

ARM 710a macrocell

Preliminary Data Sheet



Document Number: ARM DDI 0033D

Issued: September 1995

Copyright Advanced RISC Machines Ltd (ARM) 1995

All rights reserved

Proprietary Notice

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

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

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

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

Change Log

Issue	Date	By	Change
A (Draft 0.1)	Dec 1994	AW	Created using ARM710a version C and ARM710C version C Data Sheets.
B	Jan 1995	AW	First formal release.
C draft1	Aug 1995	AP	Changes as a result of review.
D	Sep 1995	AP	Bus modified.

Contents

1	Introduction	1-1
	1.1 Introduction	1-2
	1.2 Block Diagram	1-4
	1.3 Functional Diagram	1-5
2	Signal Description	2-1
	2.1 Signal Descriptions	2-2
3	Programmer's Model	3-1
	3.1 Register Configuration	3-2
	3.2 Operating Mode Selection	3-4
	3.3 Registers	3-4
	3.4 Exceptions	3-7
	3.5 Reset	3-11
4	Instruction Set	4-1
	4.1 Instruction Set Summary	4-2
	4.2 The Condition Field	4-3
	4.3 Branch and Branch with link (B, BL)	4-4
	4.4 Data Processing	4-6
	4.5 PSR Transfer (MRS, MSR)	4-15
	4.6 Multiply and Multiply-Accumulate (MUL, MLA)	4-19
	4.7 Single Data Transfer (LDR, STR)	4-21
	4.8 Block Data Transfer (LDM, STM)	4-27
	4.9 Single Data Swap (SWP)	4-34
	4.10 Software Interrupt (SWI)	4-36
	4.11 Coprocessor Instructions on ARM710a macrocell	4-38
	4.12 Coprocessor Data Operations (CDP)	4-39
	4.13 Coprocessor Data Transfers (LDC, STC)	4-41
	4.14 Coprocessor Register Transfers (MRC, MCR)	4-45
	4.15 Undefined instruction	4-48
	4.16 Instruction Set Examples	4-49
	4.17 Instruction Speed Summary	4-53



Contents

5	Configuration	5-1
5.1	Internal Coprocessor Instructions	5-2
5.2	Registers	5-3
6	Instruction and Data Cache (IDC)	6-1
6.1	Cacheable Bit	6-2
6.2	IDC Operation	6-2
6.3	IDC Validity	6-2
6.4	Read-lock-write	6-3
6.5	IDC Enable/Disable and Reset	6-3
7	Write Buffer (WB)	7-1
7.1	Bufferable Bit	7-2
7.2	Write Buffer Operation	7-2
8	Coprocessors	8-1
8.1	Coprocessors	8-2
9	Memory Management Unit	9-1
9.1	MMU Program Accessible Registers	9-3
9.2	Address Translation	9-4
9.3	Translation Process	9-5
9.4	Level One Descriptor	9-6
9.5	Page Table Descriptor	9-6
9.6	Section Descriptor	9-7
9.7	Translating Section References	9-8
9.8	Level Two Descriptor	9-9
9.9	Translating Small Page References	9-10
9.10	Translating Large Page References	9-11
9.11	MMU Faults and CPU Aborts	9-12
9.12	Fault Address & Fault Status Registers (FAR & FSR)	9-12
9.13	Domain Access Control	9-14
9.14	Fault Checking Sequence	9-15
9.15	External Aborts	9-17
9.16	Interaction of the MMU, IDC and Write Buffer	9-18
9.17	Effect of Reset	9-19
10	Bus Clocking	10-1
10.1	Fastbus Extension	10-2
10.2	Standard Mode	10-4
11	Bus Interface	11-1
11.1	ARM710a macrocell Cycle Speed	11-2
11.2	Bus Interface Signals	11-2
11.3	Cycle Types	11-3
11.4	Addressing Signals	11-8
11.5	Memory Request Signals	11-9
11.6	Data Signal Timing	11-10
11.7	Maximum Sequential Length	11-11
11.8	Read-lock-write	11-12
11.9	Use of the nWAIT pin	11-13
11.10	Use of the ALE Pin	11-14
11.11	Use of the nENDOUT Output	11-17
11.12	Bus interface Sampling Points	11-17

	11.13	Big-endian / Little-endian Operation	11-20
	11.14	Use of Byte Lane Selects (nBLS[3:0])	11-21
	11.15	Memory Access Sequence Summary	11-23
	11.16	ARM710a macrocell Cycle Type Summary	11-28
12		DC Parameters	12-1
	12.1	Absolute Maximum Ratings	12-2
	12.2	DC Operating Conditions	12-2
	12.3	DC Characteristics	12-3
13		AC Parameters in Standard Mode	13-1
	13.1	Test Conditions	13-2
	13.2	Relationship between FCLK & MCLK in Synchronous Mode	13-2
	13.3	Main Bus Signals	13-4
14		AC Parameters with Fastbus Extension	14-1
	14.1	Test Conditions	14-2
	14.2	Main Bus Signals	14-3



Contents

1

Introduction

This chapter provides an introduction to the ARM710a macrocell.

1.1	Introduction	1-2
1.2	Block Diagram	1-4
1.3	Functional Diagram	1-5



Introduction

1.1 Introduction

The ARM710a macrocell is a general purpose 32-bit microprocessor with 8kByte cache, enlarged write buffer and Memory Management Unit (MMU) for use as a macrocell in highly integrated, high-performance system solutions. The CPU within ARM710a macrocell is the ARM7. The ARM710a macrocell is software compatible with the ARM processor family.

The ARM710a macrocell architecture is based on 'Reduced Instruction Set Computer' (RISC) principles, and the instruction set and related decode mechanism are greatly simplified compared with microprogrammed 'Complex Instruction Set Computers' (CISC).

The mixed data and instruction cache together with the write buffer substantially raise the average execution speed and reduce the average amount of memory bandwidth required by the processor. This allows the external bus structure to support additional processors or Direct Memory Access (DMA) channels with minimal performance loss.

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 instruction set comprises ten basic instruction types:

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

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

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

ARM710a macrocell is a fully static macrocell and has been designed to minimise its power requirements. This makes it ideal for portable applications where both these features are essential.

Datasheet notation:

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

ARM710a macrocell is a macrocell variant of the ARM710a, differing from it in the following respects:

- no IEEE 1149.1 test interface
- it is an unpackaged macrocell
- CMOS signal interface
- separate data in and data out buses

See also [Appendix A, Differences between ARM610 and ARM710a macrocell](#) for further details.



Introduction

1.2 Block Diagram

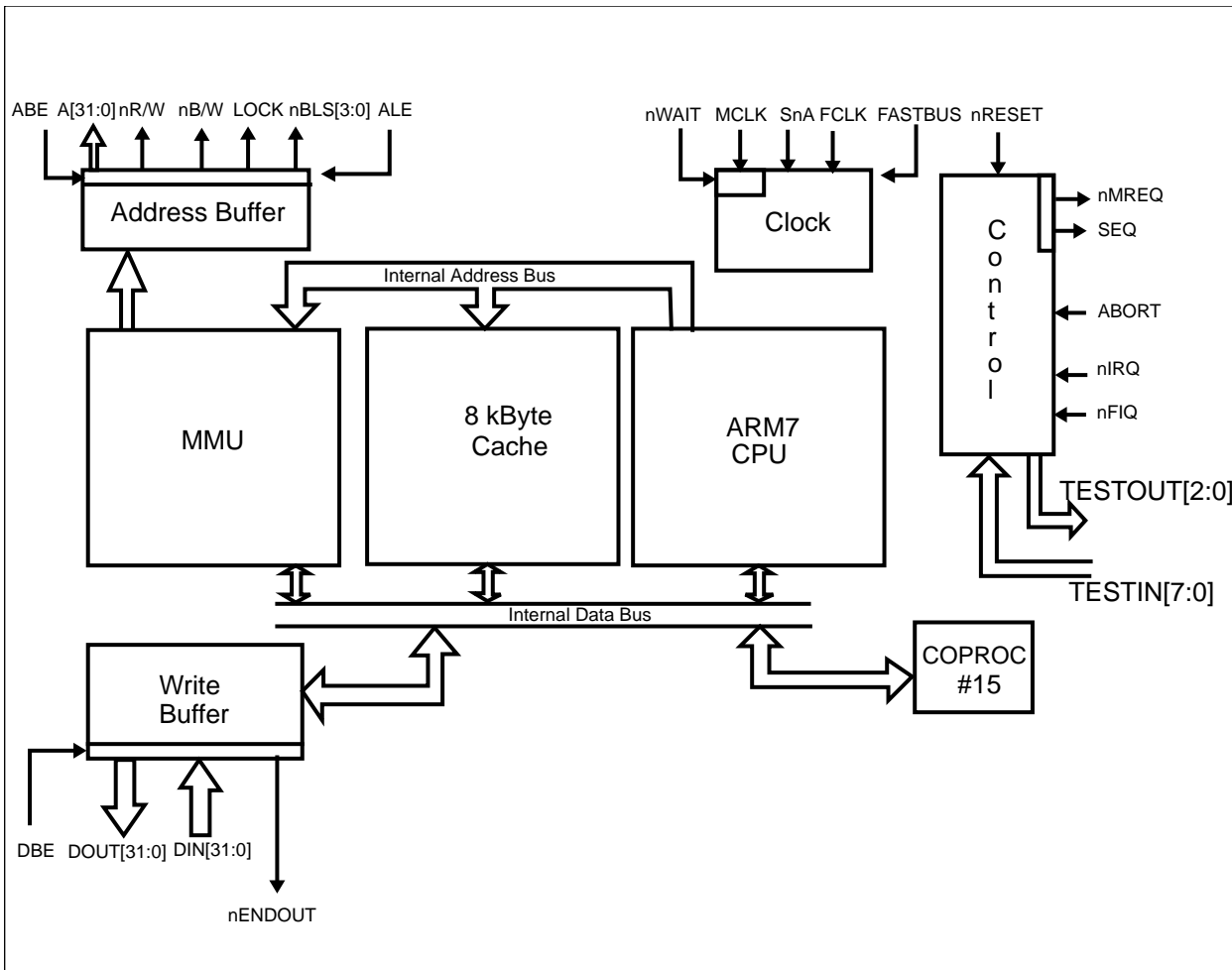


Figure 1-1: ARM710a macrocell block diagram

1.3 Functional Diagram

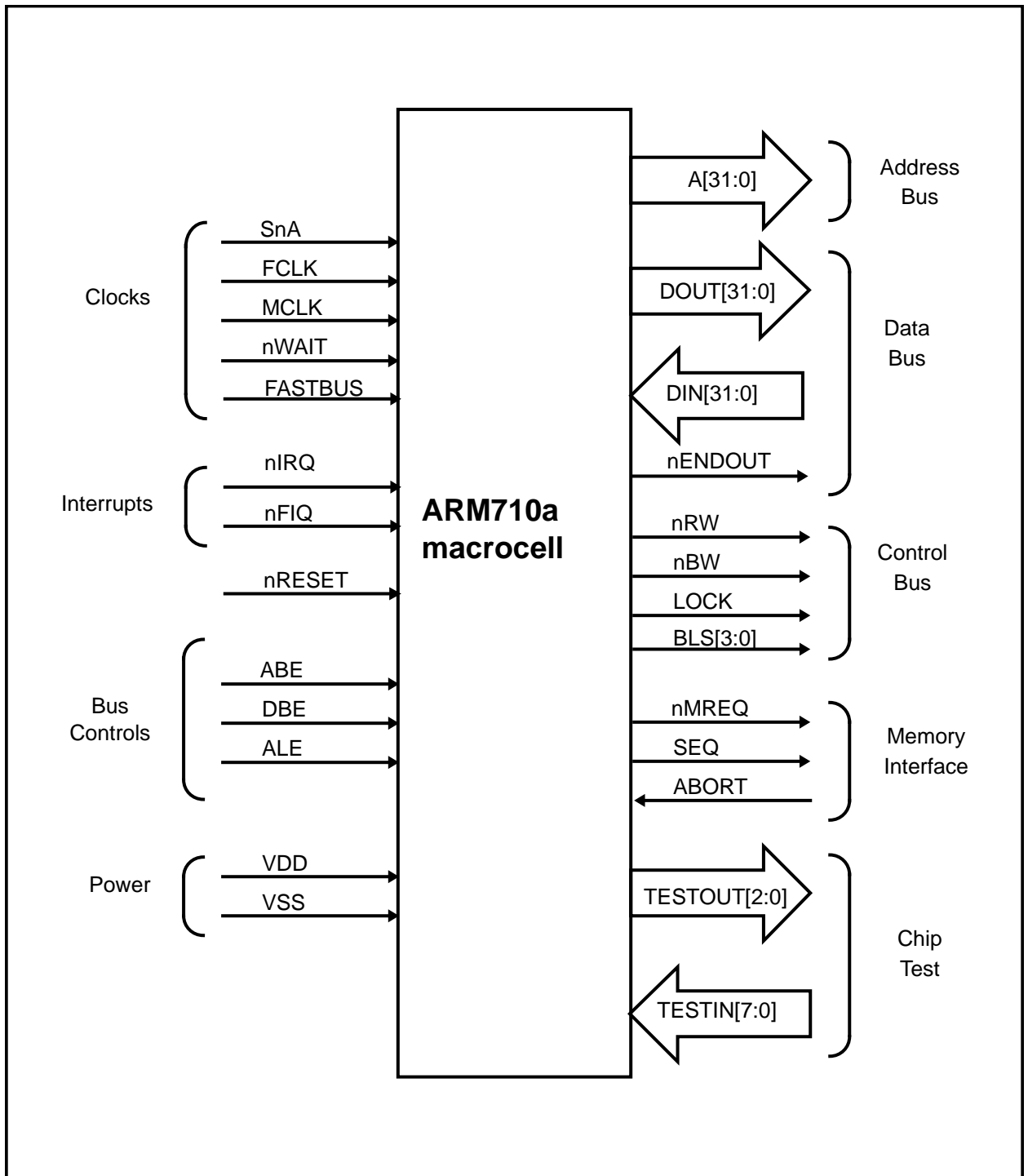


Figure 1-2: Functional diagram

Introduction

2

Signal Description

This chapter describes the signals.

2.1 Signal Descriptions

2-2



Signal Description

2.1 Signal Descriptions

Name	Type	Description:
A[31:0]	OCZ	Address bus. This bus signals the address requested for memory accesses. Normally it changes during MCLK HIGH, subject to ALE .
ABE	IC	Address bus enable. When this input is LOW, the address bus A[31:0] , nRW , nBLS[3:0] , nBW and LOCK are put into a high impedance state (Note 1).
ABORT	IC	External abort. Allows the memory system to tell the processor that a requested access has failed. Only monitored when ARM710a macrocell is accessing external memory.
ALE	IC	Address latch enable. This input is used to control transparent latches on the address bus A[31:0] , nBW , nBLS[3:0] , nRW & LOCK . Normally these signals change during MCLK HIGH, but they may be held by driving ALE LOW. The functionality of this signal changes with and without Fastbus extension, see 11.10 Use of the ALE Pin on page 11-14.
DIN[31:0]	IC	Input data bus. During read operations (when nRW is LOW), the input data must be valid before the falling edge of MCLK . To avoid dissipating static current, ensure that this bus is always driven to a known value.
DOUT[31:0]	OC	Output data bus. During write operations (when nRW is HIGH), the output data will become valid during MCLK LOW. At high clock frequencies the data may not become valid until just after the rising edge of MCLK (see 13.3 Main Bus Signals on page 13-4). The value of this bus is undefined when the signal nENDOUT is HIGH.
DBE	IC	Data bus enable. When this input is LOW, the nENDOUT output is forced HIGH. This can be used to force the ARM710a macrocell off the system databus. DBE should be HIGH at all times when the ARM is allowed to drive the system bus.
FCLK	ICK	Fast clock input, only used without fastbus extension. When the ARM710a macrocell CPU is accessing the cache or performing an internal cycle in this mode, it is clocked with the Fast Clock, FCLK .
FASTBUS	IC	Bus mode select signal. Selects between the standard mode bus interface, and clocking, and the ARM710a macrocell fastbus mode. When LOW selects ARM710 bus, when HIGH selects fastbus mode.
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 changes while MCLK is HIGH and remains HIGH during the locked memory sequence. LOCK is latched by ALE .
MCLK	ICK	Memory clock input. This clock times all ARM710a macrocell memory accesses. The LOW or HIGH period of MCLK may be stretched for slow peripherals; alternatively, the nWAIT input may be used with a free-running MCLK to achieve similar effects.

Table 2-1: Signal descriptions

Signal Description

Name	Type	Description:
nBLS[3:0]	OCZ	Not Byte Lane Selects. These are active LOW and signify which bytes of the memory are being accessed. For a word access all will be LOW. Normally they change during MCLK HIGH, subject to ALE (see 11.14 Use of Byte Lane Selects (nBLS[3:0]) on page 11-21).
nBW	OCZ	Not byte / word. An output signal used by the processor to indicate to the external memory system when a data transfer of a byte length is required. nBW is HIGH for word transfers and LOW for byte transfers, and is valid for both read and write operations. The signal changes while MCLK is HIGH. nBW is latched by ALE .
nENDOUT	OC	Not enable data out. This active LOW signal can be used to control tri-state drivers connected to DOUT[31:0] . It will be LOW during write cycles on the bus. It is conditioned by the DBE input, and will be HIGH when DBE is LOW.
nFIQ	IC	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, and must be held LOW until a suitable response is received from the processor.
nIRQ	IC	Not interrupt request. As nFIQ , but with lower priority. May be taken LOW asynchronously to interrupt the processor when the IRQ enable is active.
nMREQ	OC	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.
nRESET	IC	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 2 full FCLK cycles or 5 full MCLK cycles whichever is greater. While nRESET is LOW the processor will perform idle cycles with incrementing addresses and nWAIT must be HIGH.
nRW	OCZ	Not read/write. When HIGH this signal indicates a processor write operation; when LOW, a read. The signal changes while MCLK is HIGH. nRW is latched by ALE .
nWAIT	IC	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.
SEQ	OC	Sequential address. This signal is the inverse of nMREQ , and is provided for compatibility with existing ARM memory systems.
SnA	IC	Synchronous / not Asynchronous. In standard ARM bus mode this signal determines the bus interface mode and should be wired HIGH or LOW depending on the desired relationship between FCLK and MCLK in the application. See Chapter 11, Bus Interface . This signal is ignored when operating with the fastbus extension.

Table 2-1: Signal descriptions (Continued)

Signal Description

Name	Type	Description:
TESTIN[7:0]	IC	Test bus input. This bus is used for testing of the device. When in normal operation, all of these signals must be tied LOW.
TESTOUT[2:0]	OC	Test bus output. This bus is used for testing of the device. When all the TESTIN[7:0] signals are driven LOW, these three outputs will be driven LOW.
VDD		Positive supply.
VSS		Ground supply.

Table 2-1: Signal descriptions (Continued)

Notes

- 1 When outputs 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.

Key to signal types:

IC	Input, CMOS threshold
OC	Output, CMOS levels
OCZ	Output, CMOS levels, tri-stateable
ICK	Clock input, CMOS levels



3

Programmer's Model

This chapter describes the programmer's model.

3.1	Register Configuration	3-2
3.2	Operating Mode Selection	3-4
3.3	Registers	3-4
3.4	Exceptions	3-7
3.5	Reset	3-11

Programmer's Model

ARM710a macrocell supports a variety of operating configurations. Some are controlled by register bits and are known as the *register configurations*. Others may be controlled by software and these are known as *operating modes*.

3.1 Register Configuration

The ARM710a macrocell processor provides 3 register configuration settings which may be changed while the processor is running and which are discussed below.

3.1.1 Big and little-endian (the bigend bit)

The bigend bit in the Control Register sets whether the ARM710a macrocell treats words in memory as being stored in big-endian or little-endian format. See [Chapter 5, Configuration](#) for more information on the Control Register. Memory is viewed as a linear collection of bytes numbered upwards from zero. Bytes 0 to 3 hold the first stored word, bytes 4 to 7 the second and so on.

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

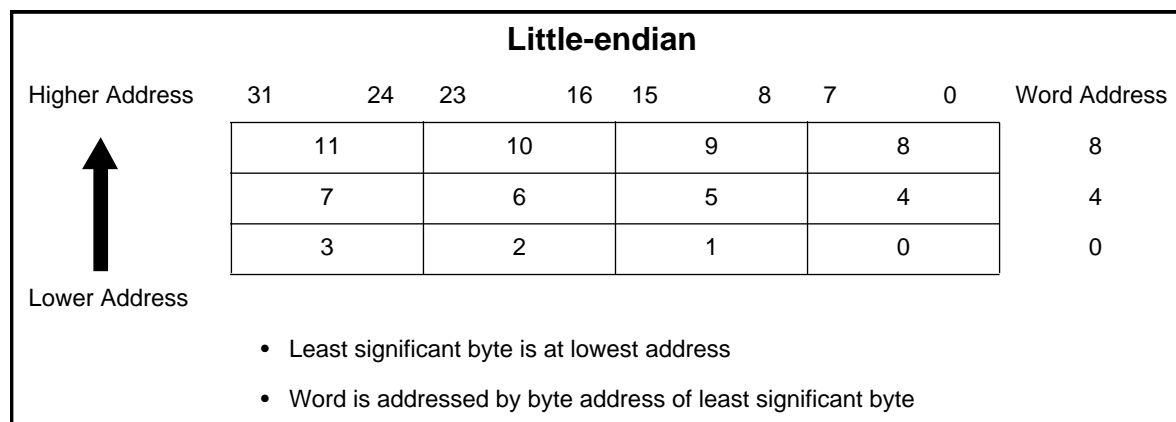


Figure 3-1: Little-endian addresses of bytes within word

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 endian-ness: see [4.7 Single Data Transfer \(LDR, STR\)](#) on page 4-21 for more details.

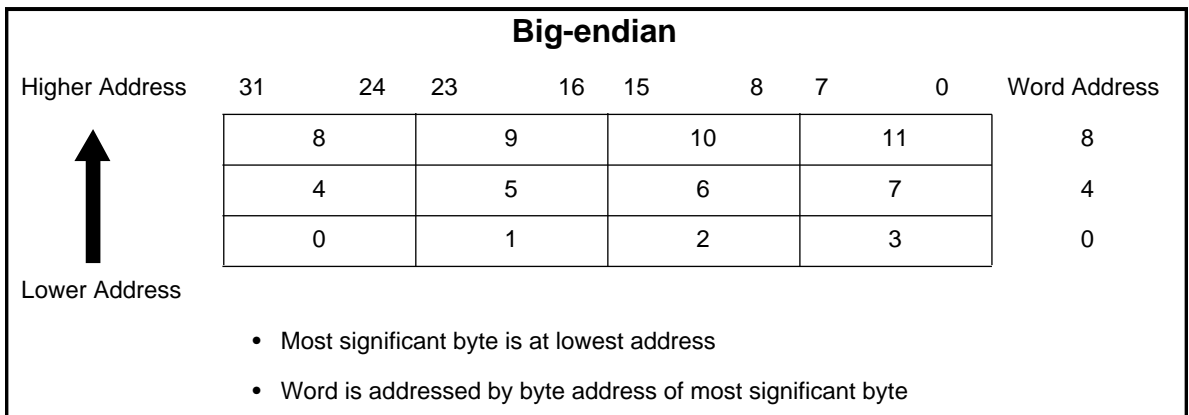


Figure 3-2: Big-endian addresses of bytes within words

3.1.2 Configuration bits for backward compatibility

The other two configuration bits, prog32 and data32, are used for backward compatibility with earlier ARM processors but should normally be set to 1. This configuration extends the address space to 32 bits, introduces major changes in the programmer's model as described below, and provides support for running existing 26 bit programs in the 32 bit environment. This mode is recommended for compatibility with future ARM processors and all new code should be written to use only the 32 bit operating modes.

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

Programmer's Model

3.2 Operating Mode Selection

ARM710a macrocell has a 32 bit data bus and a 32 bit address bus. The processor supports *byte* (8 bit) and *word* (32 bit) data types, where words must be aligned to four byte boundaries. Instructions are exactly one word, and data operations (eg ADD) are only performed on word quantities. Load and store operations can transfer either bytes or words.

ARM710a macrocell supports six modes of operation:

- 1 User mode (usr): the normal program execution state
- 2 FIQ mode (fiq): fast interrupt for data transfer or channel processes
- 3 IRQ mode (irq): used for general purpose interrupt handling
- 4 Supervisor mode (svc): a protected mode for the operating system
- 5 Abort mode (abt): entered after a data or instruction prefetch abort
- 6 Undefined mode (und): entered when an undefined instruction is executed

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

3.3 Registers

The processor has a total of 37 registers made up of 31 general 32 bit registers and 6 status registers. At any one time 16 general registers (R0 to R15) and one or two status registers are visible to the programmer. The visible registers depend on the processor mode. The other registers, known as the *banked registers*, are switched in to support IRQ, FIQ, Supervisor, Abort and Undefined mode processing. ♦Figure 3-3: *Register organisation* on page 3-5 shows how the registers are arranged, with the banked registers shaded.

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

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

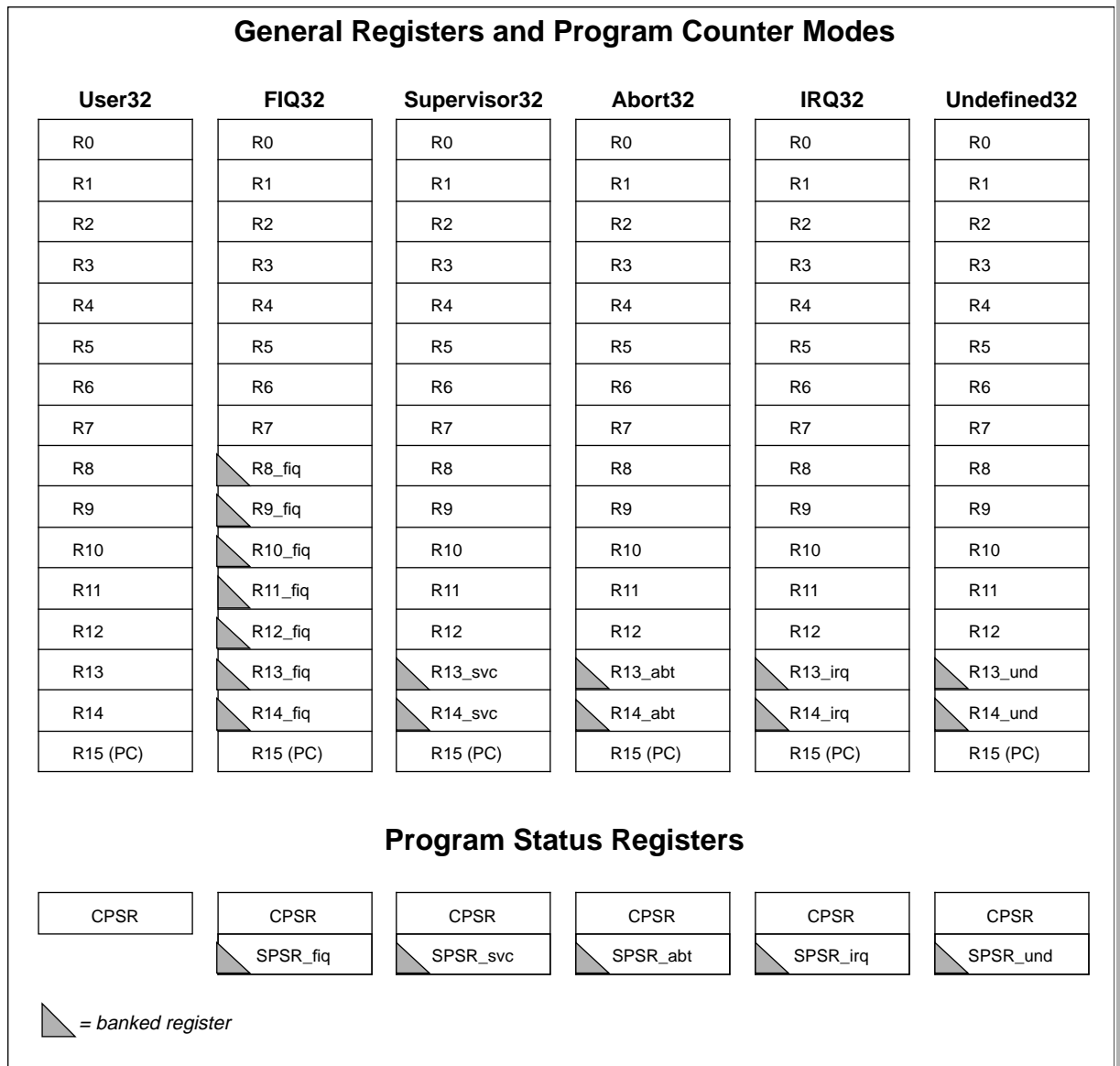


Figure 3-3: Register organisation

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

Programmer's Model

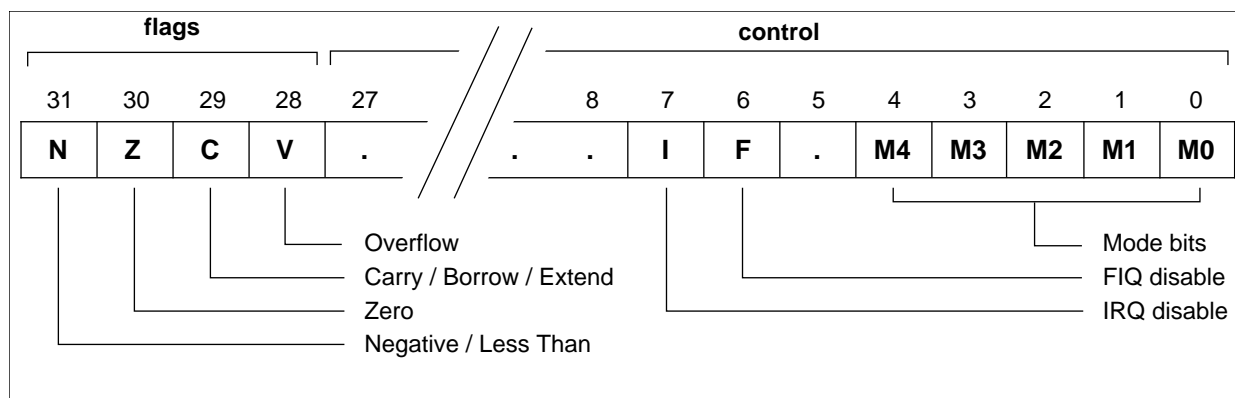


Figure 3-4: Format of the program status registers (PSRs)

The format of the Program Status Registers is shown in **Figure 3-4: Format of the program status registers (PSRs)**. The N, Z, C and V bits are the *condition code flags*. The condition code flags in the CPSR may be changed as a result of arithmetic and logical operations in the processor and may be tested by all instructions to determine if the instruction is to be executed.

The I and F bits are the *interrupt disable bits*. The I bit disables IRQ interrupts when it is set and the F bit disables FIQ interrupts when it is set. The M0, M1, M2, M3 and M4 bits (M[4:0]) are the *mode bits*, and these determine the mode in which the processor operates. The interpretation of the mode bits is shown in **Table 3-1: The mode bit**. Not all bit combinations define a valid processor mode. Only those explicitly described shall be used. The user should be aware that if any illegal value is programmed into the mode bits, M[4:0], the processor will enter an unrecoverable state. If this occurs, reset should be applied.

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

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

Table 3-1: The mode bit

3.4 Exceptions

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

ARM710a macrocell handles exceptions by making use of the banked registers to save state. The old PC and CPSR contents are copied into the appropriate R14 and SPSR and the PC and mode bits in the CPSR bits are forced to a value which depends on the exception. Interrupt disable flags are set where required to prevent otherwise unmanageable nestings of exceptions. In the case of a re-entrant interrupt handler, R14 and the SPSR should be saved onto a stack in main memory before re-enabling the interrupt; when transferring the SPSR register to and from a stack, it is important to transfer the whole 32 bit value, and not just the flag or control fields. When multiple exceptions arise simultaneously, a fixed priority determines the order in which they are handled. This is listed later in [3.4.7 Exception priorities](#) on page 3-11.

3.4.1 FIQ

The FIQ (Fast Interrupt reQuest) exception is externally generated by taking the **nFIQ** input LOW. This input can except asynchronous transitions, and is delayed by one clock cycle for synchronisation before it can affect the processor execution flow. FIQ is designed to support a data transfer or channel process, and has sufficient private registers to remove the need for register saving in such applications (thus minimising the overhead of context switching). The FIQ exception may be disabled by setting the F flag in the CPSR (but note that this is not possible from User mode). If the F flag is clear, ARM710a macrocell checks for a LOW level on the output of the FIQ synchroniser at the end of each instruction.

