Chapter A1
Introduction to the ARM Architecture

This chapter introduces the ARM architecture and contains the following sections:

1.1 About the ARM architecture

The ARM architecture has been designed to allow very small, yet high-performance implementations. The architectural simplicity of ARM processors leads to very small implementations, and small implementations allow devices with very low power consumption.

The ARM is a *Reduced Instruction Set Computer* (RISC), as it incorporates these typical RISC architecture features:

- a large uniform register file
- a *load/store* architecture, where data-processing operations only operate on register contents, not directly on memory contents
- simple addressing modes, with all load/store addresses being determined from register contents and instruction fields only
- uniform and fixed-length instruction fields, to simplify instruction decode.

In addition, the ARM architecture gives you:

- control over both the *Arithmetic Logic Unit* (ALU) and shifter in every data-processing instruction to maximize the use of an ALU and a shifter
- auto-increment and auto-decrement addressing modes to optimize program loops
- Load and Store Multiple instructions to maximize data throughput
- conditional execution of all instructions to maximize execution throughput.

These enhancements to a basic RISC architecture allow ARM processors to achieve a good balance of high performance, low code size, low power consumption and low silicon area.

1.1.1 ARM registers

ARM has 31 general-purpose 32-bit registers. At any one time, 16 of these registers are visible. The other registers are used to speed up exception processing. All the register specifiers in ARM instructions can address any of the 16 visible registers.

The main bank of 16 registers is used by all unprivileged code. These are the User mode registers. User mode is different from all other modes as it is unprivileged, which means:

- User mode is the only mode which cannot switch to another processor mode without generating an exception
- memory systems and coprocessors might allow User mode less access to memory and coprocessor functionality than a privileged mode.

Two of the 16 visible registers have special roles:

Link register	Register 14 is the *Link Register* (LR). This register holds the address of the next instruction after a Branch and Link (`BL`) instruction, which is the instruction used to make a subroutine call. At all other times, R14 can be used as a general-purpose register.
Program counter	Register 15 is the *Program Counter* (PC). It can be used in most instructions as a pointer to the instruction which is two instructions after the instruction being executed. All ARM instructions are four bytes long (one 32-bit word) and are always aligned on a word boundary. This means that the bottom two bits of the PC are always zero, and therefore the PC contains only 30 non-constant bits.

The remaining 14 registers have no special hardware purpose. Their uses are defined purely by software. Software normally uses R13 as a *Stack Pointer* (SP).

For more details on registers, please refer to *Registers* on page A2-4.

1.1.2 Exceptions

ARM supports five types of exception, and a privileged processing mode for each type. The five types of exceptions are:

- fast interrupt
- normal interrupt
- memory aborts, which can be used to implement memory protection or virtual memory
- attempted execution of an undefined instruction
- software interrupt (`SWI`) instructions which can be used to make a call to an operating system.

When an exception occurs, some of the standard registers are replaced with registers specific to the exception mode. All exception modes have replacement *banked* registers for R13 and R14. The fast interrupt mode has more registers for fast interrupt processing.

When an exception handler is entered, R14 holds the return address for exception processing. This is used to return after the exception is processed and to address the instruction that caused the exception.

Register 13 is banked across exception modes to provide each exception handler with a private stack pointer. The fast interrupt mode also banks registers 8 to 12 so that interrupt processing can begin without the need to save or restore these registers.

There is a sixth privileged processing mode, System mode, which uses the User mode registers. This is used to run tasks that require privileged access to memory and/or coprocessors, without limitations on which exceptions can occur during the task.

For more details on exceptions, please refer to *Exceptions* on page A2-13.

The exception process

When an exception occurs, the ARM processor halts execution after the current instruction and begins execution at one of a number of fixed addresses in memory, known as the *exception vectors*. There is a separate vector location for each exception.

An operating system installs a handler on every exception at initialization. Privileged operating system tasks are normally run in System mode to allow exceptions to occur within the operating system without state loss.

1.1.3 Status registers

All processor state other than the general-purpose register contents is held in *status registers*. The current operating processor status is in the *Current Program Status Register* (CPSR). The CPSR holds:

- 4 condition code flags (Negative, Zero, Carry and oVerflow)
- 2 interrupt disable bits, one for each type of interrupt
- 5 bits which encode the current processor mode
- 1 bit which encodes whether ARM or Thumb instructions are being executed.

Each exception mode also has a *Saved Program Status Register* (SPSR) which holds the CPSR of the task immediately before the exception occurred. The CPSR and the SPSRs are accessed with special instructions.

For more details on status registers, please refer to *Program status registers* on page A2-9.

1.2 ARM instruction set

The ARM instruction set can be divided into six broad classes of instruction:

- *Branch instructions*
- *Data-processing instructions* on page A1-6
- *Status register transfer instructions* on page A1-7
- *Load and store instructions* on page A1-7
- *Coprocessor instructions* on page A1-8
- *Exception-generating instructions* on page A1-9.

Most data-processing instructions and one type of coprocessor instruction can update the four condition code flags in the CPSR (Negative, Zero, Carry and oVerflow) according to their result.

Almost all ARM instructions contain a 4-bit *condition* field. One value of this field specifies that the instruction is executed unconditionally.

Fourteen other values specify *conditional execution* of the instruction. If the condition code flags indicate that the corresponding condition is true when the instruction starts executing, it executes normally. Otherwise, the instruction does nothing. The 14 available conditions allow:

- tests for equality and non-equality
- tests for <, <=, >, and >= inequalities, in both signed and unsigned arithmetic
- each condition code flag to be tested individually.

The sixteenth value of the condition field is used for a few instructions which do not allow conditional execution.

1.2.1 Branch instructions

As well as allowing many data-processing or load instructions to change control flow by writing the PC, a standard Branch instruction is provided with a 24-bit signed offset, allowing forward and backward branches of up to 32MB.

There is a Branch and Link (`BL`) option that also preserves the address of the instruction after the branch in R14, the LR. This provides a subroutine call which can be returned from by copying the LR into the PC.

There are also branch instructions which can switch instruction set, so that execution continues at the branch target using the Thumb instruction set. These allow ARM code to call Thumb subroutines, and ARM subroutines to return to a Thumb caller. Similar instructions in the Thumb instruction set allow the corresponding Thumb → ARM switches. An overview of the Thumb instruction set is provided in Chapter A6 *The Thumb Instruction Set*.

1.2.2 Data-processing instructions

The data-processing instructions perform calculations on the general-purpose registers. There are four types of data-processing instructions:

- *Arithmetic/logic instructions*
- *Comparison instructions*
- *Multiply instructions*
- *Count Leading Zeros instruction* on page A1-7.

Arithmetic/logic instructions

There are twelve arithmetic/logic instructions which share a common instruction format. These perform an arithmetic or logical operation on up to two source operands, and write the result to a destination register. They can also optionally update the condition code flags based on the result.

Of the two source operands:

- one is always a register
- the other has two basic forms:
  - an immediate value
  - a register value, optionally shifted.

If the operand is a shifted register, the shift amount can be either an immediate value or the value of another register. Four types of shift can be specified. Every arithmetic/logic instruction can therefore perform an arithmetic/logic and a shift operation. As a result, ARM does not have dedicated shift instructions.

Because the *Program Counter* (PC) is a general-purpose register, arithmetic/logic instructions can write their results directly to the PC. This allows easy implementation of a variety of jump instructions.

Comparison instructions

There are four comparison instructions which use the same instruction format as the arithmetic/logic instructions. These perform an arithmetic or logical operation on two source operands, but do not write the result to a register. They always update the condition flags based on the result.

The source operands of comparison instructions take the same forms as those of arithmetic/logic instructions, including the ability to incorporate a shift operation.

Multiply instructions

Multiply instructions come in two classes. Both types multiply two 32-bit register values and store their result:

32-bit result	Normal. Stores the 32-bit result in a register.
64-bit result	Long. Stores the 64-bit result in two separate registers.

Both types of multiply instruction can optionally perform an accumulate operation.

Count Leading Zeros instruction

The Count Leading Zeros (`CLZ`) instruction determines the number of zero bits at the most significant end of a register value, up to the first 1 bit. This number is written to the destination register of the `CLZ` instruction.

1.2.3 Status register transfer instructions

The status register transfer instructions transfer the contents of the CPSR or an SPSR to or from a general-purpose register. Writing to the CPSR can:

- set the values of the condition code flags
- set the values of the interrupt enable bits
- set the processor mode.

1.2.4 Load and store instructions

The following load and store instructions are available:

- *Load and Store Register*
- *Load and Store Multiple registers* on page A1-8
- *Swap register and memory contents* on page A1-8.

Load and Store Register

Load Register instructions can load a 32-bit word, a 16-bit halfword or an 8-bit byte from memory into a register. Byte and halfword loads can be automatically zero-extended or sign-extended as they are loaded.

Store Register instructions can store a 32-bit word, a 16-bit halfword or an 8-bit byte from a register to memory.

Load and Store Register instructions have three primary addressing modes, all of which use a *base register* and an *offset* specified by the instruction:

- In *offset addressing*, the memory address is formed by adding or subtracting an offset to or from the base register value.
- In *pre-indexed addressing*, the memory address is formed in the same way as for offset addressing. As a side-effect, the memory address is also written back to the base register.
- In *post-indexed addressing*, the memory address is the base register value. As a side-effect, an offset is added to or subtracted from the base register value and the result is written back to the base register.

In each case, the offset can be either an immediate or the value of an *index register*. Register-based offsets can also be scaled with shift operations.

As the PC is a general-purpose register, a 32-bit value can be loaded directly into the PC to perform a jump to any address in the 4GB memory space.

Load and Store Multiple registers

Load Multiple (LDM) and Store Multiple (STM) instructions perform a block transfer of any number of the general-purpose registers to or from memory. Four addressing modes are provided:

- pre-increment
- post-increment
- pre-decrement
- post-decrement.

The base address is specified by a register value, which can be optionally updated after the transfer. As the subroutine return address and PC values are in general-purpose registers, very efficient subroutine entry and exit sequences can be constructed with LDM and STM:

- A single STM instruction at subroutine entry can push register contents and the return address onto the stack, updating the stack pointer in the process.
- A single LDM instruction at subroutine exit can restore register contents from the stack, load the PC with the return address, and update the stack pointer.

LDM and STM instructions also allow very efficient code for block copies and similar data movement algorithms.

Swap register and memory contents

A swap (SWP) instruction performs the following sequence of operations:

1. It loads a value from a register-specified memory location.
2. It stores the contents of a register to the same memory location.
3. It writes the value loaded in step 1 to a register.

By specifying the same register for steps 2 and 3, the contents of a memory location and a register are interchanged.

The swap operation performs a special indivisible bus operation that allows atomic update of semaphores. Both 32-bit word and 8-bit byte semaphores are supported.

1.2.5 Coprocessor instructions

There are three types of coprocessor instructions:

Data-processing instructions

These start a coprocessor-specific internal operation.

Data transfer instructions

These transfer coprocessor data to or from memory. The address of the transfer is calculated by the ARM processor.

Register transfer instructions

These allow a coprocessor value to be transferred to or from an ARM register.

1.2.6 Exception-generating instructions

Two types of instruction are designed to cause specific exceptions to occur.

Software interrupt instructions

`SWI` instructions cause a software interrupt exception to occur. These are normally used to make calls to an operating system, to request an OS-defined service. The exception entry caused by a `SWI` instruction also changes to a privileged processor mode. This allows an unprivileged task to gain access to privileged functions, but only in ways permitted by the OS.

Software breakpoint instructions

`BKPT` instructions cause an abort exception to occur. If suitable debugger software is installed on the abort vector, an abort exception generated in this fashion is treated as a breakpoint. If debug hardware is present in the system, it can instead treat a `BKPT` instruction directly as a breakpoint, preventing the abort exception from occurring.

In addition to the above, the following types of instruction cause an Undefined Instruction exception to occur:

- coprocessor instructions which are not recognized by any hardware coprocessor
- most instruction words that have not yet been allocated a meaning as an ARM instruction.

In each case, this exception is normally used either to generate a suitable error or to initiate software emulation of the instruction.

Chapter A2
Programmer's Model

This chapter introduces the ARM programmer's model. It contains the following sections:

2.1 Data types

ARM processors support the following data types:

Byte	8 bits.
Halfword	16 bits (halfwords must be aligned to two-byte boundaries).
Word	32 bits (words must be aligned to four-byte boundaries).

Note

- All three types are supported in ARM architecture version 4 and above. Only bytes and words were supported prior to ARM architecture version 4.
- When any of these types is described as *unsigned*, the N-bit data value represents a non-negative integer in the range 0 to $+2^{N}-1$, using normal binary format.
- When any of these types is described as *signed*, the N-bit data value represents an integer in the range -2^{N-1} to $+2^{N-1}-1$, using two's complement format.
- All data operations, for example `ADD`, are performed on word quantities.
- Load and store operations can transfer bytes, halfwords and words to and from memory, automatically zero-extending or sign-extending bytes or halfwords as they are loaded.
- ARM instructions are exactly one word (and are aligned on a four-byte boundary). Thumb instructions are exactly one halfword (and are aligned on a two-byte boundary).

2.2 Processor modes

The ARM architecture supports the seven processor modes shown in Table 2-1.

Table 2-1 ARM version 4 processor modes

Processor mode		Description
User	usr	Normal program execution mode
FIQ	fiq	Supports a high-speed data transfer or channel process
IRQ	irq	Used for general-purpose interrupt handling
Supervisor	svc	A protected mode for the operating system
Abort	abt	Implements virtual memory and/or memory protection
Undefined	und	Supports software emulation of hardware coprocessors
System	sys	Runs privileged operating system tasks (ARM architecture version 4 and above)

Mode changes can be made under software control, or can be caused by external interrupts or exception processing.

Most application programs execute in User mode. While the processor is in User mode, the program being executed is unable to access some protected system resources or to change mode, other than by causing an exception to occur (see *Exceptions* on page A2-13). This allows a suitably written operating system to control the use of system resources.

The modes other than User mode are known as *privileged modes*. They have full access to system resources and can change mode freely. Five of them are known as *exception modes*:

- FIQ
- IRQ
- Supervisor
- Abort
- Undefined.

These are entered when specific exceptions occur. Each of them has some additional registers to avoid corrupting User mode state when the exception occurs (see *Registers* on page A2-4 for details).

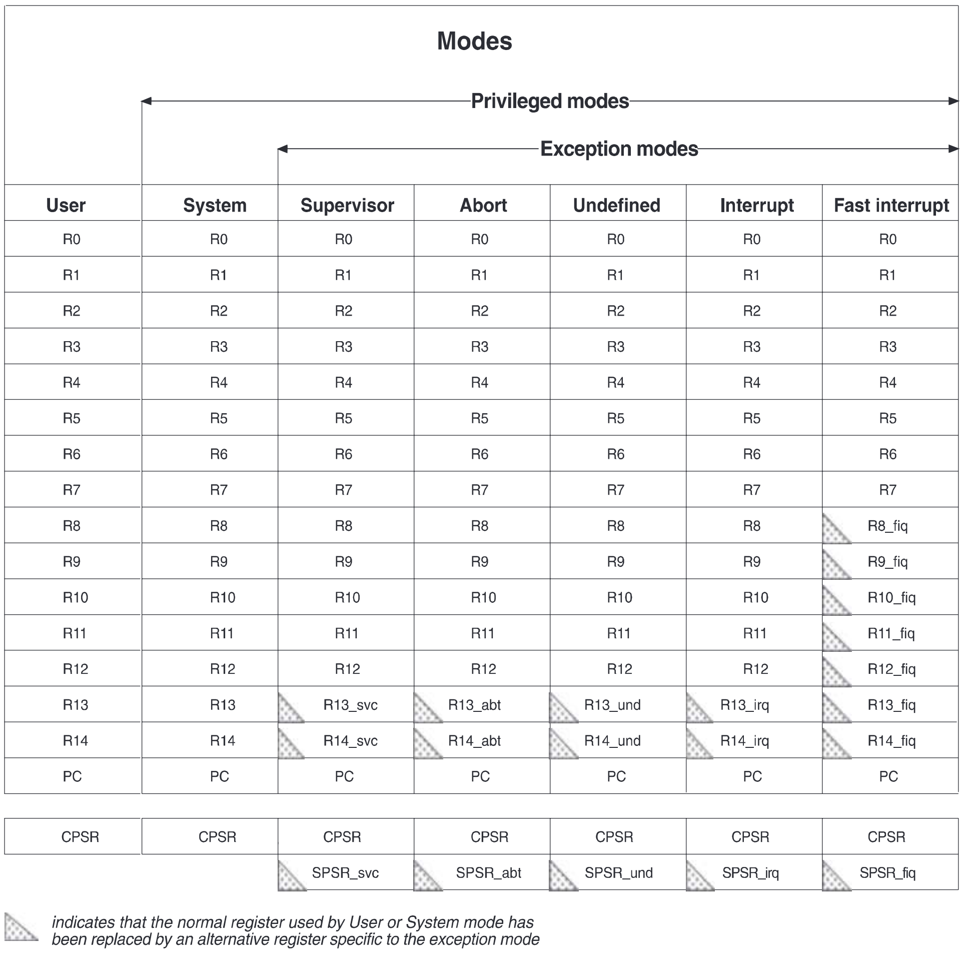
The remaining mode is System mode, and is only present in ARM architecture version 4 and above. It is not entered by any exception and has exactly the same registers available as User mode. However, it is a privileged mode and is therefore not subject to the User mode restrictions. It is intended for use by operating system tasks which need access to system resources, but wish to avoid using the additional registers associated with the exception modes. Avoiding such use ensures that the task state is not corrupted by the occurrence of any exception.

2.3 Registers

The ARM processor has a total of 37 registers:

- 31 general-purpose registers, including a program counter. These registers are 32 bits wide and are described in *General-purpose registers* on page A2-5.
- 6 status registers. These registers are also 32 bits wide, but only 12 of the 32 bits are allocated or need to be implemented. These are described in *Program status registers* on page A2-9.

Registers are arranged in partially overlapping banks, with a different register bank for each processor mode, as shown in Figure 2-1. At any time, 15 general-purpose registers (R0 to R14), one or two status registers and the program counter are visible. Each column of Figure 2-1 shows which general-purpose and status registers are visible in the indicated processor mode.



Modes

Privileged modes: System to Fast interrupt; Exception modes: Supervisor to Fast interrupt

User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	▨ R8_fiq
R9	R9	R9	R9	R9	R9	▨ R9_fiq
R10	R10	R10	R10	R10	R10	▨ R10_fiq
R11	R11	R11	R11	R11	R11	▨ R11_fiq
R12	R12	R12	R12	R12	R12	▨ R12_fiq
R13	R13	▨ R13_svc	▨ R13_abt	▨ R13_und	▨ R13_irq	▨ R13_fiq
R14	R14	▨ R14_svc	▨ R14_abt	▨ R14_und	▨ R14_irq	▨ R14_fiq
PC	PC	PC	PC	PC	PC	PC

User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		▨ SPSR_svc	▨ SPSR_abt	▨ SPSR_und	▨ SPSR_irq	▨ SPSR_fiq

▨ *indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode*

Figure 2-1 Register organization

2.4 General-purpose registers

The general-purpose registers R0-R15 can be split into three groups. These groups differ in the way they are banked and in their special-purpose uses:

- *The unbanked registers, R0-R7*
- *The banked registers, R8-R14*
- Register 15, the PC, is described in *The program counter, R15* on page A2-7.

2.4.1 The unbanked registers, R0-R7

Registers R0 to R7 are *unbanked registers*. This means that each of them refers to the same 32-bit physical register in all processor modes. They are completely general-purpose registers, with no special uses implied by the architecture, and can be used wherever an instruction allows a general-purpose register to be specified.

2.4.2 The banked registers, R8-R14

Registers R8 to R14 are *banked registers*. The physical register referred to by each of them depends on the current processor mode. Where a particular physical register is intended, without depending on the current processor mode, a more specific name (as described below) is used. Almost all instructions allow the banked registers to be used wherever a general-purpose register is allowed.

Note

A few exceptions to this rule are noted in the individual instruction descriptions. Where a restriction exists on the use of banked registers, it always applies to all of R8 to R14. For example, R8 to R12 are subject to such restrictions even in systems in which FIQ mode is never used and so only one physical version of the register is ever in use.

Registers R8 to R12 have two banked physical registers each. One is used in all processor modes other than FIQ mode, and the other is used in FIQ mode. Where it is necessary to be specific about which version is being referred to, the first group of physical registers are referred to as R8_usr to R12_usr and the second group as R8_fiq to R12_fiq.

Registers R8 to R12 do not have any dedicated special purposes in the architecture. However, for interrupts that are simple enough to be processed using registers R8 to R14 only, the existence of separate FIQ mode versions of these registers allows very fast interrupt processing. Examples of this usage can be found in *Single-channel DMA transfer* on page A9-13 and *Dual-channel DMA transfer* on page A9-13.

Registers R13 and R14 have six banked physical registers each. One is used in User and System modes, while each of the remaining five is used in one of the five exception modes. Where it is necessary to be specific about which version is being referred to, you use names of the form:

```
R13_<mode>
R14_<mode>
```

where `<mode>` is the appropriate one of `usr`, `svc` (for Supervisor mode), `abt`, `und`, `irq` and `fiq`.

Register R13 is normally used as a stack pointer and is also known as the SP. In the ARM instruction set, this is by convention only, as there are no defined instructions or other functionality which use R13 in a special-case manner. However, there are such instructions in the Thumb instruction set, as described in Chapter A6 *The Thumb Instruction Set*.

Each exception mode has its own banked version of R13, which should normally be initialized to point to a stack dedicated to that exception mode. On entry, the exception handler typically stores to this stack the values of other registers to be used. By reloading these values into the registers when it returns, the exception handler can ensure that it does not corrupt the state of the program that was being executed when the exception occurred.

Register R14 (also known as the *Link Register* or LR) has two special functions in the architecture:

- In each mode, the mode's own version of R14 is used to hold subroutine return addresses. When a subroutine call is performed by a `BL` or `BLX` instruction, R14 is set to the subroutine return address. The subroutine return is performed by copying R14 back to the program counter. This is typically done in one of the two following ways:
  - Execute either of these instructions:
    ```
    MOV PC,LR
    BX LR
    ```
  - On subroutine entry, store R14 to the stack with an instruction of the form:
    ```
    STMFD SP!,{<registers>,LR}
    ```
    and use a matching instruction to return:
    ```
    LDMFD SP!,{<registers>,PC}
    ```
- When an exception occurs, the appropriate exception mode's version of R14 is set to the exception return address (offset by a small constant for some exceptions). The exception return is performed in a similar way to a subroutine return, but using slightly different instructions to ensure full restoration of the state of the program that was being executed when the exception occurred. See *Exceptions* on page A2-13 for more details.

Register R14 can be treated as a general-purpose register at all other times.

Note

When nested exceptions are possible, the two special-purpose uses might conflict. For example, if an IRQ interrupt occurs when a program is being executed in User mode, none of the User mode registers are necessarily corrupted. But if an interrupt handler running in IRQ mode re-enables IRQ interrupts and a nested IRQ interrupt occurs, any value the outer interrupt handler is holding in R14_irq at the time is overwritten by the return address of the nested interrupt.

System programmers need to be careful about such interactions. The usual way to deal with them is to ensure that the appropriate version of R14 does not hold anything significant at times that nested exceptions can occur. When this is hard to do in a straightforward way, it is usually best to change to another processor mode during entry to the exception handler, before re-enabling interrupts or otherwise allowing nested exceptions to occur. (In ARM architecture version 4 and above, System mode is usually the best mode to use for this purpose.)

2.4.3 The program counter, R15

Register R15 holds the *Program Counter* (PC). It can often be used in place of one of the general-purpose registers R0 to R14, and is therefore considered one of the general-purpose registers. However, there are also many instruction-specific restrictions or special cases about its use. These are noted in the individual instruction descriptions. Usually, the instruction is UNPREDICTABLE if R15 is used in a manner that breaks these restrictions.

The Program Counter is always used for a special purpose, as described in:

- *Reading the program counter*
- *Writing the program counter* on page A2-8.

Reading the program counter

When an instruction reads R15 without breaking any of the restrictions on its use, the value read is the address of the instruction plus 8 bytes. As ARM instructions are always word-aligned, bits[1:0] of the resulting value are always zero. (In T variants of the architecture, this behavior changes during Thumb state execution - see Chapter A6 *The Thumb Instruction Set* for details.)

This way of reading the PC is primarily used for quick, position-independent addressing of nearby instructions and data, including position-independent branching within a program.

An exception to the above rule occurs when an `STR` or `STM` instruction stores R15. Such instructions can store either the address of the instruction plus 8 bytes, like other instructions that read R15, or the instruction's own address plus 12 bytes. Whether the offset of 8 or the offset of 12 is used is IMPLEMENTATION DEFINED. An implementation must use the same offset for all `STR` and `STM` instructions that store R15. It cannot use 8 for some of them and 12 for others.

Because of this exception, it is usually best to avoid the use of `STR` and `STM` instructions that store R15. If this is difficult, use a suitable instruction sequence in the program to ascertain which offset the implementation uses. For example, if R0 points to an available word of memory, then the following instructions put the offset of the implementation in R0:

```
SUB  R1, PC, #4    ; R1 = address of following STR instruction
STR  PC, [R0]      ; Store address of STR instruction + offset,
LDR  R0, [R0]      ; then reload it
SUB  R0, R0, R1    ; Calculate the offset as the difference
```

Note

The rules about how R15 is read apply only to reads by instructions. In particular, they do not necessarily describe the values placed on a hardware address bus during instruction fetches. Like all other details of hardware interfaces, such values are IMPLEMENTATION DEFINED.

