ARM 8

Data Sheet

Document Number: ARM DDI0080C

Issued: July 1996

ARM POWERED ™

ENGLAND
Advanced RISC Machines Limited
Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN
UK

Telephone:	+44 1223 400400
Facsimile:	+44 1223 400410
Email:	info@armltd.co.uk

GERMANY
Advanced RISC Machines Limited
Otto-Hahn Str. 13b
85521 Ottobrunn-Riemerling
Munich
Germany

Telephone:	+49 89 608 75545
Facsimile:	+49 89 608 75599
Email:	info@armltd.co.uk

JAPAN
Advanced RISC Machines K.K.
KSP West Bldg, 3F 300D, 3-2-1 Sakado
Takatsu-ku, Kawasaki-shi
Kanagawa
213 Japan

Telephone:	+81 44 850 1301
Facsimile:	+81 44 850 1308
Email:	info@armltd.co.uk

USA
ARM USA Incorporated
Suite 5
985 University Avenue
Los Gatos
CA 95030 USA

Telephone:	+1 408 399 5199
Facsimile:	+1 408 399 8854
Email:	info@arm.com

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

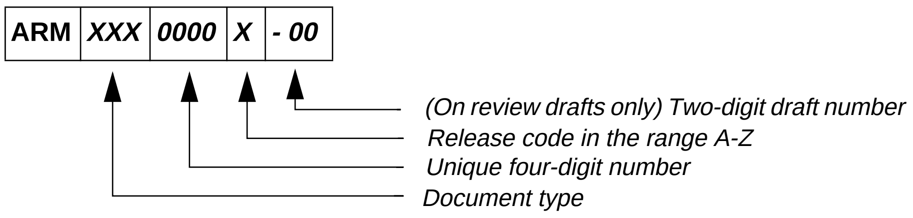
ARM
Advanced RISC Machines

Open Access

Key

Document Number

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


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

Document Status

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

Confidentiality status is one of:

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

Information status is one of:

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

Change Log

Issue	Date	By	Change
A	Dec 1994	EH/BJH	Created
- 00	Jan 1995	BH/AW	Preliminary Draft finalised
B	July 1995	SFK	Minor edits
C	July 1996	KTB	Edits and reformatting

Contents

1 Introduction

This chapter introduces the ARM8 processor.

Introduction

1.1 Overview of ARM8

The ARM8 is part of the Advanced RISC Machines (ARM) family of general purpose 32-bit microprocessors, which offer very low-power consumption and price for high-performance devices. The architecture is based on Reduced Instruction Set Computer (RISC) principles, and the instruction set and related decode mechanism are much simpler than those of microprogrammed Complex Instruction Set Computers. This simplicity results in a high instruction throughput and impressive real-time interrupt response from a small and cost-effective chip.

The ARM8 processor is a fully static CMOS implementation of the ARM which allows the clock to be stopped in any part of the cycle with extremely low residual power consumption and no loss of state. It has been designed to provide better system performance through the new implementation and internal architecture design, giving lower average cycles per instruction (CPI) and a higher clock speed than previous ARM implementations.

The inclusion of a branch predicting Prefetch Unit and a double bandwidth on-chip memory interface means that the overall CPI and power consumption of the Core are reduced. This is mainly because fewer instructions are passed to the Core: some branches can be removed from the instruction stream altogether.

The ARM8 design is optimised for use in systems that provide fast on-chip memory (such as a cache), and can take advantage of the double-bandwidth interface. Control signals are provided that make power-efficient use of on-chip memory accesses. When used on its own, the ARM8 may exhibit performance degradations should the memory system be incapable of supplying more than an average of one word per cycle.

The memory interface has been designed to maximise performance without incurring high costs in the memory system. Speed-critical control signals are pipelined to allow system control functions to be implemented in standard low-power logic, and these control signals facilitate the exploitation of on-chip memory.

ARM8 has a 32-bit address bus, but can be operated in 26-bit modes for backwards compatibility with earlier processors.

1.2 ARM8 Instruction Set

The ARM8 uses the ARM instruction set, which comprises eleven basic instruction types:

- Two perform high-speed operations on data in a bank of thirty one 32-bit registers, using the on-chip arithmetic logic unit, shifter and multiplier.
- Three control data transfer between the registers and memory, one optimised for flexibility of addressing, another for rapid context switching and the third for swapping data.
- Three adjust the flow, privilege level and system control parameters of execution.
- Three are dedicated to the control of coprocessors which allow the functionality of the instruction set to be extended in an open and uniform way. Note that ARM8 only supports register transfers between ARM8 and coprocessors.

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

ARM8 employs a Prefetch Buffer, along with instruction and data pipelining, to ensure that all parts of the processing and memory systems can operate continuously. Typically, while one instruction is being executed:

- its successor is being decoded
- the third is being fetched from the Prefetch Buffer
- further instructions are being prefetched from memory
- data for previous instructions is being read/written

1.3 Architecture

The ARM8 consists of a Core and a *Prefetch Unit* (PU). An ARM8 system would typically comprise the ARM8 and some fast on-chip memory. The memory system is expected to be able to provide the Core (or the PU) with two words of data (or instructions) on each cycle. These double-bandwidth transfers may be supplied with no wait states for maximum performance, or with wait states at a reduced overall performance.

The Core and PU can be forced to make single-bandwidth transfers (at a reduced performance) should the memory system dictate this.

1.4 The ARM8 Prefetch Unit (PU)

The presence of the double-bandwidth interface to on-chip memory means that instructions can be prefetched (and pre-processed) ahead of the Core. The ARM8 Prefetch Unit (PU) prefetches and buffers instructions, and makes use of the extra bandwidth by removing some of the Branches from the instruction stream altogether - giving them a CPI of zero. The removal procedure uses a scheme that predicts whether or not a Branch will be taken. If a Branch is predicted taken, then its destination address is calculated, and further instructions are fetched from there.

Some branches *cannot* be predicted, because the prediction takes one or more cycles, and occasionally the PU becomes empty. Around 65% of all Branches are predictable, and calculations show that the average Branch CPI can be reduced to around 1.0 compared to ARM7's average Branch CPI of about 2.4.

1.5 The ARM8 Core

The most significant change to the Core when compared to earlier ARM processors is the extension of the data pipeline to four stages, making the ARM8 a processor with a 5-stage pipeline. This means that execution is spread over more cycles, reducing the amount of work done at each stage and thus allowing the use of higher clock rates.

The four-stage data pipeline requires more careful instruction scheduling to maximise performance: for example, a cycle is wasted each time an LDR instruction is followed immediately by an instruction that uses the loaded value. Improvements to the ARM C compiler will take account of this wherever possible.

In addition to the pipeline changes, the shifter and adder now operate in parallel rather than in series. This also reduces the Core cycle time (since the adder and shifter no longer contribute to the same cycle) but means that the shifter and adder cannot be used in the same cycle. The penalty for this change is an extra cycle for instructions that require both the adder and some shifter operations, since the result of the shifter goes through the adder in the following cycle (note that this does not apply to simple left-shifts by 0, 1, 2 or 3). Usually, under 1% of instructions require the extra cycle (rising to 10% for some code). The increase in clock speed resulting from the parallel arrangement , however, outweighs this extra cycle penalty.

Double-bandwidth reads to the on-chip memory reduces the average Load Multiple CPI value by nearly a factor of 1.5 for most software. Also, the single Load and Store instructions have been reduced to a single cycle for the normal cases. (This assumes zero-wait-state on-chip memory and no interlocking.)

The ARM8 multiplier is bigger than that of ARM7 and similar (in algorithm) to that of ARM70DM. It operates on 8 bits per cycle.

Existing code will run on ARM8 with only a few rare exceptions as a result of the above changes. Please refer to ***Chapter 11, Backward Compatibility*** for further information.

Instruction set changes and additions are detailed in this datasheet and are summarised in ***Appendix A, Instruction Set Changes***.

1.6 Interfaces

1.6.1 Coprocessor

The Coprocessor interface is entirely on-chip. This means that existing coprocessors are not directly supported.

The on-chip interface provides a means to support all of the current coprocessor instructions in hardware should software emulation be too slow. Existing coprocessors may be connected via an on-chip “interfacing coprocessor” should the functionality of this off-chip device be required. Such an “interfacing coprocessor” would conform to the ARM8 coprocessor interface on one side, and the off-chip coprocessor on the other. This interfacing coprocessor is not implemented as part of ARM8.

Note: ARM8 only supports register-transfers in the coprocessor interface.

1.6.2 Hardware Debug/Scan Support

ARM8 does not support this feature.

1.7 ARM8 Core Block Diagram

The block diagram for the ARM8 core is shown in ***Figure 1-1: ARM8 data-flow block diagram*** on page 1-8.

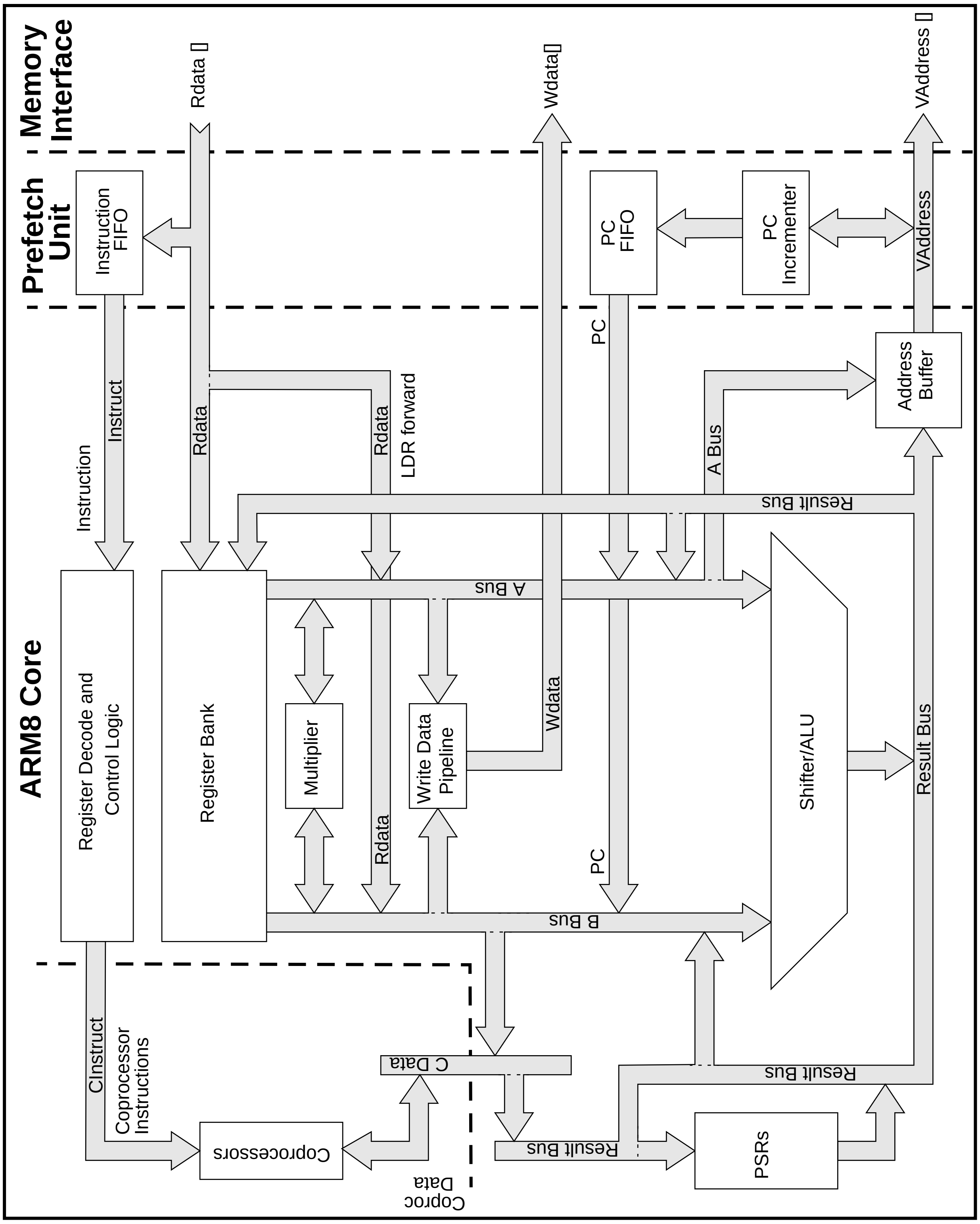


Figure 1-1: ARM8 data-flow block diagram

2 Signal Description

This chapter lists the ARM8 signal descriptions.

Signal Description

In the following tables, Phase 1 refers to the time that **gclk** is LOW, and Phase 2 to the time that **gclk** is HIGH.

2.1 Clocking Signals

Signal	I/O	Description
gclk	IN	**Global clock input** This clock signal drives the ARM8 core, the Prefetch Unit, and all coprocessors that are present.
Confirm	IN	**Stable Phase 1, Changes Phase 2** This signal is used to gate the internal clock of ARM8 in order to stop ARM8 continuing when the Memory System is unable to deliver what was requested of it in time. During Phase 1, if **NConfirm** is HIGH, the ARM8's internal clock is stopped from going low. If **NConfirm** is LOW during Phase1 the ARM8's internal clock continues as normal (ie. is **gclk**)

Table 2-1: Clocking signals

2.2 Configuration and Control Signals

Signal	I/O	Description
BIGEND	IN	**Stable Phase 1, Changes Phase 2** When this signal is HIGH, the processor treats bytes in memory as being in big-endian format. When it is LOW, little-endian format is assumed. This is a hardware configuration signal, and is expected to be hardwired to a chosen value. If it is required to change this during operation, then ensure that no data memory accesses occur during or immediately after that change.
ISYNC	IN	**Stable Phase 1, Changes Phase 2** This configuration signal controls whether the **nFIQ** and **nIRQ** signal inputs should be synchronised by ARM8. When this signal is LOW then the interrupts are synchronised by ARM8, otherwise they are not.
nFIQ	IN	**Asynchronous (ISYNC = 0), Synchronous (ISYNC = 1): Changes Phase 1, Stable Phase 2** This is the Active LOW Fast Interrupt request to ARM8. When driven LOW it interrupts the processor if the appropriate interrupt enable bit of the CPSR is active (LOW). This is a level-sensitive input, and must be held LOW until the processor provides a suitable response.
nIRQ	IN	**Asynchronous (ISYNC = 0), Synchronous (ISYNC = 1): Changes Phase 1, Stable Phase 2** This is the Active LOW Interrupt Request to ARM8. This behaves in the same way as does the **nFIQ** signal, but has a lower priority than **nFIQ**. When driven LOW, the processor will be interrupted if the appropriate interrupt enable bit of the CPSR is active (LOW).
PredictOn	IN	**Stable Phase 1, Changes Phase 2** When this signal is HIGH, Branch Prediction is turned ON. When it is LOW, no Branch Prediction takes place, and all branches are passed to the core.
nReset	IN	**Asynchronous** This is a level-sensitive input signal which starts the processor from a known address. A low level makes the processor abnormally terminate the execution of any current instruction. When **nRESET** becomes HIGH for at least one clock cycle, the processor will restart from address 0. **nRESET** must remain low for at least two clock cycles.

Table 2-2: Configuration and control signals

2.3 ARM8 <-> Memory Interface Signals

Please refer to ***Chapter 6, Memory Interface***, for timing details and further information, in particular on the values that **ARequest[]**, **AResponse[]**, **RRequest[]** and **AResponse[]** can take.

Signal	I/O	Description
VAddress[31:0]	OUT	**Stable Phase 1, Changes Phase 2** This is a single bandwidth bus that is driven by the ARM8 to the memory system. It provides the Virtual Addresses whenever the memory system requires it. At the end of any Phase 2, when **ARequest[]** is not AREQ_NONE, this bus contains the address associated with the ACCESS Request. **VAddress[]** is also driven with coprocessor data during the execute stage of an MCR instruction. This additional functionality can reduce the routing requirements of the CData bus in some external memory systems.
Wdata[31:0]	OUT	**Changes Phase 1, Stable Phase 2** This is a single bandwidth bus that is driven by the ARM8 to the memory system. It provides the data values for a Store operation to the Address on **VAddress[]**. The value of **Wdata[31:0]** will be left unchanged in other Phase 1s.
Privileged	OUT	†**Stable Phase 1, Changes Phase 2** At the end of any Phase 2 when **ARequest[]** is not AREQ_NONE, this signal indicates whether the memory access is being made from a privileged (HIGH) or from User (LOW) mode. Its value at the end of any other Phase 2 should be ignored by the memory system.
TwentySixBit	OUT	**Stable Phase 1, Changes Phase 2** At the end of any Phase 2 when **ARequest[]** is not AREQ_NONE, this signal indicates whether the memory access originates from a 26-bit mode (HIGH) or a 32-bit mode (LOW). Its value at the end of any other Phase 2 should be ignored by the memory system.
WdataOE	IN	**Asynchronous** This input signal directly controls whether ARM8 drives the **Wdata[]** bus (**WdataOE** HIGH) or does not drive **Wdata[]** (**WdataOE** LOW). External memory systems may make use of this functionality to implement **Wdata[]** as a multi-source bus.
ARequest[]	OUT	†**Stable Phase 1, Changes Phase 2** This control bus contains the ARM8 Request to the memory system and indicates how it should use the **VAddress[]** and **Wdata[]** buses during the next cycle.
AResponse[]	IN	**Changes Phase 1, Stable Phase 2** During Phase 1 this control bus changes to reflect a provisional response to the **ARequest[]** made at the end of the previous Phase 2.

Table 2-3: ARM8 <-> Memory interface signals

Signal	I/O	Description
IExhausted	IN	**Changes Phase 1, Stable Phase 2** During Phase 1 the memory system changes this signal to a provisional indication of whether the ARM8 can request further Sequential Instructions from the instruction buffer, using **RRequestIC** or **RRequestIP** without the need to issue an **ARequest[]** as well.
DExhausted	IN	**Changes Phase 1,Stable Phase 2** During Phase 1 the memory system changes this signal to a provisional indication of whether the ARM8 can request further Sequential Data (during LDM instructions) from the data buffer, using **RRequestD[]** without the need to issue an **ARequest[]** as well.
Rdata[31:0]	IN	**Changes Phase 1 and Phase 2** This is a double bandwidth bus that is driven by the Memory System. Data or Instructions (as specified by the previous **RRequestD[]** and **NRRequestI**) are returned to the ARM8. The first word is driven onto **Rdata[]** during, and is sampled at the end of, Phase 2. The second word, if it exists, is driven onto **Rdata[]** during, and is sampled at the end of the following, Phase 1.The **RResponse[]** control bus indicates what is being returned on **Rdata[]**.
RRequestD[]	OUT	†**Stable Phase 1, Changes Phase 2** This control bus contains the ARM8 request to the memory system for what DATA to return on the **Rdata[]** bus during the next cycle. This request may ask for none, one or two Data words from the memory system.
RRequestIC	OUT	†**Stable Phase 1, Changes Phase 2** This control signal indicates the ARM8 core's request for instructions to be returned on the **Rdata[]** bus during the next cycle. When this signal is zero at the end of Phase 2, no instruction return request is being made by the core. When it is one, a return request for two instructions is being made by the core.
RRequestIP	OUT	†**Stable Phase 1, Changes Phase 2** This control signal indicates the ARM8 prefetch unit's request for instructions be returned on the **Rdata[]** bus during the next cycle. When this signal is zero at the end of Phase 2, no instruction return request is being made by the prefetch unt. When it is one, a return request for two instructions is being made by the prefetch unit.
RResponse[]	IN	**Changes Phase 1, Stable Phase 2.** During Phase 1 this control bus changes to reflect a provisional response to the **RRequestD[], RRequestIC** and **RRequestIP** made at the end of the previous Phase 2.

Table 2-3: ARM8 <-> Memory interface signals (Continued)

†These signals can change in Phase 1 if **Confirm** is taken LOW and either **RResponse[]** or **AResponse[]** are changed by the memory system.

2.4 ARM8 <-> Co-processor Interface Signals

Please refer to ***Chapter 7, Coprocessor Interface*** for detailed timing and further information.

Signal	I/O	Description
CData[31:0]	IN/OUT†	**Changes Phase 1, Stable Phase 2** This is a 32-bit bidirectional bus that changes in Phase 1. When an MRC instruction is in the Execute stage of the ARM8, then this bus will be driven in Phase 1 to transfer register data from the Coprocessor to the ARM8. ARM8 is designed such that for an MCR instruction, the data to transfer to the coprocessor is put onto the VAddress bus and optionally onto the CData bus. In the first implementation of ARM8 the CData bus is not used, and the bus is treated as input-only. †In the first implementation of ARM8, CData is an input-only bus.
CInstruct[25:0]	OUT	**Changes Phase 1, Stable Phase 2** When an instruction enters the Decode stage of the ARM8 at the end of Phase 2, then during the following Phase 1: • If it is a Coprocessor Instruction then **CInstruct[25:0]** become the corresponding bits of the instruction. • If it is NOT a Coprocessor instruction then **CInstruct[25:24]** become "11" and **CInstruct[23:0]** are undefined. When no instruction enters the Decode stage of ARM8 at the end of Phase 2, then all of **CInstruct[25:0]** become undefined during the following Phase 1.
CEnterDecode	OUT	**Changes Phase 1, Stable Phase 2** This control signal is set during Phase 1 if an instruction entered the Decode stage of ARM8 at the end of the preceding Phase 2.
CEnterExecute	OUT	**Changes Phase 1, Stable Phase 2** This control signal is set during Phase 1 if an instruction entered the Execute stage of ARM8 at the end of the preceding Phase 2.
CExecute	OUT	**Changes Phase 1, Stable Phase 2** During the Phase 1, one cycle after **CEnterExecute** is asserted, or 3 phases after the Coprocessor asserts **CBusyWaitE**, this control signal is set HIGH if the instruction should complete its execution or brought LOW if it should not. If it is brought LOW, no permanent change to the Coprocessor state must take place as a result of that instruction.
CBounceD	IN	**Stable Phase 1, Changes Phase 2** This control signal is set during Phase 2 to bounce the instruction in the Decode stage of ARM8 if it enters the Execute stage of ARM8 at the end of the next Phase 1 (i.e. as a result of **CEnterExecute**=1). Otherwise the signal is cleared to indicate that the instruction should not bounce.

Table 2-4: ARM8 <-> Coprocessor interface signals

Signal	I/O	Description
CBounceE	IN	**Stable Phase 1, Changes Phase 2** When the instruction in the Execute stage of ARM8 is busy-waiting, this signal is set in Phase 2 to bounce it at the end of the following Phase 1, or cleared to indicate that it should not bounce. When the instruction in the Execute stage of the ARM8 is NOT busy-waiting, this signal must be cleared during Phase 2. This means that this mechanism may not be used to bounce instructions in the Execute stage of ARM8 unless they are busy-waiting.
CBusyWaitD	IN	**Stable Phase 1, Changes Phase 2** This control signal is set in Phase 2 to indicate that the instruction in the Decode stage of the ARM8 will require busy-waiting if it enters the Execute stage at the end of the next Phase 1 (i.e. as a result of **CEnterExecute**=1). The signal is cleared to indicate that no busy-waiting should occur.
CBusyWaitE	IN	**Stable Phase 1, Changes Phase 2** When the instruction in the Execute stage of ARM8 is busy-waiting, this signal is set in Phase 2 to indicate that it should continue to busy-wait at the end of the following Phase 1, or cleared to indicate that it should not. When the instruction in the Execute stage of the ARM8 is NOT busy-waiting, this signal must be cleared during Phase 2. This means that busy-waiting may not be restarted once it has finished, and if **CBusyWaitD** was not asserted as the instruction was entering the Execute stage, then the instruction cannot be busy-waited at all.
Interlock	OUT	**Stable Phase 1, Changes Phase 2** If this signal becomes 1 during any Phase 2, this indicates that the ARM8 will become interlocked in the immediately following cycle. The coprocessor must delay its use of **CExecute** and **CData[]**. This signal must not be evaluated until the END of Phase 1.
CPrivileged	OUT	**Stable Phase 1, Changes Phase 2** During a Phase 1 in which **CEnterDecode** becomes 1, this signal indicates whether the instruction arriving on **CInstruct[] i**s being executed: • in a privileged mode (1) • in an unprivileged mode (0) Its value in Phase 1s where **CEnterDecode** becomes 0 should be ignored by coprocessors.

Table 2-4: ARM8 <-> Coprocessor interface signals (Continued)

3 Programmer's Model

This chapter describes the operating configurations supported by ARM8. Some are controlled by hardware and are known as *hardware configurations*. Others may be controlled by software and are referred to as *operating modes*.

3.1 Hardware Configurations

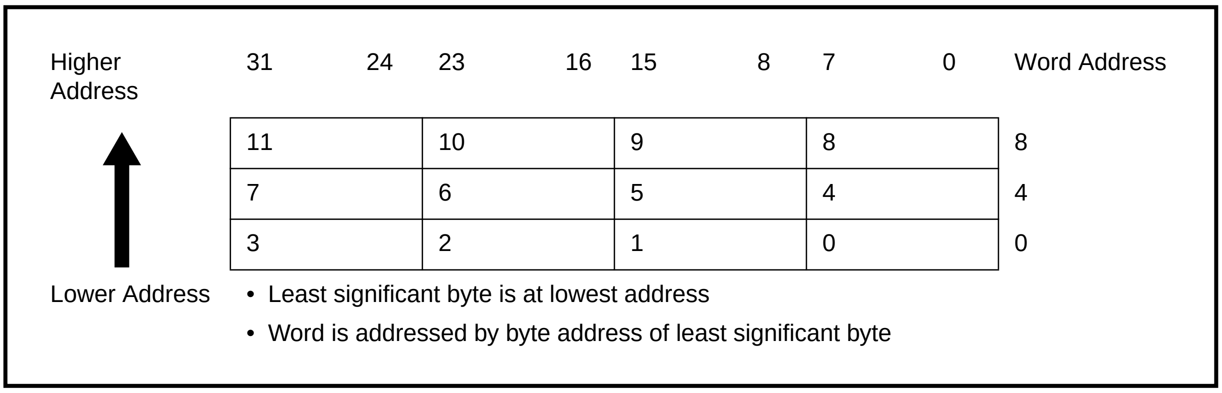
3.1.1 Big- and Little-Endian Memory Formats (the BIGEND Signal)

Memory is viewed as a linear collection of bytes numbered upwards from zero. Bytes 0 to 3 hold the first stored word, bytes 4 to 7 the second and so on. ARM8 can treat words in memory as being stored either in big-endian or little-endian format, depending on the state of the **BIGEND** input.

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

Little-endian format

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



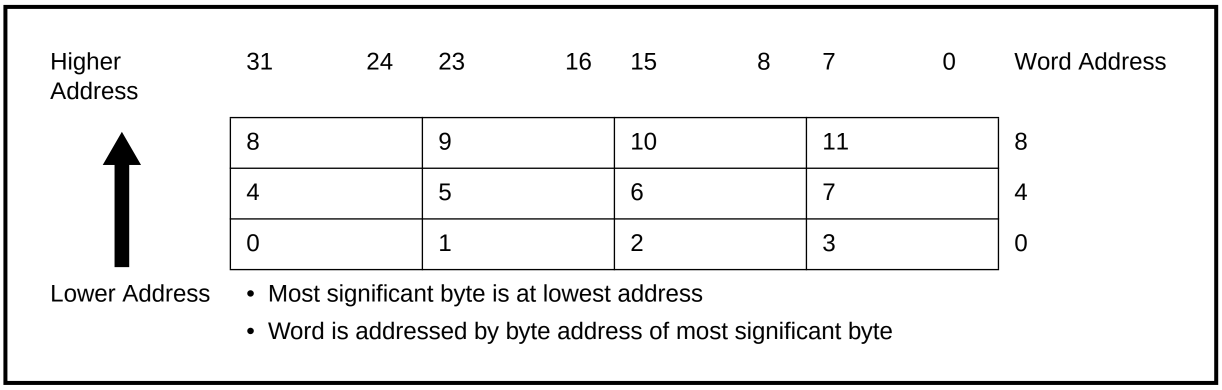
31 24	23 16	15 8	7 0	Word Address
11	10	9	8	8
7	6	5	4	4
3	2	1	0	0

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

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

Big-endian format

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



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

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

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

3.1.2 Interrupt Synchronisation (the ISYNC Signal)

This signal controls the synchronisation of the **nFIQ** and **nIRQ** inputs. When **ISYNC** is LOW, the interrupts are synchronised to the next falling edge of **gclk**. If **ISYNC** is HIGH the inputs are not synchronised by the ARM and so these inputs must be applied in synchrony as directed in ***2.2 Configuration and Control Signals*** on page 2-3.

3.2 Operating Modes

ARM8 supports *byte* (8-bit), *half-word* (16-bit), and *word* (32-bit) data types. Instructions are exactly one word long, and must be aligned to four-byte boundaries. Data operations, such as ADD, are only performed on word quantities. Load and Store operations are able to transfer bytes, half-words or words.

ARM8 supports seven modes of operation:

Mode	**Description**
User mode (usr)	normal program execution state
FIQ mode (fiq)	used for fast or higher priority interrupt handling
IRQ mode (irq)	used for general-purpose interrupt handling
Supervisor mode (svc)	a protected mode for the operating system
System mode (sys)	a privileged user mode for the operating system
Abort mode (abt)	entered after a data or instruction prefetch abort
Undefined mode (und)	entered when an undefined instruction is executed

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

3.3 Registers

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

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

R14 is used as the subroutine link register (LR). This receives a copy of R15 when a Branch and Link (BL) instruction is executed. At all other times it may be treated as a general-purpose register. The corresponding banked registers R14_svc, R14_irq, R14_fiq, R14_abt and R14_und are similarly used to hold the return values of R15 when interrupts and exceptions arise, or when Branch and Link instructions are executed within exception routines.