When a FIQ is detected, ARM710a macrocell:

- 1 Saves the address of the next instruction to be executed plus 4 in R14_fiq; saves CPSR in SPSR_fiq
- 2 Forces M[4:0]=10001 (FIQ mode) and sets the F and I bits in the CPSR
- 3 Forces the PC to fetch the next instruction from address 0x1C

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

Programmer's Model

3.4.2 IRQ

The IRQ (Interrupt ReQuest) exception is a normal interrupt caused by a LOW level on the **nIRQ** input. It has a lower priority than FIQ, and is masked out when a FIQ sequence is entered. Its effect may be masked out at any time by setting the I bit in the CPSR (but note that this is not possible from User mode). If the I flag is clear, ARM710a macrocell checks for a LOW level on the output of the IRQ synchroniser at the end of each instruction. When an IRQ is detected, ARM710a macrocell:

- 1 Saves the address of the next instruction to be executed plus 4 in R14_irq; saves CPSR in SPSR_irq
- 2 Forces M[4:0]=10010 (IRQ mode) and sets the I bit in the CPSR
- 3 Forces the PC to fetch the next instruction from address 0x18

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

3.4.3 Abort

An abort can be signalled by either the internal Memory Management Unit or from the external **ABORT** input. An abort indicates that the current memory access cannot be completed. For instance, in a virtual memory system the data corresponding to the current address may have been moved out of memory onto a disc, and considerable processor activity may be required to recover the data before the access can be performed successfully. ARM710a macrocell checks for aborts during memory access cycles. When successfully aborted ARM710a macrocell will respond in one of two ways:

- 1 If the abort occurred during an instruction prefetch (a *Prefetch Abort*), the prefetched instruction is marked as invalid but the abort exception does not occur immediately. If the instruction is not executed, for example as a result of a branch being taken while it is in the pipeline, no abort will occur. An abort will take place if the instruction reaches the head of the pipeline and is about to be executed.
- 2 If the abort occurred during a data access (a *Data Abort*), the action depends on the instruction type.
 - a) Single data transfer instructions (LDR, STR) will write back modified base registers and the Abort handler must be aware of this.
 - b) The swap instruction (SWP) is aborted as though it had not executed, though externally the read access may take place.
 - c) Block data transfer instructions (LDM, STM) complete, and if write-back is set, the base is updated. If the instruction would normally have overwritten the base with data (i.e. LDM with the base in the transfer list), this overwriting is prevented. All register overwriting is prevented after the Abort is indicated, which means in particular that R15 (which is always last to be transferred) is preserved in an aborted LDM instruction.

If the MMU is enabled, an encoded 4-bit status value and the 4-bit domain number are placed in the FSR (fault status register). The virtual address which caused the abort is placed in the FAR (fault address register). See [9.12 Fault Address & Fault Status Registers \(FAR & FSR\)](#) on page 9-12.

When either a prefetch or data abort occurs, ARM710a macrocell:

- 1 Saves the address of the aborted instruction plus 4 (for prefetch aborts) or 8 (for data aborts) in R14_abt; saves CPSR in SPSR_abt.
- 2 Forces M[4:0]=10111 (Abort mode) and sets the I bit in the CPSR.
- 3 Forces the PC to fetch the next instruction from either address 0x0C (prefetch abort) or address 0x10 (data abort).

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

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

Note *The ARM710a macrocell only implements the late abort configuration. ARM610 designs should be directly compatible as long as they used late aborts.*

Note that there are restrictions on the use of the external abort signal. See [9.15 External Aborts](#) on page 9-17.

3.4.4 Software interrupt

The software interrupt instruction (SWI) is used for getting into Supervisor mode, usually to request a particular supervisor function. When a SWI is executed, ARM710a macrocell:

- 1 Saves the address of the SWI instruction plus 4 in R14_svc; saves CPSR in SPSR_svc
- 2 Forces M[4:0]=10011 (Supervisor mode) and sets the I bit in the CPSR
- 3 Forces the PC to fetch the next instruction from address 0x08

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

Programmer's Model

3.4.5 Undefined instruction trap

When the ARM710a macrocell comes across an instruction which it cannot handle (see [Chapter 4, Instruction Set](#)), it will take the undefined instruction trap. This includes all coprocessor instructions, except MCR and MRC operations which access the internal control coprocessor.

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

When ARM710a macrocell takes the undefined instruction trap it:

- 1 Saves the address of the Undefined or coprocessor instruction plus 4 in R14_und; saves CPSR in SPSR_und.
- 2 Forces M[4:0]=11011 (Undefined mode) and sets the I bit in the CPSR
- 3 Forces the PC to fetch the next instruction from address 0x04

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

3.4.6 Vector summary

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

Table 3-2: Vector summary

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

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

3.4.7 Exception priorities

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

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

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

If a data abort occurs at the same time as a FIQ, and FIQs are enabled (i.e. the F flag in the CPSR is clear), ARM710a macrocell will enter the data abort handler and then immediately proceed to the FIQ vector. A normal return from FIQ will cause the data abort handler to resume execution. Placing data abort at a higher priority than FIQ is necessary to ensure that the transfer error does not escape detection; the time for this exception entry should be added to worst case FIQ latency calculations.

3.5 Reset

When the **nRESET** signal goes LOW, ARM710a macrocell abandons the executing instruction and then performs idle cycles from incrementing word addresses.

When **nRESET** goes HIGH again, ARM710a macrocell does the following:

- 1 Overwrites R14_svc and SPSR_svc by copying the current values of the PC and CPSR into them. The value of the saved PC and CPSR is not defined.
- 2 Forces M[4:0]=10011 (Supervisor mode) and sets the I and F bits in the CPSR.
- 3 Forces the PC to fetch the next instruction from address 0x00

At the end of the reset sequence, the MMU is disabled and the TLB is flushed, so forces "flat" translation (i.e. the physical address is the virtual address, and there is no permission checking); alignment faults are also disabled; the cache is disabled and flushed; the write buffer is disabled and flushed; the ARM7 CPU core is put into 26 bit data and address mode and little-endian mode.

Note that due to the reset synchronisers, there will be approximately 4 cycles between **nRESET** going HIGH and the fetch from 0x00.

Programmer's Model

This chapter describes the instruction set.

4.1	Instruction Set Summary	4-2
4.2	The Condition Field	4-3
4.3	Branch and Branch with link (B, BL)	4-4
4.4	Data Processing	4-6
4.5	PSR Transfer (MRS, MSR)	4-15
4.6	Multiply and Multiply-Accumulate (MUL, MLA)	4-19
4.7	Single Data Transfer (LDR, STR)	4-21
4.8	Block Data Transfer (LDM, STM)	4-27
4.9	Single Data Swap (SWP)	4-34
4.10	Software Interrupt (SWI)	4-36
4.11	Coprocessor Instructions on ARM710a macrocell	4-38
4.12	Coprocessor Data Operations (CDP)	4-39
4.13	Coprocessor Data Transfers (LDC, STC)	4-41
4.14	Coprocessor Register Transfers (MRC, MCR)	4-45
4.15	Undefined instruction	4-48
4.16	Instruction Set Examples	4-49
4.17	Instruction Speed Summary	4-53

Instruction Set - Summary

4.1 Instruction Set Summary

A summary of the ARM710a macrocell instruction set is shown in **Figure 4-1: Instruction set summary**.

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

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

Figure 4-1: Instruction set summary

4.2 The Condition Field

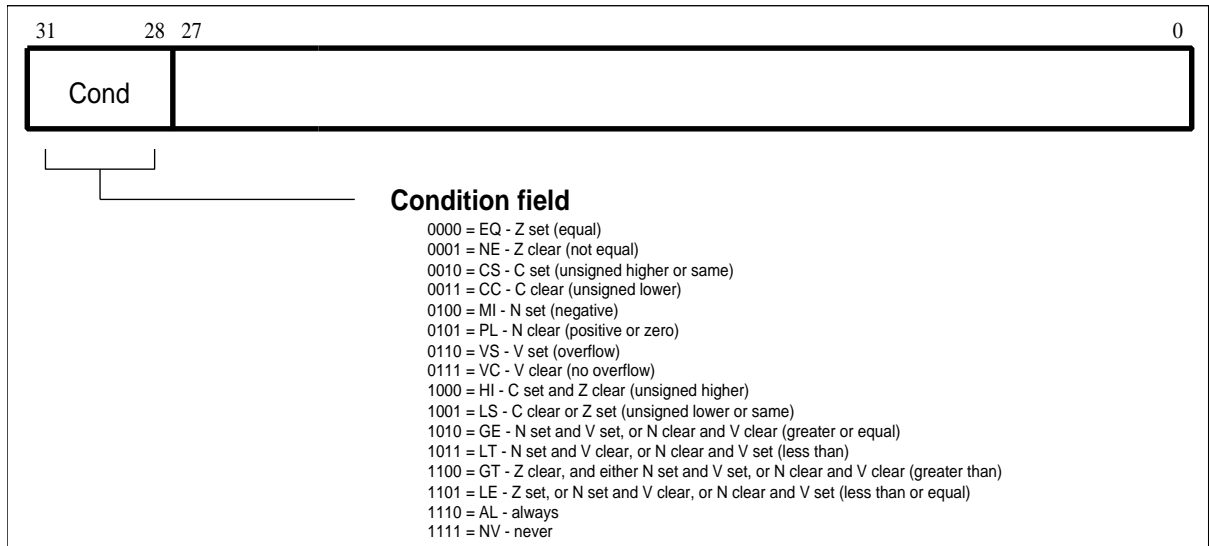


Figure 4-2: Condition codes

All ARM710a macrocell instructions are conditionally executed, which means that their execution may or may not take place depending on the values of the N, Z, C and V flags in the CPSR. The condition encoding is shown in **Figure 4-2: Condition codes**.

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

The other condition codes have meanings as detailed in **Figure 4-2: Condition codes**, for instance code 0000 (Equal) causes the instruction to be executed only if the Z flag is set. This would correspond to the case where a compare (CMP) instruction had found the two operands to be equal. If the two operands were different, the compare instruction would have cleared the Z flag and the instruction will not be executed.

Instruction Set - B, BL

4.3 Branch and Branch with link (B, BL)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in [Figure 4-3: Branch instructions](#) on page 4-4.

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

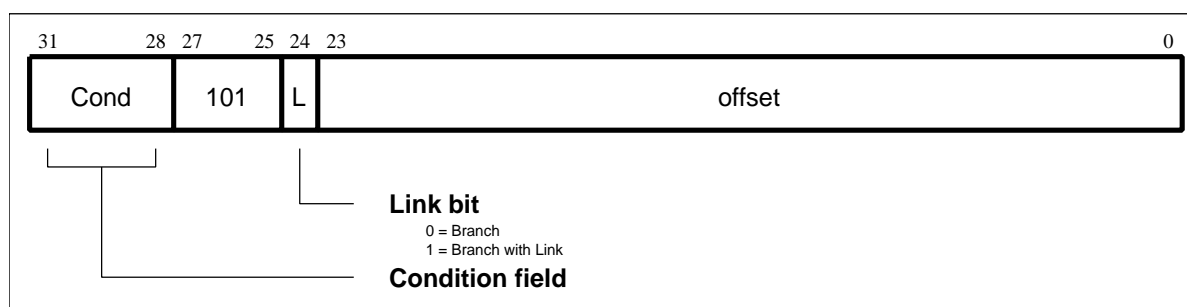


Figure 4-3: Branch instructions

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

4.3.1 The link bit

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

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

4.3.2 Instruction cycle times

Branch and Branch with Link instructions take 3 instruction fetches. For more information see [4.17 Instruction Speed Summary](#) on page 4-53.

4.3.3 Assembler syntax

B{L}{cond} <expression>

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

{cond} is a two-character mnemonic as shown in [Figure 4-2: Condition codes](#) on page 4-3 (EQ, NE, VS etc). If absent then AL (ALways) will be used.

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

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

4.3.4 Examples

```
here  BAL  here    ; assembles to 0xEAFFFFFFFFE (note effect
      B    there   of PC offset) 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 computed address

      ADDS 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
```

Instruction Set - Data processing

4.4 Data Processing

The instruction is only executed if the condition is true, defined at the beginning of this chapter. The instruction encoding is shown in [Figure 4-4: Data processing instructions](#) on page 4-7.

The instruction produces a result by performing a specified arithmetic or logical operation on one or two operands. The first operand is always a register (Rn). The second operand may be a shifted register (Rm) or a rotated 8 bit immediate value (Imm) according to the value of the I bit in the instruction. The condition codes in the CPSR may be preserved or updated as a result of this instruction, according to the value of the S bit in the instruction. Certain operations (TST, TEQ, CMP, CMN) do not write the result to Rd. They are used only to perform tests and to set the condition codes on the result and always have the S bit set. The instructions and their effects are listed in [Table 4-1: ARM data processing instructions](#) on page 4-8

Instruction Set - Data processing

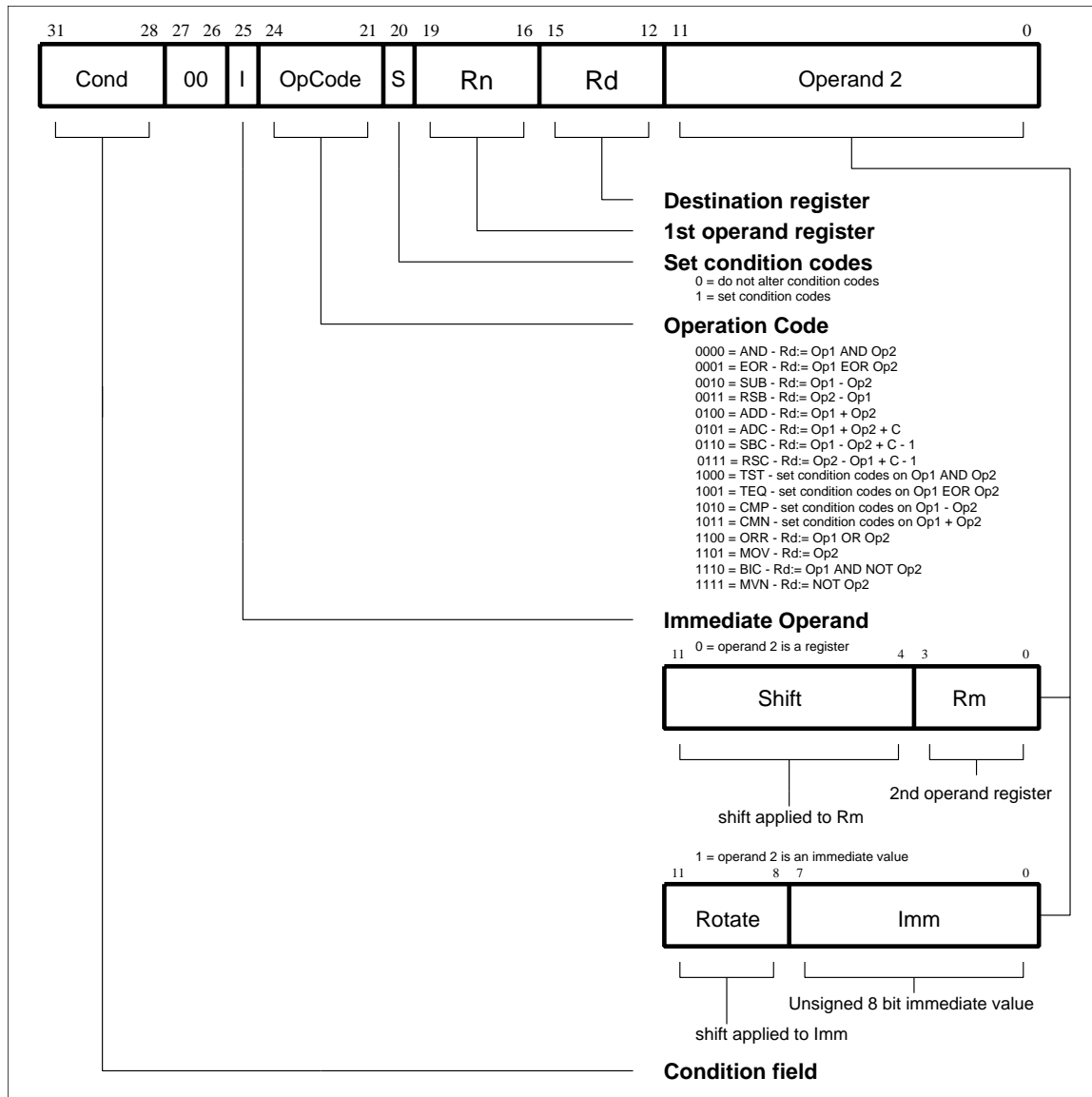


Figure 4-4: Data processing instructions

4.4.1 CPSR flags

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

Instruction Set - Data processing

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

Table 4-1: ARM data processing instructions

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

4.4.2 Shifts

When the second operand is specified to be a shifted register, the operation of the barrel shifter is controlled by the Shift field in the instruction. This field indicates the type of shift to be performed (logical left or right, arithmetic right or rotate right). The amount by which the register should be shifted may be contained in an immediate field in the instruction, or in the bottom byte of another register (other than R15). The encoding for the different shift types is shown in **Figure 4-5: ARM shift operations**.

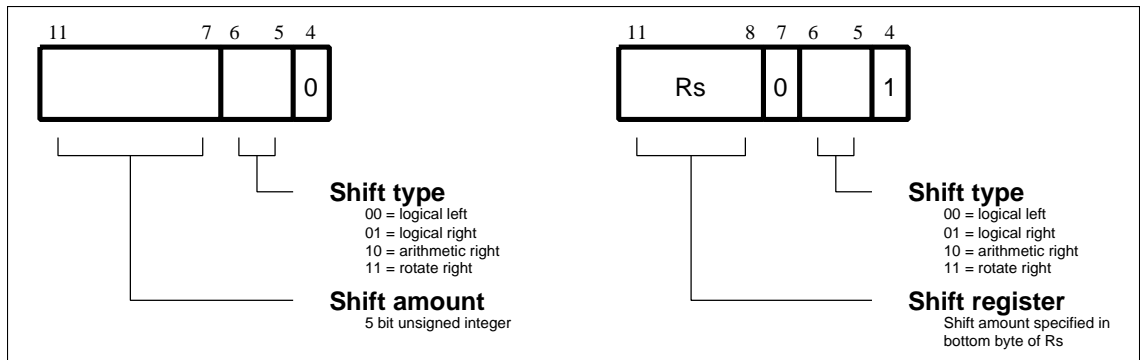


Figure 4-5: ARM shift operations

Instruction specified shift amount

When the shift amount is specified in the instruction, it is contained in a 5 bit field which may take any value from 0 to 31. A logical shift left (LSL) takes the contents of Rm and moves each bit by the specified amount to a more significant position. The least significant bits of the result are filled with zeros, and the high bits of Rm which do not map into the result are discarded, except that the least significant discarded bit becomes the shifter carry output which may be latched into the C bit of the CPSR when the ALU operation is in the logical class (see above). For example, the effect of LSL #5 is shown in **Figure 4-6: Logical shift left**.

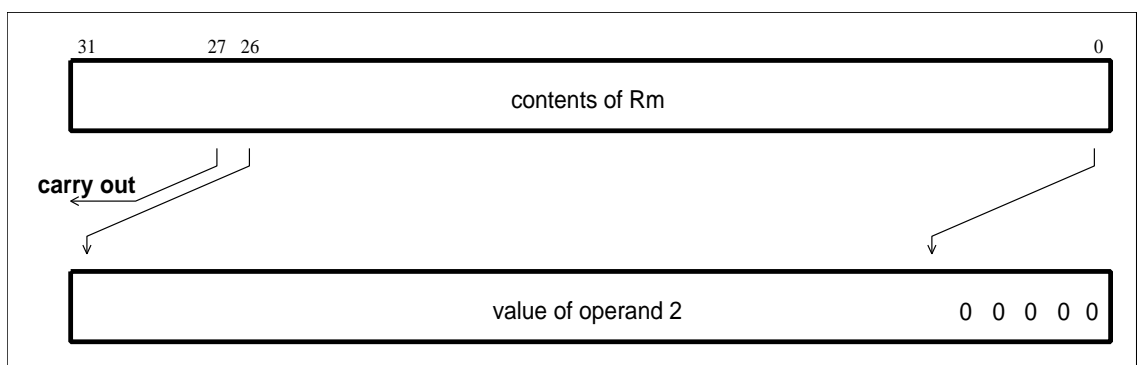


Figure 4-6: Logical shift left

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.

Instruction Set - Shifts

A logical shift right (LSR) is similar, but the contents of Rm are moved to less significant positions in the result. LSR #5 has the effect shown in [Figure 4-7: Logical shift right](#) on page 4-10.

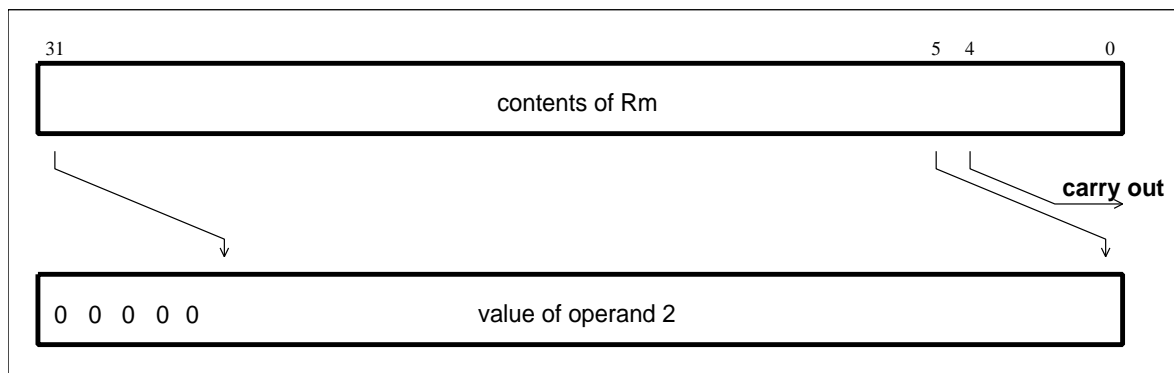


Figure 4-7: Logical shift right

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 will convert LSR #0 (and ASR #0 and ROR #0) into LSL #0, and allow LSR #32 to be specified.

An arithmetic shift right (ASR) is similar to logical shift right, except that the high bits are filled with bit 31 of Rm instead of zeros. This preserves the sign in 2's complement notation. For example, ASR #5 is shown in [Figure 4-8: Arithmetic shift right](#).

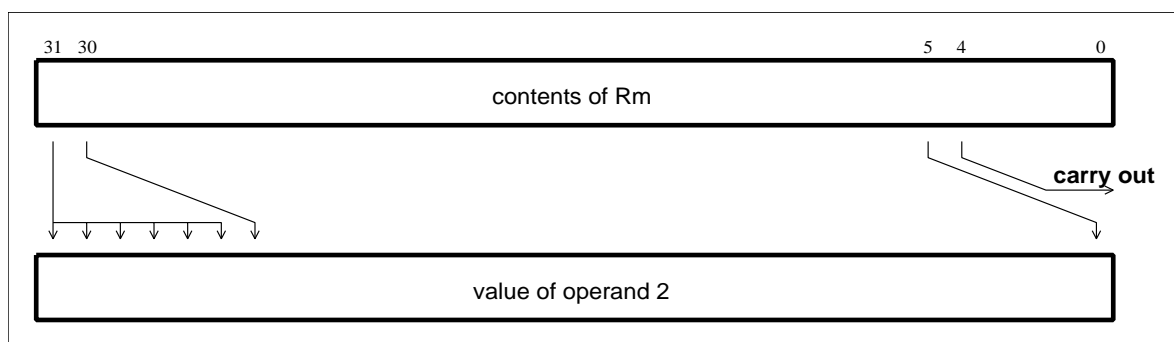


Figure 4-8: Arithmetic shift right

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, according to the value of bit 31 of Rm.

Rotate right (ROR) operations reuse the bits which 'overshoot' in a logical shift right operation by reintroducing them at the high end of the result, in place of the zeros used to fill the high end in logical right operations. For example, ROR #5 is shown in [Figure 4-9: Rotate right](#).

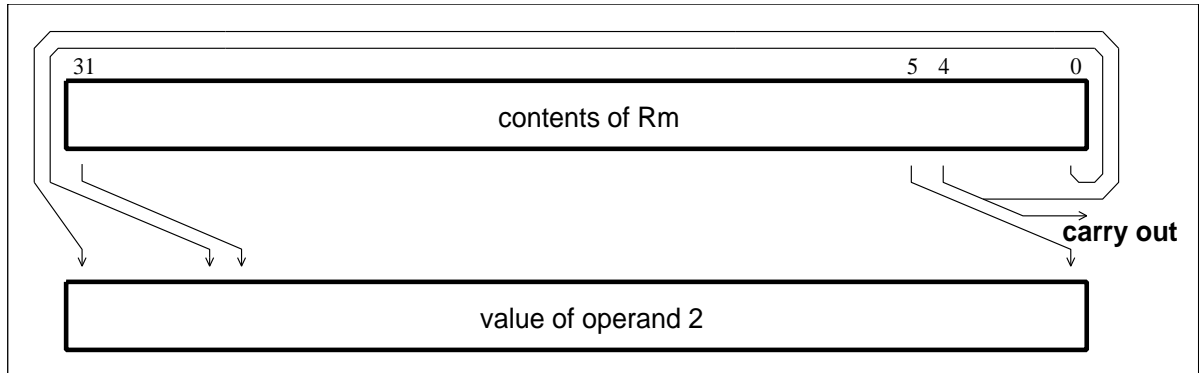


Figure 4-9: Rotate right

The form of the shift field which might be expected to give ROR #0 is used to encode a special function of the barrel shifter, rotate right extended (RRX). This is a rotate right by one bit position of the 33 bit quantity formed by appending the CPSR C flag to the most significant end of the contents of Rm as shown in [Figure 4-10: Rotate right extended](#).

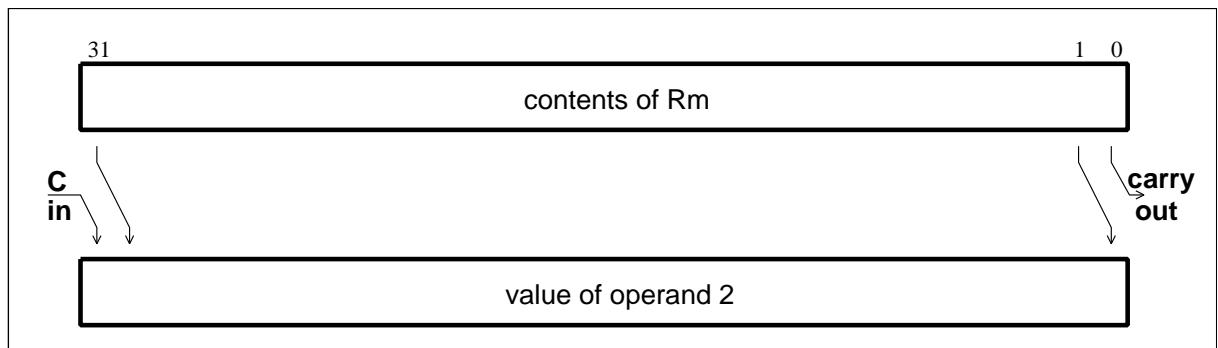


Figure 4-10: Rotate right extended

Register specified shift amount

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

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

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

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

- 1 LSL by 32 has result zero, carry out equal to bit 0 of Rm.
- 2 LSL by more than 32 has result zero, carry out zero.
- 3 LSR by 32 has result zero, carry out equal to bit 31 of Rm.

Instruction Set - TEQ, TST, CMP & CMN

- 4 LSR by more than 32 has result zero, carry out zero.
- 5 ASR by 32 or more has result filled with and carry out equal to bit 31 of Rm.
- 6 ROR by 32 has result equal to Rm, carry out equal to bit 31 of Rm.
- 7 ROR by n where n is greater than 32 will give the same result and carry out as ROR by n-32; therefore repeatedly subtract 32 from n until the amount is in the range 1 to 32 and see above.

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

4.4.3 Immediate operand rotates

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

4.4.4 Writing to R15

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

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

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

4.4.5 Using R15 as an operand

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

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

4.4.6 TEQ, TST, CMP & CMN opcodes

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

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

4.4.7 Instruction cycle times

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

Normal Data Processing	1 instruction fetch
Data Processing with register specified shift	1 instruction fetch + 1 internal cycle
Data Processing with PC written	3 instruction fetches
Data Processing with register specified shift and PC written	3 instruction fetches and 1 internal cycle

See [4.17 Instruction Speed Summary](#) on page 4-53 for more information.

4.4.8 Assembler syntax

- 1 MOV,MVN - single operand instructions
<opcode>{cond}{S} Rd,<Op2>
- 2 CMP,CMN,TEQ,TST - instructions which do not produce a result.
<opcode>{cond} Rn,<Op2>
- 3 AND,EOR,SUB,RSB,ADD,ADC,SBC,RSC,ORR,BIC
<opcode>{cond}{S} Rd,Rn,<Op2>

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

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

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

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

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

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

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

Instruction Set - TEQ, TST, CMP & CMN

4.4.9 Examples

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

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

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

MOV   PC,R14         ; return from subroutine

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

4.5 PSR Transfer (MRS, MSR)

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

The MRS and MSR instructions are formed from a subset of the Data Processing operations and are implemented using the TEQ, TST, CMN and CMP instructions without the S flag set. The encoding is shown in *Figure 4-11: PSR transfer* on page 4-16.

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

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

4.5.1 Operand restrictions

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

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

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

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

Instruction Set - MRS, MSR

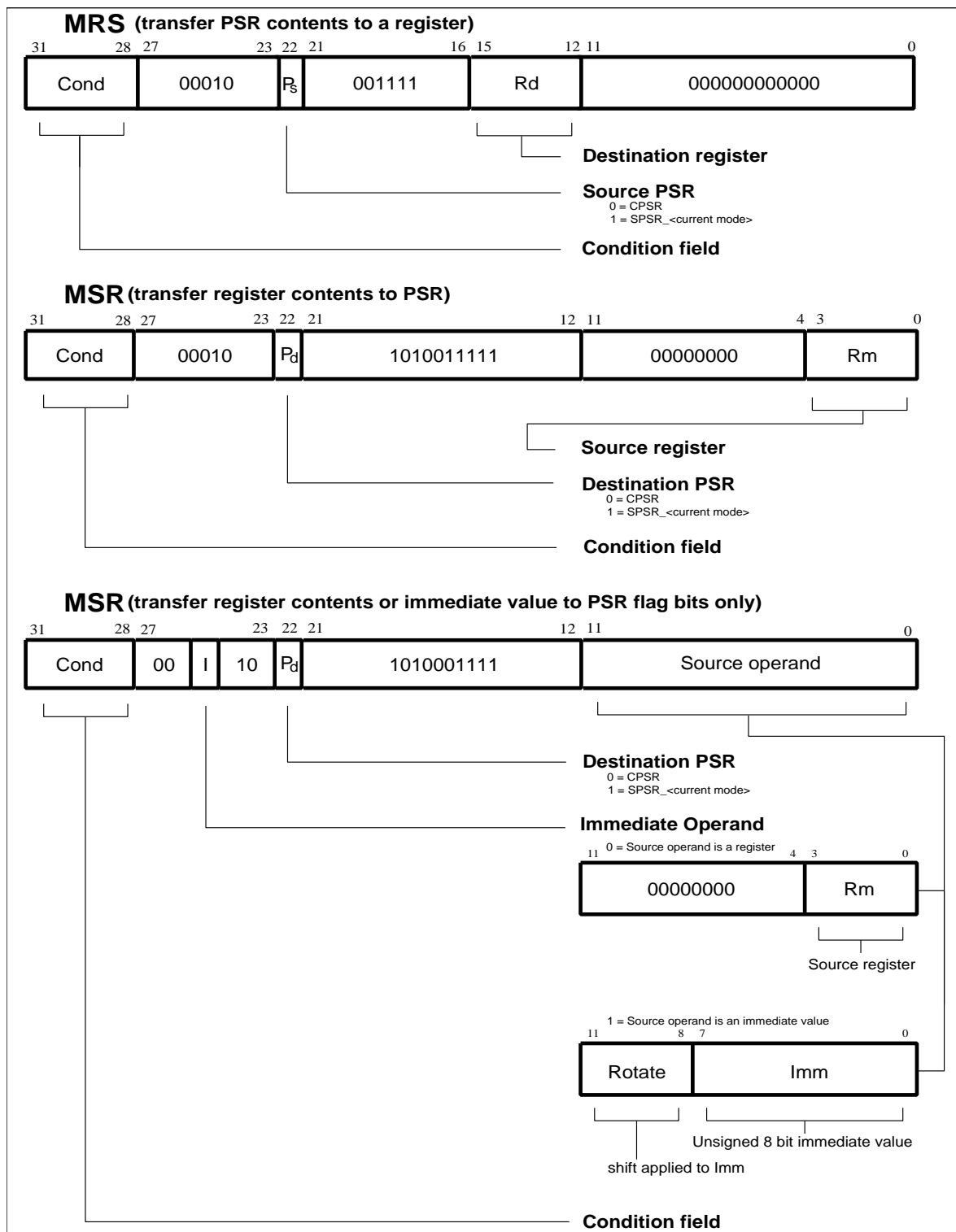


Figure 4-11: PSR transfer

4.5.2 Reserved bits

Only eleven bits of the PSR are defined in ARM710a macrocell (N,Z,C,V,I,F & M[4:0]); the remaining bits (PSR[27:8,5]) are reserved for use in future versions of the processor. To ensure the maximum compatibility between ARM710a macrocell programs and future processors, the following rules should be observed:

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

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

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

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

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

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

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

4.5.3 Instruction cycle times

PSR Transfers take 1 instruction fetch. For more information see [4.17 Instruction Speed Summary](#) on page 4-53.