Writing the program counter

When an instruction writes R15 without breaking any of the restrictions on its use, the normal result is that the value written to R15 is treated as an instruction address and a branch occurs to that address.

Since ARM instructions are required to be word-aligned, values written to R15 are normally expected to have bits[1:0] == 0b00. The precise rules for this depend on the architecture version:

- In ARM architecture versions 3 and below, bits[1:0] of a value written to R15 are ignored, so that the actual destination address of the instruction is (value written to R15) AND `0xFFFFFFFC`.
- In ARM architecture versions 4 and above, bits[1:0] of a value written to R15 in ARM state must be 0b00. If they are not, the results are UNPREDICTABLE.

Similarly, in T variants of ARM architecture versions 4 and above, Thumb instructions are required to be halfword-aligned. Bit[0] of a value written to R15 in Thumb state is ignored, so that the actual destination address of the instruction is (value written to R15) AND `0xFFFFFFFE`.

Several instructions have their own rules for interpreting values written to R15. For example, `BX` and other instructions designed to transfer between ARM and Thumb states use bit[0] of the value to select whether to execute the code at the destination address in ARM state or Thumb state. Special rules of this type are described on the individual instruction pages, and override the general rules in this section.

2.5 Program status registers

The *current program status register* (CPSR) is accessible in all processor modes. It contains condition code flags, interrupt disable bits, the current processor mode, and other status and control information. Each exception mode also has a *saved program status register* (SPSR), that is used to preserve the value of the CPSR when the associated exception occurs.

Note

User mode and System mode do not have an SPSR, because they are not exception modes. All instructions which read or write the SPSR are UNPREDICTABLE when executed in User mode or System mode.

The format of the CPSR and the SPSRs is shown below.

31	30	29	28	27	26 8	7	6	5	4	3	2	1	0
N	Z	C	V	Q	DNM(RAZ)	I	F	T	M 4	M 3	M 2	M 1	M 0

2.5.1 The condition code flags

The N, Z, C, and V (Negative, Zero, Carry and oVerflow) bits are collectively known as the *condition code flags*, often referred to as *flags*. The condition code flags in the CPSR can be tested by most instructions to determine whether the instruction is to be executed.

The condition code flags are usually modified by:

- Execution of a comparison instruction (`CMN`, `CMP`, `TEQ` or `TST`).
- Execution of some other arithmetic, logical or move instruction, where the destination register of the instruction is not R15. Most of these instructions have both a flag-preserving and a flag-setting variant, with the latter being selected by adding an `S` qualifier to the instruction mnemonic. Some of these instructions only have a flag-preserving version. This is noted in the individual instruction descriptions.

In either case, the new condition code flags (after the instruction has been executed) usually mean:

N Is set to bit 31 of the result of the instruction. If this result is regarded as a two's complement signed integer, then N = 1 if the result is negative and N = 0 if it is positive or zero.

Z Is set to 1 if the result of the instruction is zero (which often indicates an *equal* result from a comparison), and to 0 otherwise.

C Is set in one of four ways:

- For an addition, including the comparison instruction `CMN`, C is set to 1 if the addition produced a carry (that is, an unsigned overflow), and to 0 otherwise.
- For a subtraction, including the comparison instruction `CMP`, C is set to 0 if the subtraction produced a borrow (that is, an unsigned underflow), and to 1 otherwise.

- For non-addition/subtractions that incorporate a shift operation, C is set to the last bit shifted out of the value by the shifter.
- For other non-addition/subtractions, C is normally left unchanged (but see the individual instruction descriptions for any special cases).

V Is set in one of two ways:

- For an addition or subtraction, V is set to 1 if signed overflow occurred, regarding the operands and result as two's complement signed integers.
- For non-addition/subtractions, V is normally left unchanged (but see the individual instruction descriptions for any special cases).

The flags can be modified in these additional ways:

- Execution of an `MSR` instruction, as part of its function of writing a new value to the CPSR or SPSR.
- Execution of `MRC` instructions with destination register R15. The purpose of such instructions is to transfer coprocessor-generated condition code flag values to the ARM processor.
- Execution of some variants of the `LDM` instruction. These variants copy the SPSR to the CPSR, and their main intended use is for returning from exceptions.
- Execution of flag-setting variants of arithmetic and logical instructions whose destination register is R15. These also copy the SPSR to the CPSR, and are mainly intended for returning from exceptions.

The Q flag

In E variants of ARM architecture 5 and above, bit[27] of the CPSR is known as the Q flag and is used to indicate whether overflow and/or saturation has occurred in some of the enhanced DSP instructions. Similarly, bit[27] of each SPSR is a Q flag, and is used to preserve and restore the CPSR Q flag if an exception occurs. For more details of the Q flag, see Chapter A10 *Enhanced DSP Extension*.

In architecture versions prior to version 5, and in non-E variants of architecture version 5 and above, bit[27] of the CPSR and SPSRs should be treated as described in *Other bits* on page A2-12.

2.5.2 The control bits

The bottom eight bits of a *Program Status Register* (PSR), incorporating I, F, T and M[4:0], are known collectively as the *control bits*. The control bits change when an exception arises and can be altered by software only when the processor is in a privileged mode.

Interrupt disable bits

I and F are the interrupt disable bits:

I bit Disables IRQ interrupts when it is set.

F bit Disables FIQ interrupts when it is set.

The T bit

The T bit *should be zero* (SBZ) on ARM architecture version 3 and below, and on non-T variants of ARM architecture version 4. No instructions exist in these architectures that can switch between ARM and Thumb states.

On T variants of ARM architecture 4 and above, the T bit has the following meanings:

T = 0	Indicates ARM execution.
T = 1	Indicates Thumb execution.

Instructions that switch between ARM and Thumb states can be used freely on implementations of these architectures.

On non-T variants of ARM architecture version 5 and above, the T bit has the following meanings:

T = 0	Indicates ARM execution.
T = 1	Forces the next instruction executed to cause an undefined instruction exception (see *Undefined Instruction exception* on page A2-15).

Instructions that switch between ARM and Thumb states can be used on implementations of these architectures, but only function correctly as long as the program remains in ARM state. If the program attempts to switch to Thumb state, the first instruction executed after the attempted switch causes an undefined instruction exception. Entry into that exception then switches back to ARM state. The exception handler can detect that this was the cause of the exception from the fact that the T bit of SPSR_und is set.

Mode bits

M0, M1, M2, M3, and M4 (M[4:0]) are the mode bits, and these determine the mode in which the processor operates. Their interpretation is shown in Table 2-2.

Table 2-2 The mode bits

M[4:0]	Mode	Accessible registers
0b10000	User	PC, R14 to R0, CPSR
0b10001	FIQ	PC, R14_fiq to R8_fiq, R7 to R0, CPSR, SPSR_fiq
0b10010	IRQ	PC, R14_irq, R13_irq, R12 to R0, CPSR, SPSR_irq
0b10011	Supervisor	PC, R14_svc, R13_svc, R12 to R0, CPSR, SPSR_svc
0b10111	Abort	PC, R14_abt, R13_abt, R12 to R0, CPSR, SPSR_abt
0b11011	Undefined	PC, R14_und, R13_und, R12 to R0, CPSR, SPSR_und
0b11111	System	PC, R14 to R0, CPSR (ARM architecture v4 and above)

Not all combinations of the mode bits define a valid processor mode. Only those combinations explicitly described can be used. If any other value is programmed into the mode bits M[4:0], the result is UNPREDICTABLE. See also Table 8-1 on page A8-9 for details of the mode bits in the 26-bit architectures.

2.5.3 Other bits

Other bits in the program status registers are reserved for future expansion. In general, programmers must take care to write code in such a way that these bits are never modified. Failure to do this might result in code which has unexpected side-effects on future versions of the architecture. See the usage notes for the `MSR` instruction on page A4-62 for more details.

2.6 Exceptions

Exceptions are generated by internal and external sources to cause the processor to handle an event, such as an externally generated interrupt or an attempt to execute an undefined instruction. The processor state just before handling the exception must be preserved so that the original program can be resumed when the exception routine has completed. More than one exception can arise at the same time.

ARM supports seven types of exception. Table 2-3 lists the types of exception and the processor mode that is used to process that exception. When an exception occurs, execution is forced from a fixed memory address corresponding to the type of exception. These fixed addresses are called the *exception vectors*.

Note

The normal vector at address `0x00000014` and the high vector at address `0xFFFF0014` are not normally used and are reserved for future expansion. The reserved vector at address `0x00000014` was used for an Address Exception vector in earlier versions of the ARM architecture which had a 26-bit address space. See Chapter A8 *The 26-bit Architectures* for more information.

Table 2-3 Exception processing modes

Exception type	Mode	Normal address	High vector address
Reset	Supervisor	`0x00000000`	`0xFFFF0000`
Undefined instructions	Undefined	`0x00000004`	`0xFFFF0004`
Software interrupt (SWI)	Supervisor	`0x00000008`	`0xFFFF0008`
Prefetch Abort (instruction fetch memory abort)	Abort	`0x0000000C`	`0xFFFF000C`
Data Abort (data access memory abort)	Abort	`0x00000010`	`0xFFFF0010`
IRQ (interrupt)	IRQ	`0x00000018`	`0xFFFF0018`
FIQ (fast interrupt)	FIQ	`0x0000001C`	`0xFFFF001C`

When an exception occurs, the banked versions of R14 and the SPSR for the exception mode are used to save state as follows:

```
R14_<exception_mode> = return link
SPSR_<exception_mode> = CPSR
CPSR[4:0] = exception mode number
CPSR[5] = 0                          /* Execute in ARM state */
if <exception_mode> == Reset or FIQ then
    CPSR[6] = 1                      /* Disable fast interrupts */
/* else CPSR[6] is unchanged */
CPSR[7] = 1                          /* Disable normal interrupts */
PC = exception vector address
```

To return after handling the exception, the SPSR is moved into the CPSR, and R14 is moved to the PC. This can be done atomically in two ways:

- using a data-processing instruction with the S bit set, and the PC as the destination
- using the Load Multiple with Restore CPSR instruction, as described in *LDM (3)* on page A4-34.

The following sections show what happens automatically when the exception occurs, and also show the recommended data-processing instruction to use to return from each exception. This instruction is always a `MOVS` or `SUBS` instruction with the PC as its destination.

Note

When the recommended data-processing instruction is a `SUBS` and a Load Multiple with Restore CPSR instruction is used to return from the exception handler, the subtraction must still be performed. This is usually done at the start of the exception handler, before the return link is stored to memory.

For example, an interrupt handler that wishes to store its return link on the stack might use instructions of the following form at its entry point:

```
    SUB     R14, R14, #4
    STMFD   SP!, {<other_registers>, R14}
```

and return using the instruction:

```
    LDMFD   SP!, {<other_registers>, PC}^
```

2.6.1 Reset

When the Reset input is asserted on the processor, the ARM processor immediately stops execution of the current instruction. When Reset is de-asserted, the following actions are performed:

```
R14_svc   = UNPREDICTABLE value
SPSR_svc  = UNPREDICTABLE value
CPSR[4:0] = 0b10011             /* Enter Supervisor mode */
CPSR[5]   = 0                   /* Execute in ARM state */
CPSR[6]   = 1                   /* Disable fast interrupts */
CPSR[7]   = 1                   /* Disable normal interrupts */
if high vectors configured then
    PC    = 0xFFFF0000
else
    PC    = 0x00000000
```

After Reset, the ARM processor begins execution at address `0x00000000` or `0xFFFF0000` in Supervisor mode with interrupts disabled. See *About the MMU architecture* on page B3-2 for more information on the effects of Reset.

Note

There is no architecturally defined way of returning from a Reset.

2.6.2 Undefined Instruction exception

If the ARM processor executes a coprocessor instruction, it waits for any external coprocessor to acknowledge that it can execute the instruction. If no coprocessor responds, an Undefined Instruction exception occurs.

If an attempt is made to execute an instruction that is UNDEFINED, an Undefined Instruction exception occurs (see *Extending the instruction set* on page A3-27).

The Undefined Instruction exception can be used for software emulation of a coprocessor in a system that does not have the physical coprocessor (hardware), or for general-purpose instruction set extension by software emulation.

When an Undefined Instruction exception occurs, the following actions are performed:

```
R14_und   = address of next instruction after the undefined instruction
SPSR_und  = CPSR
CPSR[4:0] = 0b11011              /* Enter Undefined mode */
CPSR[5]   = 0                    /* Execute in ARM state */
/* CPSR[6] is unchanged */
CPSR[7]   = 1                    /* Disable normal interrupts */
if high vectors configured then
    PC    = 0xFFFF0004
else
    PC    = 0x00000004
```

To return after emulating the undefined instruction use:

```
    MOVS PC,R14
```

This restores the PC (from R14_und) and CPSR (from SPSR_und) and returns to the instruction following the undefined instruction.

In some coprocessor designs, an internal exceptional condition caused by one coprocessor instruction is signaled *imprecisely* by refusing to respond to a later coprocessor instruction. In these circumstances, the Undefined Instruction handler takes whatever action is necessary to clear the exceptional condition, then returns to the second coprocessor instruction. To do this use:

```
    SUBS PC,R14,#4
```

2.6.3 Software Interrupt exception

The Software Interrupt instruction (`SWI`) enters Supervisor mode to request a particular supervisor (operating system) function. When a `SWI` is executed, the following actions are performed:

```
R14_svc    = address of next instruction after the SWI instruction
SPSR_svc   = CPSR
CPSR[4:0] = 0b10011              /* Enter Supervisor mode */
CPSR[5]    = 0                   /* Execute in ARM state */
/* CPSR[6] is unchanged */
CPSR[7]    = 1                   /* Disable normal interrupts */
if high vectors configured then
    PC     = 0xFFFF0008
else
    PC     = 0x00000008
```

To return after performing the SWI operation, use the following instruction to restore the PC (from R14_svc) and CPSR (from SPSR_svc) and return to the instruction following the SWI:

```
    MOVS PC,R14
```

2.6.4 Prefetch Abort (instruction fetch memory abort)

A memory abort is signaled by the memory system. Activating an abort in response to an instruction fetch marks the fetched instruction as invalid. A Prefetch Abort exception is generated if the processor tries to execute the invalid instruction. If the instruction is not executed (for example, as a result of a branch being taken while it is in the pipeline), no Prefetch Abort occurs.

In ARM architecture version 5 and above, a Prefetch Abort exception can also be generated as the result of executing a `BKPT` instruction. For details, see *BKPT* on page A4-14 (ARM instruction) and *BKPT* on page A7-24 (Thumb instruction).

When an attempt is made to execute an aborted instruction, the following actions are performed:

```
R14_abt    = address of the aborted instruction + 4
SPSR_abt   = CPSR
CPSR[4:0] = 0b10111              /* Enter Abort mode */
CPSR[5]    = 0                   /* Execute in ARM state */
/* CPSR[6] is unchanged */
CPSR[7]    = 1                   /* Disable normal interrupts */
if high vectors configured then
    PC     = 0xFFFF000C
else
    PC     = 0x0000000C
```

To return after fixing the reason for the abort, use:

```
    SUBS PC,R14,#4
```

This restores both the PC (from R14_abt) and CPSR (from SPSR_abt), and returns to the aborted instruction.

2.6.5 Data Abort (data access memory abort)

A memory abort is signaled by the memory system. Activating an abort in response to a data access (load or store) marks the data as invalid. A Data Abort exception occurs before any following instructions or exceptions have altered the state of the CPU. The following actions are performed:

```
R14_abt   = address of the aborted instruction + 8
SPSR_abt  = CPSR
CPSR[4:0] = 0b10111              /* Enter Abort mode */
CPSR[5]   = 0                    /* Execute in ARM state */
/* CPSR[6] is unchanged */
CPSR[7]   = 1                    /* Disable normal interrupts */
if high vectors configured then
    PC    = 0xFFFF0010
else
    PC    = 0x00000010
```

To return after fixing the reason for the abort use:

```
    SUBS PC,R14,#8
```

This restores both the PC (from R14_abt) and CPSR (from SPSR_abt), and returns to re-execute the aborted instruction.

If the aborted instruction does not need to be re-executed use:

```
    SUBS PC,R14,#4
```

Effects of data-aborted instructions

Instructions that access data memory can modify memory by storing one or more values. If a Data Abort occurs in such an instruction, the value of each memory location that the instruction stores to is:

- unchanged if the memory system does not permit write access to the memory location
- UNPREDICTABLE otherwise.

Instructions that access data memory can modify registers in three ways:

- By loading values into one or more of the general-purpose registers, which can include the PC.
- By specifying *base register writeback*, in which the base register used in the address calculation has a modified value written to it. All instructions that allow this to be specified have UNPREDICTABLE results if base register writeback is specified and the base register is the PC, so only general-purpose registers other than the PC can legitimately be modified in this way.
- By loading values into coprocessor registers.

If a Data Abort occurs, the values left in these registers are determined by the following rules:

1. The PC value on entry to the data abort handler is `0x00000010` or `0xFFFF0010`, and the R14_abt value is determined from the address of the aborted instruction. Neither is affected in any way by the results of any PC load specified by the instruction.

2. If base register writeback is not specified, the base register value is unchanged. This applies even if the instruction loaded its own base register and the memory access to load the base register occurred earlier than the aborting access.

   For example, suppose the instruction is:

   ```
   LDMIA R0,{R0,R1,R2}
   ```

   and the implementation loads the new R0 value, then the new R1 value and finally the new R2 value. If a Data Abort occurs on any of the accesses, the value in the base register R0 of the instruction is unchanged. This applies even if it was the load of R1 or R2 that aborted, rather than the load of R0.

3. If base register writeback is specified, the value left in the base register is determined by the *abort model* of the implementation, as described in *Abort models*.

4. If the instruction only loads one general-purpose register, the value in that register is unchanged.

5. If the instruction loads more than one general-purpose register, UNPREDICTABLE values are left in destination registers which are neither the PC nor the base register of the instruction.

6. If the instruction loads coprocessor registers, UNPREDICTABLE values are left in the destination coprocessor registers, unless otherwise specified in the instruction set description of the specific coprocessor.

Abort models

The abort model used by an ARM implementation is IMPLEMENTATION DEFINED, and is one of the following:

Base Restored Abort Model

If a Data Abort occurs in an instruction which specifies base register writeback, the value in the base register is unchanged.

Base Updated Abort Model

If a Data Abort occurs in an instruction which specifies base register writeback, the base register writeback still occurs.

In either case, the abort model applies uniformly across all instructions. An implementation does not use the Base Restored Abort Model for some instructions and the Base Updated Abort Model for others.

Note

In some ARMv3 and earlier implementations, a third abort model (the *Early Abort Model*) was used. In this model, base register writeback occurred for `LDC`, `LDM`, `STC` and `STM` instructions, and the base register was unchanged for all other instructions.

The Early Abort Model is not valid in ARM architecture versions 3M, 4 and above.

Some of these implementations optionally allowed a *Late Abort Model* to be selected. This is identical to the Base Updated Abort Model.

2.6.6 Interrupt request (IRQ) exception

The IRQ exception is generated externally by asserting the IRQ input on the processor. It has a lower priority than FIQ (see Table 2-4 on page A2-20), and is masked out when an FIQ sequence is entered.

Interrupts are disabled when the I bit in the CPSR is set. If the I bit is clear, ARM checks for an IRQ at instruction boundaries.

Note

The I bit can only be changed from a privileged mode.

When an IRQ is detected, the following actions are performed:

```
R14_irq   = address of next instruction to be executed + 4
SPSR_irq  = CPSR
CPSR[4:0] = 0b10010              /* Enter IRQ mode */
CPSR[5]   = 0                    /* Execute in ARM state */
/* CPSR[6] is unchanged */
CPSR[7]   = 1                    /* Disable normal interrupts */
if high vectors configured then
    PC    = 0xFFFF0018
else
    PC    = 0x00000018
```

To return after servicing the interrupt, use:

```
    SUBS PC,R14,#4
```

This restores both the PC (from R14_irq) and CPSR (from SPSR_irq), and resumes execution of the interrupted code.

2.6.7 Fast interrupt request (FIQ) exception

The FIQ exception is generated externally by asserting the FIQ input on the processor. FIQ is designed to support a data transfer or channel process, and has sufficient private registers to remove the need for register saving in such applications, therefore minimizing the overhead of context switching.

Fast interrupts are disabled when the F bit in the CPSR is set. If the F bit is clear, ARM checks for an FIQ at instruction boundaries.

Note

The F bit can only be changed from a privileged mode.

When an FIQ is detected, the following actions are performed:

```
R14_fiq   = address of next instruction to be executed + 4
SPSR_fiq  = CPSR
CPSR[4:0] = 0b10001             /* Enter FIQ mode */
CPSR[5]   = 0                   /* Execute in ARM state */
CPSR[6]   = 1                   /* Disable fast interrupts */
CPSR[7]   = 1                   /* Disable normal interrupts */
if high vectors configured then
    PC    = 0xFFFF001C
else
    PC    = 0x0000001C
```

To return after servicing the interrupt, use:

```
    SUBS PC, R14,#4
```

This restores both the PC (from R14_fiq) and CPSR (from SPSR_fiq), and resumes execution of the interrupted code.

The FIQ vector is deliberately the last vector to allow the FIQ exception-handler software to be placed directly at address `0x0000001C` or `0xFFFF001C`, without requiring a branch instruction from the vector.

2.6.8 Exception priorities

Table 2-4 shows the exception priorities:

Table 2-4 Exception priorities

Priority		Exception
Highest	1	Reset
	2	Data Abort
	3	FIQ
	4	IRQ
	5	Prefetch Abort
Lowest	6	Undefined instruction SWI

Undefined instruction and software interrupt cannot occur at the same time, as they each correspond to particular (non-overlapping) decodings of the current instruction, and both must be lower priority than prefetch abort, as a prefetch abort indicates that no valid instruction was fetched.

The priority of a Data Abort exception is higher than FIQ, which ensures that the data abort handler is entered before the FIQ handler is entered (so that the Data Abort is resolved after the FIQ handler has completed).

2.6.9 High vectors

Some ARM implementations allow the exception vector locations to be moved from their normal address range `0x00000000-0x0000001C` at the bottom of the 32-bit address space, to an alternative address range `0xFFFF0000-0xFFFF001C` near the top of the address space. These alternative locations are known as the *high vectors*.

It is IMPLEMENTATION DEFINED whether the high vectors are supported. When they are, a hardware configuration input selects whether the normal vectors or the high vectors are to be used.

The ARM instruction set does not contain any instructions which can directly change whether normal or high vectors are configured. However, if the standard System Control coprocessor is attached to an ARM processor which supports the high vectors, bit[13] of coprocessor 15 register 1 can be used to switch between using the normal vectors and the high vectors (see *Register 1: Control register* on page B2-13).

2.7 Memory and memory-mapped I/O

This section discusses memory and memory-mapped I/O, mainly with regard to the assumptions ARM processor implementations make about how their memory systems behave and how programs should access memory. As a result, it describes the memory system *from the outside*, specifying how it should behave without going into much detail of how this behavior can or should be implemented. More details of how some standard memory systems behave can be found in *Part B: Memory and System Architectures*.

The ARM architecture allows a wide variety of memory system designs, using the range of memory and I/O devices which are available. This makes it difficult to specify absolute rules about how a memory system should behave.

Many of the rules below can be broken if the hardware and software are designed appropriately. However, breaking these rules is discouraged, for the following reasons:

- It might make implementing the memory system more difficult.
- It might cause difficulties in porting the system (hardware and/or software) to future ARM processors.
- Standard software (such as compilers and other software toolkit components) might not work with the rule-breaking system.

2.7.1 Address space

The ARM architecture uses a single, flat address space of 2^{32} 8-bit bytes. Byte addresses are treated as unsigned numbers, running from 0 to 2^{32} - 1.

This address space is regarded as consisting of 2^{30} 32-bit words, each of whose addresses is *word-aligned*, which means that the address is divisible by 4. The word whose word-aligned address is A consists of the four bytes with addresses A, A+1, A+2 and A+3.

In ARM architecture version 4 and above, the address space is also regarded as consisting of 2^{31} 16-bit halfwords, each of whose addresses is *halfword-aligned* (divisible by 2). The halfword whose halfword-aligned address is A consists of the two bytes with addresses A and A+1.

Note

Prior to ARM architecture version 3, the address space was only 2^{26} bytes, with addresses running from 0 to 2^{26} - 1. This address space was split into 2^{24} words.

Some implementations of subsequent non-T variants of the ARM architecture include backwards-compatibility features to allow execution of code written for this address space. These features are described in Chapter A8 *The 26-bit Architectures*. Their use for any purpose other than executing old code is strongly discouraged.

These backwards-compatibility features are not compatible with T variants of the architecture, due to conflicting uses of bits[1:0] of R15.

Address calculations are normally performed using ordinary integer instructions. This means that they normally *wrap around* if they overflow or underflow the address space. This means that the result of the calculation is reduced modulo 2^{32}. However, to minimize the chances of incompatibility if the address space is extended in the future, programs should not be written so that they rely on this behavior. Address calculations should be written so that their results would still lie in the range 0 to 2^{32} - 1 if they were calculated without wrap-around.

Most branch instructions calculate their targets by adding an instruction-specified offset to the value of the PC and writing the result back to the PC. If the overall effect of this calculation of:

```
(address_of_current_instruction) + 8 + offset
```

is to overflow or underflow the address space, the instruction is technically UNPREDICTABLE because it relies on address wrap-around. The result of this is that forward branches past address `0xFFFFFFFF` and backward branches past address `0x00000000` should not be used.

