ARM710

Data Sheet

Document Number: ARM DDI 0024E

Issued: Dec 1994

Change Log

Issue	Date	By	Change
A	July 1993	TP	Unified FrameMaker version created.
B	Sept 1993	PM	Corrected TOC.
C	July 1994	EH	General corrections.
D	Aug 94	BJH	Updated linefetch diagram.
E	Dec 94	PB	Edited.

Preface

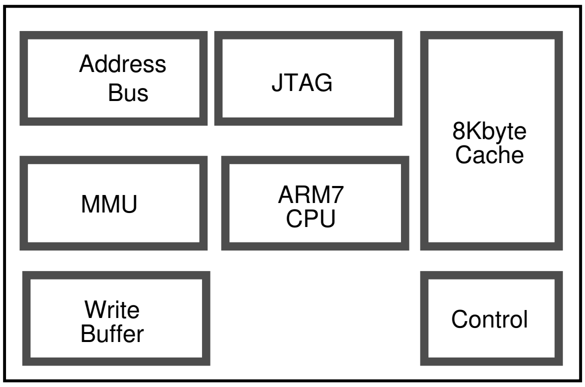
The ARM710 is a general purpose 32-bit microprocessor with 8kByte cache, write buffer and Memory Management Unit (MMU) combined in a single chip. The ARM710 offers high-level RISC performance yet its fully static design ensures minimal power consumption, making it ideal for portable, low-cost systems.

The innovative 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. This results in a high instruction throughput and impressive real-time interrupt response from a small and cost-effective chip.

Applications

- *Personal computer devices e.g.PDAs*
- *High-performance, real-time control systems*
- *Portable telecommunications*
- *Data communications equipment*
- *Consumer products*
- *Automotive*

Feature Summary



- *High performance RISC*
  25 MIPS sustained @ 33 MHz (33 MIPS peak)
- *Fast sub-microsecond interrupt response*
  for real-time applications
- *Memory Management Unit (MMU)*
  support for virtual memory systems
- *Excellent high-level language support*
- *8kByte of instruction & data cache*
- *Big and Little Endian operating modes*
- *Write Buffer*
  enhancing performance
- *IEE 1149.1 Boundary scan*
- *Fully static operation, low power consumption*
  ideal for power sensitive applications
- *144 Thin Quad Flat Pack (TQFP) package*
- *Low-power CMOS process*
  1.5mA/MHz@3.3V
- *3V and 5V operation*

Table of Contents

Table of Contents

1.0 Introduction

ARM710 is a general purpose 32-bit microprocessor with 8kByte cache, enlarged write buffer and Memory Management Unit (MMU) combined in a single chip. The CPU within ARM710 is the ARM7. The ARM710 is software compatible with the ARM processor family and can be used with ARM support chips.

The ARM710 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 on-chip 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 memory 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.

ARM710 is a fully static part 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

ARM710 Data Sheet

ARM710 is a variant of the ARM700, differing from that device in the following respects:

- no external coprocessor bus interface
- dedicated chip test port added
- device packaging

ARM710 is an enhanced and updated ARM610, differing from that device in the following respects:

- cache size increased from 4kB to 8kB
- increased maximum clock frequency
- improved write buffer
- enlarged Translation Lookaside Buffer (TLB) in MMU

1.1 Block Diagram

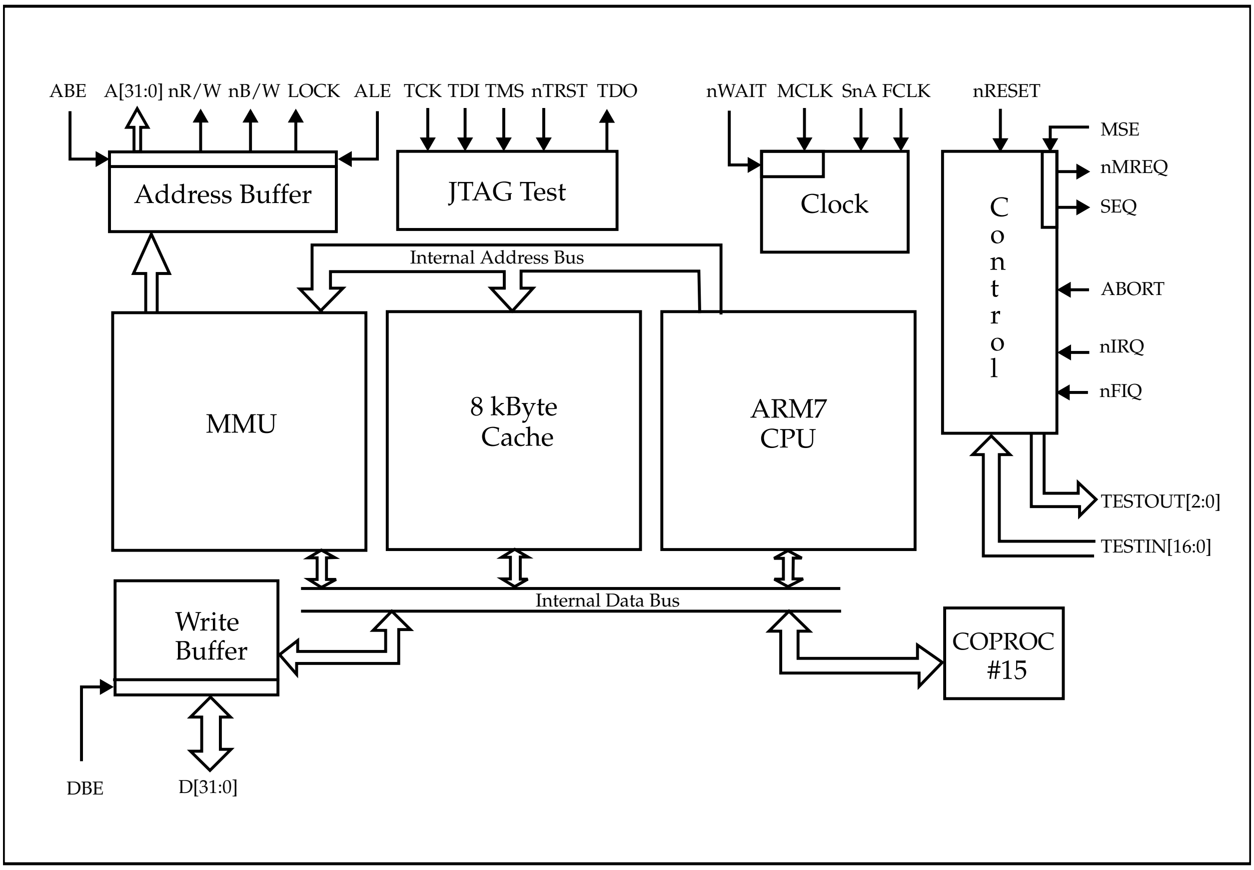


Figure 1: ARM710 Block Diagram

1.2 Functional Diagram

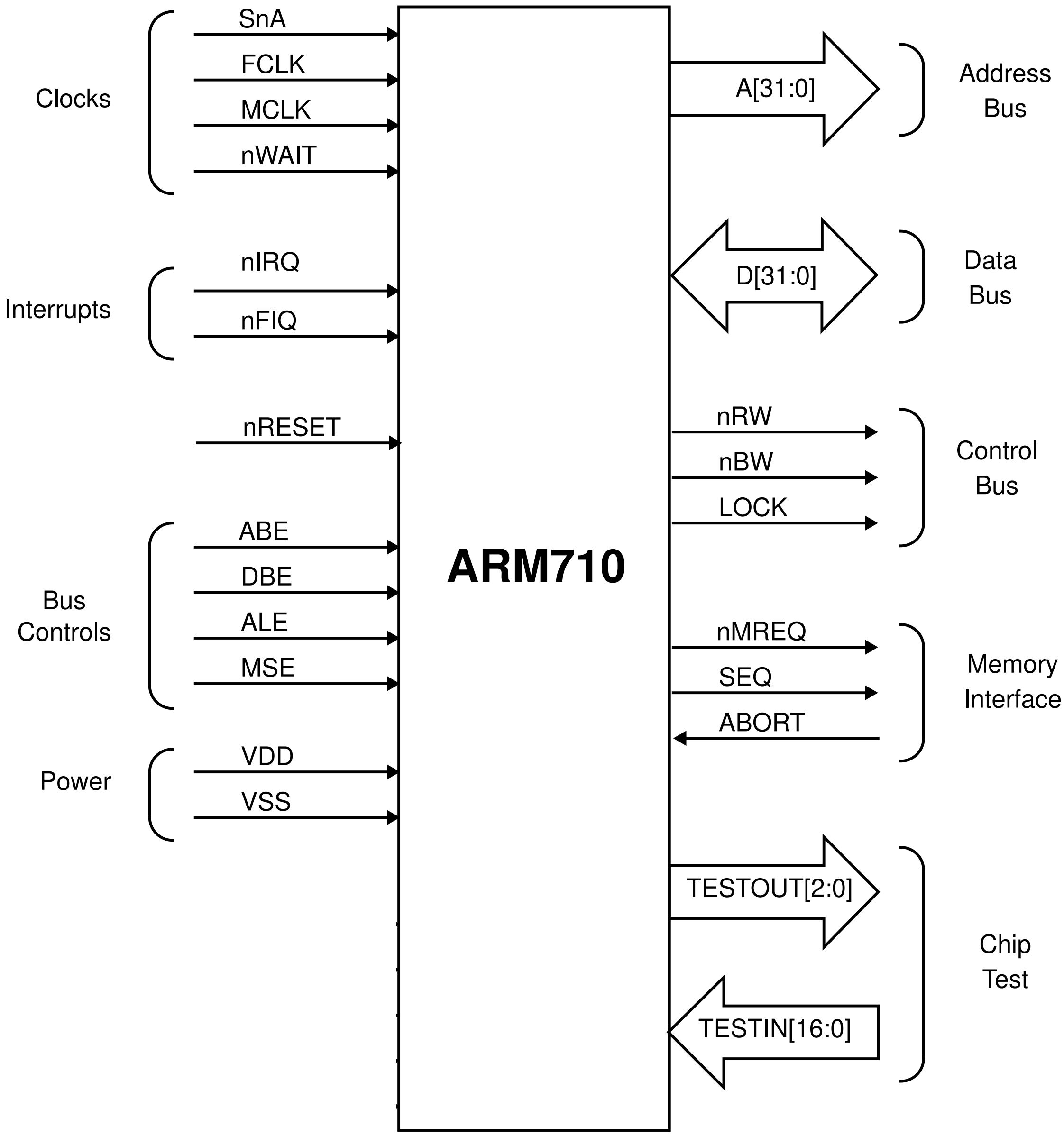


Figure 2: Functional Diagram

2.0 Signal Description

Name	Type	Description
A[31:0]	OCZ	Address Bus. This bus signals the address requested for memory accesses. Normally it changes during **MCLK** HIGH.
ABE	IC	Address bus enable. When this input is LOW, the address bus **A[31:0]**, **nRW**, **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 ARM710 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**, **nRW** & **LOCK**. Normally these signals change during **MCLK** HIGH, but they may be held by driving **ALE** LOW. See *Section 13.2.1: Tald Measurement* on page 120.
D[31:0]	ICOCZ	Data bus. These are bi-directional signal paths used for data transfers between the processor and external memory. For read operations (when **nRW** is LOW), the input data must be valid before the falling edge of **MCLK**. For write operations (when **nRW** is HIGH), the output data will become valid while **MCLK** is LOW. At high clock frequencies the data may not become valid until just after the MCLK rising edge (see *Section 13.3: Main Bus Signals* on page 121).
DBE	IC	Data bus enable. When this input is LOW, the data bus, **D[31:0]** is put into a high impedance state (Note 1). The drivers will always be high impedance except during write operations, and **DBE** must be driven HIGH in systems which do not require the data bus for DMA or similar activities.
FCLK	ICK	Fast clock input. When the ARM710 CPU is accessing the cache or performing an internal cycle, it is clocked with the Fast Clock, **FCLK**.
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 ARM710 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.
MSE	IC	Memory request/sequential enable. When this input is LOW, the **nMREQ** and **SEQ** outputs are put into a high impedance state (Note 1).
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**.
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.

Table 1: Signal Descriptions

Name	Type	Description
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	OCZ	Not memory request. A pipelined signal that changes while **MCLK** is LOW to indicate whether or not in the following cycle, the processor will be accessing external memory. When **nMREQ** is LOW, the processor will be accessing external memory
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 which ever 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**.
nTRST	IC	Test interface reset. Note this signal does NOT have an internal pullup resistor. This signal must be pulsed or driven LOW to achieve normal device operation, in addition to the normal device reset (**nRESET**).
nWAIT	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	OCZ	Sequential address. This signal is the inverse of **nMREQ**, and is provided for compatibility with existing ARM memory systems.
SnA	IC	Synchronous / not Asynchronous. 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 10.0: Bus Interface.*
TEST IN[16:0]	IC	Test bus input. This bus is used for off-board testing of the device. When the device is fitted to a circuit all these signals must be tied LOW.
TEST OUT[2:0]	OCZ	Test bus output. This bus is used for off-board testing of the device. When the device is fitted to a circuit and all the **TESTIN[16:0]** signals are driven LOW, these three outputs will be driven LOW. Note that these signals may not be tristated, except via the JTAG test port.
TCK	IC	Test interface reference Clock. This times all the transfers on the JTAG test interface.
TDI	IC	Test interface data input. Note this signal does *not* have an internal pullup resistor.
TDO	OCZ	Test interface data output. Note this signal does *not* have an internal pullup resistor.
TMS	IC	Test interface mode select. Note this signal does *not* have an internal pullup resistor.
VDD		Positive supply. 15 pins are allocated to **VDD** in the 144 PQFP package.
VSS		Ground supply. 15 pins are allocated to **VSS** in the 144 PQFP package.

Table 1: Signal Descriptions

Notes:

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

Key to Signal Types: **IC** - Input, CMOS threshold
OCZ - Output, CMOS levels, tri-stateable
ICOCZ - Input/output tri-stateable, CMOS thresholds
ICK - Clock input, CMOS levels

3.0 Programmer's Model

ARM710 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 ARM710 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 ARM710 treats words in memory as being stored in Big Endian or Little Endian format. See *Chapter 5.0: 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.

Little Endian

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

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

Figure 3: Little Endian addresses of bytes within words

In the Big Endian scheme the most significant byte of a word is stored at the lowest numbered byte and the least significant byte is stored at the highest numbered byte. Byte 0 of the memory system should therefore be connected to data lines 31 through 24 (**D[31:24]**). Load and store are the only instructions affected by the endian-ness: see *Section 4.7: Single data transfer (LDR, STR)* on page 36 for more details.

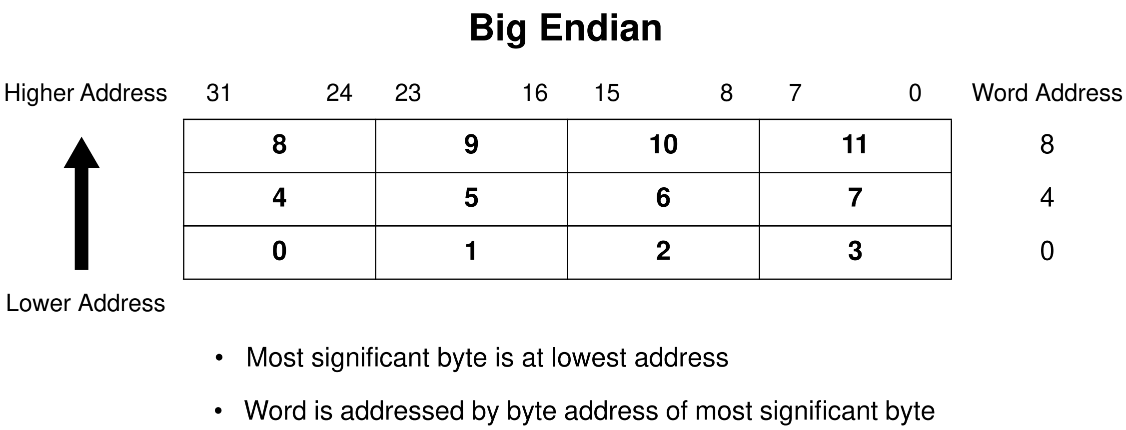


Figure 4: Big Endian addresses of bytes within words

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 (see *16.0: Appendix - Backward Compatibility*) but should normally be set to 1. This configuration extends the address space to 32 bits, introduces major changes in the programmer's model as described below, and provides support for running existing 26 bit programs in the 32 bit environment. This mode is recommended for compatibility with future ARM processors and all new code should be written to use only the 32 bit operating modes.

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

3.2 Operating Mode Selection

ARM710 has a 32 bit data bus and a 32 bit address bus. The processor supports *byte* (8 bit) and *word* (32 bit) data tyoes, 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.

ARM710 supports six modes of operation:

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

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

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

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

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

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

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

3.3 Registers

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

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

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

General Registers and Program Counter Modes

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

Program Status Registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

Figure 5: Register Organisation

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

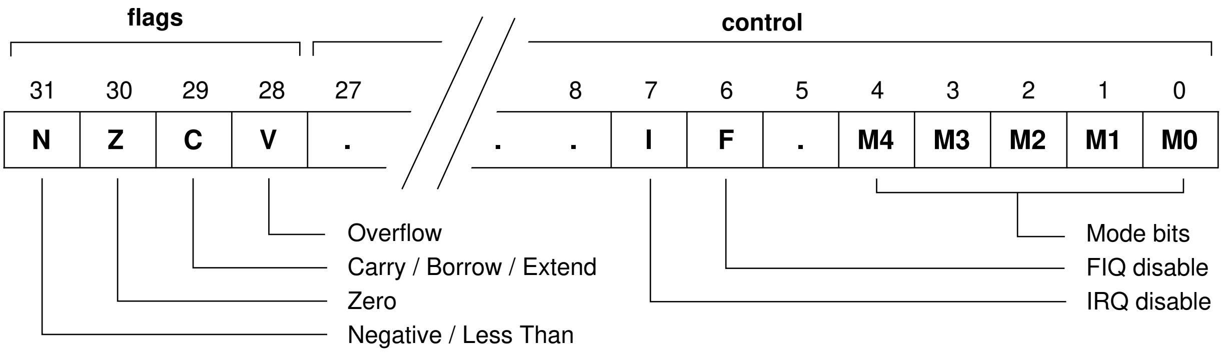


Figure 6: Format of the Program Status Registers (PSRs)

The format of the Program Status Registers is shown in *Figure 6: Format of the Program Status Registers (PSRs)*. The N, Z, C and V bits are the *condition code flags*. The condition code flags in the CPSR may be changed as a result of arithmetic and logical operations in the processor and may be tested by all instructions to determine if the instruction is to be executed.

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

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

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

Table 2: The Mode Bits

3.4 Exceptions

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

ARM710 handles exceptions by making use of the banked registers to save state. The old PC and CPSR contents are copied into the appropriate R14 and SPSR and the PC and mode bits in the CPSR bits are forced to a value which depends on the exception. Interrupt disable flags are set where required to prevent otherwise unmanageable nestings of exceptions. In the case of a re-entrant interrupt handler, R14 and the SPSR should be saved onto a stack in main memory before re-enabling the interrupt; when transferring the SPSR register to and from a stack, it is important to transfer the whole 32 bit value, and not just the flag or control fields. When multiple exceptions arise simultaneously, a fixed priority determines the order in which they are handled. This is listed later in *Section 3.4.7: Exception Priorities* on page 17.

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, ARM710 checks for a LOW level on the output of the FIQ synchroniser at the end of each instruction.

When a FIQ is detected, ARM710:

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

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

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

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

3.4.2 IRQ

The IRQ (Interrupt ReQuest) exception is a normal interrupt caused by a LOW level on the **nIRQ** input. It has a lower priority than FIQ, and is masked out when a FIQ sequence is entered. Its effect may be masked out at any time by setting the I bit in the CPSR (but note that this is not possible from User mode). If the I flag is clear, ARM710 checks for a LOW level on the output of the IRQ synchroniser at the end of each instruction. When an IRQ is detected, ARM710:

(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. **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. ARM710 checks for aborts during memory access cycles. When successfully aborted ARM710 will respond in one of two ways:

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

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

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

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

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

When either a prefetch or data abort occurs, ARM710:

(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 that there are restrictions on the use of the external abort signal. See *Chapter 9.0: Memory Management Unit (MMU)*.

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, ARM710:

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

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

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

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

3.4.5 Undefined instruction trap

When the ARM710 comes across an instruction which it cannot handle (see *Chapter 4.0: 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 ARM710 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: 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), ARM710 will enter the data abort handler and then immediately proceed to the FIQ vector. A normal return from FIQ will cause the data abort handler to resume execution. Placing data abort at a higher priority than FIQ is necessary to ensure that the transfer error does not escape detection; the time for this exception entry should be added to worst case FIQ latency calculations.

3.4.8 Interrupt Latencies

Calculating the worst case interrupt latency for the ARM710 is quite complex due to the cache, MMU and write buffer and is dependant on the configuration of the whole system. Please see *Application Note - Calculating the ARM710 Interrupt Latency*.

3.5 Reset

When the **nRESET** signal goes LOW, ARM710 abandons the executing instruction and then performs idle cycles from incrementing word addresses.

When **nRESET** goes HIGH again, ARM710 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.

4.0 Instruction Set

4.1 Instruction Set Summary

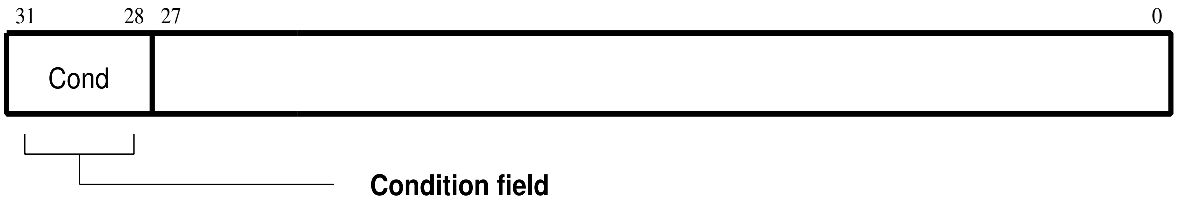
A summary of the ARM710 instruction set is shown in *Figure 7: Instruction Set Summary*.