4.5.4 Assembler syntax

- 1 MRS - transfer PSR contents to a register
MRS{cond} Rd,<psr>
- 2 MSR - transfer register contents to PSR
MSR{cond} <psr>,Rm
- 3 MSR - transfer register contents to PSR flag bits only
MSR{cond} <psrf>,Rm

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

Instruction Set - MRS, MSR

4 MSR - transfer immediate value to PSR flag bits only

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

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

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

Rd, Rm expressions evaluating to a register number other than R15

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

<psrf> CPSR_flg or SPSR_flg

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

4.5.5 Examples

In User mode the instructions behave as follows:

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

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

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

In privileged modes the instructions behave as follows:

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

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

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

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

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

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

4.6 Multiply and Multiply-Accumulate (MUL, MLA)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in **Figure 4-12: Multiply instructions**.

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

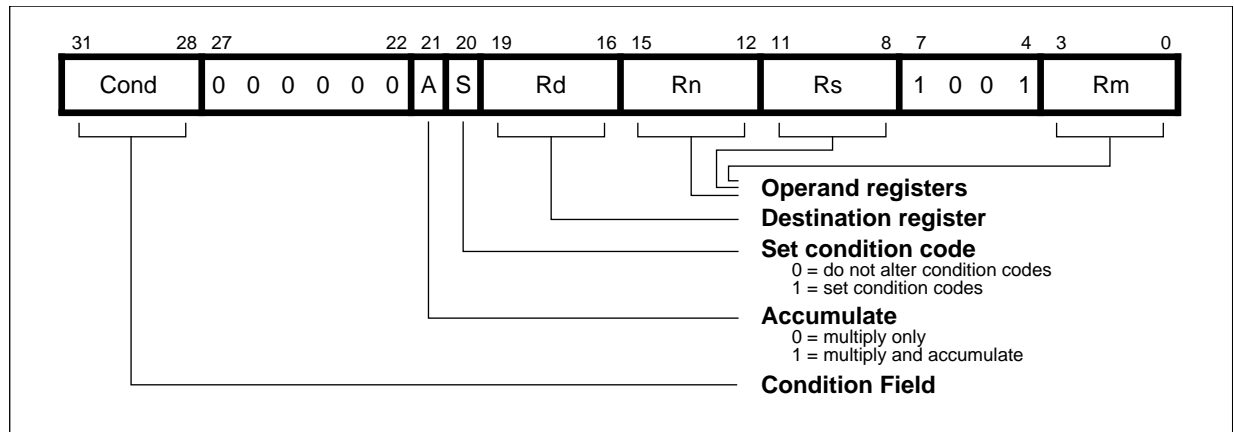


Figure 4-12: Multiply instructions

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

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

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

4.6.1 Operand restrictions

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

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

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

Instruction Set - MUL, MLA

4.6.2 CPSR flags

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

4.6.3 Instruction cycle times

The Multiply instructions take 1 instruction fetch and m internal cycles. For more information see section 4.17 Instruction Speed Summary on page 53.

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

4.6.4 Assembler syntax

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

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

{cond} two-character condition mnemonic, see [Figure 4-2: Condition codes](#) on page 4-3

{S} set condition codes if S present

Rd, Rm, Rs and Rn

expressions evaluating to a register number other than R15.

4.6.5 Examples

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


4.7 Single Data Transfer (LDR, STR)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in **Figure 4-13: Single data transfer instructions** on page 4-21.

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

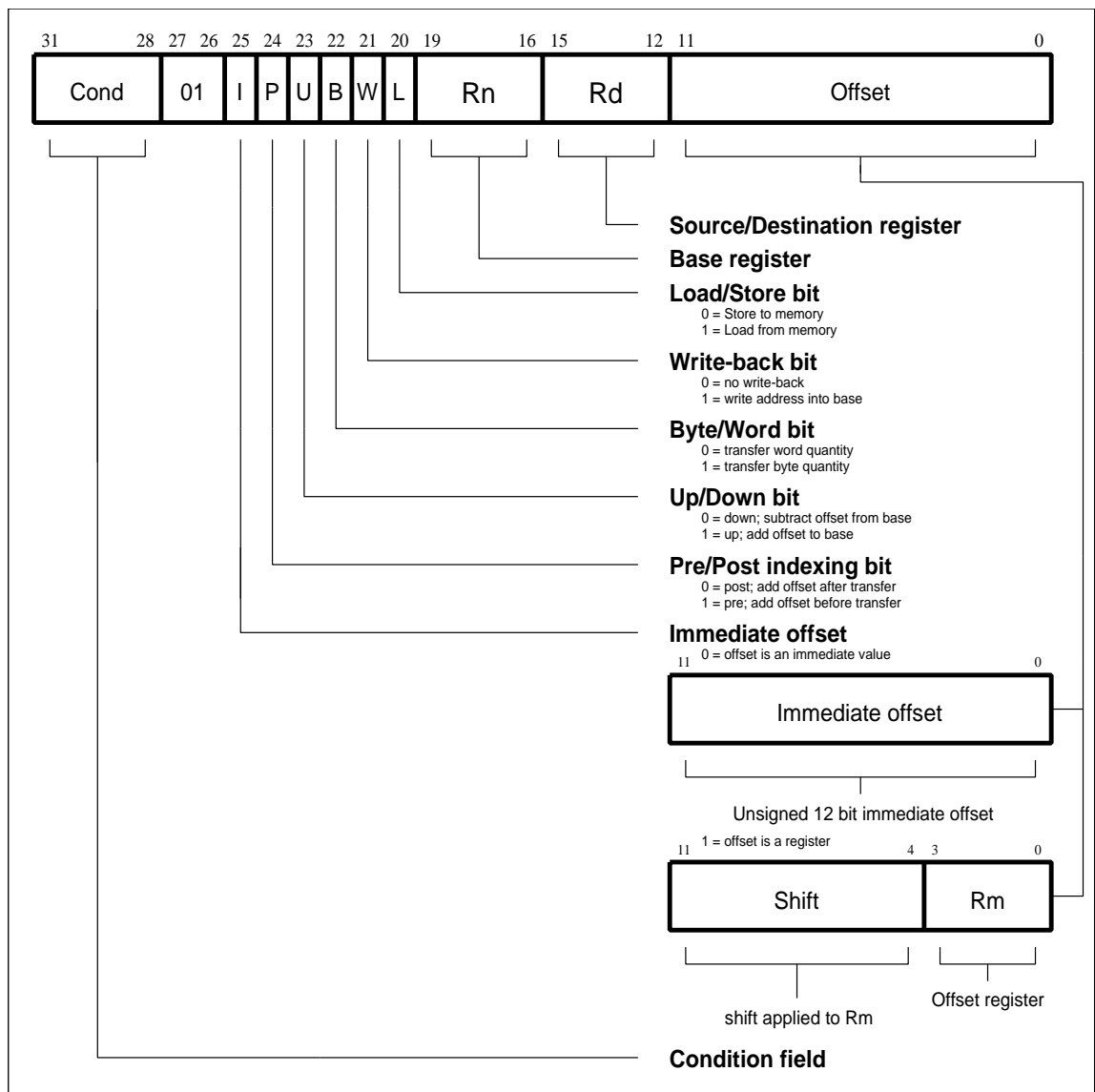


Figure 4-13: Single data transfer instructions

Instruction Set - LDR, STR

4.7.1 Offsets and auto-indexing

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

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

4.7.2 Shifted register offset

The 8 shift control bits are described in the data processing instructions section. However, the register specified shift amounts are not available in this instruction class. See [4.4.2 Shifts](#) on page 4-9.

4.7.3 Bytes and words

This instruction class may be used to transfer a byte (B=1) or a word (B=0) between an ARM710a macrocell register and memory.

The action of LDR(B) and STR(B) instructions is influenced by the 3 instruction fetches. For more information see [4.17 Instruction Speed Summary](#) on page 4-53. The two possible configurations are described below.

Little-endian configuration

A byte load (LDRB) expects the data on data bus inputs 7 through 0 if the supplied address is on a word boundary, on data bus inputs 15 through 8 if it is a word address plus one byte, and so on. The selected byte is placed in the bottom 8 bits of the destination register, and the remaining bits of the register are filled with zeros. Please see [Figure 3-2: Big-endian addresses of bytes within words](#) on page 3-3.

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

A word load (LDR) will normally use a word aligned address. However, an address offset from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits 0 to 7. This means that half-words accessed at offsets 0 and 2 from the word boundary will be correctly loaded into

bits 0 through 15 of the register. Two shift operations are then required to clear or to sign extend the upper 16 bits. This is illustrated in **Figure 4-14: Little-endian offset addressing** on page 4-23.

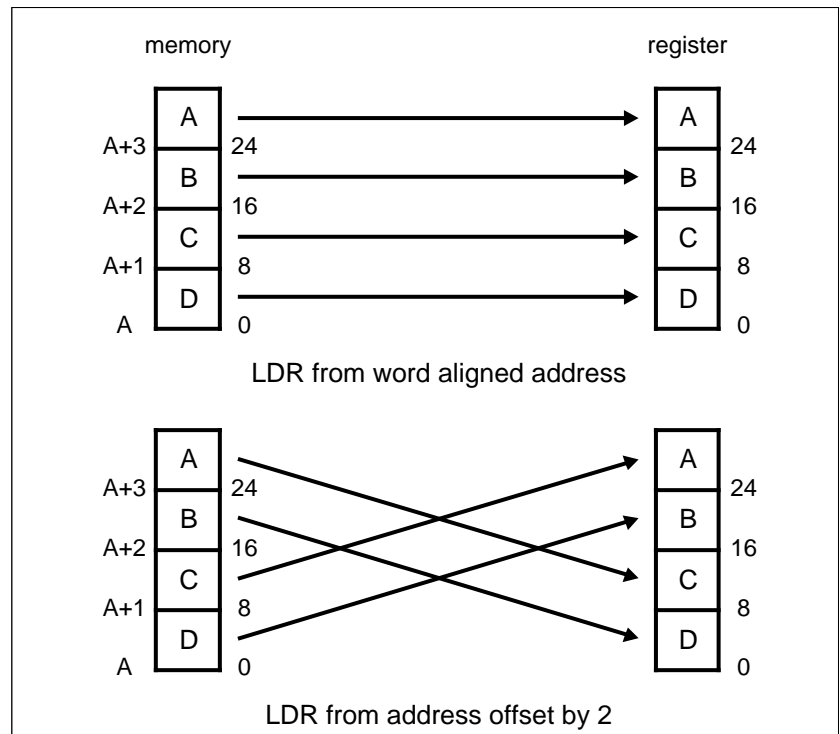


Figure 4-14: Little-endian offset addressing

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

Big-endian configuration

A byte load (LDRB) expects the data on data bus inputs 31 through 24 if the supplied address is on a word boundary, on data bus inputs 23 through 16 if it is a word address plus one byte, and so on. The selected byte is placed in the bottom 8 bits of the destination register and the remaining bits of the register are filled with zeros. Please see **Figure 3-2: Big-endian addresses of bytes within words** on page 3-3.

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

Instruction Set - LDR, STR

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

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

4.7.4 Use of R15

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

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

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

4.7.5 Restriction on the use of base register

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

For example:

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

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

4.7.6 Data aborts

A transfer to or from a legal address may cause problems for a memory management system. For instance, in a system which uses virtual memory the required data may be absent from main memory. The memory management unit or external hardware connected to the **ABORT** input can signal an abort, whereupon the Data Abort trap will be taken. It is up to the system software to resolve the cause of the problem, then the instruction can be restarted and the original program continued. The address that was accesses at the time of the abort is stored in the FAR, the cause of the abort is stored in the FSR and R14_abt stores the address of the instruction that caused the abort +8. For more detailed information about aborts, see [3.4.3 Abort](#) on page 3-8.

4.7.7 Instruction cycle times

Normal LDR instructions take 1 instruction fetch, 1 data read and 1 internal cycle and LDR PC take 3 instruction fetches, 1 data read and 1 internal cycle. For more information see [4.17 Instruction Speed Summary](#) on page 4-53.

STR instructions take 1 instruction fetch and 1 data write incremental cycles to execute.

4.7.8 Assembler syntax

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

LDR load from memory into a register

STR store from a register into memory

{cond} two-character condition mnemonic, see [Figure 4-2: Condition codes](#) on page 4-3

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

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

Rd an expression evaluating to a valid register number.

<Address> 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

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

- 3 A post-indexed addressing specification:

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

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

Rn, Rm expressions evaluating to a register number. If Rn is R15 then the assembler will subtract 8 from the offset value to allow for ARM710a macrocell pipelining. In this case base write-back shall not be specified.

Instruction Set - LDR, STR

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

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

4.7.9 Examples

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

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

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

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

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

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

4.8 Block Data Transfer (LDM, STM)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in **Figure 4-15: Block data transfer instructions** on page 4-27.

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

4.8.1 The register list

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

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

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

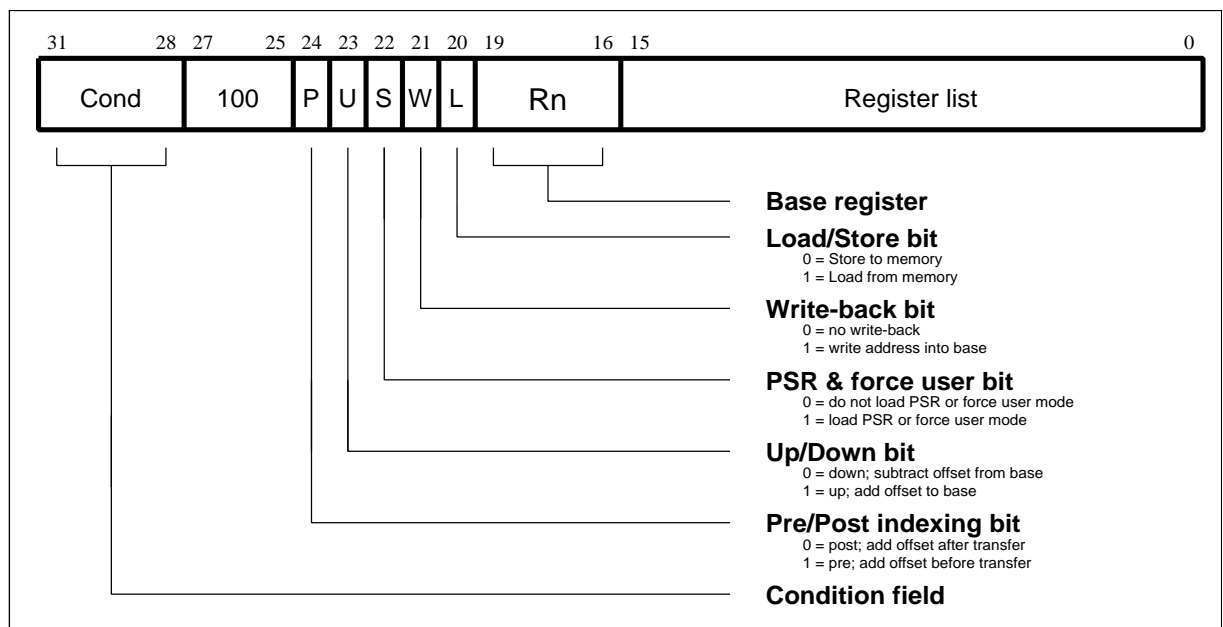


Figure 4-15: Block data transfer instructions

Instruction Set - LDM, STM

4.8.2 Addressing modes

The transfer addresses are determined by the contents of the base register (R_n), the pre/post bit (P) and the up/down bit (U). The registers are transferred in the order lowest to highest, so R_{15} (if in the list) will always be transferred last. The lowest register also gets transferred to/from the lowest memory address. By way of illustration, consider the transfer of R_1 , R_5 and R_7 in the case where $R_n=0x1000$ and write back of the modified base is required ($W=1$). *Figures 4-16, 4-17, 4-18 and 4-19* show the sequence of register transfers, the addresses used, and the value of R_n after the instruction has completed.

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

4.8.3 Address alignment

The address should normally be a word aligned quantity and non-word aligned addresses do not affect the instruction. The bottom 2 address bits are ignored by the LDM instruction. No rotating of data will occur for an LDM from a non-aligned address. If this is required then a series of LDRs should be used instead. However, the bottom 2 bits of the address will appear on **A[1:0]** and might be interpreted by the memory system.

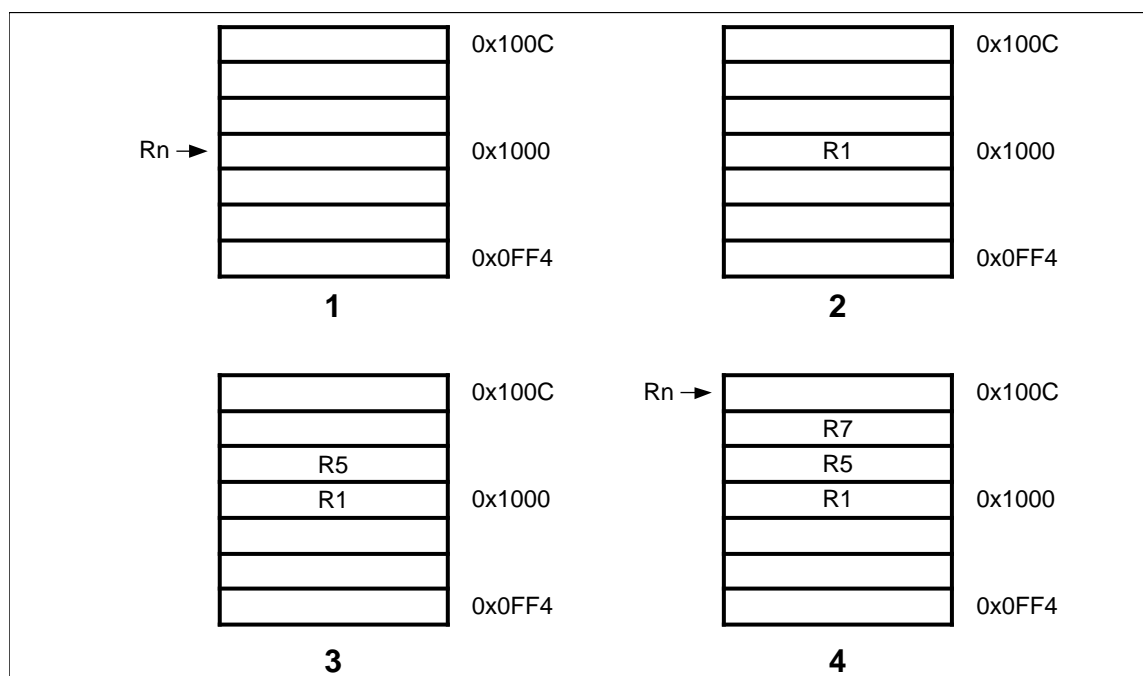


Figure 4-16: Post-increment addressing

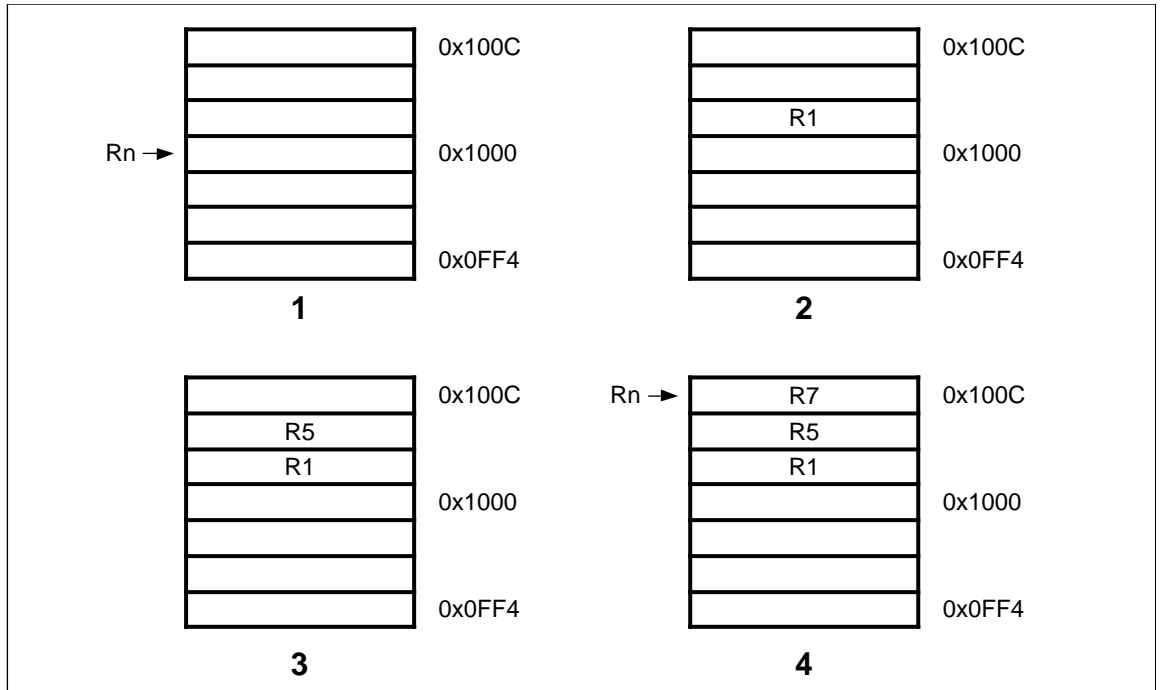


Figure 4-17: Pre-increment addressing

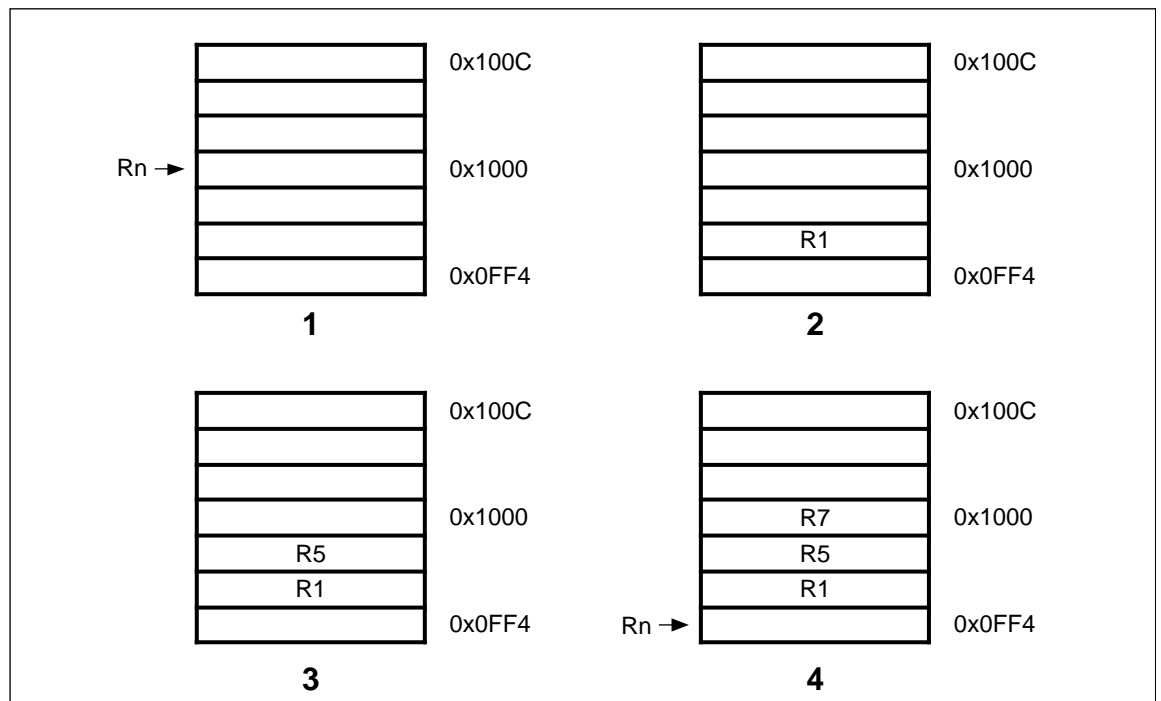


Figure 4-18: Post-decrement addressing

Instruction Set - LDM, STM

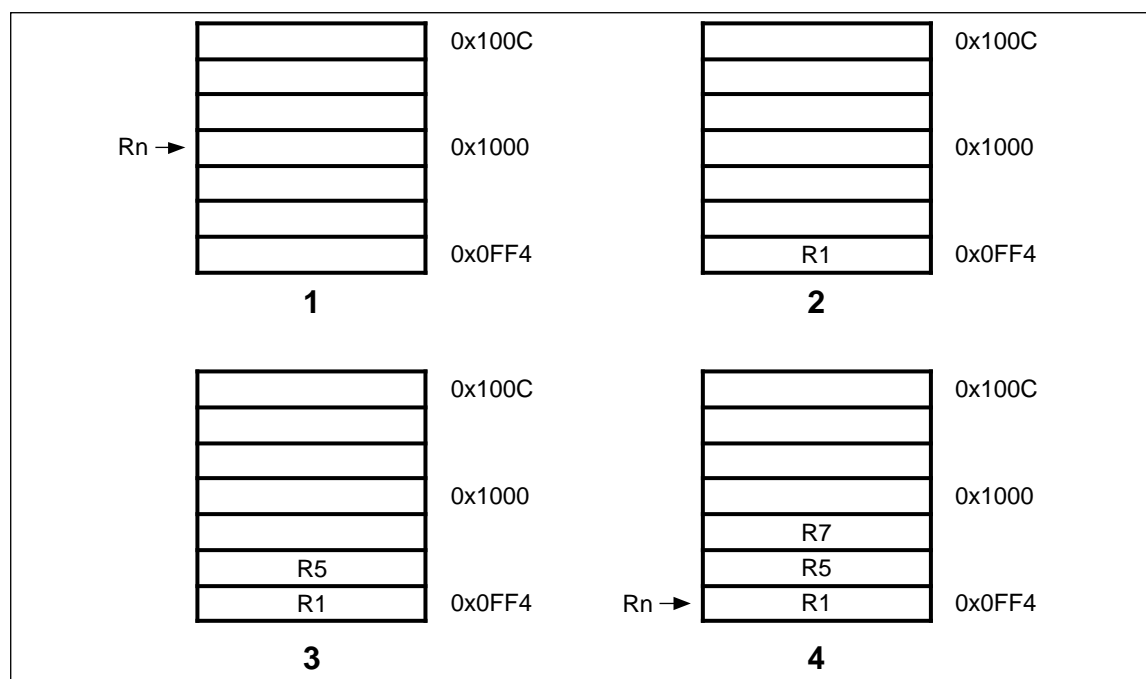


Figure 4-19: Pre-decrement addressing

4.8.4 Use of the S bit

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

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

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

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

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

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

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

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

4.8.5 Use of R15 as the base

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

4.8.6 Inclusion of the base in the register list

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

4.8.7 Data aborts

Some legal addresses may be unacceptable to a memory management system. This can happen on any transfer during a multiple register store or load, and must be recoverable if ARM710a macrocell is to be used in a virtual memory system. The memory management unit or external hardware connected to the **ABORT** input can signal an abort, whereupon the Data Abort trap will be taken. It is up to the system software to resolve the cause of the problem, then the instruction can be restarted and the original program continued. The address that was accessed at the time of the abort is stored in the FAR, the cause of the abort is stored in the FSR and r14_abt stores the address of the instruction that caused the abort +8. For more detailed information about aborts, see [3.4.3 Abort](#) on page 3-8

Aborts during STM instructions

If the abort occurs during a store multiple instruction, ARM710a macrocell takes little action until the instruction completes, whereupon it enters the data abort trap. The external memory controller is responsible for preventing erroneous writes to the memory if external hardware has generated the abort. The only change to the internal state of the processor will be the modification of the base register if write-back was specified, and this must be reversed by software (and the cause of the abort resolved) before the instruction may be retried.

Aborts during LDM instructions

When ARM710a macrocell detects a data abort during a load multiple instruction, it modifies the operation of the instruction to ensure that recovery is possible.

- 1 Overwriting of registers stops when the abort happens. The aborting load will not take place but earlier ones may have overwritten registers. The PC is always the last register to be written and so will always be preserved. In the case of MMU generated aborts, no registers will be overwritten and the abort occurs on the first word. The only exception to this is LDMs across platform boundaries when the abort may occur on the first word in the new section or page.

Instruction Set - LDM, STM

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

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

4.8.8 Instruction cycle times

Normal LDM instructions take 1 instruction fetch, n data reads and 1 internal cycle and LDM PC takes 3 instruction fetches, n data reads and 1 internal cycle. For more information see [4.17 Instruction Speed Summary](#) on page 4-53.

STM instructions take 1 instruction fetch, n data reads and 1 internal cycle to execute.

n is the number of words transferred.

4.8.9 Assembler syntax

<LDM STM>{cond}<FD ED FA EA IA IB DA DB> Rn{!},<Rlist>{^}	
{cond}	two character condition mnemonic, see Figure 4-2: Condition codes on page 4-3
Rn	an expression evaluating to a valid register number
<Rlist>	a list of registers and register ranges enclosed in {} (eg {R0,R2-R7,R10}).
{!}	if present requests write-back (W=1), otherwise W=0
{^}	if present set S bit to load the CPSR along with the PC, or force transfer of user bank when in privileged mode

Addressing mode names

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



name	stack	other	L bit	P bit	U bit
pre-increment load	LDMED	LDMIB	1	1	1
post-increment load	LDMFD	LDMIA	1	0	1
pre-decrement load	LDMEA	LDMDB	1	1	0
post-decrement load	LDMFA	LMDMA	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 4-2: Addressing mode names

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

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

4.8.10 Examples

```

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

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

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

```

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

```

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

BL    somewhere      ; this nested call will overwrite R14

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

```

Instruction Set - SWP

4.9 Single Data Swap (SWP)

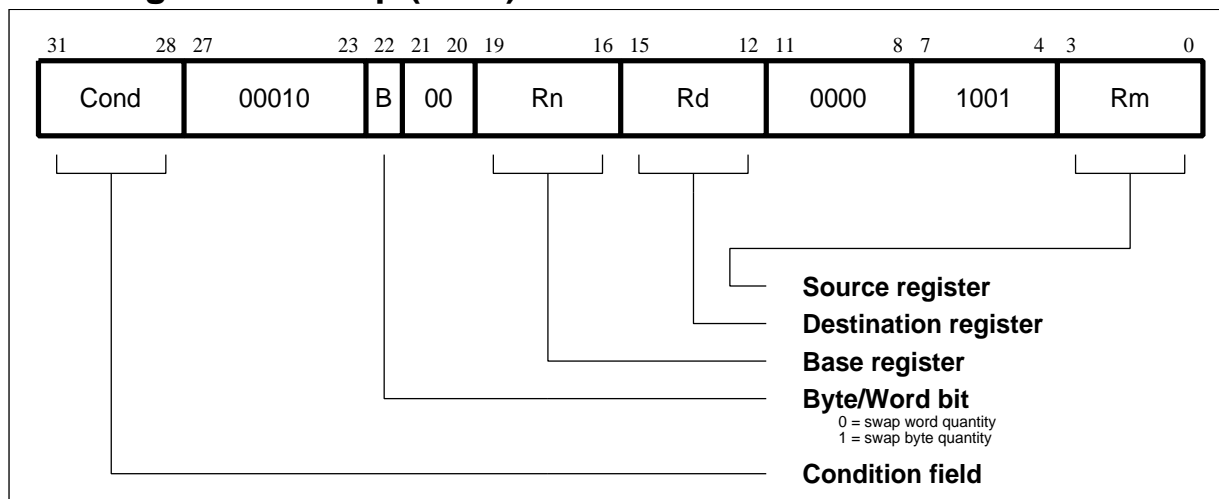


Figure 4-20: Swap instruction

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in [Figure 4-20: Swap instruction](#).