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

General Registers and Program Counter

System & User	FIQ	Supervisor	Abort	IRQ	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	◣ R8_fiq	R8	R8	R8	R8
R9	◣ R9_fiq	R9	R9	R9	R9
R10	◣ R10_fiq	R10	R10	R10	R10
R11	◣ R11_fiq	R11	R11	R11	R11
R12	◣ R12_fiq	R12	R12	R12	R12
R13	◣ R13_fiq	◣ R13_svc	◣ R13_abt	◣ R13_irq	◣ R13_und
R14	◣ R14_fiq	◣ R14_svc	◣ R14_abt	◣ R14_irq	◣ R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

Program Status Registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	◣ SPSR_fiq	◣ SPSR_svc	◣ SPSR_abt	◣ SPSR_irq	◣ SPSR_und

Figure 3-3: Register organisation

Supervisor, IRQ, Abort and Undefined mode programs which require more than these two banked registers are expected to save some or all of the caller's registers (R0 to R12) on their respective stacks. They are then free to use these registers, which they will restore before returning to the caller.

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

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

3.3.1 Program Status Register Format

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

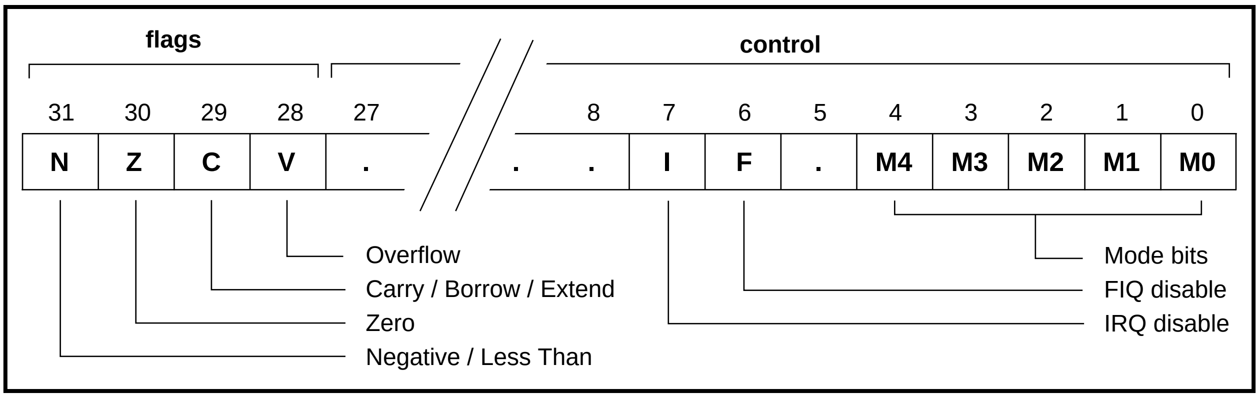


Figure 3-4: Program Status Register (PSR) format

Condition code flags

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

Interrupt disable bits

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

Mode bits

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

Control bits

The bottom 8 bits of a PSR (incorporating I, F and M[4:0]) are known collectively as the *control bits*. These will change when an exception arises. If the processor is operating in a privileged mode, they can also be manipulated by software.

Reserved bits

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

M[4:0]	Mode	Accessible Registers	
10000	User	PC, R14..R0	CPSR
10001	FIQ	PC, R14_fiq..R8_fiq, R7..R0	CPSR, SPSR_fiq
10010	IRQ	PC, R14_irq..R13_irq, R12..R0	CPSR, SPSR_irq
10011	Supervisor	PC, R14_svc..R13_svc, R12..R0	CPSR, SPSR_svc
10111	Abort	PC, R14_abt..R13_abt, R12..R0	CPSR, SPSR_abt
11011	Undefined	PC, R14_und..R13_und, R12..R0	CPSR, SPSR_und
11111	System	PC, R14..R0	CPSR

Table 3-1: The mode bits

3.4 Exceptions

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

When handling an exception, ARM8 makes use of the banked registers to save state. The old PC and CPSR contents are copied into the appropriate R14 and SPSR, and the PC and the CPSR mode bits are forced to a value which depends on the exception. Where necessary, the interrupt disable flags are set to prevent otherwise unmanageable nestings of exceptions; this is detailed in the following sections.

In the case of a re-entrant interrupt handler, R14 and the SPSR should be saved onto a stack in main memory before the interrupt is re-enabled. When transferring the SPSR register to and from a stack, it is important to transfer the whole 32-bit value and not just the flag or control fields. When multiple exceptions arise simultaneously, a fixed priority determines the order in which they are handled: see ***3.4.7 Exception Priorities*** on page 3-13 for more information.

3.4.1 FIQ

The FIQ (Fast Interrupt reQuest) exception is externally generated by taking the **nFIQ** input LOW. This input can accept asynchronous transitions provided that **ISYNC** is LOW so that the ARM will perform the synchronisation. This synchronisation delays the effect of the input transition on the processor execution flow for one cycle. If **ISYNC** is HIGH then all transitions must be made synchronously according to ***2.2 Configuration and Control Signals*** on page 2-3.

FIQ is designed to support a fast or high priority interrupt, and has sufficient private registers to remove the need for register saving in such applications (thus minimising the overhead of context switching). The FIQ exception may be disabled by setting the CPSR's F flag (but note that this is not possible from User mode). If the F flag is clear, ARM8 checks for a LOW level on the FIQ logic output at the end of each instruction (including cancelled ones), and at the end of any coprocessor busy-wait cycle (allowing the busy-wait state to be interrupted).

On detecting a FIQ, ARM8:

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

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

3.4.2 IRQ

The IRQ (Interrupt ReQuest) exception is externally generated by taking the **nIRQ** input LOW. This input can accept asynchronous transitions provided that ISYNC is LOW so that the ARM will perform the synchronisation. This synchronisation delays the effect of the input transition on the processor execution flow for one cycle. If ISYNC is HIGH then all transitions must be made synchronously according to ***2.2 Configuration and Control Signals*** on page 2-3.

IRQ has a lower priority that FIQ and is automatically masked out when a FIQ sequence is entered. The IRQ exception may be disabled by setting the CPSR’s I flag (but note that this is not possible from User mode). If the I flag is clear, ARM8 checks for a LOW level on the IRQ logic output at the end of each instruction (including cancelled ones) and at the end of any coprocessor busy-wait cycle (allowing the busy-wait state to be interrupted).

On detecting an IRQ, ARM8:

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

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

3.4.3 Aborts

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

Abort Sources

Aborts can be generated by Store instructions (STR, STM, SWP (for the write part)), by Data Read instructions (LDR, LDM, SWP (for the read part)) and by Instruction Prefetching. The aborts caused by stores can only be signalled by setting the **AResponse[]** signal to ARESP_ABORT. Data reads and instruction prefetches can signal an abort either through the **AResponse[]** signal or through **RResponse[]** setting RRESP_EXTABORT_I (for instruction fetches) or RRESP_EXTABORT_D (for Data Reads and Swaps).

Please refer to ***Chapter 6, Memory Interface*** for further details on the Response signals.

Note: Although there are two methods for signalling aborts, ARM8 treats both of them identically.

Signalling Aborts

This section gives guidelines on the intended use of the different abort signalling methods that the ARM8 provides. The abort sources can be split into two types:

- Memory Management Aborts
- External Memory Aborts

Memory Management Aborts: When a memory access is made to an address that the Memory Management Unit considers needs further attention by some service routine, then this is signalled to ARM8 in the **AResponse[]** control signal by returning ARESP_ABORT. This may happen, for instance, in a virtual memory system when the data corresponding to the current address has been swapped out to disc. This requires considerable processor activity in order to recover the data from disc before the access can be retried and performed successfully.

External Memory Aborts: When an external memory access generates some form of error, then this can be signalled to the ARM8 in the **RResponse[]** control signal by returning RRESP_EXTABORT_I (if this occurred during an Instruction fetch), or RRESP_EXTABORT_D (if it occurred during a Data Read). This sort of abort may be generated by some Parity-checking hardware, for example, where the Data read may fail the parity check and require some action from the processor in response.

Abort types

Aborts are classified as either Prefetch or Data Abort types depending upon the transaction taking place at the time. Each type has its own exception vector to allow branching to the relevant service routine to deal with them. These exception vectors are the Prefetch Abort Vector and the Data Abort Vector and their locations are summarised in ***3.4.6 Exception Vector Summary*** on page 3-13.

Prefetch Aborts

If the transaction taking place when the abort happened was an Instruction fetch, then a Prefetch Abort is indicated. The instruction is marked as invalid, but the abort exception vector is not taken immediately. Only if the instruction is about to get executed will the Prefetch Abort exception vector be taken. In the case of a marked abort on a prefetched instruction after a predicted branch, if the branch's condition code determines that the branch has been wrongly predicted, then the abort-marked instruction does not fall into the Prefetch Abort trap - it should never have been fetched, as far as the real instruction stream is concerned.

For Prefetch Aborts, ARM8:

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

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

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

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

Data Aborts

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

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

For Data Aborts, ARM8:

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

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

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

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

3.4.4 Software interrupt

The software interrupt instruction (SWI) is used for entering Supervisor mode, usually to request a particular supervisor function. When a SWI is executed, ARM8:

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

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

3.4.5 Undefined instruction trap

When the ARM8 decodes an instruction bit-pattern that it cannot process, it takes the undefined instruction trap.

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

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

When ARM8 takes the undefined instruction trap, it:

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

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

3.4.6 Exception Vector Summary

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

Table 3-2: Exception vectors

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

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

3.4.7 Exception Priorities

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

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

Not all of the exceptions can occur at once: Undefined Instruction and Software Interrupt are mutually exclusive, since they each correspond to particular (non-overlapping) decodings of the current instruction.

If a data abort occurs at the same time as a FIQ, and FIQs are enabled (ie the CPSR’s F flag is clear), ARM8 enters the data abort handler and then immediately proceeds to the FIQ vector. A normal return from FIQ will cause the data abort handler to resume execution. Placing data abort at a higher priority than FIQ is necessary to ensure that the transfer error does not escape detection. The time for this exception entry should be added to worst-case FIQ latency calculations.

3.5 Reset

nRESET can be asserted asynchronously, but must be removed whilst **gclk** is LOW. When the **nRESET** signal goes LOW, ARM8 abandons the executing instruction. When **nRESET** goes HIGH again, ARM8:

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

4 Instruction Set

This chapter details the ARM8 instruction set.

Instruction Set

4.1 Summary

The ARM8 instruction set is summarized below.

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

Figure 4-1: ARM8 instruction set

Note: The instruction cycle times given in this section assume that there is no register interlocking. For details of interlock behaviour, please refer to ***Chapter 8, Instruction Cycle Timings Summary***.

4.2 Reserved Instructions and Usage Restrictions

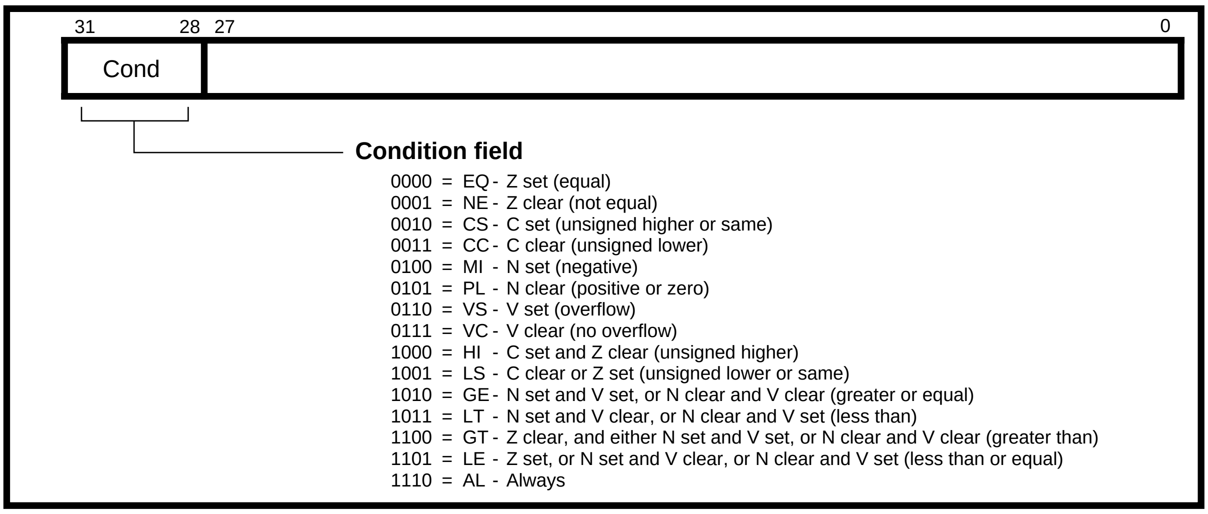
ARM8 enters an Undefined Instruction trap if it encounters an instruction bit pattern that it does not recognize. However, there are some bit patterns which are not defined, but which do not cause the Undefined Instruction trap to be taken. These *reserved* instructions must not be used, as their action may change in future ARM implementations, and may differ from previous ARM implementations.

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

Please refer to ***4.18 Undefined Instructions*** on page 4-65 for details of which instruction bit patterns fall into the Undefined Instruction trap.

4.3 The Condition Field

All ARM8 instructions are conditionally executed. This means that their execution may or may not take place depending on the values of the N, Z, C and V flags in the CPSR. ***Figure 4-2: Condition Codes*** shows the condition encoding.



0000	EQ	Z set (equal)
0001	NE	Z clear (not equal)
0010	CS	C set (unsigned higher or same)
0011	CC	C clear (unsigned lower)
0100	MI	N set (negative)
0101	PL	N clear (positive or zero)
0110	VS	V set (overflow)
0111	VC	V clear (no overflow)
1000	HI	C set and Z clear (unsigned higher)
1001	LS	C clear or Z set (unsigned lower or same)
1010	GE	N set and V set, or N clear and V clear (greater or equal)
1011	LT	N set and V clear, or N clear and V set (less than)
1100	GT	Z clear, and either N set and V set, or N clear and V clear (greater than)
1101	LE	Z set, or N set and V clear, or N clear and V set (less than or equal)
1110	AL	Always

Figure 4-2: Condition Codes

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

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

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

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

4.4 Branch and Branch with Link (B, BL)

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

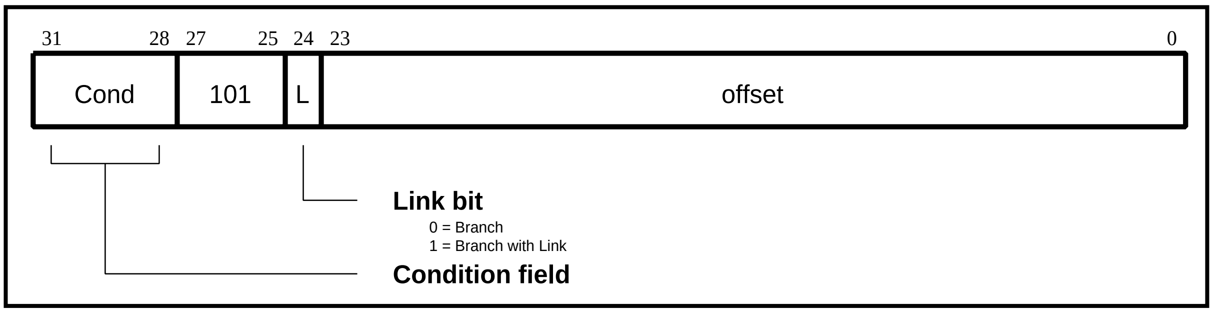


Figure 4-3: Branch instructions

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

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

4.4.1 The link bit

Branch with Link (BL) writes the old PC into the link register (R14) of the current bank. In the process, 4 is subtracted from the PC value, so that R14 will contain the address of the instruction immediately following the BL instruction. The CPSR is not saved with the PC.

To return from a routine called by Branch with Link, use:

`MOV PC,R14`	if the link register is still valid.

or

`LDM Rn!,{..PC}`	if the link register has been saved onto a stack pointed to by Rn.

4.4.2 Branch prediction and removal

The ARM8 Prefetch Unit will attempt to remove a Branch instruction before it reaches the Core. If a Branch is predictable and predicted taken, the Prefetch Unit will start prefetching from the target address, so removing the Branch altogether if predicted correctly. For more information, refer to ***Chapter 5, The Prefetch Unit***.

4.4.3 Instruction cycle times

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

4.4.4 Assembler syntax

Branch instructions have the following syntax:

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

where

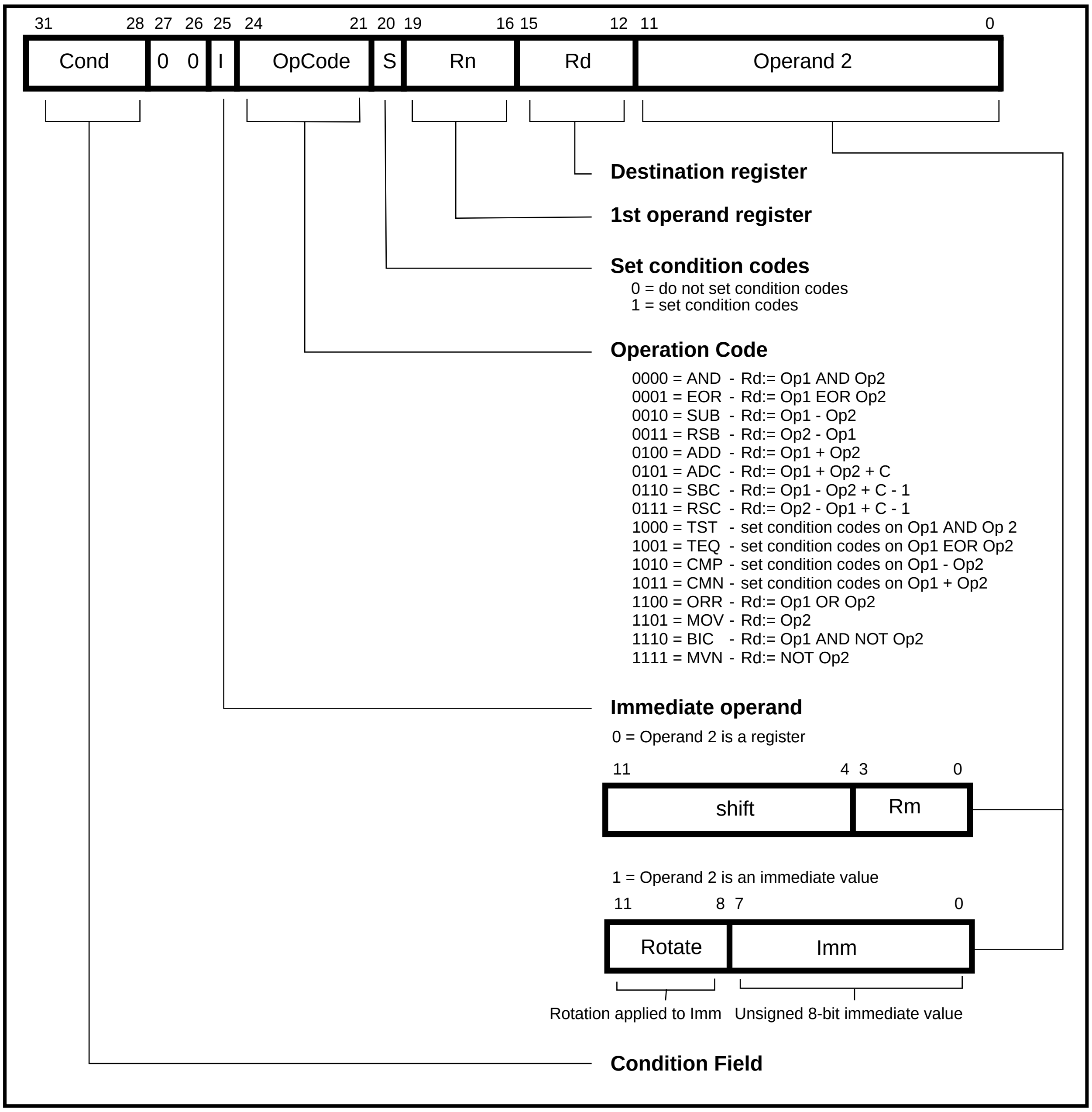
`{L}`	requests a Branch with Link.
`{cond}`	is one of the two-character mnemonics, shown in ***Figure 4-2: Condition Codes*** on page 4-3. The assembler assumes AL (ALways) if no condition is specified.
`<expression>`	is the destination address. The assembler calculates the offset, taking into account that the PC is 8 ahead of the current instruction.

4.4.5 Examples

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

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

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

4.5 Data Processing Instructions

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



31 28 27 26 25 24 21 20 19 16 15 12 11 0
Cond
0 0
I
OpCode
S
Rn
Rd
Operand 2
Destination register
1st operand register
Set condition codes
0 = do not set condition codes
1 = set condition codes
Operation Code
0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1
0100 = ADD - Rd:= Op1 + Op2
0101 = ADC - Rd:= Op1 + Op2 + C
0110 = SBC - Rd:= Op1 - Op2 + C - 1
0111 = RSC - Rd:= Op2 - Op1 + C - 1
1000 = TST - set condition codes on Op1 AND Op 2
1001 = TEQ - set condition codes on Op1 EOR Op2
1010 = CMP - set condition codes on Op1 - Op2
1011 = CMN - set condition codes on Op1 + Op2
1100 = ORR - Rd:= Op1 OR Op2
1101 = MOV - Rd:= Op2
1110 = BIC - Rd:= Op1 AND NOT Op2
1111 = MVN - Rd:= NOT Op2
Immediate operand
0 = Operand 2 is a register
11 4 3 0
shift
Rm
1 = Operand 2 is an immediate value
11 8 7 0
Rotate
Imm
Rotation applied to Imm
Unsigned 8-bit immediate value
Condition Field

Figure 4-4: Data processing instructions

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

- The first operand is always a register (Rn).
- The second operand may be a shifted register (Rm) or a rotated 8-bit immediate value (Imm) depending on the value of the instruction's I bit.

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

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

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

Assembler mnemonic	OpCode	Action	Note
AND	0000	operand1 AND operand2	
EOR	0001	operand1 EOR operand2	
SUB	0010	operand1 - operand2	
RSB	0011	operand2 - operand1	
ADD	0100	operand1 + operand2	
ADC	0101	operand1 + operand2 + carry	
SBC	0110	operand1 - operand2 + carry - 1	
RSC	0111	operand2 - operand1 + carry - 1	
TST	1000	as AND, but result is not written	Rd is ignored and should be 0x0000
TEQ	1001	as EOR, but result is not written	Rd is ignored and should be 0x0000
CMP	1010	as SUB, but result is not written	Rd is ignored and should be 0x0000
CMN	1011	as ADD, but result is not written	Rd is ignored and should be 0x0000
ORR	1100	operand1 OR operand2	
MOV	1101	operand2	Rn is ignored and should be 0x0000
BIC	1110	operand1 AND NOT operand2	Bit clear
MVN	1111	NOT operand2	Rn is ignored and should be 0x0000

Table 4-1: ARM data processing instructions

4.5.1 Effects on CPSR flags

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

Logical operations

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

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

N	is set to the logical value of bit 31 of the result.
Z	is set if and only if the result is all zeros.
C	is set to the carry out from the shifter (so is unchanged when no shift operation occurs - see ***4.5.2 Shifts*** and ***4.5.3 Immediate operand rotates*** for the exact details of this).
V	is preserved.

Arithmetic operations

The arithmetic operations (SUB, RSB, ADD, ADC, SBC, RSC, CMP, CMN) treat each operand as a 32-bit integer (either unsigned or 2's complement signed).

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

N	is set to the value of bit 31 of the result. This indicates a negative result if the operands are being treated as 2’s complement signed.
Z	is set if and only if the result is zero.
C	is set to the carry out of bit 31 of the ALU.
V	is set if a signed overflow occurs into bit 31 of the result. This can be ignored if the operands are considered as unsigned, but warns of a possible error if they are being treated as 2’s complement signed.

4.5.2 Shifts

When the second operand is a shifted register, the instruction’s Shift field controls the operation of the shifter. This indicates the type of shift to be performed (Logical Left or Right, Arithmetic Right or Rotate Right).

The amount by which the register should be shifted may be contained either in an immediate field in the instruction, or in the bottom byte of another register (other than R15). The encoding for the different shift types is shown in ***Figure 4-5: ARM shift operations***.

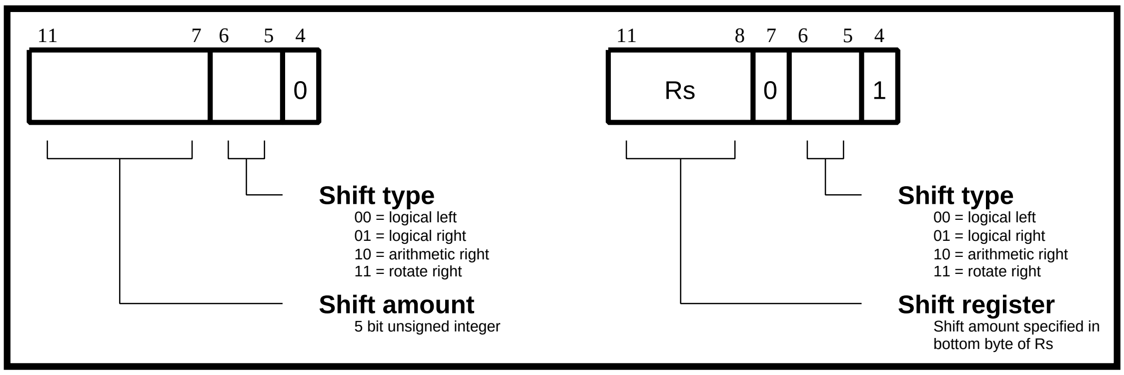


Figure 4-5: ARM shift operations

Instruction-specified shifts

When specified in the instruction, the shift amount is contained in a 5-bit field which may take any value from 0 to 31. A logical shift left (LSL) takes the contents of Rm, and moves each bit to a more significant position by the specified amount. The least significant bits of the result are filled with zeros, and the high bits of Rm that do not map into the result are discarded, with the exception of the least significant discarded bit. This becomes the shifter carry output, which may be latched into the C bit of the CPSR when the ALU operation is in the logical class (see ***Logical operations*** on page 4-8).

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

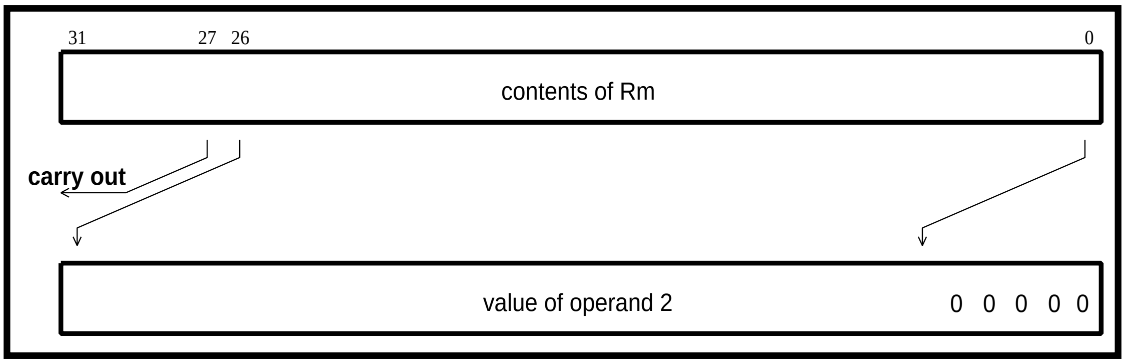


Figure 4-6: Logical shift left

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

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

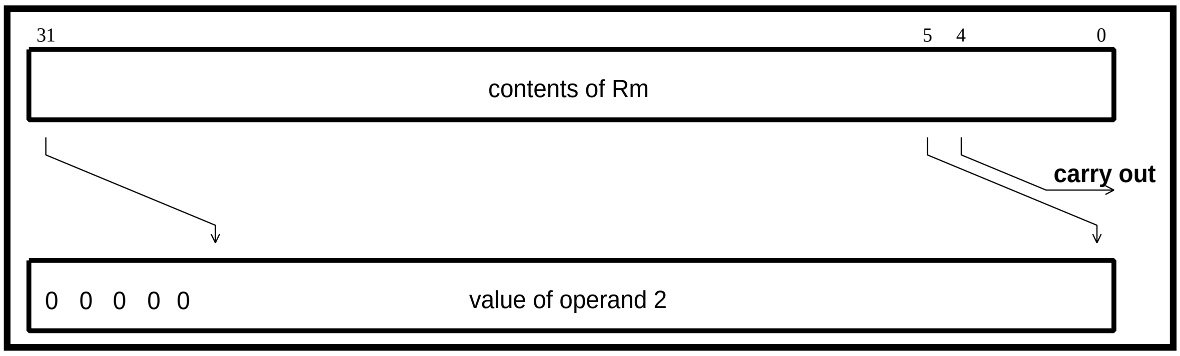


Figure 4-7: Logical shift right

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

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

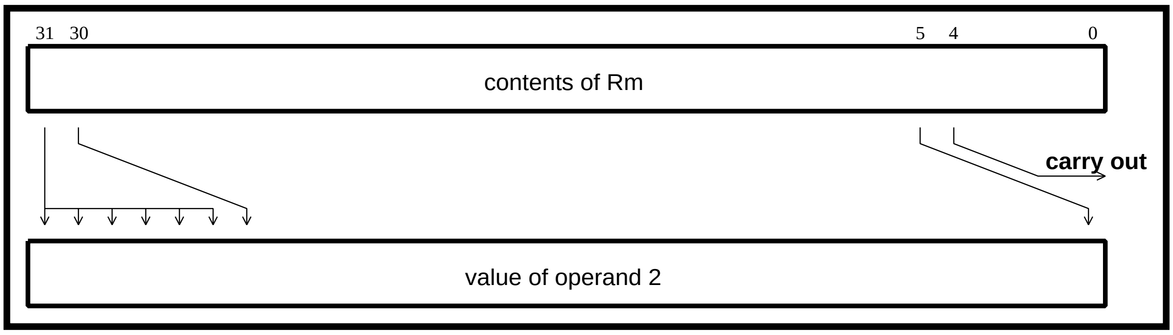


Figure 4-8: Arithmetic shift right

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

Rotate right: Rotate right (ROR) operations re-use the bits which “overshoot” in a logical shift right operation by reintroducing them at the high end of the result, in place of the zeros used to fill the high end in logical shift right operations. To illustrate this, the effect of ROR #5 is shown in ***Figure 4-9: Rotate right***.