Also, normal sequential execution of instructions effectively calculates:

```
(address_of_current_instruction) + 4
```

after each instruction to determine which instruction to execute next. If this calculation overflows the top of the address space, the result is again technically UNPREDICTABLE. In other words, programs should not rely on sequential execution of the instruction at address `0x00000000` after the instruction at address `0xFFFFFFFC`.

Note

The above only applies to instructions that are executed, including those which fail their condition code check. Most ARM implementations prefetch instructions ahead of the currently-executing instruction. If this prefetching overflows the top of the address space, it does not cause the implementation's behavior to become UNPREDICTABLE until and unless the prefetched instructions are actually executed.

`LDC`, `LDM`, `STC`, and `STM` instructions access a sequence of words at increasing memory addresses, effectively incrementing a memory address by 4 for each load or store. If this calculation overflows the top of the address space, the result is again technically UNPREDICTABLE. In other words, programs should not use these instructions in such a way that they access the word at address `0x00000000` sequentially after the word at address `0xFFFFFFFC`.

2.7.2 Endianness

The rules in *Address space* on page A2-22 require that for a word-aligned address A:

- The word at address A consists of the bytes at addresses A, A+1, A+2 and A+3.
- The halfword at address A consists of the bytes at addresses A and A+1.
- The halfword at address A+2 consists of the bytes at addresses A+2 and A+3.
- The word at address A therefore consists of the halfwords at addresses A and A+2.

However, this does not totally specify the mappings between words, halfwords and bytes.

A memory system uses one of the two following mapping schemes. This choice is known as the *endianness* of the memory system.

- In a *little-endian* memory system:
  - — a byte or halfword at a word-aligned address is the least significant byte or halfword within the word at that address
  - — a byte at a halfword-aligned address is the least significant byte within the halfword at that address.
- In a *big-endian* memory system:
  - — a byte or halfword at a word-aligned address is the most significant byte or halfword within the word at that address
  - — a byte at a halfword-aligned address is the most significant byte within the halfword at that address.

For a word-aligned address A, Table 2-5 and Table 2-6 show how the word at address A, the halfwords at addresses A and A+2, and the bytes at addresses A, A+1, A+2 and A+3 map on to each other for each endianness.

Table 2-5 Big-endian memory system

31 24	23 16	15 8	7 0
Word at address A			
Halfword at address A		Halfword at address A+2	
Byte at address A	Byte at address A+1	Byte at address A+2	Byte at address A+3

Table 2-6 Little-endian memory system

31	20 19	12 11 10 9 8	5 4 3 2 1 0
Word at address A			
Halfword at address A+2		Halfword at address A	
Byte at address A+3	Byte at address A+2	Byte at address A+1	Byte at address A

It is IMPLEMENTATION DEFINED whether an ARM implementation supports little-endian memory systems, big-endian memory systems, or both.

The ARM instruction set does not contain any instructions that directly select the endianness. Instead, an ARM implementation which supports both endiannesses has a hardware input to configure it to match the endianness of the attached memory system. If a standard System Control coprocessor is attached to such an ARM implementation, this configuration input can be changed by writing to bit[7] of register 1 of the System Control coprocessor (see *Register 1: Control register* on page B2-13).

If an ARM implementation is configured for a memory system of one endianness but is actually attached to a memory system of the opposite endianness, only word-sized instruction fetches, data loads and data stores can be relied upon. Other memory accesses have UNPREDICTABLE results.

When the standard System Control coprocessor is attached to an ARM processor that supports both endiannesses, bit[7] of the coprocessor's register 1 is cleared on reset. This means that the ARM processor is configured for a little-endian memory system immediately after reset. If it is attached to a big-endian memory system, one of the first things the reset handler must do is switch the configured endianness to big-endian, using an instruction sequence like:

```
MRC    p15, 0, r0, c1, c0    ; r0 := CP15 register 1
ORR    r0, r0, #0x80         ; Set bit[7] in r0
MCR    p15, 0, r0, c1, c0    ; CP15 register 1 := r0
```

This must be done before there is any possibility of a byte or halfword data access occurring, or of a Thumb instruction being executed.

Note

The rules on endianness imply that word loads and stores are not affected by the configured endianness. Because of this, it is not possible to reverse the order of the bytes in a word by storing it, changing the configured endianness, and reloading the stored word. Instead, use one of the code sequences in *Swapping endianness* on page A9-4.

More generally, there is no point in changing the configured endianness of an ARM processor to be different from that of the memory system it is attached to, because no additional architecturally defined operations become available as a result of doing so. So normally, the only time the configured endianness is changed is at reset to make it match the memory system endianness.

2.7.3 Unaligned memory accesses

The ARM architecture normally expects all memory accesses to be suitably aligned. In particular, the address used for a word access should normally be word-aligned, and the address used for a halfword access should normally be halfword-aligned. Memory accesses which are not aligned in this way are called *unaligned* memory accesses.

Unaligned instruction fetches

If an address which is not word-aligned is written to R15 during ARM state execution, the result is normally either UNPREDICTABLE or that bits[1:0] of the address are ignored. If an address which is not halfword-aligned is written to R15 during Thumb state execution, bit[0] of the address is normally ignored. See *Writing the program counter* on page A2-8 and individual instruction descriptions for more details. As a result, a value which is read from R15 during execution of valid code always has bits[1:0] zero for ARM state execution and bit[0] zero for Thumb state execution.

When it is specified that these bits are ignored, ARM implementations are not required to ensure that they are cleared from the address sent to memory for the instruction fetch. They can instead send the value written to R15 unchanged to memory, and require the memory system to ignore bits[1:0] of the address for an ARM instruction fetch and bit[0] for a Thumb instruction fetch.

Unaligned data accesses

The architecturally defined behavior of a load/store instruction which generates an unaligned access is one of the following:

- It is UNPREDICTABLE.
- It ignores the low-order address bits that make the access unaligned. This means it effectively uses the formula `(address AND 0xFFFFFFFE)` for a halfword access, and uses the formula `(address AND 0xFFFFFFFC)` for a word access.
- It ignores the low-order address bits that make the access unaligned for the memory access itself, but then uses those low-order bits to control a rotation of the loaded data. (This behavior applies only to the `LDR` and `SWP` instructions.)

Which of these three options applies to a load/store instruction depends on which instruction it is, and is documented on the individual instruction pages.

ARM implementations are not required to ensure that the low-order address bits that make an access unaligned are cleared from the address they send to memory. They can instead send the address as calculated by the load/store instruction unchanged to memory, and require the memory system to ignore address[0] for a halfword access and address[1:0] for a word access.

Note

When an instruction ignores the low-order address bits that make an access unaligned, the pseudo-code in the instruction description does not mask them out explicitly. Instead, the Memory[<address>,<size>] function used in the pseudo-code masks them out implicitly. This function is defined in the *Glossary*.

2.7.4 Prefetching and self-modifying code

Many ARM implementations fetch instructions from memory before execution of the instructions that precede them has completed. This behavior is known as *prefetching* the instruction.

Prefetching an instruction does not commit the ARM implementation to actually executing the instruction. Two typical cases in which the instruction is not subsequently executed are:

- When an exception occurs, execution of the current instruction is completed, all further prefetched instructions are discarded, and execution of the instructions at the exception vector is started.
- When a branch is taken, any instructions that have already been prefetched from sequential locations beyond the branch are discarded.

ARM implementations are free to choose how far ahead of the current point of execution they prefetch instructions, or even to have a dynamically varying number of prefetched instructions. The original ARM implementation prefetched two instructions ahead of the instruction currently being executed, but implementations are free to choose to prefetch more or less than this.

Note

When an instruction reads the PC, it gets the address of the instruction that is two after itself:

- for ARM instructions, it gets its own address plus 8
- for Thumb instructions, it gets its own address plus 4.

Historically, there is a link between this two-instruction offset for PC reads and the two-instruction prefetch of the original ARM implementation. However, this link is not architectural. An implementation which prefetches a different number of instructions still ensures that an instruction which reads the PC gets the address of the instruction two after itself.

As well as being free to choose how many instructions to prefetch, an ARM implementation can choose which possible future execution path to prefetch along. For example, after a branch instruction, it can choose to prefetch either the instruction following the branch or the instruction at the branch target. This is known as *branch prediction*.

A potential problem with all forms of instruction prefetching is that the instruction in memory might be changed after it was prefetched but before it is executed. If this happens, the modification to the instruction in memory does not normally prevent the already prefetched copy of the instruction from executing to completion.

For example, in the following code sequence, the `STR` instruction replaces the `SUB` instruction that follows it by a copy of the `ADD` instruction:

```
    LDR     r0, AddInstr
    STR     r0, NextInstr
NextInstr
    SUB     r1, r1, #1
    :
    :
AddInstr
    ADD     r1, r1, #1
```

However, the instruction executed after the `STR` instruction is normally the `SUB` instruction on the first occasion that this code is executed, because the `SUB` instruction was prefetched before the instruction in memory was changed. The `ADD` instruction is not executed until the second time that the code sequence is executed.

Furthermore, even this behavior cannot be guaranteed, because:

- On the first occasion that the code sequence is executed, it is possible that an interrupt will occur immediately after the `STR`. If it does, the `SUB` instruction that had been prefetched is discarded. When the interrupt handler returns, the instruction at `NextInstr` is prefetched again, and is the `ADD` instruction this time. So while the `SUB` instruction is normally executed on the first occasion that the code is executed, it is possible that the `ADD` instruction is executed instead.
- Either the ARM processor or the memory system is allowed to keep copies of instructions fetched from memory and use those copies instead of repeating the instruction fetch if the instruction is executed again. If this occurs, a copy of the `SUB` instruction can be executed on the second or subsequent occasion that the code sequence is executed.

  The main reason that this might occur is that the memory system contains separate instruction and data caches (see *Instruction cache coherency* on page B5-11). However, other possibilities also exist. For example, some forms of branch prediction hardware keep copies of the instructions at branch targets.

The overall result is that code which writes one or more instructions to memory and then executes them (known as *self-modifying code*) cannot be executed reliably on ARM processors without special precautions. Programming techniques that involve the use of self-modifying code are to be avoided as far as possible.

Instruction Memory Barriers (IMBs)

In many systems, however, it is impossible to avoid the use of self-modifying code entirely. For example, any system which allows a program to be loaded into memory and then executed is using self-modifying code.

Each implementation therefore defines a sequence of operations that can be used in the middle of a self-modifying code sequence to make it execute reliably. This sequence is called an *Instruction Memory Barrier* (IMB), and often depends both on the ARM processor implementation and on the memory system implementation.

The IMB sequence must be executed after the new instructions have been stored to memory and before they are executed, for example, after a program has been loaded and before its entry point is branched to. Any self-modifying code sequence which does not use an IMB in this way has UNPREDICTABLE behavior.

Because the exact sequence of operations to be performed by an IMB depends on the ARM and memory system implementations, it is recommended that software is designed so that the IMB sequence is provided as a call to an easily replaceable *system dependencies* module, rather than being included in-line where it is needed. This eases porting to other ARM processors and memory systems.

Also, in many implementations, the IMB sequence includes operations that are only usable from privileged processor modes, such as the cache cleaning and invalidation operations supplied by the standard System Control coprocessor (see Chapter B5 *Caches and Write Buffers*). To allow User mode programs to use the IMB sequence, it is recommended that it is supplied as an operating system call, invoked by a `SWI` instruction.

In systems that use the 24-bit immediate in a `SWI` instruction to specify the required operating system service, it is recommended that the IMB sequence is requested by the instruction:

```
SWI 0xF00000
```

This call takes no parameters and does not return a result, and should use the same calling conventions as a call to a C function with prototype:

```
void IMB(void);
```

apart from the fact that a `SWI` instruction is used for the call, rather than a `BL` instruction.

Some implementations can use knowledge of the range of addresses to which new instructions have been stored to reduce the execution time cost of an IMB. It is therefore also recommended that a second operating system call is supplied which does an IMB with respect to a specified address range only. On systems that use the 24-bit immediate in a `SWI` instruction to specify the required operating system service, this should be requested by the instruction:

```
SWI 0xF00001
```

and should use similar calling conventions to those used by a call to a C function with prototype:

```
void IMB_Range(unsigned long start_addr, unsigned long end_addr);
```

where the address range runs from `start_addr` (inclusive) to `end_addr` (exclusive).

Note

- When the standard ARM Procedure Calling Standard is used, this means that `start_addr` is passed in R0 and `end_addr` in R1.
- On some ARM implementations, the execution time cost of an IMB can be very large (many thousands of clock cycles), even when a small address range is specified. For small scale uses of self-modifying code, this is likely to lead to a major loss of performance. It is therefore recommended that self-modifying code is only used where it is unavoidable and/or it produces sufficiently large execution time benefits to offset the cost of the IMB.

Other uses for IMBs

Some memory systems allow virtual-to-physical address mapping, in which the physical memory location corresponding to an address generated by the ARM processor can be changed. If this address mapping is changed after an instruction has been prefetched but before it is executed, and the address of the instruction is affected by the change of address mapping, then the wrong instruction is executed.

This is very similar to the situation that arises if a store occurs to an instruction address after it has been prefetched but before it is executed. In both cases, the instruction held at the memory address is being changed, either because a value is being stored to it or because a different physical memory location becomes associated with the address. The same solution is therefore used when the virtual-to-physical address mapping is changed. The IMB sequence must be executed after a change of virtual-to-physical address mapping and before any attempt to execute an instruction from a memory area whose address mapping has been changed.

Another similar case occurs if memory access permissions are changed between prefetching and executing an instruction. If access was not permitted when the instruction was prefetched but is permitted when it is executed, an unexpected Prefetch Abort exception might occur. In the opposite case that access was permitted when the instruction was prefetched and is no longer permitted when it is executed, there might be a security hole in the system.

Memory access permissions can typically be changed either by explicitly writing new access permission settings to the memory system, or because the memory system supports different access permissions for User mode and privileged modes and one of the following occurs:

- An exception occurs in User mode, causing the processor to switch to a privileged mode.
- Privileged code changes mode to User mode.

All ARM implementations ensure that the following events do not cause any instructions to be executed after having been prefetched with the wrong access permissions:

- An exception occurring in User mode.
- Execution of one of the instructions designed for exception return causing a change from a privileged mode to User mode. These instructions are the ones which have a side-effect of copying the SPSR of the current mode to the CPSR, namely:
  - The data processing instructions `ADCS`, `ADDS`, `ANDS`, `BICS`, `EORS`, `MOVS`, `MVNS`, `ORRS`, `RSBS`, `RSCS`, `SBCS` and `SUBS` when their destination register is R15. (However, only `MOVS` and `SUBS` are commonly used for exception return.)
  - The form of the `LDM` instruction described in *LDM (3)* on page A4-34.

The same is not guaranteed in the remaining cases where memory access permissions might change between prefetching and executing an instruction. These are:

- Explicitly writing new access permission settings to the memory system.
- Changing from a privileged mode to User mode by means of an `MSR` instruction.

In these cases, an IMB sequence needs to be executed shortly after the change of access permissions, and none of the instructions executed after the change of access permissions and before the Instruction Memory Barrier should be affected by the change of access permissions.

However, the cost of a full IMB can often be avoided in these cases. In particular, the instruction word associated with any particular address has not changed, so it is usually possible to avoid cache flushes. An implementation can therefore define restricted versions of the IMB sequence to be used in these cases.

In the case of an `MSR` instruction changing from a privileged mode to User mode, a restricted version of the IMB sequence that works on all ARM processors to date is simply to execute any instruction that writes to the PC, other than the branch instructions described in the following sections:

- *B, BL* on page A4-10
- *BLX (1)* on page A4-16
- *B (1)* on page A7-18
- *B (2)* on page A7-20
- *BL, BLX(1)* on page A7-26.

In other words, the mode change should not affect the access permissions of any instructions that can be reached from the `MSR` instruction by any combination of:

- Normal sequential execution of instructions.
- For each branch from the above list that can be reached in this way, execution of the instruction at its target. (The branch instructions in the list are precisely those that have a fixed, statically determined target.)

This set of instructions is occasionally referred to elsewhere in this manual as the set of instructions that can be reached by *predictable subsequent execution* from the `MSR` instruction.

2.7.5 Memory-mapped I/O

The standard way to perform I/O functions on ARM systems is by the use of *memory-mapped I/O*. This uses special memory addresses which supply I/O functions when they are loaded from or stored to. Typically, loading from a memory-mapped I/O address is used for input, and storing to a memory-mapped I/O address is used for output. Both loads and stores can also be used to perform control functions, either instead of or in addition to their normal input or output function.

The behavior of a memory-mapped I/O location usually differs from that expected of a normal memory location. For example, two successive loads from a normal memory location return the same value each time unless there has been an intervening store to that location. For a memory-mapped I/O location, the value returned by the second load can be different from the value returned by the first load. Typically, this is because the first load has a side-effect (such as removing the loaded value from a buffer) or because of a side-effect of an intervening load or store to another memory-mapped I/O location.

These differences in behavior mainly affect the use of caches and write buffers in the memory system. This is discussed in Chapter B5 *Caches and Write Buffers*. In short, memory-mapped I/O locations are normally marked as uncachable and unbufferable, to avoid changes to the number, type, order, or timing of the accesses made to them.

Instruction fetches from memory-mapped I/O

As described in *Prefetching and self-modifying code* on page A2-27, ARM implementations can vary considerably with regard to when they fetch instructions from memory. As a result, it is strongly recommended that memory-mapped I/O locations are only used for data loads and stores, not for instruction fetches. Any system design which relies on executing instructions fetched from a memory-mapped I/O location is likely to be hard to port to future ARM implementations.

Data accesses to memory-mapped I/O

An instruction sequence accesses data memory at various points during its execution, generating a sequence of load and store accesses. Provided these loads and stores access normal memory locations, they only interact with each other if they access the same memory location. As a result, loads and stores to distinct normal memory locations can be performed in a different order to that implied by the instruction sequence, without changing the final result of the sequence. This freedom to change the order of memory accesses can be exploited by a memory system to improve performance (for example, by the use of caches and write buffers).

Furthermore, data accesses to the same normal memory location have other properties that can be exploited to improve performance. These include:

- Successive loads from the same location without an intervening store generate identical results.
- A load from a location returns the last value stored to that location.
- Multiple accesses of one data size can sometimes be merged into a single, larger size access. For example, separate stores to the two halfwords contained within a word can be merged to produce a single word store.

However, if the memory words, halfwords or bytes accessed by the code sequence are memory-mapped I/O locations, one access can generate a side-effect which changes the results of a subsequent access to a different location. If this happens, the time order of individual accesses makes a difference to the final results of the code sequence. Also, a load access to a memory-mapped I/O location can have a side-effect that changes the result of a subsequent access to the same location. Accesses to memory-mapped I/O locations must therefore not be optimized away, and their time order must not be changed.

It is also important that for memory-mapped I/O, the data size of each memory access is maintained. For example, a code sequence that specifies 4 byte reads from 4 sequential byte addresses must not be merged into a single word read when accessing memory-mapped I/O. Such a system might cause the final results of the code sequence to be different from that intended. Similarly a system which splits word accesses up into many byte accesses might cause memory-mapped I/O devices not to operate as expected.

Each ARM implementation provides a mechanism to ensure that no changes are made to the number of accesses in a sequence of data memory accesses, or to their data sizes, or time order. This mechanism consists of IMPLEMENTATION DEFINED requirements on the memory accesses whose number, data sizes, and time order are to be preserved. If these requirements are not adhered to for accesses to memory-mapped I/O locations, unexpected behavior might occur.

Typical requirements include:

- Constraints on memory attributes of the memory-mapped I/O locations. For example, in the standard memory system architectures described in *Part B: Memory and System Architectures*, the memory locations must be uncachable and unbufferable.
- Constraints on the sizes or alignments of the accesses to the memory-mapped I/O locations. For example, if an ARM implementation has a 16-bit external data bus, it might prohibit the use of 32-bit accesses to memory-mapped I/O locations, since they cannot be performed in a single bus cycle.
- A requirement for additional external hardware. For example, an alternative possibility for an ARM implementation with a 16-bit external bus is to allow 32-bit accesses to memory-mapped I/O locations, but require external hardware to re-assemble the two 16-bit bus accesses into a single 32-bit access to the I/O device.

If a sequence of data memory accesses includes some accesses which meet the requirements for memory-mapped I/O accesses and some which do not, then:

- The number and data sizes of the accesses that meet the requirements are preserved. In particular, they are not merged with each other or with the accesses that do not meet the requirements in any way. The accesses which do not meet the requirements can be merged with each other.
- The time order of the accesses which meet the requirements are preserved relative to each other. Their time order relative to accesses which do not meet the requirements is not guaranteed.

Time ordering of LDM and STM instructions

The `LDM` instruction performs a sequence of loads from successive words in memory, and the `STM` instruction performs a similar sequence of stores. The rules described above for accessing memory-mapped I/O apply to the sequence of word accesses within one of these instructions in the same way as they do to a series of separate memory access instructions.

The time order of the sequence of memory accesses performed by an `LDM` or `STM` instruction is only architecturally defined under limited circumstances. The rules for this are:

- If the register list in the instruction includes the PC, the time order of the sequence of memory accesses is not defined. (This means that such `LDM` and `STM` instructions are not suitable for accessing memory-mapped I/O.)
- If the register list in the instruction does not include the PC, the time order of the sequence of memory accesses is in order of memory address, starting with the lowest address and ending with the highest address. (This order is identical to ascending register number order within the list of registers to be loaded or stored.)
- If *all* of the memory accesses generated by an `LDM` or `STM` meet the IMPLEMENTATION DEFINED requirements to be treated as memory-mapped I/O locations, then their number, data sizes and time order are preserved.

- If some of the memory accesses generated by an `LDM` or `STM` meet the IMPLEMENTATION DEFINED requirements to be treated as memory-mapped I/O locations, but others do not, then their number, data sizes and time order are not guaranteed to be preserved. In particular, the ARM processor and memory system do not even necessarily preserve the relative time order of the accesses that do meet the requirements. This is an exception to the normal rules that govern what happens when some accesses meet the requirements and others do not.

  For example, with the standard memory systems described in *Part B: Memory and System Architectures*, the time order of the memory accesses is not guaranteed to be preserved if the `LDM` or `STM` crosses the boundary between a cachable area of memory and an uncachable, unbufferable area. Such `LDM` and `STM` instructions are therefore not suitable for memory-mapped I/O.

Chapter A3
The ARM Instruction Set

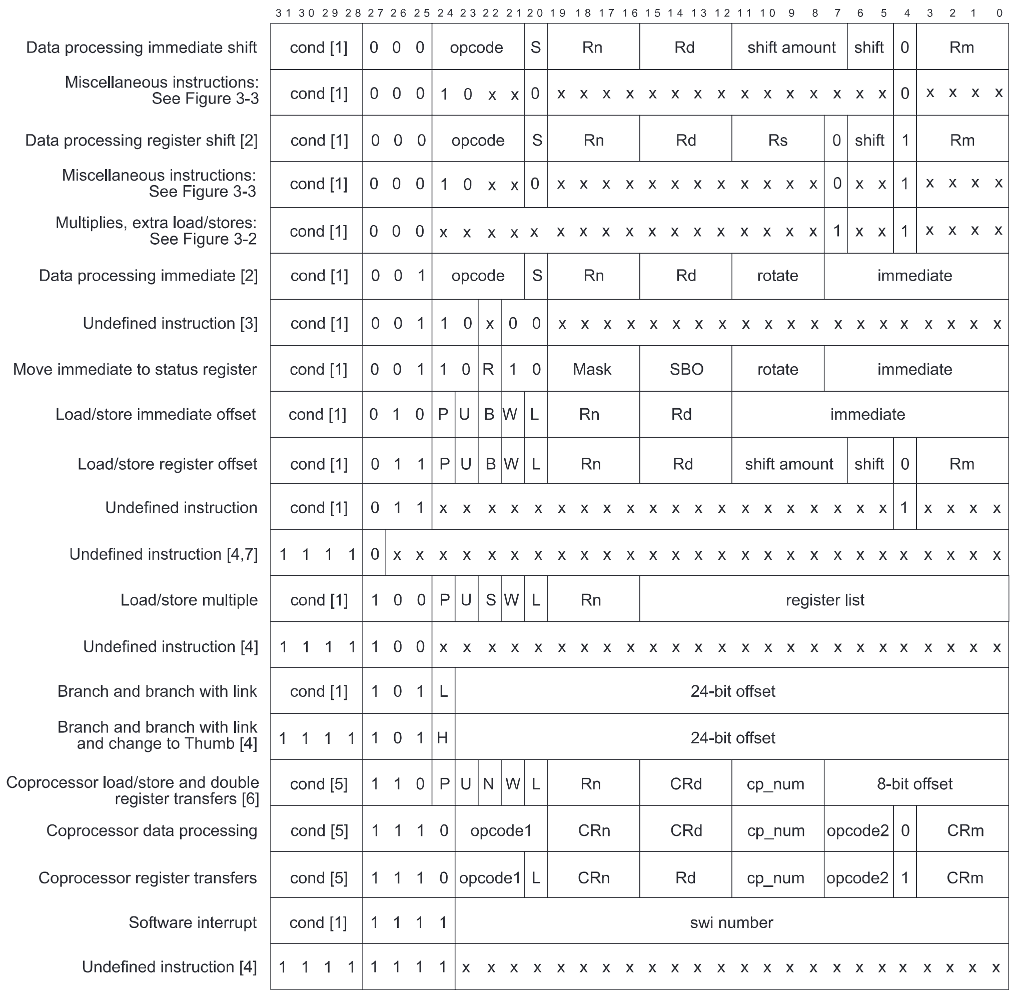
This chapter describes the ARM instruction set and contains the following sections:

3.1 Instruction set encoding

Figure 3-1 shows the ARM instruction set encoding.