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

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

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

4.9.1 Bytes and words

This instruction class may be used to swap a byte (B=1) or a word (B=0) between an ARM710a macrocell register and memory. The SWP instruction is implemented as a LDR followed by a STR and the action of these is as described in the section on single data transfers. In particular, the description of big and little-endian configuration applies to the SWP instruction.

4.9.2 Use of R15

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

4.9.3 Data aborts

The memory management unit or external hardware connected to the **ABORT** input can signal an abort, whereupon the Data Abort trap will be taken. It is up to the system software to resolve the cause of the problem, then the instruction can be restarted and the original program continued. The address that was accessed at the time of the abort is stored in the FAR, the cause of the abort is stored in the FSR and R14_abt stores the address of the instruction that caused the abort +8. For more detailed information about aborts, see [3.4.3 Abort](#) on page 3-8

4.9.4 Instruction cycle times

Swap instructions take 1 instruction fetch, 1 data read, 1 data write and 1 internal cycle. For more information see [4.17 Instruction Speed Summary](#) on page 4-53.

4.9.5 Assembler syntax

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

{cond} - two-character condition mnemonic, see [Figure 4-2: Condition codes](#) on page 4-3

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

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

4.9.6 Examples

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

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

SWPEQ  R0,R0,[R1]    ; conditionally swap the contents of the
                    ; Software interrupt (SWI)
```

Instruction Set - SWI

4.10 Software Interrupt (SWI)

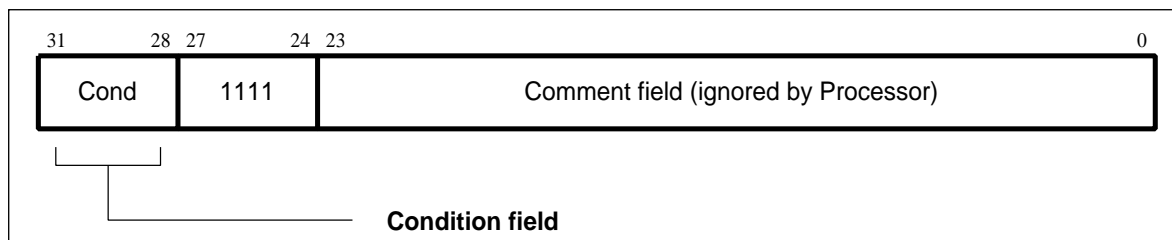


Figure 4-21: Software interrupt instruction

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in [Figure 4-21: Software interrupt instruction](#) on page 4-36.

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

4.10.1 Return from the supervisor

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

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

4.10.2 Comment field

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

4.10.3 Instruction cycle times

Software interrupt instructions take 3 instruction fetches. For more information see [4.17 Instruction Speed Summary](#) on page 4-53.

4.10.4 Assembler syntax

SWI{cond} <expression>

{cond} two character condition mnemonic, see [Figure 4-2: Condition codes](#) on page 4-3

<expression> is evaluated and placed in the comment field (which is ignored by ARM710a macrocell).

4.10.5 Examples

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

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

```
0x08 B Supervisor    ; SWI entry point
```

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

```
Zero    EQU    0
ReadC   EQU    256
WriteI  EQU    512
```

Supervisor

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

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

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

Instruction Set - SWI

4.11 Coprocessor Instructions on ARM710a macrocell

The ARM710a macrocell, unlike some other ARM processors, does not have an external coprocessor interface. The ARM710a macrocell only supports a single on chip coprocessor, #15, which is used to program the on-chip control registers. This only supports the Coprocessor Register Transfer instructions (MRC and MCR).

All other coprocessor instructions will cause the ARM710a macrocell to take the undefined instruction trap. These coprocessor instructions can be emulated in software by the undefined trap handler. Even though external coprocessors cannot be connected to ARM710a macrocell, the coprocessor instructions are still described here in full for completeness. Any external coprocessor referred to will be a software emulation.

4.12 Coprocessor Data Operations (CDP)

Use of the CDP instruction on the ARM710a macrocell will cause an undefined instruction trap to be taken, which may be used to emulate the coprocessor instruction.

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in **Figure 4-22: 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 ARM710a macrocell, and it will not wait for the operation to complete. The coprocessor could contain a queue of such instructions awaiting execution, and their execution can overlap other activity, allowing the coprocessor and the ARM710a macrocell to perform independent tasks in parallel.

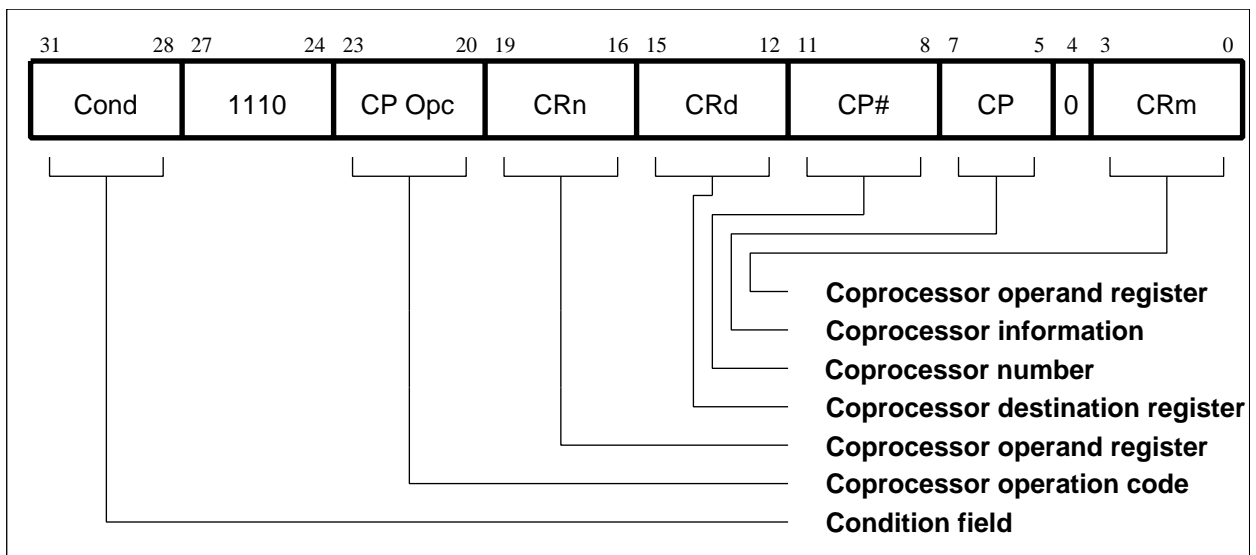


Figure 4-22: Coprocessor data operation instruction

4.12.1 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 will ignore any instruction which does not contain its number in the CP# field.

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

Instruction Set - CDP

4.12.2 Instruction cycle times

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

4.12.3 Assembler syntax

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

4.12.4 Examples

```

CDP    p1,10,c1,c2,c3 ; request coproc 1 to do operation 10
                        ; on CR2 and CR3, and put the result in CR1
CDPEQ  p2,5,c1,c2,c3,2 ; if Z flag is set request coproc 2 to do
                        ; operation 5 (type 2) on CR2 and CR3,
                        ; and put the result in CR1

```



4.13 Coprocessor Data Transfers (LDC, STC)

Use of the LDC or STC instruction on the ARM710a macrocell will cause an undefined instruction trap to be taken, which may be used to emulate the coprocessor instruction.

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in **Figure 4-23: Coprocessor data transfer instructions** on page 4-41.

This class of instruction is used to load (LDC) or store (STC) a subset of a coprocessor'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.

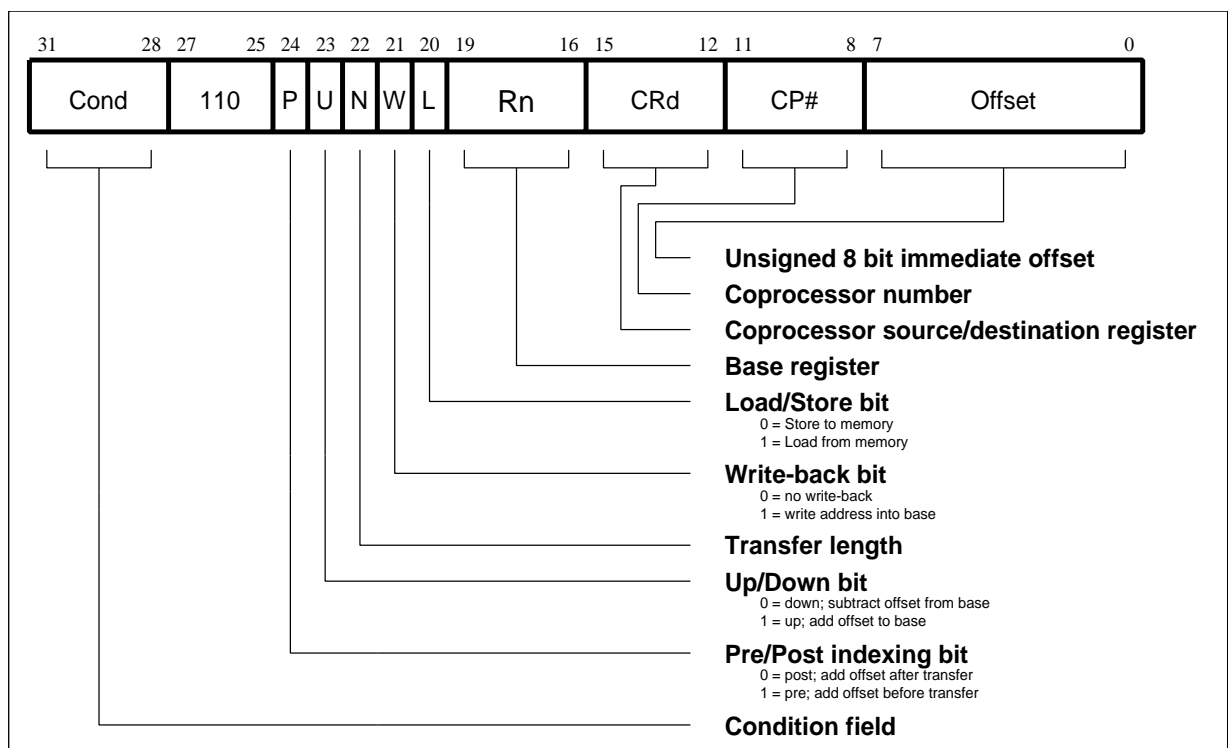


Figure 4-23: Coprocessor data transfer instructions

Instruction Set - LDC, STC

4.13.1 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.

4.13.2 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.

4.13.3 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.

4.13.4 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.

4.13.5 Data aborts

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

4.13.6 Instruction cycle times

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

4.13.7 Assembler syntax

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

LDC load from memory to coprocessor

STC store from coprocessor to memory

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

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

p# the unique number of the required coprocessor

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

<Address> 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

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

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

Instruction Set - LDC, STC

4.13.8 Examples

```
LDC    p1,c2,table    ; load c2 of coproc 1 from address table,
                      ; using a PC relative address.
STCEQLp2,c3,[R5,#24]!; conditionally store c3 of coproc 2 into
                      ; an address 24 bytes up from R5, write this
                      ; address back to R5, and use long transfer
                      ; option (probably to store multiple words)
```

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

4.14 Coprocessor Register Transfers (MRC, MCR)

Use of the MRC or MCR instruction on the ARM710a macrocell to a coprocessor other than number 15 will cause an undefined instruction trap to be taken, which may be used to emulate the coprocessor instruction.

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in **Figure 4-24: Coprocessor register transfer instructions** on page 4-45.

This class of instruction is used to communicate information directly between ARM710a macrocell 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 ARM710a macrocella 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.

Note the ARM710a macrocell has an internal coprocessor (#15) for control of on-chip functions. Accesses to this coprocessor are performed by coprocessor register transfers.

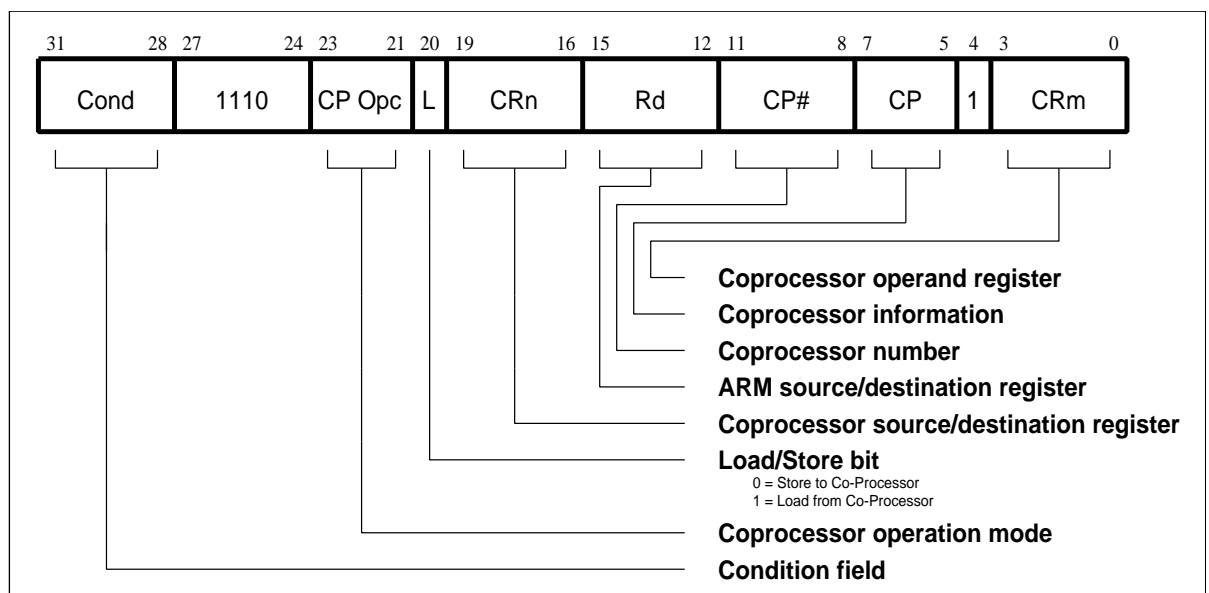


Figure 4-24: Coprocessor register transfer instructions

Instruction Set - MRC, MCR

4.14.1 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 depending on the particular operation specified.

4.14.2 Transfers to R15

When a coprocessor register transfer to ARM710a macrocell 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.


4.14.3 Transfers from R15

A coprocessor register transfer from ARM710a macrocell with R15 as the source register will store the PC+12.

4.14.4 Instruction cycle times

Access to the internal configuration register takes 1 instruction fetch cycle and 3 internal cycles. All other MRC instructions default to software emulation, and the number of cycles taken will depend on the coprocessor support software.

4.14.5 Assembler syntax

<MCR MRC>{<cond> p#,<expression1>,Rd,cn,cm{,<expression2>}}	
MRC	move from coprocessor to ARM710a macrocell register (L=1)
MCR	move from ARM710a macrocell 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	an expression evaluating to a valid ARM710a macrocell register number
cn and cm	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



4.14.6 Examples

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

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

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

Instruction Set - Undefined

4.15 Undefined instruction

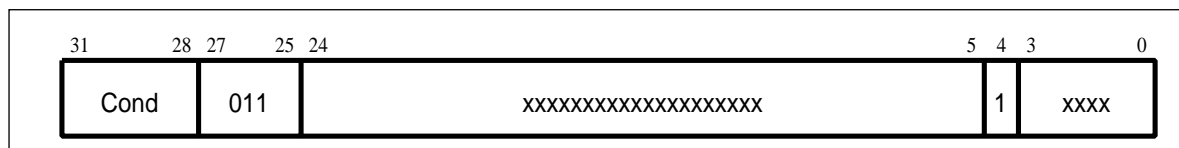


Figure 4-25: Undefined instruction

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction format is shown in *Figure 4-25: Undefined instruction*.

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

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

4.15.1 Assembler syntax

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

4.16 Instruction Set Examples

The following examples show ways in which the basic ARM710a macrocell 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.

4.16.1 Using the conditional instructions

1 using conditionals for logical OR

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

can be replaced by

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

2 absolute value

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

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

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

4 combining discrete and range tests

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

5 division and remainder

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

```

                                     ; enter with numbers in Ra and Rb
                                     ;
Div1  MOV   Rcnt,#1                 ; bit to control the division
      CMP   Rb,#0x80000000         ; move Rb until greater than Ra
      CMPCC Rb,Ra
      MOVCC Rb,Rb,ASL#1
      MOVCC Rcnt,Rcnt,ASL#1
      BCC   Div1
      MOV   Rc,#0
Div2  CMP   Ra,Rb                  ; test for possible subtraction
      SUBCS Ra,Ra,Rb              ; subtract if ok
```

Instruction Set - Examples

```

ADDCS Rc,Rc,Rcnt      ; put relevant bit into result
MOVS  Rcnt,Rcnt,LSR#1 ; shift control bit
MOVNE Rb,Rb,LSR#1     ; halve unless finished
BNE   Div2
                        ;
                        ; divide result in Rc
                        ; remainder in Ra

```

4.16.2 Pseudo random binary sequence generator

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

```

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

```

4.16.3 Multiplication by constant using the barrel shifter

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


```
MOV  Ra, Rb, LSL #n
```
- 2 Multiplication by 2^{n+1} (3,5,9,17..)


```
ADD  Ra,Ra,Ra,LSL #n
```
- 3 Multiplication by 2^{n-1} (3,7,15..)


```
RSB  Ra,Ra,Ra,LSL #n
```
- 4 Multiplication by 6


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

5 Multiply by 10 and add in extra number

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

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

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

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

b) If $C \bmod 4 = 1$, 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 $C \bmod 4 = 3$, 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
```

4.16.4 Loading a word from an unknown alignment

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

Instruction Set - Examples

4.16.5 Loading a halfword (little-endian)

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

4.16.6 Loading a halfword (big-endian)

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


4.17 Instruction Speed Summary

Due to the pipelined architecture of the CPU, instructions overlap considerably. In a typical cycle one instruction may be using the data path while the next is being decoded and the one after that is being fetched. For this reason the following table presents the incremental number of cycles required by an instruction, rather than the total number of cycles for which the instruction uses part of the processor. Elapsed time (in cycles) for a routine may be calculated from these figures which are shown in **Table 4-3: ARM Instruction speed summary** on page 4-53. These figures assume that the instruction is actually executed. Unexecuted instructions take one instruction fetch cycle.

Instruction	Cycle count
Data Processing - normal	1 instruction fetch
with register specified shift	1 instruction fetch and 1 internal cycle
with PC written	3 instruction fetches
with register specified shift & PC written	3 instruction fetches and 1 internal cycle
MSR, MRS	1 instruction fetch
LDR - normal	1 instruction fetch, 1 data read and 1 internal cycle
if the destination is the PC	3 instruction fetches, 1 data read and 1 internal cycle
STR	1 instruction fetch and 1 data write
LDM - normal	1 instruction fetch, n data reads and 1 internal cycle
if the destination is the PC	3 instruction fetches, n data reads and 1 internal cycle
STM	1 instruction fetch and n data writes
SWP	1 instruction fetch, 1 data read, 1 data write and 1 internal cycle
B,BL	3 instruction fetches
SWI, trap	3 instruction fetches
MUL,MLA	1 instruction fetch and m internal cycles
CDP	the undefined instruction trap will be taken
LDC	the undefined instruction trap will be taken
STC	the undefined instruction trap will be taken
MCR	1 instruction fetch and 3 internal cycles for coproc 15
MRC	1 instruction fetch and 3 internal cycles for coproc 15

Table 4-3: ARM Instruction speed summary

Instruction Set - Examples

Where:

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

The time taken for:

- an internal cycle - will always be one **FCLK** cycle
- an instruction fetch and data read - will be **FCLK** if a cache hit occurs, otherwise a full memory access is performed.
- a data write - will be **FCLK** if the write buffer (if enabled) has available space, otherwise the write will be delayed until the write buffer has free space. If the write buffer is not enabled a full memory access is always performed.
- Co-processor cycles - all coprocessor operations except MCR or MRC to registers 0 to 7 on coprocessor #15 (used for internal control) will cause the undefined instruction trap to be taken.
- memory accesses - can be found in the Bus Interface section.

Due to the presence of the cache and MMU, it is not possible to predict exactly the number of cycles required for the execution of a piece of code.

An access may hit or miss the cache and a cache miss to a cacheable area will cause a linefetch. An MMU translation table walk may also be required. These will increase the number of cycles taken by a section of code.

5

Configuration

This chapter describes the configuration.

5.1	Internal Coprocessor Instructions	5-2
5.2	Registers	5-3

Configuration

The operation and configuration of ARM710a macrocell 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.

To ensure backwards compatibility of future CPUs, all reserved or unused bits in registers and coprocessor instructions should be programmed to '0'. Invalid registers must not be read/written. The following bits shall be programmed to '0':

Register 1 bits[31:11]

Register 2 bits[13:0]

Register 5 bits[31:0]

Register 6 bits[11:0]

Register 7 bits[31:0]

Note *The grey areas in the register and translation diagrams are reserved and should be programmed 0 for future compatibility.*

5.1 Internal Coprocessor Instructions

The on-chip registers may be read using MRC instructions and written using MCR instructions. These operations are only allowed in non-user modes and the undefined instruction trap will be taken if accesses are attempted in user mode.

The CP15 register map may change in later ARM processors. We strongly recommend you structure software such that any code accessing coprocessor 15 is contained in a single module. It can then be updated easily.

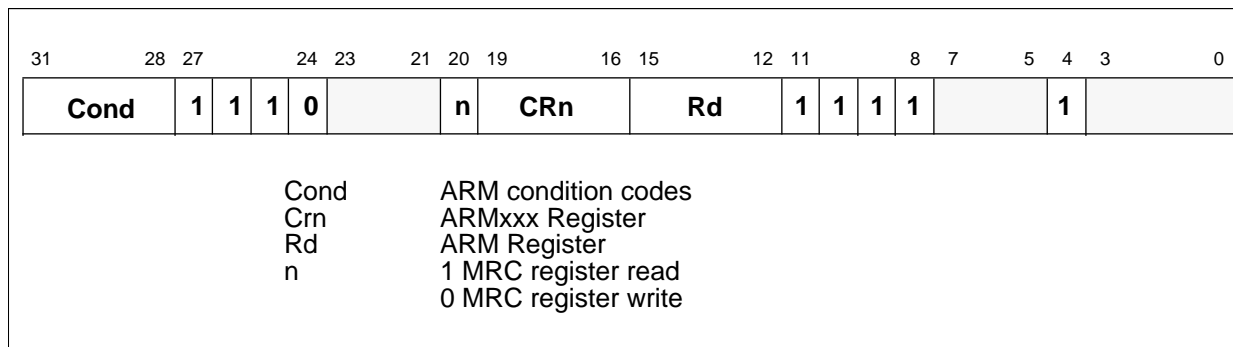


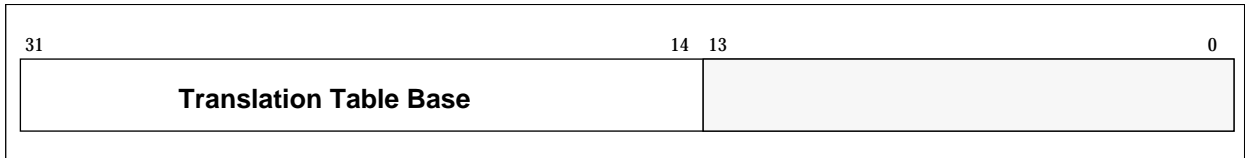
Figure 5-1: Format of internal coprocessor instructions MRC and MCR

Configuration

M Bit 0	Enable/disable 0 - on-chip Memory Management Unit turned off 1 - on-chip Memory Management Unit turned on.
A Bit 1	Address fault enable/disable 0 - alignment fault disabled 1 - alignment fault enabled
C Bit 2	Cache enable/disable 0 - Instruction / data cache turned off 1 - Instruction / data cache turned on
W Bit 3	Write buffer enable/disable 0 - Write buffer turned off 1 - Write buffer turned on
P Bit 4	ARM 32/26-bit program space 0 - 26 bit Program Space selected 1 - 32 bit Program Space selected
D Bit 5	ARM 32/26-bit data space 0 - 26 bit Data Space selected 1 - 32 bit Data Space selected
B Bit 7	Big/little -endian 0 - Little-endian operation 1 - Big-endian operation
S Bit 8	System This bit controls the ARM710a macrocell permission system. Refer to 9.6 Section Descriptor on page 9-7.
R Bit 9	ROM This bit controls the ARM710a macrocell permission system. Refer to 9.6 Section Descriptor on page 9-7.

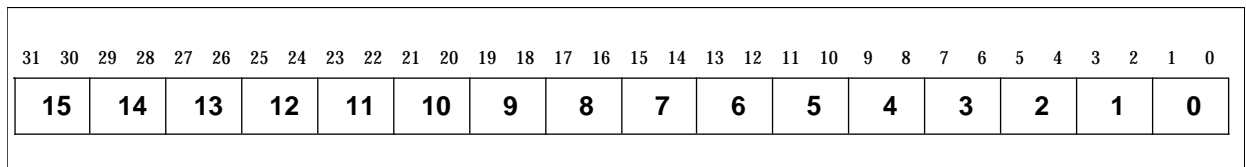
5.2.3 Register 2 Translation Table Base

Register 2 is a write-only register which holds the base of the currently active Level One page table.



5.2.4 Register 3 Domain Access Control

Register 3 is a write-only register which holds the current access control for domains 0 to 15. See [9.13 Domain Access Control](#) on page 9-14 for the access permission definitions and other details



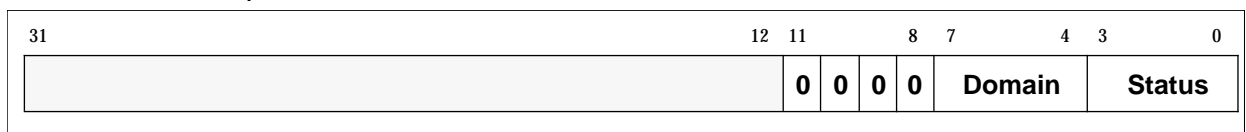
5.2.5 Register 4 Reserved

Register 4 is Reserved. Accessing this register has no effect, but should never be attempted.

5.2.6 Register 5

Read: fault status

Reading register 5 returns the status of the last data fault. It is not updated for a prefetch fault. See [Chapter 9](#), for more details. Note that only the bottom 12 bits are returned. The upper 20 bits will be the last value on the internal data bus, and therefore will have no meaning. Bits 11:8 are always returned as zero



Write: translation lookaside buffer flush

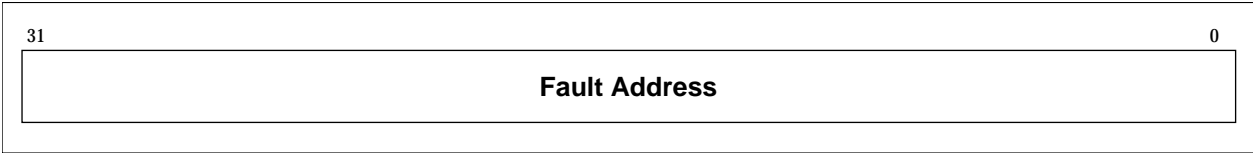
Writing Register 5 flushes the TLB. (The data written is discarded).

Configuration

5.2.7 Register 6

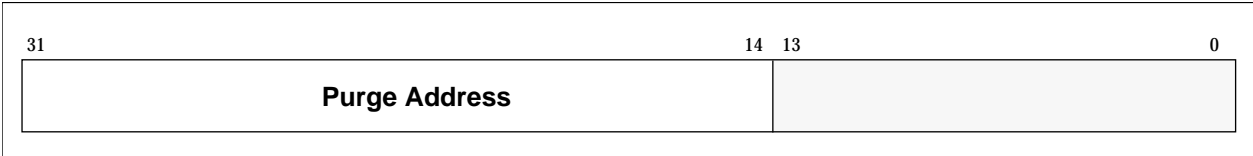
Read: fault address

Reading register 6 returns the virtual address of the last data fault.



Write: TLB purge

Writing Register 6 purges the TLB; the data is treated as an address and the TLB is searched for a corresponding page table descriptor. If a match is found, the corresponding entry is marked as invalid. This allows the page table descriptors in main memory to be updated and invalid entries in the on-chip TLB to be purged without requiring the entire TLB to be flushed



5.2.8 Register 7 IDC Flush

Register 7 is a write-only register. The data written to this register is discarded and the IDC is flushed.

5.2.9 Registers 8 - 15 Reserved

Accessing any of these registers will cause the undefined instruction trap to be taken.



6

Instruction and Data Cache (IDC)

This chapter describes Instruction and Data Cache.

6.1	Cacheable Bit	6-2
6.2	IDC Operation	6-2
6.3	IDC Validity	6-2
6.4	Read-lock-write	6-3
6.5	IDC Enable/Disable and Reset	6-3

Instruction and Data Cache (IDC)

ARM710a macrocell contains a 8kByte mixed instruction and data cache. The IDC has 512 lines of 16 bytes (4 words), arranged as a 4 way set associative cache, and uses the virtual addresses generated by the processor core. The IDC is always reloaded a line at a time (4 words). It may be enabled or disabled via the ARM710a macrocell Control Register and is disabled on **nRESET**. The operation of the cache is further controlled by the *Cacheable*, or C, bit stored in the Memory Management Page Table (see [Chapter 9, 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.

6.1 Cacheable 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 ARM710a macrocell'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.

6.2 IDC Operation

The C bit in the ARM710a macrocell Control Register and the *Cacheable* bit in the MMU page tables only affect loading data into the Cache. The Cache will always be searched regardless of these two bits, and if the data is found it will be used, so when the cache is disabled it should also be flushed.

6.2.1 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.

6.2.2 Uncacheable reads **C = 0**

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

6.3 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.

6.3.1 Software IDC flush

The entire IDC may be marked as invalid by writing to the ARM710a macrocell IDC Flush Register (Register 7). The cache will be flushed immediately the register is written, but note that the following two instruction fetches may come from the cache before the register is written.

6.3.2 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.

6.4 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.

6.5 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.

6.5.1 To enable the IDC

To enable the IDC, make sure that the MMU is enabled first by setting bit 0 in the 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.

6.5.2 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.

Instruction and Data Cache (IDC)

7

Write Buffer (WB)

This chapter describes the Write Buffer.

7.1	Bufferable Bit	7-2
7.2	Write Buffer Operation	7-2

Write Buffer (WB)

The ARM710a macrocell 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 ARM710a macrocell Control Register and the buffer is disabled and flushed on reset. The operation of the write buffer is further controlled by one bit, B, or Bufferable, which is 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 the B bit in the corresponding page table must be set.

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

7.1 Bufferable Bit

This bit controls whether a write operation may or may not use the write buffer. Typically main memory will be bufferable and I/O space unbufferable. The Bufferable bit can be configured for both pages and sections.

7.2 Write Buffer Operation

When the CPU performs a write operation, the translation entry for that address is inspected and the state of the B bit determines the subsequent action. If the write buffer is disabled via the ARM710a macrocell Control Register, bufferable writes are treated in the same way as unbuffered writes.

7.2.1 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.

7.2.2 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 write completes externally, which may require synchronisation and several external clock cycles.

7.2.3 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.*

7.2.4 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.

7.2.5 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.*

Write Buffer (WB)

8

Coprocessors

This chapter describes the coprocessors.

8.1 Coprocessors

8-2

Coprocessors

8.1 Coprocessors

ARM710a macrocell has no external coprocessor bus, so it is not possible to add external coprocessors to this device.

ARM710a macrocell still has an internal coprocessor designated #15 for internal control of the device. All coprocessor operations except MCR or MRC to registers 0 through 7 on coprocessor #15 will cause the undefined instruction trap to be taken.

This chapter describes the Memory Management Unit.