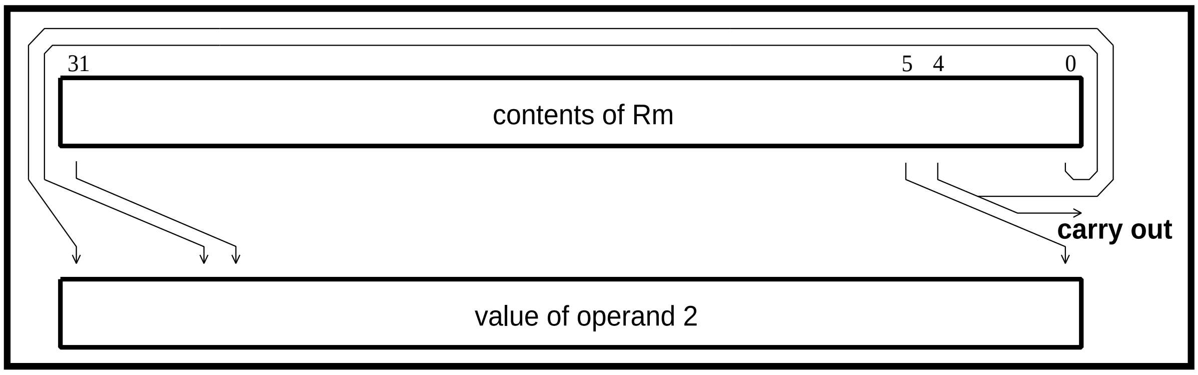


Figure 4-9: Rotate right

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

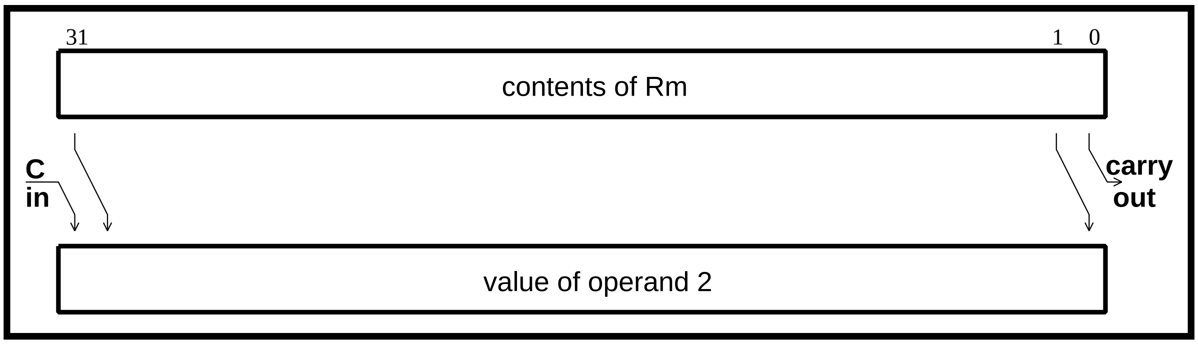


Figure 4-10: Rotate right extended

Register-specified shifts

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

Byte value	Description
0	the unchanged contents of Rm will be used as the second operand, and the old value of the CPSR C flag will be passed on as the shifter carry output
1- 31	the shifted result will exactly match that of an instruction specified shift with the same value and shift operation
32	the result will be a logical extension of the shift described above: • LSL by 32 has result zero, carry out equal to bit 0 of Rm. • LSL by more than 32 has result zero, carry out zero. • LSR by 32 has result zero, carry out equal to bit 31 of Rm. • LSR by more than 32 has result zero, carry out zero. • ASR by 32 or more has result filled with and carry out equal to bit 31 of Rm. • ROR by 32 has result equal to Rm, carry out equal to bit 31 of Rm. • ROR by *n* where *n* is greater than 32 will give the same result and carry out as ROR by *n*-32; therefore repeatedly subtract 32 from *n* until the amount is in the range 1 to 32

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

4.5.3 Immediate operand rotates

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

If the value in the 'Rotate' field is zero, the shifter carry out is set to the old value of the CPSR C flag. Otherwise, the shifter carry out is set to bit 31 of the shifter result, just as though an ROR had been performed (see ***Figure 4-9: Rotate right*** on page 4-11).

4.5.4 Writing to R15

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

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

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

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

4.5.5 Using R15 as an operand

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

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

4.5.6 MOV and MVN opcodes

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

4.5.7 TEQ, TST, CMP and CMN opcodes

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

In 32-bit modes, the TEQP form of the instruction used in earlier processors should not be used: the PSR transfer operations (MRS, MSR) must be used instead. Please refer to ***Appendix B, 26-bit Operations on ARM8*** for information on 26-bit mode operation.

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

4.5.8 Instruction cycle times

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

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

Table 4-2: Instruction cycle times

4.5.9 Assembler syntax

The data processing instructions have the following syntax:

One operand instructions

MOV, MVN

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

Instructions that do not produce a result

CMP, CMN, TEQ, TST

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

Two operand instructions

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

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

where:

`{cond}`	is a two-character condition mnemonic. The assembler assumes AL (ALways) if no condition is specified.
`{S}`	if present, specifies that the CPSR flags will be affected (implied for CMP, CMN, TEQ, TST).
`Rd`	is an expression evaluating to a valid register number.
`Rn`	is an expression evaluating to a valid register number.
`<Op2>`	is `Rm{,<shift>}` or `#<expression>`, where `<shift>` is one of:

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

`<shiftname>` can be:

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

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

If there is a choice of forms (for example as in #0, which can be represented using 0 rotated by 0, 2, 4,...30) the assembler will use a rotation by 0 wherever possible. This affects whether C will be changed in a logical operation with the S bit set - see ***4.5.3 Immediate operand rotates*** on page 4-12. If the rotation is 0, then C won't be modified. If the rotation is non-zero, it will be set to the last rotated bit as shown in ***Figure 4-9: Rotate right*** on page 4-11.

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

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

where:

`<immediate>`	is a number in the range 0-255
`<rotate>`	is an even number in the range 0-30

4.5.10 Examples

```
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

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

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

4.6 PSR Transfer (MRS, MSR)

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

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. ***Figure 4-11: MSR (transfer register contents or immediate value to PSR)*** on page 4-17 and ***Figure 4-12: MRS (transfer PSR contents to a register)*** on page 4-18 show the encodings.

These instructions allow access to the CPSR and SPSR registers.

MRS allows the contents of the CPSR or SPSR_<mode> register to be moved to a general register. MSR allows the contents of a general register or an immediate value to be moved to the CPSR or SPSR_<mode> register, with the options of affecting:

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

4.6.1 MSR operands

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

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

4.6.2 Operand restrictions

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

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

R15 cannot be specified as the source or destination register.

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

4.6.3 Reserved bits

Only eleven bits of the PSR are defined in ARM8 (N, Z, C, V, I, F and M[4:0]). The remaining bits (PSR[27:8,5]) are reserved for use in future versions of the processor.

To ensure the maximum compatibility between ARM8 programs and future processors, you should observe the following rules:

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

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

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

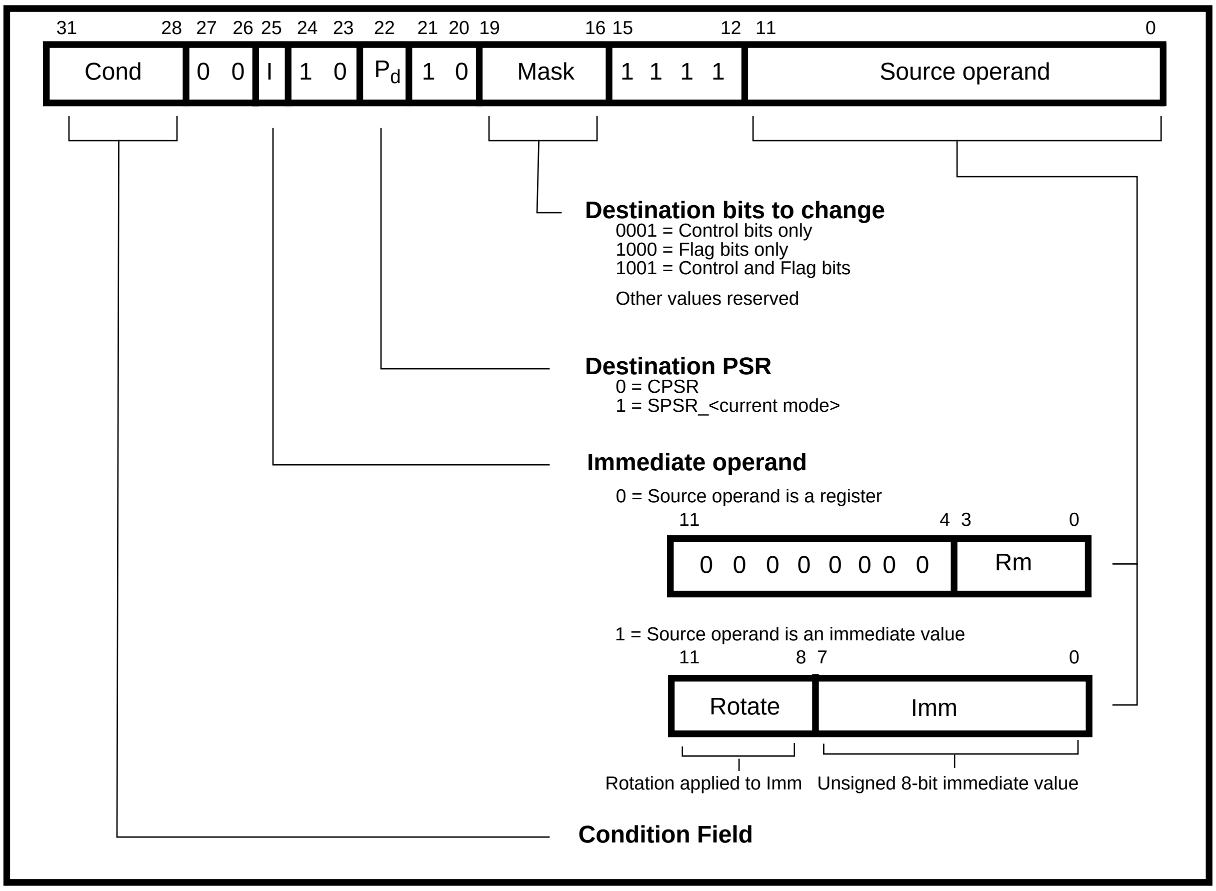


Figure 4-11: MSR (transfer register contents or immediate value to PSR)

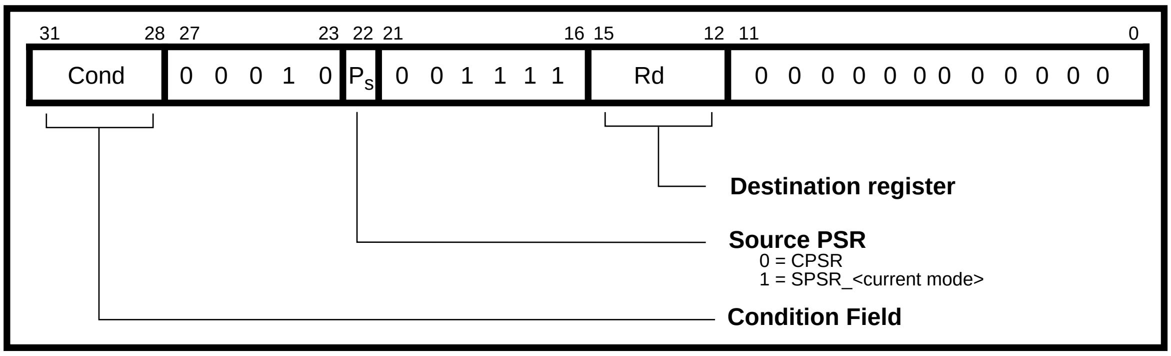


Figure 4-12: MRS (transfer PSR contents to a register)

For example, the following sequence performs a mode change:

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

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

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

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

4.6.4 Instruction cycle times

The MRS instruction takes 1 cycle.

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

4.6.5 Assembler syntax

The PSR transfer instructions have the following syntax:

Transfer PSR contents to a register

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

Transfer register contents to PSR

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

Transfer immediate value to PSR

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

where:

`{cond}`	is a two-character condition mnemonic. The assembler assumes AL (ALways) if no condition is specified.
`Rd` and `Rm`	are expressions evaluating to a register number other than R15.
`<psr>`	is CPSR or SPSR.
`<fields>`	is one of:
	_c to set the control field mask bit (bit 0)
	_x to set the extension field mask bit (bit 1)
	_s to set the status field mask bit (bit 2)
	_f to set the flags field mask bit (bit 3)
`#<expression>`	is used by the assembler to generate a shifted immediate 8-bit field. If this impossible, the assembler gives an error.

4.6.6 Previous, deprecated MSR assembler syntax

This section describes the old assembler syntax for MSR instructions. These will still work on ARM8, but should be replaced by the new syntax as described in section ***4.6.5 Assembler syntax*** on page 4-19.

Transfer register contents to PSR

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

Transfer immediate value to PSR

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

where:

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

4.6.7 Examples

User mode

In User mode, the instructions behave as follows:

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

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

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

System mode

In system mode, the instructions behave as follows:

```
MSR   CPSR_all,Rm             ; CPSR[31:0]  <- Rm[31:0]
MSR   CPSR_flg,Rm             ; CPSR[31:28] <- Rm[31:28]
MSR   CPSR_ctl,Rm             ; CPSR[7:0]   <- Rm[7:0]

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

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

Other privileged modes

In other privileged modes, the instructions behave as follows:

```
MSR   CPSR_all,Rm             ; CPSR[31:0]  <- Rm[31:0]
MSR   CPSR_flg,Rm             ; CPSR[31:28] <- Rm[31:28]
MSR   CPSR_ctl,Rm             ; CPSR[7:0]   <- Rm[7:0]

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

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

MSR   SPSR_all,Rm             ; SPSR_<mode>[31:0]  <- Rm[31:0]
MSR   SPSR_flg,Rm             ; SPSR_<mode>[31:28] <- Rm[31:28]
MSR   SPSR_ctl,Rm             ; SPSR_<mode>[7:0]   <- Rm[7:0]

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

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

4.7 Multiply and Multiply-Accumulate (MUL, MLA)

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

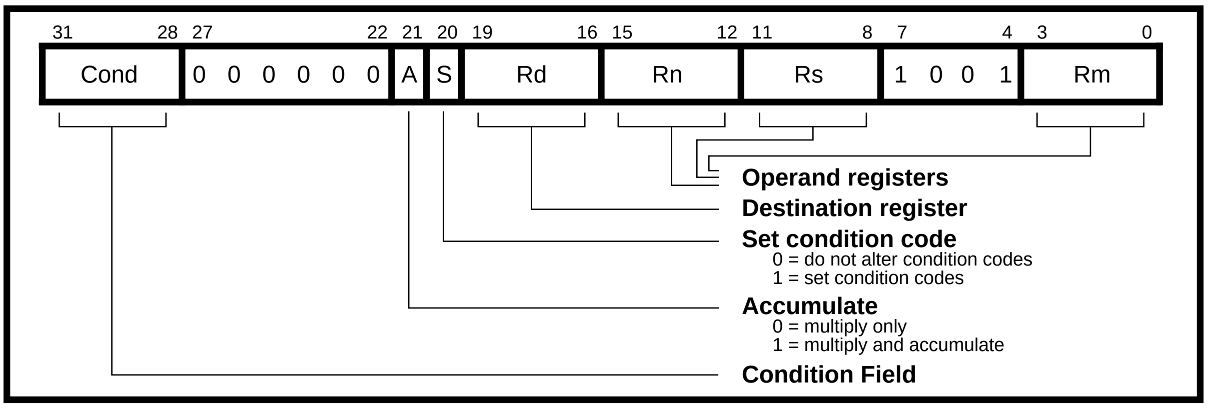


Figure 4-13: Multiply instructions

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

Multiply instruction

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

Multiply-accumulate

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

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

Operand A	Operand B	Result
0xFFFFFFF6	0x00000014	0xFFFFFF38

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

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

4.7.1 Operand restrictions

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

4.7.2 CPSR flags

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

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

4.7.3 Instruction cycle times

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

Basic cycle count	6 (including any accumulate)
Early termination	-(0 to 3) (see ***8.4 Multiply and Multiply-Accumulate*** on page 8-4 for further details)

4.7.4 Assembler syntax

The multiply instructions have the following syntax:

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

where:

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

4.7.5 Examples

```
MUL         R1,R2,R3    ; R1:=R2*R3

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

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

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

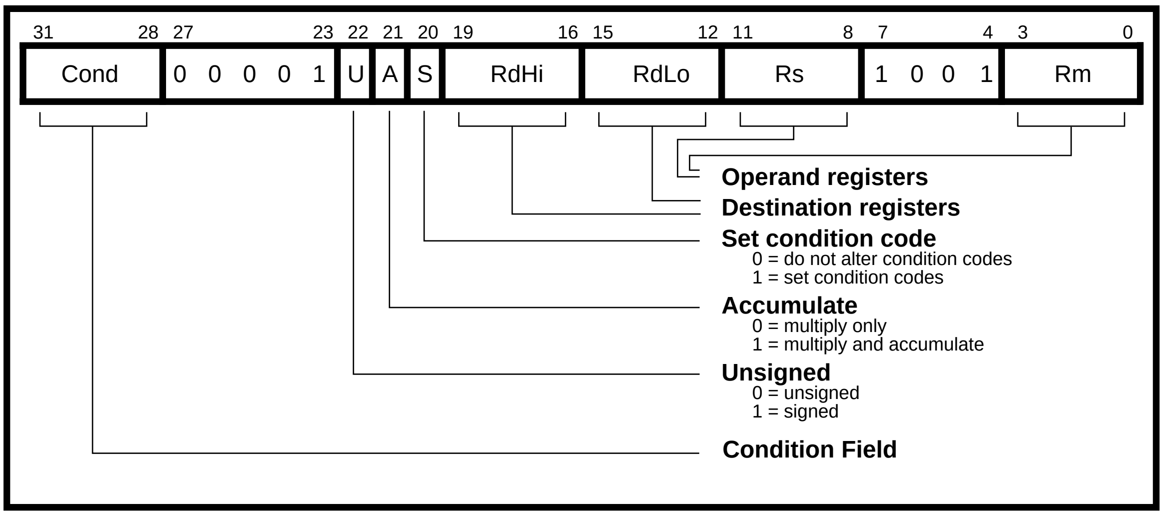


Figure 4-14: Multiply long instructions

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

Multiply (UMULL and SMULL)

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

Multiply-accumulate (UMLAL and SMLAL)

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

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

4.8.1 Operand restrictions

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

4.8.2 CPSR flags

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

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

4.8.3 Instruction cycle times

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

Basic cycle count	7 (including any accumulate)
Early termination	-(0 to 3) (see ***8.4 Multiply and Multiply-Accumulate*** on page 8-4 for further details)

4.8.4 Assembler syntax

The multiply long instructions have the following syntax:

Unsigned multiply long (32 x 32 = 64)

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

Unsigned multiply and accumulate long (32 x 32 + 64 = 64)

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

Signed multiply long (32x 32 = 64)

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

Signed multiply and accumulate long (32 x 32 + 64 = 64)

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

where

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

```
Examples

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

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

4.9 Single Data Transfer (LDR, STR)

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

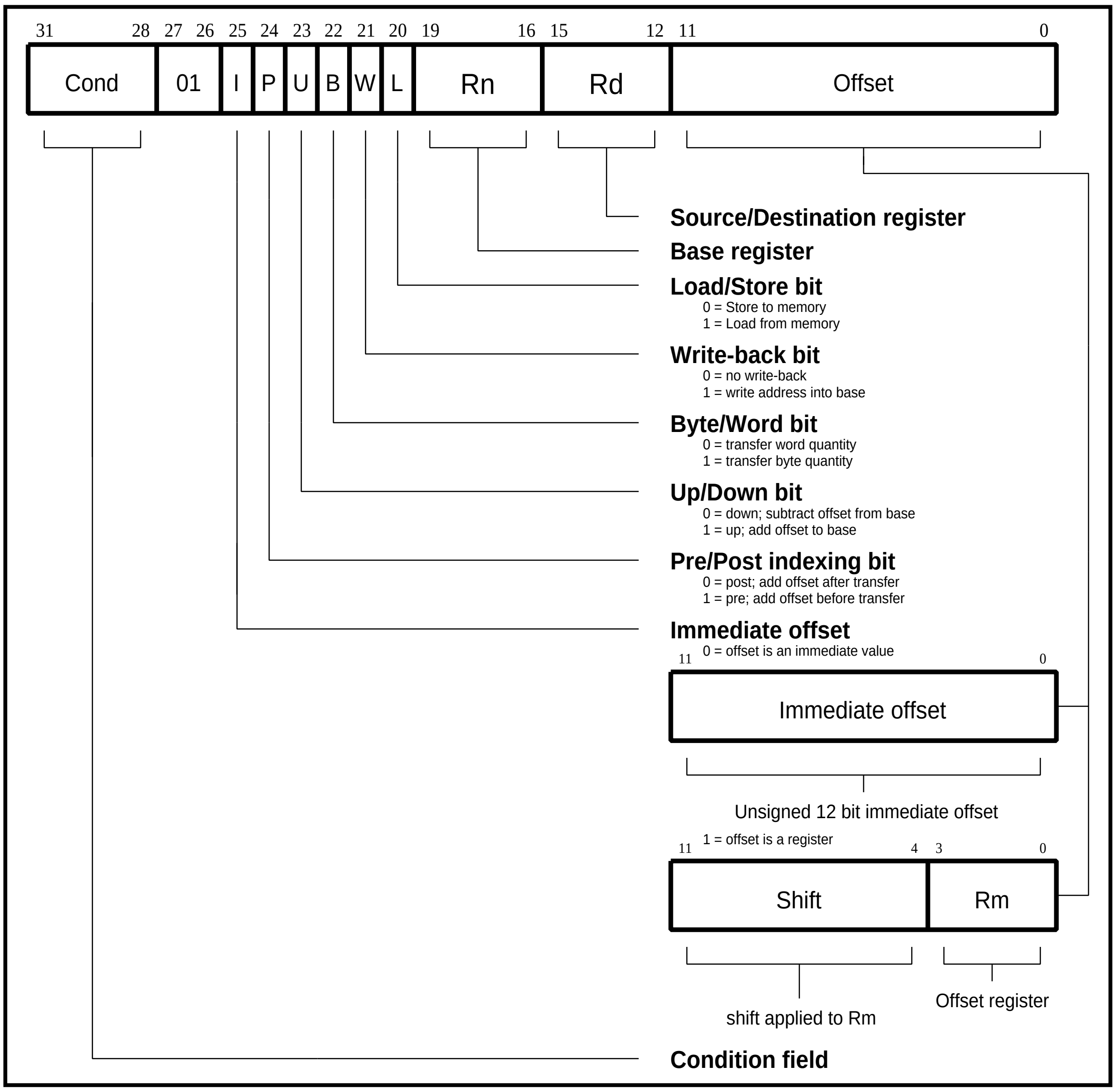


Figure 4-15: Single data transfer instructions

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

4.9.1 Offsets and auto-indexing

The offset from the base may be either a 12-bit unsigned binary immediate value in the instruction, or a second register (possibly shifted in some way).

The offset may be added to (U=1) or subtracted from (U=0) the base register Rn. The offset modification may be performed either before (pre-indexed, P=1) or after (post-indexed, P=0) the base is used as the transfer address.

The W bit gives optional auto increment and decrement addressing modes. The modified base value may be written back into the base (W=1), or the old base value may be kept (W=0).

In the case of post-indexed addressing, the write-back bit is redundant, since the old base value can be retained by setting the offset to zero. Therefore post-indexed data transfers always write back the modified base. The only use of the W bit in a post-indexed data transfer is in privileged mode code, where setting the W bit forces non-privileged mode for the transfer, allowing the operating system to generate a user address in a system where the memory management hardware makes suitable use of this facility.

4.9.2 Shifted register offset

The 8 shift control bits are described in ***4.5.2 Shifts*** on page 4-9. However, register-specified shift amounts are not available in this instruction class.

4.9.3 Bytes and words

This instruction class may be used to transfer a byte (B=1) or a word (B=0) between an ARM8 register and memory.

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

- Little-endian
- Big-endian

Little-endian configuration

Byte load (LDRB)	expects the data on data bus inputs 7 through 0 if the supplied address is on a word boundary, on data bus inputs 15 through 8 if it is a word address plus one byte, and so on. The selected byte is placed in the bottom 8 bits of the destination register, and the remaining bits of the register are filled with zeros. Please see ***Figure 3-1: Little-endian addresses of bytes within words*** on page 3-2.
Byte store (STRB)	repeats the bottom 8 bits of the source register four times across data bus outputs 31 through 0. The external memory system should activate the appropriate byte subsystem to store the data.
Word load (LDR)	Any non word-aligned address will cause the data read to be rotated into the register so that the addressed byte occupies bits 0 to 7. This means that halfwords accessed at offsets 0 and 2 from the word boundary will be correctly loaded into bits 0 through 15 of the register. Two shift operations are then required to clear or to sign extend the upper 16 bits. This is illustrated in ***Figure 4-16: Little-endian offset addressing*** on page 4-28.

Note: The LDRH and LDRSH insrtuctions provide a more efficient way to load half-words on ARM8. This method of loading half-words should therefore only be used if compatibility with previous ARM processors is required. See ***4.10 Halfword and Signed Data Transfer*** on page 4-33 for further details.

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

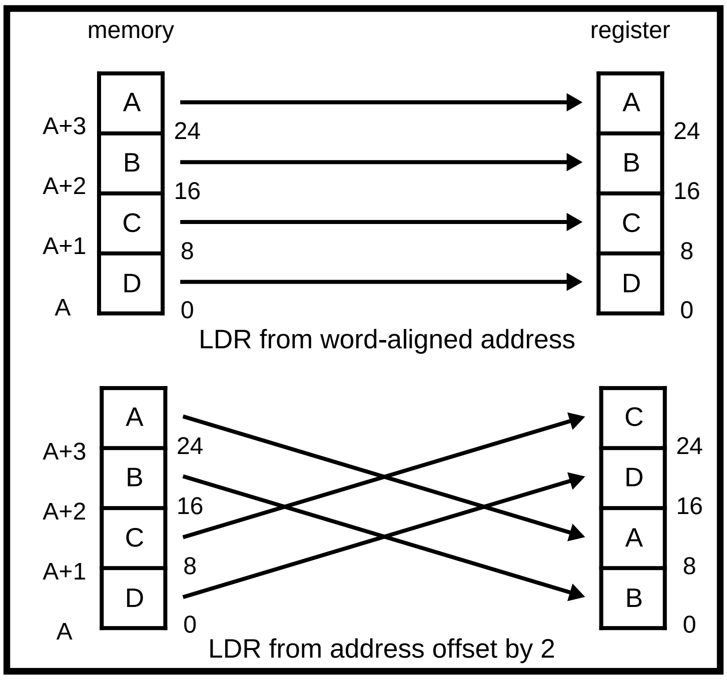


Figure 4-16: Little-endian offset addressing

Big-endian configuration

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

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

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

Note: The LDRH and LDRSH instructions provide a more efficient way to load half-words on ARM8. This method of loading half-words should therefore only be used if compatibility with previous ARM processors is required. See ***4.10 Halfword and Signed Data Transfer*** on page 4-33 for details.

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

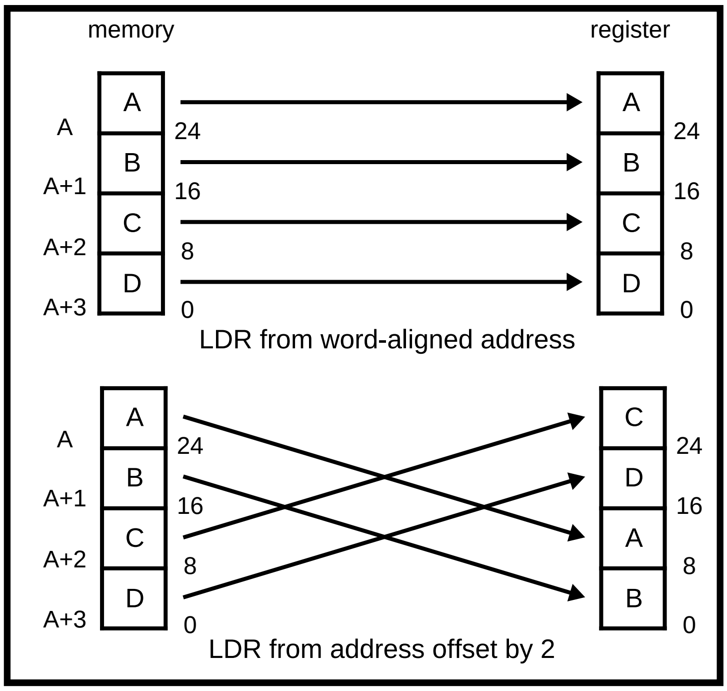


Figure 4-17: Big-endian offset addressing

4.9.4 Use of R15

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

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

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

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

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

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

4.9.5 Restrictions on the use of the base register

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

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

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

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

4.9.6 Data aborts

Please refer to ***3.4.3 Aborts*** on page 3-10 for details of aborts in general.

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

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

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

4.9.7 Instruction cycle times

LDR instructions take 1 cycle:

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

STR instructions take 1 cycle:

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

4.9.8 Assembler syntax

The single data transfer instructions have the following syntax:

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

where:

`LDR`	loads from memory into a register.
`STR`	stores from a register into memory.
`{cond}`	is a two-character condition mnemonic. If omitted, the assembler assumes ALways.
`{B}`	if present, specifies byte transfer. If omitted, word transfer is used.
`{T}`	if present, sets the W bit in a post-indexed instruction, forcing non-privileged mode for the transfer cycle. T is not allowed when a pre-indexed addressing mode is specified or implied.
`Rd`	is an expression evaluating to a valid register number.
`<Addr>`	is one of:

An `<expression>` specifying an address:

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

A pre-indexed addressing specification:

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

A post-indexed addressing specification:

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

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

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

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

4.9.9 Examples

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

STR    R1,[R2],R4      ; store R1 at R2. Write back R2+R4 to R2

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

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

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

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

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

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in ***Figure 4-18: Halfword and signed data transfer with register offset*** and ***Figure 4-19: Halfword and signed data transfer with immediate offset***.

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

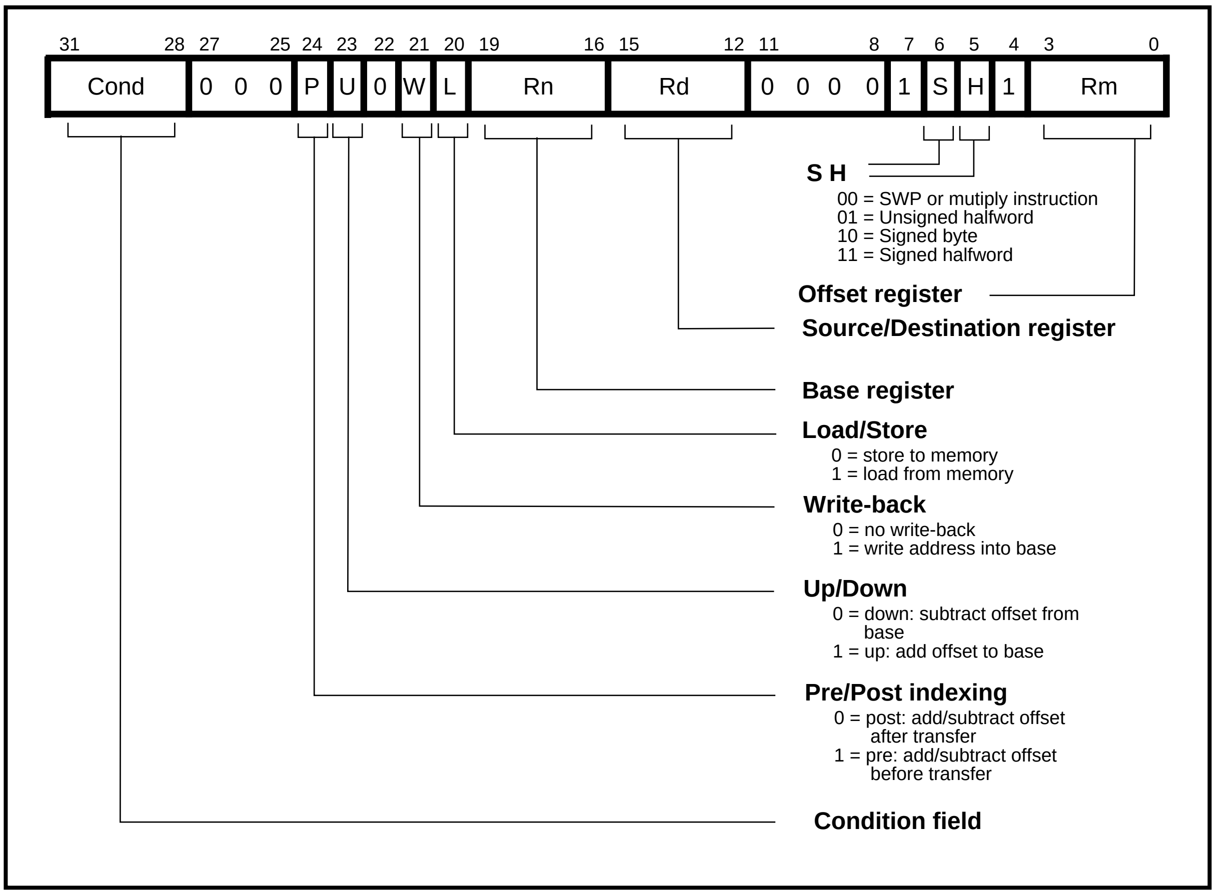


Figure 4-18: Halfword and signed data transfer with register offset

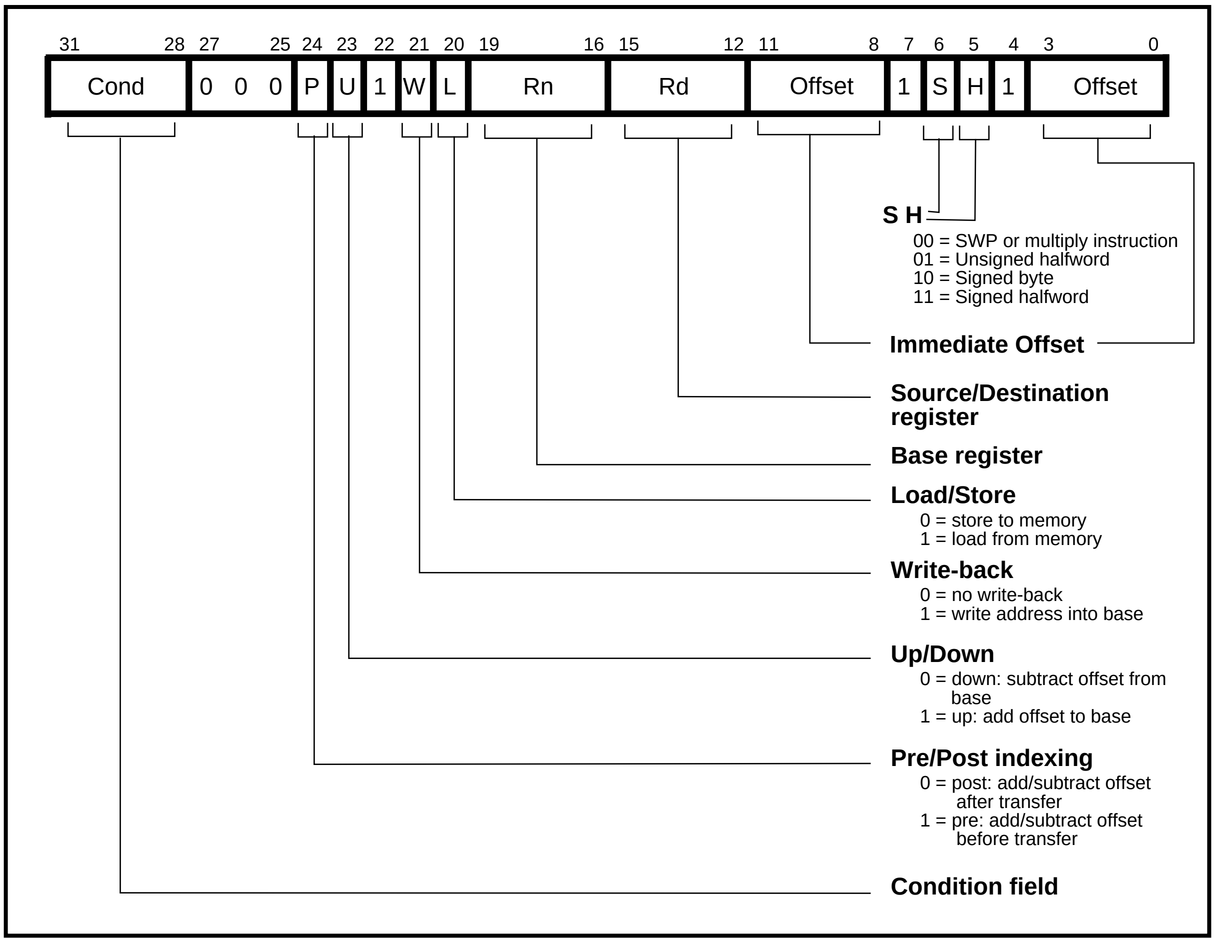


Figure 4-19: Halfword and signed data transfer with immediate offset

4.10.1 Offsets and auto-indexing

The offset from the base may be either an 8-bit unsigned binary immediate value in the instruction, or a second register. In the case of an immediate value, bits 11:8 (xxxx) and bits 3:0 (yyyy) combine to form the offset (xxxxyyyy). The offset may be added to (U=1) or subtracted from (U=0) the base register Rn. The offset modification may be performed either before (pre-indexed, P=1) or after (post-indexed, P=0) the base register is used as the transfer address.

The W bit gives optional auto-increment and decrement addressing modes. The modified base value may be written back into the base (W=1), or the old base may be kept (W=0). In the case of post-indexed addressing, the write back bit is redundant and is always set to zero, since the old base value can be retained if necessary by setting the offset to zero. Therefore post-indexed data transfers always write back the modified base.

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

4.10.2 Halfword load and stores

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

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

4.10.3 Signed byte and halfword loads

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

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

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

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

4.10.4 Endianness and byte/halfword selection

Little-endian configuration

Signed byte load (LDRSB): This load expects data on data bus inputs 7 through to 0 if the supplied address is on a word boundary, on data bus inputs 15 through to 8 if it is a word address plus one byte, and so on. The selected byte is placed in the bottom 8 bits of the destination register, and the remaining bits of the register are filled with the sign bit, the most significant bit of the byte. Please see ***Figure 3-1: Little-endian addresses of bytes within words*** on page 3-2.

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

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

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

Big-endian configuration

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

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

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

4.10.5 Use of R15

Do not specify R15 as:

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

Base register

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

4.10.6 Restrictions on the use of the base register

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

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

4.10.7 Data aborts

Please refer to ***3.4.3 Aborts*** on page 3-10 for details of aborts in general.

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

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

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

4.10.8 Instruction cycle times

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

Load instructions take 1 cycle.

Store instructions take 1 cycle.

4.10.9 Assembler syntax

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

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

1 An expression which generates an address:

```
<expression>
```

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

2 A pre-indexed addressing specification:

`[Rn]`	offset of zero
`[Rn,<#expression>]{!}`	offset of `<expression>` bytes
`[Rn,{+/-}Rm]{!}`	offset of +/- contents of index register

3 A post-indexed addressing specification:

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

Rn and `Rm`	are expressions evaluating to a register number. If Rn is R15, neither post-indexed addressing nor `{!}` should be specified.
`{!}`	writes back the base register (sets the W bit) if ! is present.

4.10.10 Examples

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

4.11 Block Data Transfer (LDM, STM)

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

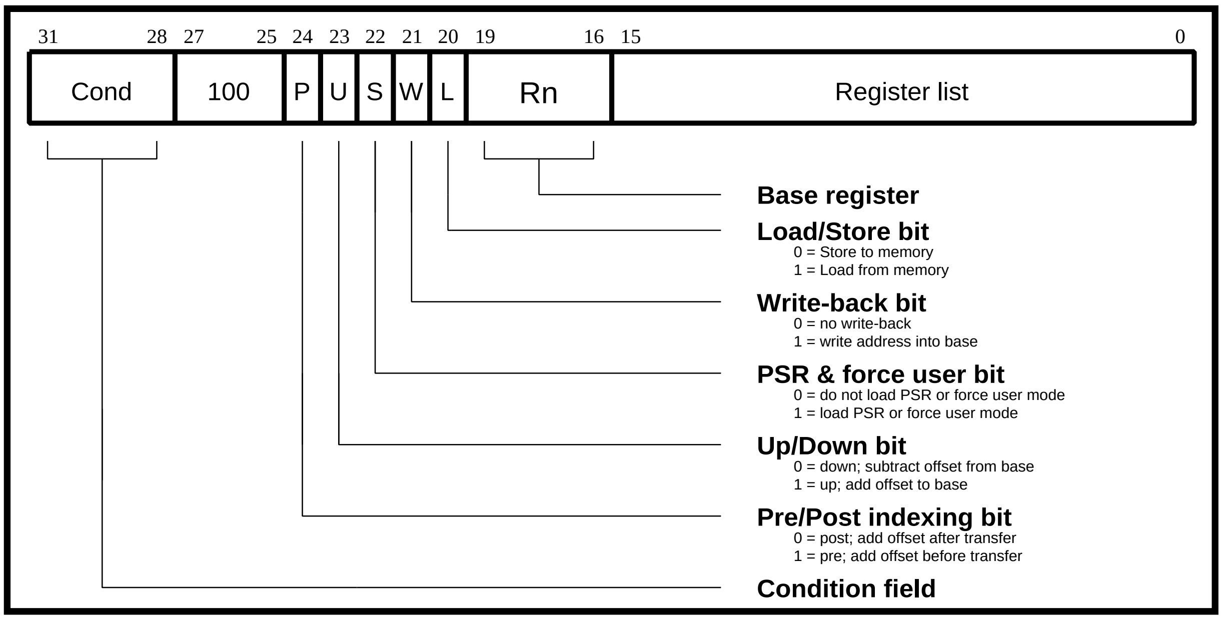

31
28
27
25
24
23
22
21
20
19
16
15
0
Cond
100
P
U
S
W
L
Rn
Register list
Base register
Load/Store bit
0 = Store to memory
1 = Load from memory
Write-back bit
0 = no write-back
1 = write address into base
PSR & force user bit
0 = do not load PSR or force user mode
1 = load PSR or force user mode
Up/Down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

Figure 4-20: Block data transfer instructions

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

4.11.1 The register list

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

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

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

4.11.2 Addressing modes

The transfer addresses are determined by the contents of the base register (Rn), the pre/post bit (P) and the up/down bit (U). The registers are stored such that the lowest register is always at the lowermost address in memory, the highest numbered register is always at the uppermost address, and the others are stored in numerical order between them.

The register transfers will occur in ascending order. By way of illustration, consider the transfer of R1, R5 and R7 in the case where Rn=0x1000 and write-back of the modified base is required (W=1). Figures ***Figure 4-21: Post-increment addressing*** to ***Figure 4-24: Pre-decrement addressing*** starting on page 4- 41 show the sequence of register transfers, the addresses used, and the value of Rn after the instruction has completed.

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

4.11.3 Address alignment

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

4.11.4 Use of the S bit

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

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

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

STM with S bit set (User bank transfer)

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

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

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

4.11.5 Use of R15

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

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

4.11.6 Inclusion of the base in the register list

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

4.11.7 Data aborts

Please refer to ***3.4.3 Aborts*** on page 3-10 for details of Aborts in general.

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

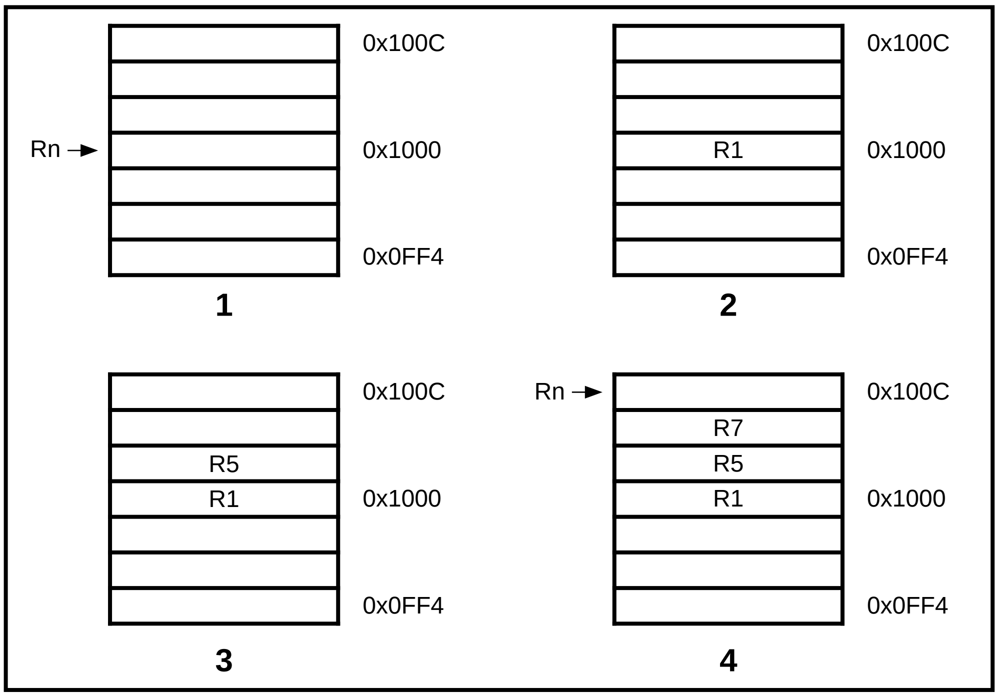


Figure 4-21: Post-increment addressing

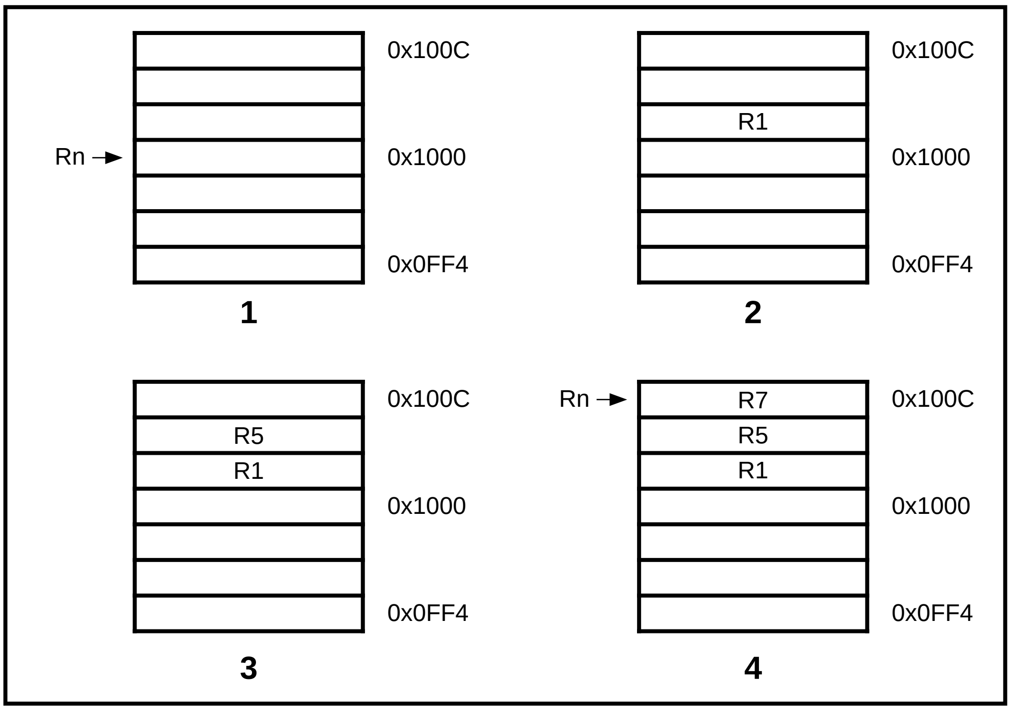


Figure 4-22: Pre-increment addressing

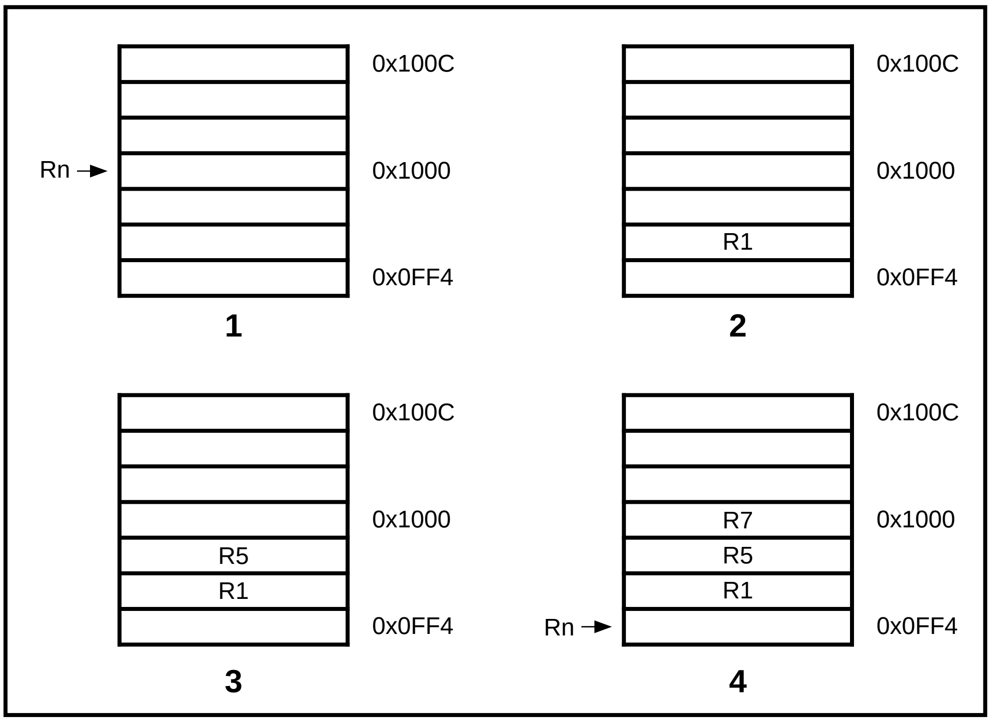


Figure 4-23: Post-decrement addressing

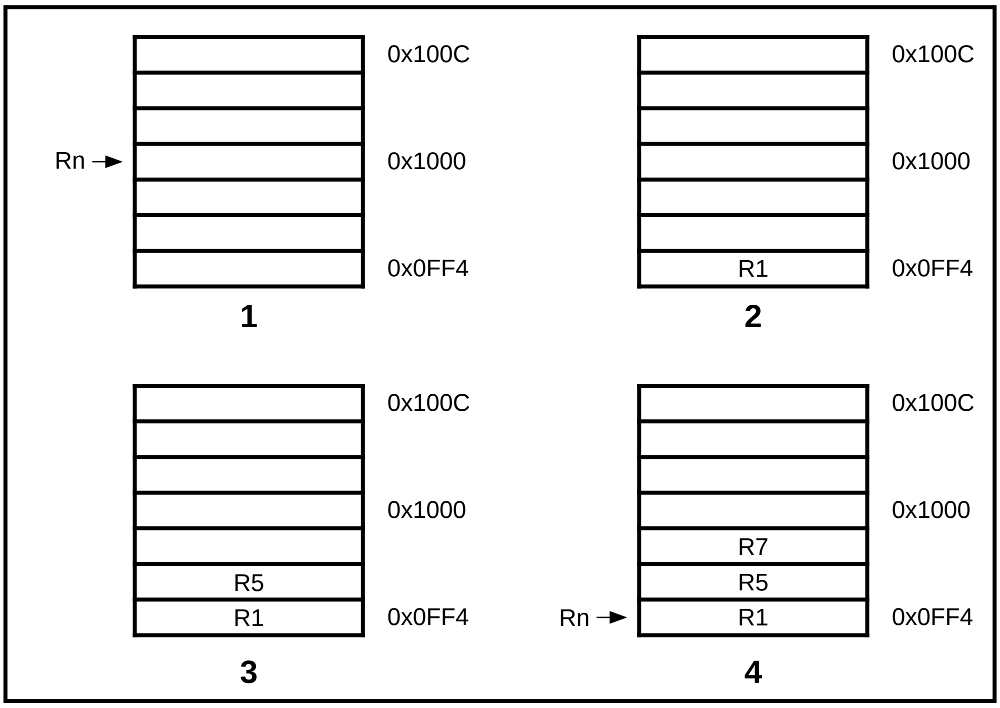


Figure 4-24: Pre-decrement addressing

4.11.8 Instruction cycle times

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

The following table shows the basic cycle count for LDM.

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

Table 4-3: Basic cycle count for LDM

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

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

Additional cycles may be incurred if the memory system indicates that it is only able to transfer one item of data where two were requested. For example, when accessing the last word in a cache line in a cached memory system. The **RResponse[]** control indicates this. See ***Chapter 6, Memory Interface*** for details.

The following table shows the cycle counts for STM instructions.

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

Table 4-4: Basic cycle count for STM

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

4.11.9 Assembler syntax

The block data transfer instructions have the following syntax:

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

where:

`LDM`	loads from memory to registers.
`STM`	stores from registers to memory.
`{cond}`	is a two-character condition mnemonic. The assembler assumes AL (ALways) if no condition is specified.
`<addressmode>`	is one of `<FD|ED|FA|EA|IA|IB|DA|DB>`. Note that `<addressmode>` is *not* optional. (See ***Table 4-5: Addressing mode names*** on page 4-46)
`Rn`	is an expression evaluating to a register number.
`<Rlist>`	is a list of registers and register ranges enclosed in {} (eg.

	{R0,R2-R7,R10}).
`{!}`	if present, requests write-back (W=1), otherwise W=0.
`{^}`	if present, sets the S bit. See ***4.11.4 Use of the S bit*** on page 4-40.

Addressing mode names

There are different assembler mnemonics for each of the addressing modes, depending on whether the instruction is being used to support stacks or for other purposes. These are shown in ***Table 4-5: Addressing mode names*** on page 4-46.

Key to table:

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

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

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

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

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

Table 4-5: Addressing mode names

4.11.10 Examples

```
LDMFD SP!,{R0,R1,R2} ; unstack 3 registers
STMIA R0,{R0-R15}    ; save all registers

LDMFD SP!,{R15}      ; unstack R15,CPSR unchanged

LDMFD SP!,{R15}^     ; unstack R15, 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
```

4.12 Single Data Swap (SWP)

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

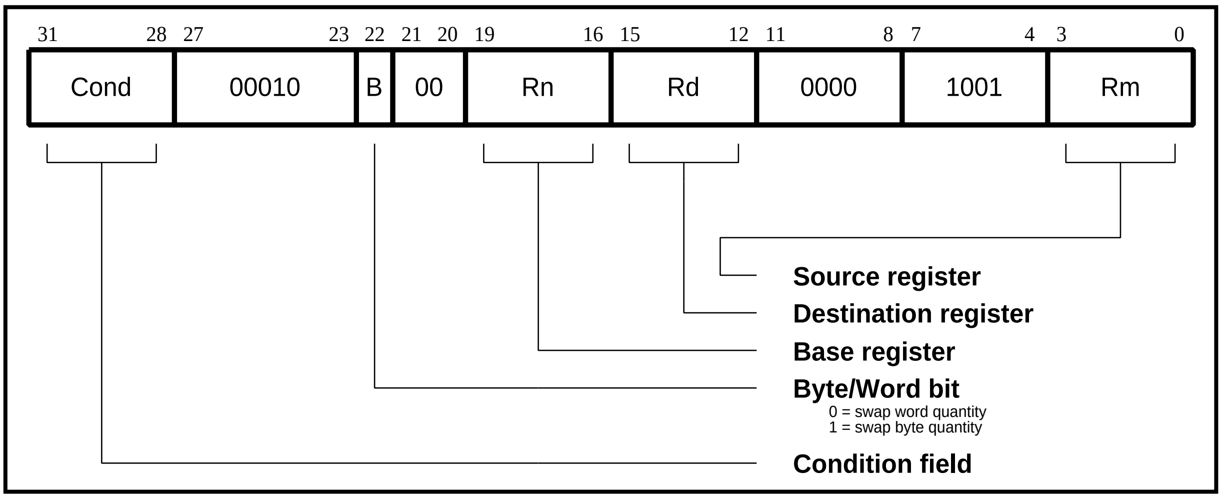


Figure 4-25: Swap instruction

The data swap instruction is used to swap a byte or word quantity between a register and external memory. It is implemented as a memory read followed by a memory write which are “locked” together. The processor cannot be interrupted until both operations have completed, and the memory manager is warned to treat them as inseparable.

This class of instruction is particularly useful for implementing software semaphores.

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

ARequest[] takes special values for 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. See ***Chapter 6, Memory Interface*** for futher details. This is important in multi-processor systems, where the swap instruction is the only indivisible instruction which may be used to implement semaphores. Do not remove control of the memory from a processor while it is performing a sequence of locked operations.

4.12.1 Bytes and words

This instruction class may be used to swap a byte (B=1) or a word (B=0) between an ARM8 register and memory. The SWP instruction is implemented as a LDR followed by a STR and the action of these is as described in ***4.9 Single Data Transfer (LDR, STR)*** on page 4-26. In particular, the description of big- and little-endian configuration applies to the SWP instruction. Note that there is no halfword SWP.

4.12.2 Use of R15

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

4.12.3 Data aborts

Please refer to ***3.4.3 Aborts*** on page 3-10 for details of Aborts in general.

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

This can happen on either the read or the write cycle. In either case, the Data Abort trap will be taken, and neither Rd nor the contents of the memory location will have been altered. It is up to the system software to resolve the cause of the problem. Once this has been done, the instruction can be restarted and the original program continued.

4.12.4 Instruction cycle times

SWP instructions take 2 cycles.

4.12.5 Assembler syntax

The SWP instruction has the following syntax:

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

where:

`{cond}`	is a two-character condition mnemonic. The assembler assumes AL (ALways) if no condition is specified.
`{B}`	specifies byte transfer. If omitted, word transfer is used.
`Rd,Rm,Rn`	are expressions evaluating to valid register numbers.

4.12.6 Examples

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

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
```

4.13 Software Interrupt (SWI)

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

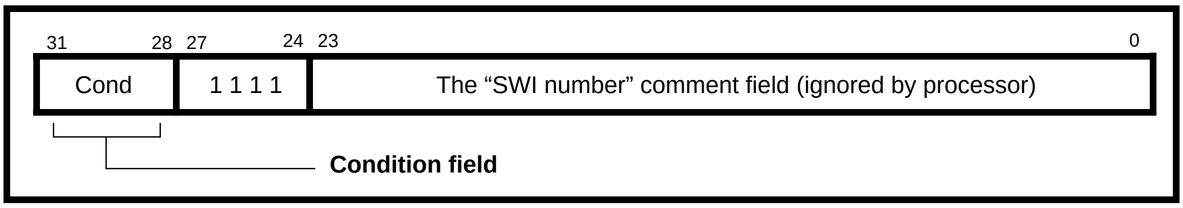


Figure 4-26: Software interrupt instruction

The software interrupt is used to enter Supervisor mode in a controlled manner. It causes the software interrupt trap to be taken, which effects the mode change. The PC is then forced to the SWI vector and the CPSR is saved in SPSR_svc. See ***3.4.4 Software interrupt*** on page 3-12 for more details.

If the SWI vector address is suitably protected (by external memory management hardware) from modification by the user, a fully protected operating system may be constructed.

4.13.1 Return from the supervisor

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

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

4.13.2 Comment field

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

4.13.3 Architecturally-defined SWIs

The ARM Architecture V4 reserves SWI numbers 0xF00000 to 0xFFFFFF inclusive for current and future Architecturally Defined SWI functions. These SWI numbers should not be used for functions other than those defined by ARM. Please see ***4.17 The Instruction Memory Barrier (IMB) Instruction*** on page 4-62 for examples of two such definitions.

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

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

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

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

4.13.4 Instruction cycle times

SWI instructions take 4 cycles to execute.

4.13.5 Assembler syntax

The SWI instruction has the following syntax:

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

where:

`{cond}`	is a two-character condition mnemonic. The assembler assumes AL (ALways) if no condition is specified.
`<expression>`	is evaluated and placed in the comment field (which is ignored by ARM8).