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 7: Instruction Set Summary

4.2 The Condition Field



Condition field

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

Figure 8: Condition Codes

All ARM710 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 8: 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 8: Condition Codes*, for instance code 0000 (EQual) causes the instruction to be executed only if the Z flag is set. This would correspond to the case where a compare (CMP) instruction had found the two operands to be equal. If the two operands were different, the compare instruction would have cleared the Z flag and the instruction will not be executed.

4.3 Branch and Branch with link (B, BL)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 9: Branch Instructions*.

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

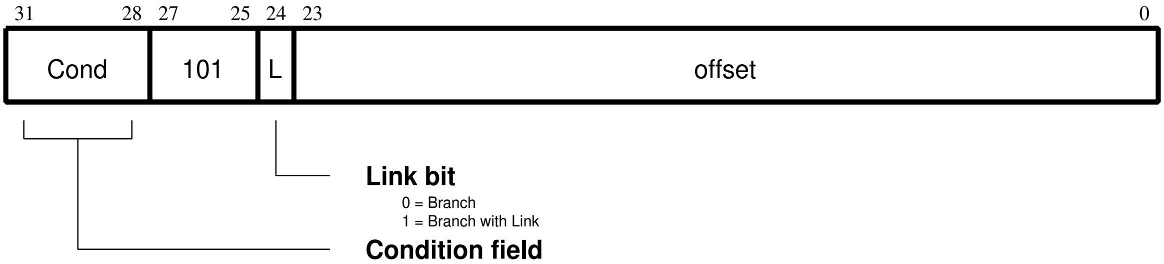


Figure 9: Branch Instructions

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 *Section 4.17: Instruction Speed Summary* on page 64.

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 8: Condition Codes* (EQ, NE, VS etc). If absent then AL (ALways) will be used.

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

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

4.3.4 Examples

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

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

      BL       sub+ROM  ; call subroutine at computed address

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

4.4 Data processing

The instruction is only executed if the condition is true, defined at the beginning of this chapter. The instruction encoding is shown in *Figure 10: Data Processing Instructions*.

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

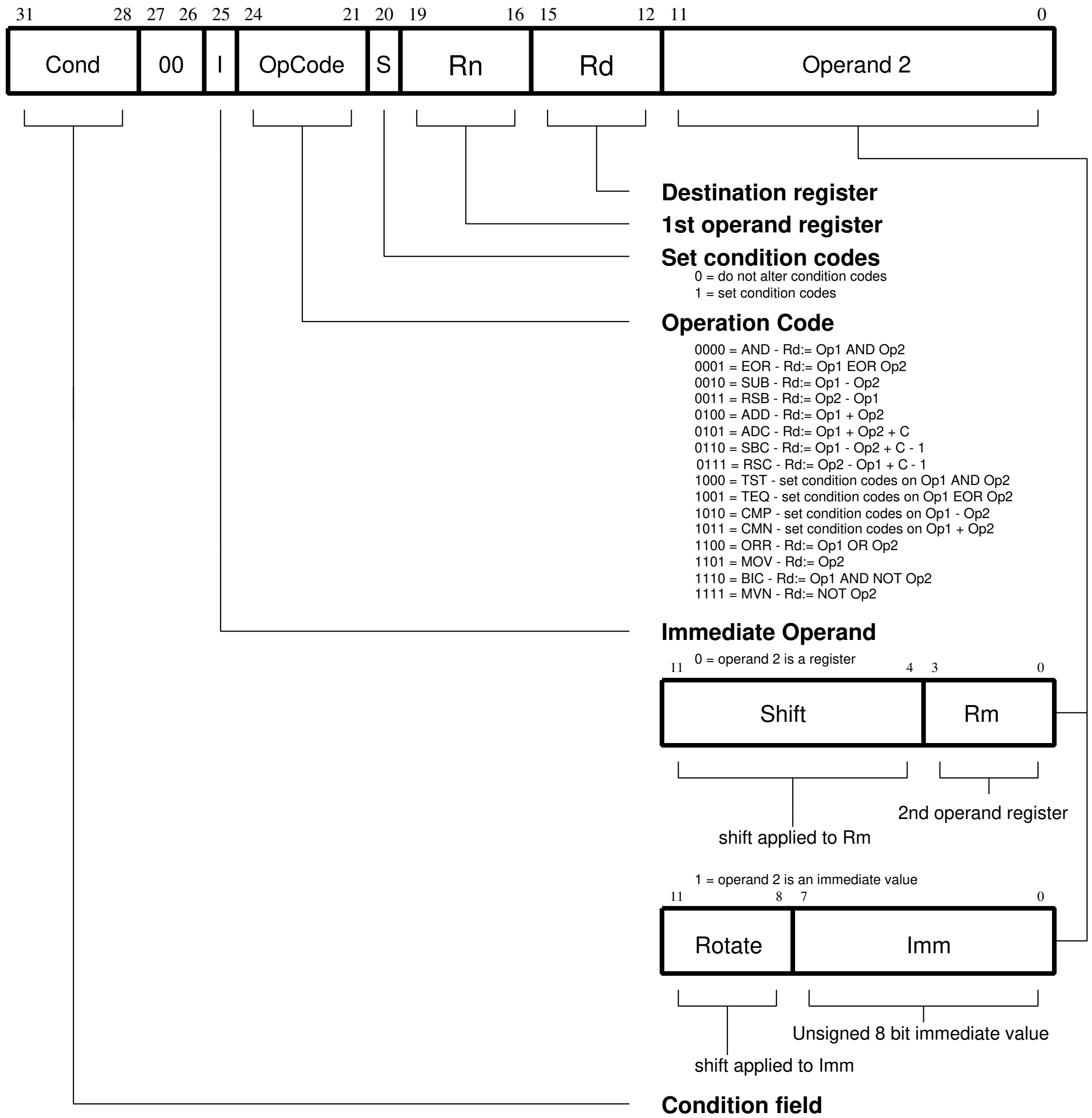


Figure 10: Data Processing Instructions

4.4.1 CPSR flags

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

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

Table 4: ARM Data Processing Instructions

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

4.4.2 Shifts

When the second operand is specified to be a shifted register, the operation of the barrel shifter is controlled by the Shift field in the instruction. This field indicates the type of shift to be performed (logical left or right, arithmetic right or rotate right). The amount by which the register should be shifted may be contained in an immediate field in the instruction, or in the bottom byte of another register (other than R15). The encoding for the different shift types is shown in *Figure 11: ARM Shift Operations*.

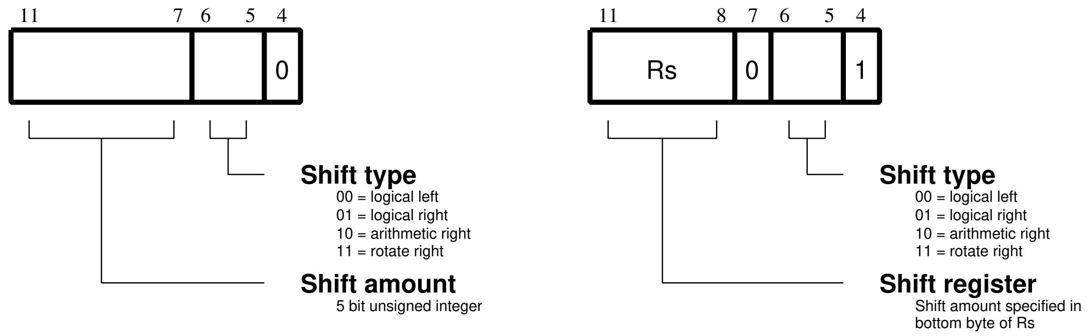


Figure 11: ARM Shift Operations

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 12: Logical Shift Left*.

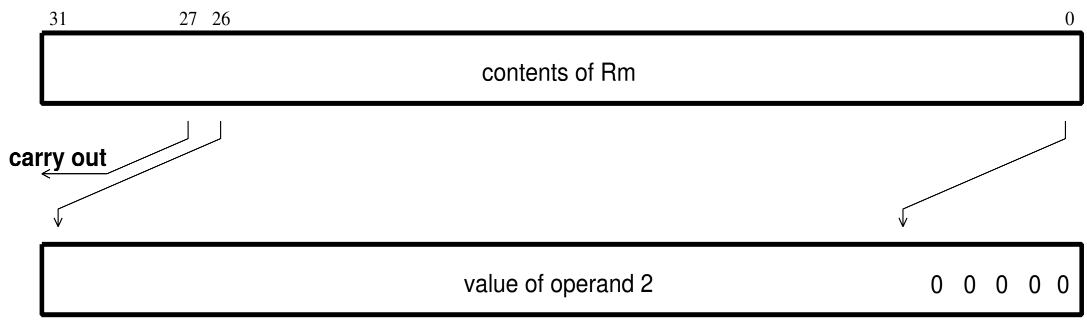


Figure 12: Logical Shift Left

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

A logical shift right (LSR) is similar, but the contents of Rm are moved to less significant positions in the result. LSR #5 has the effect shown in *Figure 13: Logical Shift Right*.

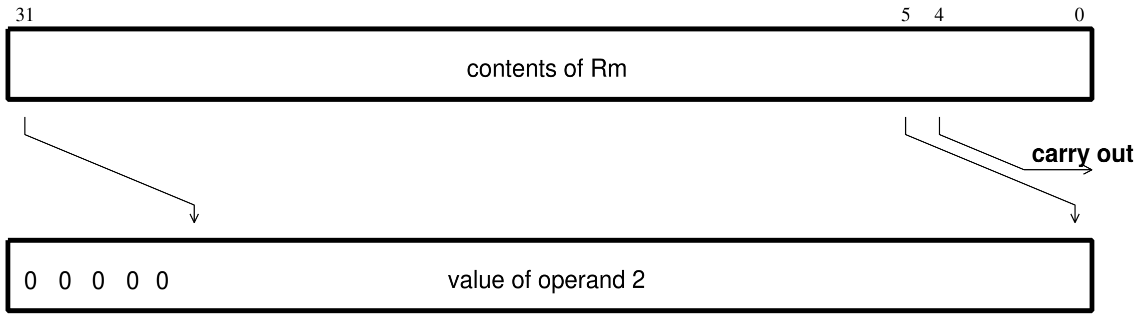


Figure 13: Logical Shift Right

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 14: Arithmetic Shift Right*.

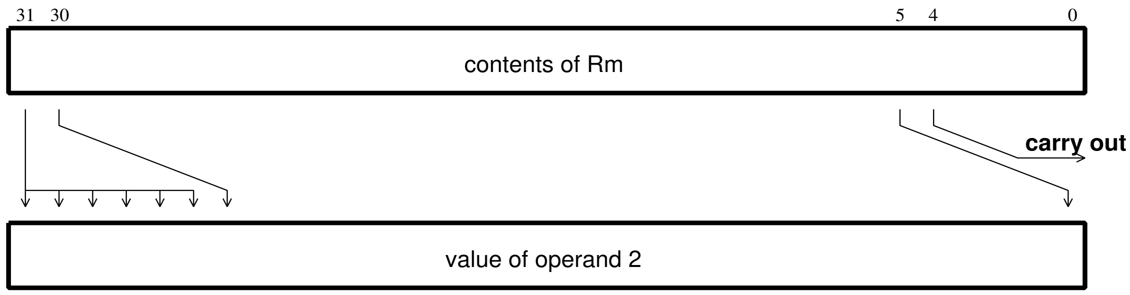


Figure 14: Arithmetic Shift Right

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 15: Rotate Right*.

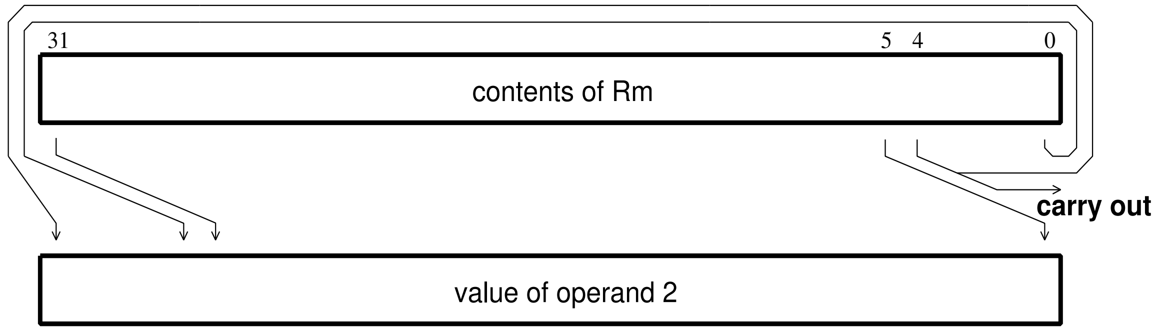


Figure 15: Rotate Right

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 16: Rotate Right Extended*.

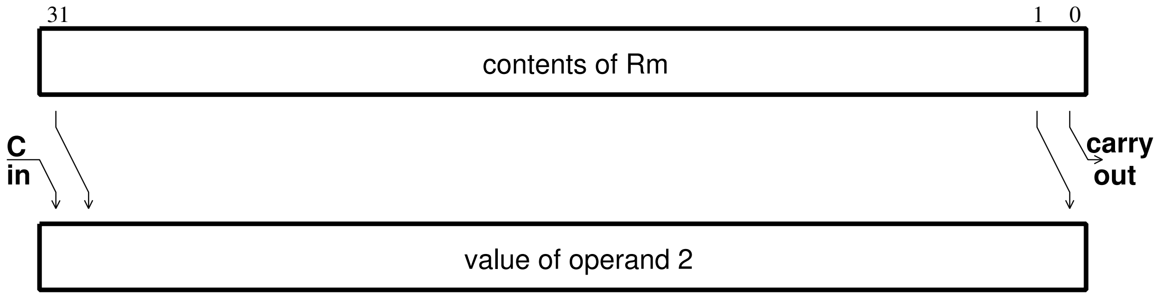


Figure 16: Rotate Right Extended

Register specified shift amount

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

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

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

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

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

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

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

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

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

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

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

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

4.4.3 Immediate operand rotates

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

4.4.4 Writing to R15

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

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

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

4.4.5 Using R15 as an operand

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

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

4.4.6 TEQ, TST, CMP & CMN opcodes

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

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

4.4.7 Instruction Cycle Times

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

Normal Data Processing	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 *Section 4.17: Instruction Speed Summary* on page 64 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 8: Condition Codes*

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

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

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

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

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

4.4.9 Examples

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

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

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

MOV     PC,R14              ; return from subroutine

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

4.5 PSR Transfer (MRS, MSR)

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

The MRS and MSR instructions are formed from a subset of the Data Processing operations and are implemented using the TEQ, TST, CMN and CMP instructions without the S flag set. The encoding is shown in *Figure 17: PSR Transfer*.

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

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

4.5.1 Operand restrictions

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

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

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

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

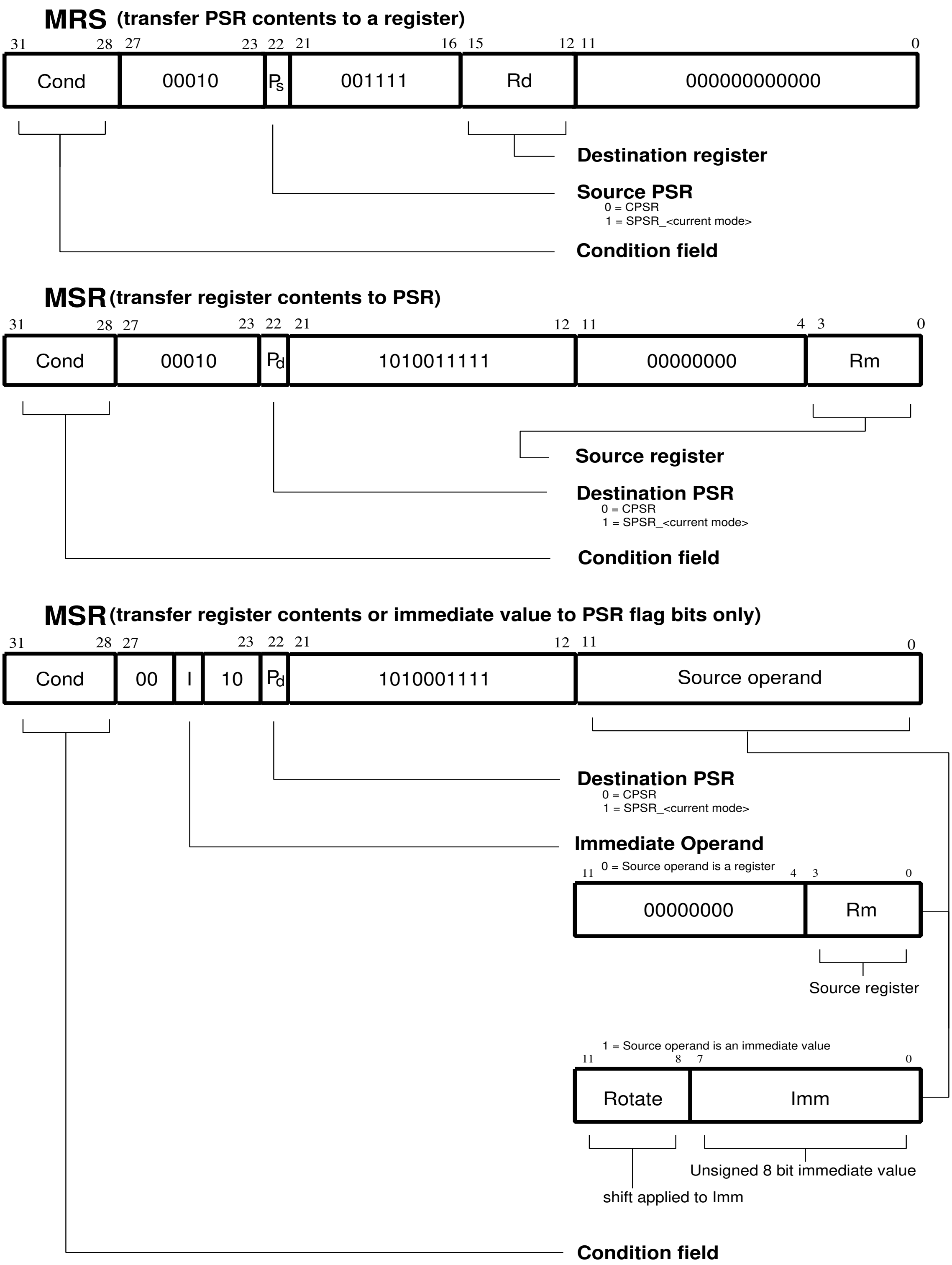


Figure 17: PSR Transfer

4.5.2 Reserved bits

Only eleven bits of the PSR are defined in ARM710 (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 ARM710 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 *Section 4.17: Instruction Speed Summary* on page 64.

4.5.4 Assembler syntax

(1) MRS - transfer PSR contents to a register

MRS{cond} Rd,<psr>

(2) MSR - transfer register contents to PSR

MSR{cond} <psr>,Rm

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

MSR{cond} <psrf>,Rm

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

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

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

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

{cond} - two-character condition mnemonic, see *Figure 8: Condition Codes*

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

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

<psrf> is CPSR_flg or SPSR_flg

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

4.5.5 Examples

In User mode the instructions behave as follows:

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

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

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

In privileged modes the instructions behave as follows:

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

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

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

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

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

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

4.6 Multiply and Multiply-Accumulate (MUL, MLA)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 18: Multiply Instructions*.

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

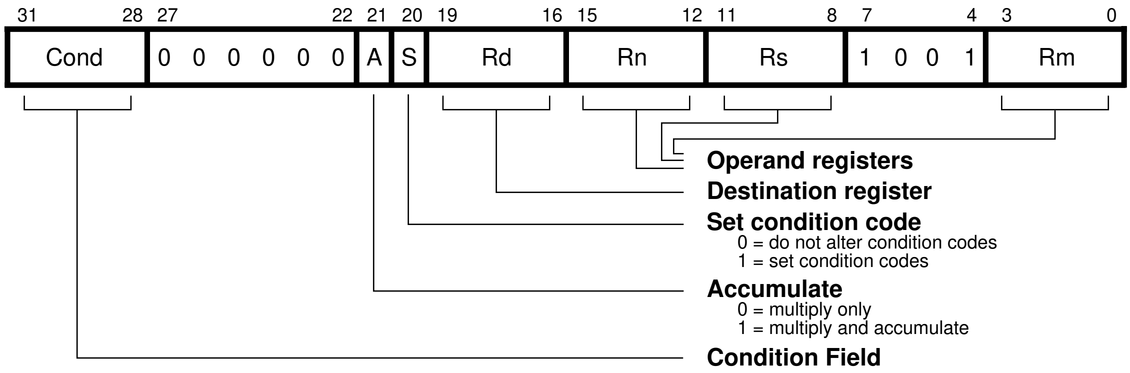


Figure 18: Multiply Instructions

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

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

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

4.6.1 Operand Restrictions

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

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

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

4.6.2 CPSR flags

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

4.6.3 Instruction Cycle Times

The Multiply instructions take 1 instruction fetch and m internal cycles. For more information see section 4.17 Instruction Speed Summary on page 64.