All other bit patterns are UNPREDICTABLE or UNDEFINED. See *Extending the instruction set* on page A3-27 for a description of the cases where instructions are UNDEFINED.

An entry in square brackets, for example [1], indicates that more information is given after the figure.



	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
Data processing immediate shift	cond [1]				0	0	0	opcode				S	Rn				Rd				shift amount					shift		0	Rm			
Miscellaneous instructions: See Figure 3-3	cond [1]				0	0	0	1	0	x	x	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0	x	x	x	x
Data processing register shift [2]	cond [1]				0	0	0	opcode				S	Rn				Rd				Rs				0	shift		1	Rm			
Miscellaneous instructions: See Figure 3-3	cond [1]				0	0	0	1	0	x	x	0	x	x	x	x	x	x	x	x	x	x	x	x	0	x	x	1	x	x	x	x
Multiplies, extra load/stores: See Figure 3-2	cond [1]				0	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	1	x	x	1	x	x	x	x
Data processing immediate [2]	cond [1]				0	0	1	opcode				S	Rn				Rd				rotate				immediate							
Undefined instruction [3]	cond [1]				0	0	1	1	0	x	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Move immediate to status register	cond [1]				0	0	1	1	0	R	1	0	Mask				SBO				rotate				immediate							
Load/store immediate offset	cond [1]				0	1	0	P	U	B	W	L	Rn				Rd				immediate											
Load/store register offset	cond [1]				0	1	1	P	U	B	W	L	Rn				Rd				shift amount					shift		0	Rm			
Undefined instruction	cond [1]				0	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	1	x	x	x	x
Undefined instruction [4,7]	1	1	1	1	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Load/store multiple	cond [1]				1	0	0	P	U	S	W	L	Rn				register list															
Undefined instruction [4]	1	1	1	1	1	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Branch and branch with link	cond [1]				1	0	1	L	24-bit offset																							
Branch and branch with link and change to Thumb [4]	1	1	1	1	1	0	1	H	24-bit offset																							
Coprocessor load/store and double register transfers [6]	cond [5]				1	1	0	P	U	N	W	L	Rn				CRd				cp_num				8-bit offset							
Coprocessor data processing	cond [5]				1	1	1	0	opcode1				CRn				CRd				cp_num				opcode2			0	CRm			
Coprocessor register transfers	cond [5]				1	1	1	0	opcode1			L	CRn				Rd				cp_num				opcode2			1	CRm			
Software interrupt	cond [1]				1	1	1	1	swi number																							
Undefined instruction [4]	1	1	1	1	1	1	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Figure 3-1 ARM instruction set summary

1. The `cond` field is not allowed to be 1111 in this line. Other lines deal with the cases where bits[31:28] of the instruction are 1111.
2. If the `opcode` field is of the form 10xx and the `S` field is 0, one of the following lines applies instead.
3. UNPREDICTABLE prior to ARM architecture version 4.
4. UNPREDICTABLE prior to ARM architecture version 5.
5. If the `cond` field is 1111, this instruction is UNPREDICTABLE prior to ARM architecture version 5.
6. The coprocessor double register transfer instructions are described in Chapter A10 *Enhanced DSP Extension*.
7. In E variants of architecture version 5 and above, the cache preload instruction `PLD` uses a small number of these instruction encodings.

3.1.1 Multiplies and extra load/store instructions

Figure 3-2 shows extra multiply and load/store instructions. An entry in square brackets, for example [1], indicates that more information is given below the figure.

	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
Multiply (accumulate)	cond	0	0	0	0	0	0	A	S	Rd	Rn	Rs	1	0	0	1	Rm
Multiply (accumulate) long	cond	0	0	0	0	1	U	A	S	RdHi	RdLo	Rs	1	0	0	1	Rm
Swap/swap byte	cond	0	0	0	1	0	B	0	0	Rn	Rd	SBZ	1	0	0	1	Rm
Load/store halfword register offset [1]	cond	0	0	0	P	U	0	W	L	Rn	Rd	SBZ	1	0	1	1	Rm
Load/store halfword immediate offset [1]	cond	0	0	0	P	U	1	W	L	Rn	Rd	HiOffset	1	0	1	1	LoOffset
Load/store two words register offset [2]	cond	0	0	0	P	U	0	W	0	Rn	Rd	SBZ	1	1	S	1	Rm
Load signed halfword/byte register offset [1]	cond	0	0	0	P	U	0	W	1	Rn	Rd	SBZ	1	1	H	1	Rm
Load/store two words immediate offset [2]	cond	0	0	0	P	U	1	W	0	Rn	Rd	HiOffset	1	1	S	1	LoOffset
Load signed halfword/byte immediate offset [1]	cond	0	0	0	P	U	1	W	1	Rn	Rd	HiOffset	1	1	H	1	LoOffset

Figure 3-2 Multiplies and extra load/store instructions

1. UNPREDICTABLE prior to ARM architecture version 4.
2. These instructions are described in Chapter A10 *Enhanced DSP Extension*.

Note

Any instruction with bits[27:25] = 000, bit[7] = 1, bit[4] = 1, and `cond` not equal to 1111, and which is not specified in Figure 3-2 or its notes, is an undefined instruction (or UNPREDICTABLE prior to ARM architecture version 4).

3.1.2 Miscellaneous instructions

Figure 3-3 shows the remaining ARM instruction encodings. An entry in square brackets, for example [1], indicates that more information is given below the figure.

	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
Move status register to register	cond	0 0 0 1 0	R	0	0	SBO	Rd	SBZ	0 0 0 0	SBZ
Move register to status register	cond	0 0 0 1 0	R	1	0	mask	SBO	SBZ	0 0 0 0	Rm
Branch/exchange instruction set [1]	cond	0 0 0 1 0	0	1	0	SBO	SBO	SBO	0 0 0 1	Rm
Count leading zeros [2]	cond	0 0 0 1 0	1	1	0	SBO	Rd	SBO	0 0 0 1	Rm
Branch and link/exchange instruction set [2]	cond	0 0 0 1 0	0	1	0	SBO	SBO	SBO	0 0 1 1	Rm
Enhanced DSP add/subtracts [4]	cond	0 0 0 1 0	op		0	Rn	Rd	SBZ	0 1 0 1	Rm
Software breakpoint [2,3]	cond	0 0 0 1 0	0	1	0	immed			0 1 1 1	immed
Enhanced DSP multiplies[4]	cond	0 0 0 1 0	op		0	Rd	Rn	Rs	1 y x 0	Rm

Figure 3-3 Miscellaneous instructions

1. Defined in ARM architecture version 5 and above, and in T variants of ARM architecture version 4.
2. This is an undefined instruction is ARM architecture version 4, and is UNPREDICTABLE prior to ARM architecture version 4.
3. If the `cond` field of this instruction is not 1110, it is UNPREDICTABLE.
4. The enhanced DSP instructions are described in Chapter A10 *Enhanced DSP Extension*.

Note

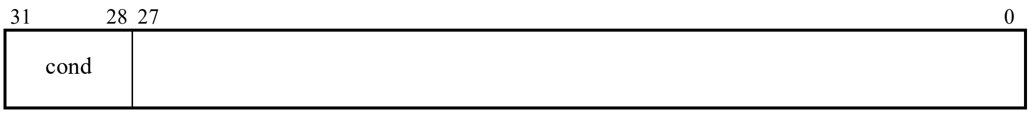
Any instruction with bits[27:23] = 00010, bit[20] = 0, bit[7] and bit[4] not both 1, and `cond` is not equal to 1111, and which is not specified in Figure 3-3 or its notes, is an undefined instruction (or UNPREDICTABLE prior to architecture version 4).

3.2 The condition field

Almost all ARM instructions can be *conditionally executed*, which means that they only have their normal effect on the programmer's model state, memory and coprocessors if the N, Z, C and V flags in the CPSR satisfy a condition specified in the instruction. If the flags do not satisfy this condition, the instruction acts as a NOP: that is, execution advances to the next instruction as normal, including any relevant checks for interrupts and prefetch aborts, but has no other effect.

Prior to ARM architecture version 5, all ARM instructions could be conditionally executed. A few instructions have been introduced subsequently which can only be executed unconditionally.

Every instruction contains a 4-bit condition code field in bits 31 to 28:



This field contains one of the 16 values described in Table 3-1 on page A3-6. Most instruction mnemonics can be extended with the letters defined in the mnemonic extension field.

If the *always* (AL) condition is specified, the instruction is executed irrespective of the value of the condition code flags. The absence of a condition code on an instruction mnemonic implies the AL condition code.

3.2.1 Condition code 0b1111

As indicated in Table 3-1 on page A3-6, if the condition field is 0b1111, the behavior depends on the architecture version:

- Prior to ARM architecture version 3, a condition field of 0b1111 meant that the instruction was never executed. The mnemonic extension for this condition was NV.

  Note

  Use of this condition is now obsolete and unsupported.

- In ARM architecture version 3 and version 4, any instruction with a condition field of 0b1111 is UNPREDICTABLE.

- In ARM architecture version 5 and above, a condition field of 0b1111 is used to encode various additional instructions which can only be executed unconditionally. All instruction encoding diagrams which show bits[31:28] as `cond` only match instructions in which these bits are not equal to 0b1111, unless otherwise stated in the individual instruction description.

Table 3-1 Condition codes

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0,N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-
1111	(NV)	See *Condition code 0b1111* on page A3-5	-

3.3 Branch instructions

All ARM processors support a branch instruction that allows a conditional branch forwards or backwards up to 32MB. As the PC is one of the general-purpose registers (R15), a branch or jump can also be generated by writing a value to R15.

A subroutine call can be performed by a variant of the standard branch instruction. As well as allowing a branch forward or backward up to 32MB, the Branch with Link (`BL`) instruction preserves the address of the instruction after the branch (the return address) in the LR (R14).

In T variants of ARM architecture version 4, and in ARM architecture version 5 and above, the Branch and Exchange (`BX`) instruction copies the contents of a general-purpose register Rm to the PC (like a `MOV PC,Rm` instruction), with the additional functionality that if bit[0] of the transferred value is 1, the processor shifts to Thumb state. Together with the corresponding Thumb instructions, this allows *interworking* branches between ARM and Thumb code.

Interworking subroutine calls can be generated by combining `BX` with an instruction to write a suitable return address to the LR, such as an immediately preceding `MOV LR,PC` instruction.

In ARM architecture version 5 and above, there are also two types of Branch with Link and Exchange (`BLX`) instruction:

- One type takes a register operand Rm, like a `BX` instruction. This instruction behaves like a `BX` instruction, and additionally writes the address of the next instruction into the LR. This provides a more efficient interworking subroutine call than a sequence of `MOV LR,PC` followed by `BX Rm`.
- The other type behaves like a `BL` instruction, branching backwards or forwards by up to 32MB and writing a return link to the LR, but shifts to Thumb state rather than staying in ARM state as `BL` does. This provides a more efficient alternative to loading the subroutine address into Rm followed by a `BLX Rm` instruction when it is known that a Thumb subroutine is being called and that the subroutine lies within the 32MB range.

A load instruction provides a way to branch anywhere in the 4GB address space (known as a *long branch*). A 32-bit value is loaded directly from memory into the PC, causing a branch. A long branch can be preceded by `MOV LR,PC` or another instruction that writes the LR to generate a long subroutine call. In ARM architecture version 5 and above, bit[0] of the value loaded by a long branch controls whether the subroutine is executed in ARM state or Thumb state, just like bit[0] of the value moved to the PC by a `BX` instruction. Prior to ARM architecture version 5, bits[1:0] of the value loaded into the PC are ignored, and a load into the PC can only be used to call a subroutine in ARM state.

In non-T variants of ARM architecture version 5, the instructions described above can cause an entry into Thumb state despite the fact that the Thumb instruction set is not present. This causes the instruction at the branch target to enter the undefined instruction trap. See *The control bits* on page A2-10 for more details.

3.3.1 Examples

```
    B      label                ; branch unconditionally to label

    BCC    label                ; branch to label if carry flag is clear

    BEQ    label                ; branch to label if zero flag is set

    MOV    PC, #0               ; R15 = 0, branch to location zero

    BL     func                 ; subroutine call to function


func    .
    .
    MOV    PC, LR               ; R15=R14, return to instruction after the BL
    MOV    LR, PC               ; store the address of the instruction
                                ; after the next one into R14 ready to return
    LDR    PC, =func            ; load a 32-bit value into the program counter
```

3.3.2 List of branch instructions

`B`, `BL`	Branch, and Branch with Link. See *B, BL* on page A4-10.
`BLX`	Branch with Link and Exchange. See *BLX (1)* on page A4-16 and *BLX (2)* on page A4-18.
`BX`	Branch and Exchange Instruction Set. See *BX* on page A4-19.

3.4 Data-processing instructions

ARM has 16 data-processing instructions, shown in Table 3-2.

Table 3-2 Data-processing instructions

Opcode	Mnemonic	Operation	Action
0000	AND	Logical AND	Rd := Rn AND shifter_operand
0001	EOR	Logical Exclusive OR	Rd := Rn EOR shifter_operand
0010	SUB	Subtract	Rd := Rn - shifter_operand
0011	RSB	Reverse Subtract	Rd := shifter_operand - Rn
0100	ADD	Add	Rd := Rn + shifter_operand
0101	ADC	Add with Carry	Rd := Rn + shifter_operand + Carry Flag
0110	SBC	Subtract with Carry	Rd := Rn - shifter_operand - NOT(Carry Flag)
0111	RSC	Reverse Subtract with Carry	Rd := shifter_operand - Rn - NOT(Carry Flag)
1000	TST	Test	Update flags after Rn AND shifter_operand
1001	TEQ	Test Equivalence	Update flags after Rn EOR shifter_operand
1010	CMP	Compare	Update flags after Rn - shifter_operand
1011	CMN	Compare Negated	Update flags after Rn + shifter_operand
1100	ORR	Logical (inclusive) OR	Rd := Rn OR shifter_operand
1101	MOV	Move	Rd := shifter_operand (no first operand)
1110	BIC	Bit Clear	Rd := Rn AND NOT(shifter_operand)
1111	MVN	Move Not	Rd := NOT shifter_operand (no first operand)

Most data-processing instructions take two source operands, though Move and Move Not take only one. The compare and test instructions only update the condition flags. Other data-processing instructions store a result to a register and optionally update the condition flags as well.

Of the two source operands, one is always a register. The other is called a *shifter operand* and is either an immediate value or a register. If the second operand is a register value, it can have a shift applied to it.

`CMP`, `CMN`, `TST` and `TEQ` always update the condition code flags. The assembler automatically sets the S bit in the instruction for them, and the corresponding instruction with the S bit clear is not a data-processing instruction, but instead lies in one of the instruction extension spaces (see *Extending the instruction set* on page A3-27). The remaining instructions update the flags if an S is appended to the instruction mnemonic (which sets the S bit in the instruction). See *The condition code flags* on page A2-9 for more details.

3.4.1 Instruction encoding

```
<opcode1>{<cond>}{S}  <Rd>, <shifter_operand>
<opcode1> := MOV | MVN

<opcode2>{<cond>}  <Rn>, <shifter_operand>
<opcode2> := CMP | CMN | TST | TEQ

<opcode3>{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
<opcode3> := ADD | SUB | RSB | ADC | SBC | RSC | AND | BIC | EOR | ORR
```

31 28	27	26	25	24 21	20	19 16	15 12	11 0
cond	0	0	I	opcode	S	Rn	Rd	shifter_operand

I bit	Distinguishes between the immediate and register forms of `<shifter_operand>`.
S bit	Signifies that the instruction updates the condition codes.
Rn	Specifies the first source operand register.
Rd	Specifies the destination register.
shifter_operand	Specifies the second source operand. See *Addressing Mode 1 - Data-processing operands* on page A5-2 for details of the shifter operands.

3.4.2 List of data-processing instructions

`ADC`	Add with Carry. See *ADC* on page A4-4.
`ADD`	Add. See *ADD* on page A4-6.
`AND`	Logical AND. See *AND* on page A4-8.
`BIC`	Logical Bit Clear. See *BIC* on page A4-12.
`CMN`	Compare Negative. See *CMN* on page A4-23.
`CMP`	Compare. See *CMP* on page A4-25.
`EOR`	Logical EOR. See *EOR* on page A4-26.
`MOV`	Move. See *MOV* on page A4-56.
`MVN`	Move Negative. See *MVN* on page A4-68.
`ORR`	Logical OR. See *ORR* on page A4-70.
`RSB`	Reverse Subtract. See *RSB* on page A4-72.
`RSC`	Reverse Subtract with Carry. See *RSC* on page A4-74.
`SBC`	Subtract with Carry. See *SBC* on page A4-76.
`SUB`	Subtract. See *SUB* on page A4-98.
`TEQ`	Test Equivalence. See *TEQ* on page A4-106.
`TST`	Test. See *TST* on page A4-107.

3.5 Multiply instructions

ARM has two classes of Multiply instruction:

- normal, 32-bit result
- long, 64-bit result.

All Multiply instructions take two register operands as the input to the multiplier. The ARM processor does not directly support a multiply-by-constant instruction due to the efficiency of shift and add, or shift and reverse subtract instructions.

3.5.1 Normal multiply

There are two Multiply instructions that produce 32-bit results:

`MUL`	Multiplies the values of two registers together, truncates the result to 32 bits, and stores the result in a third register.
`MLA`	Multiplies the values of two registers together, adds the value of a third register, truncates the result to 32 bits, and stores the result in a fourth register. This can be used to perform multiply-accumulate operations.

Both Multiply instructions can optionally set the N (Negative) and Z (Zero) condition code flags. No distinction is made between signed and unsigned variants. Only the least significant 32 bits of the result are stored in the destination register, and the sign of the operands does not affect this value.

3.5.2 Long multiply

There are four Multiply instructions that produce 64-bit results (long multiply).

Two of the variants multiply the values of two registers together and store the 64-bit result in third and fourth registers. There are signed (`SMULL`) and unsigned (`UMULL`) variants. The signed variants produce a different result in the most significant 32 bits if either or both of the source operands is negative.

The remaining two variants multiply the values of two registers together, add the 64-bit value from the third and fourth registers and store the 64-bit result back into those registers (third and fourth). There are signed (`SMLAL`) and unsigned (`UMLAL`) variants. These instructions perform a long multiply and accumulate.

All four long multiply instructions can optionally set the N (Negative) and Z (Zero) condition code flags.

3.5.3 Examples

```
MUL    R4, R2, R1              ; Set R4 to value of R2 multiplied by R1
MULS   R4, R2, R1              ; R4 = R2 x R1, set N and Z flags
MLA    R7, R8, R9, R3          ; R7 = R8 x R9 + R3
SMULL  R4, R8, R2, R3          ; R4 = bits 0 to 31 of R2 x R3
                               ; R8 = bits 32 to 63 of R2 x R3
UMULL  R6, R8, R0, R1          ; R8, R6 = R0 x R1
UMLAL  R5, R8, R0, R1          ; R8, R5 = R0 x R1 + R8, R5
```

3.5.4 List of multiply instructions

`MLA`	Multiply Accumulate. See *MLA* on page A4-54.
`MUL`	Multiply. See *MUL* on page A4-66.
`SMLAL`	Signed Multiply Accumulate Long. See *SMLAL* on page A4-78.
`SMULL`	Signed Multiply Long. See *SMULL* on page A4-80.
`UMLAL`	Unsigned Multiply Accumulate Long. See *UMLAL* on page A4-109.
`UMULL`	Unsigned Multiply Long. See *UMULL* on page A4-111.

3.6 Miscellaneous arithmetic instructions

In addition to the normal data-processing and multiply instructions, versions 5 and above of the ARM architecture include a Count Leading Zeros (`CLZ`) instruction. This instruction returns the number of 0 bits at the most significant end of its operand before the first 1 bit is encountered (or 32 if its operand is zero). Two typical applications for this are:

- To determine how many bits the operand should be shifted left in order to *normalize* it, so that its most significant bit is 1. (This can be used in integer division routines.)
- To locate the highest priority bit in a bit mask.

3.6.1 Instruction encoding

```
CLZ{<cond>}  <Rd>, <Rm>
```

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	1	0	1	1	0	SBO	Rd	SBO	0	0	0	1	Rm

Rd	Specifies the destination register.
Rm	Specifies the operand register.

3.6.2 List of miscellaneous arithmetic instructions

`CLZ` Count Leading Zeros. See *CLZ* on page A4-22.

3.7 Status register access instructions

There are two instructions for moving the contents of a program status register to or from a general-purpose register. Both the CPSR and SPSR can be accessed.

Each status register is split into four 8-bit fields that can be individually written:

Bits[31:24]	The flags field.
Bits[23:16]	The status field.
Bits[15:8]	The extension field.
Bits[7:0]	The control field.

To date, the ARM architecture does not use the status and extension fields, and three bits are unused in the flags field. The four condition code flags occupy bits[31:28]. In E variants of architecture versions 5 and above, the Q flag occupies bit[27]. See *The Q flag* on page A10-5 for more information on the Q flag. The control field contains two interrupt disable bits, five processor mode bits, and the Thumb bit on ARM architecture version 5 and above and on T variants of ARM architecture version 4 (see *The T bit* on page A2-11).

The unused bits of the status registers might be used in future ARM architectures, and must not be modified by software. Therefore, a read-modify-write strategy must be used to update the value of a status register to ensure future compatibility.

The status registers are readable to allow the read part of the read-modify-write operation, and to allow all processor state to be preserved (for instance, during process context switches).

The status registers are writable to allow the write part of the read-modify-write operation, and allow all processor state to be restored.

3.7.1 CPSR value

Altering the value of the CPSR has three uses:

- sets the value of the condition code flags (and of the Q flag when it exists) to a known value
- enables or disable interrupts
- changes processor mode (for instance, to initialize stack pointers).

Note

The T bit must not be changed directly by writing to the CPSR, but only via the `BX` instruction, and in the implicit SPSR to CPSR moves in instructions designed for exception return. Attempts to enter or leave Thumb state by directly altering the T bit can have UNPREDICTABLE consequences.

3.7.2 Examples

These examples assume that the ARM processor is already in a privileged mode. If the ARM processor starts in User mode, only the flag update has any effect.

```
MRS     R0, CPSR                      ; Read the CPSR
BIC     R0, R0, #0xF0000000           ; Clear the N, Z, C and V bits
MSR     CPSR_f, R0                    ; Update the flag bits in the CPSR
                                      ; N, Z, C and V flags now all clear

MRS     R0, CPSR                      ; Read the CPSR
ORR     R0, R0, #0x80                 ; Set the interrupt disable bit
MSR     CPSR_c, R0                    ; Update the control bits in the CPSR
                                      ; interrupts (IRQ) now disabled

MRS     R0, CPSR                      ; Read the CPSR
BIC     R0, R0, #0x1F                 ; Clear the mode bits
ORR     R0, R0, #0x11                 ; Set the mode bits to FIQ mode
MSR     CPSR_c, R0                    ; Update the control bits in the CPSR
                                      ; now in FIQ mode
```

3.7.3 List of status register access instructions

`MRS`	Move PSR to General-purpose Register. See *MRS* on page A4-60.
`MSR`	Move General-purpose Register to PSR. See *MSR* on page A4-62.

3.8 Load and store instructions

The ARM architecture supports two broad types of instruction which load or store the value of a single register from or to memory:

- The first type can load or store a 32-bit word or an 8-bit unsigned byte.
- The second type can load or store a 16-bit unsigned halfword, and can load and sign extend a 16-bit halfword or an 8-bit byte. This type of instruction is only available in ARM architecture version 4 and above.

3.8.1 Addressing modes

In both types of instruction, the addressing mode is formed from two parts:

- the base register
- the offset.

The base register can be any one of the general-purpose registers (including the PC, which allows PC-relative addressing for position-independent code).

The offset takes one of three formats:

Immediate The offset is an unsigned number that can be added to or subtracted from the base register. Immediate offset addressing is useful for accessing data elements that are a fixed distance from the start of the data object, such as structure fields, stack offsets and input/output registers.

For the word and unsigned byte instructions, the immediate offset is a 12-bit number. For the halfword and signed byte instructions, it is an 8-bit number.

Register The offset is a general-purpose register (not the PC), that can be added to or subtracted from the base register. Register offsets are useful for accessing arrays or blocks of data.

Scaled register The offset is a general-purpose register (not the PC) shifted by an immediate value, then added to or subtracted from the base register. The same shift operations used for data-processing instructions can be used (Logical Shift Left, Logical Shift Right, Arithmetic Shift Right and Rotate Right), but Logical Shift Left is the most useful as it allows an array indexed to be scaled by the size of each array element.

Scaled register offsets are only available for the word and unsigned byte instructions.

As well as the three types of offset, the offset and base register are used in three different ways to form the memory address. The addressing modes are described as follows:

Offset The base register and offset are added or subtracted to form the memory address.

Pre-indexed The base register and offset are added or subtracted to form the memory address. The base register is then updated with this new address, to allow automatic indexing through an array or memory block.

Post-indexed The value of the base register alone is used as the memory address. The base register and offset are added or subtracted and this value is stored back in the base register, to allow automatic indexing through an array or memory block.

3.8.2 Load and Store word or unsigned byte instructions

Load instructions load a single value from memory and write it to a general-purpose register.

Store instructions read a value from a general-purpose register and store it to memory.

Load and Store instructions have a single instruction format:

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

31 28	27 26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0 1	I	P	U	B	W	L	Rn	Rd	addressing_mode_specific

I, P, U, W Are bits that distinguish between different types of `<addressing_mode>`.

L bit Distinguishes between a Load (L==1) and a Store instruction (L==0).

B bit Distinguishes between an unsigned byte (B==1) and a word (B==0) access.

Rn Specifies the base register used by `<addressing_mode>`.

Rd Specifies the register whose contents are to be loaded or stored.