4.13.6 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 at the SWI vector address, for instance:

```
      B Supervisor      ; SWI entry point
      .
      .
EntryTable              ; addresses of supervisor routines
      DCD     ZeroRtn
      DCD     ReadCRtn
      DCD     WriteIRtn
      .
      .
      Zero    EQU  0
      ReadC   EQU  256
      WriteI  EQU  512

Supervisor

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

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

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

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

```
      ADR   R2,EntryTable
```

is equivalent to

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

4.14 Coprocessor Data Operations (CDP)

ARM8 expects the coprocessor interface to bounce all CDP instructions so that the undefined instruction trap will be taken. This may be used to emulate the coprocessor instruction. If the coprocessor does not bounce CDP, unpredictable behaviour will result.

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in ***Figure 4-27: Coprocessor data operation instruction***.

This class of instruction is used to tell a coprocessor to perform some internal operation. No result is communicated back to the ARM8, and it may not wait for the operation to complete. The coprocessor could contain a queue of such instructions awaiting execution, and their execution can overlap other activity, allowing the coprocessor and the ARM8 to perform independent tasks in parallel.

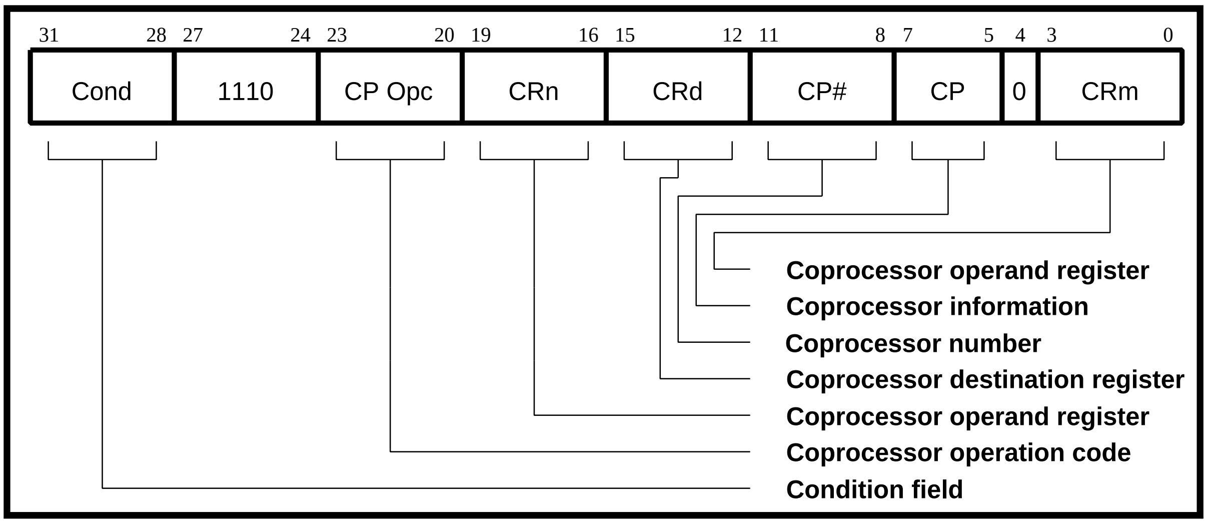


Figure 4-27: Coprocessor data operation instruction

4.14.1 The coprocessor fields

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

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

4.14.2 Instruction cycle times

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

4.14.3 Assembler syntax

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

where:

`{cond}`	two character condition mnemonic, see ***Figure 4-2: Condition Codes*** on page 4-3
`p#`	the unique number of the required coprocessor
`<expression1>`	evaluated to a constant and placed in the CP Opc field
`cd, cn` and `cm`	evaluate to the valid coprocessor register numbers CRd, CRn and CRm respectively
`<expression2>`	where present is evaluated to a constant and placed in the CP field

4.14.4 Examples

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

4.15 Coprocessor Data Transfers (LDC, STC)

ARM8 expects the coprocessor interface to bounce all LDC and STC instructions so that the undefined instruction trap is taken. This may be used to emulate the coprocessor instructions. If the coprocessor does not bounce these instructions, unpredictable behaviour will result.

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in ***Figure 4-29: Coprocessor register transfer instructions***.

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

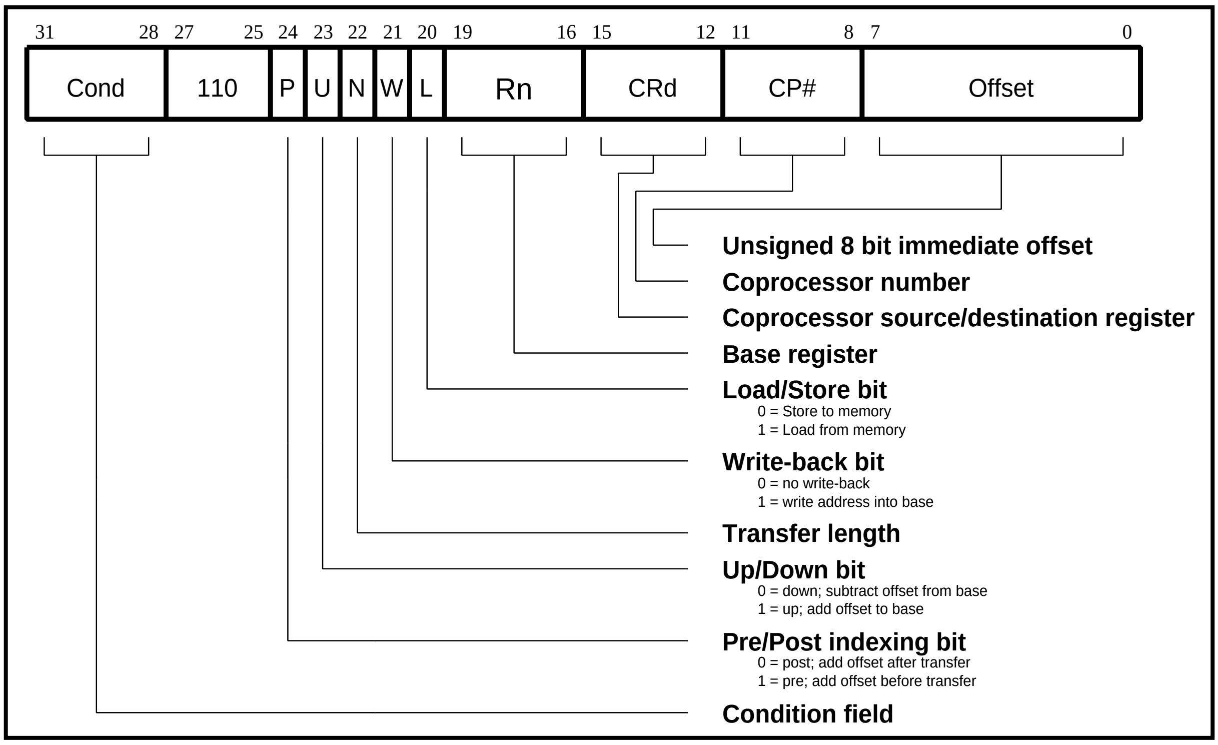


Figure 4-28: Coprocessor data transfer instructions

4.15.1 The coprocessor fields

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

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

4.15.2 Addressing modes

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

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

The value of the base register, modified by the offset in a pre-indexed instruction, is used as the address for the transfer of the first word. The second word (if more than one is transferred) will go to or come from an address one word (4 bytes) higher than the first transfer, and the address will be incremented by one word for each subsequent transfer. Instructions where P=0 and W=0 are reserved, and must not be used.

4.15.3 Address alignment

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

4.15.4 Use of R15

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

4.15.5 Data aborts

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

4.15.6 Instruction cycle times

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

4.15.7 Assembler syntax

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

where:

`LDC`	load from memory to coprocessor
`STC`	store from coprocessor to memory
`{L}`	when present, perform long transfer (N=1), otherwise perform short transfer (N=0)
`{cond}`	two character condition mnemonic. See ***Figure 4-2: Condition Codes*** on page 4-3.
`p#`	the unique number of the required coprocessor
`cd`	expression evaluating to a valid coprocessor register number that is placed in the CRd field
`<Addr>`	can be:

1. An expression which generates an address:

   `<expression>`

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

2. A pre-indexed addressing specification:

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

3. A post-indexed addressing specification:

`[Rn],<#expression>`	offset of `<expression>` bytes
`{!}`	write back the base register (set the W bit) if `!` is present
Rn	expression evaluating to a valid ARM8 register number

4.15.8 Examples

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

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

4.16 Coprocessor Register Transfers (MRC, MCR)

Please refer to ***Chapter 4, Instruction Set*** for details of the coprocessor interface and timing.

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in ***Figure 4-29: Coprocessor register transfer instructions***.

This class of instruction is used to communicate information directly between ARM8 and a coprocessor. An example of a coprocessor to processor register transfer (MRC) instruction would be a FIX of a floating point value held in a coprocessor, where the floating point number is converted into a 32-bit integer within the coprocessor, and the result is then transferred to a processor register. A FLOAT of a 32-bit value in a processor register into a floating point value within the coprocessor illustrates the use of a processor register to coprocessor transfer (MCR).

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

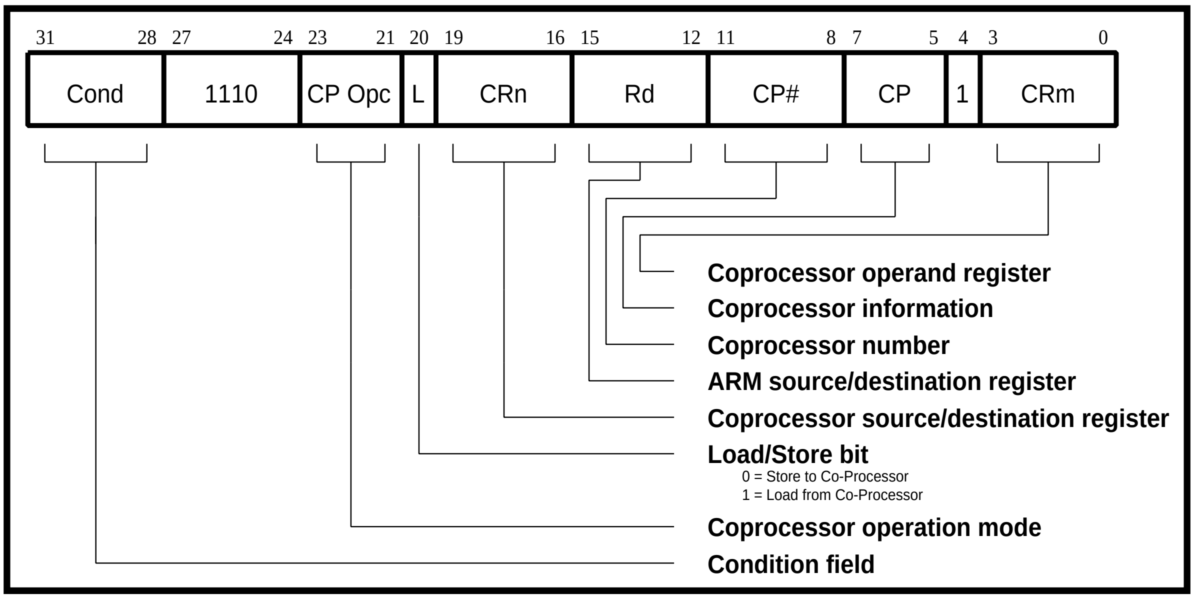

31 28 27 24 23 21 20 19 16 15 12 11 8 7 5 4 3 0
Cond
1110
CP Opc
L
CRn
Rd
CP#
CP
1
CRm
Coprocessor operand register
Coprocessor information
Coprocessor number
ARM source/destination register
Coprocessor source/destination register
Load/Store bit
0 = Store to Co-Processor
1 = Load from Co-Processor
Coprocessor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

Figure 4-29: Coprocessor register transfer instructions

4.16.1 The coprocessor fields

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

4.16.2 Transfers from R15

Do not specify a coprocessor register transfer from ARM8 with R15 as the source register.

4.16.3 Transfers to R15

When a coprocessor register transfer to ARM8 has R15 as the destination, bits 31, 30, 29 and 28 of the transferred word are copied into the N, Z, C and V flags respectively. The other bits of the transferred word are ignored, and the PC and other CPSR bits are unaffected by the transfer.

4.16.4 Instruction cycle times

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

4.16.5 Assembler syntax

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

where:

`MRC`	move from coprocessor to ARM8 register (L=1)
`MCR`	move from ARM8 register to coprocessor (L=0)
`{cond}`	two-character condition mnemonic, see ***Figure 4-2: Condition Codes*** on page 4-3
`p#`	the unique number of the required coprocessor
`<expression1>`	evaluated to a constant and placed in the CP Opc field
`Rd`	is an expression evaluating to a valid ARM8 register number
`cn` and `cm`	are expressions evaluating to the valid coprocessor register numbers CRn and CRm respectively
`<expression2>`	where present is evaluated to a constant and placed in the CP field

4.16.6 Examples

```
MRC    p2,5,R3,c5,c6   ; request coproc 2 to perform operation 5
                       ; on c5 and c6, and transfer the (single
                       ; 32-bit word) result back to R3

MCR    p6,0,R4,c6,c7   ; request coproc 6 to perform operation 0
                       ; on R4 and place the result in c6, in a
                       ; way that may be influenced by c7

MRCEQ  p3,9,R3,c5,c6,2 ; conditionally request coproc 3 to
                       ; perform operation 9 (type 2) on c5 and
                       ; c6, and transfer the result back to R3
```

4.17 The Instruction Memory Barrier (IMB) Instruction

An Instruction Memory Barrier (IMB) Instruction is used to ensure that correct instruction flow occurs after instruction memory locations are altered in any way - by self-modifying code for example. The recommended implementation of the IMB instructions is via an architecturally defined SWI function (see ***4.13 Software Interrupt (SWI)*** on page 4-50). The instruction encoding for the recommended IMB instruction implementations is shown below:

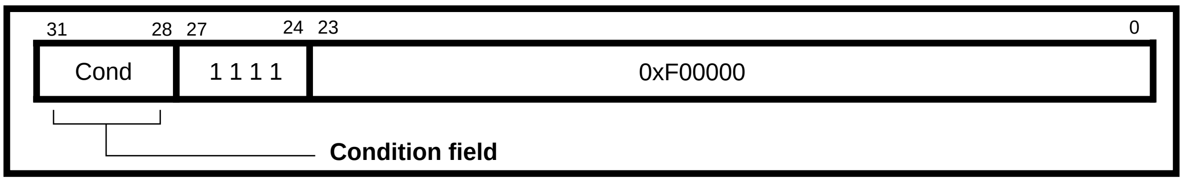


Figure 4-30: IMB instruction

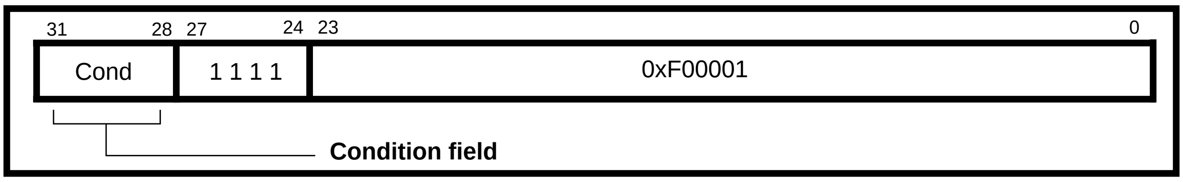


Figure 4-31: IMBRange instruction

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

4.17.1 Use

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

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

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

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

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

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

4.17.2 Assember syntax

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

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

4.17.3 Examples

Loading code from disk

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

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

Running BitBlt code

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

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

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

Self-decompressing code

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

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

4.18 Undefined Instructions

This section shows the instruction bit patterns that will cause the Undefined Instruction trap to be taken if ARM8 attempts to execute them. This vector location is defined in ***3.4.6 Exception Vector Summary*** on page 3-13. There are a number of such bit pattern classes, and these can be used to cause unimplemented instructions (for example LDC) to be emulated through the Undefined Instruction trap service routine code:

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

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

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

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

The Undefined Instruction trap is taken:

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

or

- by all coprocessor instructions whose condition is met and which are bounced by any coprocessor. For ARM8, the coprocessor interface must bounce all CDP, LDC and STC instructions

4.18.1 Assembler syntax

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

4.19 Instruction Set Examples

The following examples show ways in which the basic ARM8 instructions can combine to give efficient code. None of these methods saves a great deal of execution time (although they may save some): mostly they just save code.

4.19.1 Using the conditional instructions

Using conditionals for logical OR

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

can be replaced by :

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

Absolute value

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

Multiplication by 4, 5 or 6 (run time)

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

Combining discrete and range tests

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

Division and remainder

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

A short general purpose divide routine follows.

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

4.19.2 Overflow detection in the ARM8

Overflow in unsigned multiply with a 32-bit result

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

Overflow in signed multiply with a 32-bit result

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

Overflow in unsigned multiply accumulate with a 32-bit result

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

Overflow in signed multiply accumulate with a 32-bit result

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

Overflow in unsigned multiply accumulate with a 64-bit result

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

Overflow in signed multiply accumulate with a 64-bit result

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

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

4.19.3 Pseudo random binary sequence generator

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

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

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

4.19.4 Multiplication by constant using shifts

1 Multiplication by 2^n (1,2,4,8,16,32..)
```
MOV      Ra, Rb, LSL #n
```

2 Multiplication by 2^n+1 (3,5,9,17..)
```
ADD      Ra,Ra,Ra,LSL #n
```

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

4 Multiplication by 6
```
ADD      Ra,Ra,Ra,LSL #1; multiply by 3
MOV      Ra,Ra,LSL#1    ; and then by 2
```

5 Multiply by 10 and add in extra number
```
ADD      Ra,Ra,Ra,LSL#2 ; multiply by 5
ADD      Ra,Rc,Ra,LSL#1 ; multiply by 2 and add in next
                        ; digit
```

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

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

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

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

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

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

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

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

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

rather than by:

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

4.19.5 Loading a word from an unknown alignment

```
                    ; enter with address in Ra (32 bits)
                    ; uses Rb, Rc; result in Rd.
                    ; Note d must be less than c e.g. 0,1
                    ;
BIC   Rb,Ra,#3      ; get word-aligned address
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) for little-endian
                    ; operations (see note below)
RSBNE Rb,Rb,#32     ; get other shift amount
ORRNE Rd,Rd,Rc,LSL Rb; combine two halves to get result
                    ; for little-endian operation (see note
                    ; below)
```

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

5 The Prefetch Unit

This chapter describes the functions of the prefetch unit.

The Prefetch Unit

5.1 Overview

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

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

5.2 The Prefetch Buffer

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

5.3 Branch Prediction

ARM8 employs static branch prediction. This is based solely on the characteristics of a Branch instruction, and uses no history information. Branch prediction is performed only when the **PredictOn** external input signal is HIGH.

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

By adding a Prefetch Buffer, it is possible to detect a Branch *before* it enters the Core. This allows the use of a different prediction scheme - for instance, one which predicts that all conditional *forward* Branches are not taken and all conditional *backward* Branches are taken. This scheme is the one implemented in ARM8 and, because it models actual conditional branch behaviour more accurately, it reduces the average branch CPI, thus improving the processor’s performance.

Using ARM8’s Prefetch Unit, around 65% of all Branches are preceded by enough non-Branch cycles to be completely predicted. The Core itself must deal with the Branches that the Prefetch Unit does not have time to predict. See ***8.1 Branch and Branch with Link (B, BL)*** for the effect of mispredictions on the instruction cycle counts.

5.3.1 Incorrect predictions and correction

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

The Prefetch Unit uses the Core’s condition codes to establish the accuracy of a prediction. If the prediction is found to be in error, the Prefetch Unit begins fetching from the saved alternate address, and cancels any instructions that have been incorrectly passed to the core. See ***8.1 Branch and Branch with Link (B, BL)*** on page 8-2 for the effect of mis-predictions on the instruction cycle counts.

5.3.2 Prediction details

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

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

The following summarises the prediction scheme:

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

Instructions will not be predicted if:

- Instruction[27:24]==“1011” AND Instruction[31:28]!=“1110” (Conditional BL)

  or
- **PredictOn** is LOW

  or
- A prefetch abort occurs when fetching the instruction

  or
- Instruction[31:28]==“1111” (Invalid condition code)

  or
- Instruction[27:25]!=“101” (Non-branch instruction)

otherwise the instruction will be predicted as taken if:

- Instruction[31:28]==“1110” (Always condition code)

  or
- Instruction[24]==“0” AND Instruction[23]==“1” (Backwards branch)

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

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

Consequences of branch prediction and the prefetch buffer

Due to the speculative prefetching of instructions that the Prefetch Unit performs, it is possible for the prefetch buffer to contain incorrect instructions. In such circumstances the prefetch buffer must be flushed, and ARM8 provides a means to do this with the IMB instruction. Please refer to ***4.17 The Instruction Memory Barrier (IMB) Instruction*** on page 4-62 for details of when and how to use the IMB instruction.

5.3.3 Turning branch prediction on and off

Branch Prediction is turned on by setting **PredictOn** HIGH and turned off by setting **PredictOn** LOW. Turning branch prediction off by switching the **PredictOn** signal from HIGH to LOW simply disables the Branch Predictor. If the Prefetch Unit is already prefetching from a speculative execution thread it will continue to do so. Thus simply switching **PredictOn** to LOW could allow that prefetch thread to fall through into a read-sensitive I/O location. In order to prevent such a situation, the following code sequence (or equivalent) should be used. This code has been written for a system that uses Coprocessor 15 register 1, bit 11, to control the state of **PredictOn** directly (as is the case in the ARM810 for instance). The code ensures that the Prefetch Unit is prefetching from a safe area when the Branch Predictor is turned off.

```
SysControl_Z    EQU     &00000800       ; PredictOn control bit 11

Branch_Predict_Off:

         ; code to save R0 and CPSR if required by your calling
         ; convention.


         ; *** Critical Code Section Starts here ***

         Branch_Predict_Bound:

         MRC   p15,0,R0,c1,c0,0 ;Read System Control
                  ;Register 1

         BIC R0,R0,#SysControl_Z;Turn off the PredictOn
                  ;bit
         MCR   p15,0,R0,c1,c0,0;Write back modified
                  ;control register 1

         ; Now the prefetching containment part:
         MSR   CPSR_flg, #0xF0000000 ;Set the carry so that
                  ;BCC will skip
         BCC   Branch_Predict_Bound ;Branch to direct the
                  ;                     prefetch activity
         ; This branch will never be taken. However backwards
         ; conditional branches are predicted taken, so the PU ;
         will be prefetching between the label
         ; Branch_Predict_Bound and this instruction when the
         ; MCR is executed.