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 8: Condition Codes*

{S} - set condition codes if S present

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

4.6.5 Examples

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

4.7 Single data transfer (LDR, STR)

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

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

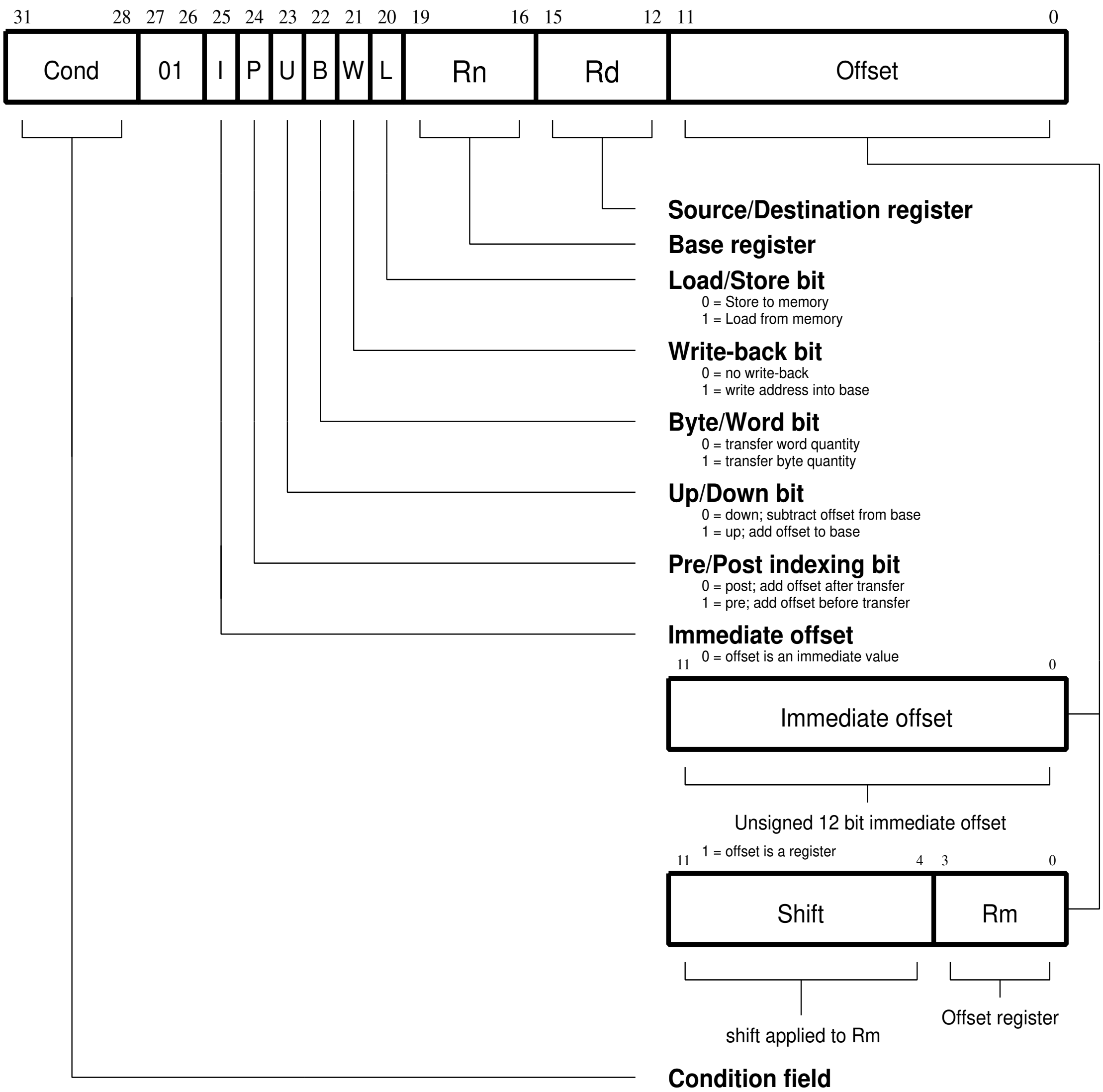


Figure 19: Single Data Transfer Instructions

4.7.1 Offsets and auto-indexing

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

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

4.7.2 Shifted register offset

The 8 shift control bits are described in the data processing instructions section. However, the register specified shift amounts are not available in this instruction class. See *Section 4.4.2: Shifts* on page 25.

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 ARM710 register and memory.

The action of LDR(B) and STR(B) instructions is influenced by the 3 instruction fetches. For more information see *Section 4.17: Instruction Speed Summary* on page 64. The two possible configurations are described below.

Little Endian Configuration

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

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

A word load (LDR) 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 20: Little Endian Offset Addressing*.

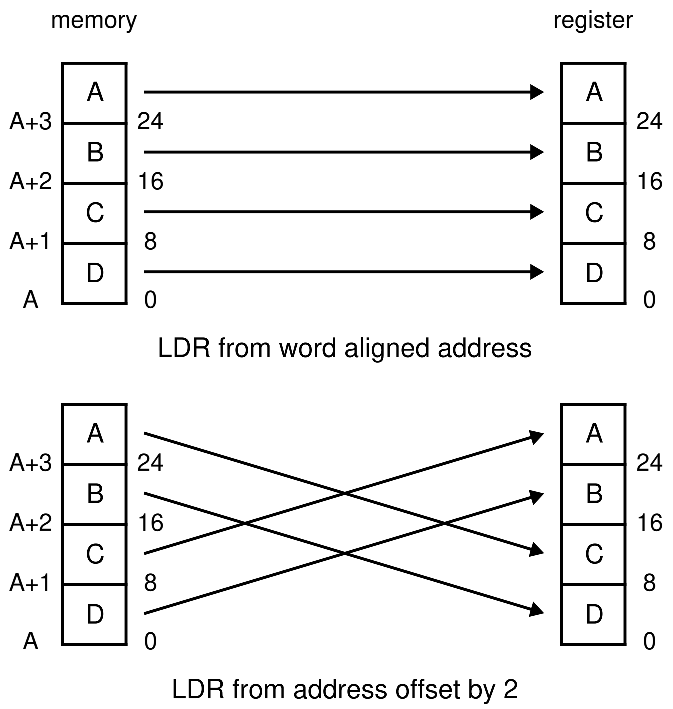


Figure 20: Little Endian Offset Addressing

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 4: Big Endian addresses of bytes within words*.

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

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

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

4.7.4 Use of R15

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

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

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

4.7.5 Restriction on the use of base register

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

For example:

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

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

4.7.6 Data Aborts

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

4.7.7 Instruction Cycle Times

Normal LDR instructions take 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 *Section 4.17: Instruction Speed Summary* on page 64.

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 8: Condition Codes*

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

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

Rd is an expression evaluating to a valid register number.

<Address> can be:

(i) An expression which generates an address:

<expression>

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

(ii) A pre-indexed addressing specification:

[Rn] offset of zero

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

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

(iii) A post-indexed addressing specification:

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

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

Rn and Rm are expressions evaluating to a register number. If Rn is R15 then the assembler will subtract 8 from the offset value to allow for ARM710 pipelining. In this case base write-back shall not be specified.

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

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

4.7.9 Examples

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

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

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

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

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

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

4.8 Block Data Transfer (LDM, STM)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 21: Block Data Transfer Instructions*.

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

4.8.1 The Register List

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

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

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

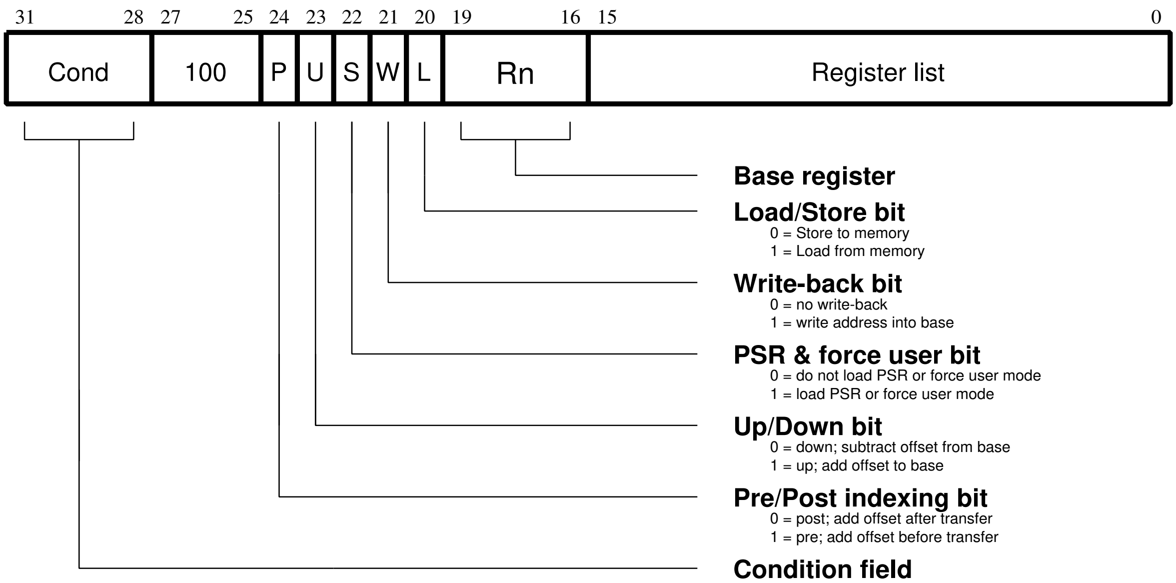


Figure 21: Block Data Transfer Instructions

4.8.2 Addressing Modes

The transfer addresses are determined by the contents of the base register (Rn), the pre/post bit (P) and the up/down bit (U). The registers are transferred in the order lowest to highest, so R15 (if in the list) will always be transferred last. The lowest register also gets transferred to/from the lowest memory address. By way of illustration, consider the transfer of R1, R5 and R7 in the case where Rn=0x1000 and write back of the modified base is required (W=1). Figures *22*, *23*, *24* and *25* show the sequence of register transfers, the addresses used, and the value of Rn after the instruction has completed.

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

4.8.3 Address Alignment

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

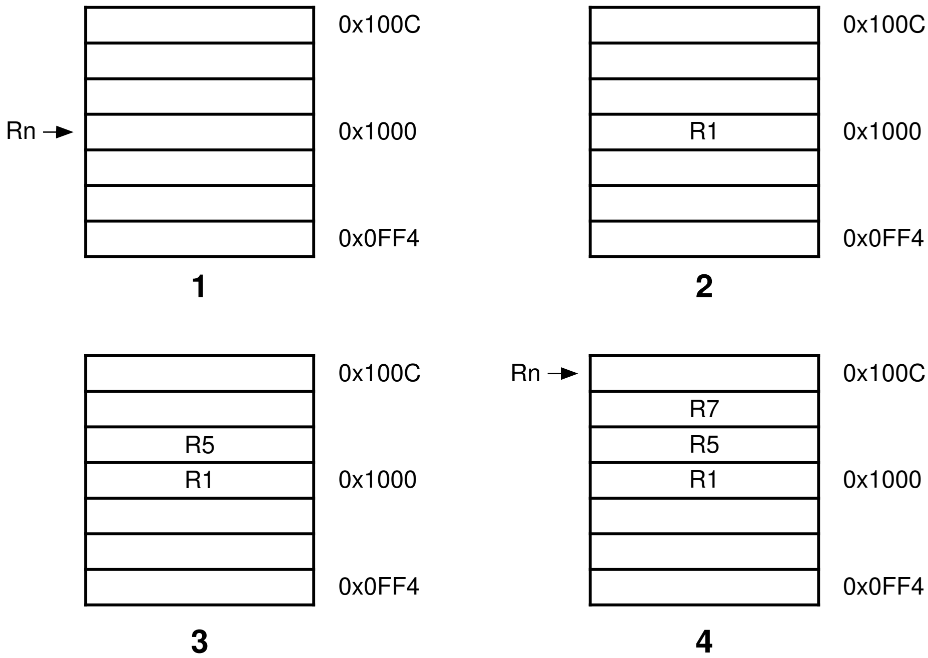


Figure 22: Post-increment addressing

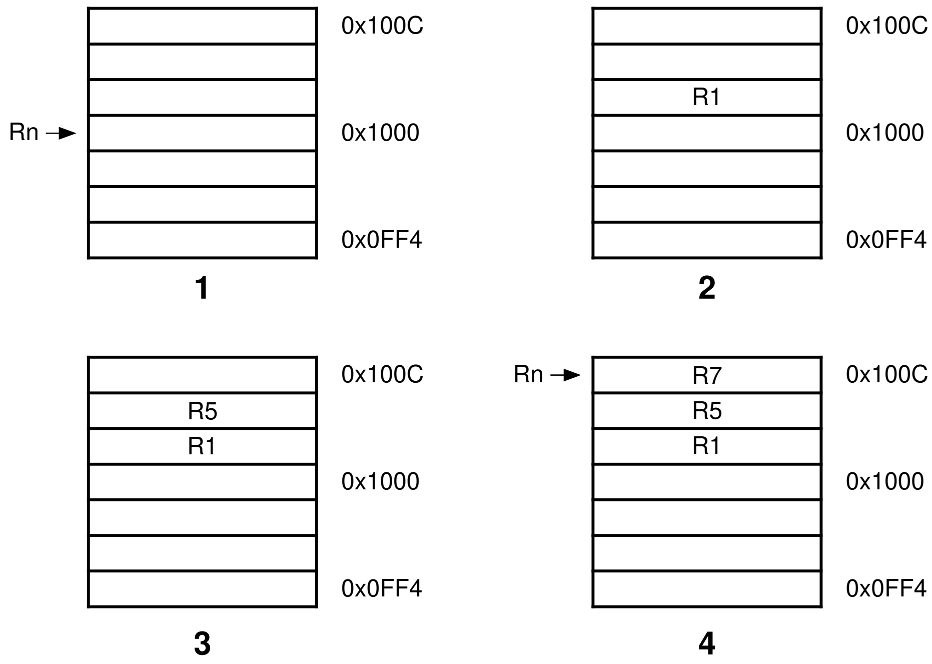


Figure 23: Pre-increment addressing

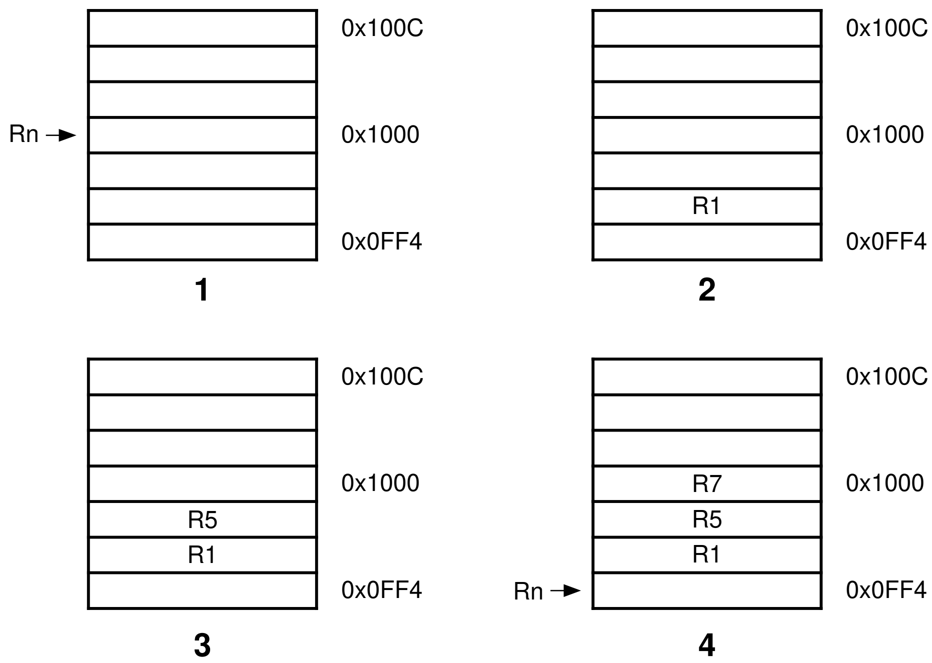


Figure 24: Post-decrement addressing

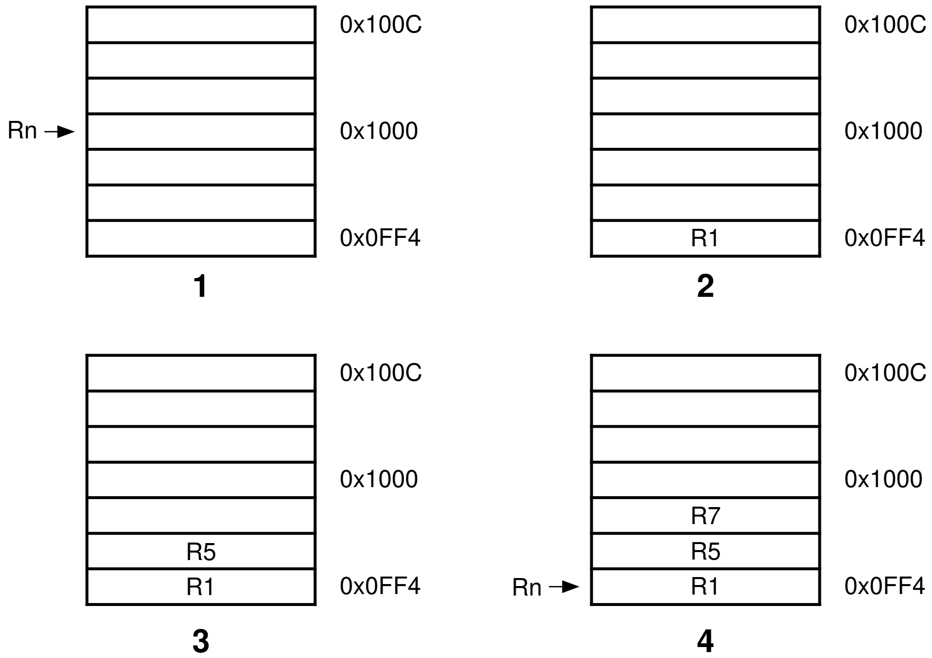


Figure 25: Pre-decrement addressing

4.8.4 Use of the S bit

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

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

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

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

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

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

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

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

4.8.5 Use of R15 as the base

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

4.8.6 Inclusion of the base in the register list

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

4.8.7 Data Aborts

Some legal addresses may be unacceptable to a memory management system, and the memory manager can indicate a problem with an address by taking the **ABORT** signal HIGH. This can happen on any transfer during a multiple register load or store, and must be recoverable if ARM710 is to be used in a virtual memory system.

Aborts during STM instructions

If the abort occurs during a store multiple instruction, ARM710 takes little action until the instruction completes, whereupon it enters the data abort trap. The memory manager is responsible for preventing erroneous writes to the memory. The only change to the internal state of the processor will be the modification of the base register if write-back was specified, and this must be reversed by software (and the cause of the abort resolved) before the instruction may be retried.

Aborts during LDM instructions

When ARM710 detects a data abort during a load multiple instruction, it modifies the operation of the instruction to ensure that recovery is possible.

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

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

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

4.8.8 Instruction Cycle Times

Normal LDM instructions take 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 *Section 4.17: Instruction Speed Summary* on page 64.

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 8: Condition Codes*

Rn is an expression evaluating to a valid register number

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

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

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

Addressing mode names

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

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

Table 5: Addressing Mode Names

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

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

4.8.10 Examples

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

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

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

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

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

BL      somewhere           ; this nested call will overwrite R14

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

4.9 Single data swap (SWP)

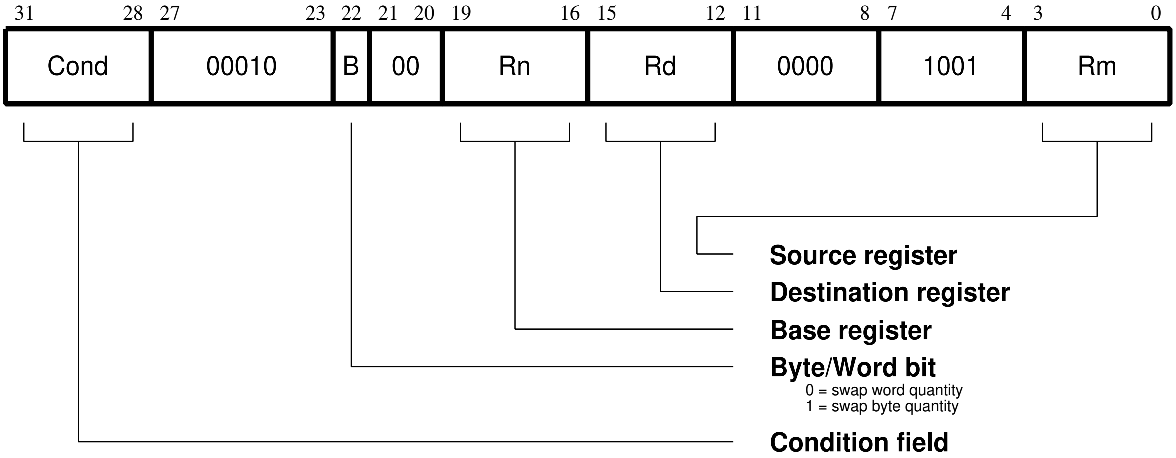


Figure 26: Swap Instruction

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 26: 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 ARM710 register and memory. The SWP instruction is implemented as a LDR followed by a STR and the action of these is as described in the section on single data transfers. In particular, the description of Big and Little Endian configuration applies to the SWP instruction.

4.9.2 Use of R15

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

4.9.3 Data Aborts

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

4.9.4 Instruction Cycle Times

Swap instructions take 1 instruction fetch, 1 data read, 1 data write and 1 internal cycle. For more information see *Section 4.17: Instruction Speed Summary* on page 64.

4.9.5 Assembler syntax

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

{cond} - two-character condition mnemonic, see *Figure 8: Condition Codes*

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

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

4.9.6 Examples

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

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

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

4.10 Software interrupt (SWI)

Figure 27: 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 27: Software Interrupt Instruction*.

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

4.10.1 Return from the supervisor

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

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

4.10.2 Comment field

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

4.10.3 Instruction Cycle Times

Software interrupt instructions take 3 instruction fetches. For more information see *Section 4.17: Instruction Speed Summary* on page 64.

4.10.4 Assembler syntax

SWI{cond} <expression>

{cond} - two character condition mnemonic, see *Figure 8: Condition Codes*

<expression> is evaluated and placed in the comment field (which is ignored by ARM710).