3.8.3 Load and Store Halfword and Load Signed Byte

Load instructions load a single value from memory and write it to a general-purpose register.

Store instructions read a value from a general-purpose register and store it to memory.

Load and Store Halfword and Load Signed Byte instructions have a single instruction format:

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

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0 0 0	P	U	I	W	L	Rn	Rd	addr_mode	1	S	H	1	addr_mode

addr_mode Are addressing-mode-specific bits.

I, P, U, W Are bits that specify the type of addressing mode (see *Addressing Mode 3 - Miscellaneous Loads and Stores* on page A5-34).

L bit Distinguishes between a Load (L==1) and a Store instruction (L==0).

S bit Distinguishes between a signed (S==1) and an unsigned (S==0) halfword access. If the L bit is zero and S bit is one, the instruction is UNPREDICTABLE.

H bit	Distinguishes between a halfword (H==1) and a signed byte (H==0) access. If the S bit and H bit are both zero, this instruction encodes a `SWP` or Multiply instruction.
Rn	Specifies the base register used by the addressing mode.
Rd	Specifies the register whose contents are to be loaded or stored.

3.8.4 Examples

```
LDR     R1, [R0]                ; Load R1 from the address in R0
LDR     R8, [R3, #4]            ; Load R8 from the address in R3 + 4
LDR     R12, [R13, #-4]         ; Load R12 from R13 - 4
STR     R2, [R1, #0x100]        ; Store R2 to the address in R1 + 0x100

LDRB    R5, [R9]                ; Load byte into R5 from R9
                                ;  (zero top 3 bytes)
LDRB    R3, [R8, #3]            ; Load byte to R3 from R8 + 3
                                ;  (zero top 3 bytes)
STRB    R4, [R10, #0x200]       ; Store byte from R4 to R10 + 0x200

LDR     R11, [R1, R2]           ; Load R11 from the address in R1 + R2
STRB    R10, [R7, -R4]          ; Store byte from R10 to addr in R7 - R4

LDR     R11, [R3, R5, LSL #2]   ; Load R11 from R3 + (R5 x 4)
LDR     R1, [R0, #4]!           ; Load R1 from R0 + 4, then R0 = R0 + 4
STRB    R7, [R6, #-1]!          ; Store byte from R7 to R6 - 1,
                                ;  then R6 = R6 - 1

LDR     R3, [R9], #4            ; Load R3 from R9, then R9 = R9 + 4
STR     R2, [R5], #8            ; Store R2 to R5, then R5 = R5 + 8

LDR     R0, [PC, #40]           ; Load R0 from PC + 0x40 (= address of
                                ;  the LDR instruction + 8 + 0x40)
LDR     R0, [R1], R2            ; Load R0 from R1, then R1 = R1 + R2

LDRH    R1, [R0]                ; Load halfword to R1 from R0
                                ;  (zero top 2 bytes)
LDRH    R8, [R3, #2]            ; Load halfword into R8 from R3 + 2
LDRH    R12, [R13, #-6]         ; Load halfword into R12 from R13 - 6
STRH    R2, [R1, #0x80]         ; Store halfword from R2 to R1 + 0x80

LDRSH   R5, [R9]                ; Load signed halfword to R5 from R9
LDRSB   R3, [R8, #3]            ; Load signed byte to R3 from R8 + 3
LDRSB   R4, [R10, #0xC1]        ; Load signed byte to R4 from R10 + 0xC1

LDRH    R11, [R1, R2]           ; Load halfword into R11 from address
                                ;  in R1 + R2
STRH    R10, [R7, -R4]          ; Store halfword from R10 to R7 - R4

LDRSH   R1, [R0, #2]!           ; Load signed halfword R1 from R0 + 2,
                                ;  then R0 = R0 + 2
```

```
LDRSB   R7, [R6, #-1]!          ; Load signed byte to R7 from R6 - 1,
                                ;  then R6 = R6 - 1
LDRH    R3, [R9], #2            ; Load halfword to R3 from R9,
                                ;  then R9 = R9 + 2
STRH    R2, [R5], #8            ; Store halfword from R2 to R5,
                                ;  then R5 = R5 + 8
```

3.8.5 List of load and store instructions

`LDR`	Load Word. See *LDR* on page A4-37.
`LDRB`	Load Byte. See *LDRB* on page A4-40.
`LDRBT`	Load Byte with User Mode Privilege. See *LDRBT* on page A4-42.
`LDRH`	Load Unsigned Halfword. See *LDRH* on page A4-44.
`LDRSB`	Load Signed Byte. See *LDRSB* on page A4-46.
`LDRSH`	Load Signed Halfword. See *LDRSH* on page A4-48.
`LDRT`	Load Word with User Mode Privilege. See *LDRT* on page A4-50.
`STR`	Store Word. See *STR* on page A4-88.
`STRB`	Store Byte. See *STRB* on page A4-90.
`STRBT`	Store Byte with User Mode Privilege. See *STRBT* on page A4-92.
`STRH`	Store Halfword. See *STRH* on page A4-94.
`STRT`	Store Word with User Mode Privilege. See *STRT* on page A4-96.

3.9 Load and Store Multiple instructions

Load Multiple instructions load a subset, or possibly all, of the general-purpose registers from memory.

Store Multiple instructions store a subset, or possibly all, of the general-purpose registers to memory.

Load and Store Multiple instructions have a single instruction format:

```
LDM{<cond>}<addressing_mode>  Rn{!}, <registers>{^}
STM{<cond>}<addressing_mode>  Rn{!}, <registers>{^}
```

where:

```
<addressing_mode> = IA | IB | DA | DB | FD | FA | ED | EA
```

31 28	27 26 25	24	23	22	21	20	19 16	15 0
cond	1 0 0	P	U	S	W	L	Rn	register list

register list The list of `<registers>` has one bit for each general-purpose register. Bit 0 is for R0, and bit 15 is for R15 (the PC).

The register syntax list is an opening bracket, followed by a comma-separated list of registers, followed by a closing bracket. A sequence of consecutive registers can be specified by separating the first and last registers in the range with a minus sign.

P, U, and W bits These distinguish between the different types of addressing mode (see *Addressing Mode 4 - Load and Store Multiple* on page A5-48).

S bit For `LDM`s that load the PC, the S bit indicates that the CPSR is loaded from the SPSR after all the registers have been loaded. For all `STM`s, and `LDM`s that do not load the PC, it indicates that when the processor is in a privileged mode, the User mode banked registers are transferred and not the registers of the current mode.

L bit This distinguishes between a Load (L==1) and a Store (L==0) instruction.

Rn This specifies the base register used by the addressing mode.

3.9.1 Examples

```
    STMFD    R13!, {R0 - R12, LR}
    LDMFD    R13!, {R0 - R12, PC}
    LDMIA    R0, {R5 - R8}
    STMDA    R1!, {R2, R5, R7 - R9, R11}
```

3.9.2 List of Load and Store Multiple instructions

`LDM`	Load Multiple. See *LDM (1)* on page A4-30.
`LDM`	User Registers Load Multiple. See *LDM (2)* on page A4-32.
`LDM`	Load Multiple with Restore CPSR. See *LDM (3)* on page A4-34.
`STM`	Store Multiple. See *STM(1)* on page A4-84.
`STM`	User Registers Store Multiple. See *STM (2)* on page A4-86.

3.10 Semaphore instructions

The ARM instruction set has two semaphore instructions:

- Swap (`SWP`)
- Swap Byte (`SWPB`).

These instructions are provided for process synchronization. Both instructions generate an atomic load and store operation, allowing a memory semaphore to be loaded and altered without interruption.

`SWP` and `SWPB` have a single addressing mode, whose address is the contents of a register. Separate registers are used to specify the value to store and the destination of the load. If the same register is specified for both of these, `SWP` exchanges the value in the register and the value in memory.

The semaphore instructions do not provide a compare and conditional write facility. If wanted, this must be done explicitly.

3.10.1 Examples

```
SWP    R12, R10, [R9]       ; load R12 from address R9 and
                            ; store R10 to address R9

SWPB   R3, R4, [R8]         ; load byte to R3 from address R8 and
                            ; store byte from R4 to address R8

SWP    R1, R1, [R2]         ; Exchange value in R1 and address in R2
```

3.10.2 List of semaphore instructions

`SWP`	Swap. See *SWP* on page A4-102.
`SWPB`	Swap Byte. See *SWPB* on page A4-104.

3.11 Exception-generating instructions

The ARM instruction set provides two types of instruction whose main purpose is to cause a processor exception to occur:

- The Software Interrupt (`SWI`) instruction is used to cause a SWI exception to occur (see *Software Interrupt exception* on page A2-16). This is the main mechanism in the ARM instruction set by which User mode code can make calls to privileged Operating System code.
- The Breakpoint (`BKPT`) instruction is used for software breakpoints in ARM architecture versions 5 and above. Its default behavior is to cause a Prefetch Abort exception to occur (see *Prefetch Abort (instruction fetch memory abort)* on page A2-16). A debug monitor program which has previously been installed on the Prefetch Abort vector can handle this exception.

  If debug hardware is present in the system, it is allowed to override this default behavior. Details of whether and how this happens are IMPLEMENTATION DEFINED.

3.11.1 Instruction encodings

```
SWI{<cond>}  <immed_24>
```

31 28	27 26 25 24	23 0
cond	1 1 1 1	immed_24

```
BKPT  <immediate>
```

31 28	27 26 25 24 23 22 21 20	19 8	7 4	3 0
1 1 1 0	0 0 0 1 0 0 1 0	immed	0 1 1 1	immed

In both `SWI` and `BKPT`, the immediate fields of the instruction are ignored by the ARM processor. The SWI or Prefetch Abort handler can optionally be written to load the instruction that caused the exception and extract these fields. This allows them to be used to communicate extra information about the Operating System call or breakpoint to the handler.

3.11.2 List of exception-generating instructions

`BKPT`	Breakpoint. See *BKPT* on page A4-14.
`SWI`	Software Interrupt. See *SWI* on page A4-100.

3.12 Coprocessor instructions

The ARM instruction set provides three types of instruction for communicating with coprocessors. These allow:

- the ARM processor to initiate a coprocessor data processing operation
- ARM registers to be transferred to and from coprocessor registers
- the ARM processor to generate addresses for the coprocessor Load and Store instructions.

The instruction set distinguishes up to 16 coprocessors with a 4-bit field in each coprocessor instruction, so each coprocessor is assigned a particular number.

Note

One coprocessor can use more than one of the 16 numbers if a large coprocessor instruction set is required.

Coprocessors execute the same instruction stream as ARM, ignoring ARM instructions and coprocessor instructions for other coprocessors. Coprocessor instructions that cannot be executed by coprocessor hardware cause an undefined instruction trap, allowing software emulation of coprocessor hardware.

A coprocessor can partially execute an instruction and then cause an exception. This is useful for handling run-time-generated exceptions, like divide-by-zero or overflow. However, the partial execution is internal to the coprocessor and is not visible to the ARM processor. As far as the ARM processor is concerned, the instruction is held at the start of its execution and completes without exception if allowed to begin execution. Any decision on whether to execute the instruction or cause an exception is taken within the coprocessor before the ARM processor is allowed to start executing the instruction.

Not all fields in coprocessor instructions are used by the ARM processor. Coprocessor register specifiers and opcodes are defined by individual coprocessors. Therefore, only generic instruction mnemonics are provided for coprocessor instructions. Assembler macros can be used to transform custom coprocessor mnemonics into these generic mnemonics, or to regenerate the opcodes manually.

3.12.1 Examples

```
CDP    p5, 2, c12, c10, c3, 4   ; Coproc 5 data operation
                                ; opcode 1 = 2, opcode 2 = 4
                                ; destination register is 12
                                ; source registers are 10 and 3

MRC    p15, 5, R4, c0, c2, 3    ; Coproc 15 transfer to ARM register
                                ; opcode 1 = 5, opcode 2 = 3
                                ; ARM destination register = R4
                                ; coproc source registers are 0 and 2

MCR    p14, 1, R7, c7, c12, 6   ; ARM register transfer to Coproc 14
                                ; opcode 1 = 1, opcode 2 = 6
                                ; ARM source register = R7
                                ; coproc dest registers are 7 and 12
```

```
LDC    p6, CR1, [R4]            ; Load from memory to coprocessor 6
                                ; ARM register 4 contains the address
                                ; Load to CP reg 1

LDC    p6, CR4, [R2, #4]        ; Load from memory to coprocessor 6
                                ; ARM register R2 + 4 is the address
                                ; Load to CP reg 4

STC    p8, CR8, [R2, #4]!       ; Store from coprocessor 8 to memory
                                ; ARM register R2 + 4 is the address
                                ; after the transfer R2 = R2 + 4
                                ; Store from CP reg 8

STC    p8, CR9, [R2], #-16      ; Store from coprocessor 8 to memory
                                ; ARM register R2 holds the address
                                ; after the transfer R2 = R2 - 16
                                ; Store from CP reg 9
```

3.12.2 List of coprocessor instructions

`CDP`	Coprocessor Data Operations. See *CDP* on page A4-20.
`LDC`	Load Coprocessor Register. See *LDC* on page A4-28.
`MCR`	Move to Coprocessor from ARM Register. See *MCR* on page A4-52.
`MRC`	Move to ARM Register from Coprocessor. See *MRC* on page A4-58.
`STC`	Store Coprocessor Register. See *STC* on page A4-82.

Note

Coprocessor instructions are not implemented in ARM architecture version 1.

3.13 Extending the instruction set

Successive versions of the ARM architecture have extended the instruction set in a number of areas. This section describes the six areas where extensions have occurred, and where further extensions might occur in the future:

- *Undefined instruction space* on page A3-28
- *Arithmetic instruction extension space* on page A3-29
- *Control instruction extension space* on page A3-30
- *Load/store instruction extension space* on page A3-32
- *Coprocessor instruction extension space* on page A3-33
- *Unconditional instruction extension space* on page A3-34.

Instructions in these areas which have not yet been allocated a meaning are either UNDEFINED or UNPREDICTABLE. To determine which, use the following rules:

1. The *decode bits* of an instruction are defined to be bits[27:20] and bits[7:4].

   In ARM architecture version 5 and above, the result of ANDing bits[31:28] together is also a decode bit. This bit determines whether the condition field is 0b1111, which is used in ARM architecture version 5 and above to encode various instructions which can only be executed unconditionally. See *Condition code 0b1111* on page A3-5 and *Unconditional instruction extension space* on page A3-34 for more information.

2. If the decode bits of an instruction are equal to those of a defined instruction, but the whole instruction is not a defined instruction, then the instruction is UNPREDICTABLE.

   For example, suppose an instruction has:
   - bits[31:28] not equal to 0b1111
   - bits[27:20] equal to 0b00010000
   - bits[7:4] equal to 0b0000

   but where:
   - bit[11] of the instruction is 1.

   Here, the instruction is in the control instruction extension space and has the same decode bits as an `MRS` instruction, but is not a valid `MRS` instruction because bit[11] of an `MRS` instruction should be zero. Using the above rule, this instruction is UNPREDICTABLE.

3. In ARM architecture version 4 and above, if the decode bits of an instruction are not equal to those of any defined instruction, then the instruction is UNDEFINED.

4. In ARM architecture version 3 and below, if the decode bits of an instruction are not equal to those of any defined instruction, then the instruction is:
   - UNDEFINED if it is in the undefined instruction space
   - UNPREDICTABLE if it is in any of the other five areas.

Each of rules 2 to 4 above applies separately to each ARM architecture version. As a result, the status of an instruction might differ between architecture versions. Usually, this happens because an instruction which was UNPREDICTABLE or UNDEFINED in an earlier architecture version becomes a defined instruction in a later version.

3.13.1 Undefined instruction space

Instructions with the following opcodes are undefined instruction space:

```
opcode[27:25] = 0b011
opcode[4] = 1
```

31 28	27	26	25	24 5	4	3 0
cond	0	1	1	x x x x x x x x x x x x x x x x x x x x	1	x x x x

The meaning of instructions in the undefined instruction space is UNDEFINED on all versions of the ARM architecture.

In general, undefined instructions might be used to extend the ARM instruction set in the future. However, it is intended that instructions with the following encoding will not be used for this:

31 28	27	26	25	24	23	22	21	20	19 8	7	6	5	4	3	2	1	0
cond	0	1	1	1	1	1	1	1	x x x x x x x x x x x x	1	1	1	1	x	x	x	x

If a programmer wants to use an undefined instruction for software purposes, with minimal risk that future hardware will treat it as a defined instruction, one of the instructions with this encoding must be used.

3.13.2 Arithmetic instruction extension space

Instructions with the following opcodes are the arithmetic instruction extension space:

```
opcode[27:24]  == 0b0000
opcode[7:4]    == 0b1001
opcode[31:28]  != 0b1111  /* Only required for version 5 and above */
```

The field names given are guidelines suggested to simplify implementation.

31 28	27 26 25 24	23 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 0	op1	Rn	Rd	Rs	1 0 0 1	Rm

Table 3-3 summarizes the instructions that have already been allocated in this area.

Table 3-3 Arithmetic instruction space

Instructions	op1	Architecture versions
`MUL`, `MULS`	000S	Version 2 and above
`MLA`, `MLAS`	001S	Version 2 and above
`UMULL`, `UMULLS`	100S	All M variants
`UMLAL`, `UMLALS`	101S	All M variants
`SMULL`, `SMULLS`	110S	All M variants
`SMLAL`, `SMLALS`	111S	All M variants

3.13.3 Control instruction extension space

Instructions with the following opcodes are the control instruction space.

```
opcode[27:26] == 0b00
opcode[24:23] == 0b10
opcode[20]    == 0
opcode[31:28] != 0b1111  /* Only required for version 5 and above */
```

and not:

```
opcode[25] == 0
opcode[7]  == 1
opcode[4]  == 1
```

The field names given are guidelines suggested to simplify implementation.

31 28	27	26	25	24	23	22 21	20	19 16	15 12	11 8	7	6 5	4	3 0
cond	0	0	0	1	0	op1	0	Rn	Rd	Rs	op2		0	Rm
cond	0	0	0	1	0	op1	0	Rn	Rd	Rs	0	op2	1	Rm
cond	0	0	1	1	0	op1	0	Rn	Rd	rotate_imm	immed_8			

Table 3-4 summarizes the instructions that have already been allocated in this area.

Table 3-4 Control extension space instructions

Instruction	Bit[25]	Bits[7:4]	op1	Architecture versions
`MRS`	0	0000	x0	Version 3 and above
`MSR` (register form)	0	0000	x1	Version 3 and above
`BX`	0	0001	01	Version 5 and above, plus T variants of version 4
`CLZ`	0	0001	11	Version 5 and above
`BLX` (register form)	0	0011	01	Version 5 and above
`QADD`	0	0101	00	E variants of version 5 and above
`QSUB`	0	0101	01	E variants of version 5 and above
`QDADD`	0	0101	10	E variants of version 5 and above
`QDSUB`	0	0101	11	E variants of version 5 and above

Table 3-4 Control extension space instructions (continued)

Instruction	Bit[25]	Bits[7:4]	op1	Architecture versions
`BKPT`	0	0111	01	Version 5 and above
`SMLA<x><y>`	0	1yx0	00	E variants of version 5 and above
`SMLAW<y>`	0	1y00	01	E variants of version 5 and above
`SMULW<y>`	0	1y10	01	E variants of version 5 and above
`SMLAL<x><y>`	0	1yx0	10	E variants of version 5 and above
`SMUL<x><y>`	0	1yx0	11	E variants of version 5 and above
`MSR` (immediate form)	1	xxxx	x1	Version 3 and above

3.13.4 Load/store instruction extension space

Instructions with the following opcodes are the load/store instruction extension space:

```
opcode[27:25] == 0b000
opcode[7]     == 1
opcode[4]     == 1
opcode[31:28] != 0b1111 /* Only required for version 5 and above */
```

and not:

```
opcode[24]  == 0
opcode[6:5] == 0
```

The field names given are guidelines suggested to simplify implementation.

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6 5	4	3 0
cond	0	0	0	P	U	B	W	L	Rn	Rd	Rs	1	op1	1	Rm

Table 3-5 summarizes the instructions that have already been allocated in this area.

Table 3-5 Load/store instructions

Instruction	Bits[24:20]					op1		Architecture versions
`SWP/SWPB`	1	0	B	0	0	0	0	Version 3 and above, plus ARMv2a
`STRH`	P	U	I	W	0	0	1	Version 4 and above
`LDRD`	P	U	I	W	0	1	0	E variants of version 5 and above, except v5TExP
`STRD`	P	U	I	W	0	1	1	E variants of version 5 and above, except v5TExP
`LDRH`	P	U	I	W	1	0	1	Version 4 and above
`LDRSB`	P	U	I	W	1	1	0	Version 4 and above
`LDRSH`	P	U	I	W	1	1	1	Version 4 and above

3.13.5 Coprocessor instruction extension space

Instructions with the following opcodes are the coprocessor instruction extension space:

```
opcode[27:23]   == 0b11000
opcode[21]      == 0
```

The field names given are guidelines suggested to simplify implementation.

31 28	27 26 25 24 23	22	21	20	19 16	15 12	11 8	7 0
cond	1 1 0 0 0	x	0	x	Rn	CRd	cp_num	offset

In ARM architecture version 3 and below, all instructions in the coprocessor instruction extension space are UNPREDICTABLE.

In all variants of architecture version 4, and in non-E variants of architecture 5, all instructions in the coprocessor instruction extension space are UNDEFINED. It is IMPLEMENTATION DEFINED how an ARM processor achieves this. The options are:

- The ARM processor might take the undefined instruction trap directly.
- The ARM processor might require attached coprocessors not to respond to such instructions. This causes the undefined instruction trap to be taken (see *Undefined Instruction exception* on page A2-15).

In E variants of architecture version 5, instructions in the coprocessor instruction extension space are treated as follows:

- Instructions with bit[22] == 0 are UNDEFINED and are handled in precisely the same way as described above for non-E variants.
- Instructions with bit[22] ==1 are the `MCRR` and `MRRC` instructions described in Chapter A10 *Enhanced DSP Extension*.

3.13.6 Unconditional instruction extension space

In ARM architecture version 5 and above, instructions with the following opcode are the unconditional instruction space:

```
opcode[31:28] == 0b1111
```

31 30 29 28	27 … 20	19 … 8	7 … 4	3 … 0
1 1 1 1	opcode1	x x x x x x x x x x x x	opcode2	x x x x

Table 3-6 summarizes the instructions that have already been allocated in this area.

Table 3-6 Unconditional instruction extension space

Instruction	opcode1	opcode2	Architecture versions
`PLD`	0 1 I 1 U 1 0 1	x x x x	E variants of version 5 and above, except v5TExP
`BLX` (address form)	1 0 1 x x x x x	x x x x	Version 5 and above
`STC2`	1 1 0 x x x x 0	x x x x	Version 5 and above
`LDC2`	1 1 0 x x x x 1	x x x x	Version 5 and above
`CDP2`	1 1 1 0 x x x x	x x x 0	Version 5 and above
`MCR2`	1 1 1 0 x x x 0	x x x 1	Version 5 and above
`MRC2`	1 1 1 0 x x x 1	x x x 1	Version 5 and above

Chapter A4
ARM Instructions

This chapter describes the syntax and usage of every ARM instruction, in the sections:

- *Alphabetical list of ARM instructions* on page A4-2
- *ARM instructions and architecture versions* on page A4-113.

4.1 Alphabetical list of ARM instructions

Every ARM instruction is listed on the following pages. Each instruction description shows:

- the instruction encoding
- the instruction syntax
- the version of the ARM architecture where the instruction is valid
- any exceptions that apply
- an example in pseudo-code of how the instruction operates
- notes on usage and special cases.

4.1.1 General notes

These notes explain the types of information and abbreviations used on the instruction pages.

Syntax abbreviations

The following abbreviations are used in the instruction pages:

`immed_`*`n`* This is an immediate value, where `n` is the number of bits. For example, an 8-bit immediate value is represented by:

`immed_8`

`offset_`*`n`* This is an offset value, where `n` is the number of bits. For example, an 8-bit offset value is represented by:

`offset_8`

The same construction is used for signed offsets. For example, an 8-bit signed offset is represented by:

`signed_offset_8`

Encoding diagram and assembler syntax

For the conventions used, see *Assembler syntax descriptions* on page Preface-xiii.

Architecture versions

This gives details of architecture versions where the instruction is valid. For details, see *Architecture versions and variants* on page Preface-v.

Exceptions

This gives details of which exceptions can occur during the execution of the instruction. Prefetch Abort is not listed in general, both because it can occur for any instruction and because if an abort occurred during instruction fetch, the instruction bit pattern is not known. (Prefetch Abort is however listed for `BKPT`, since it can generate a Prefetch Abort exception without these considerations applying.)

Operation

This gives a pseudo-code description of what the instruction does. For details of conventions used in this pseudo-code, see *Pseudo-code descriptions of instructions* on page Preface-xii.

Information on usage

Usage sections are included where appropriate to supply suggestions and other information about how to use the instruction effectively.

4.1.2 ADC

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	0 1 0 1	S	Rn	Rd	shifter_operand

The `ADC` (Add with Carry) instruction adds the value of `<shifter_operand>` and the Carry flag to the value of `<Rn>` and stores the result in `<Rd>`. The condition code flags are optionally updated, based on the result.

Syntax

`ADC{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>`

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`S` Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the `N` and `Z` flags are set according to the result of the addition, and the `C` and `V` flags are set according to whether the addition generated a carry (unsigned overflow) and a signed overflow, respectively. The rest of the CPSR is unchanged.
- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>` Specifies the destination register of the instruction.

`<Rn>` Specifies the register that contains the first operand for the addition.