9.1	MMU Program Accessible Registers	9-3
9.2	Address Translation	9-4
9.3	Translation Process	9-5
9.4	Level One Descriptor	9-6
9.5	Page Table Descriptor	9-6
9.6	Section Descriptor	9-7
9.7	Translating Section References	9-8
9.8	Level Two Descriptor	9-9
9.9	Translating Small Page References	9-10
9.10	Translating Large Page References	9-11
9.11	MMU Faults and CPU Aborts	9-12
9.12	Fault Address & Fault Status Registers (FAR & FSR)	9-12
9.13	Domain Access Control	9-14
9.14	Fault Checking Sequence	9-15
9.15	External Aborts	9-17
9.16	Interaction of the MMU, IDC and Write Buffer	9-18
9.17	Effect of Reset	9-19

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, 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.

9.1 MMU Program Accessible Registers

The ARM710a macrocell Processor provides several 32-bit registers which determine the operation of the MMU. The format for these registers is shown in **Figure 9-1: MMU register summary** on page 9-3. A brief description of the registers is provided below. Each register will be discussed in more detail within the section that describes its use.

Data is written to and read from the MMU's registers using the ARM CPU's MRC and MCR coprocessor instructions.

The **Translation Table Base Register** holds 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 the sixteen Domains (D15-D0).

Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1 write	0	0	0	0	0	Control																0	R	S	B	1	D	P	W	C	A	M
2 write	Translation Table Base																															
3 write	Domain Access Control																															
	15		14		13		12		11		10		9		8		7		6		5		4		3		2		1		0	
5 read	Fault Status																0	0	0	0	Domain				Status							
5 write	Flush TLB																															
6 read	Fault Address																															
6 write	Purge Address																															

Figure 9-1: MMU register summary

Note The registers not shown are reserved and should not be used.

The **Fault Status Register** indicates the domain and type of access being attempted when an abort occurred. 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 different for internal and external faults (as indicated by bit 0 in the register) and is shown in **Table 9-4: Priority encoding of fault status** on page 9-13. A write to this register flushes the TLB.

The **Fault Address Register** holds the virtual address of the access which was attempted when a fault occurred. A write to this register causes the data written to be treated as an address and, if it is found in the TLB, the entry is marked as invalid. (This operation is known as a TLB purge). The Fault Status Register and Fault Address Register are only updated for data faults, not for prefetch faults.

Memory Management Unit

9.2 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.

9.3 Translation Process

9.3.1 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 9-2: Translation table base register*; the table must reside on a 16kB boundary.

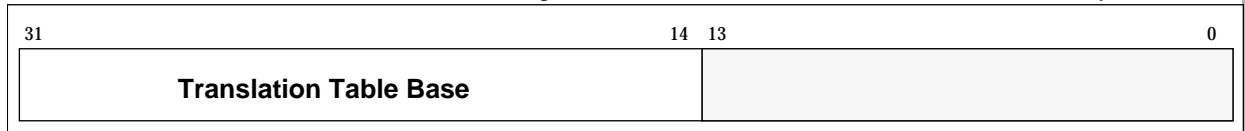


Figure 9-2: Translation table base register

9.3.2 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 9-3: 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)

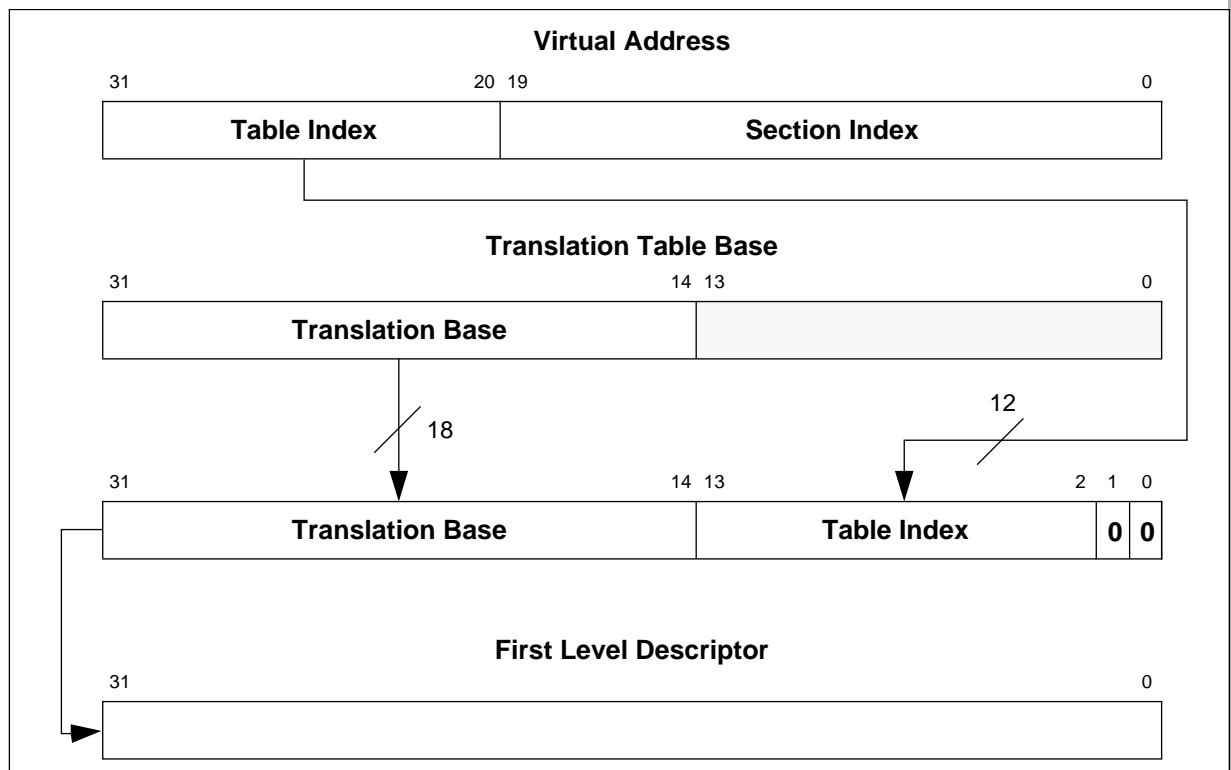


Figure 9-3: Accessing the translation table first level descriptors

Memory Management Unit

9.4 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.

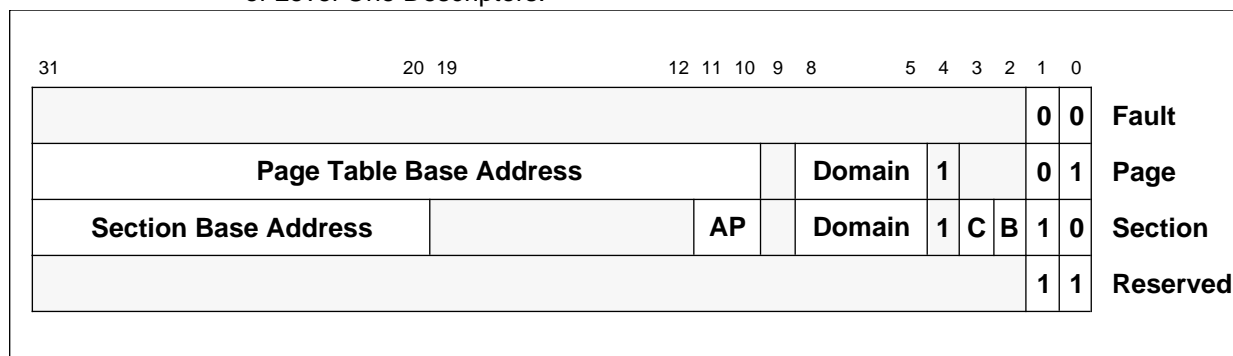


Figure 9-4: Level one descriptors

The two least significant bits indicate the descriptor type and validity, 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 9-1: Interpreting level one descriptor bits [1:0]

9.5 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 9-7: Small page translation](#) on page 9-10).

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

9.6 Section Descriptor

Bits 3:2 (C, & B) control the cache- and write-buffer-related functions as follows:

C - Cacheable: indicates that data at this address will be placed in the cache (if the cache is enabled).

B - Bufferable: indicates that data at this address will be written through the write buffer (if the write buffer is enabled).

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 9-2: Interpreting access permission \(AP\) Bits](#) on page 9-7. 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 Supervisor	User	Notes
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 9-2: 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.

Note *The meaning of the C and B bits may change in later ARM processors. We strongly recommend you structure software such that code which manipulates the MMU page tables is contained in a single module. It can then be updated easily when you port it to a different ARM processor.*

Memory Management Unit

9.7 Translating Section References

Figure 9-5: *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.

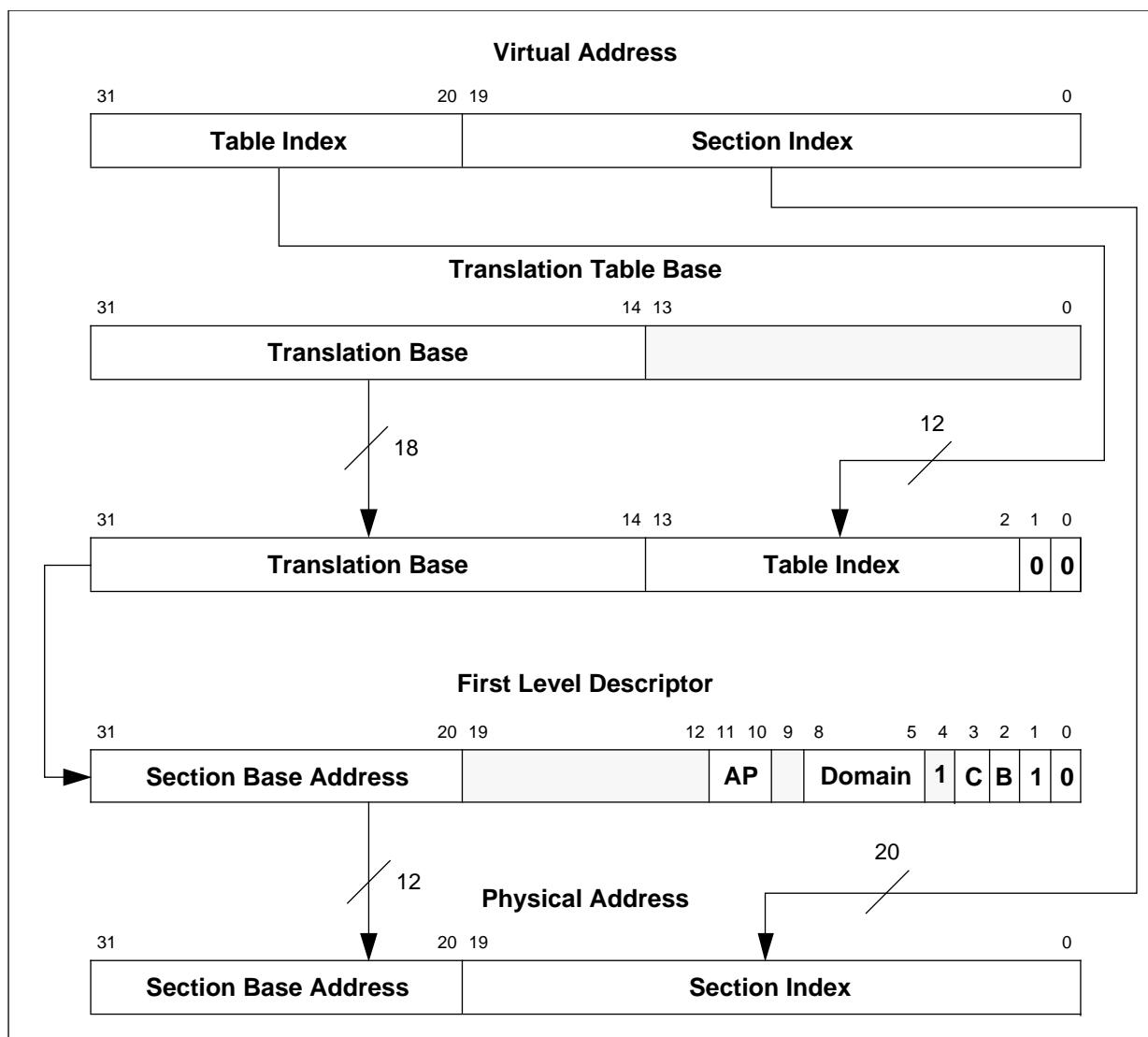


Figure 9-5: Section translation

9.8 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 9-7: Small page translation](#) on page 9-10, 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

31	20 19		16 15		12	11	10	9	8	7	6	5	4	3	2	1	0		
																	0	0	Fault
Large Page Base Address					ap3	ap2	ap1	ap0	C	B	0	1	Large Page						
Small Page Base Address					ap3	ap2	ap1	ap0	C	B	1	0	Small Page						
																	1	1	Reserved

Figure 9-6: 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 9-3: Interpreting page table entry bits 1:0

Bit 2 B - Bufferable: indicates that data at this address will be written through the write buffer (if the write buffer is enabled).

Bit 3 C - Cacheable: indicates that data at this address will be placed in the IDC (if the cache is enabled).

Bits 11:4 specify the access permissions (ap3 - ap0) for the four sub-pages and interpretation of these bits is described earlier in [Table 9-1: Interpreting level one descriptor bits \[1:0\]](#) on page 9-6.

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 9-7: Small page translation* on page 9-10 and  *Figure 9-8: Large page translation* on page 9-11).

Memory Management Unit

9.9 Translating Small Page References

Figure 9-7: *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).

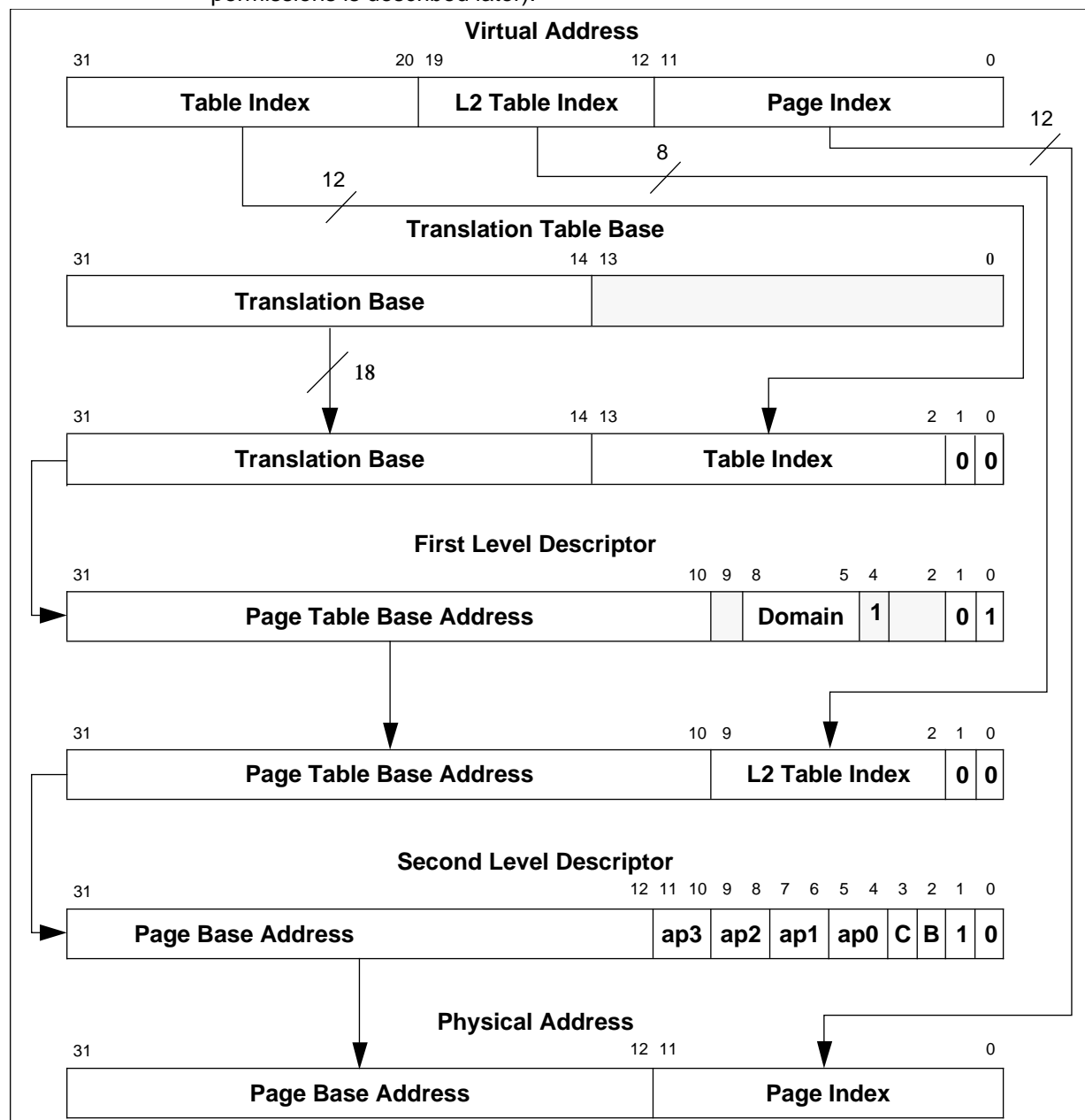


Figure 9-7: *Small page translation*

9.10 Translating Large Page References

Figure 9-8: Large page translation 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.

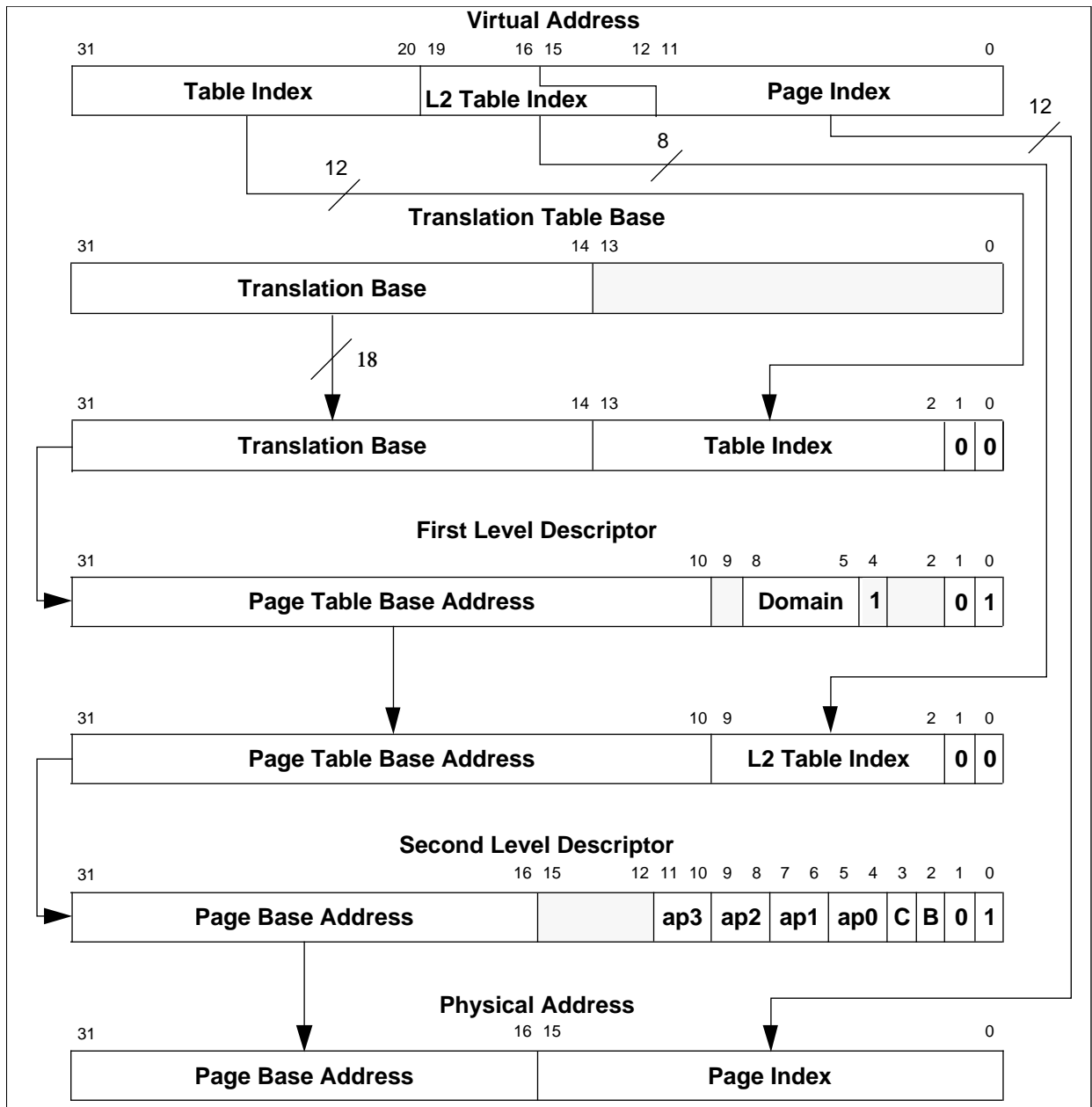


Figure 9-8: Large page translation

Memory Management Unit

9.11 MMU Faults and CPU Aborts

The MMU generates four types of faults:

- Alignment Fault
- Translation Fault
- Domain Fault
- Permission 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.

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.

If the ARM710a macrocell is operating in fastbus mode an internally aborting access may cause the address on the external address bus to change, even though the external bus cycle has been cancelled. The address that is placed on the bus will be the translation of the address that caused the abort, though in the case of the a Translation Fault the value of this address will be undefined. No memory access will be performed to this address.

9.12 Fault Address & Fault Status Registers (FAR & 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 which caused 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 9-4: Priority encoding of fault status** on page 9-13.

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.

Memory Management Unit

	Source		FS[32:10]	Domain[3:0]	FAR
Highest	Alignment		00x1	x	valid
	Bus Error (translation)	level1	1100	x	valid
		level2	1110	valid	valid
	Translation	Section	0101	Note 2	valid
		Page	0111	valid	valid
	Domain	Section	1001	valid	valid
Page		1011	valid	valid	
Lowest	Permission	Section	1101	valid	valid
		Page	1111	valid	valid
	Bus Error (linefetch)	Section	0100	valid	Note 3
		Page	0110	valid	Note 3
	Bus Error (other)	Section	1000	valid	valid
		Page	1010	valid	valid

Table 9-4: Priority encoding of fault status

x is undefined, and may read as 0 or 1

Notes

- 1 Any abort masked by the priority encoding may be regenerated by fixing the primary abort and restarting the instruction.
- 2 In fact this register will contain bits[8:5] of the Level 1 entry which are undefined, but would encode the domain in a valid entry.
- 3 The FAR will contain the address of the start of the linefetch.

Memory Management Unit

9.13 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 9-9: Domain access control register format** on page 9-14 illustrates how the 32 bits of the register are allocated to define the sixteen 2-bit domains.

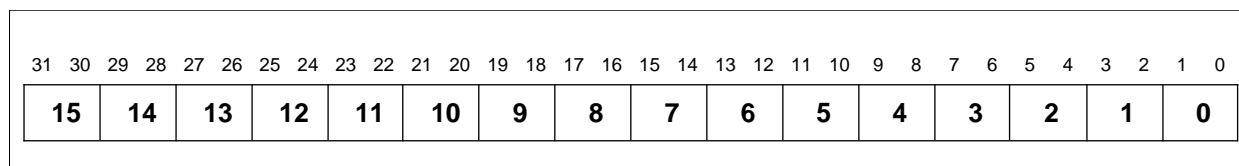


Figure 9-9: Domain access control register format

Table 9-5: 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 9-5: Interpreting access bits in domain access control register

9.14 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.

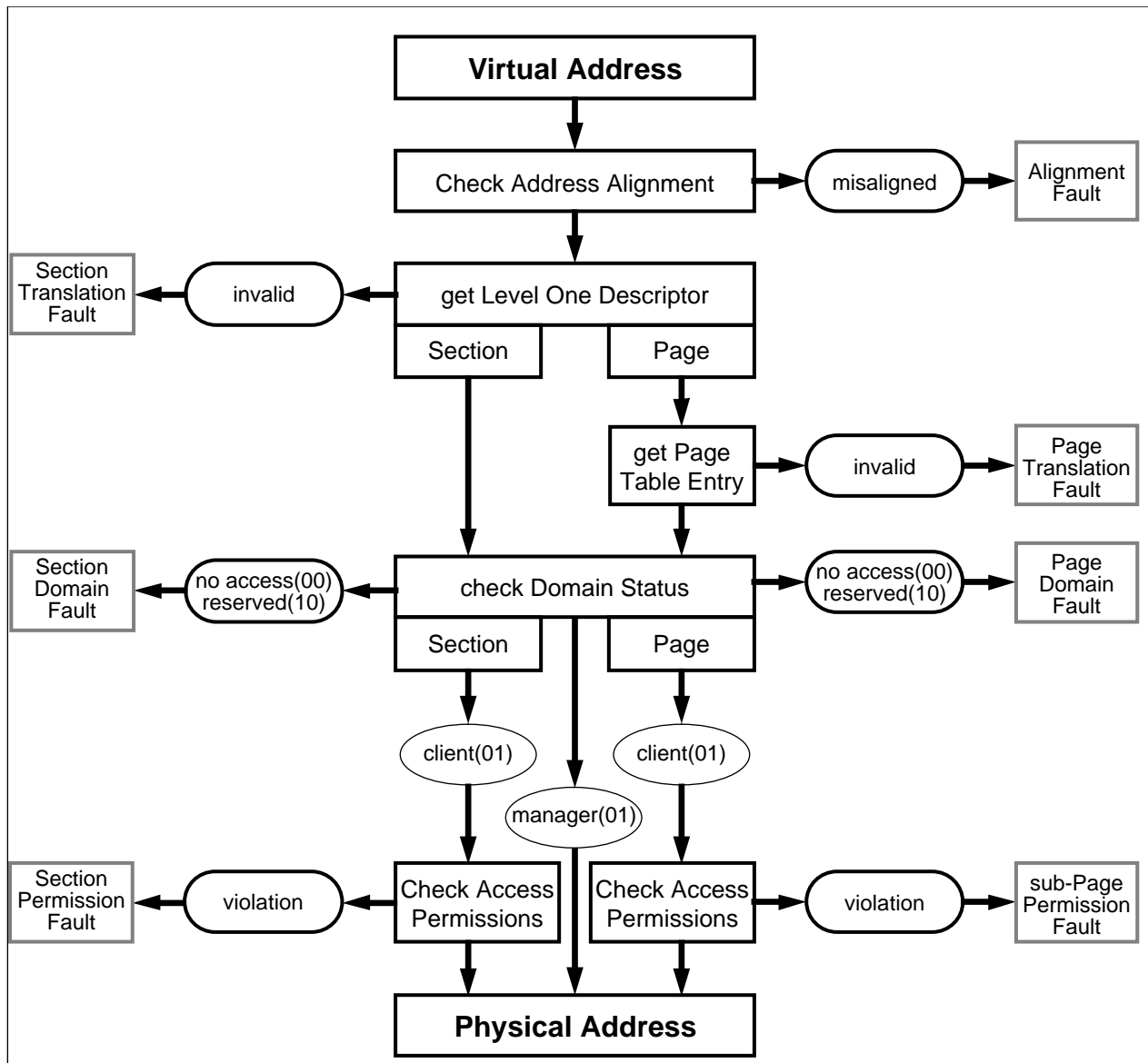


Figure 9-10: Sequence for checking faults

Memory Management Unit

9.14.1 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.

9.14.2 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.

9.14.3 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 9-2: Interpreting access permission (AP) Bits** on page 9-7. 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.

9.14.4 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 9-2: Interpreting access permission (AP) Bits** on page 9-7. 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 9-2: Interpreting access permission \(AP\) Bits](#) on page 9-7), the only difference being that the fault generated is a sub-page permission fault.

9.15 External Aborts

In addition to the MMU-generated aborts, ARM710a macrocell has an external abort signal 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 signal must be used with great care. The following section describes the restrictions.

The following accesses may be aborted and restarted safely. If any of the following are aborted the external access will cease on the next cycle. 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 will be purged, so it will not contain invalid data. If the abort happens on a word that has been requested by the ARM710a macrocell, it will be aborted, otherwise the cache line will be purged but program flow will *not* be interrupted. The line is therefore purged 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.

Note *Areas of memory which can generate an external abort on a location which has previously been read successfully must not be marked a cacheable or unbufferable. This applies to both the MMU page tables and the configuration register. If all writes to an area of memory abort, we recommend that you mark it as read only in the MMU, otherwise mark it as uncacheable and unbufferable.*

Memory Management Unit

9.16 Interaction of the MMU, IDC and Write Buffer

The MMU, IDC and WB may be enabled/disabled independently. However, in order for the write buffer or the cache to be enabled the MMU must also be enabled. 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 9-6: Valid MMU, IDC & 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 the two instructions following the enabling of the MMU will have 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 the WB by clearing bit 3 in the Control Register.
- 2 Disable the IDC by clearing bit 2 in the Control Register.
- 3 Disable the MMU by clearing bit 0 in the Control Register.

Note *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 may be done simultaneously.

9.17 Effect of Reset

See [3.5 Reset](#) on page 3-11.

Memory Management Unit

10

Bus Clocking

This chapter describes the bus interface clocking:

10.1	Fastbus Extension	10-2
10.2	Standard Mode	10-4

Bus Clocking

The ARM710a macrocell bus interface can be operated either using the standard mode of operation or using the new fastbus extension:

Standard mode

- backwards Compatible with ARM610
- two clocks, **FCLK** and **MCLK**
- synchronous or fully asynchronous operation

Fastbus extension

- single device clock
- enhanced **ALE** functionality to ease design
- increased maximum **MCLK** frequency

For new designs it is possible to operate the device using the fastbus extension. In this fastbus mode, the device is clocked off a single clock, and the bus is operated at the same frequency as the core. This will allow the bus interface to be clocked faster than if the device is operated in standard mode. It is recommended that this mode of operation be used in systems with high speed memory, and a single clock.

For designs using low cost, low speed memory, and wishing to operate the core at a faster speed it is recommended that standard mode is used.

As the ARM710a macrocell is a fully static design the clock can be stopped indefinitely in either mode of operation. Care should be taken though to ensure that the memory system will not dissipate power in the state in which it is stopped.

10.1 Fastbus Extension

Using the fastbus extension, the ARM710a macrocell has a single input clock, **MCLK**. It is used to clock the internals of the device, and qualified by **nWAIT**, controls the memory interface:

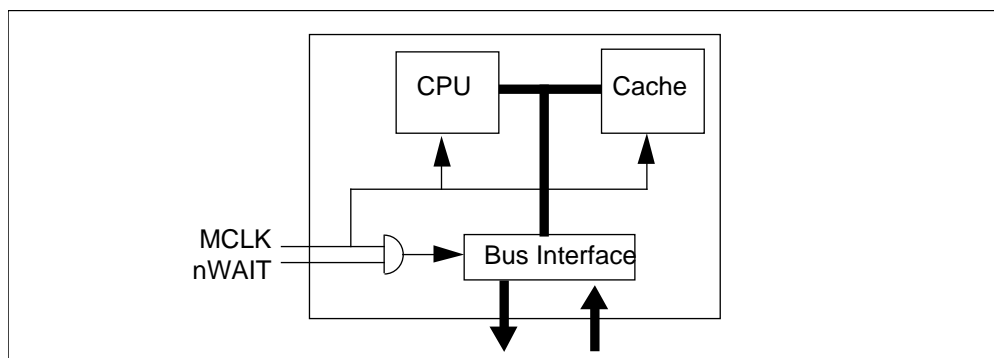


Figure 10-1: Conceptual device clocking using the fastbus extension

The signal **nWAIT** is used to insert entire **MCLK** cycles into the bus cycle timing. **nWAIT** may only change when **MCLK** is LOW, and extends the memory access by inserting **MCLK** cycles into the access whilst **nWAIT** is asserted. [◀11.9 Use of the nWAIT pin](#) on page 11-13 describes the use of **nWAIT** in detail.

It is preferable to use **nWAIT** to extend memory cycles, rather than stretching **MCLK** externally to the device because it is possible for the core to be accessing the Cache while bus activity is occurring. This allows the maximum performance, as the Core can continue execution in parallel with the memory bus activity. All **MCLK** cycles are available to the CPU and Cache, regardless of the state of **nWAIT**.

In some circumstances, it may be desirable to stretch **MCLK** phases in order to match memory timing which is not an integer multiple of **MCLK** cycles. There are certain cases when this results in a higher performance than using **nWAIT** to extend the access by an integer number of cycles. CPU and Cache operation can only continue in parallel with buffered writes to the external bus. For all external read accesses the CPU will be stalled until the bus activity has completed. So if read accesses can be achieved faster by stretching **MCLK** rather than using **nWAIT**, this will result in improved performance. An example of where this may be useful would be to interface to a ROM which has a cycle time of 2.5 times the **MCLK** period.