4.10.5 Examples

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

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

```
0x08 B Supervisor           ; SWI entry point

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

Zero     EQU  0
ReadC    EQU  256
WriteI   EQU  512

Supervisor

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

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

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

4.11 Coprocessor Instructions on ARM710

The ARM710, unlike some other ARM processors, does not have an external coprocessor interface. The ARM710 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 ARM710 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 ARM710, 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 ARM710 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 28: 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 ARM710, 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 ARM710 to perform independent tasks in parallel.

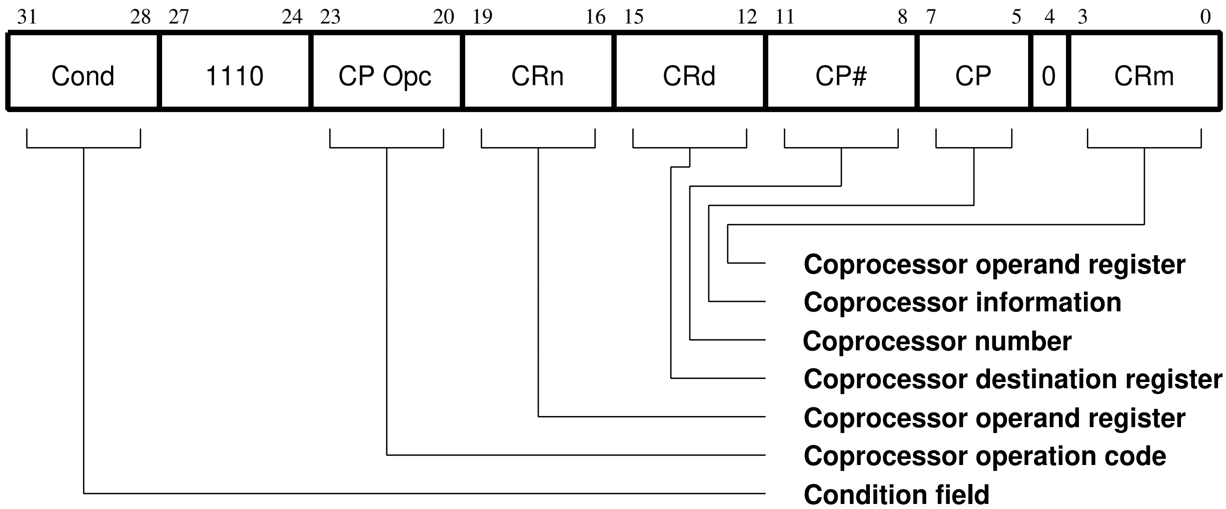


Figure 28: Coprocessor Data Operation Instruction

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.

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 8: Condition Codes*

p# - the unique number of the required coprocessor

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

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

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

4.12.4 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 ARM710 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 29: Coprocessor Data Transfer Instructions*.

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

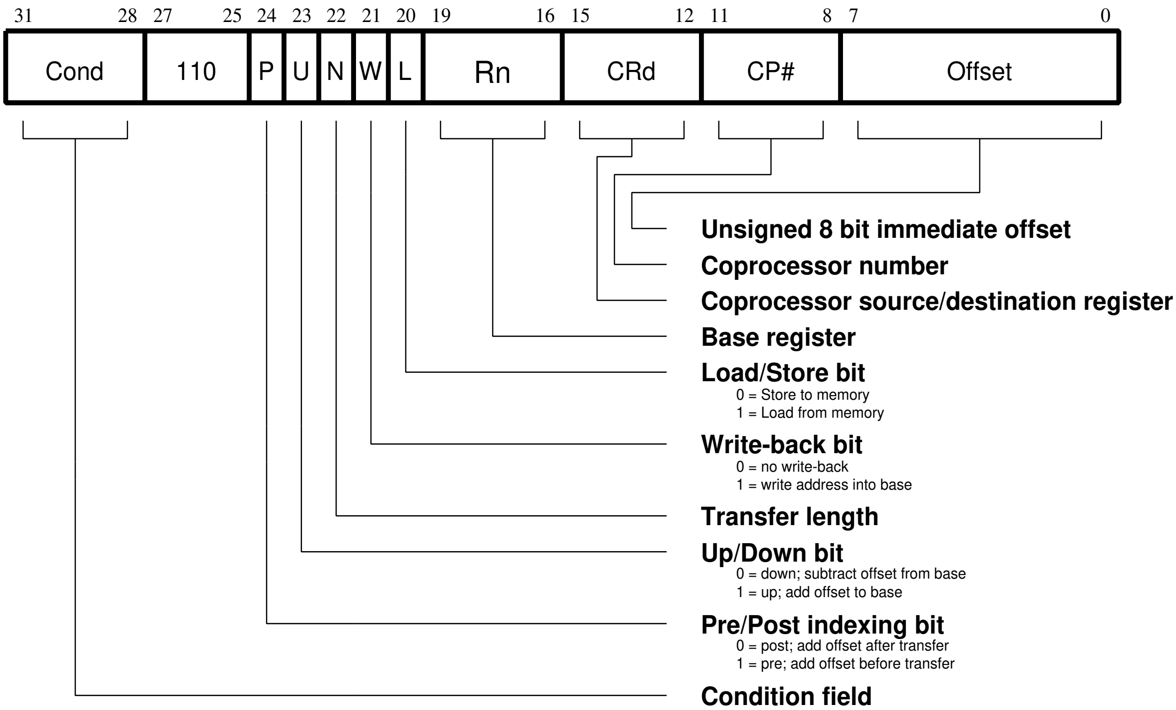


Figure 29: Coprocessor Data Transfer Instructions

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

4.13.6 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 8: Condition Codes*

p# - the unique number of the required coprocessor

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

<Address> can be:

(i) An expression which generates an address:

<expression>

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

(ii) A pre-indexed addressing specification:

[Rn] offset of zero

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

(iii) A post-indexed addressing specification:

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

Rn is an expression evaluating to a valid 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

4.13.8 Examples

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

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

4.14 Coprocessor register transfers (MRC, MCR)

Use of the MRC or MCR instruction on the ARM710 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 30: Coprocessor Register Transfer Instructions*.

This class of instruction is used to communicate information directly between ARM710 and a coprocessor. An example of a coprocessor to processor register transfer (MRC) instruction would be a FIX of a floating point value held in a coprocessor, where the floating point number is converted into a 32 bit integer within the coprocessor, and the result is then transferred to a processor register. A FLOAT of a 32 bit value in a processor register into a floating point value within the coprocessor illustrates the use of a processor register to coprocessor transfer (MCR).

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

Note the ARM710 has an internal coprocessor (#15) for control of on-chip functions. Accesses to this coprocessor are performed by coprocessor register transfers.

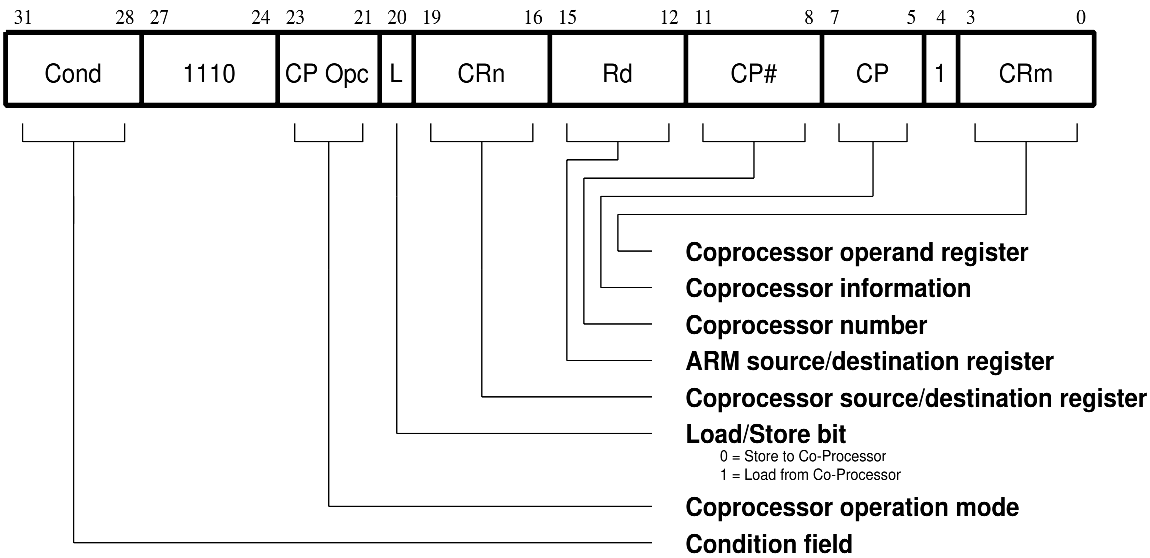


Figure 30: Coprocessor Register Transfer Instructions

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 which depends on the particular operation specified.

4.14.2 Transfers to R15

When a coprocessor register transfer to ARM710 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 ARM710 with R15 as the source register will store the PC+12.

4.14.4 Instruction Cycle Times

Access to the internal configuration register takes 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 ARM710 register (L=1)

MCR - move from ARM710 register to coprocessor (L=0)

{cond} - two character condition mnemonic, see *Figure 8: Condition Codes*

p# - the unique number of the required coprocessor

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

Rd is an expression evaluating to a valid ARM710 register number

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

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

4.14.6 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
```

4.15 Undefined instruction

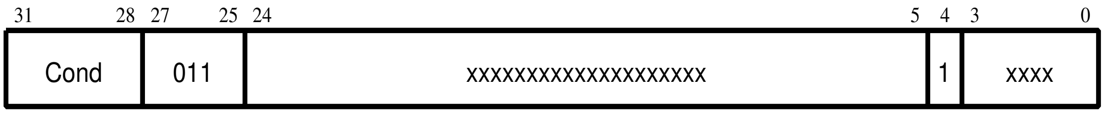


Figure 31: Undefined Instruction

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 31: 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 ARM710 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 pupose divide routine follows.

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

4.16.2 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 newbits needed (i.e. 32 bits). The entire operation can be done in 5 S cycles:

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

4.16.3 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 MOD 4 = 1, say C = 2^n*D+1, D odd, n>1:

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

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

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

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

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

rather than by:

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

4.16.4 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
```

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 6: ARM Instruction Speed Summary*. These figures assume that the instruction is actually executed. Unexecuted instructions take one instruction fetch cycle.

Instruction	Cycle count
Data Processing - normal with register specified shift with PC written with register specified shift & PC written	1 instruction fetch 1 instruction fetch and 1 internal cycle 3 instruction fetches 3 instruction fetches and 1 internal cycle
MSR, MRS	1 instruction fetch
LDR - normal if the destination is the PC	1 instruction fetch, 1 data read and 1 internal cycle 3 instruction fetches, 1 data read and 1 internal cycle
STR	1 instruction fetch and 1 data write
LDM - normal if the destination is the PC	1 instruction fetch, n data reads and 1 internal cycle 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 6: ARM Instruction Speed Summary

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.

5.0 Configuration

The operation and configuration of ARM710 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.

Note: *The grey areas in the register and translation diagrams are reserved and , where values are shown, must be programmed to the values specified.*

5.1 Internal Coprocessor Instructions

The on-chip registers may be read from and written to using a special case of the coprocessor register transfer instructions MRC and MCR, where the target coprocessor is #15. Please note that attempting to use MRC or MCR with a coprocessor other than #15 will cause ARM710 to take the undefined instruction trap. Note also that MRC and MCR can only be used in privileged (non-User) modes: see *Section 4.14: Coprocessor register transfers (MRC, MCR)* on page 58 for a general description of the MRC and MCR instructions.

The figure below shows the format of the instructions when they are used to configure the ARM710:

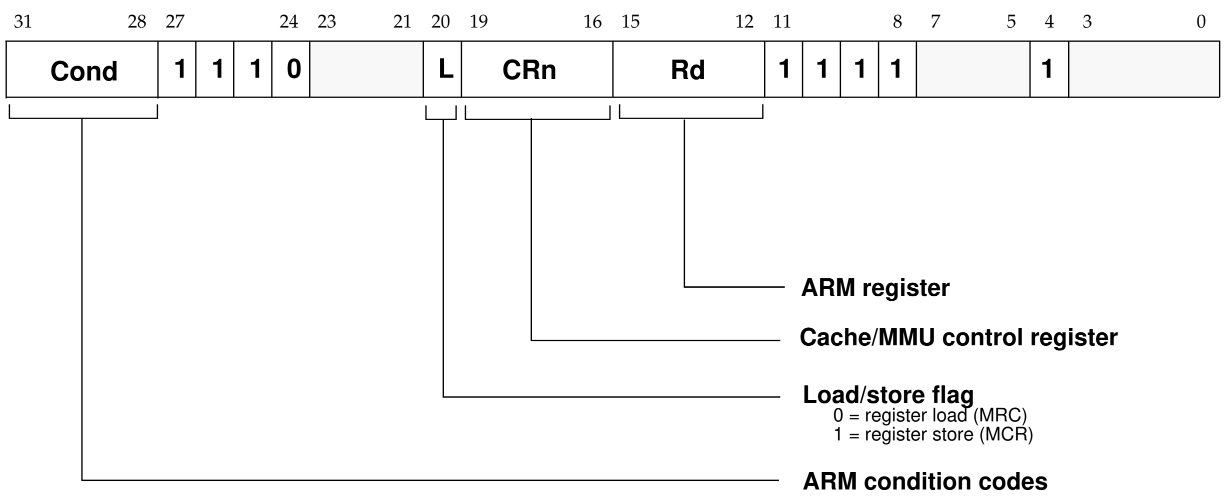


Figure 32: Format of MRC and MCR instructions for Cache and MMU control

5.2 Registers

ARM710 contains a number of registers that control the operation of the cache and MMU. These are listed in the table below:

Register	Register Reads	Register Writes
0	ID Register	Reserved
1	Reserved	Control
2	Reserved	Translation Table Base
3	Reserved	Domain Access Control
4	Reserved	Reserved
5	Fault Status	Flush TLB
6	Fault Address	Purge TLB
7	Reserved	Flush IDC
8-15	Reserved	Reserved

Table 7: Cache and MMU control registers

Attempts to read or write registers in the range 0-7 whose respective access type is reserved will fail, but will *not* cause the undefined instruction trap to be taken. For this reason such attempts must not be made. Attempts to access registers in the range 8-15 will result in the undefined instruction trap being taken.

5.2.1 Register 0: ID Register

Register 0 is a read-only identity register that returns the ARM Ltd code for this chip: 0x4100710x.

31 — 24	23 — 16	15 — 4	3 — 0
41	**00**	**710**	**Revision**

5.2.2 Register 1: Control Register.

31																					10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	R	S	B	1	D	P	W	C	A	M

Register 1 is write-only and contains control bits, all of which are forced LOW by reset. The control bits have the following functions:

No	Name	Function
0	M	**Enable/disable** 0 - on-chip Memory Management Unit turned off 1 - on-chip Memory Management Unit turned on.
1	A	**Address Fault Enable/Disable** 0 - alignment fault disabled 1 - alignment fault enabled
2	C	**Cache Enable/Disable** 0 - Instruction / data cache turned off 1 - Instruction / data cache turned on
3	W	**Write buffer Enable/Disable** 0 - Write buffer turned off 1 - Write buffer turned on
4	P	**ARM 32/26 Bit Program Space** 0 - 26 bit Program Space selected 1 - 32 bit Program Space selected
5	D	**ARM 32/26 Bit Data Space** 0 - 26 bit Data Space selected 1 - 32 bit Data Space selected
7	B	**Big/Little Endian** 0 - Little-endian operation 1 - Big-endian operation
8	S	**System** ARM710 permission system control. See *Section 9.6: Section Descriptor* on page 83.
9	R	**ROM** ARM710 permission system control. See *Section 9.6: Section Descriptor* on page 83.

Table 8: Control Register Bits

5.2.3 Register 2: Translation Table Base Register

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

31 – 14	13 – 0
Translation Table Base	

5.2.4 Register 3: Domain Access Control Register

Register 3 is a write-only register which holds the current access control for domains 0 to 15, as shown below:.

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

See *Section 9.13: Domain Access Control* on page 90 for details.

5.2.5 Register 4: Reserved

Register 4 is reserved: accessing it has no effect, but should never be attempted.

5.2.6 Register 5: Fault Status and TLB Flush Register

Read: Fault Status
Reading register 5 returns the status of the last data fault. It is not updated for a prefetch fault. See *Chapter 9.0: Memory Management Unit (MMU)* 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.

31 – 12	11			8	7 – 4	3 – 0
	0	**0**	**0**	**0**	**Domain**	**Status**

Write: Translation Lookaside Buffer Flush
Writing Register 5 flushes the TLB. (The data written is discarded).

5.2.7 Register 6: Fault Address and TLB Purge Register

Read: Fault Address
Reading register 6 returns the virtual address of the last data fault.

31 – 0
Fault Address

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.

31 … 14	13 … 0
Purge Address	

5.2.8 Register 7: IDC Flush Register

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.

6.0 Instruction and Data Cache (IDC)

ARM710 contains a 8kByte mixed instruction and data cache. The IDC has 256 lines of 32 bytes (8 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 (8 words). It may be enabled or disabled via the ARM710 Control Register and is disabled on **nRESET**. The operation of the cache is further controlled the *Cacheable,* or C, bit stored in the Memory Management Page Table (see *Chapter 9.0: Memory Management Unit (MMU)*.). 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 ARM710'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

In the ARM710 the cache will be searched regardless of the state of the C bit, only reads that miss the cache will be affected.

6.2.1 Cacheable Reads C = 1

A linefetch of 8 words will be performed 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 ARM710 IDC Flush Register (Register 7). The cache will be flushed immediately after 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 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.

7.0 Write Buffer (WB)

The ARM710 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 ARM710 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 pin will be ignored.

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 ARM710 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** 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.

8.0 Coprocessors

ARM710 has no external coprocessor bus, so it is not possible to add external coprocessors to this device. If this is required, then the ARM700 should be used.

ARM710 still has an internal coprocessor designated #15 for internal control of the device. All coprocessor operations except MCR or MRC to registers 0 to 7 on coprocessor #15 will cause the undefined instruction trap to be taken.

9.0 Memory Management Unit (MMU)

The 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 ARM710 Processor provides several 32-bit registers which determine the operation of the MMU. The format for these registers is shown in *Figure 33: MMU Register Summary*. 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 33: 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 12: Priority Encoding of Fault Status*. 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.

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 34: Translation Table Base Register*; the table must reside on a 16kB boundary.

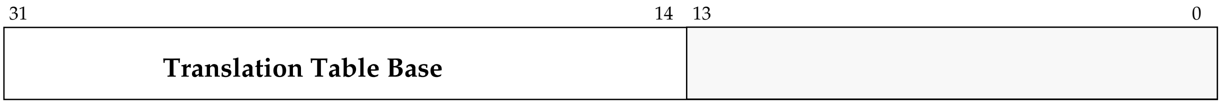


Figure 34: Translation Table Base Register

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 35: 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).

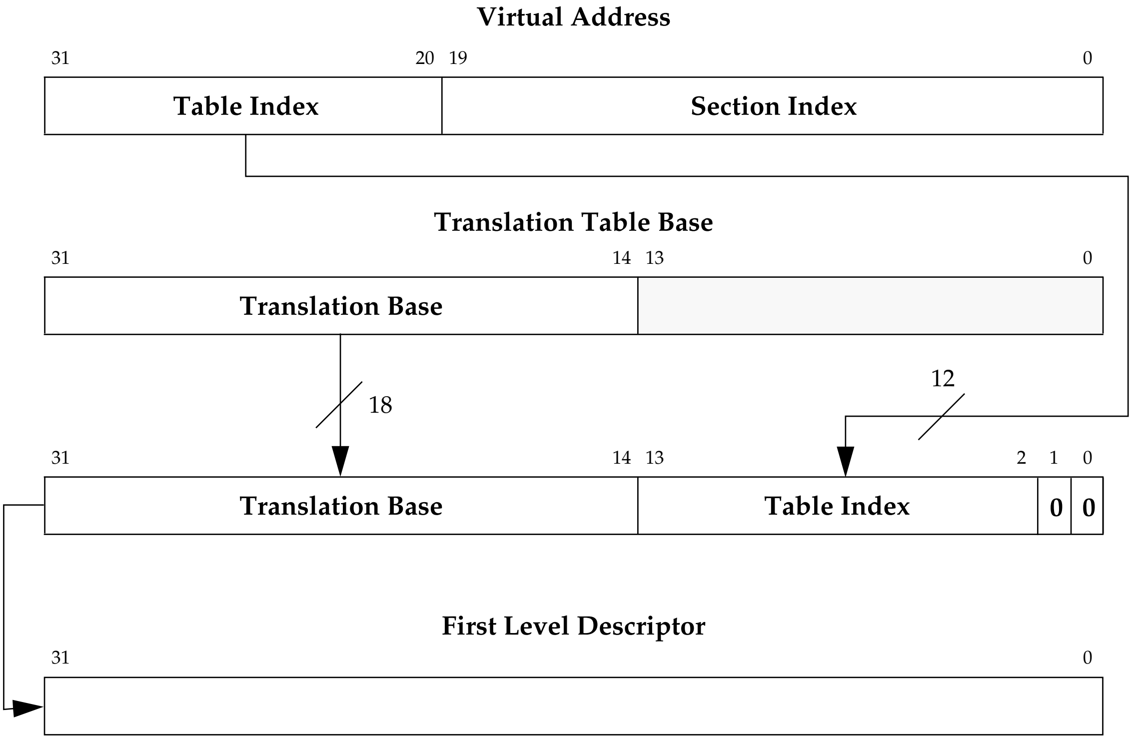


Figure 35: Accessing the Translation Table First Level Descriptors

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.

31–20	19–12	11–10	9	8–5	4	3	2	1	0	
								0	0	**Fault**
Page Table Base Address				**Domain**	**1**			0	1	**Page**
Section Base Address		**AP**		**Domain**	**1**	**C**	**B**	1	0	**Section**
								1	1	**Reserved**

Figure 36: 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: 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 39: Small Page Translation*).

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 10: Interpreting Access Permission (AP) Bits*. 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	Permissions 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 10: 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.

9.7 Translating Section References

Figure 37: 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.

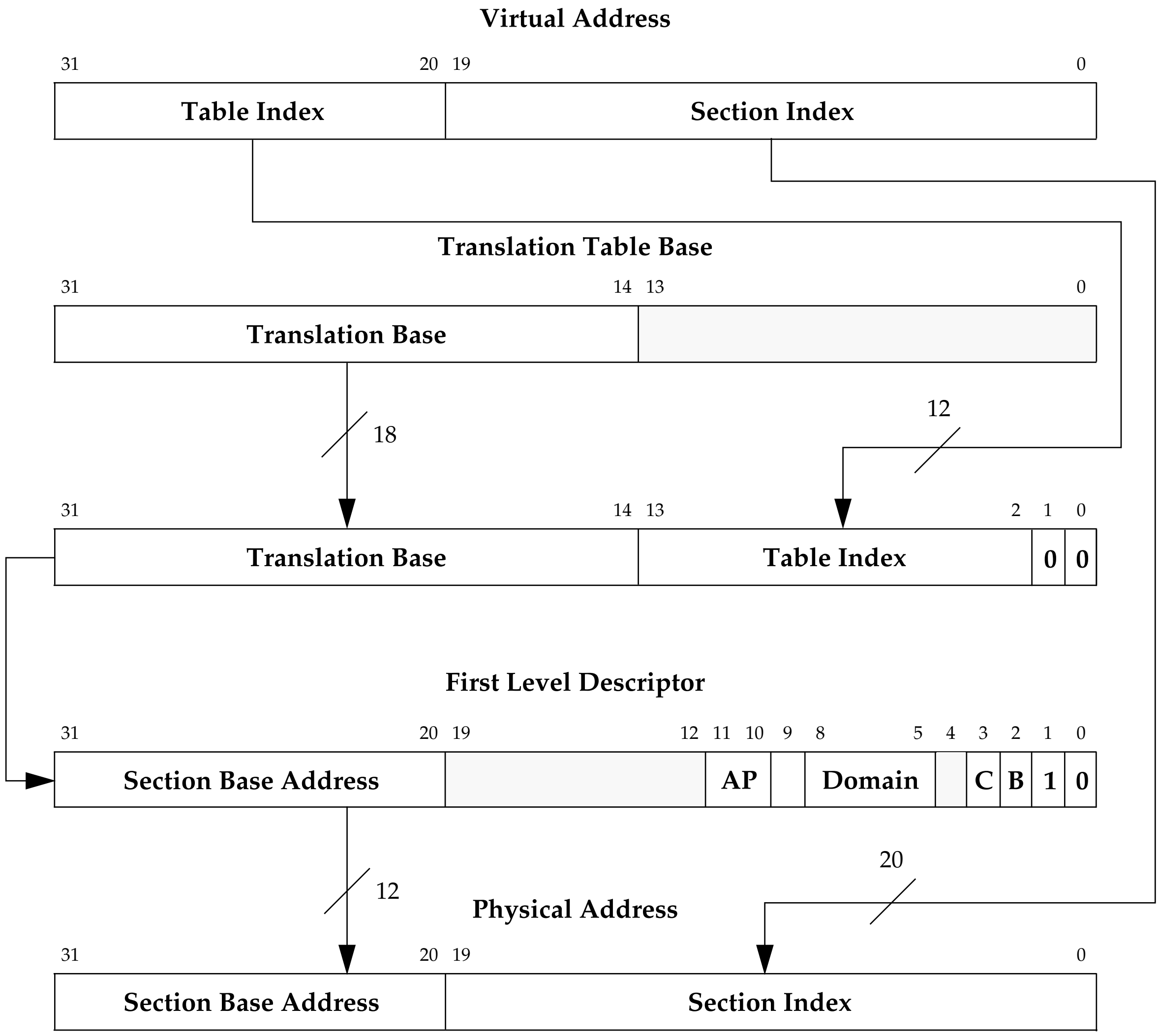


Figure 37: Section Translation

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 39: Small Page Translation*, 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 38: 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 11: 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: Interpreting Level One Descriptor Bits [1:0]*.

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 39: Small Page Translation* and *Figure 40: Large Page Translation*).

9.9 Translating Small Page References

Figure 39: 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).

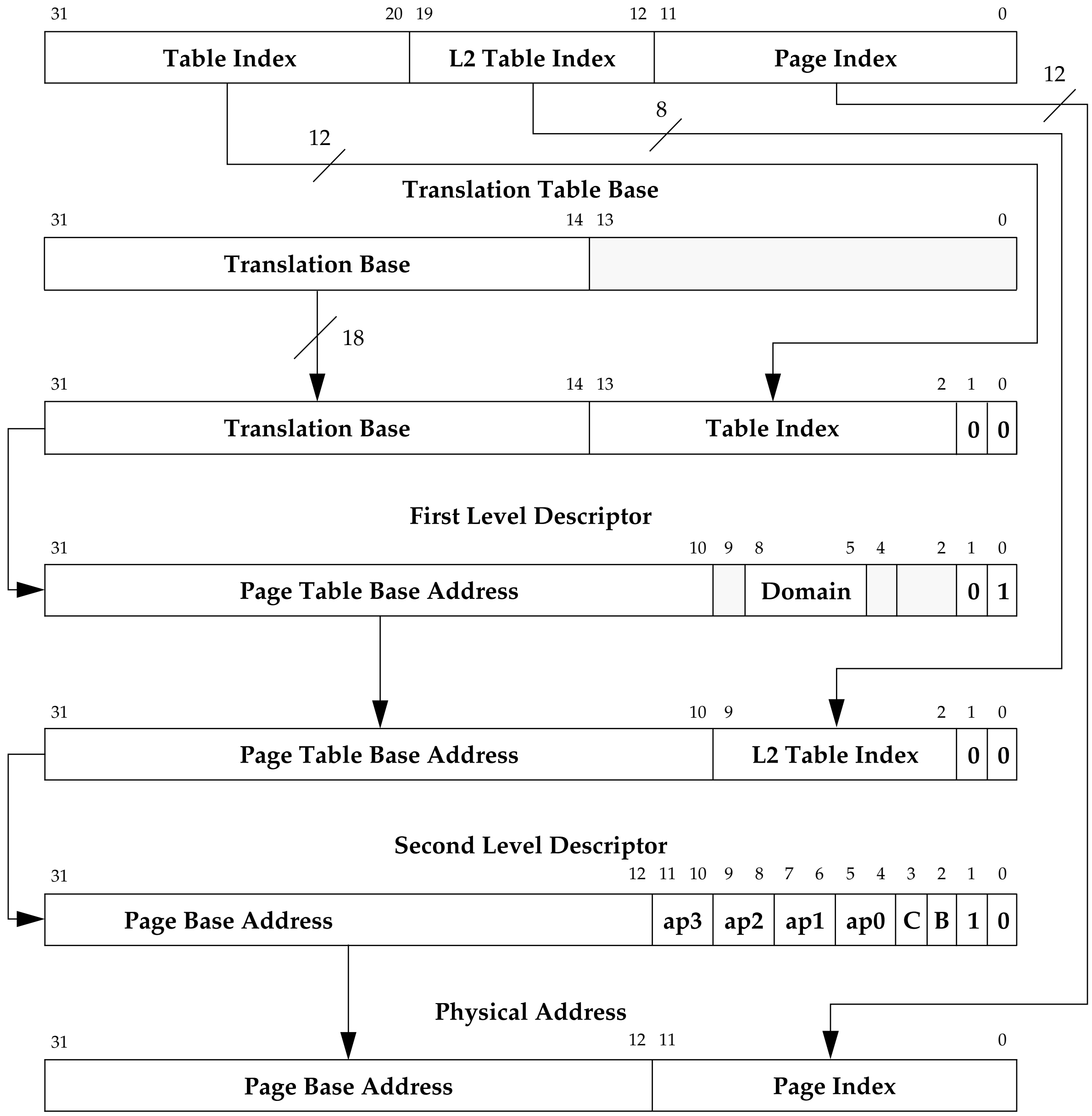


Figure 39: Small Page Translation

9.10 Translating Large Page References

Figure 40: 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.

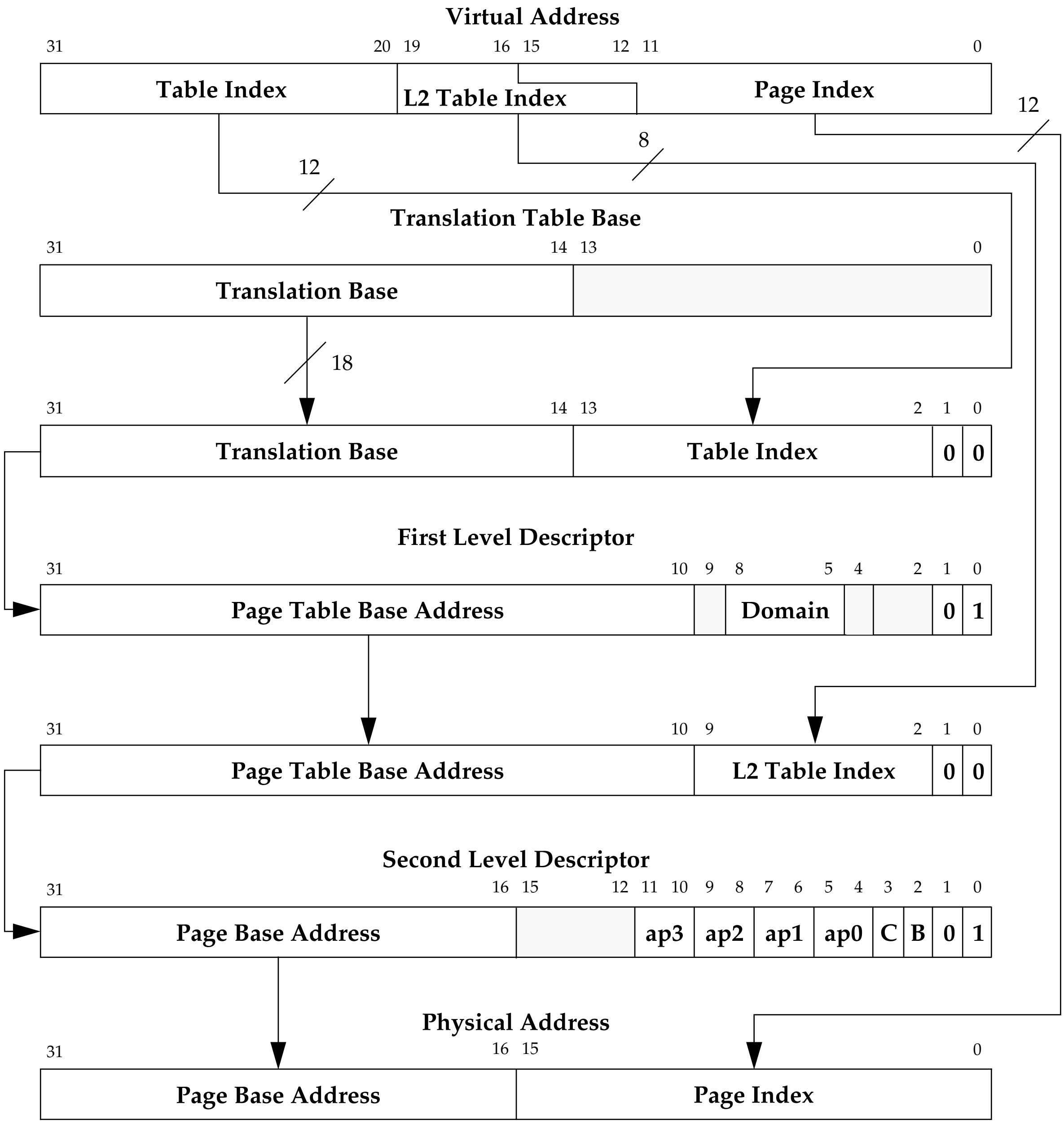


Figure 40: Large Page Translation

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.

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 12: Priority Encoding of Fault Status*.

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

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

	Source		FS[3210]	Domain[3:0]	FAR
Highest	Bus Error (linefetch)	Section	0100	valid	valid
		Page	0110	valid	valid
	Bus Error (other)	Section	1000	valid	valid
		Page	1010	valid	valid
	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
	Permission	Section	1101	valid	valid
Lowest		Page	1111	valid	valid

Table 12: 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.

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 41: Domain Access Control Register Format* illustrates how the 32 bits of the register are allocated to define the sixteen 2-bit domains.

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

Figure 41: Domain Access Control Register Format

Table 13: 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 13: 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.

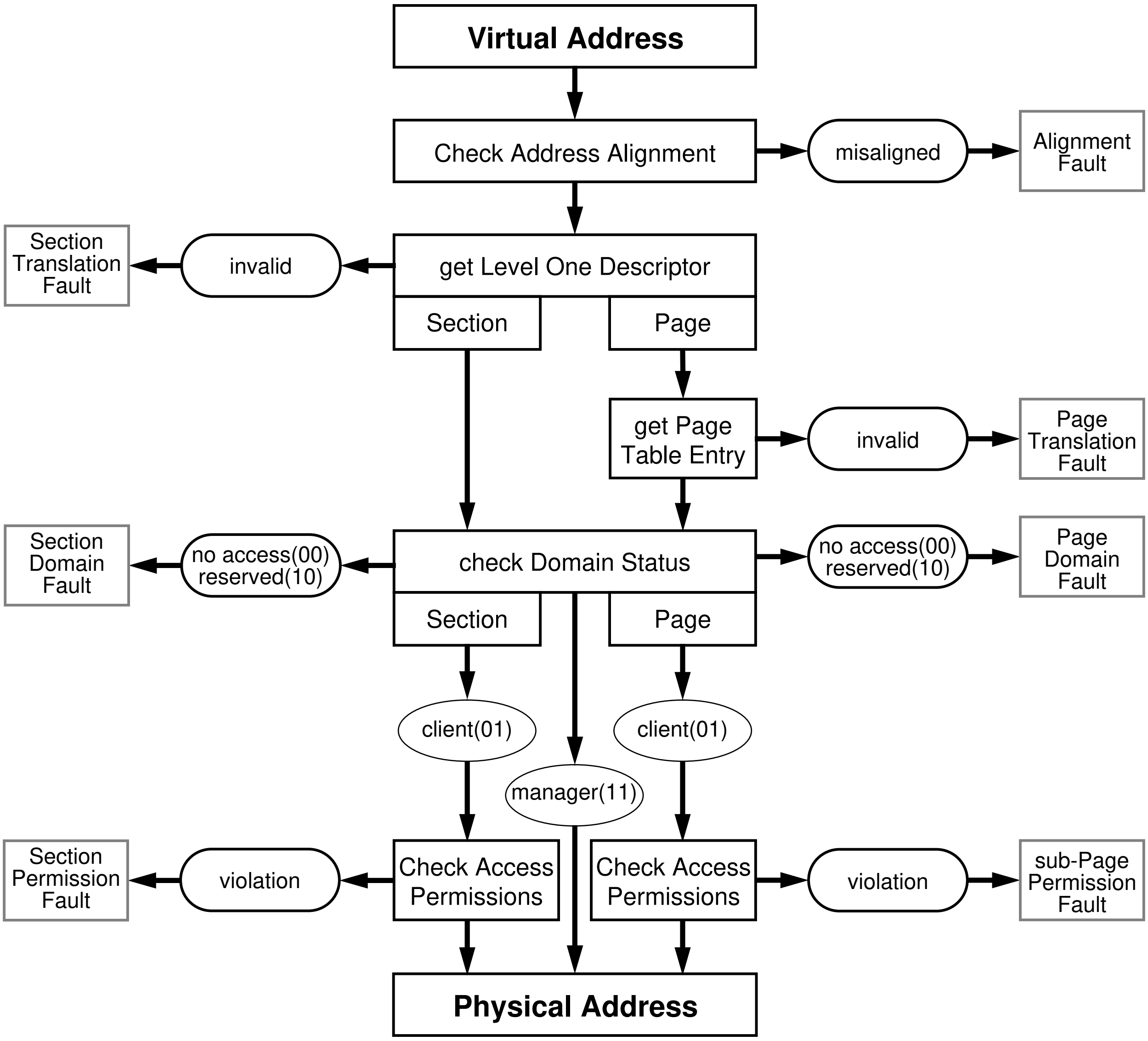


Figure 42: Sequence for Checking Faults