`<shifter_operand>`

Specifies the second operand for the addition. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `ADC`. Instead, see *Extending the instruction set* on page A3-27 to determine which instruction it is.

Architecture version

All

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Rd = Rn + shifter_operand + C Flag
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = CarryFrom(Rn + shifter_operand + C Flag)
        V Flag = OverflowFrom(Rn + shifter_operand + C Flag)
```

Usage

`ADC` is used to synthesize multi-word addition. If register pairs R0, R1 and R2, R3 hold 64-bit values (where R0 and R2 hold the least significant words) the following instructions leave the 64-bit sum in R4, R5:

```
    ADDS R4,R0,R2
    ADC  R5,R1,R3
```

If the second instruction is changed from:

```
    ADC  R5,R1,R3
```

to:

```
    ADCS R5,R1,R3
```

the resulting values of the flags indicate:

`N`	The 64-bit addition produced a negative result.
`C`	An unsigned overflow occurred.
`V`	A signed overflow occurred.
`Z`	The most significant 32 bits are all zero.

The following instruction produces a single-bit Rotate Left with Extend operation (33-bit rotate through the Carry flag) on R0:

```
    ADCS R0,R0,R0
```

See *Data-processing operands - Rotate right with extend* on page A5-17 for information on how to perform a similar rotation to the right.

4.1.3 ADD

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	0 1 0 0	S	Rn	Rd	shifter operand

The `ADD` instruction adds the value of `<shifter_operand>` to the value of register `<Rn>`, and stores the result in the destination register `<Rd>`. The condition code flags are optionally updated, based on the result.

Syntax

```
ADD{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

where:

`<cond>` Is the condition under which the instruction is executed. *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`S` Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the `N` and `Z` flags are set according to the result of the addition, and the `C` and `V` flags are set according to whether the addition generated a carry (unsigned overflow) and a signed overflow, respectively. The rest of the CPSR is unchanged.
- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>` Specifies the destination register of the instruction.

`<Rn>` Specifies the register that contains the first operand for the addition.

`<shifter_operand>`

Specifies the second operand for the addition. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `ADD`. Instead, see *Extending the instruction set* on page A3-27 to determine which instruction it is.

Architecture version

All

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Rd = Rn + shifter_operand
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = CarryFrom(Rn + shifter_operand)
        V Flag = OverflowFrom(Rn + shifter_operand)
```

Usage

The `ADD` instruction is used to add two values together to produce a third.

To increment a register value in `Rx` use:

```
ADD Rx, Rx, #1
```

Constant multiplication of `Rx` by 2^n+1 into `Rd` can be performed with:

```
ADD Rd, Rx, Rx, LSL #n
```

To form a PC-relative address use:

```
ADD Rs, PC, #offset
```

where the offset must be the difference between the required address and the address held in the PC, where the PC is the address of the `ADD` instruction itself plus 8 bytes.

4.1.4 AND

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	0 0 0 0	S	Rn	Rd	shifter_operand

The `AND` instruction performs a bitwise AND of the value of register `<Rn>` with the value of `<shifter_operand>`, and stores the result in the destination register `<Rd>`. The condition code flags are optionally updated, based on the result.

Syntax

```
AND{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`S` Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the N and Z flags are set according to the result of the operation, and the C flag is set to the carry output bit generated by the shifter (see *Addressing Mode 1 - Data-processing operands* on page A5-2). The V flag and the rest of the CPSR are unaffected.
- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>` Specifies the destination register of the instruction.

`<Rn>` Specifies the register that contains the first operand for the operation.

`<shifter_operand>`

Specifies the second operand for the operation. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `AND`. Instead, see *Extending the instruction set* on page A3-27 to determine which instruction it is.

Architecture version

All

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Rd = Rn AND shifter_operand
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

Usage

`AND` is most useful for extracting a field from a register, by ANDing the register with a mask value that has 1s in the field to be extracted, and 0s elsewhere.

4.1.5 B, BL

31 … 28	27	26	25	24	23 … 0
cond	1	0	1	L	signed_immed_24

The `B` (Branch) and `BL` (Branch and Link) instructions cause a branch to a target address, and provide both conditional and unconditional changes to program flow.

Syntax

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

where:

`L` Causes the `L` bit (bit 24) in the instruction to be set to 1. The resulting instruction stores a return address in the link register (R14). If L is omitted, the L bit is 0 and the instruction simply branches without storing a return address.

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`<target_address>`
Specifies the address to branch to. The branch target address is calculated by:

1. Sign-extending the 24-bit signed (two's complement) immediate to 32 bits.
2. Shifting the result left two bits.
3. Adding this to the contents of the PC, which contains the address of the branch instruction plus 8.

The instruction can therefore specify a branch of approximately ±32MB.

Architecture version

All

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    if L == 1 then
        LR = address of the instruction after the branch instruction
    PC = PC + (SignExtend(signed_immed_24) << 2)
```

Usage

The `BL` instruction is used to perform a subroutine call. The return from subroutine is achieved by copying the LR to the PC. Typically, this is done by one of the following methods:

- Executing a `BX R14` instruction, on architecture versions that support that instruction.
- Executing a `MOV PC,R14` instruction.
- Storing a group of registers and R14 to the stack on subroutine entry, using an instruction of the form:

  ```
  STMFD R13!,{<registers>,R14}
  ```

  and then restoring the register values and returning with an instruction of the form:

  ```
  LDMFD R13!,{<registers>,PC}
  ```

To calculate the correct value of signed_immed_24, the assembler (or other toolkit component) needs to:

1. Form the base address for this branch instruction. This is the address of the instruction, plus 8. In other words, this base address is equal to the PC value used by the instruction.
2. Subtract the base address from the target address to form a byte offset. This offset is always a multiple of four, because all ARM instructions are word-aligned.
3. If the byte offset is outside the range −33554432 to +33554428, use an alternative code-generation strategy or produce an error as appropriate.
4. Otherwise, set the signed_immed_24 field of the instruction to bits{25:2] of the byte offset.

Notes

Memory bounds	Branching backwards past location zero and forwards over the end of the 32-bit address space is UNPREDICTABLE.

4.1.6 BIC

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	1 1 1 0	S	Rn	Rd	shifter_operand

The `BIC` (Bit Clear) instruction performs a bitwise AND of the value of register `<Rn>` with the complement of the value of `<shifter_operand>`, and stores the result in the destination register `<Rd>`. The condition code flags are optionally updated, based on the result.

Syntax

```
BIC{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`S` Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the N and Z flags are set according to the result of the operation, and the C flag is set to the carry output bit generated by the shifter (see *Addressing Mode 1 - Data-processing operands* on page A5-2). The V flag and the rest of the CPSR are unaffected.
- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>` Specifies the destination register of the instruction.

`<Rn>` Specifies the register that contains the first operand for the operation.

`<shifter_operand>`

Specifies the second operand for the operation. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `BIC`. Instead, see *Extending the instruction set* on page A3-27 to determine which instruction it is.

Architecture version

All

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Rd = Rn AND NOT shifter_operand
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

Usage

`BIC` can be used to clear selected bits in a register. For each bit, `BIC` with 1 clears the bit, and `BIC` with 0 leaves it unchanged.

4.1.7 BKPT

31 28	27 26 25 24 23 22 21 20	19 8	7 4	3 0
1 1 1 0	0 0 0 1 0 0 1 0	immed	0 1 1 1	immed

The `BKPT` (Breakpoint) instruction causes a software breakpoint to occur. This breakpoint can be handled by an exception handler installed on the prefetch abort vector. In implementations which also include debug hardware, the hardware can optionally override this behavior and handle the breakpoint itself. When this occurs, the prefetch abort vector is not entered.

Syntax

```
BKPT  <immediate>
```

where:

`<immediate>`	Is a 16-bit immediate value, the top 12 bits of which are placed in bits[19:8] of the instruction, and the bottom 4 bits of which are placed in bits[3:0] of the instruction. This value is ignored by the ARM hardware, but can be used by a debugger to store additional information about the breakpoint.

Architecture version

Version 5 and above

Exceptions

Prefetch Abort

Operation

```
if (not overridden by debug hardware)
    R14_abt   = address of BKPT instruction + 4
    SPSR_abt  = CPSR
    CPSR[4:0] = 0b10111             /* Enter Abort mode */
    CPSR[5]   = 0                   /* Execute in ARM state */
    /* CPSR[6] is unchanged */
    CPSR[7]   = 1                   /* Disable normal interrupts */
    if high vectors configured then
        PC    = 0xFFFF000C
    else
        PC    = 0x0000000C
```

Usage

The exact usage of the `BKPT` instruction depends on the debug system being used. A debug system can use the `BKPT` instruction in two ways:

- Debug hardware, if present, does not override the normal behavior of the `BKPT` instruction, and so the prefetch abort vector is entered. If the system also allows real prefetch aborts to occur, the prefetch abort handler determines, in a system-dependent manner, whether the vector entry occurred as a result of a `BKPT` instruction or as a result of a real prefetch abort, and branches to debug code or prefetch abort code accordingly. Otherwise, the prefetch abort handler just branches straight to debug code.

  When used in this manner, the `BKPT` instruction must be avoided within abort handlers, as it corrupts R14_abt and SPSR_abt. For the same reason, it must also be avoided within FIQ handlers, since an FIQ interrupt can occur within an abort handler.

- Debug hardware does override the normal behavior of the `BKPT` instruction and handles the software breakpoint itself. When finished, it typically either resumes execution at the instruction following the `BKPT`, or replaces the `BKPT` in memory with another instruction and resumes execution at that instruction.

  When `BKPT` is used in this manner, R14_abt and SPSR_abt are not corrupted, and so the above restrictions about its use in abort and FIQ handlers do not apply.

Notes

Condition field The `BKPT` instruction must be unconditional. If bits[31:28] of the instruction encode a valid condition other than the `AL` (always) condition, the instruction is UNPREDICTABLE.

Hardware override Debug hardware in an implementation is specifically permitted to override the normal behavior of the `BKPT` instruction. Because of this, software must not use this instruction for purposes other than those documented by the debug system being used (if any). In particular, software cannot rely on the Prefetch Abort exception occurring, unless either there is guaranteed to be no debug hardware in the system or the debug system specifies that it will occur.

For more information, consult the documentation for the debug system being used.

4.1.8 BLX (1)

31	30	29	28	27	26	25	24	23	0
1	1	1	1	1	0	1	H	signed_immed_24	

This form of the `BLX` (Branch with Link and Exchange) instruction is used to call a Thumb subroutine from the ARM instruction set at an address specified in the instruction. This instruction is unconditional (always causing a change in program flow) and preserves the address of the instruction following the branch in the link register (R14). Execution of Thumb instructions begins at the target address.

Syntax

```
BLX  <target_addr>
```

where:

`<target_addr>`	Specifies the address of the Thumb instruction to branch to. The branch target address is calculated by: 1. Sign-extending the 24-bit signed (two's complement) immediate to 32 bits 2. Shifting the result left two bits 3. Setting bit[1] of the result of step 2 to the H bit 4. Adding the result of step 3 to the contents of the PC, which contains the address of the branch instruction plus 8. The instruction can therefore specify a branch of approximately ±32MB.

Architecture version

Version 5 and above

Exceptions

None

Operation

```
LR = address of the instruction after the BLX instruction
T Flag = 1
PC = PC + (SignExtend(signed_immed_24) << 2) + (H << 1)
```

Usage

To return from a Thumb subroutine called via `BLX` to the ARM caller, use the Thumb instruction:

```
    BX    R14
```

as described in *BX* on page A7-32, or use this instruction on subroutine entry:

```
    PUSH {<registers>,R14}
```

and this instruction to return:

```
    POP  {<registers>,PC}
```

To calculate the correct value of signed_immed_24, the assembler (or other toolkit component) needs to:

1. Form the base address for this branch instruction. This is the address of the instruction, plus 8. In other words, this base address is equal to the PC value used by the instruction.
2. Subtract the base address from the target address to form a byte offset. This offset is always even, because all ARM instructions are word-aligned and all Thumb instructions are halfword-aligned.
3. If the byte offset is outside the range −33554432 to +33554430, use an alternative code-generation strategy or produce an error as appropriate.
4. Otherwise, set the signed_immed_24 field of the instruction to bits{25:2] of the byte offset, and the H bit of the instruction to bit[1] of the byte offset.

Notes

Condition	Unlike most other ARM instructions, this instruction cannot be executed conditionally.
Bit[24]	This bit is used as bit[1] of the target address.

4.1.9 BLX (2)

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 1 0 0 1 0	SBO	SBO	SBO	0 0 1 1	Rm

This form of `BLX` is used to call an ARM or Thumb subroutine from the ARM instruction set, at an address specified in a register. The branch target address is the value of register Rm, with its bit[0] forced to zero. The instruction set to be used at the branch target is chosen by setting the CPSR T bit to bit[0] of Rm.

Register 14 is set to a return address. To return from the subroutine, use a `BX R14` instruction, or store R14 on the stack and re-load the stored value into the PC.

Syntax

```
BLX{<cond>}  <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rm>`	Is the register containing the address of the target instruction. Bit[0] of Rm is 0 to select a target ARM instruction, or 1 to select a target Thumb instruction. If R15 is specified for `<Rm>`, the results are UNPREDICTABLE.

Architecture version

Version 5 and above

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    LR = address of instruction after the BLX instruction
    T Flag = Rm[0]
    PC = Rm AND 0xFFFFFFFE
```

Notes

ARM/Thumb state transfers

If Rm[1:0] == 0b10, the result is UNPREDICTABLE, as branches to non word-aligned addresses are impossible in ARM state.

4.1.10 BX

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 1 0 0 1 0	SBO	SBO	SBO	0 0 0 1	Rm

The `BX` (Branch and Exchange) instruction branches to an address held in a register Rm, with an optional switch to Thumb execution. The branch target address is the value of register Rm, with its bit[0] forced to zero. The instruction set to be used at the branch target is chosen by setting the CPSR T bit to bit[0] of Rm.

Syntax

```
BX{<cond>}  <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rm>`	Holds the value of the branch target address. Bit[0] of Rm is 0 to select a target ARM instruction, or 1 to select a target Thumb instruction.

Architecture version

Version 5 and above, plus T variants of version 4

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    T Flag = Rm[0]
    PC = Rm AND 0xFFFFFFFE
```

Notes

ARM/Thumb state transfers

If Rm[1:0] == 0b10, the result is UNPREDICTABLE, as branches to non word-aligned addresses are impossible in ARM state.

Use of R15 Register 15 can be specified for `<Rm>`, but doing so is discouraged.

In a `BX R15` instruction, R15 is read as normal for ARM code, that is, it is the address of the `BX` instruction itself plus 8. The result is to branch to the second following word, executing in ARM state. This is precisely the same effect that would have been obtained if a `B` instruction with an offset field of 0 had been executed, or an `ADD PC,PC,#0` or `MOV PC,PC` instruction. In new code, use these instructions in preference to the more complex `BX PC` instruction.

4.1.11 CDP

31 28	27	26	25	24	23 20	19 16	15 12	11 8	7 5	4	3 0
cond	1	1	1	0	opcode_1	CRn	CRd	cp_num	opcode_2	0	CRm

The `CDP` (Coprocessor Data Processing) instruction tells the coprocessor whose number is cp_num to perform an operation that is independent of ARM registers and memory. If no coprocessors indicate that they can execute the instruction, an Undefined Instruction exception is generated.

Syntax

```
CDP{<cond>}  <coproc>, <opcode_1>, <CRd>, <CRn>, <CRm>, <opcode_2>
CDP2         <coproc>, <opcode_1>, <CRd>, <CRn>, <CRm>, <opcode_2>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.
`CDP2`	Causes the condition field of the instruction to be set to 0b1111. This provides additional opcode space for coprocessor designers. The resulting instructions can only be executed unconditionally.
`<coproc>`	Specifies the name of the coprocessor, and causes the corresponding coprocessor number to be placed in the cp_num field of the instruction. The standard generic coprocessor names are p0, p1, ..., p15.
`<opcode_1>`	Specifies (in a coprocessor-specific manner) which coprocessor operation is to be performed.
`<CRd>`	Specifies the destination coprocessor register for the instruction.
`<CRn>`	Specifies the coprocessor register that contains the first operand for the instruction.
`<CRm>`	Specifies the coprocessor register that contains the second operand for the instruction.
`<opcode_2>`	Specifies (in a coprocessor-specific manner) which coprocessor operation is to be performed.

Architecture version

`CDP` is in Version 2 and above.

`CDP2` is in Version 5 and above.

Exceptions

Undefined Instruction

Operation

```
if ConditionPassed(cond) then
    Coprocessor[cp_num]-dependent operation
```

Usage

`CDP` is used to initiate coprocessor instructions that do not operate on values in ARM registers or in main memory. An example is a floating-point multiply instruction for a floating-point coprocessor.

Notes

Coprocessor fields Only instruction bits[31:24], bits[11:8], and bit[4] are architecturally defined. The remaining fields are recommendations, for compatibility with ARM Development Systems.

Unimplemented coprocessor instructions

Hardware coprocessor support is optional, regardless of the architecture version. An implementation can choose to implement a subset of the coprocessor instructions, or no coprocessor instructions at all. Any coprocessor instructions that are not implemented instead cause an undefined instruction trap.

4.1.12 CLZ

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	1	0	1	1	0	SBO	Rd	SBO	0	0	0	1	Rm

The `CLZ` (Count Leading Zeros) instruction returns the number of binary zero bits before the first binary one bit in a register value. The source register is scanned from the most significant bit (bit[31]) towards the least significant bit (bit[0]). The result value is 32 if no bits are set in the source register, and zero if bit[31] is set.

This instruction does not update the condition code flags.

Syntax

```
CLZ{<cond>}  <Rd>, <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register for the operation. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.
`<Rm>`	Specifies the source register for this operation. If R15 is specified for `<Rm>`, the result is UNPREDICTABLE.

Architecture version

Version 5 and above

Exceptions

None

Operation

```
if Rm == 0
    Rd = 32
else
    Rd = 31 - (bit position of most significant '1' in Rm)
```

Usage

To normalize the value of register `Rm`, use `CLZ` followed by a left shift of `Rm` by the resulting `Rd` value. This shifts `Rm` so that its most significant 1 bit is in bit[31]. Using `MOVS` rather than `MOV` sets the Z flag in the special case that `Rm` is zero and so does not have a most significant 1 bit:

```
    CLZ   Rd, Rm
    MOVS  Rm, Rm, LSL Rd
```

4.1.13 CMN

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	1 0 1 1	1	Rn	SBZ	shifter_operand

The `CMN` (Compare Negative) instruction compares a register value with the negative of another arithmetic value. The condition flags are updated, based on the result of adding the second arithmetic value to the register value, so that subsequent instructions can be conditionally executed.

Syntax

```
CMN{<cond>}  <Rn>, <shifter_operand>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rn>` Specifies the register that contains the first operand for the operation.

`<shifter_operand>`

Specifies the second operand for the operation. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `CMN`. Instead, see *Extending the instruction set* on page A3-27 to determine which instruction it is.

Architecture version

All

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    alu_out = Rn + shifter_operand
    N Flag = alu_out[31]
    Z Flag = if alu_out == 0 then 1 else 0
    C Flag = CarryFrom(Rn + shifter_operand)
    V Flag = OverflowFrom(Rn + shifter_operand)
```

Usage

CMN performs a comparison by adding the value of <shifter_operand> to the value of register <Rn>, and updates the condition code flags (based on the result). This is almost equivalent to subtracting the negative of the second operand from the first operand, and setting the flags on the result.

The difference is that the flag values generated can differ when the second operand is 0 or 0x80000000. For example, this instruction always leaves the C flag = 1:

```
CMP Rn, #0
```

while this instruction always leaves the C flag = 0:

```
CMN Rn, #0
```

4.1.14 CMP

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	1 0 1 0	1	Rn	SBZ	shifter_operand

The `CMP` (Compare) instruction compares a register value with another arithmetic value. The condition flags are updated, based on the result of subtracting the second arithmetic value from the register value, so that subsequent instructions can be conditionally executed.

Syntax

```
CMP{<cond>}  <Rn>, <shifter_operand>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rn>`	Specifies the register that contains the first operand for the operation.
`<shifter_operand>`	Specifies the second operand for the operation. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction. If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `CMP`. Instead, see *Extending the instruction set* on page A3-27 to determine which instruction it is.

Architecture version

All

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    alu_out = Rn - shifter_operand
    N Flag = alu_out[31]
    Z Flag = if alu_out == 0 then 1 else 0
    C Flag = NOT BorrowFrom(Rn - shifter_operand)
    V Flag = OverflowFrom(Rn - shifter_operand)
```

4.1.15 EOR

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	0 0 0 1	S	Rn	Rd	shifter_operand

The `EOR` (Exclusive OR) instruction performs a bitwise Exclusive-OR of the value of register `<Rn>` with the value of `<shifter_operand>`, and stores the result in the destination register `<Rd>`. The condition code flags are optionally updated, based on the result.

Syntax

```
EOR{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`S` Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the N and Z flags are set according to the result of the operation, and the C flag is set to the carry output bit generated by the shifter (see *Addressing Mode 1 - Data-processing operands* on page A5-2). The V flag and the rest of the CPSR are unaffected.
- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>` Specifies the destination register of the instruction.

`<Rn>` Specifies the register that contains the first operand for the operation.

`<shifter_operand>`

Specifies the second operand for the operation. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `EOR`. Instead, see *Extending the instruction set* on page A3-27 to determine which instruction it is.

Architecture version

All

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Rd = Rn EOR shifter_operand
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

Usage

`EOR` can be used to invert selected bits in a register. For each bit, `EOR` with 1 inverts that bit, and `EOR` with 0 leaves it unchanged.

4.1.16 LDC

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7 0
cond	1	1	0	P	U	N	W	1	Rn	CRd	cp_num	8_bit_word_offset

The `LDC` (Load Coprocessor) instruction loads memory data from the sequence of consecutive memory addresses calculated by `<addressing_mode>` to the coprocessor whose number is cp_num. If no coprocessors indicate that they can execute the instruction, an Undefined Instruction exception is generated.

Syntax

```
LDC{<cond>}{L}   <coproc>, <CRd>, <addressing_mode>
LDC2{L}          <coproc>, <CRd>, <addressing_mode>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.
`LDC2`	Causes the condition field of the instruction to be set to 0b1111. This provides additional opcode space for coprocessor designers. The resulting instructions can only be executed unconditionally.
`L`	Sets the N bit (bit[22]) in the instruction to 1 and specifies a long load (for example, double-precision instead of single-precision data transfer). If L is omitted, the N bit is 0 and the instruction specifies a short load.
`<coproc>`	Specifies the name of the coprocessor, and causes the corresponding coprocessor number to be placed in the cp_num field of the instruction. The standard generic coprocessor names are p0, p1, ..., p15.
`<CRd>`	Specifies the coprocessor destination register of the instruction.

`<addressing_mode>`

Is described in *Addressing Mode 5 - Load and Store Coprocessor* on page A5-56. It determines the P, U, Rn, W and 8_bit_word_offset bits of the instruction.

The syntax of all forms of `<addressing_mode>` includes a *base register* `<Rn>`. Some forms also specify that the instruction modifies the base register value (this is known as *base register writeback*).

Architecture version

`LDC` is in Version 2 and above.

`LDC2` is in Version 5 and above.

Exceptions

Undefined Instruction, Data Abort

Operation

```
if ConditionPassed(cond) then
    address = start_address
    load Memory[address,4] for Coprocessor[cp_num]
    while (NotFinished(Coprocessor[cp_num]))
        address = address + 4
        load Memory[address,4] for Coprocessor[cp_num]
    assert address == end_address
```

Usage

`LDC` is useful for loading coprocessor data from memory.

Notes

Coprocessor fields Only instruction bits[31:23], bits[21:16], and bits[11:0] are ARM architecture-defined. The remaining fields (bit[22] and bits[15:12]) are recommendations, for compatibility with ARM Development Systems.

In the case of the Unindexed addressing mode (P==0, U==1, W==0), instruction bits[7:0] are also not defined by the ARM architecture, and can be used to specify additional coprocessor options.

Data abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-17.

Non word-aligned addresses

Load coprocessor register instructions ignore the least significant two bits of `address`.

Alignment If an implementation includes a System Control coprocessor (see Chapter B2 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

Unimplemented coprocessor instructions

Hardware coprocessor support is optional, regardless of the architecture version. An implementation can choose to implement a subset of the coprocessor instructions, or no coprocessor instructions at all. Any coprocessor instructions that are not implemented instead cause an undefined instruction trap.

4.1.17 LDM (1)

31 28	27 26 25	24	23	22	21	20	19 16	15 0
cond	1 0 0	P	U	0	W	1	Rn	register_list

This form of the `LDM` (Load Multiple) instruction is useful for block loads, stack operations and procedure exit sequences. It loads a non-empty subset, or possibly all, of the general-purpose registers from sequential memory locations.

The general-purpose registers loaded can include the PC. If they do, the word loaded for the PC is treated as an address and a branch occurs to that address. In ARM architecture version 5 and above, bit[0] of the loaded value determines whether execution continues after this branch in ARM state or in Thumb state, as though a `BX (loaded_value)` instruction had been executed. In earlier versions of the architecture, bits[1:0] of the loaded value are ignored and execution continues in ARM state, as though the instruction `MOV PC,(loaded_value)` had been executed.