When operating the device with **FASTBUS** HIGH, the input **FCLK** and **SnA** are not used. To prevent unwanted power dissipation ensure that they do not float to an undefined level. New designs should tie these signals LOW for compatibility with future products.

Operating using the fastbus extension changes the operation of the **ALE** input. Instead of directly controlling the address latches, it is used to select between conventional address timing, (**ALE** HIGH) and delayed address timing (**ALE** LOW). This is described in [◀11.10 Use of the ALE Pin](#) on page 11-14.

If using the device in fastbus mode (**FASTBUS** HIGH) use the AC parameters as given in [◀Chapter 14, AC Parameters with Fastbus Extension](#).

Bus Clocking

10.2 Standard Mode

Using the standard mode of operation (without the fastbus extension), **FASTBUS** tied LOW, the ARM710a macrocell has two input clocks **FCLK** and **MCLK**.

The bus interface is always controlled by the memory clock, **MCLK**, qualified by **nWAIT**. However the core and cache will be clocked by the fast clock, **FCLK**.

In standard mode the **FCLK** frequency must be greater than or equal to the **MCLK** frequency at all times. This relationship must be maintained on a cycle by cycle basis.

When running in this mode, memory access cycles can be stretched either by using **nWAIT**, or by stretching phases of **MCLK**. The resulting performance will be determined by the access time, regardless of which method is used.

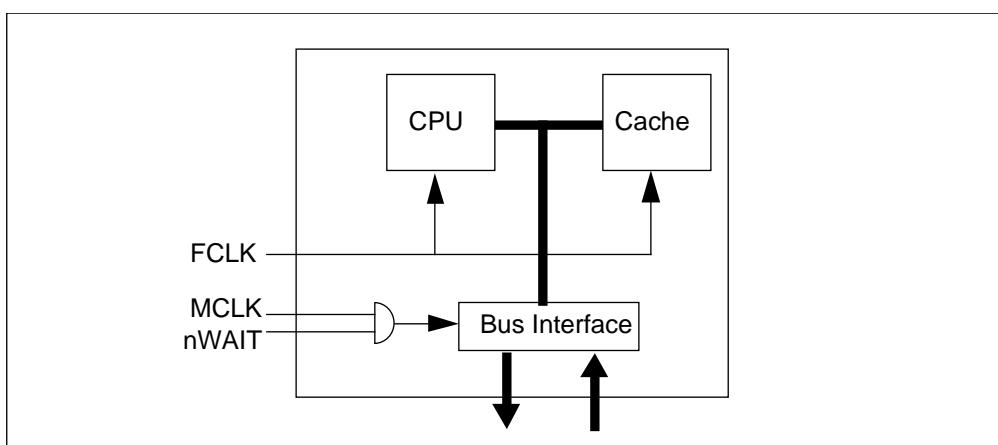


Figure 10-2: Conceptual device clocking in standard mode

When using standard mode, the ARM710a macrocell bus interface has two distinct modes of operation: synchronous and asynchronous, which are selected by tying **SnA** either HIGH or LOW. The two modes differ in the relationship between **FCLK** and **MCLK**:

- in asynchronous mode (**SnA** LOW) the clocks may be completely asynchronous and of unrelated frequency
- in synchronous mode (**SnA** HIGH) **MCLK** may only make transitions before the falling edge of **FCLK**.

In systems where a satisfactory relationship exists between **FCLK** and **MCLK**, synchronization penalties can be avoided by selecting the synchronous mode of operation.

If using the device in standard mode (**FASTBUS** LOW) please use the AC parameters as given in [Chapter 13, AC Parameters in Standard Mode](#).

10.2.1 Asynchronous mode

In this mode **FCLK** and **MCLK** may be completely asynchronous. This mode should be selected, by tying **SnA** LOW, when the two clocks are of unrelated frequency. There is a synchronisation penalty whenever the internal core clock switches between the two input clocks. This penalty is symmetric, and varies between nothing and a whole period of the clock to which the core is resynchronising so when changing from **FCLK** to **MCLK**, the average resynchronisation penalty is half an **MCLK** period, and similarly when changing from **MCLK** to **FCLK**, it is half an **FCLK** period.

10.2.2 Synchronous mode

In this mode, selected by tying **SnA** HIGH, there is a tightly defined relationship between **FCLK** and **MCLK**. **MCLK** may only make transitions on the falling edge of **FCLK**. Some jitter between the two clocks is permitted, but **MCLK** must meet the setup and hold requirements relative to **FCLK**. See [Figure 11-12: Sampling points at maximum frequency](#) on page 11-18.

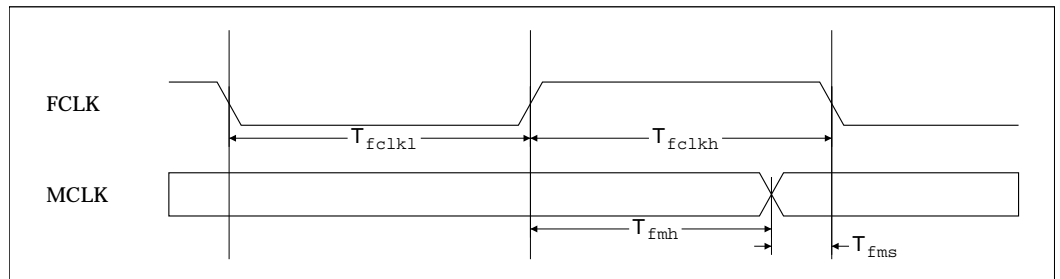


Figure 10-3: Relationship of FCLK and MCLK in synchronous mode

Bus Clocking

This chapter describes the operation of the bus interface:

11.1	ARM710a macrocell Cycle Speed	11-2
11.2	Bus Interface Signals	11-2
11.3	Cycle Types	11-3
11.4	Addressing Signals	11-8
11.5	Memory Request Signals	11-9
11.6	Data Signal Timing	11-10
11.7	Maximum Sequential Length	11-11
11.8	Read-lock-write	11-12
11.9	Use of the nWAIT pin	11-13
11.10	Use of the ALE Pin	11-14
11.11	Use of the nENDOUT Output	11-17
11.12	Bus interface Sampling Points	11-17
11.13	Big-endian / Little-endian Operation	11-20
11.14	Use of Byte Lane Selects (nBLS[3:0])	11-21
11.15	Memory Access Sequence Summary	11-23
11.16	ARM710a macrocell Cycle Type Summary	11-28

Bus Interface

11.1 ARM710a macrocell Cycle Speed

The bus interface is controlled by **MCLK**, and all timing parameters are referenced with respect to this clock. The speed of the memory may be controlled in one of two ways.

- 1 The LOW and HIGH phases of the clock may be stretched.
- 2 **nWAIT** can be used to insert entire **MCLK** cycles into the access. When LOW, this signal maintains the LOW phase of the cycle by gating out **MCLK**. See [11.9 Use of the nWAIT pin](#) on page 11-13.

When using the fastbus extension it is recommended that **nWAIT** is used to extend memory accesses rather than stretching **MCLK** directly. This is discussed in [10.1 Fastbus Extension](#) on page 10-2.

11.2 Bus Interface Signals

The signals in the Bus interface can be grouped into 3 categories:

Addressing signals:

A[31:0]
nRW
nBW
LOCK
nBLS[3:0]

Memory Request signals:

nMREQ
SEQ

Data sampled signals:

DIN[31:0]
DOUT[31:0]
ABORT

Tri-state control signal:

nENDOUT

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 **ALE** pin.

Note *Unless otherwise specified, all diagrams in this chapter show the ARM710a macrocell operating with the **ALE** pin held HIGH.*

11.3 Cycle Types

The ARM710a 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 11-1: 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.

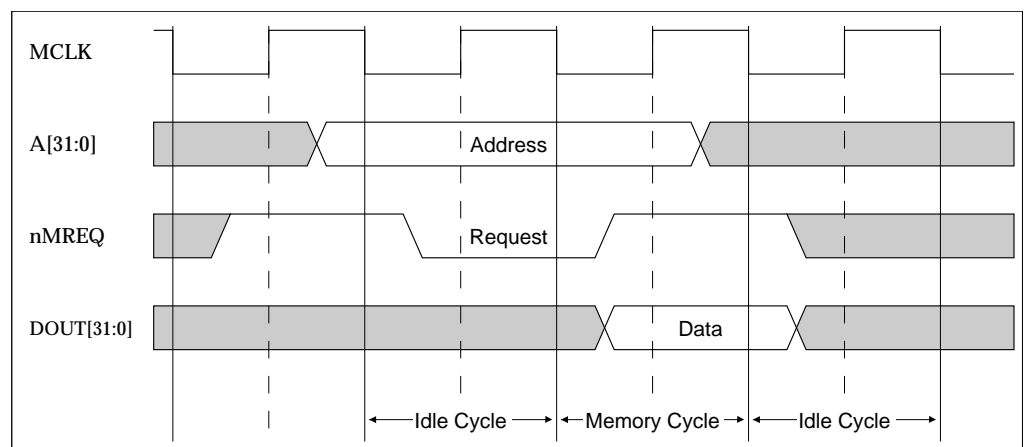


Figure 11-1: Simplified single cycle access

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 ARM710a 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 (i.e. a, a+4, a+8 etc.). See **Figure 11-2: Simplified sequential access** on page 11-4. 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 **ALE** held HIGH. The operation of **ALE** is described in **11.10 Use of the ALE Pin** on page 11-14.

Bus Interface

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 [11.9 Use of the nWAIT pin](#) on page 11-13.

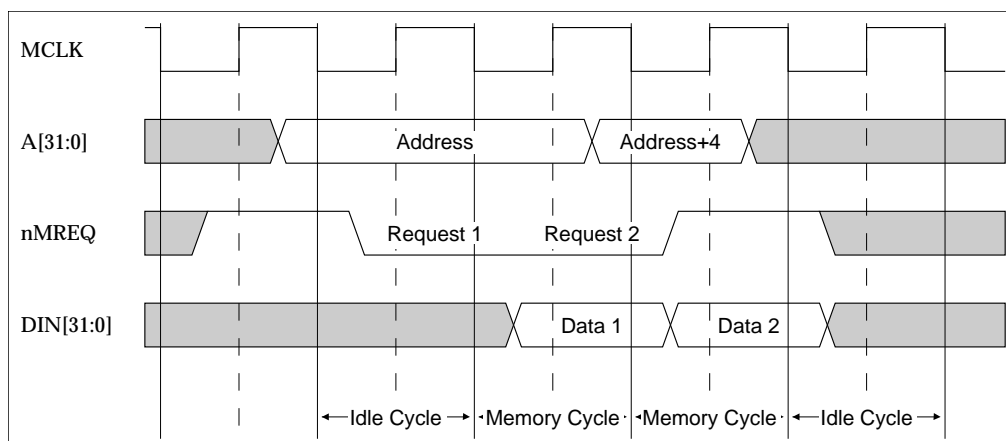


Figure 11-2: Simplified sequential access

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 11-3: Single word read or write**. The timing parameters are defined in **Chapter 13, AC Parameters in Standard Mode** and **Chapter 14, AC Parameters with Fastbus Extension**.

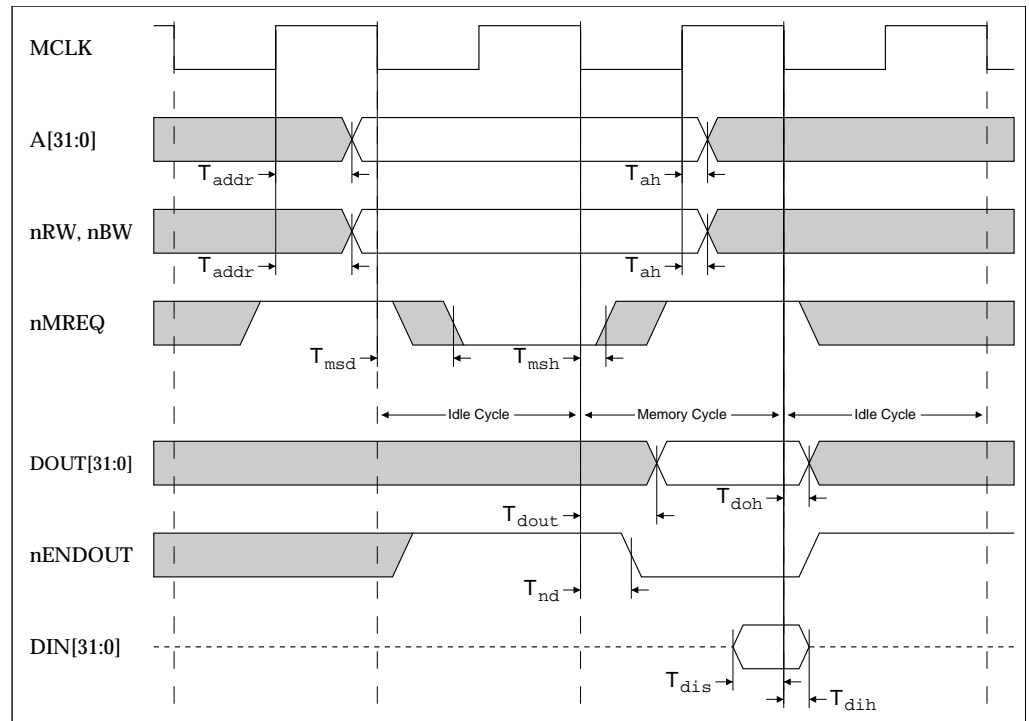


Figure 11-3: Single word read or write

Bus Interface

After a non-sequential access as shown in [Figure 11-3: Single word read or write](#) on page 11-5 the interface can perform sequential *memory cycles*. See [Figure 11-4: Two word sequential read or write](#) on page 11-6.

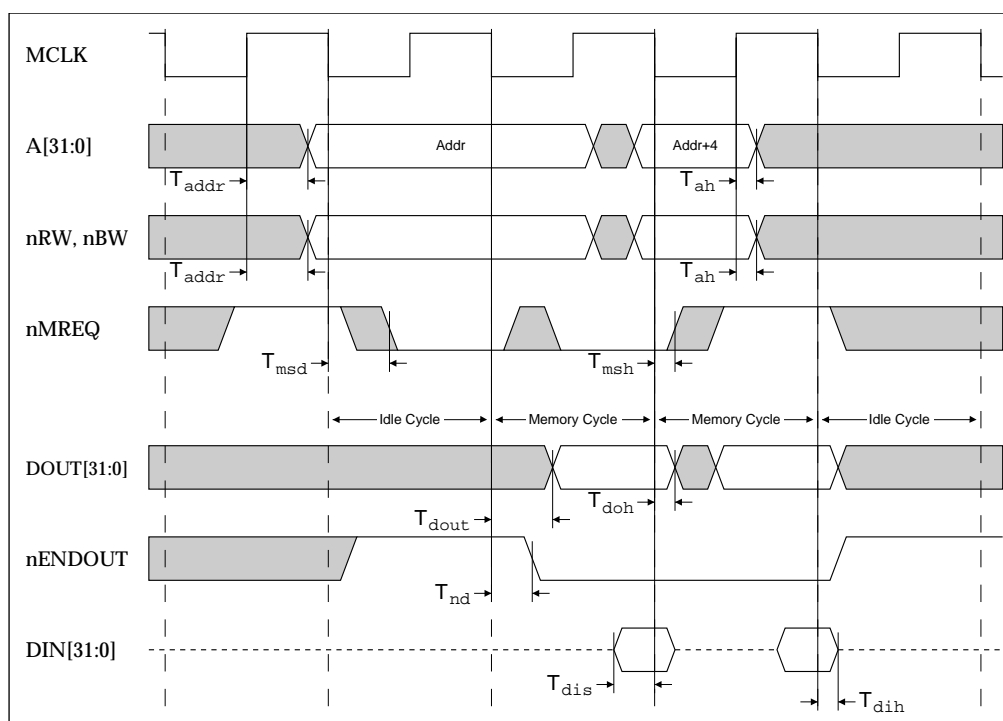


Figure 11-4: Two word sequential read or write

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 11-5: Minimum interval between bus accesses** on page 11-7

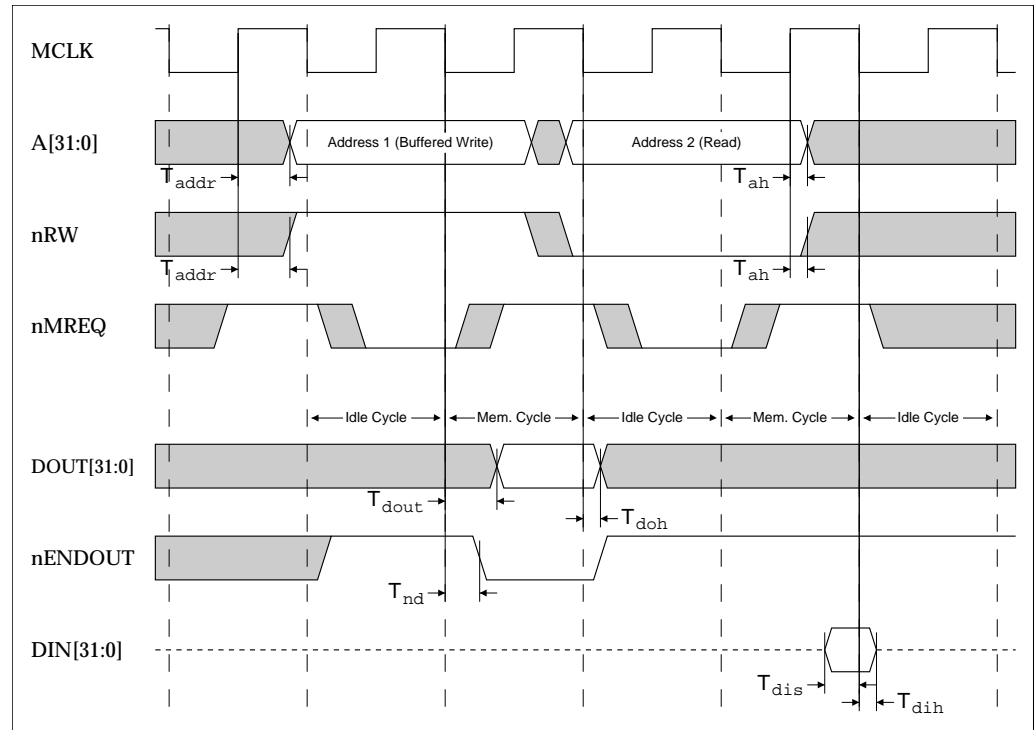


Figure 11-5: Minimum interval between bus accesses

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. See **Figure 11-10: Use of nWAIT to increase memory decode time** on page 11-16. The case shown is that of a write followed by a read. It could also have been a pair of back to back writes.

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 **ALE** should be used to modify the address timing, see **11.10 Use of the ALE Pin** on page 11-14, 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 ARM710a macrocell bus interface is discussed in **11.12 Bus interface Sampling Points** on page 11-17.

Bus Interface

11.4 Addressing Signals

The timing of the addressing signals can be modified using the **ALE** pin. If this pin is HIGH the addressing signals will be timed from the rising edge of the memory clock **MCLK**.

This is 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 or a byte quantity. These are differentiated by the signal **nBW**. Again, **nBW** may not change during sequential accesses. It is not possible to perform sequential byte accesses.

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 11-5: Minimum interval between bus accesses](#) on page 11-7.

If the **FASTBUS** pin is LOW, the **ALE** pin directly controls the latches on the addressing signals. These latches are closed, and all of the addressing signals held in their current state when **ALE** is LOW. When **ALE** is HIGH the latches are open, and the addressing signals are free to change.

When operating the device with the **FASTBUS** pin HIGH, and the **ALE** pin LOW, the addresses are latched until the LOW phase of **MCLK**. See [Figure 11-6: Single word read or write with delayed addressing](#) on page 11-9. This is discussed further in [11.10 Use of the ALE Pin](#) on page 11-14.

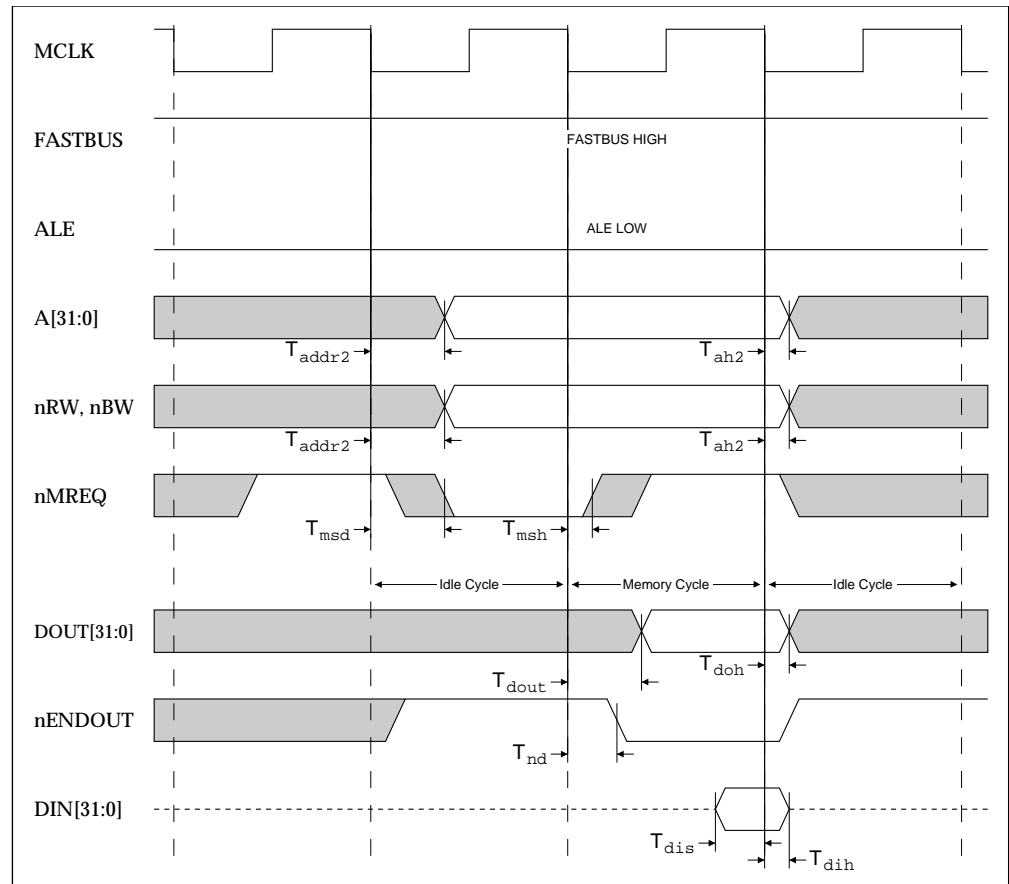


Figure 11-6: Single word read or write with delayed addressing

11.5 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 ARM710a macrocell 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 [11.9 Use of the nWAIT pin](#) on page 11-13.

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.

Bus Interface

11.6 Data Signal Timing

11.6.1 DIN[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 13, AC Parameters in Standard Mode](#) and [Chapter 14, AC Parameters with Fastbus Extension](#).

In a low power system it is important to ensure that **DIN[31:0]** 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 are left enabled during sleep to hold the bus at a defined level.

11.6.2 DOUT[31:0]

During a write access, the data on **DOUT[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 [11.9 Use of the nWAIT pin](#) on page 11-13.

11.6.3 ABORT

The **ABORT** signal is sampled at the end of the *memory* cycle, on both read and write accesses. The effect of **ABORT** on the operation of the ARM710a macrocell is discussed in [Chapter 3, Programmer's Model](#).

An **ABORT** can be flagged on any *memory* cycle, however it will be ignored on buffered writes, which cannot be aborted.

The effect of **ABORT** during linefetches is slightly different to that during other access. During a linefetch the ARM710a macrocell will fetch 4 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.

11.7 Maximum Sequential Length

The ARM710a macrocell 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 page size is greater than 256 words.

Bus Interface

11.8 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 **ALE**. If **ALE** 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 ARM has started a read-lock-write sequence it cannot be interrupted until it has completed.

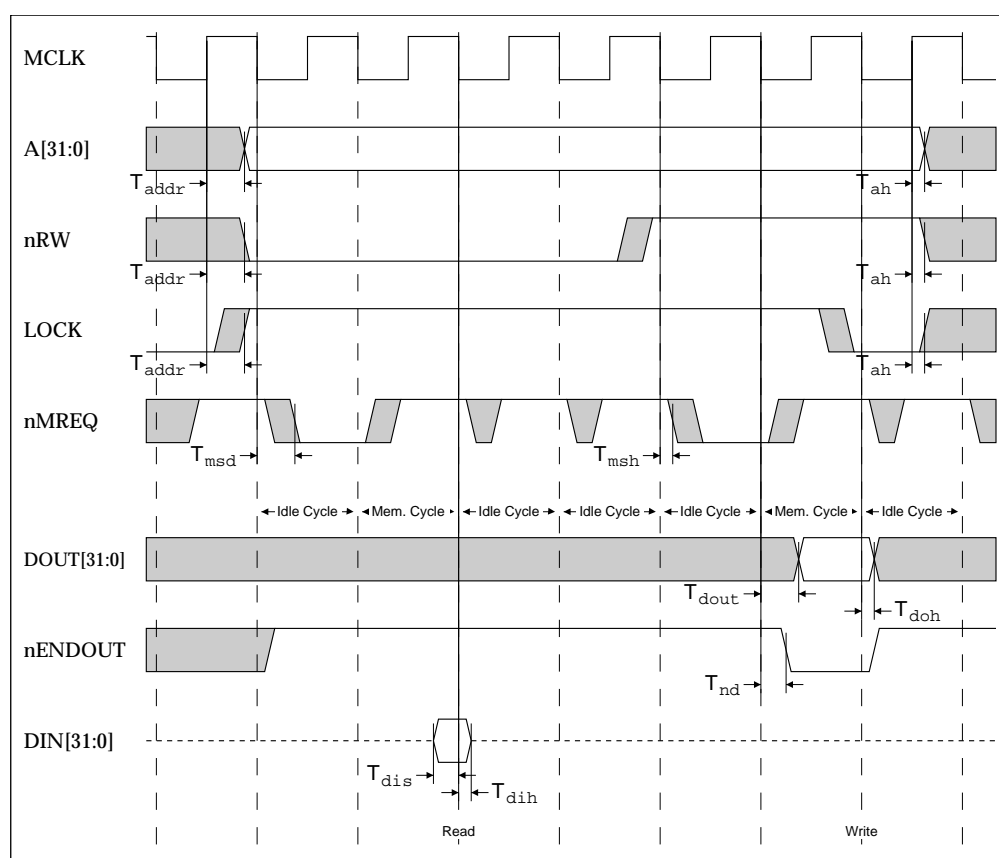


Figure 11-7: Read - locked - write

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.

11.9 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 was 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.

In standard mode, if **ALE** is being used to latch an address when **nWAIT** is taken LOW, the address and control signals will change when **ALE** returns HIGH, regardless of the state of **nWAIT**. See [Figure 11-8: Use of the nWAIT pin to stop ARM710a macrocell for 1 MCLK cycle](#) on page 11-13.

In fastbus mode the address timing is dependant on **nWAIT** as follows:

- If **ALE** 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 **ALE** 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.

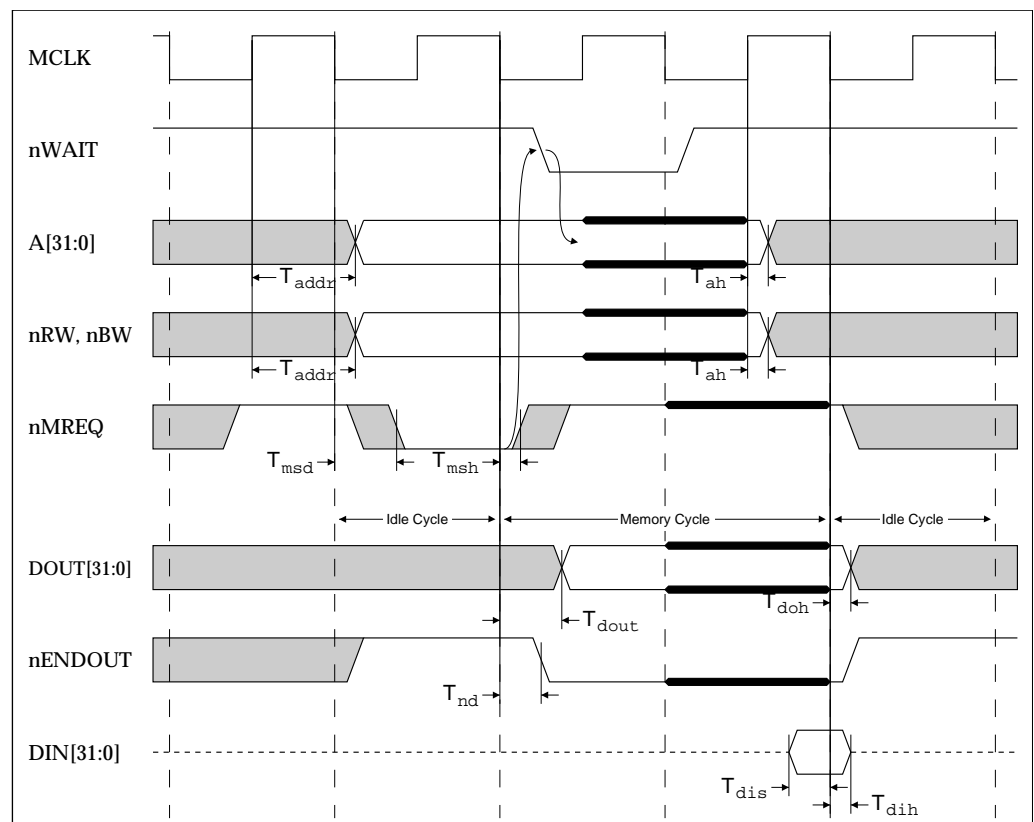



Figure 11-8: Use of the nWAIT pin to stop ARM710a macrocell for 1 MCLK cycle

Bus Interface

The heavy bars indicate the cycle for which signals will be stable as a result of asserting **nWAIT**.

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 11-8: Use of the **nWAIT** pin to stop ARM710a macrocell for 1 MCLK cycle on page 11-13

11.10 Use of the ALE Pin


The **ALE** pin operates differently with and without fastbus extension. In both cases it is used to modify the timing of the addressing signals.

Without fastbus extension (**FASTBUS** LOW), **ALE** directly controls the address latches. If **ALE** is held HIGH the address will flow out during the HIGH phase of **MCLK**. By taking **ALE** LOW, the current address is latched, and further transitions on **A[31:0]**, **nBLS[3:0]**, **nRW**, **nBW** and **LOCK** are prevented. The falling edge of **ALE** can be up to T_{ald} after the rising edge of **MCLK** to guarantee that the Address and associated signals will not change.

With fastbus extension (**FASTBUS** HIGH), **ALE** is used to modify the address timing.

- If **ALE** 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 **ALE** is LOW the address timing is modified, and the address changes during the following LOW phase of **MCLK**. See  Figure 14-2: ARM710a macrocell bus timing, **ALE** LOW on page 14-4.

It is possible to change **ALE** in the LOW phase of **MCLK** to vary the address timing during a cycle. For example, it may be desirable to have the addresses held when accessing a ROM, but to normally have early addresses for address decoding. In this case **ALE** would be taken LOW in the first cycle of the ROM access to switch to late address timing. This would hold the addresses as required by the ROM. In the LOW phase, after the access completes, **ALE** could be taken HIGH to switch back to normal address timing.

 Figure 11-9: Use of **ALE** in fastbus mode shows the use of **ALE** to alter the address timing within a burst access.