         ; *** Critical Code Section Ends here ***

         ; Return to caller with normal calling convention
         ; restoring CPSR and R0 if required.
         MOV    PC, LR   (or whatever)
```

This code sequence is highly implementation-specific. It is strongly recommended that software systems are structured so that turning off Branch Prediction is performed by one piece of system software that can be easily updated when your software is migrated to future ARM Microprocessors.

6 Memory Interface

This chapter describes the ARM8 memory interface.

Memory Interface

6.1 Overview

This interface users the concept of word buffers to store sequential data and instructions. There is a buffer for each, and this means that the sequentiality of instruction fetching need not be interrupted by data accesses; sequential instruction fetches can be made, even though data may have been read or written between fetches. This concept fits well with word lines in a cached system.

The ARM8 interface to the memory system is designed to read instructions or data at twice the bandwidth that instructions are required by the core. This enables the Prefetch Unit to improve the performance of the ARM8 by predicting Branches and removing them from the instruction stream that is presented to the Core, reducing the CPI of Branches.

Double bandwidth reads also improve the CPI of LDMs, and as a result, the memory-to-ARM8 interface is required to work at twice the processor clock frequency. The interface has been designed so that it is not necessary to turn around the direction of any bus at this speed; saving the resultant clock speed penalty of a guaranteed non-overlap time. The interface comprises:

- Three unidirectional 32-bit data buses:
  - VAddress
  - Wdata
  - Rdata
- Four control buses
  - ARequest
  - AResponse
  - RRequest
  - RResponse
- Five other control signals
  - Privileged
  - TwentySixBit
  - IExhausted
  - DExhausted
  - Confirm

The **VAddress** bus provides addresses to the memory system.

> Addresses are only provided by the Core or the Prefetch Unit when the instruction or data buffers are empty or need to be reloaded. This reduces the traffic on the **VAddress** bus at the cost of a small incrementer in the memory system.

The **Wdata** bus provides write data to the memory system.

The **Rdata** bus transfers read data and instructions from the memory system.

Please refer to ***2.3 ARM8 <-> Memory Interface Signals*** on page 2-4 for signal descriptions.

6.2 Memory Interface Timing

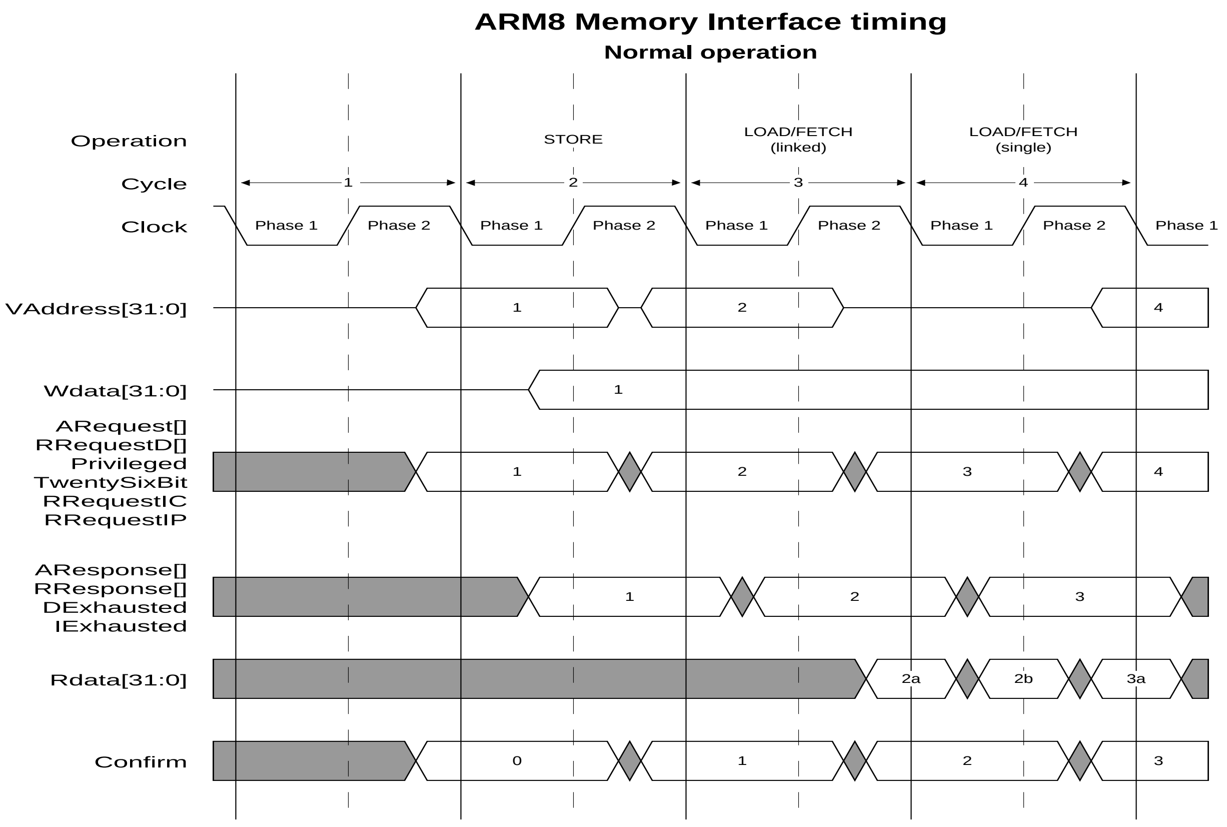


Figure 6-1: Memory interface timing diagram

6.3 Details of the Memory System Interface

This interface has been designed for use with a cached Memory System - such as in an ARM810 - and permits the streaming of Instructions and Data words over a single bus. The interface is based on the ARM8 issuing requests to the Memory System at the end of Phase 2, and receiving responses from the Memory System that are set up at the end of the next Phase 1 and are stable for Phase 2. Based on these responses from the Memory System, ARM8 will issue the next requests by the end of the next Phase 2. See ***Figure 6-1: Memory interface timing diagram*** on page 6-3.

This timing gives the Memory System less than one phase in which to give the correct response, so the response given is a provisional one at that point. The provisional response then has to be confirmed by the end of Phase 2 using the **Confirm** signal.

If **Confirm** is HIGH, ARM8 will respond by stopping its internal clock because the provisional response was wrong (for example: in a cached Memory System like ARM810, this would happen during a cache miss, an uncacheable read or write, an MMU Abort or an MMU TLB miss). Then once the Memory System has corrected itself, it will return the appropriate response to ARM8 and then bring **Confirm** LOW, allowing ARM8 to restart its clock.

The effect of this mechanism is that when ARM8 issues a request, it always expects the correct response at the end of the cycle. If the cache cannot respond correctly within one cycle, then ARM8's internal clock is stopped until the correct response can be given. The clock stopping holds the clock in Phase 2.

6.3.1 Types of requests to the memory system

There are two types of request that ARM8 can make, these are described below.

Access requests

ACCESS requests are those associated with addresses on **VAddress[31:0]** and also write data on **Wdata[31:0]** being driven by ARM8. These requests can be to load data or instructions from the address specified by **VAddress[]**, or to store the data presently on **Wdata[]** at the address specified by **VAddress[]**.

Return requests

RETURN requests are those associated with reading instructions or data words from the Memory System. These words are ready for ARM8, one at the end of Phase 2 and other at the end of Phase 1 of the next cycle on the **Rdata[31:0]** bus.

Making requests to the memory system

ACCESS and RETURN requests may be issued together at the end of each phase 2: either because they are linked or because the requests can be handled together due to the separate write and read data buses (**Wdata[]** and **Rdata[]** respectively).
The requests are shown in ***Table 6-1: Memory system requests*** on page 6-5.

Type	Request
Linked	• ACCESS request to load Instructions into the instruction buffer, and RETURN request for the same instructions from the instruction buffer. • ACCESS request to load Data into the data buffer, and RETURN request to get the same data from the data buffer.
Single	• RETURN request to get instructions, previously loaded into the instruction buffer. • RETURN request to get data, previously loaded into the data buffer. • ACCESS request to store data at the specified address on **VAddress[]**.
Parallel	• ACCESS request to store data, and a RETURN request to get instructions, previously loaded into the instruction buffer.

Table 6-1: Memory system requests

6.3.2 Summary of access request types (ARequest[])

The types of Access Requests that can be made are shown below in ***Figure 6-2: Access request type summary***.

Note: This table does not show the signal encodings of these types.

(M) indicates that there are more words to be loaded or stored as part of the same load or store multiple.

SIGNAL	Operation/Description	Access type	Relevance
AREQ_NONE	None		
AREQ_LOAD(M)	Load Data from the address on **VAddress[]**.	Normal	LDR/LDM
AREQ_LOAD_S(M)	Load Data from the address on **VAddress[],** the address being sequential to the previous **AREQ_LOAD** or **AREQ_LOAD_S**.	Sequential to last Data Load	LDM
AREQ_LOAD_B	Load Data from the address on **VAddress[]**. The byte must be supplied on **Rdata[]** in the correct bit positions: **Address MOD 4** / **Endian** / **Rdata bits** 0 / Little / 7:0 1 / Little / 15:8 2 / Little / 23:16 3 / Little / 31:24 0 / Big / 31:24 1 / Big / 23:16 2 / Big / 15:8 3 / Big / 7:0 The other 24 bits are ignored.	Byte	LDRB LDRSB

Figure 6-2: Access request type summary

SIGNAL	Operation/Description	Access type	Relevance
AREQ_LOAD_H	Load Data from the address on **VAddress[]**. The halfword must be supplied on **Rdata[]** in the correct bit positions: Address MOD 4 / Endian / Rdata bits 0 / Little / 15:0 2 / Little / 31:16 0 / Big / 31:16 2 / Big / 15:0 The behaviour when the Address MOD 4 is 1 or 3 is left undefined and should not be used. The other 16 bits are ignored.	Half-Word	LDRH LDRSH
AREQ_LOAD_X	Do the Load part of a SWP with the Data from the address on **VAddress[]**.	Swap	SWP
AREQ_LOAD_BX	Do the Load part of a SWPB with the Data from the address on **VAddress[]**.	Byte Swap	SWPB
AREQ_FETCH	Load Instruction(s) from the address given on **VAddress[]**.	Normal	
AREQ_FETCH_S	Load Instruction(s) from the address given on **VAddress[]** the address being sequential to the previous instruction fetch. (A load/load multiple or store/store multiple could occur between instruction fetches and these fetches would still be considered as sequential.)	Sequential to last Instruction fetch.	
AREQ_SPEC	A speculative request to load instruction(s) from the address given on **VAddress[]**. The memory system does not have to supply these instruction(s) as this is a speculative request.	Normal, speculative	
AREQ_SPEC_S	A speculative request to load instruction(s) from the address sequential to the last instruction loaded. The memory system does not have to supply these instruction(s) as this is a speculative request. (A load/load multiple or store/store multiple could occur between instruction fetches and these fetches would still be considered as sequential.)	Sequential to last instruction fetch, speculative	
AREQ_STORE(M)	Store the Word put onto **Wdata[]** during the next Phase 1 at the address currently on **Vaddress[]**.	Normal	STR/STM
AREQ_STORE_S(M)	Store the Word put onto **Wdata[]** during the next Phase 1 at the address sequential to the previous store address.	Sequential to the last Data store request (the next Word)	STM

Figure 6-2: Access request type summary (Continued)

SIGNAL	Operation/Description	Access type	Relevance
AREQ_STORE_B	Store the Byte put onto **Wdata[]** during the next Phase 1 at the address currently on **VAddress[]**. The Byte will be replicated four times across all of **Wdata[]**.	Byte	STRB
AREQ_STORE_H	Store the Halfword put onto **Wdata[]** during the next Phase 1 at the address currently on **VAddress[]**. The Halfword will be replicated twice across all of **Wdata[]**.	Half_Word	STRH
AREQ_STORE_X	Do the Store part of a SWP to the address on **VAddress[]**. The Data Word will be driven onto **Wdata[]** during the following Phase 1.	Swap	SWP
AREQ_STORE_BX	Do the Store part of a SWPB to the address on **VAddress[]**. The Data Byte will be replicated four times and driven across all of **Wdata[]** in the following Phase 1.	Byte	SWPB
AREQ_CONTROL	An MCR instruction is being executed with the data put onto **VAddress[]**. See ***Chapter 7, Coprocessor Interface***. This allows an MCR instruction to control the memory system.	Special	MCR

Figure 6-2: Access request type summary (Continued)

6.3.3 Instruction return requests (RRequestIC, RRequestIP)

Instruction RETURN requests are specified via the **RRequestIC** and **RRequestIP** signals. If **RRequestIC** or **RRequestIP** is HIGH at the end of Phase 2, two instruction words have been requested, otherwise none have been requested.

Note If both a data and an instruction RETURN request is given at the same time, the memory system must give priority to the data RETURN request.

6.3.4 Summary of data return request types (RRequestD[])

Table 6-2: Return request type summary shows the types of data Return Requests.

Note: This does not show the signal encodings of these types.

Signal	Operation
RREQD_NONE	Return 0 Data words from the Memory System
RREQD_ONE	Return 1 Data word from the Memory System
RREQD_TWO	Return 2 Data words from the Memory System

Table 6-2: Return request type summary

6.4 Types of Responses from the Memory System

There are two types of responses that the Memory System can make:

- Access Responses
- Return Responses

Access responses

These indicate a response to the ACCESS request and show :

- no ACCESS request was made or the speculative instruction fetch has not been performed.
- the MMU generated an abort
- the ACCESS request has completed

Table 6-3: Access response type summary shows the types of Access Responses, but does not show the signal encodings of these types.

Signal	Description
ARESP_NOTDONE	No ACCESS request was made or the speculative instruction request has not been performed
ARESP_ABORT	The ACCESS Request has completed with an (MMU) generated abort
ARESP_DONE	The ACCESS Request has completed normally or an external abort has occurred

Table 6-3: Access response type summary

Return responses

These indicate a response to the RETURN request and show:

1. On the **RResponse** bus:
   a) no data or instruction words are being returned
   b) whether there was an abort associated with the data or instructions being returned
   c) what part of the request was granted (eg. whether data or instructions are being returned, and whether one word or two is being returned)
   d) whether a speculative prefetch was granted or not
2. On the **IExhausted** and **DExhausted** signals:
   a) whether it will be possible to request sequential instructions and/or data after the current RETURN request is complete

Table 6-4: Return response type summary shows the types of these Return Responses, but does not show the signal encodings of these types.

Signal	Description
RRESP_NOTHING	No data or instruction words are being returned because: • No RETURN request was made, or • No instructions are being returned for the speculative instruction fetch request, or • An access request has completed with an MMU abort.
RRESP_EXTABORT_D	An External Abort was generated on the Data RETURN Request. If an Instruction RETURN Request exists, then it has not succeeded and must be re-issued if it is still wanted. This may need a linked ACCESS request if the instruction buffer is empty.
RRESP_EXTABORT_I	An External Abort was generated on the Instruction RETURN Request AND there is no Data RETURN Request.
RRESP_DATA1	The Data RETURN Request has completed normally and is returning 1 Data word. If an Instruction RETURN Request exists, then it has not succeeded and must be re-issued if it is still wanted.This may need a linked ACCESS request if the instruction buffer is empty.
RRESP_DATA2	The Data RETURN Request has completed normally and is returning 2 Data words; if an Instruction RETURN Request exists, then it has not succeeded and must be re-issued if it is still wanted. This may need a linked ACCESS request if the instruction buffer is empty.
RRESP_INSTR1	The Instruction RETURN Request has completed normally and is returning 1 Instruction word. No Data RETURN Request was made.
RRESP_INSTR2	The Instruction RETURN Request has completed normally and is returning 2 Instruction words. No Data RETURN Request was made.

Table 6-4: Return response type summary

7 Coprocessor Interface

This chapter describes the interface between the ARM8 and any on-chip coprocessors, and gives descriptions and timing diagrams to show its operation.

Coprocessor Interface

7.1 Introduction

ARM8 only supports register transfer operations on this interface. All other coprocessor instructions must be bounced by the coprocessor so that they will take the Undefined Instruction trap.

For a description of all the interface signals referred to in this chapter please refer to ***2.4 ARM8 <-> Co-processor Interface Signals*** on page 2-6.

In the Timing Diagrams that follow, **P** and **N** refer to the Previous and Next instruction's signal space. The values of the signals during these times are not relevant to the instruction under consideration.

Note: This interface will only support on-chip coprocessors directly. If the use of an off-chip coprocessors is required, then this will have to be done through an on-chip "interfacing" coprocessor. This document does not address the design of such an "interfacing" coprocessor.

7.2 Overview

Coprocessors need to determine which instructions they are supposed to execute. ARM8 does this by means of a “pipeline follower” in the coprocessor; as each instruction arrives from memory, it enters the coprocessor’s pipeline follower in parallel with going into the real ARM pipeline. As each instruction enters the Execute stage of the ARM8 pipeline the processor informs the coprocessor whether the instruction needs to be executed (the condition codes may indicate otherwise for example).

The following summarizes the interface:

1. The ARM8 coprocessor interface follows ARM8’s pipeline with a delay of 1 phase.
2. Many things can happen to an instruction as it progresses through ARM8’s pipeline. For example, an instruction might get cancelled in the Decode stage of the core and then be overwritten by another instruction from the Prefetch Unit whilst a multi-cycle instruction is executing. The instruction thus “vanishes” from ARM8’s pipeline. In order that the coprocessor can track this, ARM8 provides a signal (**CEnterExecute**) to the coprocessor indicating that an instruction is moving from the Decode stage to the Execute stage of its pipeline.
3. Coprocessors typically receive their final confirmation for instruction execution (via **CExecute**) during the second phase after the instruction enters the Execute stage of ARM8.
4. All instructions are broadcast to coprocessors. Only if the instruction is a coprocessor instruction will the coprocessors be offered a chance to execute it. Otherwise, the coprocessors are told that the instruction is not a coprocessor instruction. (Note: this means that coprocessors will no longer be required to reject Undefined instructions that are not Coprocessor instructions such as in the ARM7.)

7.3 Operational Summary

The bus for transferring instructions to the coprocessor(s) is called **CInstruct[25:0]**. **CInstruct[]** is only 26 bits wide as coprocessors have no need to access the condition code (bits 31:28), and because all coprocessor instructions have bits 27 and 26 equal to 1.

The coprocessor is told whether the instruction entering the Decode stage is a coprocessor instruction or not. This is done by forcing **CInstruct[25:24]** to be “11” for non-coprocessor instructions. Any other value on **CInstruct[25:24]** indicates a coprocessor instruction.

This interface does not offer Undefined instructions to the coprocessor. So, for example, instructions with bit pattern xxxx 011x xxxx xxxx xxxx xxxx xxx1 xxxx will not be offered for execution to the coprocessor.

The signal **CEnterDecode** indicates that the instruction on the **CInstruct[]** bus has just entered ARM8’s Decode stage. Signal **CEnterExecute** indicates that the instruction in ARM8’s Decode stage has just entered the ARM8’s Execute stage.

Normally, one phase after **CEnterExecute** is asserted, signal **CExecute** is asserted if ARM8 has decided that the instruction really needs to be executed - this means that it has not failed its condition, been cancelled by the Prefetch Unit or been discarded because of a data abort in the preceding instruction. See ***7.5 Busy-Waiting and Interrupts*** on page 7-6 for further information about the rules for making permanent changes to coprocessor state.

While the coprocessor can legitimately start executing the instruction before **CExecute** is asserted, and indeed is expected to in some cases, the instruction must not be allowed to cause any permanent changes to coprocessor state until the phase 1 after **CExecute** has been asserted. (Note that transmitting data to ARM8 via **CData[]** does not constitute a permanent change: ARM8 is responsible for ensuring that such data is discarded if the instruction is not to be executed.)

7.4 Data Buses

One bidirectional data bus exists between ARM8 and the (on-chip) coprocessors: **CData[31:0]**. This is used to transport register data for MRC and MCR instructions between ARM8 and the coprocessor. Values on the bus change during Phase 1 of the cycle that the instruction occupies in the Execute stage of ARM8's pipeline, or equivalently in any Phase 1 when it will be in the Execute stage of the coprocessor pipeline in the next phase.

ARM8 has the added functionality of putting MCR data onto **VAddress** in phase 2 of the cycle that the instruction occupies in the execute stage of ARM8's pipeline. The first implementation of ARM8 makes use of this functionality to simplify the memory system routing and does not use the **CData** during MCR instructions. In this implementation, **CData** is just treated as a unidirectional input bus.

7.5 Busy-Waiting and Interrupts

The coprocessor is permitted to stall (or “busy-wait”) the processor during execution of a coprocessor instruction if, for example, it is still busy with an earlier coprocessor instruction. To do so, the coprocessor associated with the decode stage instruction asserts a signal **CBusyWaitD** during Phase 2. When the instruction concerned advances to the Execute stage of ARM8’s pipeline (as indicated by **CEnterExecute**), it enters a busy-wait loop. The coprocessor may then assert a signal **CBusyWaitE** during phase 2 for as many cycles as it wants in order to keep the instruction in the busy-wait loop.

If the instruction is not busy-waiting when it enters the Execute stage, **CBusyWaitE** must be set to LOW by the coprocessor.

For interrupt latency reasons ARM8 may be interrupted from a busy-wait state, causing execution of the instruction to be abandoned. Abandoning execution is done through an extension to the definition of the **CExecute** signal: in addition to being significant one phase after **CEnterExecute** is asserted, it is also significant on each subsequent cycle for as long as the instruction is being busy-waited.

During the busy-wait loop:

CExecute=0	means execution of the instruction must no longer be attempted
CExecute=1	means the instruction should still be executed.

This means that permanent changes to coprocessor state may not occur until **CExecute**=1 at the start of the following Phase 1 in which the instruction is no longer being busy-waited.

The following figures show the signal timings of busy-waited instructions:

Figure 7-2: 1-cycle busy-waited operation of an MCR instruction

Figure 7-4: 1-cycle busy-waited operation of an MRC instruction

Figure 7-5: 2-cycle busy-waited operation of an MRC instruction

7.6 MCR Instructions

In the same Phase 1 that **CEnterExecute** is asserted (Phase 1 of the Execute cycle of ARM8's pipeline), ARM8 drives the data from the ARM register (Rd) on to **CData[]**.

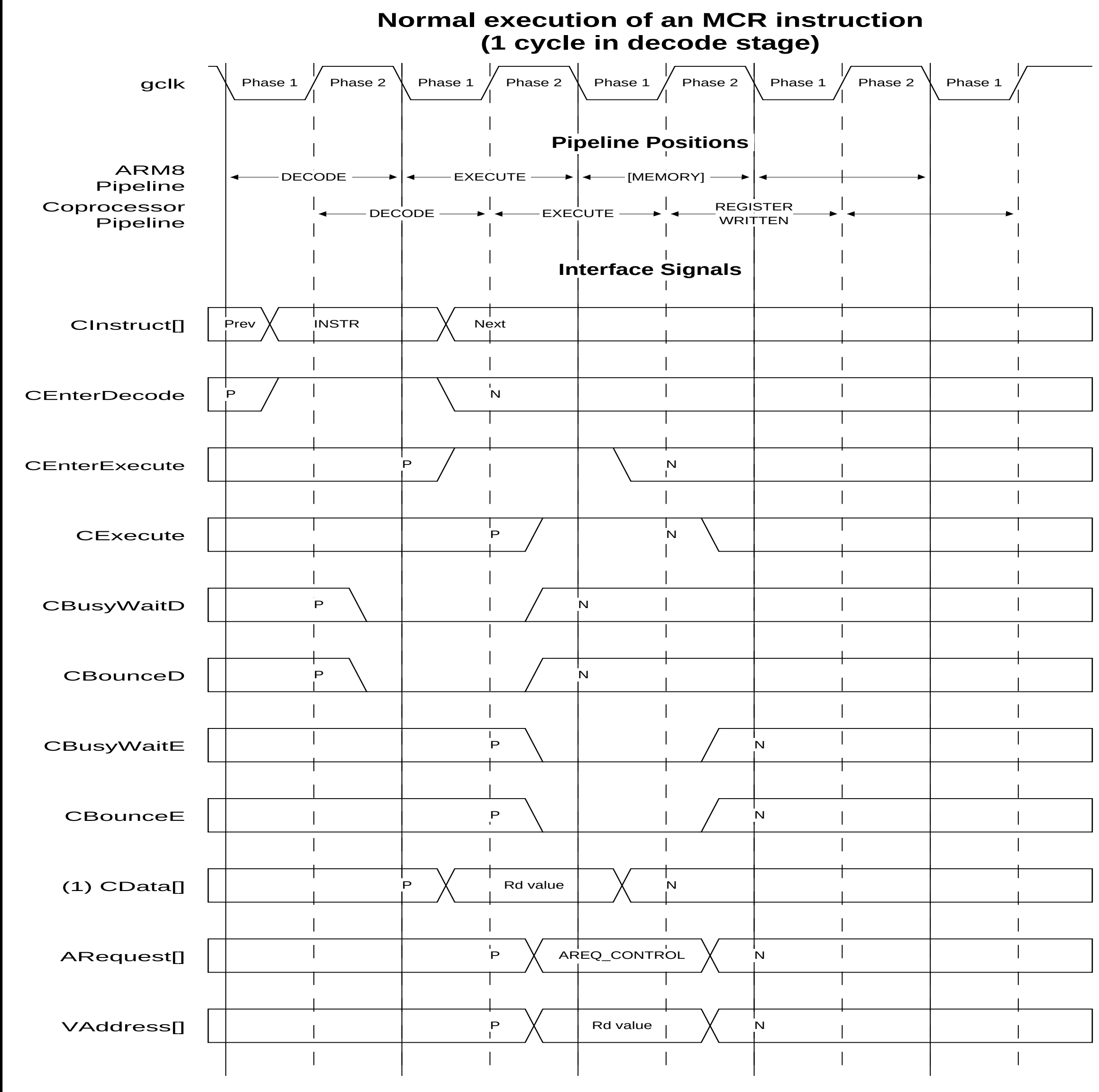
In addition, ARM8 uses the **ARequest[]** and **VAddress[]** memory interface signals to make a special memory ACCESS request during the following Phase 2. The memory request sets:

ARequest[]	equal to AREQ_CONTROL
VAddress[]	equal to the value of the ARM register (Rd)

This provides a mechanism for optional system-specific bus requests, that need to specify an address, to be made via the MCR instruction. An example in a cached system would be to flush a specific cache line.

Figure 7-1: Normal operation of an MCR instruction shows the signal timings for normal operation of an MCR instruction.

If the MCR instruction is busy-waited, then the same data is driven onto the **VAddress[]** bus each cycle until the instruction is executed and/or abandoned. On the coprocessor side, the only complication is that it must not write the data to its destination register until **CExecute** is asserted. The AREQ_CONTROL memory request is only made when, and if, the instruction is finally executed.



1 The first implementation does not drive CData

Figure 7-1: Normal operation of an MCR instruction

Figure 7-2: 1-cycle busy-waited operation of an MCR instruction shows the signal timings for an MCR instruction which is busy-waited for one cycle.

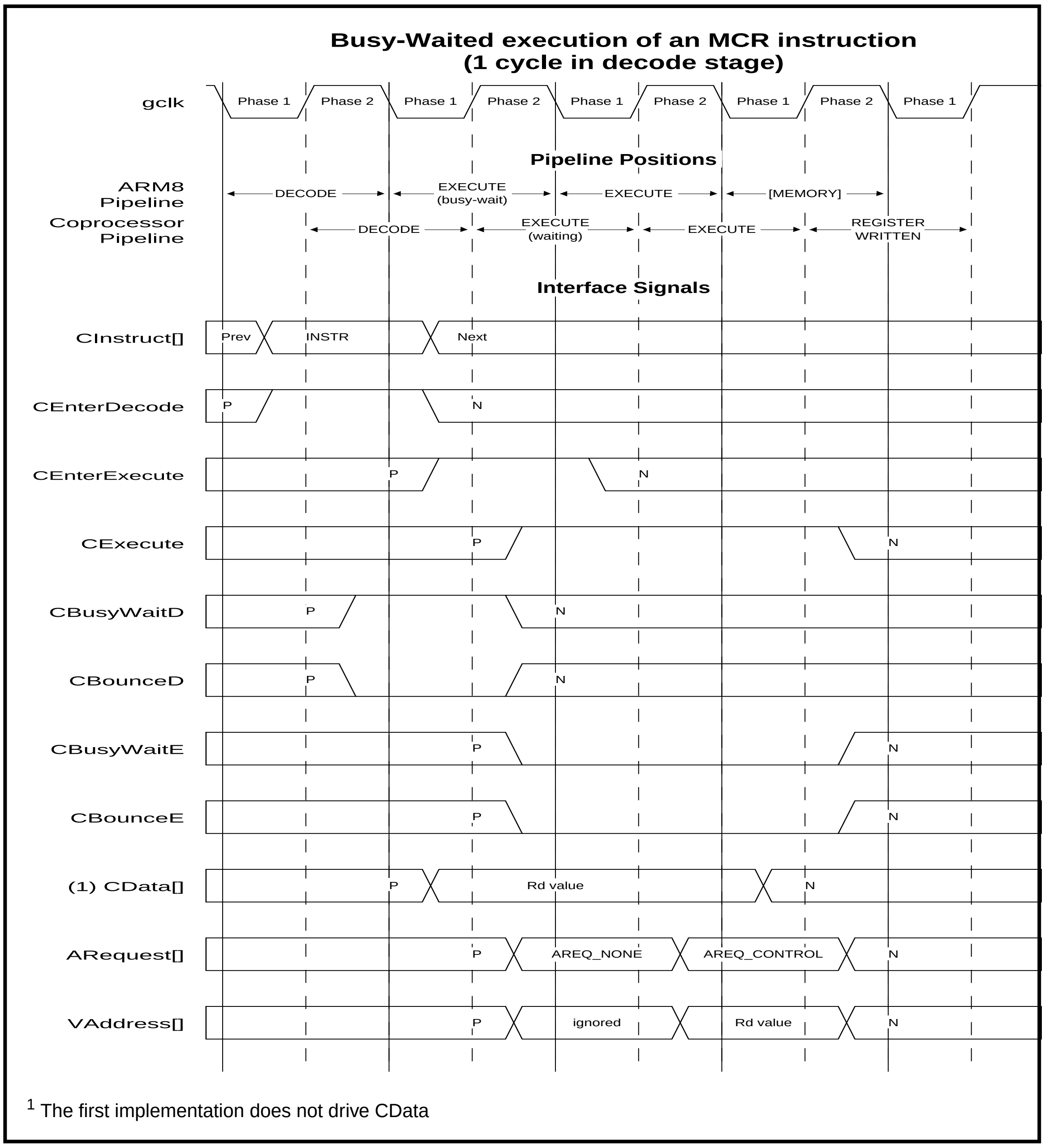


Figure 7-2: 1-cycle busy-waited operation of an MCR instruction

7.7 MRC Instructions

These are similar to MCR instructions with the main exception that the AREQ_CONTROL memory operation is not performed, and the data bus is driven to ARM8 from the coprocessor. The **CData[]** bus must be driven by the Coprocessor in Phase 1 of ARM8's Execute cycle.

Figure 7-3: Normal operation of an MRC instruction shows the signal timings for the normal operation of an MRC instruction.

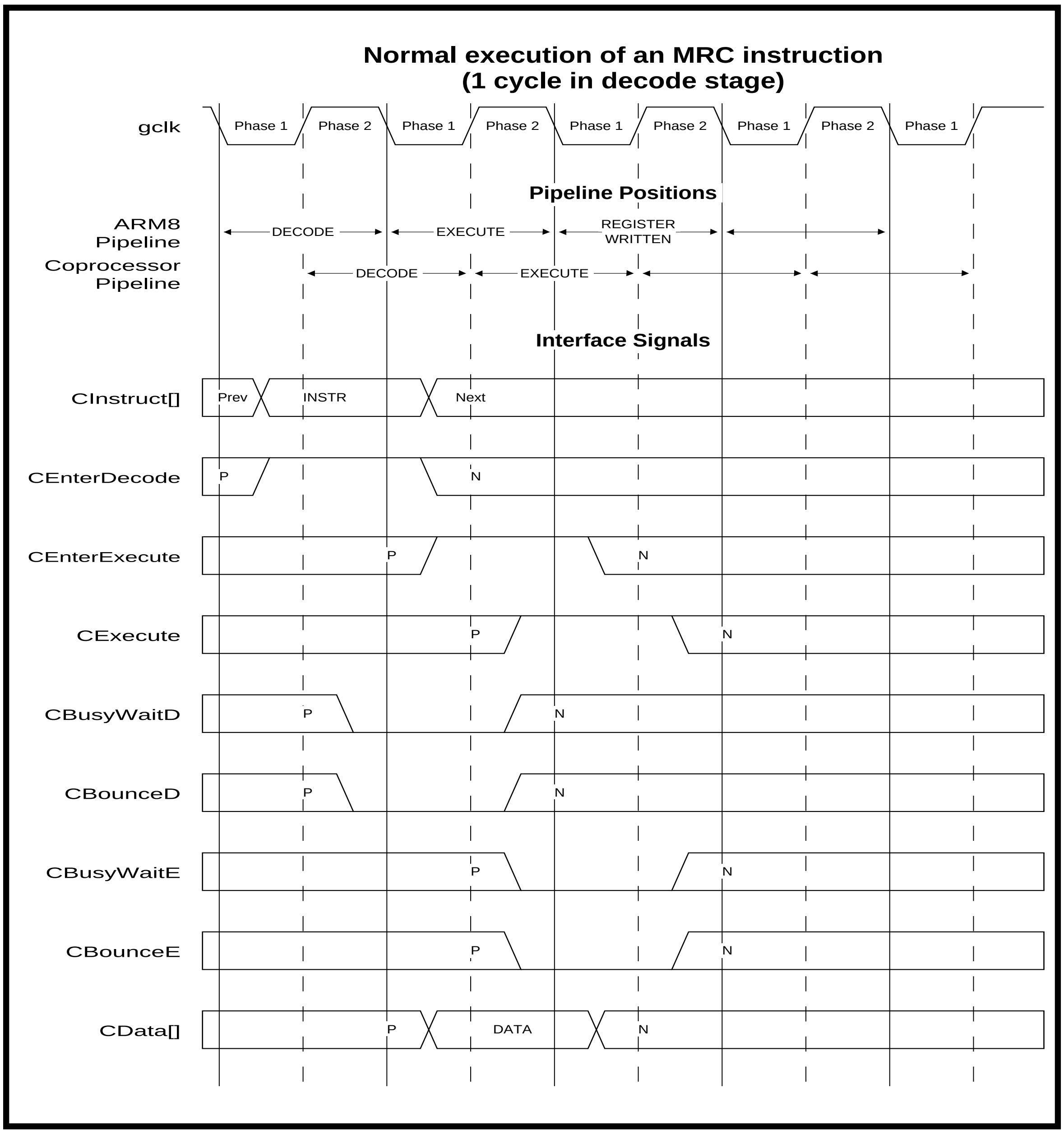


Figure 7-3: Normal operation of an MRC instruction

If the timing requirements, for the return of data onto **Cdata[]** in the same Phase 1 as **CEnterExecute** is asserted, is too stringent, the coprocessor can be designed to always busy-wait MRC instructions for one or more cycles. This gives more cycles in which to complete the operation.

If the MCR instruction is busy-waited, then the data only needs to be driven on to the **Cdata[]** bus for the cycle in which it is finally executed.

Figure 7-4: 1-cycle busy-waited operation of an MRC instruction shows the signal timings for an MRC instruction which is busy-waited for one cycle.

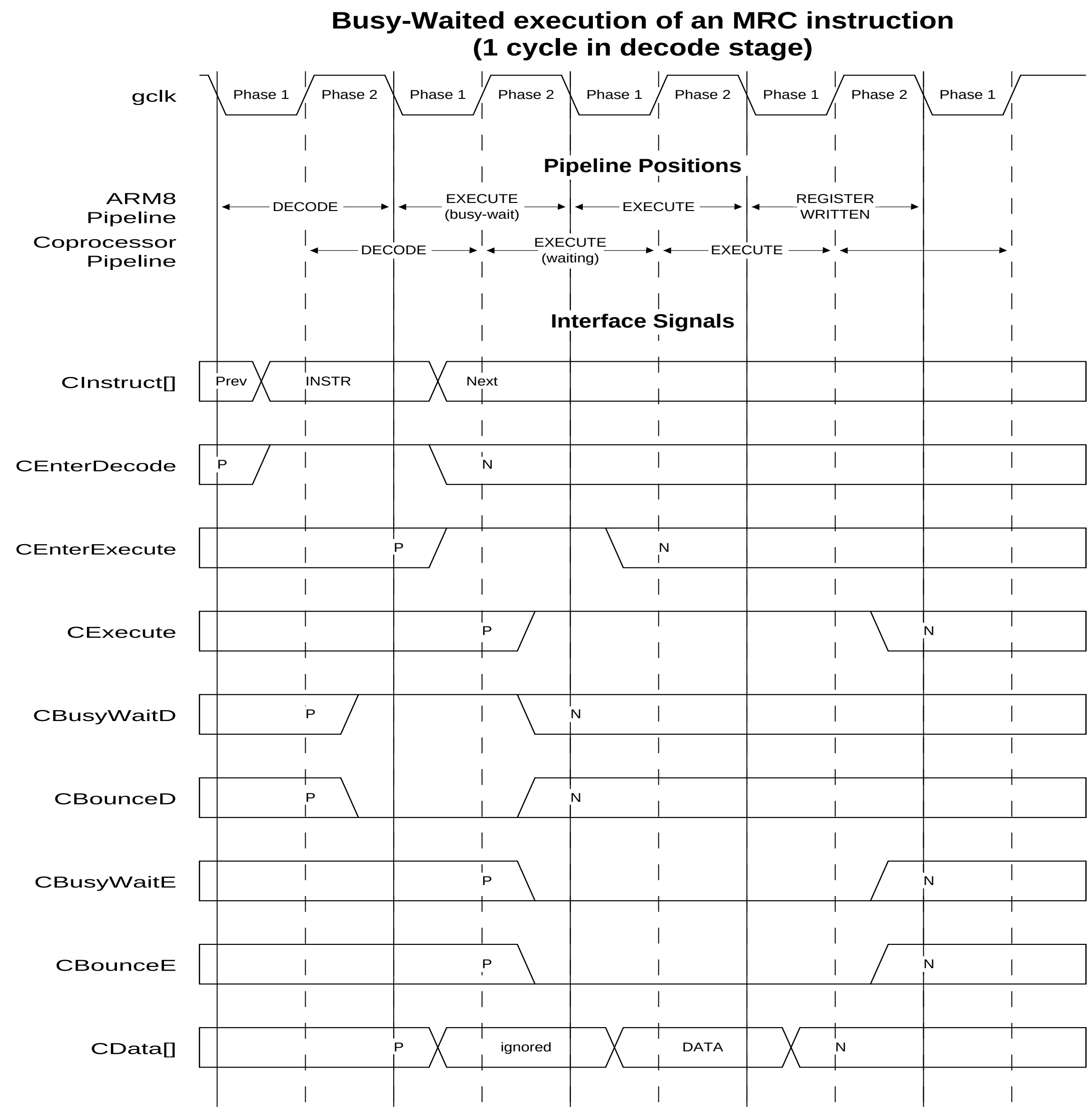


Figure 7-4: 1-cycle busy-waited operation of an MRC instruction

Figure 7-5: 2-cycle busy-waited operation of an MRC instruction shows the signal timings for an MRC instruction which is busy-waited for two cycles.

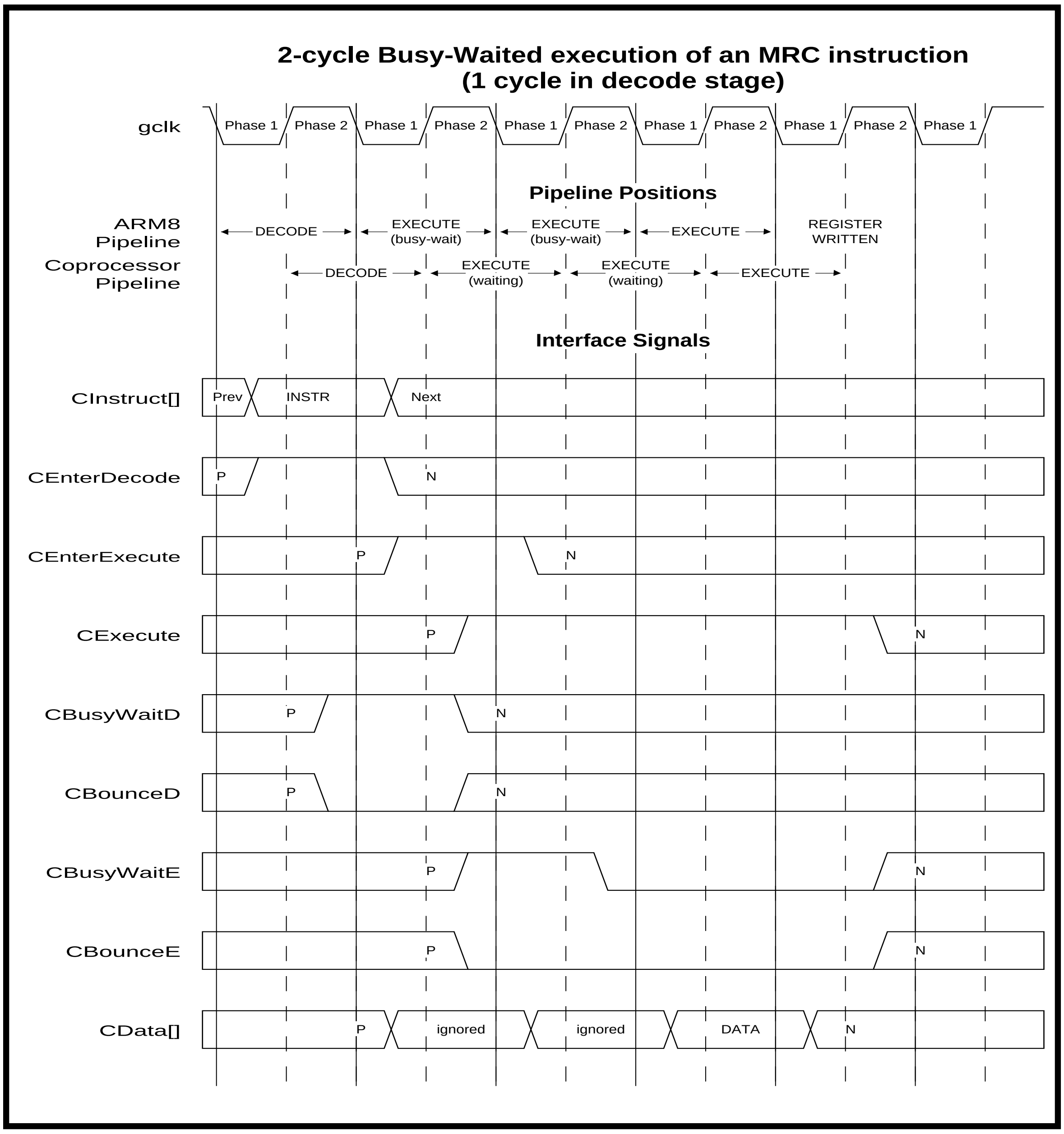


Figure 7-5: 2-cycle busy-waited operation of an MRC instruction

7.8 Cancelled Coprocessor Instructions

Figure 7-6: Cancelled instruction operation shows the signal timings for any coprocessor instruction that spends one cycle in the Decode stage, is not busy-waited, but fails its condition codes and is cancelled by ARM8.

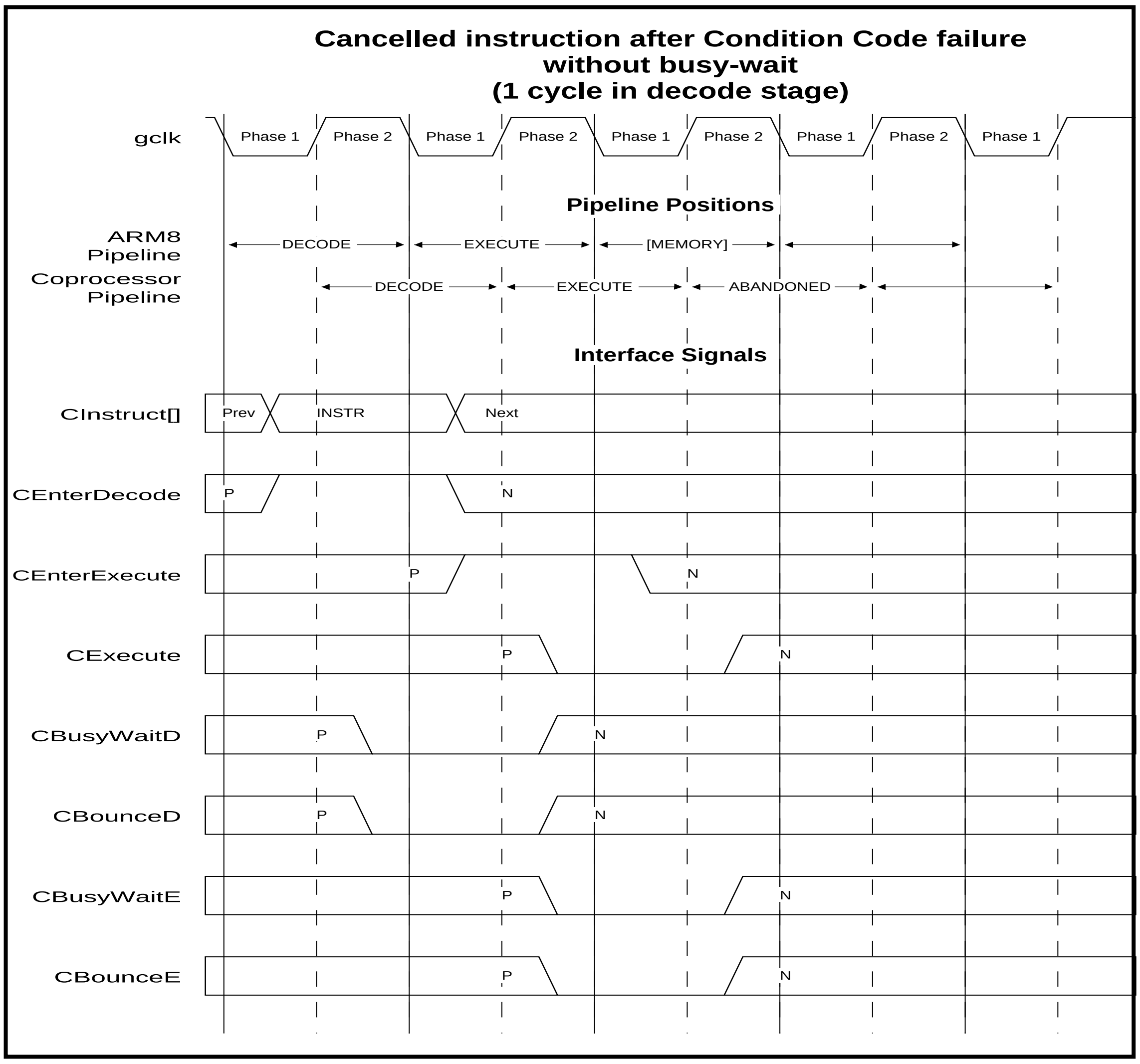


Figure 7-6: Cancelled instruction operation

7.9 Bouncing Coprocessor Instructions and Absent Coprocessors

7.9.1 Bouncing coprocessor instructions

A coprocessor can “bounce” coprocessor instructions if it is unable to deal with them itself. Such bounced instructions cause ARM8 to take the Undefined instruction exception vector, whereupon the bounced instruction can be emulated or dealt with in some other way.

The coprocessor asserts the signal **CBounceD** during Phase 2 to cause the bounce. If the instruction concerned advances to the Execute stage of ARM8’s pipeline during the following Phase 1 (as indicated by **CEnterExecute**), ARM8 treats the instruction as an Undefined Instruction rather than continuing to try and execute it.

Figure 7-7: Bounced instruction operation shows the signal timings for a Bounced Instruction.

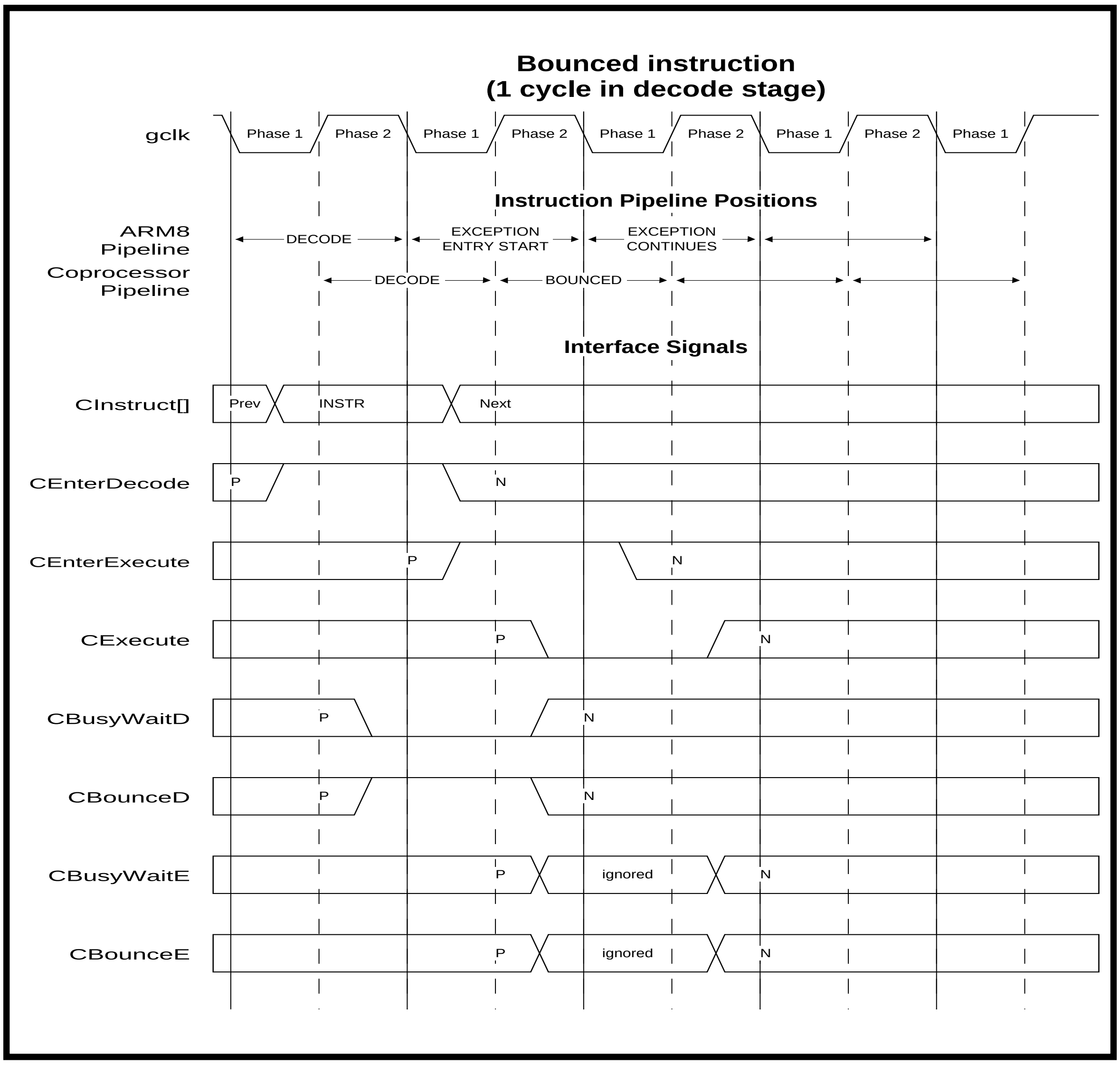


Figure 7-7: Bounced instruction operation

At any time that a coprocessor instruction is busy-waiting, the coprocessor can “bounce” it instead of continuing to busy-wait it or letting it proceed to execution. To do this, the coprocessor asserts the signal **CBounceE** during Phase 2. This causes the busy-wait loop to be abandoned and ARM8 will again treat the instruction as an Undefined instruction rather than continuing to try and execute it.

Figure 7-8: Busy-wait followed by bounce operation shows the signal timings for an instruction that is busy-waited then Bounced.

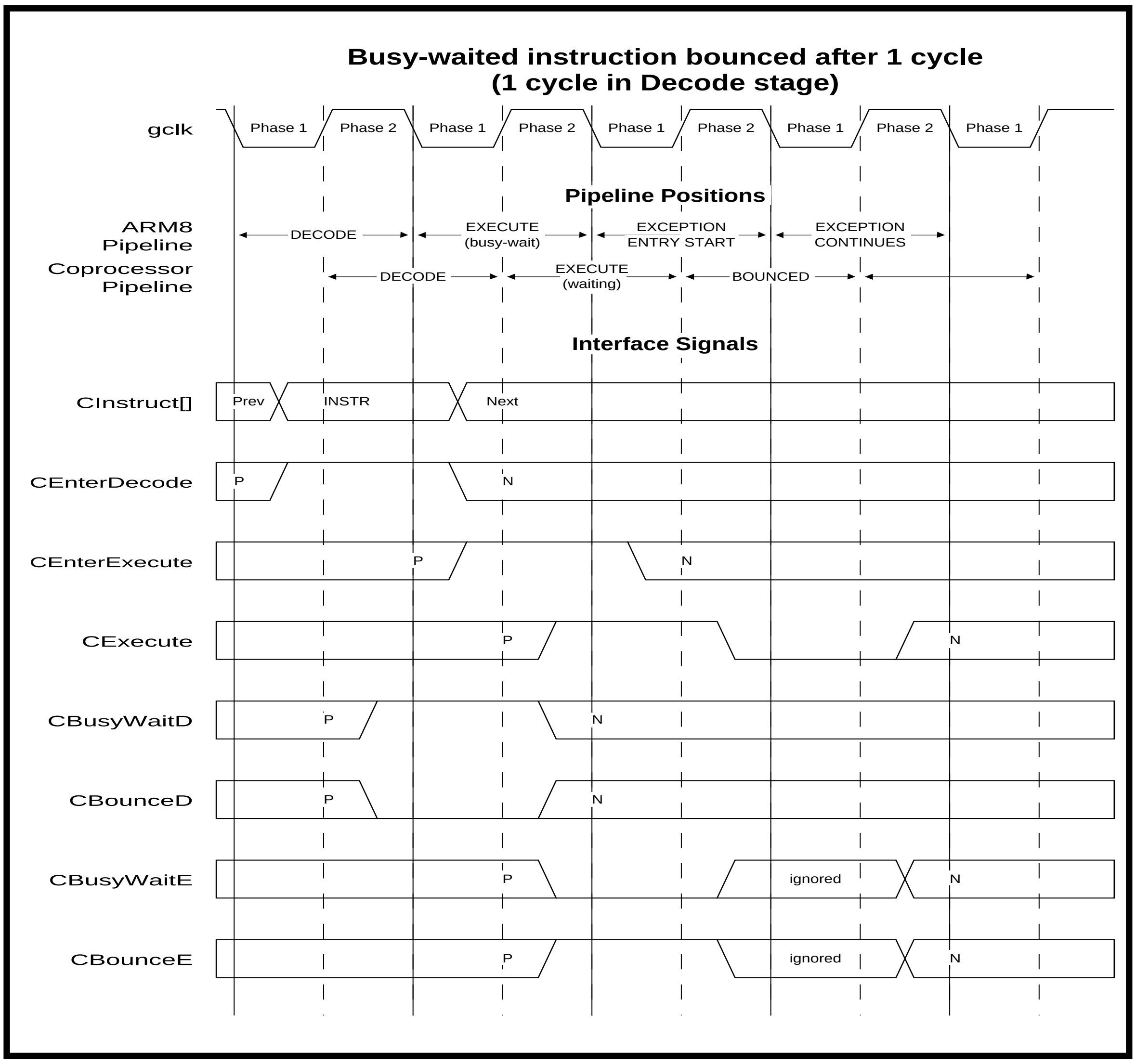


Figure 7-8: Busy-wait followed by bounce operation

7.9.2 Absent coprocessors

The ARM8 coprocessor interface assumes that every coprocessor exists, and requires that the hardware will respond to its instructions. This is essentially saying that some coprocessor in the system simply bounces every coprocessor instruction that is for an absent coprocessor.

For instance, in a system that only has coprocessor 15 (CP15) for system control, rather than simply ignoring coprocessors 0 to 14, the interface signals should be driven so as to bounce all coprocessor instructions except those for CP15.

As a second example, in a future chip with an on-chip “interfacing” coprocessor, CP15 would just handle coprocessor 15 instructions. The “interfacing” coprocessor would try to get any of the off-chip coprocessors to handle all instructions for coprocessors 0 to 14. This would be done by sending those instructions to the off-chip coprocessors and then bouncing them via CBounceE if no response occurred.

7.10 Interlocking

When the coprocessor reads an ARM register during an MCR instruction, it is possible that an interlock will occur if the previous instruction had just loaded the same register (and in some cases the second preceding instruction).

When this occurs, the coprocessor is prevented from executing the instruction by the **Interlock** signal from the ARM8. This signal is produced in Phase 2 of each cycle, and if it is HIGH at the end of Phase 1 then this means that the Execute cycle that ARM8 is about to start is interlocked, and will be repeated for the next cycle. The **Interlock** signal must not be evaluated until the end of Phase 1.

Figure 7-9: Interlocked MCR instruction operation shows the signal timings for an Interlocked MCR Instruction.

Any coprocessor is expected to respond to an Interlock by delaying any effects that the instruction may have until the next cycle. If the next cycle is not interlocked again, then the instruction will have its normal effect then - with the expectation that the **CExecute** signal will be produced a cycle later overall.