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 10: Interpreting Access Permission (AP) Bits*. 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 10: Interpreting Access Permission (AP) Bits*. 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 10: Interpreting Access Permission (AP) Bits*), 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, ARM710 has an external abort pin which may be used to flag an error on an external memory access. However, not all accesses can be aborted in this way, so this pin must be used with great care. The following section describes the restrictions.

The following accesses may be aborted and restarted safely. 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 ARM710, 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.

9.16 Interaction of the MMU, IDC and Write Buffer

The MMU, IDC and WB may be enabled/disabled independently. However there are only five valid combinations. 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 14: 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 *Section 3.5: Reset* on page 18.

10.0 Bus Interface

The ARM710 has two input clocks **FCLK** and **MCLK**. The bus interface is always controlled by **MCLK**. The core CPU switches between the two clocks according to the operation being carried out. For example, when reading data from the cache, it is clocked by **FCLK**,whereas it is clocked by **MCLK** when reading data from unached external memory. The ARM710 control logic ensures that the correct clock is used internally and switches between the two clocks automatically. At all times **FCLK** must be greater than or equal to **MCLK** in frequency.

The ARM710 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.

10.1 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. Thus when changing from **FCLK** to **MCLK** the average resynchronisation penalty is half a **MCLK** period, and similarly when changing from **MCLK** to **FCLK** it is half a **FCLK** period.

10.2 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 not be later than **FCLK**. Refer to *Section 12.2: DC Operating Conditions* on page 117.

10.3 ARM710 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**. **nWAIT** may only change when **MCLK** is LOW. See *Section 10.15: Use of the nWAIT pin* on page 102.

10.4 Cycle Types

There are two basic cycle types performed by an ARM710. These are *idle* cycles and *memory* cycles. Idle cycles and memory cycles are combined to perform memory accesses. The two cycle types are differentiated by the signal **nMREQ**. (**SEQ** is the inverse of **nMREQ**, and is provided for backwards compatibility with earlier memory controllers). **nMREQ** HIGH indicates an idle cycle, and **nMREQ** LOW indicates a memory access. However, **nMREQ** is pipelined, and so its value determines what type the following cycle will be. **nMREQ** becomes valid during the LOW phase of the cycle before the one to which it refers.

The address from ARM710 becomes valid during the HIGH phase of **MCLK**. It is also pipelined, and its value refers to the following memory access.

10.5 Memory Access

There are two types of memory access. These are *non-sequential* and *sequential*. The non-sequential cycles occur when a new memory access takes place. Sequential cycles occur when the cycle is of the same type as, and the address of is 1 word (4 bytes) greater than, the previous access. So for example, a single word access consists of a non-sequential access, and a two word access consists of a non-sequential access followed by a sequential access.

Non-sequential accesses consist of an idle cycle followed by a memory cycle, and sequential accesses consist simply of a memory cycle. In the case of a non-sequential access, the address is valid throughout the idle cycle, allowing extra time for memory decoding.

10.6 Read/Write

Memory accesses may be read or write, differentiated by the signal **nRW**. This signal has the same timing as the address, so is likewise pipelined, and refers to the following cycle. In the case of a write, the ARM710 outputs data on the data bus during the memory cycle. It becomes valid during **MCLK** LOW, and is held until the end of the cycle. In the case of a read, then data is sampled at the end of the memory cycle. **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 a non-sequential access.

10.7 Byte/Word

Likewise, any memory access may be of a word or a byte quantity. These are differentiated by the signal **nBW**, which also has the same timing as the address, ie it becomes valid in the HIGH phase of **MCLK** in the cycle before the one to which it refers. **nBW** LOW indicates a byte access. Again, **nBW** may not change during sequential accesses.

10.8 Maximum Sequential Length

As explained above, the ARM710 will perform sequential memory accesses whenever the cycle is of the same type (ie byte/word, 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 simply turned into a non-sequential access, and then further accesses continue sequentially as before.

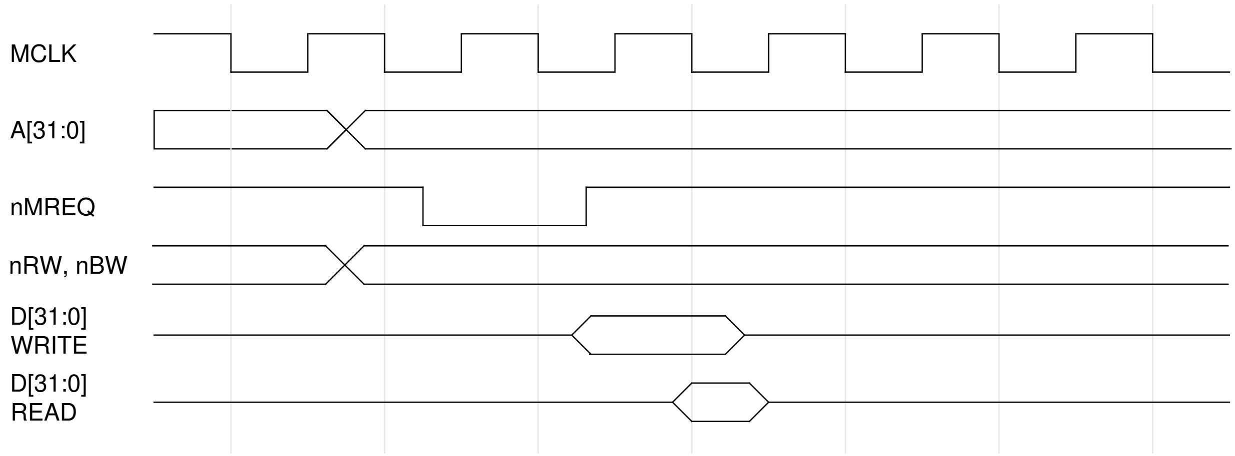


Figure 43: One Word Read or Write

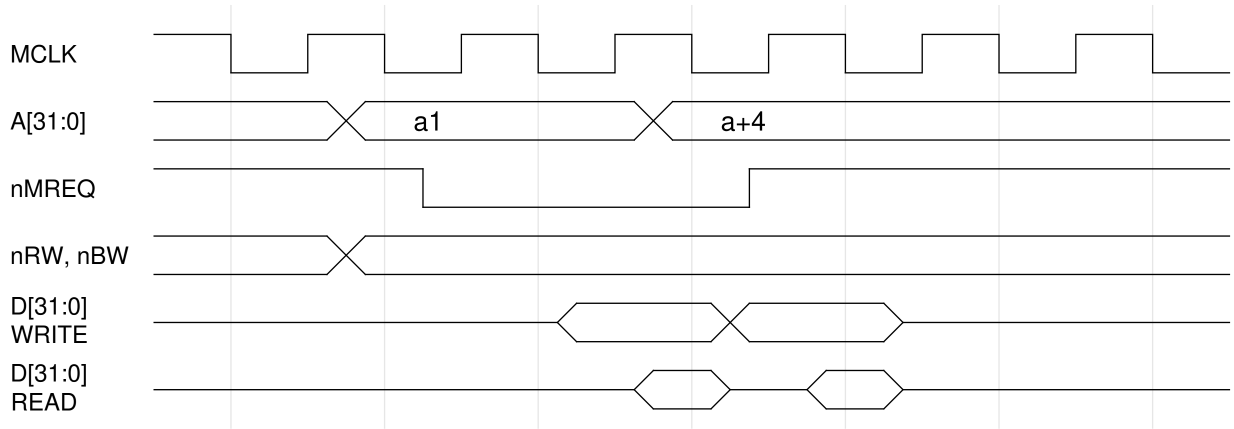


Figure 44: Two Word Sequential Read or Write

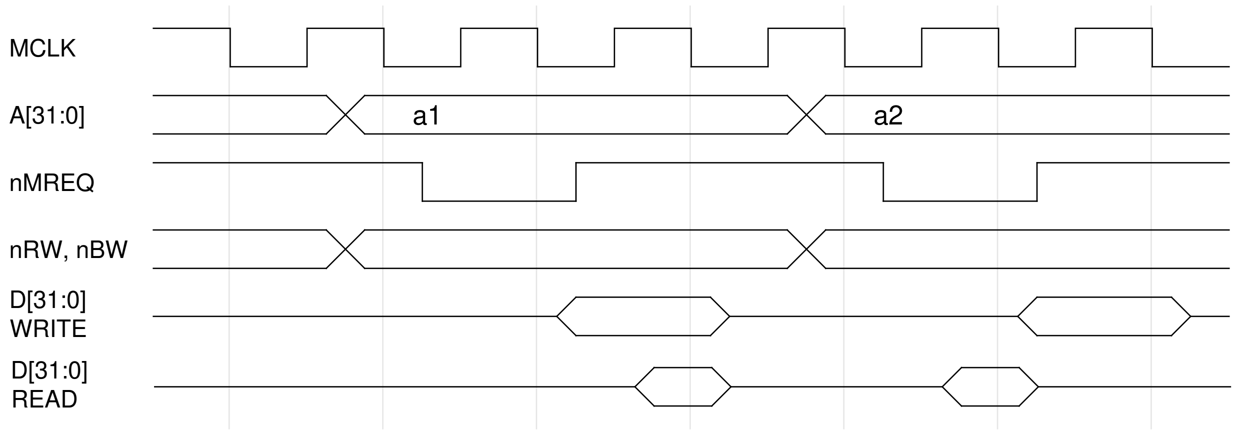


Figure 45: Two Word Non-Sequential Unbuffered accesses

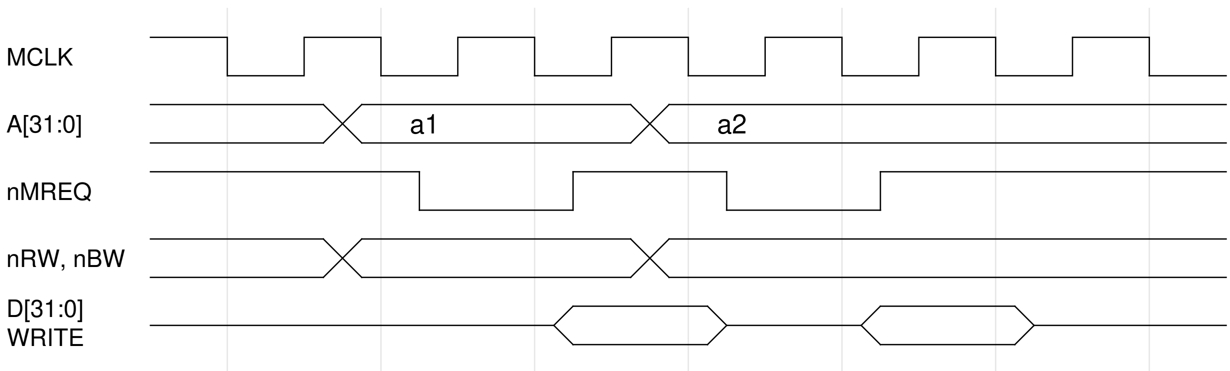


Figure 46: Two Word Non-Sequential Buffered Writes

10.9 Memory Access Types

ARM710 performs many different bus accesses, 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 ARM710 are:

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

10.10 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.

10.11 Buffered Write

The external bus cycle of a buffered write is identical to and indistinguishable from the bus cycle of an unbuffered write. 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 access on the bus will start with a non-sequential access.

10.12 Linefetch

This access appears on the bus as a non-sequential access followed by seven sequential accesses. Note that linefetch accesses always start on an 8-word boundary, and are always word accesses. So if the instruction which caused the linefetch was a byte load instruction (eg LDRB), then the linefetch access will be a word access on the bus. *Figure 47: Linefetch* shows the start of a linefetch.

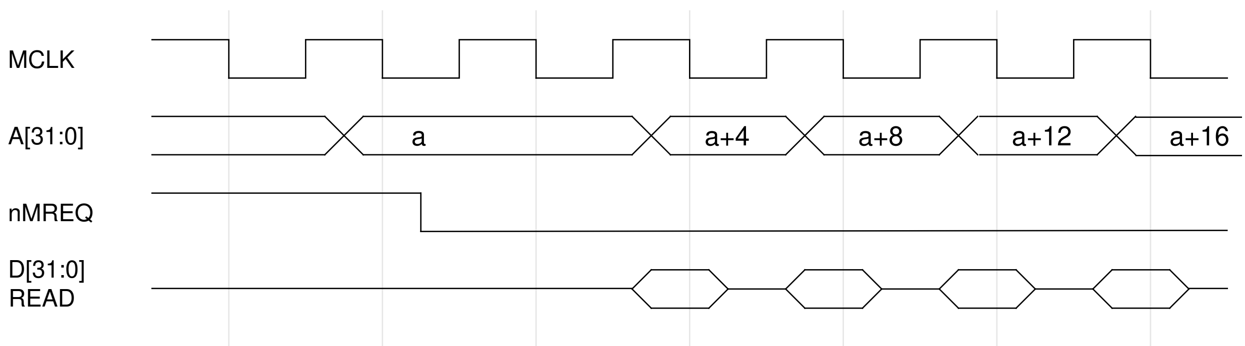


Figure 47: Linefetch

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 ARM710, 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.

10.13 Translation fetches

These accesses are required to obtain the translation data for an access. There are two types, Level 1 & 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, then the access would be preceded by the translation fetch(es) which would always be word read accesses.

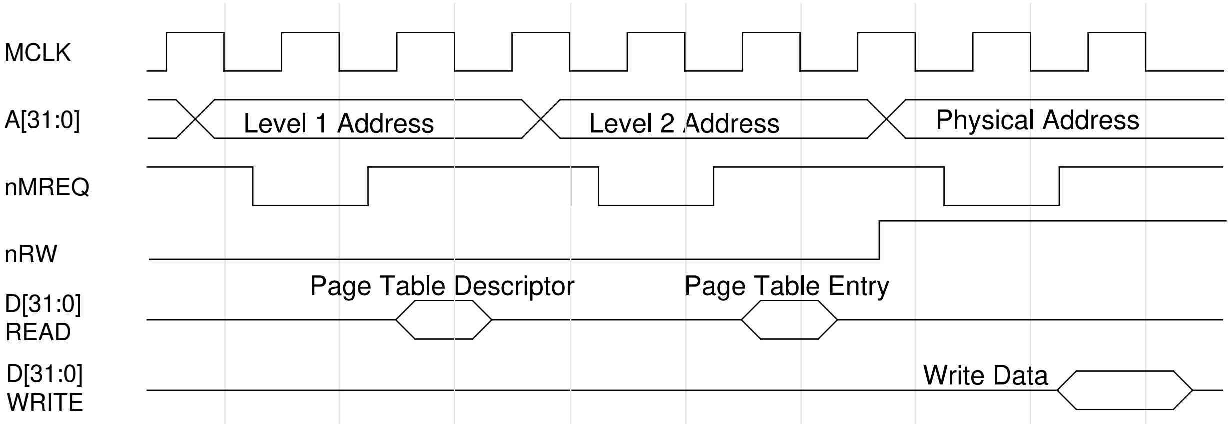


Figure 48: Translation Table-walking Sequence (write) For Page

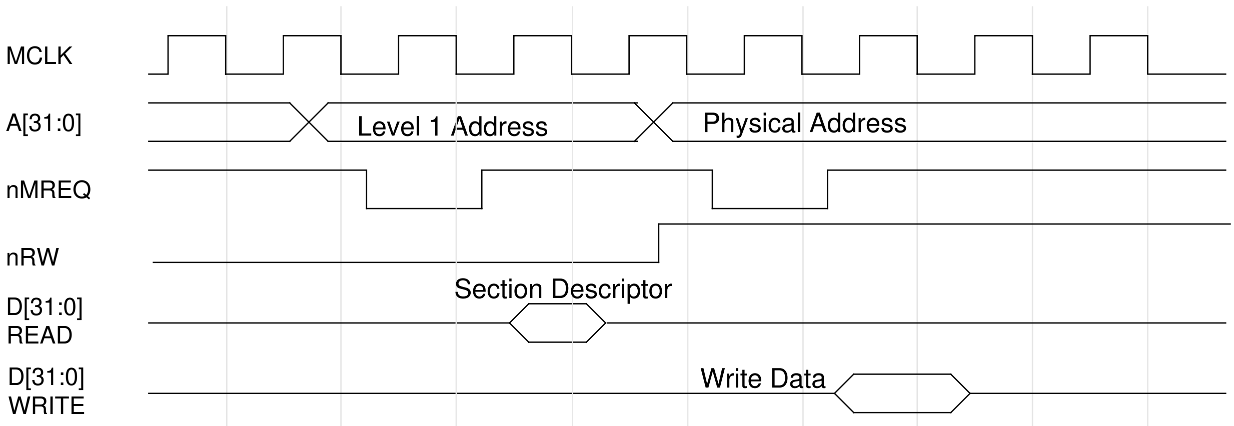


Figure 49: Translation Table-walking Sequence (write) For Section

10.14 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, and both are treated as non-sequential accesses. The cycle is differentiated by the **LOCK** signal. **LOCK** has the timing of address, ie it goes HIGH in the HIGH phase of **MCLK** at the start of the read access. However, it always goes LOW at the end of the write access.

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.

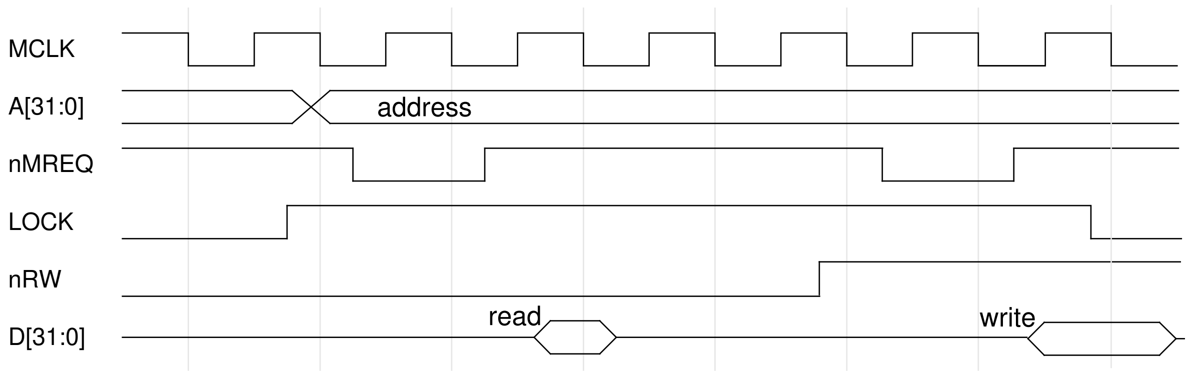


Figure 50: Read - Locked - Write

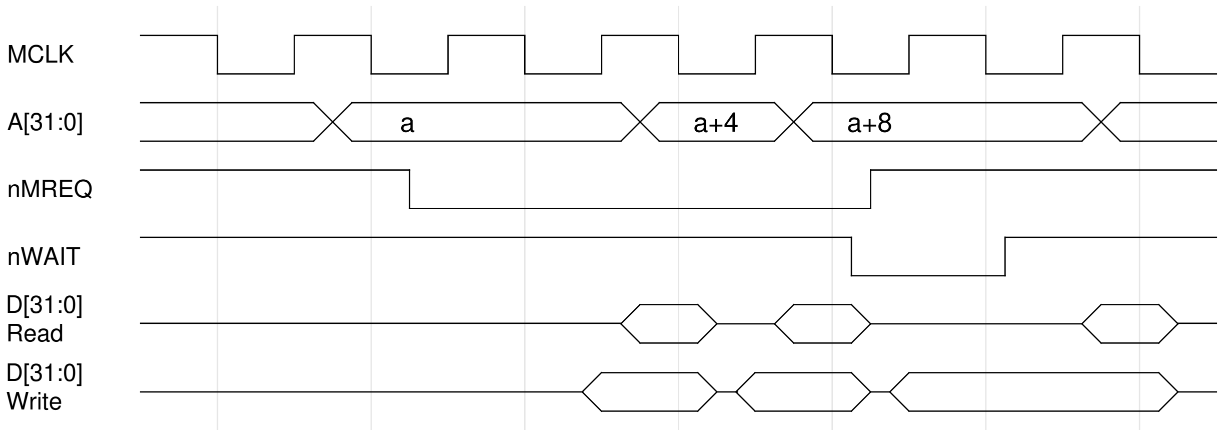


Figure 51: Use of nWAIT pin to stop ARM710 for 1 MCLK cycle

10.15 Use of the nWAIT pin

The **nWAIT** pin can be used to stretch 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 raised. All other outputs cannot change from the time **nWAIT** goes LOW until the next **MCLK** HIGH phase after **nWAIT** returns HIGH. If **ALE** is being used to latch an address when **nWAIT** is taken LOW, the address and control signals will changes when **ALE** returns HIGH regardless of the state of **nWAIT**. See *Figure 51: Use of nWAIT pin to stop ARM710 for 1 MCLK cycle*.

10.16 ARM710 Cycle Type Summary

	Operation	nRW	A[31:0]	nMREQ	D[31:0]
	Idle	old	old	i	
	Linefetch	r	a	i	
		r	a	m	
		r	a+4	m	d
		r	a+8	m	d
		r	a+12	m	d
		r	a+16	m	d
		r	a+20	m	d
		r	a+24	m	d
		r	a+28	m	d
		r	a+28	i	d
Uncacheable Read / Unbuffered Write	Start	r/w	a	i	
		r/w	a	m	
					d
	Repeat	r/w	a+n	m	
					d
	End	r/w	old	i	
Buffered Write	Start	w	a	i	
		w	a	m	
					d
	Repeat	w	a+n	m	
					d
Read-Lock-Write	Read phase	r	aL	i	
		r	aL	m	
		r	aL	i	d
		r	aL	i	
	Write phase - Unbuffered	w	aL	i	
		w	aL	m	
		w	aL	i	d
	Write phase - Aborted	r	aL	i	
		r	aL	i	
		r	aL	i	

Table 15: Cycle Type Summary

Key to Cycle Type Summary:

r	- Read (**nRW** LOW)
r/w	- applies equally to Read and Write
w	- Write (**nRW** HIGH)
old	- signal remains at previous value
a	- first Address
a+n	- next sequential address
aL	- Read-Lock-Write Address
i	- Idle cycle (**nMREQ** HIGH)
m	- Memory cycle (**nMREQ** LOW)
d	- valid data on data bus

Each line in *Table 15: Cycle Type Summary* 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.

11.0 Boundary Scan Test Interface

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

11.1 Overview

The boundary-scan interface provides a means of testing the core of the device when it is fitted to a circuit board, and a means of driving and sampling all the external pins of the device irrespective of the core state. This latter function permits testing of both the device's electrical connections to the circuit board, and (in conjunction with other devices on the circuit board having a similar interface) testing the integrity of the circuit board connections between devices. The interface intercepts all external connections within the device, and each such "cell" is then connected together to form a serial register (the boundary scan register). The whole interface is controlled via 5 dedicated pins: **TDI**, **TMS**, **TCK**, **nTRST** and **TDO**. *Figure 52: Test Access Port (TAP) Controller Sate Transitions* shows the state transitions that occur in the TAP controller.

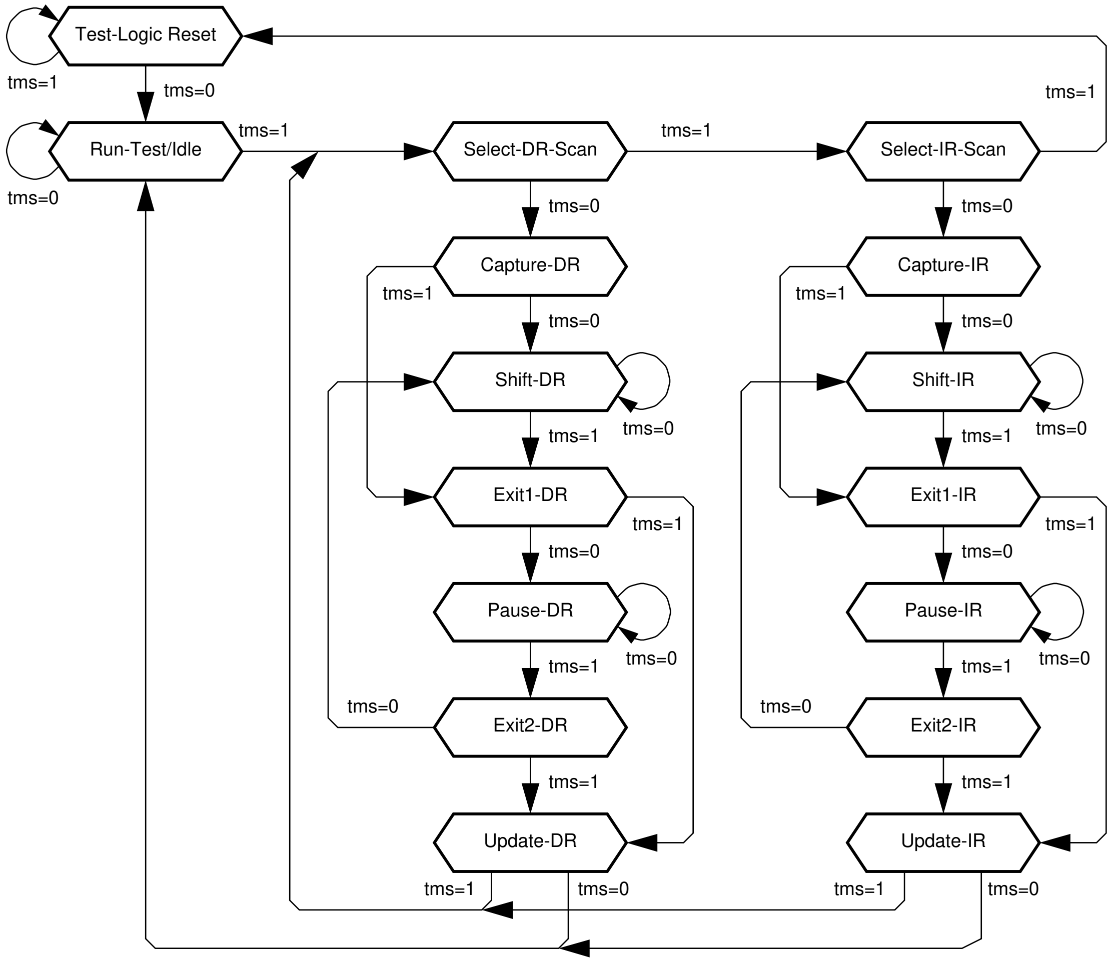


Figure 52: Test Access Port (TAP) Controller Sate Transitions

11.2 Reset

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

The action of reset (either a pulse or a DC level) is as follows:

System mode is selected (i.e. the boundary scan chain does *not* intercept any of the signals passing between the pads and the core).

IDcode mode is selected. If **TCK** is pulsed, the contents of the ID register will be clocked out of **TDO**.

11.3 Pullup Resistors

The IEEE 1149.1 standard effectively requires that **TDI**, **nTRST** and **TMS** should have internal pullup resistors. In order to minimise static current draw, these resistors are *not* fitted to ARM710. Accordingly, the 4 inputs to the test interface (the above 3 signals plus **TCK**) must all be driven to good logic levels to achieve normal circuit operation.

11.4 Instruction Register

The instruction register is 4 bits in length.

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

11.5 Public Instructions

The following public instructions are supported:

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

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

11.5.1 EXTEST (0000)

The BS (boundary-scan) register is placed in test mode by the EXTEST instruction.

The EXTEST instruction connects the BS register between **TDI** and **TDO**.

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