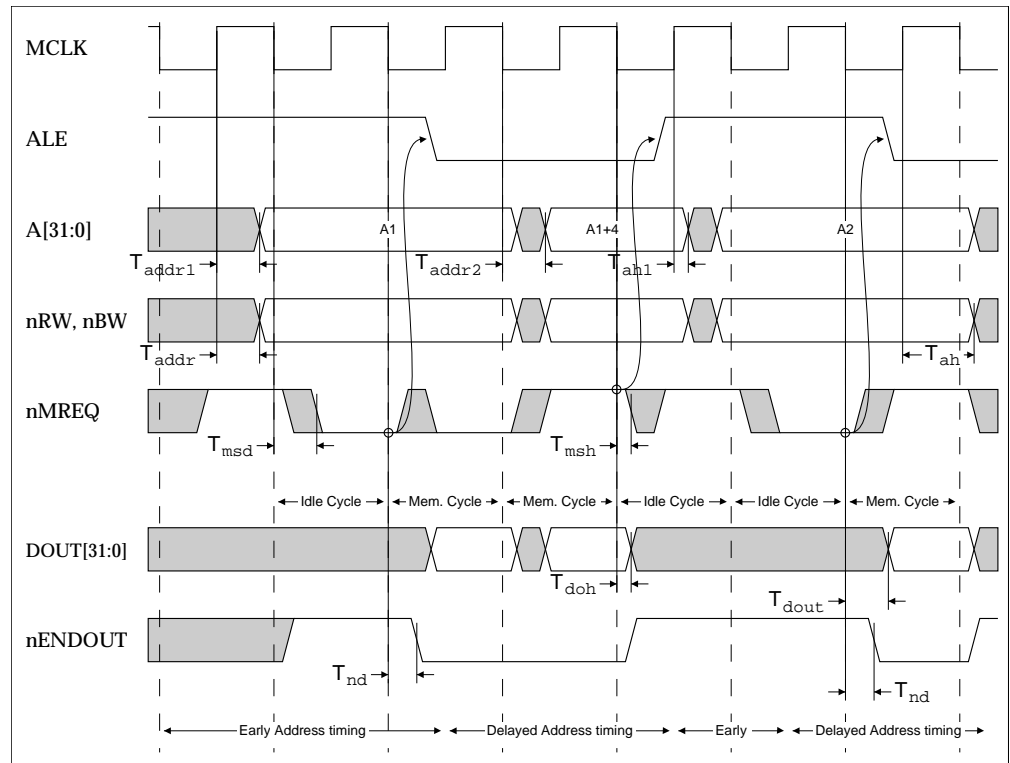


Figure 11-9: Use of ALE in fastbus mode

Note Back to back buffered writes can occur with only a single idle cycle between the two write cycles. This means that the second address will be delayed if this technique is used. A wait state may be required to allow sufficient time for memory decoding.

► **Figure 11-10: Use of nWAIT to increase memory decode time** shows a pair of single cycle buffered writes, with **nWAIT** being used to add an extra cycle after the first access.

Bus Interface

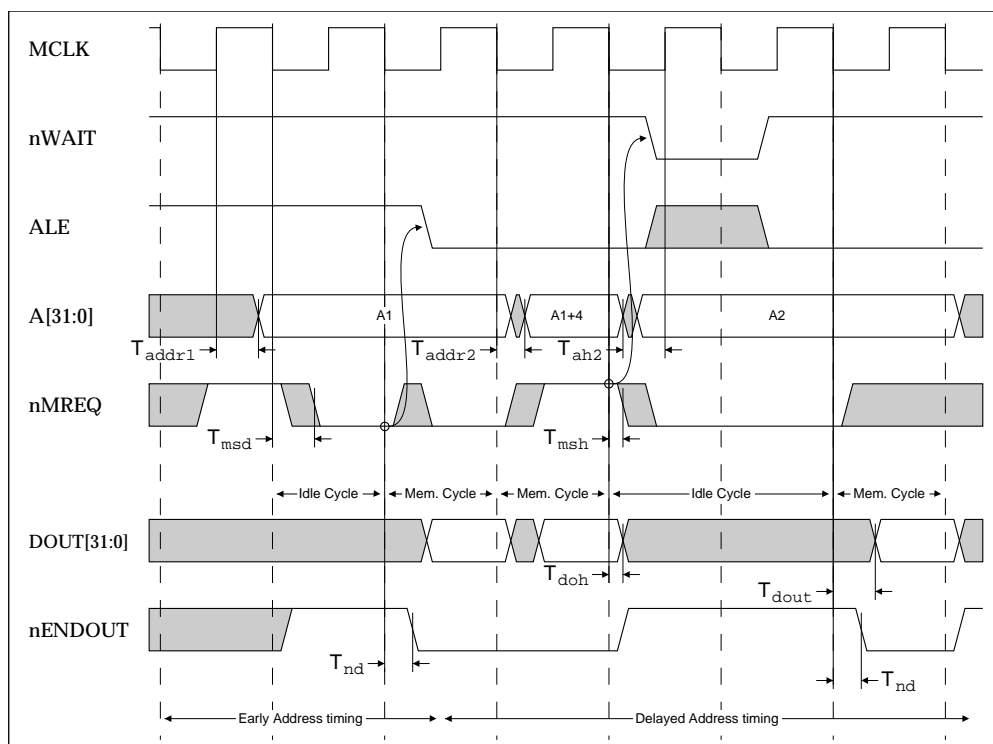


Figure 11-10: Use of nWAIT to increase memory decode time

11.11 Use of the nENDOUT Output

The signal **nENDOUT** can be used directly to drive a set of tri-state drivers connected to the **DOUT[31:0]** bus. **nENDOUT** will only go LOW during valid write cycles, and will always be forced HIGH by **DBE** LOW.

The use of external bus drivers allows the drive strength to be matched to the capacitive load of the system bus, which will allow the selection of the optimum speed/power consumption trade-off for each system.

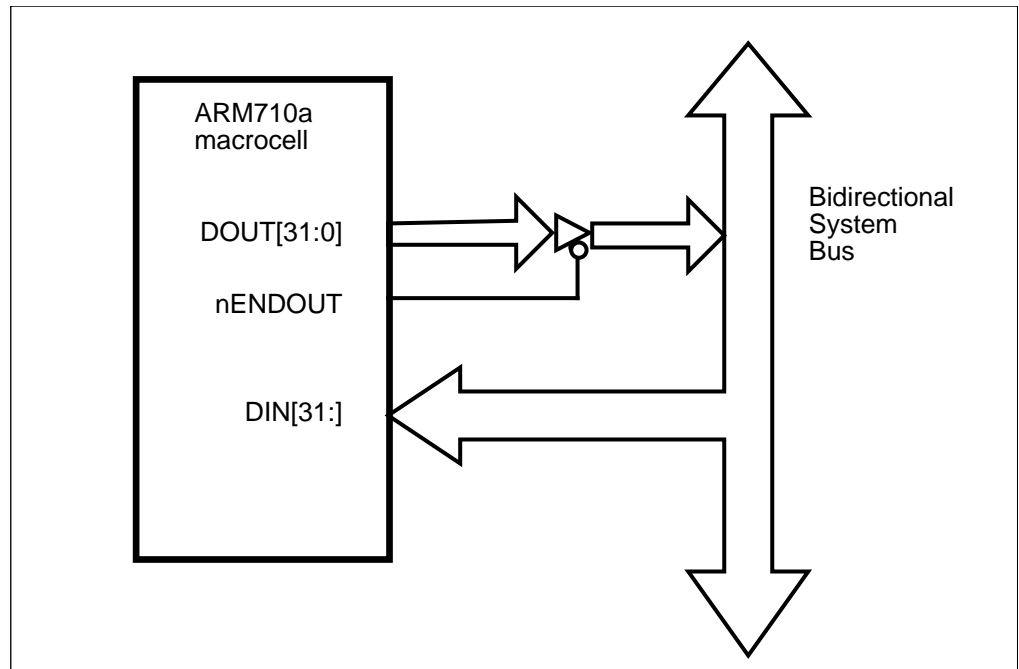


Figure 11-11: Connection of external tri-state drivers

11.12 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 technique is to sample the value of **nMREQ** only when **nWAIT** is HIGH. This is shown by the transfer bars in **Figure 11-12: Sampling points at maximum frequency** and **Figure 11-13: Sampling points at reduced frequency**.

Bus Interface

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.

11.12.1 Fast Operation

If the ARM710a macrocell 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.

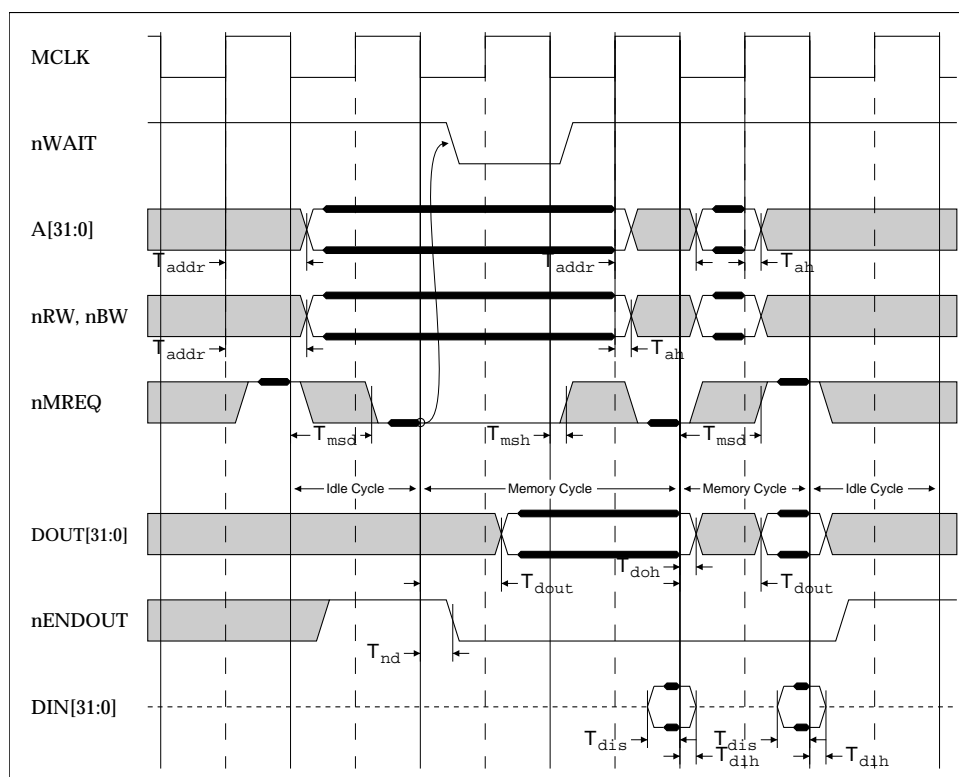


Figure 11-12: Sampling points at maximum frequency

Signal	Earliest Recommended Sample Point
A[31:0]	Set-up to MCLK RISING
nBLS[3:0], nRW, nBW, LOCK	Set-up to MCLK RISING
nMREQ, SEQ	Set-up to MCLK FALLING
DOUT[31:0]	Set-up to MCLK FALLING

Table 11-1: 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.

11.12.2 Reduced 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.

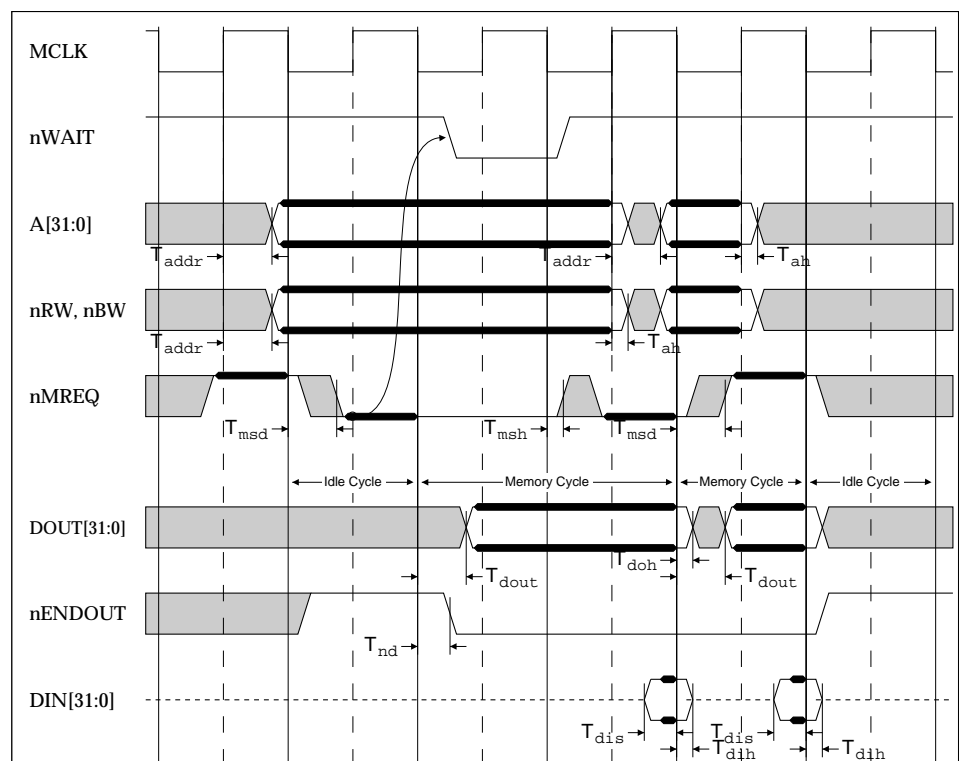


Figure 11-13: Sampling points at reduced frequency

Signal	Earliest Recommended Sample Point
A[31:0]	MCLK FALLING
nBLS[3:0], nRW, nBW, LOCK	MCLK FALLING
nMREQ, SEQ	MCLK RISING
DOUT[31:0]	MCLK RISING

Table 11-2: Sampling points at Reduced frequency

Bus Interface

11.13 Big-endian / Little-endian Operation

The ARM710a macrocell 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.

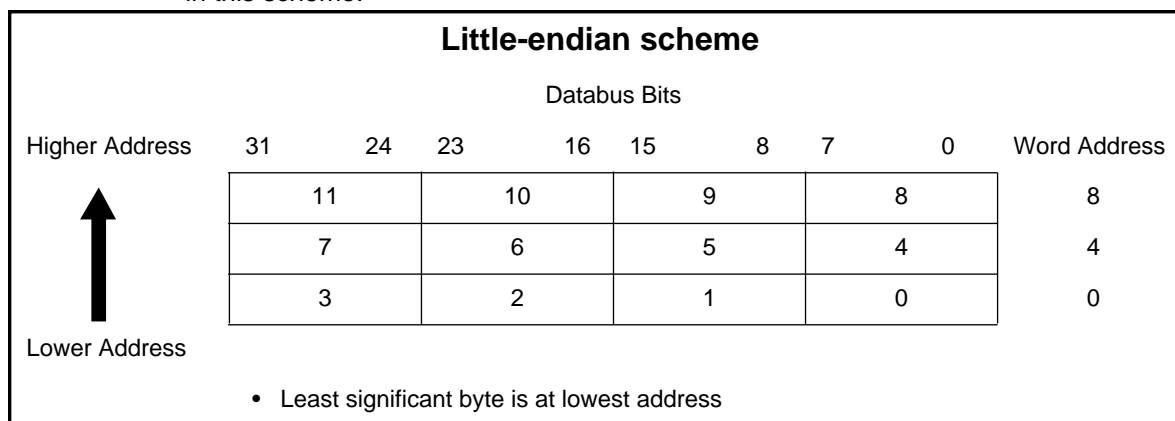


Figure 11-14: Little-endian addresses of bytes within word

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 [4.7 Single Data Transfer \(LDR, STR\)](#) on page 4-21 for more details.

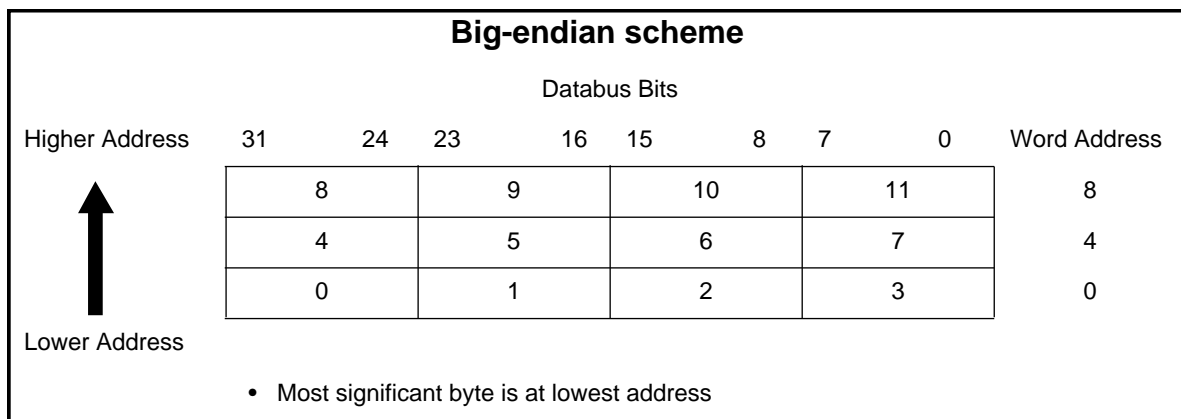


Figure 11-15: Big-endian addresses of bytes within words

11.13.1 Word operations

All word operations expect the data to be presented on data bus inputs 31 through 0. The external memory system should ignore the bottom two bits of the address if a word operation is indicated.

11.13.2 Byte operations

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

Little-endian operation

A byte load (LDRB) expects the data on **DIN[31:0]** 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. See *Figure 11-14: Little-endian addresses of bytes within word* on page 11-20.

Big-endian operation

A byte load (LDRB) expects the data on **DIN[31:0]** 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. See *Figure 11-15: Big-endian addresses of bytes within words* on page 11-20.

11.14 Use of Byte Lane Selects (nBLS[3:0])

The Byte Lane Selects are active low signals which indicate which bytes of the memory system are being accessed during a memory cycle. They are generated by a combinatorial decode of the bottom 2 address bits, and have the same timing as the address:

nBW	A[1:0]	nBLS[3:0]
1	XX	0000
0	00	1110
0	01	1101
0	10	1011
0	11	0111

Table 11-3: Selected bytes within the memory

Bus Interface

This decoding is independent of whether big-endian or little-endian operation is selected. Currently only 5 combinations of these signals are used. Further combinations may be used in future ARM processors. We recommend that you use the **nBLS[]** signals in new designs. The Byte Lane Selects relate to the Databus as follows:

Signal	Byte
nBLS[0]	DIN[7:0]/DOUT[7:0]
nBLS[1]	DIN[15:8]/DOUT[15:8]
nBLS[2]	DIN[23:16]/DOUT[23:16]
nBLS[3]	DIN[31:24]/DOUT[31:24]

Table 11-4: Little-endian operation

Signal	Byte
nBLS[0]	DIN[31:24]/DOUT[31:24]
nBLS[1]	DIN[23:16]/DOUT[23:16]
nBLS[2]	DIN[15:8]/DOUT[15:8]
nBLS[3]	DIN[7:0]/DOUT[7:0]

Table 11-5: Big-endian operation

The memory system should decode the Byte Lane Selects as appropriate for the area of memory which is being accessed.

Note that during byte reads it should be ensured that all of the bytes of the databus are driven to a defined level. Floating input levels on the other bytes of the databus may result in increased power consumption.

11.15 Memory Access Sequence Summary

ARM710a macrocell 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 ARM710a macrocell are:

Unbuffered Write	Level 1 translation fetch
Uncached Read	Level 2 translation fetch
Buffered Write	Read-Lock-Write sequence
Linefetch	

See also [11.16 ARM710a macrocell Cycle Type Summary](#) on page 11-28.

11.15.1 Unbuffered 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) of the instruction requesting the cycle.

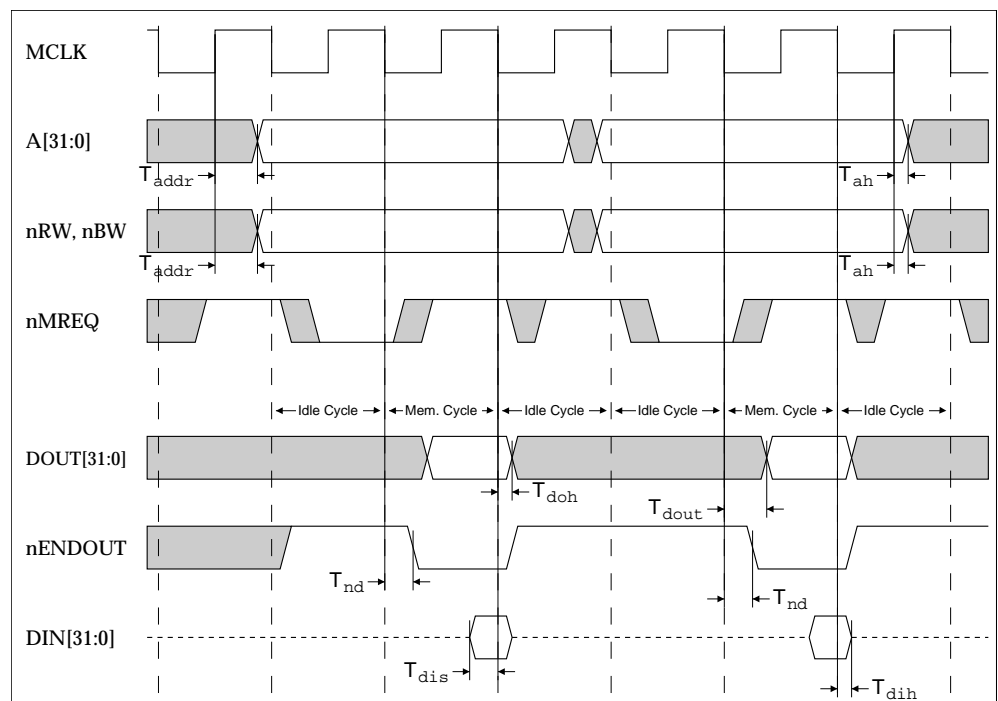


Figure 11-16: Two single word non-sequential unbuffered accesses

Bus Interface

11.15.2 Buffered 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) 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.

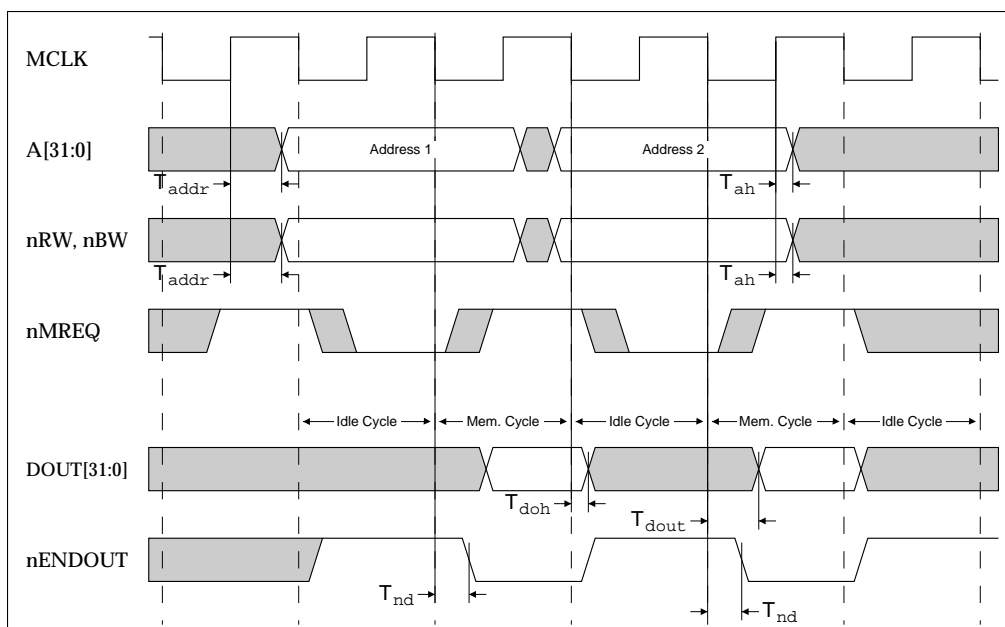


Figure 11-17: Two single word non-sequential buffered writes

Note that in the case of a pair of buffered writes, there may only be a single idle cycle between the two accesses. Refer to [Figure 11-10: Use of nWAIT to increase memory decode time](#) on page 11-16.

11.15.3 Linefetch

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 access will be a series of word accesses on the bus. **Figure 11-18: Linefetch** shows a linefetch.

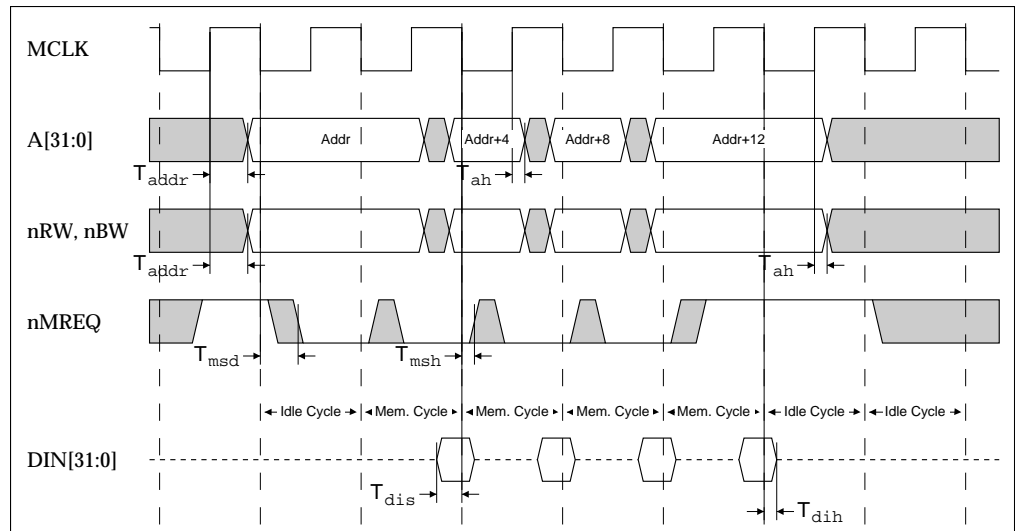


Figure 11-18: Linefetch

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 purged but program flow will *not* be interrupted.

Bus Interface

11.15.4 Translation 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.

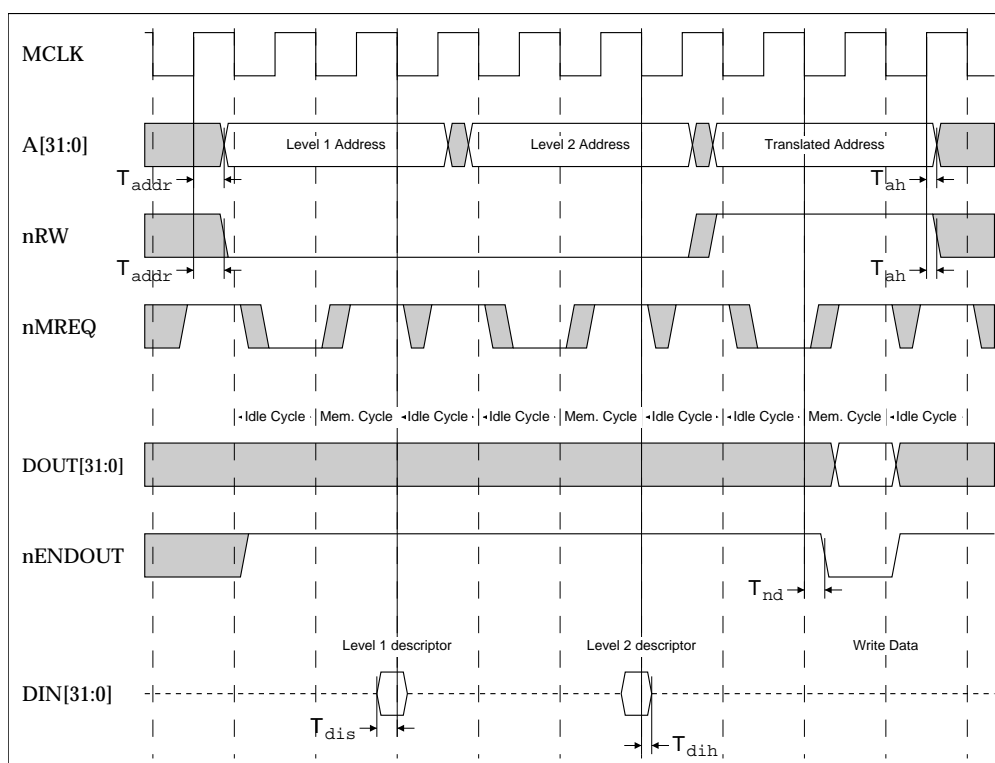


Figure 11-19: Translation table-walking sequence for page

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.

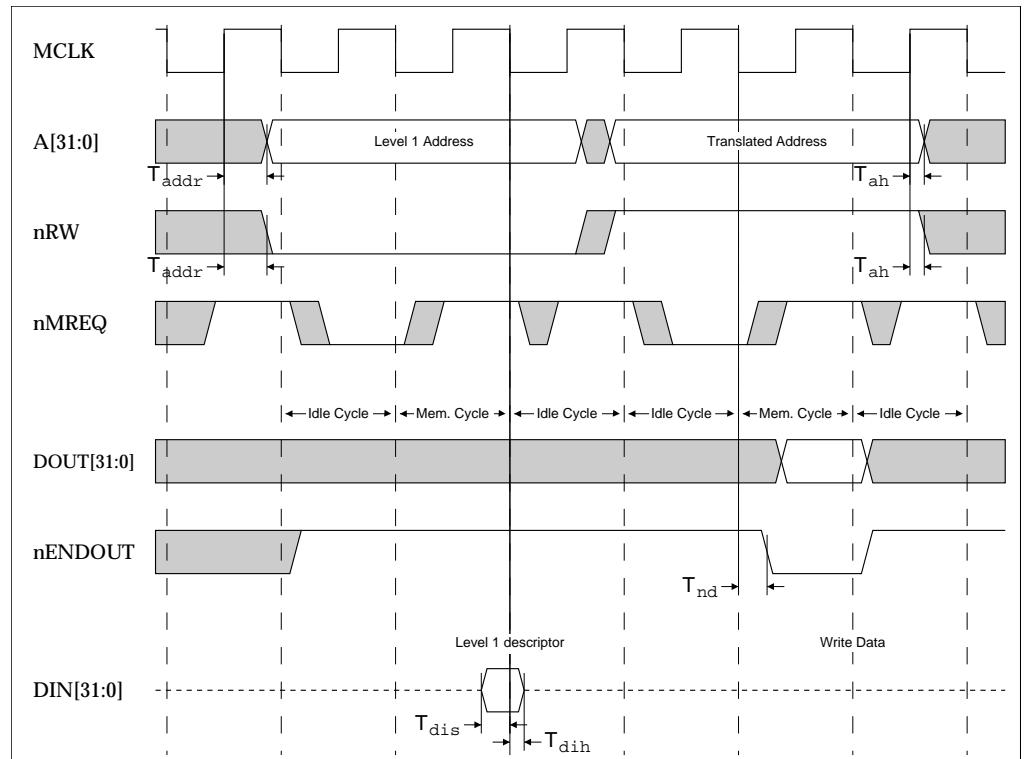


Figure 11-20: Translation table-walking sequence for section

11.16 ARM710a macrocell 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	
	Start	write	a	idle	
		write	a	memory	data
	Repeat	write	a+n	memory	
					data
	Read phase	read	aL	idle	
		read	aL	memory	
		read	aL	idle	data
		read	aL	idle	
Buffered Write	Write phase	write	aL	idle	
		write	aL	memory	
		write	aL	idle	data
	Write phase after aborted read	read	aL	idle	
		read	aL	idle	
		read	aL	idle	
	Section Translation Fetch	read	l1a	idle	
		read	l1a	memory	
		read	l1a	idle	data

Table 11-6: Cycle type summary

Operation	nRW	A[31:0]	nMREQ	D[31:0]
Start	read	l1a	idle	
	read	l1a	memory	
	read	l1a	idle	data
	read	l2a	idle	
	read	l2a	memory	
	read	l2a	idle	data

Page Translation Fetch

Table 11-6: 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 11-6: Cycle type summary** on page 11-28 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.

Bus Interface

12

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.

12.1	Absolute Maximum Ratings	12-2
12.2	DC Operating Conditions	12-2
12.3	DC Characteristics	12-3



DC Parameters

12.1 Absolute Maximum Ratings

Symbol	Parameter	Min	Max	Units	Note
Vip	Voltage applied to any signal	VSS-0.3	VDD+0.3	V	1
Ts	Storage temperature	-40	125	deg C	1

Table 12-1: ARM710a macrocell 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.

12.2 DC Operating Conditions

Symbol	Parameter	Min	Typ	Max	Units	Notes
VDD	Supply voltage				V	3
Vihc	IC input HIGH voltage	0.8xVDD		VDD	V	1,4
Vilc	IC input LOW voltage	0.0		0.2xVDD	V	1,4
Vohc	OCZ output HIGH voltage	0.9xVDD		VDD	V	1,2
Volc	OCZ output LOW voltage	0.0		0.1xVDD	V	1,2
Ta	Ambient operating temperature	0		70	°C	

Table 12-2: ARM710a macrocell DC operating conditions

Notes

- 1 Voltages measured with respect to VSS.
- 2 OCZ - Output, CMOS levels, tri-stateable.
- 3 This parameter is process dependent.
- 4 Operating the device with Vih less than VDD or Vil greater than VSS will result in increased power consumption.