It is possible that ARM8 will interlock an instruction, and that the coprocessor will busy-wait it at the same time. These have very similar effects as both are expected to delay the real start of execution of the instruction. The delay should be maintained until the instruction is neither interlocked nor busy-waited. In particular, the coprocessor should not delay the production of the **CBusyWaitD** and **CBusyWaitE** signals because of an interlock.

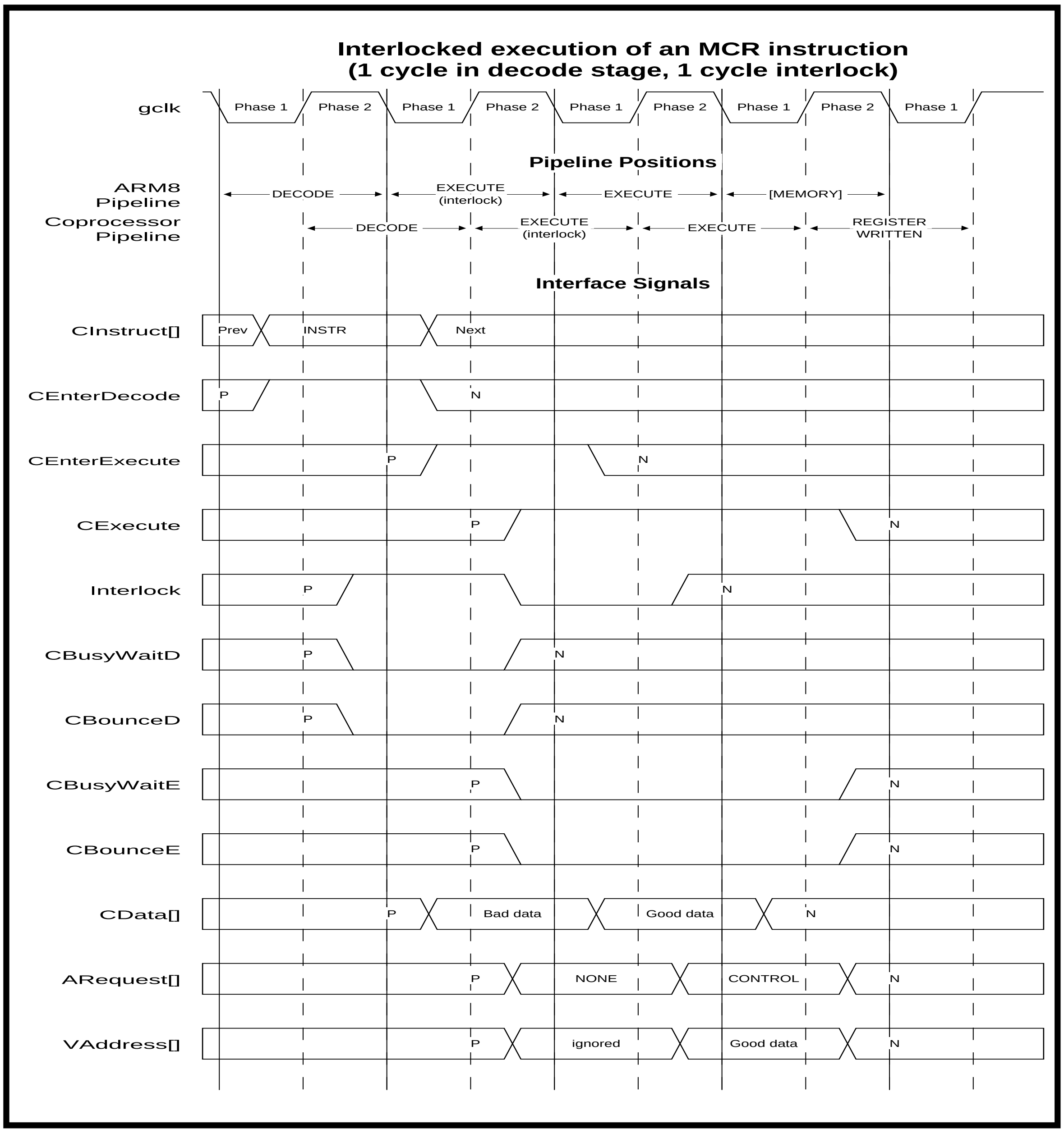


Figure 7-9: Interlocked MCR instruction operation

7.11 Coprocessor Instructions not Supported on this Interface

As indicated in the introduction, ARM8 only supports register transfers between the ARM and coprocessors. Thus, coprocessor instructions MCR and MRC are supported, and CDP, LDC and STC are not supported. These unsupported coprocessor instructions must be bounced by the coprocessor, otherwise ARM8 will behave unpredictably.

8 Instruction Cycle Timings Summary

This chapter summarises the cycle count timings for all ARM8 instructions.

Instruction Cycle Timings Summary

8.1 Branch and Branch with Link (B, BL)

The following table gives cycle counts for predicted branches based on the number of instructions that are in the PU FIFO at the time the prediction is made. This number will depend upon when the PU FIFO was last flushed, how many instruction fetch requests the PU has been able to make, and how many instructions were actually fetched.

Summary of Branch cycle counts	**Unpredicted**		**Predicted correctly**						**Predicted incorrectly**			
	Taken Branch	**Untaken Branch**	**As a taken Branch**				**As an untaken Branch**		**As a taken Branch (but not taken)**		**As an untaken Branch (but taken)**	
#Instructions in PU FIFO	**0+**	**0+**	**0**	**1**	**2**	**3+**	**0**	**1+**	**0-3**	**4+**	**0**	**1+**
B	3	1	3	2	1	0	1	0	1	2	3	3
BL[1]	3	1	3	2	1	1	1	1	N/A	N/A	N/A	N/A

Table 8-1: Branch cycle counts

[1] BL is only predicted if it is unconditional, and is therefore never predicted incorrectly.

Notes: Prefetch Unit flushes are caused by:

- an incorrect prediction of a taken branch (predicted not-taken)
- any SWI
- data processing operations that write to the PC (Rd=15)
- LDR that writes to the PC (Rd=15)
- LDM that writes to the PC (R15 in the register list)
- exception vector entry
- PU starvation due to core priority on successive LDRs, STRs, LDMs or STMs

8.2 PSR Transfers (MRS, MSR)

MRS takes 1 cycle.

MSR takes 1 cycle, + 2 cycles if the destination is CPSR and not CPSR_flag.

8.3 Data Processing Instructions

Instruction	Normal (Base) cycles	Complex Shift (other than LSL by 0,1,2 or 3)	If the PC is written, and the..		Register-specified shift
			S bit is NOT set	S bit is set	
ADC	1	+1	+2	+3	+1
ADD	1	+1	+2	+3	+1
AND	1		+2	+3	+1
BIC	1		+2	+3	+1
CMN	1	+1			+1
CMP	1	+1			+1
EOR	1		+2	+3	+1
MOV	1		+2	+3	+1
MVN	1		+2	+3	+1
ORR	1		+2	+3	+1
RSB	1	+1	+2	+3	+1
RSC	1	+1	+2	+3	+1
SBC	1	+1	+2	+3	+1
SUB	1	+1	+2	+3	+1
TEQ	1				+1
TST	1				+1

8.4 Multiply and Multiply-Accumulate (MUL, SMULL, UMULL, MLA, SMLAL, UMLAL)

The number of 8-bit multiplier array cycles required to complete the multiply is indicated by **m** in ***Table 8-2: Multiply and multiply-accumulate*** on page 8-4. This is controlled by the value of the multiplier operand specified by Rs:

m=1	Rs[31:8] are all either all 0s or all 1s (excepting all 1s for UMULL and UMLAL)
m=2	Rs[31:16] are all either all 0s or all 1s (excepting all 1s for UMULL and UMLAL)
m=3	Rs[31:24] are all either all 0s or all 1s (excepting all 1s for UMULL and UMLAL)
m=4	Otherwise

Instruction	m=4	m=3	m=2	m=1
Multiply	6	5	4	3
Multiply-Accumulate	6	5	4	3
Multiply long	7	6	5	4
Multiply-Accumulate long	7	6	5	4

Table 8-2: Multiply and multiply-accumulate

8.5 Block Data Transfers (LDM, STM)

LDM instruction

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

Table 8-3: LDM instruction

STM instruction

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

Table 8-4: STM instruction

8.6 Single Data Transfers (LDR, STR)

Instruction	Base cycle count	Register-specified Offset	Register-specified Offset with complex shift (LSL by anything other than 0,1,2 or 3)	Loading the PC (R15)
LDR,LDRH, LDRSB, LDRSH	1	-	+1	+4
STR, STRH	1	+1	-	n/a

Table 8-5: Single data transfers

8.7 Single Data Swap (SWP)

SWP2 cycles

8.8 Software Interrupt (SWI)

SWI4 cycles +f

where f is the number of cycles in the SWI service routine

8.9 Coprocessor Register Transfers (MRC, MCR)

MCR1 cycle + number of busy-wait cycles (if any)

MRC1 cycle + number of busy-wait cycles (if any)

8.10 Undefined Instructions

Undef4 cycles +h

where h is the number of cycles in the Undefined Instruction Trap service routine.

8.11 Interlocking Instructions

Pipelining in the ARM8 leads to cases where data loaded by one instruction cannot be used in the following instruction without incurring instruction cycle interlock penalties.

When a load instruction is followed by an instruction which wants to use the loaded value, a 1-cycle penalty is usually incurred. The penalty may be 0 or 2 in some cases.

When a load instruction is followed by a 1-cycle instruction and then an instruction that wants to use the loaded value, a 1-cycle penalty may be incurred.

9 AC Parameters

This chapter lists the AC parameters for the ARM8.

Note: this chapter is incomplete at this time and does not yet contain timing data.

AC Parameters

9.1 AC Parameters for the ARM8 Interface

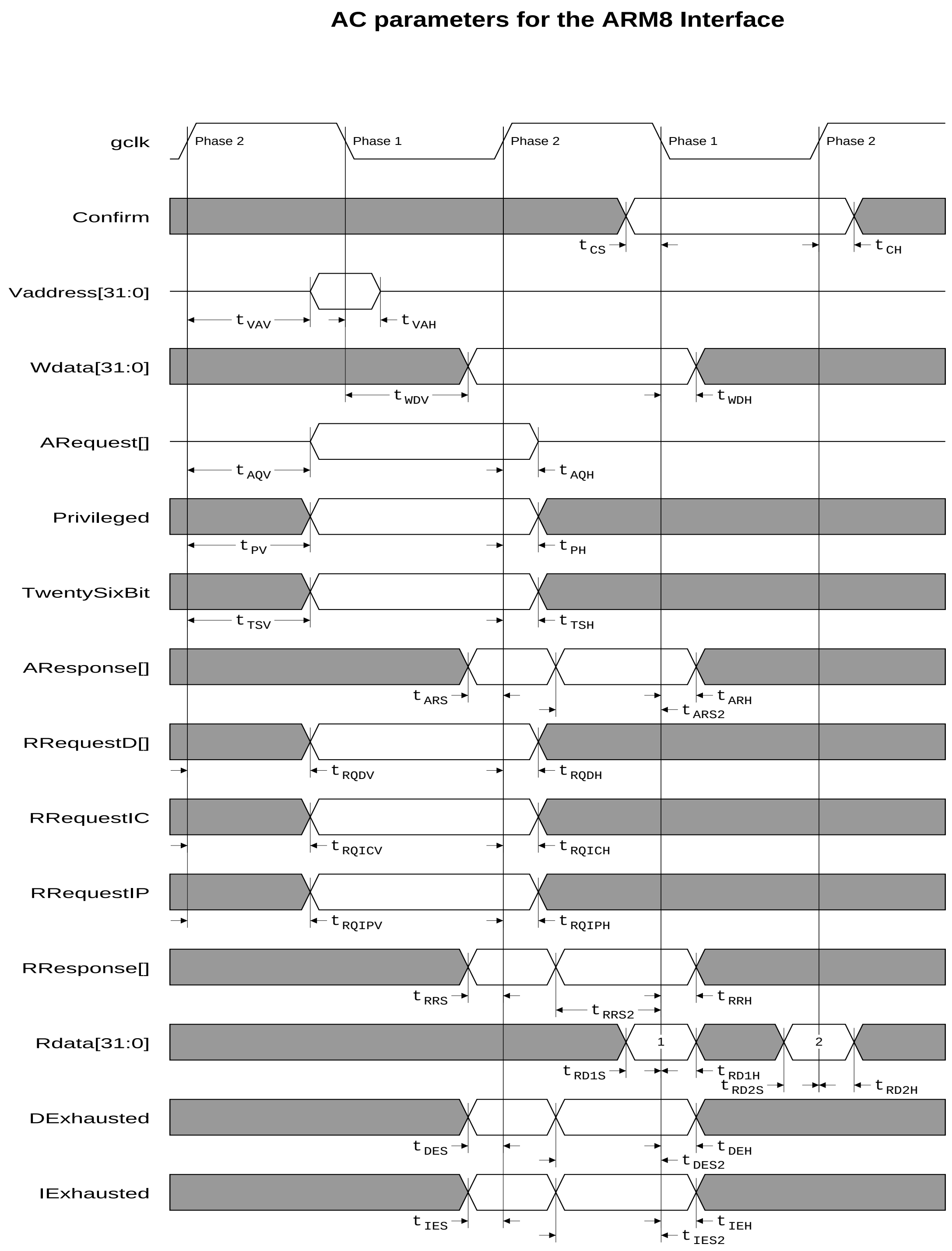


Figure 9-1: AC parameters for the ARM8 Interface

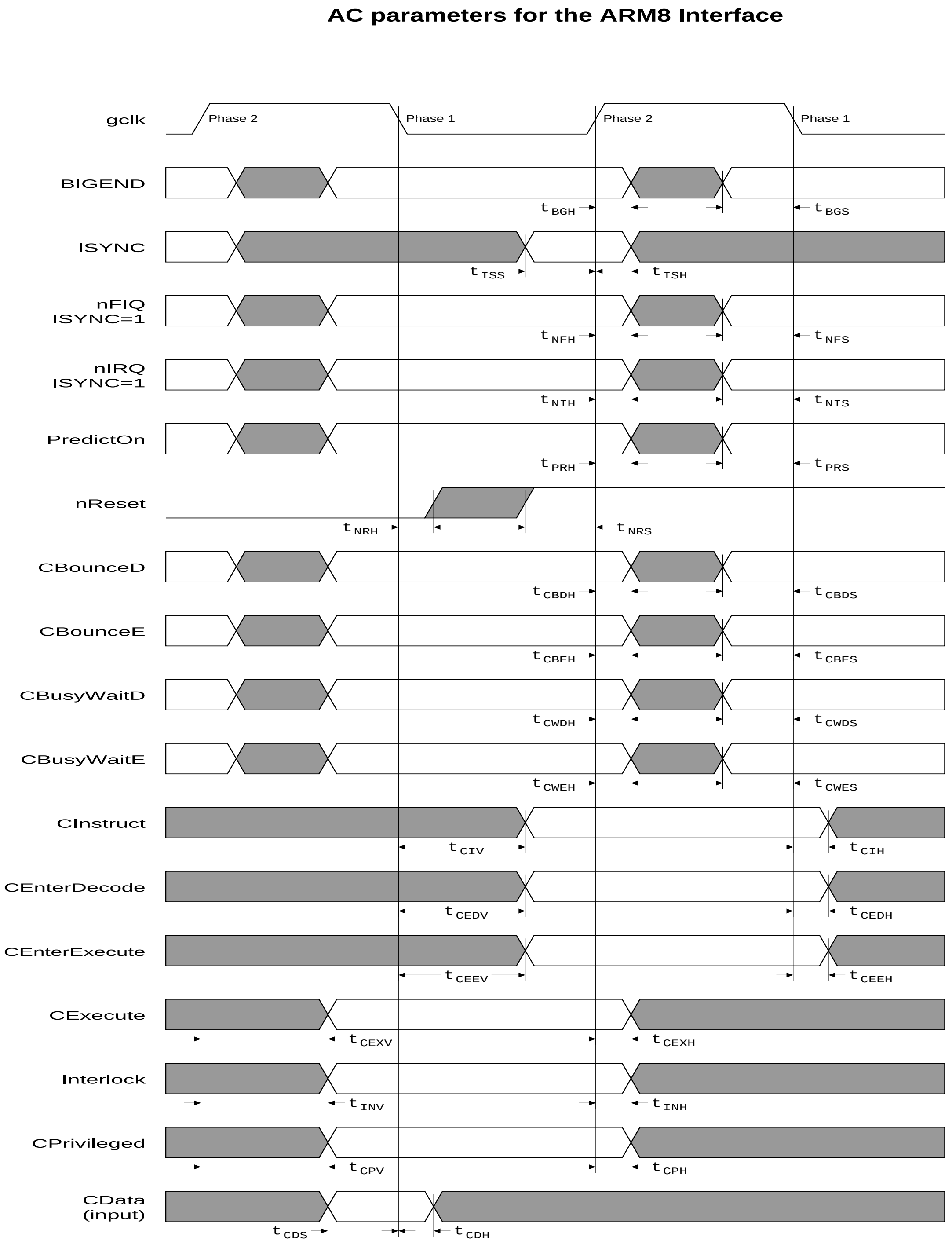


Figure 9-2: AC parameters for the ARM8 Interface

9.2 AC Parameters Table

Provisional: subject to change due to characterization.

Symbols	Parameters	Min (ns)	Max
tCKL	Clock low time	8	
tCKH	Clock high time	8	
tCS	Confirm setup to CKf	0.7	
tCH	Confirm hold from CKr	0.3	
tVAV	VAddress valid from Ckr	6.7	
tVAH	VAddress hold from CKf	0.5	
tWDV	Wdata valid from Ckf	3.5	
tWDH	Wdata hold from CKf	0.5	
tAQV	ARequest valid from Ckr	6.4	
tAQH	ARequest hold from CKr	0.5	
tPV	Privileged valid from Ckr	6.4	
tPH	Privileged hold from CKr	0.5	
tTSV	TwentySixBit valid from Ckr	5.0	
tTSH	TwentySixBit hold from CKr	0.5	
tARS	AResponse setup to CKr (1)	5.0	
tAR2S	AResponse setup to CKf (2)	9.0	
tARH	AResponse hold from CKf	0.3	
tRQDV	RRequestD valid from Ckr	6.4	
tRQDH	RRequestD hold from CKr	0.5	
tRQICV	RRequestIC valid from Ckr	6.4	
tRQICH	RRequestIC hold from CKr	0.5	
tRQIPV	RRequestIP valid from Ckr	6.4	
tRQIPH	RRequestIP hold from CKr	0.5	
tRRS	RResponse setup to CKr (1)	5.0	
tRR2S	RResponse setup to CKf (2)	9.0	
tRRH	RResponse hold from CKf	0.3	
tRD1S	Rdata ph2 data setup to CKf	1.5	
tRD1H	Rdata ph2 data hold from CKf	0.3	

Symbols	Parameters	Min (ns)	Max
tRD2S	Rdata ph1 data setup to CKr	5.0	
tRD2H	Rdata ph1 data hold from CKr	0.3	
tIES	IExhausted setup to CKr (1)	5.0	
tIE2S	IExhausted setup to CKf (2)	9.0	
tIEH	IExhausted hold from CKf	0.3	
tDES	DExhausted setup to CKr (1)	5.0	
tDE2S	DExhausted setup to CKf (2)	9.0	
tDEH	DExhausted hold from CKf	0.3	
tBGH	BIGEND hold from CKr	0.7	
tBGS	BIGEND setup to CKf	3.0	
tISS	ISYNC setup to CKr	2.0	
tISH	ISYNC hold from CKr	0.7	
tNFH	nFIQ hold from CKr	0.7	
tNFS	nFIQ setup to CKf	2.2	
tNIH	nIRQ hold from CKr	0.7	
tNIS	nIRQ setup to CKf	2.2	
tPRH	PredictOn hold from CKr	0.7	
tPRS	PredictOn setup to CKf	3.6	
tNRH	nReset hold from CKf	0.5	
tNRS	nReset setup to CKr	4.8	
tCBDH	CBounceD hold from CKr	0.7	
tCBDS	CBounceD setup to CKf	2.2	
tCBEH	CBounceE hold from CKr	0.7	
tCBES	CBounceE setup to CKf	2.2	
tCWDH	CBusyWaitD hold from CKr	0.7	
tCWDS	CBusyWaitD setup to CKf	2.2	
tCWEH	CBusyWaitE hold from CKr	0.7	
tCWES	CBusyWaitE setup to CKf	2.2	
tCIV	CInstruct valid from CKf	4.3	
tCIH	CInstruct hold from CKf	0.5	
tCEDV	CEnterDecode valid from CKf	4.3	
tCEDH	CEnterDecode hold from CKf	0.5	

AC Parameters

Symbols	Parameters	Min (ns)	Max
tCEEV	CEnterExecute valid from CKf	4.3	
tCEEH	CEnterExecute hold from CKf	0.5	
tCEXV	CExecute valid from CKr	4.3	
tCEXH	CExecute hold from CKr	0.5	
tINV	Interlock valid from CKr	7.9	
tINH	Interlock hold from CKr	0.5	
tCPV	CPrivileged valid from CKr	7.9	
tCPH	CPrivileged hold from CKr	0.5	
tCDS	CData setup to CKf	3.0	
tCDH	CData hold from CKf	0.7	

Notes

1 **AResponse**, **RResponse**, **DExhausted** and **IExhausted** are normally expected to change in phase 1 and meet **tARS**, **tRRS**, **tDES** and **tIES** respectively. However, if **Confirm** is used to indicate that the phase 1 (provisional) reponses were incorrect, then as far as ARM8 is concerned the clock is stretched while high and the new (modified responses) arrive in what looks like phase 2. In this case the second set of setup times: **tARS2**, **tRRS2**, **tDES2** and **tIES2** apply.

2 Meeting the phase 1 setup requirements ensures that the request signals (**ARequest**, **RRequestD**, **RRequestIC**, **RRequestIP**, **Privileged** and **TwentySixBit**) change monotonically. Chnges in the response signals when **Confirm** is active can result in combinatorial changes onto **Vaddress**, **ARequest**, **Privileged**, **TwentySixBit**, **RRequestD**, **RRequestIC** and **RRequestIP**.

10 DC Parameters

Not available for this release.

11 Backward Compatibility

This chapter summarises the changes in ARM8 when compared to previous ARM processors.

ARM8 will be able to run binary code targetted to earlier processors with only a few exceptions. The following changes from previous implementations of the ARM should be noted. See ***Appendix A, Instruction Set Changes*** for a more detailed discussion.

Backward Compatibility

11.1 Instruction Memory Barrier

The requirement for an Instruction Memory Barrier (IMB) instruction means that if code changes the instruction stream and then tries to execute it without an intervening IMB, the consequences will be unpredictable. For example, there must be an IMB instruction between loading code into memory and executing it. See ***4.17 The Instruction Memory Barrier (IMB) Instruction*** on page 4-62 for more information.

11.2 Undefined Instructions

In ARM8, unallocated instruction bit patterns in the instruction set space enter the Undefined Instruction trap. See ***4.18 Undefined Instructions*** on page 4-65 for further information.

11.3 PC Offset

Rare ARM7 instructions which read a stored R15 value from memory as the address of the instruction plus an offset of 12 will now either use an offset of 8 instead or will no longer be valid on ARM8. The following summarises their behaviour on ARM8:

- STR instructions with Rd = R15 store the address of the instruction plus 8
- STM instructions with R15 in the list of registers to be stored store the address of the instruction plus 8
- Data processing instructions with a register-specified shift and at least one of Rm and Rn equal to R15 are no longer valid
- MCR instructions with R15 as the source register are no longer valid

11.4 Write-back

Loading a register with write-back to it will have unpredictable effects.

The rules governing whether stores with write-back to the stored register store the register’s old or new value differ from those of ARM7. See ***4.11.6 Inclusion of the base in the register list*** on page 4-41 for further details.

11.5 Misaligned PC Loads and Stores

Misaligned loads or stores of the PC have unpredictable effects.

11.6 Data Aborts

In all cases where a data abort occurs, any base register is restored to its original value (before the instruction started), regardless of whether writeback is specified or not.

A Instruction Set Changes

This appendix gives an overview of changes to the instruction set when compared to ARM7.

Instruction Set Changes

A.1 General Compatibility

Existing code will run subject to all of the restrictions described in all ARM Datasheets up to ARM8, in addition to those in this data sheet.

As previous code does not have the IMB instructions, some code may not be compatible in certain circumstances.

For example:

- Code that constructs a routine in memory and then branches to it will be incompatible unless branch prediction has been turned off, or a calculated branch was used to get to it.
- Code that constructs a routine in memory, and then falls through to it sequentially will be incompatible unless the fall-through code instruction has been constructed at least 12 instructions in advance of its execution.

A.2 Instruction Set Differences

This section describes the instruction set additions and changes that have been made for ARM8.

A.2.1 New features

An Instruction Memory Barrier (IMB) instruction

This tells the ARM to flush any stored information about the instruction stream, and must be issued between modifying an instruction area and executing it.

Please refer to ***4.17 The Instruction Memory Barrier (IMB) Instruction*** on page 4-62 and ***Appendix D, Implementing the Instruction Memory Barrier Instruction*** for details.

Half-word and signed byte support

This has been added to the instruction set. Please refer to ***4.10 Halfword and Signed Data Transfer*** on page 4-33 for details.

A.2.2 Existing instructions

STM instructions with base writeback and the base register in the register list

This concerns the order of writeback and reading the value to be stored:

- If the base register is the lowest numbered register in the list, then the original base value is stored.
- Otherwise, the stored value is undefined at present.

LDM instructions with base writeback and the base register in the register list

This has no function, since the written-back register value is overwritten by the loaded value.

The behaviour is now architecturally undefined.

LDR instructions with writeback which load the base register

The behaviour is already architecturally undefined; see Application Note A002.

LDRB PC

The behaviour is already architecturally undefined; see Application Note A002.

LDR PC from a misaligned address

The behaviour is now architecturally undefined.

STRB PC

The behaviour is architecturally undefined; see Application Note A002.

STR PC to a misaligned address

The behaviour is now architecturally undefined.

STR PC

These were expected to store the address of the instruction plus 12, not the normal address of instruction plus 8.

These now store the address of instruction plus 8.

STM ...,PC}

These were expected to store the address of the instruction plus 12, not the normal address of instruction plus 8.

These now store the address of instruction plus 8.

Data processing instructions that do a register-controlled shift and have either or both of the main operand registers equal to the PC.

The behaviour is now architecturally undefined.

MCR instructions (coprocessor register transfers from ARM to coprocessor) with the PC as the source register.

The behaviour is now architecturally undefined.

B

26-bit Operations on ARM8

This appendix describes the 26-bit operations on ARM8.

B.1 Introduction

To maintain compatibility with earlier ARM processors, it is possible to execute code in 26-bit operating modes **usr26**, **fiq26**, **irq26** and **svc26**. Details of how to do this have already been written for earlier ARM processors, and these have been included here for your information.

This appendix summarises how 26-bit binary code will be able to run on the ARM8 processor. The details below show the instruction and performance differences when ARM8 is operated in 26-bit modes. The last section describes the hardware changes that affect 26-bit operation.

Use of 26-bit modes for any reason other than executing existing 26-bit code is strongly discouraged, as this will no longer be supported in ARM processors after the ARM8. It is also worth noting that ARM8's performance in 26-bit modes may be poorer than in 32-bit modes.

B.2 Instruction Differences

When ARM8 is executing in a 26-bit mode, the top 6 bits of the PC are forced to be zero at all times. The following restrictions must be obeyed to avoid problems due to the Prefetch Unit having prefetched an unknown distance beyond the current instruction:

- Do not enter any 26-bit mode when at an address outside the 26-bit address space.
- Do not execute code sequentially from address 0x03FFFFFC to address 0x00000000 in 26-bit code.

An additional requirement for 32-bit and 26-bit operations is that if a system contains code that is intended for execution in both 26-bit and 32-bit modes, an IMB instruction must accompany any change from any 26-bit mode to any 32-bit mode, and vice versa. It is therefore advisable to keep code intended for 26-bit modes and code intended for 32-bit modes completely separate.

26-bit operation removes some of the instruction constraints placed on 32-bit code. 26-bit code must obey the constraints laid out for 32-bit code with the following exceptions:

1. CMN, CMP, TEQ, TST

   A second form of these instructions becomes available, which is encoded in the instruction by setting the Rd field to “1111” and in the assembler syntax by using:

   ```
   <opcode>{cond}P
   ```

   in place of the normal

   ```
   <opcode>{cond}
   ```

   In all modes, the normal setting of the CPSR flags is suppressed for the new form of the instruction. Instead, the normal arithmetic result is calculated (Op1+Op2, Op1-Op2, Op1 EOR Op2 and Op1 AND Op2 for CMN, CMP, TEQ and TST respectively) and used to set selected CPSR bits. In user mode, the N, Z, C and V bits of the CPSR are set to bits 31 to 28 of the arithmetic result; in non-user modes, the N, Z, C, V, I, F, M1 and M0 bits of the CPSR are set to bits 31 to 26, 1 and 0 of the arithmetic result.

   The CMNP, CMPP, TEQP and TSTP instructions take a base of 3 cycles to execute, along with the extra cycles listed in ***4.5.8 Instruction cycle times*** on page 4-13 for complex and register-specified shifts.

2. Data processing instructions with destination register R15 and the S bit set

   These become valid in User mode, and their behaviour in all modes is altered.

   In all modes, the normal setting of the CPSR flags from the current mode’s SPSR is suppressed. In user mode, the N, Z, C and V bits of the CPSR are set to bits 31 to 28 of the arithmetic result; in non-user modes, the N, Z, C, V, I, F, M1 and M0 bits of the CPSR are set to bits 31 to 26, 1 and 0 of the arithmetic result.

3. LDM with R15 in Register list, and the S bit set

   This becomes valid in User mode, and its behaviour in all modes is altered.

In all modes, the normal setting of the CPSR flags from the current mode's SPSR is suppressed. In user mode, the N, Z, C and V bits of the CPSR are set to bits 31 to 28 of the value loaded for R15; in non-user modes, the N, Z, C, V, I, F, M1 and M0 bits of the CPSR are set to bits 31 to 26, 1 and 0 of the value loaded for R15.

4 Address Exceptions

The address exceptions which occur on true 26-bit ARM processors cannot occur on ARM8. If required, these should now be generated externally to the ARM8 as aborts, along with an abort handler routine which recognises the address exception.

Note: Some unusual coding cases may present problems: for example, LDMs and STMs wrapping around from the top of 26-bit memory space to the bottom. It is thought that such cases are not in common use, and so should not present any difficulties.

B.3 Performance Differences

This information is provisional at this release of the data sheet. Implementation details may affect performance.

There is no cycle count performance degradation for operating in 26-bit mode; the cycle counts are the same as those for 32-bit mode operations. However, there may be degradation due to the additional software overheads in getting to and from 32-bit-mode-only operations.

B.4 Hardware Compatibility Issues

This section describes the ways in which the ARM8 will differ from previous ARM processors, as far as its hardware is concerned, for 26-bit compatibility.

B.4.1 Pin-out

ARM8 will not have the two configuration pins, **DATA32** and **PROG32** as can be found on ARM6 and ARM7 processors.

As such, the processor's normal mode of operation is in full 32-bit modes: as if both of these pins were in their active HIGH state. Aborts on Read and Write of the Exception Vectors can be done by the Memory Manager, thus stimulating the original hardware configurations in software.

C

Comparing ARM6 and Earlier ARM Processors

This appendix describes the differences between the ARM6 series and earlier ARM processors. It is included here for completeness as it provides additional information on the differences between 26-bit and 32-bit codes to that described in ***Appendix B, 26-bit Operations on ARM8***. The information in the rest of the datasheet supersedes this appendix.

C.1 Introduction

The ARM6 series (ARM6, ARM60 and ARM600) is a family of ARM processors which have 32-bit program counters. Earlier ARMs (ARM2, ARM3 and ARM2aS) had a 26-bit program counter (PC). This appendix describes the major differences between the two types of processor.

C.2 The Program Counter and Program Status Register

The introduction of the larger program counter has meant that the flags and control bits of R15 (the combined PC and PSR) have been moved to a separate register. The extra space in the new register (the CPSR, Current Program Status Register) allows for more control bits. A further 3 mode bits have been added to allow for a larger number of operating modes.

The removal of the PSR to a separate register also means that it is no longer possible to save these flags automatically in R14 when a Branch with Link (BL) instruction is executed, or when an exception occurs. Program analysis has shown that the saving of these flags is only required in 3% of subroutine calls, so there is only a slight overhead in explicitly saving them on a stack when necessary. To cope with the requirement of saving them when an exception occurs, 5 further registers have been provided to hold a copy of the CPSR at the time of the exception. These registers are the Saved Program Status Registers (SPSRs). There is one SPSR for each of the modes that the processor may enter as a result of the various types of exception.

The expansion of the PC to 32 bits also means that the Branch instruction, being limited to +/-32 MB, can no longer specify a branch to the entire program space. Branches greater than +/-32 MB can be made with other instructions, but the equivalent of the Branch with Link instruction will require a separate instruction to save the PC in R14.

C.3 Operating Modes

There are a total of 10 operating modes in two overlapping sets. Four modes - **User26**, **IRQ26**, **FIQ26** and **Supervisor26** - allow the processor to behave like earlier ARM processors with a 26-bit PC. These correspond to the four operating modes of the ARM2 and ARM3 processors. A further four operating modes correspond to these, but with the processor running with the full 32-bit PC: these are **User32**, **IRQ32**, **FIQ32** and **Supervisor32**.

The final two modes are **Undefined32** and **Abort32**, and are entered when the Undefined instruction and Abort exceptions occur. They have been added to remove restrictions on Supervisor mode programs which exist with the ARM2 and ARM3 processors. The two sets of User, FIQ, IRQ and Supervisor modes each share a set of banked registers to allow them to maintain some private state at all times. The Abort and Undefined modes also have a pair of banked registers each for the same purpose.

C.4 Instruction Set Changes

The instruction set is changed in two major areas: new instructions have been introduced and restrictions have been placed on existing ones.

C.4.1 New instructions

The new instructions allow access to the CPSR and SPSR registers. They are formed by using opcodes from the Data Processing group of instructions that were previously unused. Specifically, these are the TST, TEQ, CMP and CMN instructions with the S flag clear. They are now known as MSR to move data into the CPSR and SPSR registers, and MRS to move from the CPSR and SPSR to a general register. The data moved to CPSR and SPSR can be either the contents of a general register or an immediate value.

C.4.2 Instruction set limitations

When configured for 32-bit program and data space, the ARM6 family supports operation in 26-bit modes for compatibility with ARM processors that have a 26-bit address space. The 26-bit modes are **User26**, **FIQ26**, **IRQ26** and **Supervisor26**. When a 26-bit mode is selected, the programmer’s model reverts to that of existing 26 bit ARMs (ARM2, ARM3, ARM2aS). The behaviour is that of the ARM2aS macrocell with the following alterations:

- Address exceptions are *never* generated. The OS may simulate the behaviour of address exception by using external logic such as a memory management unit to generate an abort if the 64 MB range is exceeded, and converting that abort into an “address exception” trap for the application.

Note: Address exceptions are still possible when the processor is configured for 26-bit program and data space.

- The new instructions to transfer data between general registers and the program status registers remain operative. The new instructions can be used by the operating system to return a 32-bit operating mode after calling a binary containing code written for a 26-bit ARM.
- All exceptions (including Undefined Instruction and Software Interrupt) return the processor to a 32-bit mode, so the operating system must be modified to handle them.
- The ARM6 family includes hardware which prevents the write operation and generates a data abort if the processor attempts to write to a location between &00000000 and &0000001F inclusive (the exception vectors) when operating in 26-bit mode. This allows the operating system to intercept all changes to the exception vectors and redirect the vector to some veneer code. The veneer code should place the processor in a 26-bit mode before calling the 26-bit exception handler.

In all other respects, the ARM6 family behaves like a 26-bit ARM when operating in 26 bit mode. The relevant bits of the CPSR appear to be incorporated back into R15 to form the PC/CPSR with the I and F bits in bits 27 and 26. The instruction set behaves like that of the ARM2aS macrocell with the addition of the MRS and MSR instructions.

C.5 Transferring between 26-bit and 32-bit Modes

A program executing in a privileged 32-bit mode can enter a 26-bit mode by executing an MSR instruction which alters the mode bits to one of the values shown below:

M[4:0]	Mode	Accessible register set
00000	usr26	PC/PSR, R14..R0, CPSR
00001	fiq26	PC/PSR, R14_fiq..R8_fiq, R7..R0, CPSR, SPSR_fiq
00010	irq26	PC/PSR, R14_irq..R13_fiq, R12..R0, CPSR, SPSR_irq
00011	svc26	PC/PSR, R14_svc..R13_svc, R12..R0, CPSR, SPSR_svc

Table 11-1: MSR instruction altering the mode bits

Transfer between 26-bit and 32-bit mode happens automatically whenever an exception occurs in 26-bit mode. Note that an exception (including Software Interrupt) arising in 26-bit mode will enter 32-bit mode and the saved value in R14 will contain only the PC, even though the PSR was also considered part of R15 when the exception arose.

In addition, the MSR instruction provides the means for a program in a privileged 26-bit mode to alter the mode bits to change to a 32-bit mode.

D

Implementing the Instruction Memory Barrier Instruction

This appendix is written to help Operating System designers understand and implement the IMB Instructions. It firstly describes the generic approach that should be used for future compatibilty and then goes on to ARM8-specific details.

D.1 Introduction

This appendix describes the processor specific code that must be included in the SWI handler to implement the two Instruction Memory Barrier (IMB) Instructions:

- IMB
- IMBRange

These are implemented as calls to specific SWI numbers. Please refer to ***4.17 The Instruction Memory Barrier (IMB) Instruction*** on page 4-62 for further details of this and for examples of use.

Two IMB instructions are provided so that when only a small area of code is altered before being executed the IMBRange instruction can be used to efficiently and quickly flush any stored instruction information from addresses within a small range rather than flushing all information about all instructions using the IMB instruction.
By flushing only the required address range information, the rest of the information remains to provide improved system performance.

D.2 ARM8 IMB Implementation

For ARM8, executing the SWI instruction is sufficient in itself to cause the IMB operation. Also, for ARM8, both the IMB and the IMBRange instructions flush *all* stored information about the instruction stream.

This means that for ARM8, all IMB instructions can be implemented in the Operating System by simply returning from the IMB/IMBRange service routine AND that the service routines can be exactly the same. The following service routine code can be used for ARM8:

```
IMB_SWI_handler
IMBRange_SWI_handler

            MOVS  PC, R14_svc; Return to the code after the SWI call
```

Note: It is strongly encouraged that in code from now on, the IMBRange instruction is used whenever the changed area of code is small: even if there is no distinction between it and the IMB instruction on ARM8. Future processors may well implement the IMBRange instruction in a much more efficient and faster manner, and code migrated from ARM8 will benefit when executed on these processors.

D.3 Generic IMB Use

Using SWI's to implement the IMB instructions means that any code that is written now will be compatible with any future processors - even if those processors implement IMB in different ways. This is achieved by changing the Operating System SWI service routines for each of the IMB SWI numbers that differ from processor to processor.

Below are examples that show what should happen during the execution of IMB instructions. These examples are taken from ***4.17.3 Examples*** on page 4-63.

The pseudo code in the square brackets shows what should happen to execute the IMB instruction (or IMBRange) in the SWI handler.

D.3.1 Loading code from disk

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

```
IMB EQU 0xF00000
    .
    .
    ; code that loads program from disk
    .
    .
    SWI IMB
       [branch to IMB service routine]
       [perform processor-specific operations to execute IMB]
       [return to code]
       .
    MOV PC, entry_point_of_loaded_program
       .
       .
```

D.3.2 Running BitBlt code

“Compiled BitBlt” routines optimise large copy operations by constructing and executing a copying loop which has been optimised for the exact operation wanted. When writing such a routine an IMB is needed between the code that constructs the loop and the actual execution of the constructed loop.

```
IMBRange EQU 0xF00001
   .
   .
   ; code that constructs loop code
   ; load R0 with the start address of the constructed loop
   ; load R1 with the end address of the constructed loop
   SWI   IMBRange
         [branch to IMBRange service routine]
         [read registers R0 and R1 to set up address range
               parameters]
         [perform processor-specific operations to execute
               IMBRange within address range]
         [return to code]
   ; start of loop code
   .
   .
```