In the CAPTURE-DR state, inputs from the system pins and outputs from the boundary-scan output cells to the system pins are captured by the boundary-scan cells. In the SHIFT-DR state, the previously captured test data is shifted out of the BS register via the **TDO** pin, whilst new test data is shifted in via the **TDI** pin to the BS register parallel input latch. In the UPDATE-DR state, the new test data is transferred into the BS register parallel output latch. Note that this data is applied immediately to the system logic and system pins. The first EXTEST vector should be clocked into the boundary-scan register, using the SAMPLE/PRELOAD instruction, prior to selecting INTEST to ensure that known data is applied to the system logic.

11.5.2 SAMPLE/PRELOAD (0011)

The BS (boundary-scan) register is placed in normal (system) mode by the SAMPLE/PRELOAD instruction.

The SAMPLE/PRELOAD instruction connects the BS register between **TDI** and **TDO**.

When the instruction register is loaded with the SAMPLE/PRELOAD instruction, all the boundary-scan cells are placed in their normal system mode of operation.

In the CAPTURE-DR state, a snapshot of the signals at the boundary-scan cells is taken on the rising edge of **TCK**. Normal system operation is unaffected. In the SHIFT-DR state, the sampled test data is shifted out of the BS register via the **TDO** pin, whilst new data is shifted in via the **TDI** pin to preload the BS register parallel input latch. In the UPDATE-DR state, the preloaded data is transferred into the BS register parallel output latch. Note that this data is not applied to the system logic or system pins while the SAMPLE/PRELOAD instruction is active. This instruction should be used to preload the boundary-scan register with known data prior to selecting the INTEST or EXTEST instructions (see the table below for appropriate guard values to be used for each boundary-scan cell).

11.5.3 CLAMP (0101)

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

When the CLAMP instruction is loaded into the instruction register, the state of all output signals is defined by the values previously loaded into the boundary-scan register. A guarding pattern (specified for this device at the end of this section) should be pre-loaded into the boundary-scan register using the SAMPLE/PRELOAD instruction prior to selecting the CLAMP instruction.

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

11.5.4 HIGHZ (0111)

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

When the HIGHZ instruction is loaded into the instruction register, all outputs are placed in an inactive drive state.

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

11.5.5 CLAMPZ (1001)

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

When the CLAMPZ instruction is loaded into the instruction register, all outputs are placed in an inactive drive state, but the data supplied to the disabled output drivers is derived from the boundary-scan cells. The purpose of this instruction is to ensure, during production testing, that each output driver can be disabled when its data input is either a 0 or a 1.

A guarding pattern (specified for this device at the end of this section) should be pre-loaded into the boundary-scan register using the SAMPLE/PRELOAD instruction prior to selecting the CLAMPZ instruction.

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

11.5.6 INTEST (1100)

The BS (boundary-scan) register is placed in test mode by the INTEST instruction.

The INTEST instruction connects the BS register between **TDI** and **TDO**.

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

In the CAPTURE-DR state, the complement of the data supplied to the core logic from input boundary-scan cells is captured, while the true value of the data that is output from the core logic to output boundary- scan cells is captured. Note that CAPTURE-DR captures the complemented value of the input cells for testability reasons.

In the SHIFT-DR state, the previously captured test data is shifted out of the BS register via the **TDO** pin, whilst new test data is shifted in via the **TDI** pin to the BS register parallel input latch. In the UPDATE-DR state, the new test data is transferred into the BS register parallel output latch. Note that this data is applied immediately to the system logic and system pins. The first INTEST vector should be clocked into the boundary-scan register, using the SAMPLE/PRELOAD instruction, prior to selecting INTEST to ensure that known data is applied to the system logic.

Single-step operation is possible using the INTEST instruction.

11.5.7 IDCODE (1110)

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

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

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

11.5.8 BYPASS (1111)

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

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

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

11.6 Test Data Registers

Figure 53: Boundary Scan Block Diagram illustrates the structure of the boundary scan logic.

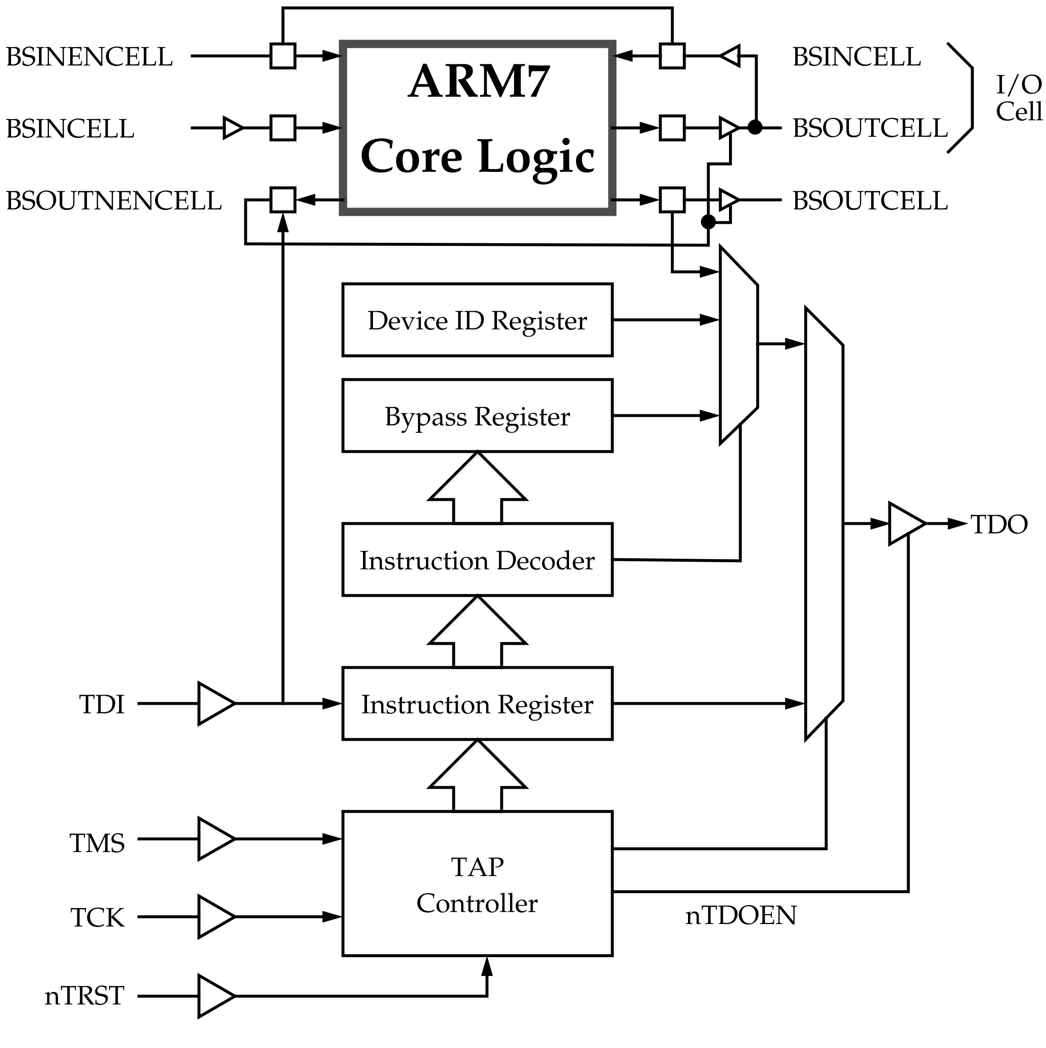


Figure 53: Boundary Scan Block Diagram

11.6.1 Bypass Register

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

Length: 1 bit

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

There is no parallel output from the bypass register.

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

11.6.2 ARM710 Device Identification (ID) Code Register

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

Length: 32 bits

The format of the ID register is as follows:

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

Please contact your supplier for the correct Device Identification Code.

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

There is no parallel output from the ID register.

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

11.6.3 ARM710 Boundary Scan (BS) Register

Purpose: The BS register consists of a serially connected set of cells around the periphery of the device, at the interface between the core logic and the system input/output pads. This register can be used to isolate the core logic from the pins and then apply tests to the core logic, or conversely to isolate the pins from the core logic and then drive or monitor the system pins.

Operating modes: The BS register is selected as the register to be connected between **TDI** and **TDO** only during the SAMPLE/PRELOAD, EXTEST and INTEST instructions. Values in the BS register are used, but are not changed, during the CLAMP and CLAMPZ instructions.

In the normal (system) mode of operation, straight-through connections between the core logic and pins are maintained and normal system operation is unaffected.

In TEST mode (ie when either EXTEST or INTEST is the currently selected instruction), values can be applied to the core logic or output pins independently of the actual values on the input pins and core logic outputs respectively. On the ARM710 all of the boundary scan cells include an update register and thus all of the pins can be controlled in the above manner. Additional boundary-scan cells are interposed in the scan chain in order to control the enabling of tristateable buses.

The correspondence between boundary-scan cells and system pins, system direction controls and system output enables is as shown in *Table 17: Boundary Scan Signals & Pins*. The cells are listed in the order in which they are connected in the boundary-scan register, starting with the cell closest to **TDI**. All boundary-scan register cells at input pins can apply tests to the on-chip core logic.

The EXTEST guard values specified in *Table 17: Boundary Scan Signals & Pins* should be clocked into the boundary-scan register (using the SAMPLE/PRELOAD instruction) before the EXTEST instruction is selected, to ensure that known data is applied to the core logic during the test. The INTEST guard values shown in the table below should be clocked into the boundary-scan register (using the SAMPLE/PRELOAD instruction) before the INTEST instruction is selected to ensure that all outputs are disabled. These guard values should also be used when new EXTEST or INTEST vectors are clocked into the boundary-scan register.

The values stored in the BS register after power-up are not defined. Similarly, the values previously clocked into the BS register are not guaranteed to be maintained across a Boundary Scan reset (from forcing **nTRST** LOW or entering the Test Logic Reset state).

11.6.4 Output Enable Boundary-scan Cells

The boundary-scan register cells Nendout, Nabe, Ntbe, and Nmse control the output drivers of tristate outputs as shown in the table below. In the case of OUTEN0 enable cells (Nendout, Ntbe), loading a 1 into the cell will place the associated drivers into the tristate state, while in the case of type INEN1 enable cells (Nabe, Nmse), loading a 0 into the cell will tristate the associated drivers.

To put all ARM710 tristate outputs into their high impedance state, a logic 1 should be clocked into the output enable boundary-scan cells Nendout and Ntbe, and a logic 0 should be clocked into Nabe and Nmse. Alternatively, the HIGHZ instruction can be used.

For example, if the on-chip core logic causes the drivers controlled by Nendout to be tristate, (ie by driving the signal **nENDOUT** HIGH), then a 1 will be observed on this cell if the SAMPLE/PRELOAD or INTEST instructions are active.

11.6.5 Single-step Operation

ARM710 is a static design and there is no minimum clock speed. It can therefore be single-stepped while the INTEST instruction is selected. This can be achieved by serialising a parallel stimulus and clocking the resulting serial vectors into the boundary-scan register. When the boundary-scan register is updated, new test stimuli are applied to the core logic inputs; the effect of these stimuli can then be observed on the core logic outputs by capturing them in the boundary-scan register.

11.7 Boundary Scan Interface Signals

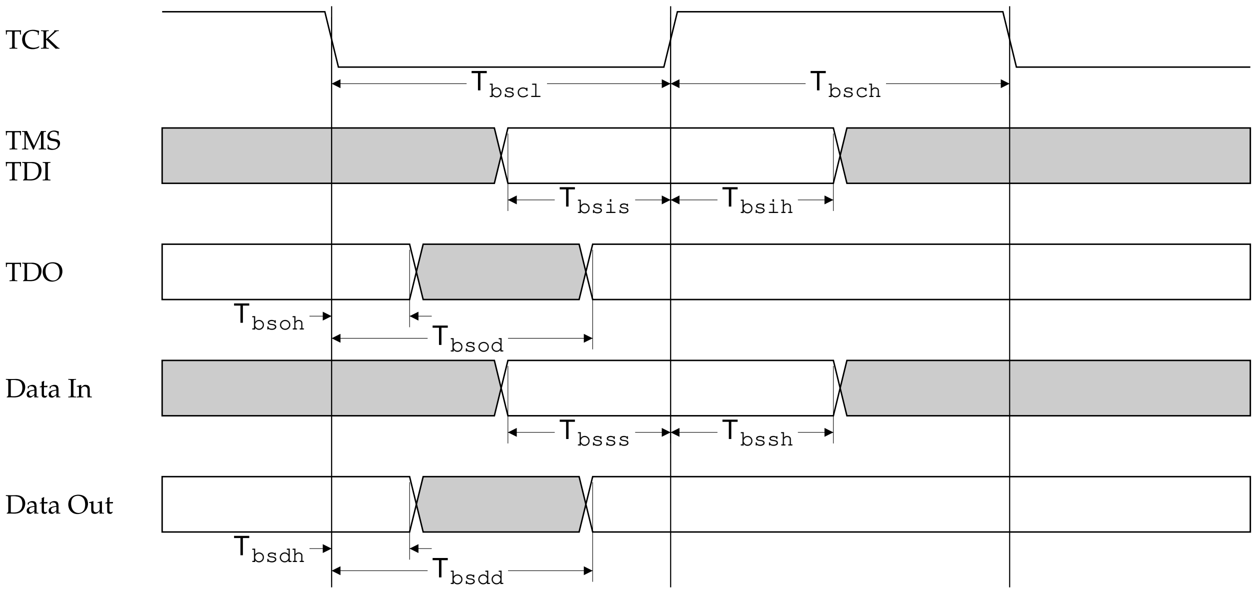


Figure 54: Boundary Scan General Timing

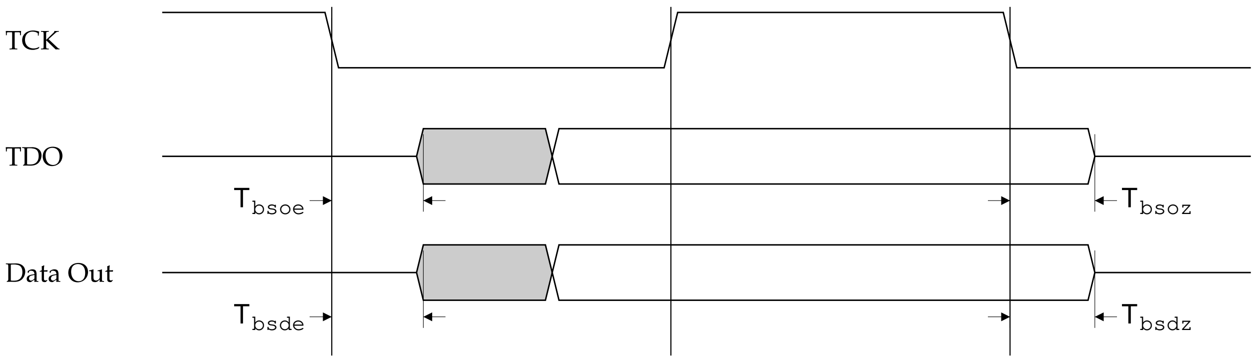


Figure 55: Boundary Scan Tri-state Timing

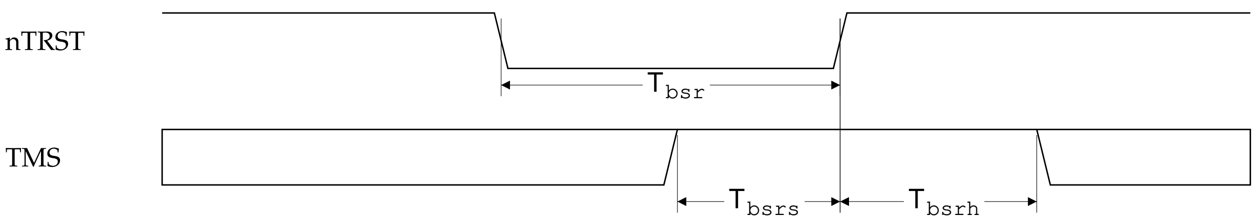


Figure 56: Boundary Scan Reset Timing

Symbol	Parameter	Min	Typ	Max	Units	Notes
Tbscl	**TCK** low period	50			ns	9
Tbsch	**TCK** high period	50			ns	9
Tbsis	**TDI**,**TMS** setup to [TCr]	10			ns	
Tbsih	**TDI**,**TMS** hold from [TCr]	10			ns	
Tbsoh	**TDO** hold time	5			ns	1
Tbsod	TCf to **TDO** valid			40	ns	1
Tbsss	I/O signal setup to [TCr]	5			ns	4
Tbssh	I/O signal hold from [TCr]	20			ns	4
Tbsdh	data output hold time	5			ns	5
Tbsdd	TCf to data output valid			40	ns	
Tbsoe	**TDO** enable time	5			ns	1,2
Tbsoz	**TDO** disable time			40	ns	1,3
Tbsde	data output enable time	5			ns	5,6
Tbsdz	data output disable time			40	ns	5,7
Tbsr	Reset period	30			ns	
Tbsrs	tms setup to [TRr]	10			ns	9
Tbsrh	tms hold from [TRr]	10			ns	9

Table 16: ARM710 Boundary Scan Interface Timing

Notes:

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

No.	Cell Name	Pin	Type	Output enable BS Cell	Guard Value IN	Guard Value EX *
from tdi						
1	a12	A[12]	OUT	Nabe		
2	a11	A[11]	OUT	Nabe		
3	a10	A[10]	OUT	Nabe		
4	a09	A[09]	OUT	Nabe		
5	a08	A[08]	OUT	Nabe		
6	a07	A[07]	OUT	Nabe		
7	a06	A[06]	OUT	Nabe		
8	a05	A[05]	OUT	Nabe		
9	a04	A[04]	OUT	Nabe		
10	a03	A[03]	OUT	Nabe		
11	a02	A[02]	OUT	Nabe		