Syntax

```
LDM{<cond>}<addressing_mode>  <Rn>{!}, <registers>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`<addressing_mode>`
Is described in *Addressing Mode 4 - Load and Store Multiple* on page A5-48. It determines the P, U, and W bits of the instruction.

`<Rn>` Specifies the base register used by `<addressing_mode>`. Using R15 as the base register `<Rn>` gives an UNPREDICTABLE result.

`!` Sets the W bit, causing the instruction to write a modified value back to its base register Rn as specified in *Addressing Mode 4 - Load and Store Multiple* on page A5-48. If `!` is omitted, the W bit is 0 and the instruction does not change its base register in this way. (However, if the base register is included in `<registers>`, it changes when a value is loaded into it.)

`<registers>`
Is a list of registers, separated by commas and surrounded by `{` and `}`. It specifies the set of registers to be loaded by the `LDM` instruction.

The registers are loaded in sequence, the lowest-numbered register from the lowest memory address (`start_address`), through to the highest-numbered register from the highest memory address (`end_address`). If the PC is specified in the register list (opcode bit[15] is set), the instruction causes a branch to the address (data) loaded into the PC.

For each of i=0 to 15, bit[i] in the register_list field of the instruction is 1 if Ri is in the list and 0 otherwise. If bits[15:0] are all zero, the result is UNPREDICTABLE.

Architecture version

All

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    address = start_address

    for i = 0 to 14
        if register_list[i] == 1 then
            Ri = Memory[address,4]
            address = address + 4

    if register_list[15] == 1 then
        value = Memory[address,4]
        if (architecture version 5 or above) then
            pc = value AND 0xFFFFFFFE
            T Bit = value[0]
        else
            pc = value AND 0xFFFFFFFC
        address = address + 4

    assert end_address = address - 4
```

Notes

Operand restrictions

If the base register `<Rn>` is specified in `<registers>`, and base register writeback is specified, the final value of `<Rn>` is UNPREDICTABLE.

Data abort For details of the effects of the instruction if a data abort occurs, see *Effects of data-aborted instructions* on page A2-17.

Non word-aligned addresses

Load Multiple instructions ignore the least significant two bits of `address` (the words are not rotated as for Load Word).

Alignment If an implementation includes a System Control coprocessor (see Chapter B2 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

ARM/Thumb state transfers (ARM architecture version 5 and above)

If bits[1:0] of a value loaded for R15 are 0b10, the result is UNPREDICTABLE, as branches to non word-aligned addresses are impossible in ARM state.

Time order The time order of the accesses to individual words of memory generated by this instruction is only defined in some circumstances. See *Data accesses to memory-mapped I/O* on page A2-32 for details.

4.1.18 LDM (2)

31 28	27 26 25	24	23	22	21	20	19 16	15	14 0
cond	1 0 0	P	U	1	0	1	Rn	0	register_list

This form of `LDM` loads User mode registers when the processor is in a privileged mode (useful when performing process swaps, and in instruction emulators). The instruction loads a non-empty subset of the User mode general-purpose registers from sequential memory locations.

Syntax

```
LDM{<cond>}<addressing_mode>  <Rn>, <registers_without_pc>^
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`<addressing_mode>`

Is described in *Addressing Mode 4 - Load and Store Multiple* on page A5-48. It determines the P and U bits of the instruction. Only the forms of this addressing mode with W == 0 are available for this form of the `LDM` instruction.

`<Rn>` Specifies the base register used by `<addressing_mode>`. Using R15 as `<Rn>` gives an UNPREDICTABLE result.

`<registers_without_pc>`

Is a list of registers, separated by commas and surrounded by `{` and `}`. This list must not include the PC, and specifies the set of registers to be loaded by the `LDM` instruction.

The registers are loaded in sequence, the lowest-numbered register from the lowest memory address (`start_address`), through to the highest-numbered register from the highest memory address (`end_address`).

For each of i=0 to 15, bit[i] in the register_list field of the instruction is 1 if Ri is in the list and 0 otherwise. If bits[15:0] are all zero, the result is UNPREDICTABLE.

`^` For an `LDM` instruction that does not load the PC, this indicates that User mode registers are to be loaded.

Architecture version

All

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    address = start_address
    for i = 0 to 14
        if register_list[i] == 1
            Ri_usr = Memory[address,4]
            address = address + 4
    assert end_address == address - 4
```

Notes

Banked registers This form of `LDM` must not be followed by an instruction which accesses banked registers (a following `NOP` is a good way to ensure this).

Writeback Setting bit[21] (the W bit) has UNPREDICTABLE results.

User and System mode

This form of `LDM` is UNPREDICTABLE in User mode or System mode.

Base register mode The base register is read from the current processor mode registers, not the User mode registers.

Data abort For details of the effects of the instruction if a data abort occurs, see *Effects of data-aborted instructions* on page A2-17.

Non word-aligned addresses

`LDM` instructions ignore the least significant two bits of `address` (words are not rotated as for Load Word).

Alignment If an implementation includes a System Control coprocessor (see Chapter B2 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

Time order The time order of the accesses to individual words of memory generated by this instruction is only defined in some circumstances. See *Data accesses to memory-mapped I/O* on page A2-32 for details.

4.1.19 LDM (3)

31 28	27 26 25	24	23	22	21	20	19 16	15	14 0
cond	1 0 0	P	U	1	W	1	Rn	1	register_list

This form of is useful for returning from an exception. It loads a subset (or possibly all) of the general-purpose registers and the PC from sequential memory locations. Also, the SPSR of the current mode is copied to the CPSR.

The value loaded for the PC is treated as an address and a branch occurs to that address. In ARM architecture version 5 and above, and in T variants of version 4, the value copied from the SPSR T bit to the CPSR T bit determines whether execution continues after the branch in ARM state or in Thumb state. In earlier architecture versions, it continues after the branch in ARM state (the only possibility in those architecture versions).

Syntax

```
LDM{<cond>}<addressing_mode>  <Rn>{!}, <registers_and_pc>^
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.
`<addressing_mode>`	Is described in *Addressing Mode 4 - Load and Store Multiple* on page A5-48. It determines the P, U, and W bits of the instruction.
`<Rn>`	Specifies the base register used by `<addressing_mode>`. Using R15 as `<Rn>` gives an UNPREDICTABLE result.
!	Sets the W bit, and the instruction writes a modified value back to its base register Rn (see *Addressing Mode 4 - Load and Store Multiple* on page A5-48). If ! is omitted, the W bit is 0 and the instruction does not change its base register in this way. (However, if the base register is included in `<registers>`, it changes when a value is loaded into it.)
`<registers_and_pc>`	Is a list of registers, separated by commas and surrounded by `{` and `}`. This list must include the PC, and specifies the set of registers to be loaded by the `LDM` instruction. The registers are loaded in sequence, the lowest-numbered register from the lowest memory address (`start_address`), through to the highest-numbered register from the highest memory address (`end_address`). For each of i=0 to 15, bit[i] in the register_list field of the instruction is 1 if Ri is in the list and 0 otherwise.
^	For an `LDM` instruction that loads the PC, this indicates that the SPSR of the current mode is copied to the CPSR.

Architecture version

All

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    address = start_address

    for i = 0 to 14
        if register_list[i] == 1 then
            Ri = Memory[address,4]
            address = address + 4

    CPSR = SPSR

    value = Memory[address,4]
    if (architecture version 4T, 5 or above) and (T Bit == 1) then
        pc = value AND 0xFFFFFFFE
    else
        pc = value AND 0xFFFFFFFC
    address = address + 4

    assert end_address = address - 4
```

Notes

User and System mode

This instruction is UNPREDICTABLE in User or System mode.

Operand restrictions

If the base register `<Rn>` is specified in `<registers_and_pc>`, and base register writeback is specified, the final value of `<Rn>` is UNPREDICTABLE.

Data abort For details of the effects of the instruction if a data abort occurs, see *Effects of data-aborted instructions* on page A2-17.

Non word-aligned addresses

Load Multiple instructions ignore the least significant two bits of `address` (the words are not rotated as for Load Word).

Alignment If an implementation includes a System Control coprocessor (see Chapter B2 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

ARM/Thumb state transfers (ARM architecture versions 4T, 5 and above)

If the SPSR T bit is 0 and bit[1] of the value loaded into the PC is 1, the results are UNPREDICTABLE because it is not possible to branch to an ARM instruction at a non word-aligned address. Note that no special precautions against this are needed on normal exception returns, because exception entries always either set the T bit of the SPSR to 1 or bit[1] of the return link value in R14 to 0.

Time order The time order of the accesses to individual words of memory generated by this instruction is not defined. See *Data accesses to memory-mapped I/O* on page A2-32 for details.

4.1.20 LDR

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	P	U	0	W	1	Rn	Rd	addr_mode

The `LDR` (Load Register) instruction loads a word from the memory address calculated by `<addressing_mode>` and writes it to register `<Rd>`. If the address is not word-aligned, the loaded value is rotated right by 8 times the value of bits[1:0] of the address. For a little-endian memory system, this rotation causes the addressed byte to occupy the least significant byte of the register. For a big-endian memory system, it causes the addressed byte to occupy bits[31:24] or bits[15:8] of the register, depending on whether bit[0] of the address is 0 or 1 respectively.

If the PC is specified as register `<Rd>`, the instruction loads a data word which it treats as an address, then branches to that address. In ARM architecture version 5 and above, bit[0] of the loaded value determines whether execution continues after this branch in ARM state or in Thumb state, as though a `BX (loaded_value)` instruction had been executed. In earlier versions of the architecture, bits[1:0] of the loaded value are ignored and execution continues in ARM state, as though a `MOV PC, (loaded_value)` instruction had been executed.

Syntax

```
LDR{<cond>}  <Rd>, <addressing_mode>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rd>` Specifies the destination register for the loaded value.

`<addressing_mode>`

Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-18. It determines the I, P, U, W, Rn and addr_mode bits of the instruction.

The syntax of all forms of `<addressing_mode>` includes a *base register* `<Rn>`. Some forms also specify that the instruction modifies the base register value (this is known as *base register writeback*).

Architecture version

All

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    if address[1:0] == 0b00 then
        value = Memory[address,4]
    else if address[1:0] == 0b01 then
        value = Memory[address,4] Rotate_Right 8
    else if address[1:0] == 0b10 then
        value = Memory[address,4] Rotate_Right 16
    else /* address[1:0] == 0b11 */
        value = Memory[address,4] Rotate_Right 24

    if (Rd is R15) then
        if (architecture version 5 or above) then
            PC = value AND 0xFFFFFFFE
            T Bit = value[0]
        else
            PC = value AND 0xFFFFFFFC
    else
        Rd = value
```

Usage

Using the PC as the base register allows PC-relative addressing, which facilitates position-independent code. Combined with a suitable addressing mode, `LDR` allows 32-bit memory data to be loaded into a general-purpose register where its value can be manipulated. If the destination register is the PC, this instruction loads a 32-bit address from memory and branches to that address.

To synthesize a Branch with Link, precede the `LDR` instruction with `MOV LR, PC`.

Notes

Data abort For details of the effects of the instruction if a data abort occurs, see *Effects of data-aborted instructions* on page A2-17.

Operand restrictions

If `<addressing_mode>` specifies base register writeback, and the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.

Alignment If an implementation includes a System Control coprocessor (see Chapter B2 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

Use of R15 If register 15 is specified for `<Rd>`, address[1:0] must be 0b00. If not, the result is UNPREDICTABLE.

ARM/Thumb state transfers (ARM architecture version 5 and above)

If bits[1:0] of the loaded value are 0b10, the result is UNPREDICTABLE, as branches to non word-aligned addresses are not possible in ARM state.

4.1.21 LDRB

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	P	U	1	W	1	Rn	Rd	addr_mode

The `LDRB` (Load Register Byte) instruction loads a byte from the memory address calculated by `<addressing_mode>`, zero-extends the byte to a 32-bit word, and writes the word to register `<Rd>`.

Syntax

```
LDR{<cond>}B  <Rd>, <addressing_mode>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rd>` Specifies the destination register for the loaded value. If register 15 is specified for `<Rd>`, the result is UNPREDICTABLE.

`<addressing_mode>`

Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-18. It determines the I, P, U, W, Rn and addr_mode bits of the instruction.

The syntax of all forms of `<addressing_mode>` includes a *base register* `<Rn>`. Some forms also specify that the instruction modifies the base register value (this is known as *base register writeback*).

Architecture version

All

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    Rd = Memory[address,1]
```

Usage

Combined with a suitable addressing mode, `LDRB` allows 8-bit memory data to be loaded into a general-purpose register where it can be manipulated.

Using the PC as the base register allows PC-relative addressing, to facilitate position-independent code.

Notes

Operand restrictions

If `<addressing_mode>` specifies base register writeback, and the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.

Data abort For details of the effects of the instruction if a data abort occurs, see *Effects of data-aborted instructions* on page A2-17.

4.1.22 LDRBT

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	0	U	1	1	1	Rn	Rd	addr_mode

The `LDRBT` (Load Register Byte with Translation) instruction loads a byte from the memory address calculated by `<post_indexed_addressing_mode>`, zero-extends the byte to a 32-bit word, and writes the word to register `<Rd>`.

If the instruction is executed when the processor is in a privileged mode, the memory system is signaled to treat the access as if the processor were in User mode.

Syntax

```
LDR{<cond>}BT  <Rd>, <post_indexed_addressing_mode>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rd>` Specifies the destination register for the loaded value. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.

`<post_indexed_addressing_mode>`

Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-18. It determines the I, U, Rn and addr_mode bits of the instruction. Only post-indexed forms of Addressing Mode 2 are available for this instruction. These forms have P == 0 and W == 0, where P and W are bit[24] and bit[21] respectively. This instruction uses P == 0 and W == 1 instead, but the addressing mode is the same in all other respects.

The syntax of all forms of `<post_indexed_addressing_mode>` includes a *base register* `<Rn>`. All forms also specify that the instruction modifies the base register value (this is known as *base register writeback*).

Architecture version

All

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    Rd = Memory[address,1]
```

Usage

`LDRBT` can be used by a (privileged) exception handler that is emulating a memory access instruction that would normally execute in User mode. The access is restricted as if it had User mode privilege.

Notes

User mode If this instruction is executed in User mode, an ordinary User mode access is performed.

Operand restrictions

If the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.

Data abort For details of the effects of the instruction if a data abort occurs, see *Effects of data-aborted instructions* on page A2-17.

4.1.23 LDRH

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	P	U	I	W	1	Rn	Rd	addr_mode	1	0	1	1	addr_mode

The `LDRH` (Load Register Halfword) instruction loads a halfword from the memory address calculated by `<addressing_mode>`, zero-extends the halfword to a 32-bit word, and writes the word to register `<Rd>`. If the address is not halfword-aligned, the result is UNPREDICTABLE.

Syntax

```
LDR{<cond>}H  <Rd>, <addressing_mode>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rd>` Specifies the destination register for the loaded value. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.

`<addressing_mode>`

Is described in *Addressing Mode 3 - Miscellaneous Loads and Stores* on page A5-34. It determines the P, U, I, W, Rn and addr_mode bits of the instruction.

The syntax of all forms of `<addressing_mode>` includes a *base register* `<Rn>`. Some forms also specify that the instruction modifies the base register value (this is known as *base register writeback*).

Architecture version

Version 4 and above

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    if address[0] == 0
        data = Memory[address,2]
    else /* address[0] == 1 */
        data = UNPREDICTABLE
    Rd = data
```

Usage

Used with a suitable addressing mode, `LDRH` allows 16-bit memory data to be loaded into a general-purpose register where its value can be manipulated.

Using the PC as the base register allows PC-relative addressing to facilitate position-independent code.

Notes

Operand restrictions

If `<addressing_mode>` specifies base register writeback, and the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.

Data abort For details of the effects of the instruction if a data abort occurs, see *Effects of data-aborted instructions* on page A2-17.

Non halfword-aligned addresses

If the load address is not halfword-aligned, the loaded value is UNPREDICTABLE.

Alignment If an implementation includes a System Control coprocessor (see Chapter B2 *The System Control Coprocessor*), and alignment checking is enabled, an address with bit[0] != 0 causes an alignment exception.

4.1.24 LDRSB

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	P	U	I	W	1	Rn	Rd	addr_mode	1	1	0	1	addr_mode

The `LDRSB` (Load Register Signed Byte) instruction loads a byte from the memory address calculated by `<addressing_mode>`, sign-extends the byte to a 32-bit word, and writes the word to register `<Rd>`.

Syntax

```
LDR{<cond>}SB  <Rd>, <addressing_mode>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rd>` Specifies the destination register for the loaded value. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.

`<addressing_mode>`

Is described in *Addressing Mode 3 - Miscellaneous Loads and Stores* on page A5-34. It determines the P, U, I, W, Rn and addr_mode bits of the instruction.

The syntax of all forms of `<addressing_mode>` includes a *base register* `<Rn>`. Some forms also specify that the instruction modifies the base register value (this is known as *base register writeback*).

Architecture version

Version 4 and above

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    data = Memory[address,1]
    Rd = SignExtend(data)
```

Usage

Used with a suitable addressing mode, `LDRSB` allows 8-bit signed memory data to be loaded into a general-purpose register where it can be manipulated. Using the PC as the base register allows PC-relative addressing, which facilitates position-independent code.

Notes

Operand restrictions

If `<addressing_mode>` specifies base register writeback, and the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.

Data abort For details of the effects of the instruction if a data abort occurs, see *Effects of data-aborted instructions* on page A2-17.

4.1.25 LDRSH

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	P	U	I	W	1	Rn	Rd	addr_mode	1	1	1	1	addr_mode

The `LDRSH` (Load Register Signed Halfword) instruction loads a halfword from the memory address calculated by `<addressing_mode>`, sign-extends the halfword to a 32-bit word, and writes the word to register `<Rd>`. If the address is not halfword-aligned, the result is UNPREDICTABLE.

Syntax

```
LDR{<cond>}SH  <Rd>, <addressing_mode>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rd>` Specifies the destination register for the loaded value. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.

`<addressing_mode>`

Is described in *Addressing Mode 3 - Miscellaneous Loads and Stores* on page A5-34. It determines the P, U, I, W, Rn and addr_mode bits of the instruction.

The syntax of all forms of `<addressing_mode>` includes a *base register* `<Rn>`. Some forms also specify that the instruction modifies the base register value (this is known as *base register writeback*).

Architecture version

Version 4 and above

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    if address[0] == 0
        data = Memory[address,2]
    else /* address[0] == 1 */
        data = UNPREDICTABLE
    Rd = SignExtend(data)
```

Usage

Used with a suitable addressing mode, `LDRSH` allows 16-bit signed memory data to be loaded into a general-purpose register where its value can be manipulated.

Using the PC as the base register allows PC-relative addressing, which facilitates position-independent code.

Notes

Operand restrictions

If `<addressing_mode>` specifies base register writeback, and the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.

Data abort For details of the effects of the instruction if a data abort occurs, see *Effects of data-aborted instructions* on page A2-17.

Non halfword-aligned addresses

If the load address is not halfword-aligned, the loaded value is UNPREDICTABLE.

Alignment If an implementation includes a System Control coprocessor (see Chapter B2 *The System Control Coprocessor*), and alignment checking is enabled, an address with bit[0] != 0 causes an alignment exception.

4.1.26 LDRT

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	0	U	0	1	1	Rn	Rd	addr_mode

The `LDRT` (Load Register with Translation) instruction loads a word from the memory address calculated by `<addressing_mode>` and writes it to register `<Rd>`. If the address is not word-aligned, the loaded data is rotated as for the `LDR` instruction (see *LDR* on page A4-37).

If the instruction is executed when the processor is in a privileged mode, the memory system is signaled to treat the access as if the processor were in User mode.

Syntax

```
LDR{<cond>}T  <Rd>, <post_indexed_addressing_mode>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rd>` Specifies the destination register for the loaded value. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.

`<post_indexed_addressing_mode>`

Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-18. It determines the I, U, Rn and addr_mode bits of the instruction. Only post-indexed forms of Addressing Mode 2 are available for this instruction. These forms have P == 0 and W == 0, where P and W are bit[24] and bit[21] respectively. This instruction uses P == 0 and W == 1 instead, but the addressing mode is the same in all other respects.

The syntax of all forms of `<post_indexed_addressing_mode>` includes a *base register* `<Rn>`. All forms also specify that the instruction modifies the base register value (this is known as *base register writeback*).

Architecture version

All

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    if address[1:0] == 0b00
        Rd = Memory[address,4]
    else if address[1:0] == 0b01
        Rd = Memory[address,4] Rotate_Right 8
    else if address[1:0] == 0b10
        Rd = Memory[address,4] Rotate_Right 16
    else /* address[1:0] == 0b11 */
        Rd = Memory[address,4] Rotate_Right 24
```

Usage

`LDRT` can be used by a (privileged) exception handler that is emulating a memory access instruction that would normally execute in User mode. The access is restricted as if it had User mode privilege.

Notes

User mode If this instruction is executed in User mode, an ordinary User mode access is performed.

Operand restrictions

If the same register is specified for `<Rd>` and `<Rn>` the results are UNPREDICTABLE.

Data abort For details of the effects of the instruction if a data abort occurs, see *Effects of data-aborted instructions* on page A2-17.

Alignment If an implementation includes a System Control coprocessor (See Chapter B2 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

4.1.27 MCR

31 28	27 26 25 24	23 21	20	19 16	15 12	11 8	7 5	4	3 0
cond	1 1 1 0	opcode_1	0	CRn	Rd	cp_num	opcode_2	1	CRm

The `MCR` (Move to Coprocessor from ARM Register) instruction passes the value of register `<Rd>` to the coprocessor whose number is cp_num. If no coprocessors indicate that they can execute the instruction, an Undefined Instruction exception is generated.

Syntax

```
MCR{<cond>}  <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>{, <opcode_2>}
MCR2         <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>{, <opcode_2>}
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.
`MCR2`	Causes the condition field of the instruction to be set to 0b1111. This provides additional opcode space for coprocessor designers. The resulting instructions can only be executed unconditionally.
`<coproc>`	Specifies the name of the coprocessor, and causes the corresponding coprocessor number to be placed in the cp_num field of the instruction. The standard generic coprocessor names are p0, p1, ..., p15.
`<opcode_1>`	Is a coprocessor-specific opcode.
`<Rd>`	Is the ARM register whose value is transferred to the coprocessor. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.
`<CRn>`	Is the destination coprocessor register.
`<CRm>`	Is an additional destination or source coprocessor register.
`<opcode_2>`	Is a coprocessor-specific opcode. If it is omitted, `<opcode_2>` is assumed to be 0.

Architecture version

`MCR` is in version 2 and above.

`MCR2` is in version 5 and above.

Exceptions

Undefined Instruction

Operation

```
if ConditionPassed(cond) then
    send Rd value to Coprocessor[cp_num]
```

Usage

`MCR` is used to initiate coprocessor instructions that operate on values in ARM registers. An example is a fixed-point to floating-point conversion instruction for a floating-point coprocessor.

Notes

Coprocessor fields Only instruction bits[31:24], bit[20], bits[15:8], and bit[4] are defined by the ARM architecture. The remaining fields are recommendations, for compatibility with ARM Development Systems.

Unimplemented coprocessor instructions

Hardware coprocessor support is optional, regardless of the architecture version. An implementation can choose to implement a subset of the coprocessor instructions, or no coprocessor instructions at all. Any coprocessor instructions that are not implemented instead cause an undefined instruction trap.

4.1.28 MLA

31 28	27 26 25 24 23 22 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 0 0 0 1	S	Rd	Rn	Rs	1 0 0 1	Rm

The `MLA` (Multiply Accumulate) multiplies signed or unsigned operands to produce a 32-bit result, which is then added to a third operand, and written to the destination register. The condition code flags are optionally updated, based on the result.

Syntax

```
MLA{<cond>}{S}  <Rd>, <Rm>, <Rs>, <Rn>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.
`S`	Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR by setting the N and Z flags according to the result of the multiply-accumulate. If `S` is omitted, the S bit of the instruction is set to 0 and the entire CPSR is unaffected by the instruction.
`<Rd>`	Specifies the destination register of the instruction.
`<Rm>`	Holds the value to be multiplied with the value of `<Rs>`.
`<Rs>`	Holds the value to be multiplied with the value of `<Rm>`.
`<Rn>`	Contains the value that is added to the product of `<Rs>` and `<Rm>`.

Architecture version

Version 2 and above

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Rd = (Rm * Rs + Rn)[31:0]
    if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = unaffected    /* See "C flag" note */
        V Flag = unaffected
```

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, `<Rs>`, or `<Rn>` has UNPREDICTABLE results.
Operand restriction	Specifying the same register for `<Rd>` and `<Rm>` has UNPREDICTABLE results.
Early termination	If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rs>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.
Signed and unsigned	Because the `MLA` instruction produces only the lower 32 bits of the 64-bit product, `MLA` gives the same answer for multiplication of both signed and unsigned numbers.
C flag	The `MLAS` instruction is defined to leave the C flag unchanged in ARM architecture version 5 and above. In earlier versions of the architecture, the value of the C flag was UNPREDICTABLE after a `MLAS` instruction.

4.1.29 MOV

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	1	1	0	1	S	SBZ	Rd	shifter_operand

The `MOV` (Move) instruction moves the value of `<shifter_operand>` to the destination register `<Rd>`. The condition code flags are optionally updated, based on the result.

Syntax

```
MOV{<cond>}{S}  <Rd>, <shifter_operand>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`S` Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the N and Z flags are set according to the value moved (post-shift if a shift is specified), and the C flag is set to the carry output bit generated by the shifter (see *Addressing Mode 1 - Data-processing operands* on page A5-2). The V flag and the rest of the CPSR are unaffected.
- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>` Specifies the destination register of the instruction.

`<shifter_operand>`
Specifies the operand for the operation. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `MOV`. Instead, see *Extending the instruction set* on page A3-27 to determine which instruction it is.

