R0, CPSR
00001	fiq26	PC/PSR, R14_fiq..R8_fiq, R7..R0, CPSR, SPSR_fiq
00010	irq26	PC/PSR, R14_irq..R13_fiq, R12..R0, CPSR, SPSR_irq
00011	svc26	PC/PSR, R14_svc..R13_svc, R12..R0, CPSR, SPSR_svc

Table D-1: MSR instruction altering the mode bits

Transfer between 26-bit and 32-bit mode happens automatically whenever an exception occurs in 26-bit mode. Note that an exception (including Software Interrupt) arising in 26-bit mode will enter 32-bit mode and the saved value in R14 will contain only the PC, even though the PSR was also considered part of R15 when the exception arose.

In addition, the MSR instruction provides the means for a program in a privileged 26-bit mode to alter the mode bits to change to a 32-bit mode.

E

Implementing the Instruction Memory Barrier Instruction

This appendix is written to help Operating System designers understand and implement the IMB Instructions. It firstly describes the generic approach that should be used for future compatibilty and then goes on to ARM810-specific details.

E.1 Introduction E-2
E.3 Generic IMB Use E-2
E.2 ARM810 IMB Implementation E-2

Implementing the Instruction Memory Barrier Instruction

E.1 Introduction

This appendix describes the processor specific code that must be included in the SWI handler to implement the two Instruction Memory Barrier (IMB) Instructions:

- IMB
- IMBRange

These are implemented as calls to specific SWI numbers. Please refer to ***4.17 The Instruction Memory Barrier (IMB) Instruction*** on page 4-64 for further details of this and for examples of use.

Two IMB instructions are provided so that when only a small area of code is altered before being executed the IMBRange instruction can be used to efficiently and quickly flush any stored instruction information from addresses within a small range rather than flushing all information about all instructions using the IMB instruction.
By flushing only the required address range information, the rest of the information remains to provide improved system performance.

E.2 ARM810 IMB Implementation

For ARM810, executing the SWI instruction is sufficient in itself to cause the IMB operation. Also, for ARM810, both the IMB and the IMBRange instructions flush *all* stored information about the instruction stream.

This means that for ARM810, all IMB instructions can be implemented in the Operating System by simply returning from the IMB/IMBRange service routine AND that the service routines can be exactly the same. The following service routine code can be used for ARM810:

```
IMB_SWI_handler
IMBRange_SWI_handler

            MOVS  PC, R14_svc; Return to the code after the SWI call
```

Note: It is strongly encouraged that in code from now on, the IMBRange instruction is used whenever the changed area of code is small: even if there is no distinction between it and the IMB instruction on ARM810. Future processors may well implement the IMBRange instruction in a much more efficient and faster manner, and code migrated from ARM810 will benefit when executed on these processors.

E.3 Generic IMB Use

Using SWI's to implement the IMB instructions means that any code that is written now will be compatible with any future processors - even if those processors implement IMB in different ways. This is achieved by changing the Operating System SWI service routines for each of the IMB SWI numbers that differ from processor to processor.

Below are examples that show what should happen during the execution of IMB instructions. These examples are taken from ***4.17.3 Examples*** on page 4-65.

The pseudo code in the square brackets shows what should happen to execute the IMB instruction (or IMBRange) in the SWI handler.

E.3.1 Loading code from disk

Code that loads a program from a disk, and then branches to the entry point of that program, must execute an IMB instruction between loading the program and trying to execute it.

```
IMBEQU 0xF00000
 .
 .
 ; code that loads program from disk
 .
 .
 SWI IMB
    [branch to IMB service routine]
    [perform processor-specific operations to execute IMB]
    [return to code]
    .
 MOV PC, entry_point_of_loaded_program
    .
    .
```

E.3.2 Running BitBlt code

“Compiled BitBlt” routines optimise large copy operations by constructing and executing a copying loop which has been optimised for the exact operation wanted. When writing such a routine an IMB is needed between the code that constructs the loop and the actual execution of the constructed loop.

```
IMBRange EQU 0xF00001
.
.
; code that constructs loop code
; load R0 with the start address of the constructed loop
; load R1 with the end address of the constructed loop
SWI   IMBRange
      [branch to IMBRange service routine]
      [read registers R0 and R1 to set up address range
            parameters]
      [perform processor-specific operations to execute
            IMBRange within address range]
      [return to code]
; start of loop code
.
.
```

Index

A

Access faults
    checking 8-20
Address translation 8-5
ALE pin
    use of 12-18

B

Boundary scan register 13-11
BYPASS
    public instruction 13-9
Bypass register 13-10

C

CLAMP
    public instruction 13-8
CLAMPZ
    public instruction 13-8
Cycle speed
    bus interface 12-2
Cycle types
    bus interface 12-4

D

DC parameters 14-1, 15-1
Device identification code register 13-11
Domain access control 8-19

E

External aborts 8-23
EXTEST
    public instruction 13-7

F

Fault address register 8-17
Fault checking 8-20
Fault status register 8-17

H

HIGHZ
    public instruction 13-8

I

IDC

cacheable bit 7-2
disable 7-3
enable 7-3
interaction with MMU and write buffer 8-24
operation 7-2
read-lock-write 7-3
reset 7-3
validity 7-2
IDCODE
public instruction 13-8
Instruction register 13-6
Interface signals
boundary scan 13-13
Interrupts 3-8, 3-9, 3-10

M

Memory access
types of 12-29
use of the ALE pin 12-18
use of the nWAIT pin 12-16
MMU
interaction with IDC and write buffer 8-24

N

nWAIT pin
use of 12-16

P

Parameters
DC 14-1, 15-1
Public instructions 13-7
Pullup resistors 13-5

R

Registers
boundary scan interface 13-10
MMU 8-3
Registes
instruction 13-6

S

SAMPLE/PRELOAD
public instruction 13-7
Signal descriptions 2-3

T

Tap controller
boundary scan interface 13-4
Translating references 8-6

W

Write buffer
interaction with MMU and IDC 8-24