12	a01	A[01]	OUT	Nabe		
13	a00	A[00]	OUT	Nabe		
14	Nabe	ABE	INEN1	-	0	
15	rlw	LOCK	OUT	Nabe		
16	Nbw	nBW	OUT	Nabe		
17	Nrw	nRW	OUT	Nabe		
18	testbus[7]	TESTIN[15]	IN	-		0
19	testbus[6]	TESTIN[14]	IN	-		0
20	testbus[5]	TESTIN[13]	IN	-		0
21	testbus[3]	TESTIN[11]	IN	-		0
22	testbus[2]	TESTIN[10]	IN	-		0
23	testbus[1]	TESTIN[9]	IN	-		0
24	testbus[0]	TESTIN[8]	IN	-		0
25	din31	D[31]	IN	-		
26	dout31	D[31]	OUT	Nendout		
27	din30	D[30]	IN	-		
28	dout30	D[30]	OUT	Nendout		
29	din29	D[29]	IN	-		
30	dout29	D[29]	OUT	Nendout		
31	din28	D[28]	IN	-		
32	dout28	D[28]	OUT	Nendout		
33	din27	D[27]	IN	-		
34	dout27	D[27]	OUT	Nendout		
34	din26	D[26]	IN	-		
35	dout26	D[26]	OUT	Nendout		
36	din25	D[25]	IN	-		
37	dout25	D[25]	OUT	Nendout		
38	din24	D[24]	IN	-		
39	dout24	D[24]	OUT	Nendout		
40	din23	D[23]	IN	-		
41	dout23	D[23]	OUT	Nendout		
42	din22	D[22]	IN	-		
43	dout22	D[22]	OUT	Nendout		
44	din21	D[21]	IN	-		
45	dout21	D[21]	OUT	Nendout		

No.	Cell Name	Pin	Type	Output enable BS Cell	Guard Value IN	Guard Value EX *
46	din20	D[20]	IN	-		
47	dout20	D[20]	OUT	Nendout		
48	din19	D[19]	IN	-		
49	dout19	D[19]	OUT	Nendout		
50	din18	D[18]	IN	-		
51	dout18	D[18]	OUT	Nendout		
52	din17	D[17]	IN	-		
53	dout17	D[17]	OUT	Nendout		
54	din16	D[16]	IN	-		
55	dout16	D[16]	OUT	Nendout		
56	din15	D[15]	IN	-		
57	dout15	D[15]	OUT	Nendout		
58	din14	D[14]	IN	-		
59	dout14	D[14]	OUT	Nendout		
60	din13	D[13]	IN	-		
61	dout13	D[13]	OUT	Nendout		
62	din12	D[12]	IN	-		
63	dout12	D[12]	OUT	Nendout		
64	din11	D[11]	IN	-		
65	dout11	D[11]	OUT	Nendout		
66	din10	D[10]	IN	-		
67	dout10	D[10]	OUT	Nendout		
68	din9	D[9]	IN	-		
69	dout9	D[9]	OUT	Nendout		
70	Nendout	-	OUTEN0	-	1	
71	din8	D[8]	IN	-		
72	dout8	D[8]	OUT	Nendout		
73	din7	D[7]	IN	-		
74	dout7	D[7]	OUT	Nendout		
75	din6	D[6]	IN	-		
76	dout6	D[6]	OUT	Nendout		
77	din5	D[5]	IN	-		
78	dout5	D[5]	OUT	Nendout		
79	din4	D[4]	IN	-		
80	dout4	D[4]	OUT	Nendout		
81	din3	D[3}	IN	-		
82	dout3	D[3]	OUT	Nendout		
83	din2	D[2]	IN	-		
84	dout2	D[2]	OUT	Nendout		
85	din1	D[1]	IN	-		
86	dout1	D[1]	OUT	Nendout		
87	din0	D[0]	IN	-		
88	dout0	D[0]	OUT	Nendout		
89	dbe	DBE	IN	-		
90	seq	SEQ	OUT	Nmse		
91	Nmreq	nMREQ	OUT	Nmse		
92	Nmse	MSE	INEN1	-	0	

Table 17: Boundary Scan Signals & Pins

No.	Cell Name	Pin	Type	Output enable BS Cell	Guard Value IN	Guard Value EX *
93	sNa	SnA	IN	-		
94	Nwait	nWAIT	IN	-		
95	mclk	MCLK	IN	-		0
96	fclk	FCLK	IN	-		0
97	abort	ABORT	IN	-		
98	Nreset	nRESET	IN	-		
99	testin[16]	TESTIN[16]	IN	-		0
100	testout[2]	TESTOUT[2]	OUT	Ntbe		
101	testout[1]	TESTOUT[1]	OUT	Ntbe		
102	testout[0]	TESTOUT[0]	OUT	Ntbe		
103	Nirq	nIRQ	IN	-		
104	Nfiq	nFIQ	IN	-		
105	Ntbe		-	OUTEN0	-	1
106	ale	ALE	IN	-		
107	a31	A[31]	OUT	Nabe		
108	a30	A[30]	OUT	Nabe		
109	a29	A[29]	OUT	Nabe		
110	a28	A[28]	OUT	Nabe		
111	a27	A[27]	OUT	Nabe		
112	a26	A[26]	OUT	Nabe		
113	a25	A[25]	OUT	Nabe		
114	a24	A[24]	OUT	Nabe		
115	a23	A[23]	OUT	Nabe		
116	a22	A[22]	OUT	Nabe		
117	a21	A[21]	OUT	Nabe		
118	a20	A[20]	OUT	Nabe		
119	a19	A[19]	OUT	Nabe		
120	a18	A[18]	OUT	Nabe		
121	a17	A[17]	OUT	Nabe		
122	a16	A[16]	OUT	Nabe		
123	a15	A[15]	OUT	Nabe		
124	a14	A[14]	OUT	Nabe		
125	a13	A[13]	OUT	Nabe		
126	a12	A[12]	OUT	Nabe		
127	a11	A[11]	OUT	Nabe		
128	a10	A[10]	OUT	Nabe		
129	a09	A[09]	OUT	Nabe		
130	a08	A[08]	OUT	Nabe		
131	a07	A[07]	OUT	Nabe		
132	a06	A[06]	OUT	Nabe		
133	a05	A[05]	OUT	Nabe		
134	a14	A[14]	OUT	Nabe		
135	a13	A[13]	OUT	Nabe		

to TDO

Table 17: Boundary Scan Signals & Pins

Key:

IN	Input pad
OUT	Output pad
INEN1	Input enable active high
OUTENO	Output enable active low
*	for Intest Extest/Clamp

12.0 DC Parameters

***** Subject to Change *****

12.1 Absolute Maximum Ratings

Symbol	Parameter	5V Min	5V Max	Units	Note
VDD	Supply voltage	VSS-0.3	VSS+7.0	V	1
Vip	Voltage applied to any pin	VSS-0.3	VDD+0.3	V	1
Ts	Storage temperature	-40	125	deg C	1

Table 18: ARM710 DC Maximum Ratings

Note:

These are stress ratings only. Exceeding the absolute maximum ratings may permanently damage the device. Operating the device at absolute maximum ratings for extended periods may affect device reliability.

12.2 DC Operating Conditions

Symbol	Parameter	Min	Typ	Max	Units	Notes
VDD	Supply voltage	2.7	3.0 - 5.0	5.5	V	
Vihc	IC input HIGH voltage	.8xVDD		VDD	V	1,2
Vilc	IC input LOW voltage	0.0		0.2xVDD	V	1,2
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 19: ARM710 DC Operating Conditions

Notes:

(1) Voltages measured with respect to VSS.

(2) IC - CMOS inputs (includes IC and ICOCZ pin types)

(3) OCZ - Output, CMOS levels, tri-stateable

12.3 DC Characteristics

Symbol	Parameter	Nom	Units	Note
IDD	Static Supply current	20	A	
Isc	Output short circuit current	100	mA	
Ilu	DC latch-up current	>500	mA	
Iin	IC input leakage current	1	uA	
Ioh	Output HIGH current (Vout = VDD-0.4V)		mA	
Iol	Output LOW current (Vout = VSS+0.4V)		mA	
Cin	Input capacitance		pF	
ESD	HMB model ESD	4	KV	2

Table 20: ARM710 DC Characteristics

Notes:

(1) Nominal values shown are derived from transient analysis simulations.

(2) ESD - 2 KV minimum

13.0 AC Parameters

*** * * Subject to change * * ***

13.1 Test Conditions

The AC timing diagrams presented in this section assume that the outputs of ARM710 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 ARM710 might be employed. The output pads of ARM710 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)
A[31:0]	50	0.072
D[31:0]	50	0.072
nR/W	50	0.072
nB/W	50	0.072
LOCK	50	0.072
nMREQ	50	0.072
SEQ	50	0.072

Table 21: ARM710 AC Test Conditions

13.2 Relationship between FCLK & MCLK in Synchronous Mode

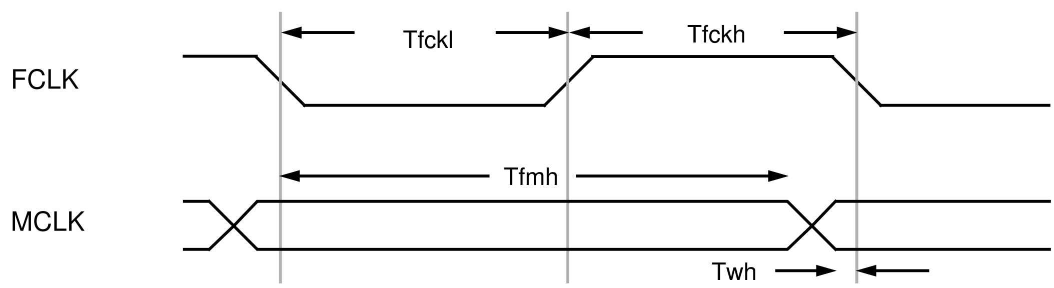


Figure 57: Clock Timing Relationship

Symbol	Parameter	5V Min	5V Max	3V Min	3V Max	Unit	Note
Tfckl	FCLK LOW time	15		20		ns	1
Tfckh	FCLK HIGH time	15		20		ns	1
Tfmh	FCLK - MCLK hold time	20		25		ns	
Tmfs	MCLK - FCLK setup	3		4		ns	

Table 22: ARM710 FCLK and MCLK Synchronous Mode relationship

NB: **FLCK** 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.

13.2.1 Tald Measurement

Tald is the maximum delay allowed in the **ALE** input transition to guarantee the address will not change:

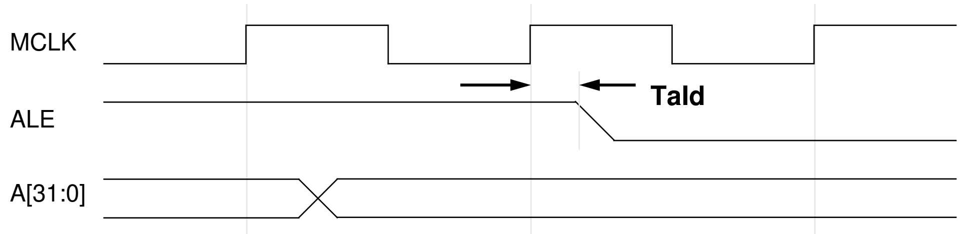


Figure 58: Tald Measurement

13.3 Main Bus Signals

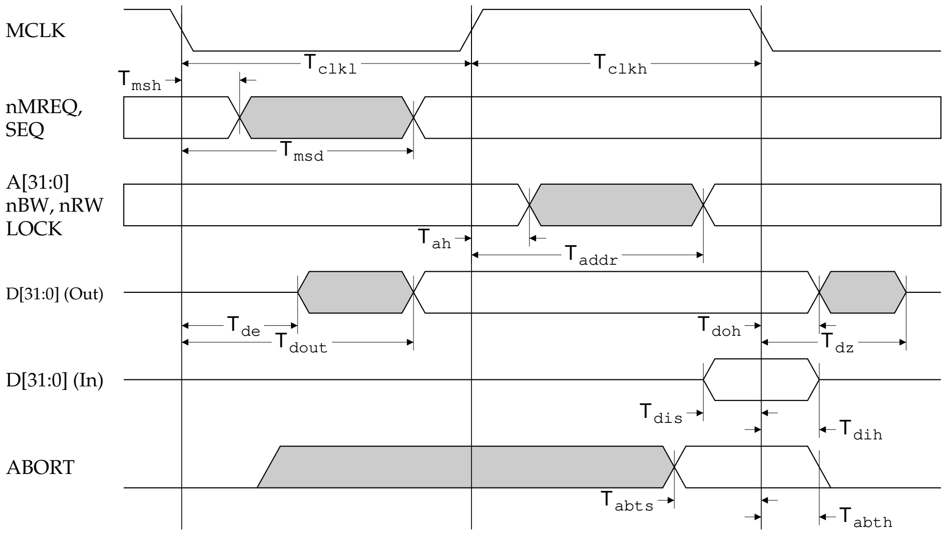


Figure 59: ARM710 Main Bus Timing

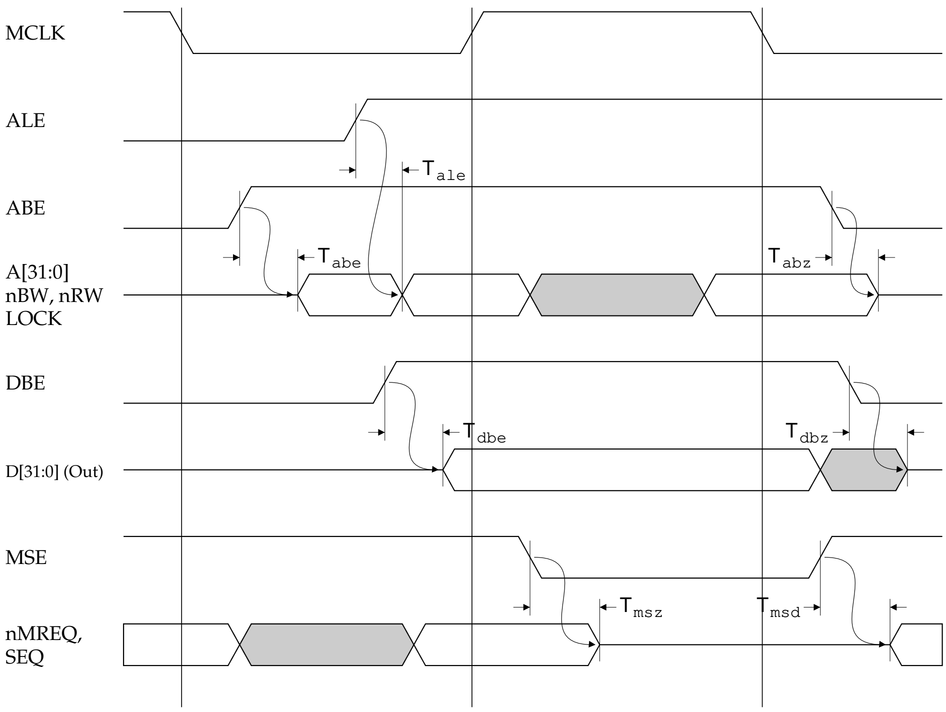


Figure 60: ARM710 Bus Enable Timing

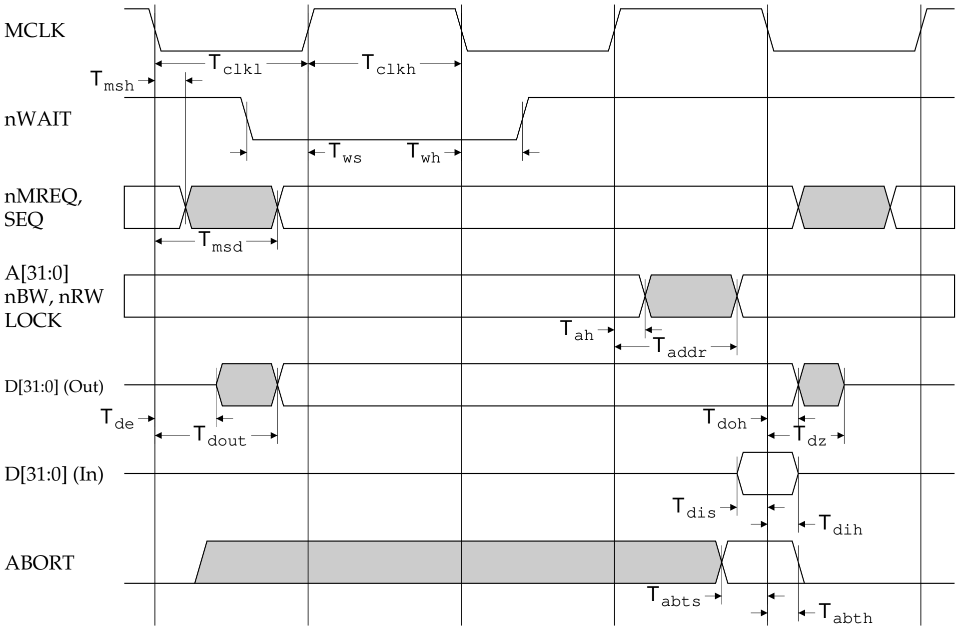


Figure 61: ARM710 nWAIT Timing

Symbol	Parameter	5V Min	5V Max	3V Min	3V Max	Unit	Note
Tmckl	MCLK LOW time	25		40		ns	1
Tmckh	MCLK HIGH time	25		40		ns	
Tws	nWAIT setup to MCLK	5				ns	
Twh	nWAIT hold from MCLK	5				ns	
Tale	address latch enable		2			ns	3
Tald	address latch disable						
Tabe	address bus enable		15			ns	2
Tabz	address bus disable		25			ns	
Taddr	MCLK to address delay		25			ns	2
Tah	address hold time	5				ns	2
Tah	address hold time	5				ns	2
Tdbe	DBE to data enable		15			ns	2
Tde	MCLK to data enable	8				ns	2
Tdbz	DBE to data disable		25			ns	

Table 23: ARM710 Bus timing

Symbol	Parameter	5V Min	5V Max	3V Min	3V Max	Unit	Note
Tdz	MCLK to data disable		25			ns	
Tdout	data out delay		32			ns	2
Tdoh	data out hold	5				ns	2
Tdis	data in setup	2				ns	
Tdih	data in hold	10				ns	
Tabts	ABORT setup time	10				ns	
Tabth	ABORT hold time	5				ns	
Tmse	nMREQ & SEQ enable		10			ns	
Tmsz	nMREQ & SEQ disable		20			ns	
Tmsd	nMREQ & SEQ delay		35			ns	
Tmsh	nMREQ & SEQ hold	5				ns	

Table 23: ARM710 Bus timing

Notes:

(1) **MCLK** timings measured between clock edges at 50% of Vdd.

(2) The timings of these buses are measured to TTL levels.

(3) See *13.2.1 Tald Measurement*.

14.0 Physical Details

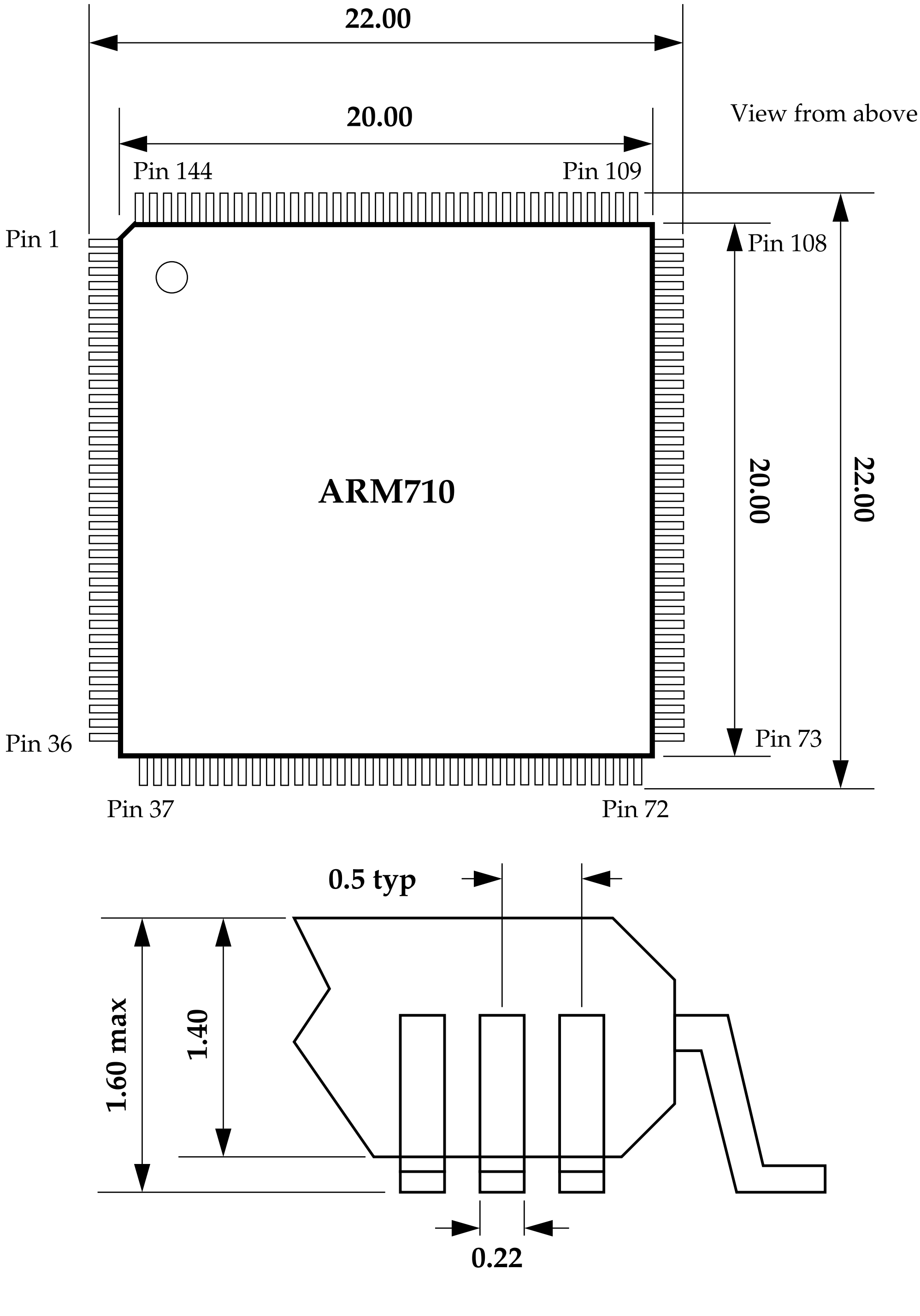


Figure 62: ARM710 144 Pin TQFP Mechanical Dimensions in mm

15.0 Pinout

Pin	Signal	Type
1	MSE	i
2	nMREQ	o
3	SEQ	o
4	DBE	i
5	Vss2	-
6	Vdd2	-
7	D[0]	i/o
8	D[1]	i/o
9	D[2]	i/o
10	D[3]	i/o
11	D[4]	i/o
12	D[5]	i/o
13	D[6]	i/o
14	D[7]	i/o
15	D[8]	i/o
16	Vss2	-
17	Vdd2	-
18	Vss1	-
19	Vdd1	-
20	D[9]	i/o
21	D[10]	i/o
22	D[11]	i/o
23	D[12]	i/o
24	D[13]	i/o
25	D[14]	i/o
26	D[15]	i/o
27	D[16]	i/o
28	D[17]	i/o
29	D[18]	i/o
30	D[19]	i/o
31	Vdd2	-
32	Vss2	-
33	D[20]	i/o
34	D[21]	i/o
35	D[22]	i/o
36	D[23]	i/o
37	D[24]	i/o
38	D[25]	i/o
39	D[26]	i/o
40	Vss1	-
41	Vss2	-
42	Vdd2	-
43	D[27]	i/o
44	D[28]	i/o
45	D[29]	i/o
46	D[30]	i/o
47	D[31]	i/o
48	TDO	o
49	TDI	i
50	nTRST	i
51	Vdd1	-
52	TMS	i
53	TCK	i
54	n/c	-
55	n/c	-
56	n/c	-
57	n/c	-
58	n/c	-
59	TESTIN[8]	i
60	TESTIN[9]	i
61	Vdd1	-
62	Vss1	-
63	TESTIN[10]	i
64	TESTIN[11]	i
65	TESTIN[12]	i
66	TESTIN[13]	i
67	TESTIN[14]	i
68	TESTIN[15]	i
69	Vss2	-
70	Vdd2	-
71	nR/W	o
72	nB/W	o
73	LOCK	o
74	ABE	i
75	A[0]	o
76	A[1]	o
77	A[2]	o
78	Vss2	-
79	Vdd2	-
80	A[3]	o
81	A[4]	o
82	A[5]	o
83	A[6]	o
84	A[7]	o
85	A[8]	o
86	A[9]	o
87	A[10]	o
88	A[11]	o
89	A[12]	o
90	Vdd2	-
91	Vss1	-
92	Vdd1	-
93	Vss2	-
94	A[13]	o
95	A[14]	o
96	A[15]	o
97	A[16]	o
98	A[17]	o
99	A[18]	o
100	A[19]	
101	A[20]	o
102	Vdd2	-
103	Vss2	-
104	A[21]	o
105	A[22]	o
106	A[23]	o
107	A[24]	o
108	A[25]	o
109	A[26]	o
110	A[27]	o
111	A[28]	o
112	Vdd2	-
133	Vss2	-
114	A[29]	o
115	A[30]	o
116	A[31]	o
117	ALE	i
118	n/c	
119	n/c	
120	n/c	
121	Vss1	-
122	Vdd1	-
123	TESTIN[7]	i
124	TESTIN[6]	i
125	TESTIN[5]	i
126	TESTIN[4]	i
127	TESTIN[3]	i
128	TESTIN[2]	i
129	TESTIN[1]	i
130	TESTIN[0]	i
131	nFIQ	
132	nIRQ	
133	TESTOUT[0]	o
134	TESTOUT[1]	o
135	TESTOUT[2]	o
136	TESTIN[16]	i
137	nRESET	i
138	ABORT	i
139	FCLK	i
140	MCLK	i
141	Vdd2	-
142	Vss2	-
143	nWAIT	i
144	SnA	i

Table 24: Pinout - ARM710 in 144 pin Thin Quad Flat Pack

16.0 Appendix - Backward Compatibility

Two of the Control Register bits, prog32 and data32, allow one of three processor configurations to be selected as follows:

(1) **26 bit program and data space** - (prog32 LOW, data32 LOW). This configuration forces ARM710 to operate like the earlier ARM processors with 26 bit address space. The programmer's model for these processors applies, but the new instructions to access the CPSR and SPSR registers operate as detailed elsewhere in this document. In this configuration it is impossible to select a 32 bit operating mode, and all exceptions (including address exceptions) enter the exception handler in the appropriate 26 bit mode.

(2) **26 bit program space and 32 bit data space** - (prog32 LOW, data32 HIGH). This is the same as the 26 bit program and data space configuration, but with address exceptions disabled to allow data transfer operations to access the full 32 bit address space.

(3) **32 bit program and data space** - (prog32 HIGH, data32 HIGH). This configuration extends the address space to 32 bits, introduces major changes in the programmer's model as described below and provides support for running existing 26 bit programs in the 32 bit environment.

The fourth processor configuration which is possible (26 bit data space and 32 bit program space) should not be selected.

When configured for 26 bit program space, ARM710 is limited to operating in one of four modes known as the 26 bit modes. These modes correspond to the modes of the earlier ARM processors and are known as:

User26

FIQ26

IRQ26 and

Supervisor26.

These are the normal operating modes in this configuration and the 26 bit modes are only provided for backwards compatibility to allow execution of programs originally written for earlier ARM processors.

The differences between ARM710 and the earlier ARM processors are documented in an *ARM Application Note 11 - "Differences between ARM6 and earlier ARM Processors"*