12.3 DC Characteristics

Symbol	Parameter	Nom	Units	Note
IDD	Static Supply current		μA	1
Isc	Output short circuit current		mA	1
Il _u	DC latch-up current		mA	1
I _{in}	IC input leakage current		μA	1
I _{oh}	Output HIGH current (V _{out} = VDD-0.4V)		mA	1
I _{ol}	Output LOW current (V _{out} = VSS+0.4V)		mA	1
C _{in}	Input capacitance		pF	1

Table 12-3: ARM710a macrocell DC characteristics

Notes

- 1 These parameters are process dependent.

DC Parameters

13

AC Parameters in Standard Mode

This chapter describes the AC Parameters in Standard Mode.

13.1	Test Conditions	13-2
13.2	Relationship between FCLK & MCLK in Synchronous Mode	13-2
13.3	Main Bus Signals	13-4

AC Parameters in Standard Mode

13.1 Test Conditions

The AC timing diagrams presented in this section assume that the outputs of ARM710a macrocell 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 ARM710a macrocell might be employed. The output pads of ARM710a macrocell 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)	Note
A[31:0]	2		1
DOUT[31:0]	2		1
nR/W	2		1
nB/W	2		1
LOCK	2		1
nMREQ	2		1
SEQ	2		1

Table 13-1: ARM710a macrocell AC test conditions

Note

- 1 These parameters are process dependent.

13.2 Relationship between FCLK & MCLK in Synchronous Mode

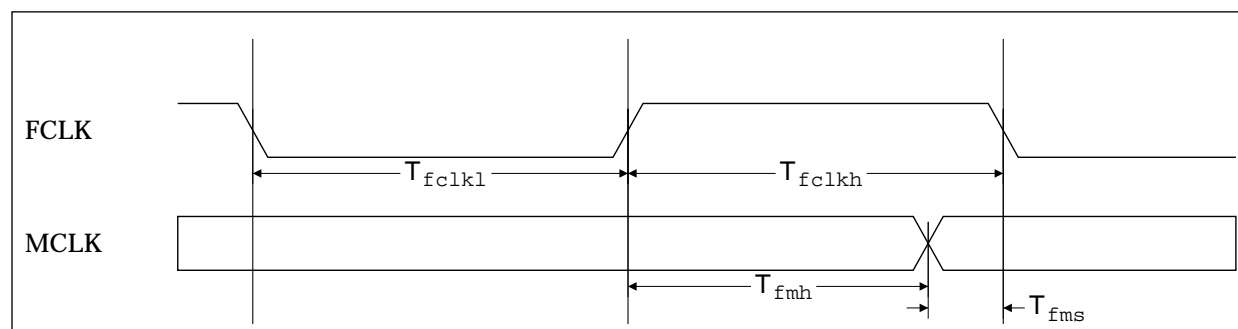


Figure 13-1: Clock timing relationship

AC Parameters in Standard Mode

Symbol	Parameter	5V Min	5V Max	3.3V Min	3.3V Max	Unit	Note
Tfckl	FCLK LOW time					ns	1
Tfckh	FCLK HIGH time					ns	1
Tfckc	FCLK cycle time					ns	1
Tfmh	FCLK - MCLK hold time					ns	2
Tmfs	MCLK - FCLK setup					ns	2

Table 13-2: ARM710a macrocell FCLK timing

NB: **FCLK** frequency must be strictly greater than or equal to **MCLK** at all times.

Notes

- 1 **FCLK** timings measured at 50% of Vdd. This applies to both synchronous and asynchronous operation.
- 2 Applicable in Synchronous mode only

13.2.1 Tald measurement

Tald is the maximum delay allowed in the **ALE** input transition to guarantee that neither address nor byte lane strobes will change.

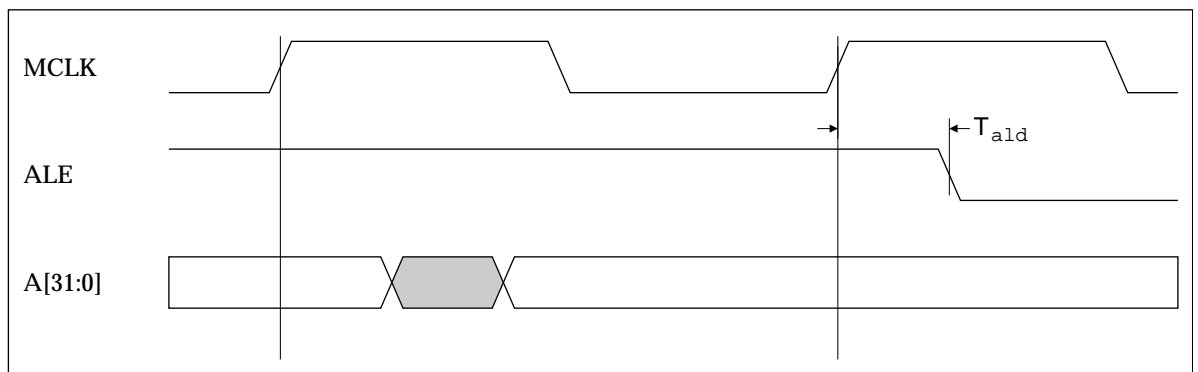


Figure 13-2: Tald measurement

AC Parameters in Standard Mode

13.3 Main Bus Signals

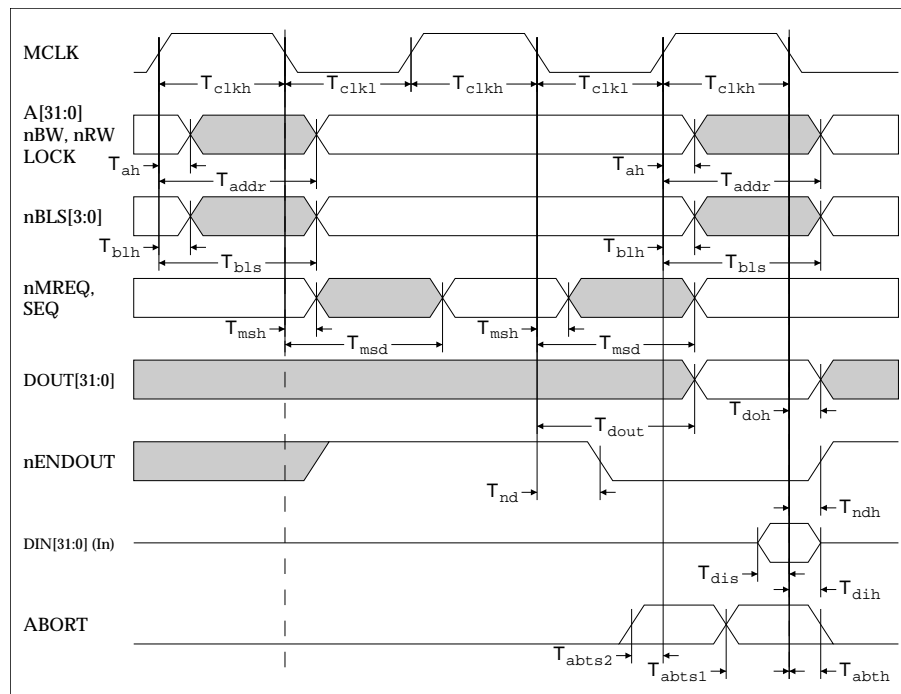


Figure 13-3: ARM710a macrocell main bus timing

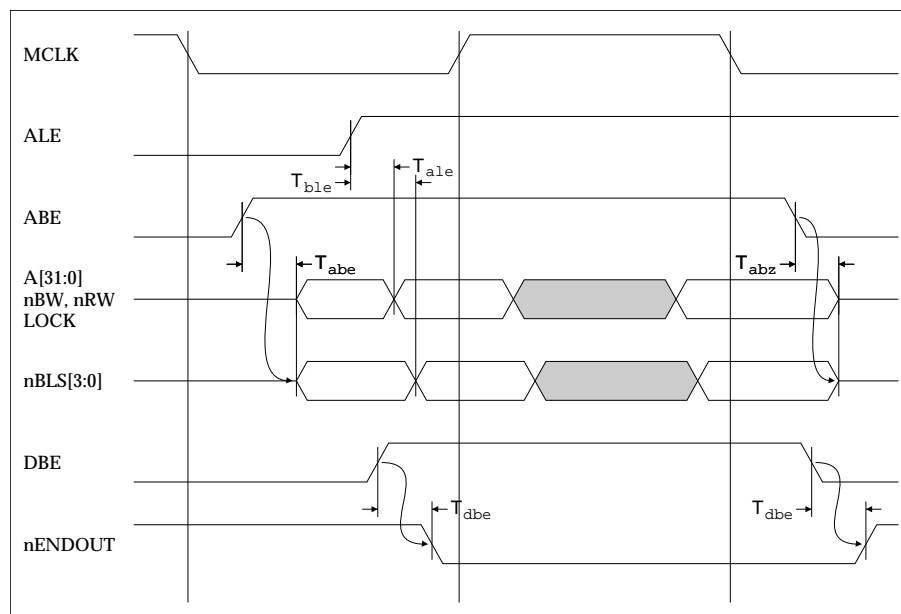


Figure 13-4: ARM710a macrocell bus enable timing

AC Parameters in Standard Mode

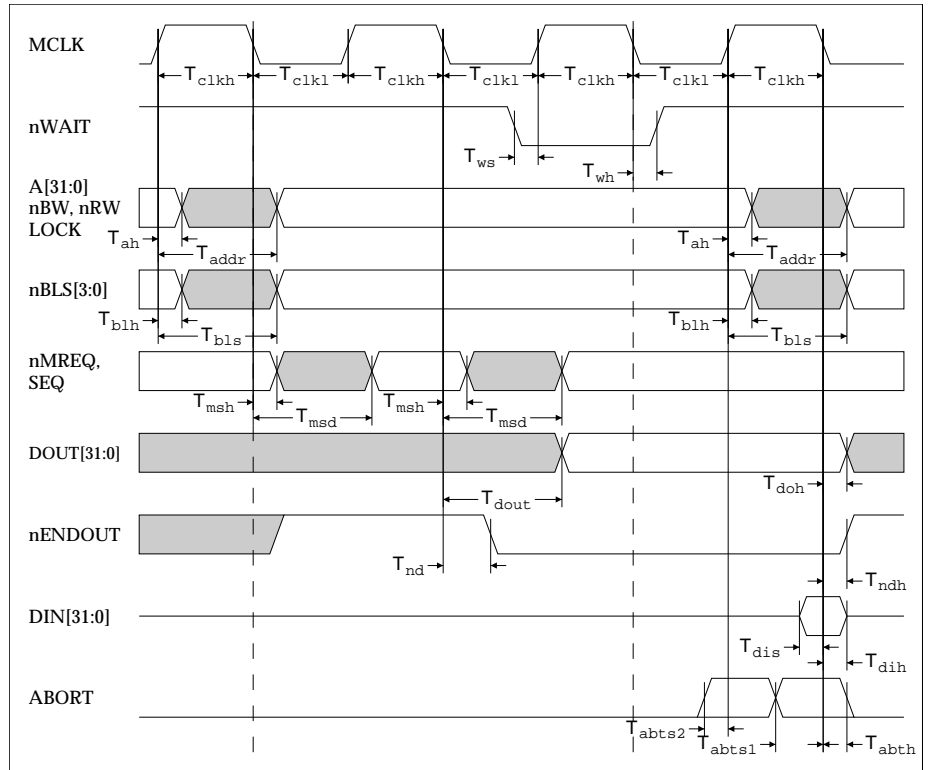


Figure 13-5: ARM710a macrocell nWAIT timing

The following timings are supplied as a guide only. Please refer to your semiconductor vendor for definitive parameters.

Symbol	Parameter	5V Min	5V Max	3.3V Min	3.3V Max	Unit	Note
Tmckl	MCLK LOW time					ns	1
Tmckh	MCLK HIGH time					ns	
Tmckc	MCLK cycle time						1
Tws	nWAIT setup to MCLK					ns	
Twh	nWAIT hold from MCLK					ns	
Tale	address latch enable					ns	3
Tble	Byte lane latch enable					ns	
Tald	address latch disable					ns	
Tabc	address bus enable					ns	2
Tabz	address bus disable					ns	
Taddr	MCLK to address delay					ns	2
Tah	address hold time					ns	2

Table 13-3: ARM710a macrocell bus timing



AC Parameters in Standard Mode

Symbol	Parameter	5V Min	5V Max	3.3V Min	3.3V Max	Unit	Note
Tbls	MCLK to byte lane delay					ns	2
Tblh	byte lane hold time					ns	2
Tnd	nENDOUT delay					ns	
Tndh	nENDOUT hold time					ns	
Tdbe	DBE to nENDOUT delay					ns	
Tdout	data out delay					ns	2
Tdoh	data out hold					ns	2
Tdis	data in setup					ns	
Tdih	data in hold					ns	
Tabts1	ABORT setup time					ns	4
Tabts2	ABORT setup time					ns	4
Tabth	ABORT hold time					ns	
Tmsd	nMREQ & SEQ delay					ns	
Tmsh	nMREQ & SEQ hold					ns	

Table 13-3: ARM710a macrocell bus timing (Continued)

Please refer to your semiconductor vendor for definitive AC Parameters.

Notes

- 1 **MCLK** timings measured between clock edges at 50% of Vdd.
- 2 The timings of these buses are measured to 50% of Vdd.
- 3 See [13.2.1 Tald measurement](#) on page 13-3.
- 4 Tabts1 is required by this device. To ensure compatibility with future ARM processors, we recommend that designs should meet Tabts2. Tabts2 is not tested on this device, and is given as a recommendation only.

14

AC Parameters with Fastbus Extension

This chapter describes the AC Parameters with the Fastbus extension. The information in this chapter is provided as a guide only. Refer to your semiconductor vendor for definitive AC Parameters.

14.1	Test Conditions	14-2
14.2	Main Bus Signals	14-3

AC Parameters with Fastbus Extension

14.1 Test Conditions

The AC timing diagrams presented in this section assume that the outputs of ARM710a macrocell 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 ARM710a macrocell might be employed. The output pads of ARM710a macrocell 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)	Note
A[31:0]	2		1
DOUT[31:0]	2		1
nR/W	2		1
nB/W	2		1
LOCK	2		1
nMREQ	2		1
SEQ	2		1

Table 14-1: ARM710a macrocell AC test conditions

Note

- 1 These parameters are process dependent.

14.2 Main Bus Signals

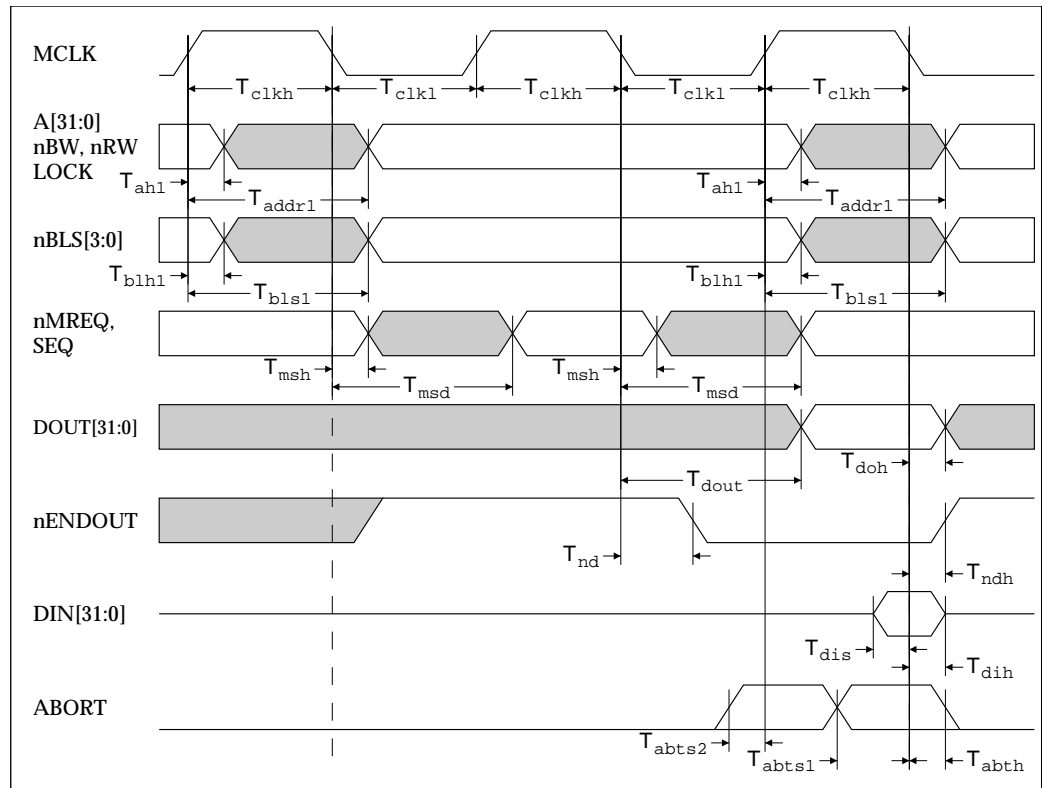


Figure 14-1: ARM710a macrocell bus timing, ALE HIGH

AC Parameters with Fastbus Extension

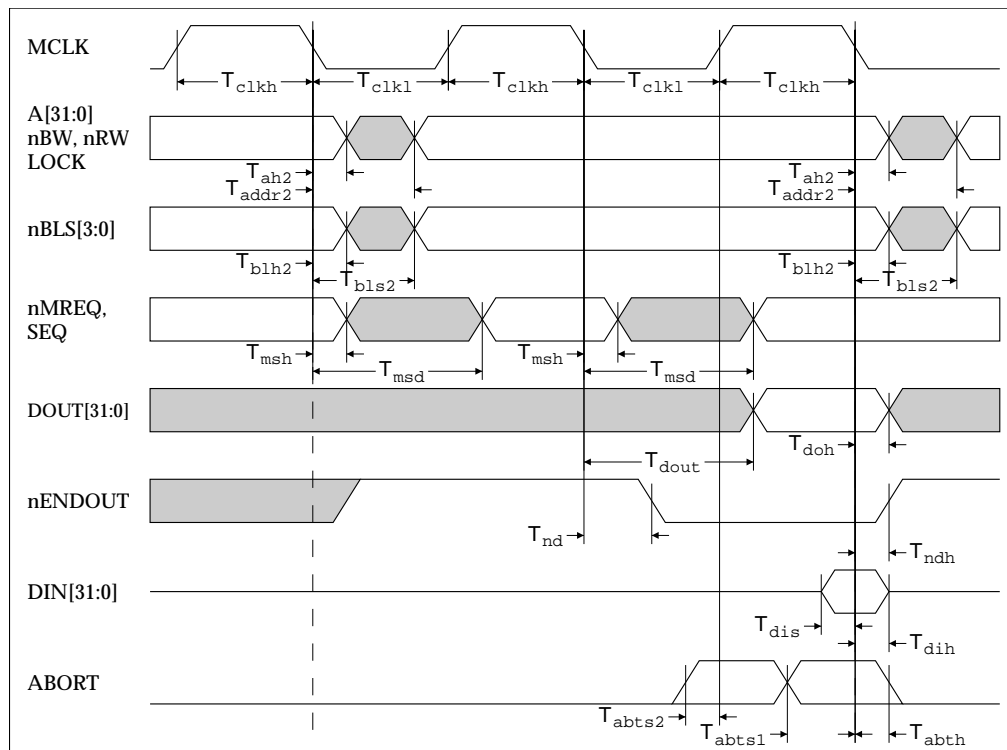


Figure 14-2: ARM710a macrocell bus timing, ALE LOW

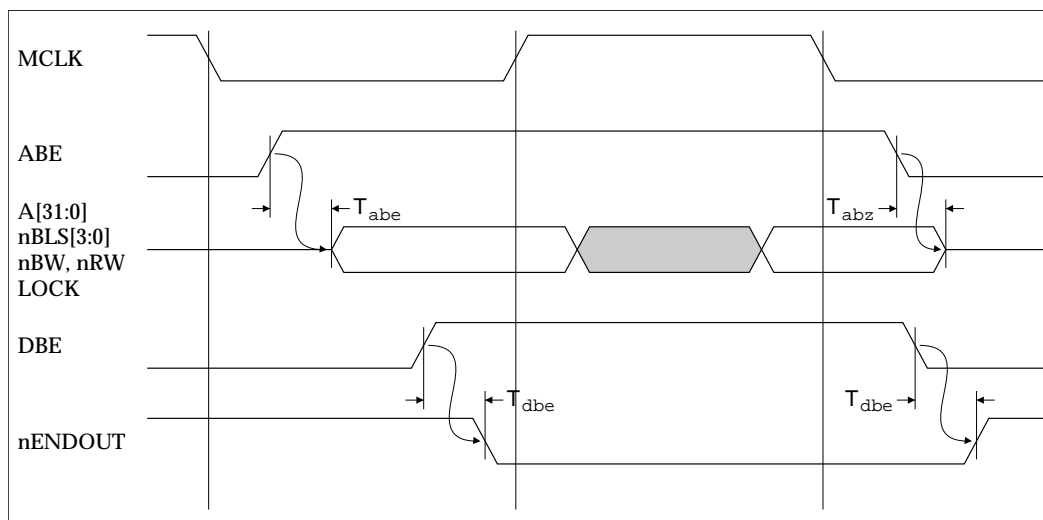


Figure 14-3: ARM710a macrocell bus enable timing

AC Parameters with Fastbus Extension

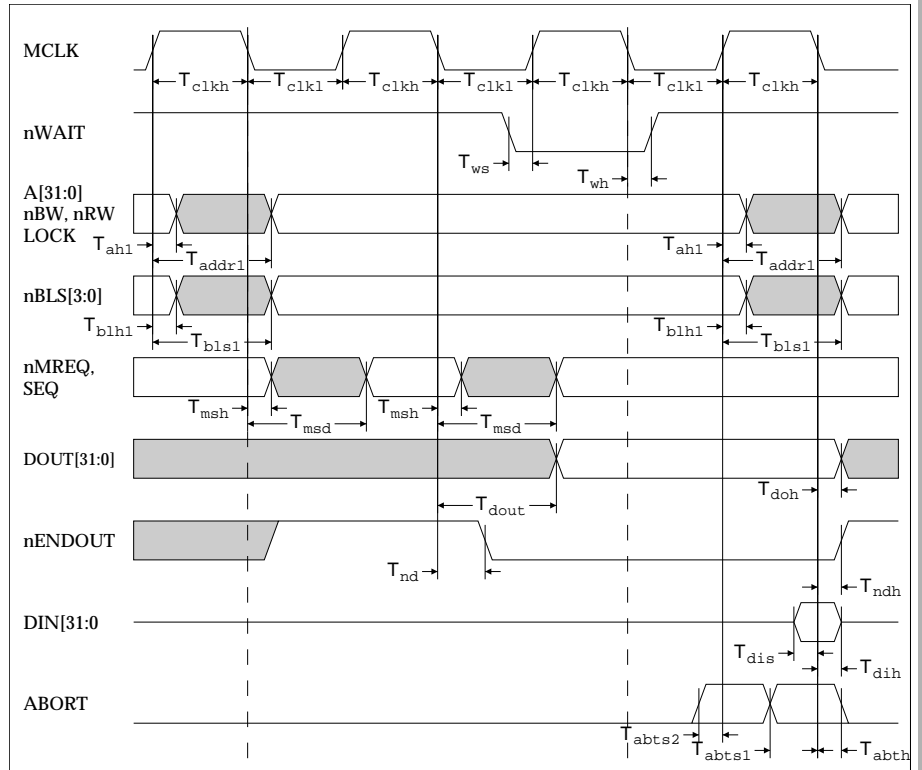


Figure 14-4: ARM710a macrocell nWAIT timing

The following timings are supplied as a guide only. Please refer to your semiconductor vendor for definitive parameters.

Symbol	Parameter	5V Min	5V Max	3.3V Min	3.3V Max	Unit	Note
T_{mckl}	MCLK LOW time					ns	1
T_{mckh}	MCLK HIGH time					ns	
T_{mckc}	MCLK cycle time					ns	1
T_{ws}	nWAIT setup to MCLK					ns	
T_{wh}	nWAIT hold from MCLK					ns	
T_{abe}	address bus enable					ns	2
T_{abz}	address bus disable					ns	
T_{addr1}	MCLK to addr. delay ALE High					ns	2,3
T_{addr2}	MCLK to addr. delay ALE Low					ns	2,4
T_{ah1}	address hold time ALE High					ns	2,3
T_{ah2}	address hold time ALE Low					ns	2,4

Table 14-2: ARM710a macrocell bus timing

AC Parameters with Fastbus Extension

Symbol	Parameter	5V Min	5V Max	3.3V Min	3.3V Max	Unit	Note
Tb1s1	MCLK to byte lane delay ALE HIGH					ns	2,3
Tb1s2	MCLK to byte lane delay ALE LOW					ns	2,4
Tb1h1	byte lane hold time ALE HIGH					ns	2,3
Tb1h2	byte lane hold time ALE LOW					ns	2,4
Tnd	nENDOUT delay					ns	
Tndh	nENDOUT hold time					ns	
Tdbe	DBE to nENDOUT delay					ns	
Tdout	data out delay					ns	2
Tdoh	data out hold					ns	2
Tdis	data in setup					ns	
Tdih	data in hold					ns	
Tabts1	ABORT setup time					ns	5
Tabts2	ABORT setup time					ns	5
Tabth	ABORT hold time					ns	
Tmsd	nMREQ & SEQ delay					ns	
Tmsh	nMREQ & SEQ hold					ns	

Table 14-2: ARM710a macrocell bus timing

Notes

- 1 **MCLK** timings measured between clock edges at 50% of Vdd.
- 2 The timings of these buses are measured to 50% Vdd.
- 3 Address timing with **ALE HIGH**.
- 4 Address timing with **ALE LOW**.
- 5 Tabts1 is required by this device. To ensure compatibility with future ARM processors, we recommend that designs should meet Tabts2. Tabts2 is not tested on this device, and is given as a recommendation only.

A

Differences between ARM610 and ARM710a macrocell

The ARM710a macrocell is designed to be compatible with the ARM610 when used in Standard Mode. This document describes the changes between the ARM710a macrocell operating in this mode, and the ARM610.

A.1 Differences ARM610 and ARM710a macrocell

A-2



Differences between ARM610 and ARM710a macrocell

A.1 Differences ARM610 and ARM710a macrocell

Fastbus input

An extra input has been added to the ARM710a macrocell, the **FASTBUS** input. This configures the device clocking and the operation of **ALE**.

Updateable bit removed from MMU tables

The U (Updateable) bit in the MMU page tables has been removed. This bit is now ignored and will have no effect, all areas are defined as updateable. On the ARM610 this bit allowed Read only, and Write only peripherals to be mapped into the same address space. When the U bit was set the Cache was not updated on Writes.

R bit added to CP0 R15

An extra bit has been added to the Control register (CP15 R0). This is the R or ROM bit. This modifies the MMU permission system to allow ROM emulation for system debugging.

Cache must be flushed when disabled

On the ARM710a macrocell the Cache must be flushed after it has been disabled. The instructions sequence for this is given in the datasheet. The flushing of the Cache is a new requirement from ARM610.

Changed ID code

The ARM710a macrocell has an ID code in the new format. The ID code for ARM710a macrocell is (in hex):

4106710x

Where x is the revision code.

Late abort timing

ARM710a macrocell only supports late abort timing. This changes the programmer's model of the device, as well as the external signal timing. Provided the signal timing recommendations in the ARM610 datasheet were followed the ARM710a macrocell will be compatible with the external abort timing. Software compatability is maintained with ARM610 software which used the late abort timing model. Changes will be required to software which assumed the early abort model.

Enhanced aborts support during linefetches

Any word in a linefetch may now safely be aborted. On ARM610 only the first word in a linefetch could be aborted safely, without corrupting data in the cache.

Spurious addresses may be broadcast

In the case of an internally aborting access, a spurious address may be broadcast externally, but no access will be performed to this location. The memory system should ignore this address.

Differences between ARM610 and ARM710a macrocell

External aborts ignored on buffered writes

The Abort input is now ignored on buffered writes, and will have no affect on the bus cycle. ARM610 allowed external aborts on buffered writes, and generated an non-restartable abort to the CPU.

As a consequence of this, the FSR code 00x0 cannot be generated.

Enlarged cache

The Instruction and Data Cache has been increased in size to 8kB from the 4kB Cache in ARM610.

The Associativity of the Cache has been reduced to 4 way set associative from the 64 way set associative Cache used on ARM610.

When an internal Abort occurs lines may be purged from Cache to remove invalid data.

Enhanced write buffer

The Address section of the Write buffer has been increased in size to contain 4 addresses rather than the 2 address FIFO in ARM610. This will provide improved performance for sequences of short stores, eg Byte operations

The data FIFO is unchanged at 8 entries.

Enhanced TLB

The TLB has been increased to 64 Entries from 32 entries on the ARM610. This will improve performance, and is transparent to the programmer.

Lower voltage operation

The ARM710a macrocell core can operate at 3.3V for reduced power consumption or at 5v for maximum performance, dependent on the semiconductor process used.

nBLS[3:0] outputs added

The nBLS[3:0] outputs are additional functionality to that of ARM610. They are an active low combinatorial decode of A[1:0]. For a word access all will be Low, for a byte access a single bit will be low indicating the selected Byte lane.

Differences between ARM610 and ARM710a macrocell

Index

A

Abort operating mode 3-4
 AC parameters
 in standard mode 13-1
 with fastbus extension 14-1
 Access faults
 checking 9-15
 Address translation 9-4
 ALE pin
 use of 11-14

B

Backward compatibility
 configuration bits 3-3
 with ARM610 A-1
 Branch instructions 4-4
 Bus interface
 asynchronous mode 10-5
 synchronous mode 10-5
 Byte lane Selects
 use of 11-21

C

CDP instruction 4-39
 Condition codes 4-3

Configuration bits
 for backward compatibility 3-3
 Configuration settings
 register 3-2
 Control register
 big endian format 3-2
 little endian format 3-2
 Coprocessor data operations 4-39
 Coprocessor instructions 4-38
 Coprocessors 8-1
 CPSR flags 4-7
 Cycle speed
 bus interface 11-2
 Cycle types
 bus interface 11-3

D

Data processing instructions 4-6
 DC parameters 12-1
 Domain access control 9-14
 Domain access control register 9-3

E

Examples
 instruction set 4-49
 Exceptions 3-7



ARM710a macrocell

abort 3-8
FIQ 3-7
IRQ 3-8
priorities 3-11
External aborts 9-17

F

FAR 9-3, 9-12
Fastbus extension 10-2
Fault address register 9-3, 9-12
Fault checking 9-15
Fault status register 9-3, 9-12
FIQ exception 3-7
FIQ operating mode 3-4
FSR 9-3, 9-12

I

IDC
cacheable bit 6-2
disable 6-3
enable 6-3
interaction with MMU and write buffer 9-18
operation 6-2
read-lock-write 6-3
reset 6-3
validity 6-2
Instruction set examples
loading a halfword 4-52
loading a word from an unknown alignment 4-51
multiply by constant 4-50
pseudo random binary sequence generator 4-50
using conditional instructions 4-49
Instruction set summary 4-2
Instruction speed summary 4-53
Internal coprocessor instructions 5-2
IRQ exception 3-8
IRQ operating mode 3-4

L

LDC instruction 4-41
LDM instruction 4-27
LDR instruction 4-21

M

MCR instruction 4-45
Memory access
types of 11-23
use of byte lane selects 11-21
use of the ALE pin 11-14
use of the nWAIT pin 11-13
MLA instruction 4-19
MMU 9-1
interaction with IDC and write buffer 9-18
MRC instruction 4-45
MRS instruction 4-15
MSR instruction 4-15
MUL instruction 4-19

N

nWAIT pin
use of 11-13

O

Operating modes
selecting 3-4

P

Parameters
AC in standard mode 13-1
AC with Fastbus extension 14-1
DC 12-1

R

Register configurations 3-2
Registers 3-4, 5-3
MMU 9-3

S

Shifts 4-9
Signal descriptions 2-2
Software interrupt instruction 3-9, 4-36
STC instruction 4-41
STM instruction 4-27

STR instruction 4-21
Supervisor operating mode 3-4
SWP instruction 4-34

T

Translating references 9-5
Translation table base register 9-3

U

Undefined instruction 4-48
Undefined instruction trap 3-10
Undefined operating mode 3-4
User operating mode 3-4

W

Write buffer 7-1
 interaction with MMU and IDC 9-18