Architecture version

All

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Rd = shifter_operand
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

Usage

`MOV` is used to:

- Move a value from one register to another.
- Put a constant value into a register.
- Perform a shift without any other arithmetic or logical operation. A left shift by *n* can be used to multiply by 2^n.
- When the PC is the destination of the instruction, a branch occurs. The instruction:

  ```
  MOV PC, LR
  ```

  can therefore be used to return from a subroutine (see instructions *B, BL* on page A4-10). In T variants of architecture 4 and in architecture 5 and above, the instruction `BX LR` must be used in place of `MOV PC, LR`, as the `BX` instruction automatically switches back to Thumb state if appropriate.
- When the PC is the destination of the instruction and the S bit is set, a branch occurs and the SPSR of the current mode is copied to the CPSR. This means that a `MOVS PC, LR` instruction can be used to return from some types of exception (see *Exceptions* on page A2-13).

4.1.30 MRC

31 28	27	26	25	24	23 21	20	19 16	15 12	11 8	7 5	4	3 0
cond	1	1	1	0	opcode_1	1	CRn	Rd	cp_num	opcode_2	1	CRm

The `MRC` (Move to ARM Register from Coprocessor) instruction causes the coprocessor whose number is cp_num to transfer a value to an ARM register or to the condition flags.

If no coprocessors indicate that they can execute the instruction an Undefined Instruction exception is generated.

Syntax

```
MRC{<cond>}  <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>{, <opcode_2>}
MRC2         <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>{, <opcode_2>}
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.
`MRC2`	Causes the condition field of the instruction to be set to 0b1111. This provides additional opcode space for coprocessor designers. The resulting instructions can only be executed unconditionally.
`<coproc>`	Specifies the name of the coprocessor, and causes the corresponding coprocessor number to be placed in the cp_num field of the instruction. The standard generic coprocessor names are p0, p1, ..., p15.
`<opcode_1>`	Is a coprocessor-specific opcode.
`<Rd>`	Specifies the destination ARM register for the instruction. If R15 is specified for `<Rd>`, the condition code flags are updated instead of a general-purpose register.
`<CRn>`	Specifies the coprocessor register that contains the first operand for the instruction.
`<CRm>`	Is an additional coprocessor source or destination register.
`<opcode_2>`	Is a coprocessor-specific opcode. If it is omitted, `<opcode_2>` is assumed to be 0.

Architecture version

`MRC` is in version 2 and above.

`MRC2` is in version 5 and above.

Exceptions

Undefined Instruction

Operation

```
if ConditionPassed(cond) then
    data = value from Coprocessor[cp_num]
    if Rd is R15 then
        N flag = data[31]
        Z flag = data[30]
        C flag = data[29]
        V flag = data[28]
    else /* Rd is not R15 */
        Rd = data
```

Usage

`MRC` has two uses:

1. If `<Rd>` specifies R15, the condition code flags bits are updated from the top four bits of the value from the coprocessor specified by `<coproc>` (to allow conditional branching on the status of a coprocessor) and the other 28 bits are ignored.

   An example of this use would be to transfer the result of a comparison performed by a floating-point coprocessor to the ARM's condition flags.

2. Otherwise the instruction writes into register `<Rd>` a value from the coprocessor specified by `<coproc>`.

   An example of this use is a floating-point to integer conversion instruction in a floating-point coprocessor.

Notes

Coprocessor fields Only instruction bits[31:24], bit[20], bits[15:8] and bit[4] are defined by the ARM architecture. The remaining fields are recommendations, for compatibility with ARM Development Systems.

Unimplemented coprocessor instructions

Hardware coprocessor support is optional, regardless of the architecture version. An implementation can choose to implement a subset of the coprocessor instructions, or no coprocessor instructions at all. Any coprocessor instructions that are not implemented instead cause an undefined instruction trap.

4.1.31 MRS

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	0	1	0	R	0	0	SBO	Rd	SBZ

The `MRS` (Move PSR to General-purpose Register) instruction moves the value of the CPSR or the SPSR of the current mode into a general-purpose register. In the general-purpose register, the value can be examined or manipulated with normal data-processing instructions.

Syntax

```
MRS{<cond>}  <Rd>, CPSR
MRS{<cond>}  <Rd>, SPSR
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register of the instruction. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.

Architecture version

Version 3 and above

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    if R == 1 then
        Rd = SPSR
    else
        Rd = CPSR
```

Usage

The `MRS` instruction is commonly used for three purposes:

- As part of a read/modify/write sequence for updating a PSR. For more details, see *MSR* on page A4-62.
- When an exception occurs and there is a possibility of a nested exception of the same type occurring, the SPSR of the exception mode is in danger of being corrupted. To deal with this, the SPSR value must be saved before the nested exception can occur, and later restored in preparation for the exception return. The saving is normally done by using an `MRS` instruction followed by a store instruction. Restoring the SPSR uses the reverse sequence of a load instruction followed by an `MSR` instruction. For an example of this usage, see *Interrupt prioritization* on page A9-15.
- In process swap code, the programmer's model state of the process being swapped out needs to be saved, including relevant PSR contents, and similar state of the process being swapped in needs to be restored. Again, this involves the use of `MRS`/store and load/`MSR` instruction sequences. For an example of this usage, see *Context switch* on page A9-16.

Notes

Opcode [11:0]	Execution of `MRS` instructions with opcode[11:0] != `0x000` is UNPREDICTABLE.
Opcode [19:16]	Execution of `MRS` instructions with opcode[19:16] != 0b1111 is UNPREDICTABLE.
User mode SPSR	Accessing the SPSR when in User mode or System mode is UNPREDICTABLE.

4.1.32 MSR

Immediate operand:

31 28	27 26 25 24 23	22	21 20	19 16	15 12	11 8	7 0
cond	0 0 1 1 0	R	1 0	field_mask	SBO	rotate_imm	8_bit_immediate

Register operand:

31 28	27 26 25 24 23	22	21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 1 0	R	1 0	field_mask	SBO	SBZ	0 0 0 0	Rm

The `MSR` (Move to Status Register from ARM Register) instruction transfers the value of a general-purpose register or immediate constant to the CPSR or the SPSR of the current mode.

Syntax

```
MSR{<cond>}  CPSR_<fields>, #<immediate>
MSR{<cond>}  CPSR_<fields>, <Rm>
MSR{<cond>}  SPSR_<fields>, #<immediate>
MSR{<cond>}  SPSR_<fields>, <Rm>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`<fields>` Is a sequence of one or more of the following:

`c`	sets the control field mask bit (bit 16)
`x`	sets the extension field mask bit (bit 17)
`s`	sets the status field mask bit (bit 18)
`f`	sets the flags field mask bit (bit 19).

`<immediate>` Is the immediate value to be transferred to the CPSR or SPSR. Allowed immediate values are 8-bit immediates (in the range `0x00` to `0xFF`) and values that can be obtained by rotating them right by an even amount in the range 0–30. These immediate values are the same as those allowed in the immediate form as shown in *Data-processing operands - Immediate* on page A5-6.

`<Rm>` Is the general-purpose register to be transferred to the CPSR or SPSR.

Architecture version

Version 3 and above

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    if opcode[25] == 1
        operand = 8_bit_immediate Rotate_Right (rotate_imm * 2)
    else /* opcode[25] == 0 */
        operand = Rm
    if R == 0 then
        if field_mask[0] == 1 and InAPrivilegedMode() then
            CPSR[7:0] = operand[7:0]
        if field_mask[1] == 1 and InAPrivilegedMode() then
            CPSR[15:8] = operand[15:8]
        if field_mask[2] == 1 and InAPrivilegedMode() then
            CPSR[23:16] = operand[23:16]
        if field_mask[3] == 1 then
            CPSR[31:24] = operand[31:24]
    else /* R == 1 */
        if field_mask[0] == 1 and CurrentModeHasSPSR() then
            SPSR[7:0] = operand[7:0]
        if field_mask[1] == 1 and CurrentModeHasSPSR() then
            SPSR[15:8] = operand[15:8]
        if field_mask[2] == 1 and CurrentModeHasSPSR() then
            SPSR[23:16] = operand[23:16]
        if field_mask[3] == 1 and CurrentModeHasSPSR() then
            SPSR[31:24] = operand[31:24]
```

Usage

This instruction is used to update the value of the condition code flags, interrupt enables, or the processor mode.

The value of a PSR should normally be updated by moving the PSR to a general-purpose register (using the `MRS` instruction), modifying the relevant bits of the general-purpose register, and restoring the updated general-purpose register value back into the PSR (using the `MSR` instruction). For example, a good way to switch the ARM to Supervisor mode from another privileged mode is:

```
    MRS   R0,CPSR                     ; Read CPSR
    BIC   R0,R0,#0x1F                 ; Modify by removing current mode
    ORR   R0,R0,#0x13                 ; and substituting Supervisor mode
    MSR   CPSR_c,R0                   ; Write the result back to CPSR
```

For maximum efficiency, `MSR` instructions should only write to those fields that they can potentially change. For example, the last instruction in the above code can only change the CPSR control field, as all bits in the other fields are unchanged since they were read from the CPSR by the first instruction. So it writes to CPSR_c, not CPSR_fsxc or some other combination of fields.

However, if the *only* reason that an `MSR` instruction cannot change a field is that no bits are currently allocated to the field, then the field must be written, to ensure future compatibility. For example, when the process swap code in *Context switch* on page A9-16 restores the new process's CPSR value, it writes to SPSR_fsxc (which is later copied to the CPSR by the `LDM` instruction which restarts the process). There are no bits allocated to the state or extension fields at present, so writing to SPSR_fc would work just as well in current architecture versions (5 and below). However, writing to SPSR_fsxc will continue to work correctly in future versions of the architecture that do have bits allocated in the state or extension fields, while writing to SPSR_fc will not.

Note

Due to deficiencies in the handling of the state and extension fields in versions 2.50 and below of the ARM Software Development Toolkit, only specify the flags and control fields when using these toolkit versions. In cases where the above guidelines mean that the state and extension fields should have been written for future compatibility, it is recommended that you keep a record of the need to change the set of fields specified when it becomes possible to do so.

The immediate form of this instruction can be used to set any of the fields of a PSR, but you must take care to adhere to the read-modify-write technique described above. The immediate form of the instruction is equivalent to reading the PSR concerned, replacing all the bits in the fields concerned by the corresponding bits of the immediate constant and writing the result back to the PSR. The immediate form must therefore only be used when the intention is to modify all the bits in the specified fields and, in particular, must not be used if the specified fields include any as-yet-unallocated bits. Failure to observe this rule might result in code which has unanticipated side-effects on future versions of the ARM architecture.

As an exception to the above rule, it is legitimate to use the immediate form of the instruction to modify the flags byte, despite the fact that bits[26:24] of the PSRs have no allocated function at present. For example, this instruction can be used to set all four flags (and clear the Q flag if the processor implements the Enhanced DSP extension):

```
    MSR     CPSR_f,#0xF0000000
```

Any functionality allocated to bits[26:24] in a future version of the ARM architecture will be designed so that such code does not have unexpected side effects.

Notes

The R bit Bit[22] of the instruction is 0 if the CPSR is to be written and 1 if the SPSR is to be written.

User mode CPSR

Any writes to CPSR[23:0] in User mode are ignored (so that User mode programs cannot change to a privileged mode).

User mode SPSR

Accessing the SPSR when in User mode is UNPREDICTABLE.

System mode SPSR

Accessing the SPSR when in System mode is UNPREDICTABLE.

Obsolete field specification

The `CPSR`, `CPSR_flg`, `CPSR_ctl`, `CPSR_all`, `SPSR`, `SPSR_flg`, `SPSR_ctl` and `SPSR_all` forms of PSR field specification have been superseded by the `csxf` format shown on page A4-62.

`CPSR`, `SPSR`, `CPSR_all` and `SPSR_all` produce a field mask of 0b1001.

`CPSR_flg` and `SPSR_flg` produce a field mask of 0b1000.

`CPSR_ctl` and `SPSR_ctl` produce a field mask of 0b0001.

The T bit The `MSR` instruction must not be used to alter the T bit in the CPSR. If such an attempt is made, the results are UNPREDICTABLE.

Addressing modes

The immediate and register forms are specified in precisely the same way as the immediate and unshifted register forms of Addressing Mode 1 (see *Addressing Mode 1 - Data-processing operands* on page A5-2). All other forms of Addressing Mode 1 yield UNPREDICTABLE results.

4.1.33 MUL

31 28	27 26 25 24 23 22 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 0 0 0 0	S	Rd	SBZ	Rs	1 0 0 1	Rm

The `MUL` (Multiply) instruction is used to multiply signed or unsigned variables to produce a 32-bit result. The condition code flags are optionally updated, based on the result.

Syntax

```
MUL{<cond>}{S}  <Rd>, <Rm>, <Rs>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.
`S`	Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR by setting the N and Z flags according to the result of the multiplication. If `S` is omitted, the S bit of the instruction is set to 0 and the entire CPSR is unaffected by the instruction.
`<Rd>`	Specifies the destination register for the instruction.
`<Rm>`	Specifies the register that contains the first value to be multiplied.
`<Rs>`	Holds the value to be multiplied with the value of `<Rm>`.

Architecture version

Version 2 and above

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Rd = (Rm * Rs)[31:0]
    if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = unaffected    /* See "C flag" note */
        V Flag = unaffected
```

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rs>` has UNPREDICTABLE results.
Operand restriction	Specifying the same register for `<Rd>` and `<Rm>` has UNPREDICTABLE results.
Early termination	If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rs>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.
Signed and unsigned	Because the `MUL` instruction produces only the lower 32 bits of the 64-bit product, `MUL` gives the same answer for multiplication of both signed and unsigned numbers.
C flag	The `MULS` instruction is defined to leave the C flag unchanged in ARM architecture version 5 and above. In earlier versions of the architecture, the value of the C flag was UNPREDICTABLE after a `MULS` instruction.

4.1.34 MVN

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	1	1	1	1	S	SBZ	Rd	shifter_operand

The `MVN` (Move Negative) instruction moves the logical one's complement of the value of `<shifter_operand>` to the destination register `<Rd>`. The condition code flags are optionally updated, based on the result.

Syntax

```
MVN{<cond>}{S}  <Rd>, <shifter_operand>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`S` Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the N and Z flags are set according to the result of the operation, and the C flag is set to the carry output bit generated by the shifter (see *Addressing Mode 1 - Data-processing operands* on page A5-2). The V flag and the rest of the CPSR are unaffected.
- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>` Specifies the destination register of the instruction.

`<shifter_operand>`

Specifies the operand for the operation. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `MVN`. Instead, see *Extending the instruction set* on page A3-27 to determine which instruction it is.

Architecture version

All

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Rd = NOT shifter_operand
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

Usage

`MVN` is used to:

- write a negative value into a register
- form a bit mask
- take the one's complement of a value.

4.1.35 ORR

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	1 1 0 0	S	Rn	Rd	shifter_operand

The `ORR` (Logical OR) instruction performs a bitwise (inclusive) OR of the value of register `<Rn>` with the value of `<shifter_operand>`, and stores the result in the destination register `<Rd>`. The condition code flags are optionally updated, based on the result.

Syntax

```
ORR{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`S` Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the N and Z flags are set according to the result of the operation, and the C flag is set to the carry output bit generated by the shifter (see *Addressing Mode 1 - Data-processing operands* on page A5-2). The V flag and the rest of the CPSR are unaffected.
- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>` Specifies the destination register of the instruction.

`<Rn>` Specifies the register that contains the first operand for the operation.

`<shifter_operand>`

Specifies the second operand for the operation. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `ORR`. Instead, see *Extending the instruction set* on page A3-27 to determine which instruction it is.

Architecture version

All

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Rd = Rn OR shifter_operand
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

Usage

`ORR` can be used to set selected bits in a register. For each bit, OR with 1 sets the bit, and OR with 0 leaves it unchanged.

4.1.36 RSB

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	0 0 1 1	S	Rn	Rd	shifter_operand

The `RSB` (Reverse Subtract) instruction subtracts the value of register `<Rn>` from the value of `<shifter_operand>`, and stores the result in the destination register `<Rd>`. The condition code flags are optionally updated, based on the result.

Syntax

```
RSB{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`S` Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the `N` and `Z` flags are set according to the result of the subtraction, and the `C` and `V` flags are set according to whether the subtraction generated a borrow (unsigned underflow) and a signed overflow, respectively. The rest of the CPSR is unchanged.
- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>` Specifies the destination register of the instruction.

`<Rn>` Specifies the register that contains the second operand for the subtraction.

`<shifter_operand>`

Specifies the first operand for the subtraction. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `RSB`. Instead, see *Extending the instruction set* on page A3-27 to determine which instruction it is.

Architecture version

All

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Rd = shifter_operand - Rn
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = NOT BorrowFrom(shifter_operand - Rn)
        V Flag = OverflowFrom(shifter_operand - Rn)
```

Usage

The following instruction stores the negation (two's complement) of `Rx` in `Rd`:

```
    RSB Rd, Rx, #0
```

Constant multiplication (of `Rx`) by 2^n–1 (into `Rd`) can be performed with:

```
    RSB Rd, Rx, Rx, LSL #n
```

Notes

C flag If `S` is specified, the C flag is set to:

`1`	if no borrow occurs
`0`	if a borrow does occur.

In other words, the C flag is used as a NOT(borrow) flag. This inversion of the borrow condition is usually compensated for by subsequent instructions. For example:

- The `SBC` and `RSC` instructions use the C flag as a NOT(borrow) operand, performing a normal subtraction if C == 1 and subtracting one more than usual if C == 0.
- The HS (unsigned higher or same) and LO (unsigned lower) conditions are equivalent to CS (carry set) and CC (carry clear) respectively.

4.1.37 RSC

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	0	1	1	1	S	Rn	Rd	shifter_operand

The `RSC` (Reverse Subtract with Carry) instruction subtracts the value of register `<Rn>` and the value of NOT(Carry flag) from the value of `<shifter_operand>`, and stores the result in the destination register `<Rd>`. The condition code flags are optionally updated, based on the result.

Syntax

```
RSC{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`S` Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the `N` and `Z` flags are set according to the result of the subtraction, and the `C` and `V` flags are set according to whether the subtraction generated a borrow (unsigned underflow) and a signed overflow, respectively. The rest of the CPSR is unchanged.
- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>` Specifies the destination register of the instruction.

`<Rn>` Specifies the register that contains the second operand for the subtraction.

`<shifter_operand>`

Specifies the first operand for the subtraction. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `RSC`. Instead, see *Extending the instruction set* on page A3-27 to determine which instruction it is.

Architecture version

All

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Rd = shifter_operand - Rn - NOT(C Flag)
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = NOT BorrowFrom(shifter_operand - Rn - NOT(C Flag))
        V Flag = OverflowFrom(shifter_operand - Rn - NOT(C Flag))
```

Usage

To negate the 64-bit value in R0,R1, use the following sequence (R0 holds the least significant word) which stores the result in R2,R3:

```
    RSBS    R2,R0,#0
    RSC     R3,R1,#0
```

Notes

C flag If `S` is specified, the C flag is set to:

`1`	if no borrow occurs
`0`	if a borrow does occur.

In other words, the C flag is used as a NOT(borrow) flag. This inversion of the borrow condition is usually compensated for by subsequent instructions. For example:

- The `SBC` and `RSC` instructions use the C flag as a NOT(borrow) operand, performing a normal subtraction if C == 1 and subtracting one more than usual if C == 0.
- The HS (unsigned higher or same) and LO (unsigned lower) conditions are equivalent to CS (carry set) and CC (carry clear) respectively.

4.1.38 SBC

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	0	1	1	0	S	Rn	Rd	shifter_operand

The `SBC` (Subtract with Carry) instruction is used to synthesize multi-word subtraction. `SBC` subtracts the value of `<shifter_operand>` and the value of NOT(Carry flag) from the value of register `<Rn>`, and stores the result in the destination register `<Rd>`. The condition code flags are optionally updated, based on the result.

Syntax

```
SBC{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`S` Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the `N` and `Z` flags are set according to the result of the subtraction, and the `C` and `V` flags are set according to whether the subtraction generated a borrow (unsigned underflow) and a signed overflow, respectively. The rest of the CPSR is unchanged.
- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>` Specifies the destination register of the instruction.

`<Rn>` Specifies the register that contains the first operand for the subtraction.

`<shifter_operand>`

Specifies the second operand for the subtraction. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `SBC`. Instead, see *Extending the instruction set* on page A3-27 to determine which instruction it is.

Architecture version

All

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Rd = Rn - shifter_operand - NOT(C Flag)
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = NOT BorrowFrom(Rn - shifter_operand - NOT(C Flag))
        V Flag = OverflowFrom(Rn - shifter_operand - NOT(C Flag))
```

Usage

If register pairs R0,R1 and R2,R3 hold 64-bit values (R0 and R2 hold the least significant words), the following instructions leave the 64-bit difference in R4,R5:

```
    SUBS    R4,R0,R2
    SBC     R5,R1,R3
```

Notes

C flag If `S` is specified, the C flag is set to:

`1`	if no borrow occurs
`0`	if a borrow does occur.

In other words, the C flag is used as a NOT(borrow) flag. This inversion of the borrow condition is usually compensated for by subsequent instructions. For example:

- The `SBC` and `RSC` instructions use the C flag as a NOT(borrow) operand, performing a normal subtraction if C == 1 and subtracting one more than usual if C == 0.
- The HS (unsigned higher or same) and LO (unsigned lower) conditions are equivalent to CS (carry set) and CC (carry clear) respectively.

4.1.39 SMLAL

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	0	1	1	1	S	RdHi	RdLo	Rs	1	0	0	1	Rm

The `SMLAL` (Signed Multiply Accumulate Long) instruction multiplies the signed value of register `<Rm>` with the signed value of register `<Rs>` to produce a 64-bit product. This product is added to the 64-bit value held in `<RdHi>` and `<RdLo>`, and the sum is written back to `<RdHi>` and `<RdLo>`. The condition code flags are optionally updated, based on the result.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.
`S`	Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR by setting the N and Z flags according to the result of the multiply-accumulate. If `S` is omitted, the S bit of the instruction is set to 0 and the entire CPSR is unaffected by the instruction.
`<RdLo>`	Supplies the lower 32 bits of the value to be added to the product of `<Rm>` and `<Rs>`, and is the destination register for the lower 32 bits of the result.
`<RdHi>`	Supplies the upper 32 bits of the value to be added to the product of `<Rm>` and `<Rs>`, and is the destination register for the upper 32 bits of the result.
`<Rm>`	Holds the signed value to be multiplied with the value of `<Rs>`.
`<Rs>`	Holds the signed value to be multiplied with the value of `<Rm>`.

Architecture version

All M variants

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    RdLo = (Rm * Rs)[31:0] + RdLo /* Signed multiplication */
    RdHi = (Rm * Rs)[63:32] + RdHi + CarryFrom((Rm * Rs)[31:0] + RdLo)
    if S == 1 then
        N Flag = RdHi[31]
        Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
        C Flag = unaffected    /* See "C and V flags" note */
        V Flag = unaffected    /* See "C and V flags" note */
```

Usage

`SMLAL` multiplies signed variables to produce a 64-bit result, which is added to the 64-bit value in the two destination general-purpose registers. The result is written back to the two destination general-purpose registers.

Notes

Use of R15	Specifying R15 for register `<RdHi>`, `<RdLo>`, `<Rm>`, or `<Rs>` has UNPREDICTABLE results.
Operand restriction	`<RdHi>`, `<RdLo>`, and `<Rm>` must be three distinct registers, or the results are UNPREDICTABLE.
Early termination	If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rs>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.
C and V flags	The `SMLALS` instruction is defined to leave the C and V flags unchanged in ARM architecture version 5 and above. In earlier versions of the architecture, the values of the C and V flags were UNPREDICTABLE after an `SMLALS` instruction.

4.1.40 SMULL

31 28	27 26 25 24 23 22 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 0 1 1 0	S	RdHi	RdLo	Rs	1 0 0 1	Rm

The `SMULL` (Signed Multiply Long) instruction multiplies the signed value of register `<Rm>` with the signed value of register `<Rs>` to produce a 64-bit result. The upper 32 bits of the result are stored in `<RdHi>`. The lower 32 bits are stored in `<RdLo>`. The condition code flags are optionally updated, based on the 64-bit result.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.
`S`	Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR by setting the N and Z flags according to the result of the multiplication. If `S` is omitted, the S bit of the instruction is set to 0 and the entire CPSR is unaffected by the instruction.
`<RdLo>`	Stores the lower 32 bits of the result.
`<RdHi>`	Stores the upper 32 bits of the result.
`<Rm>`	Holds the signed value to be multiplied with the value of `<Rs>`.
`<Rs>`	Holds the signed value to be multiplied with the value of `<Rm>`.

Architecture version

All M variants

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    RdHi = (Rm * Rs)[63:32] /* Signed multiplication */
    RdLo = (Rm * Rs)[31:0]
    if S == 1 then
        N Flag = RdHi[31]
        Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
        C Flag = unaffected    /* See "C and V flags" note */
        V Flag = unaffected    /* See "C and V flags" note */
```

Usage

`SMULL` multiplies signed variables to produce a 64-bit result in two general-purpose registers.

Notes

Use of R15	Specifying R15 for register `<RdHi>`, `<RdLo>`, `<Rm>`, or `<Rs>` has UNPREDICTABLE results.
Operand restriction	`<RdHi>`, `<RdLo>`, and `<Rm>` must be three distinct registers, or the results are UNPREDICTABLE.
Early termination	If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rs>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.
C and V flags	The `SMULLS` instruction is defined to leave the C and V flags unchanged in ARM architecture version 5 and above. In earlier versions of the architecture, the values of the C and V flags were UNPREDICTABLE after an `SMULLS` instruction.

4.1.41 STC

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7 0
cond	1	1	0	P	U	N	W	0	Rn	CRd	cp_num	8_bit_word_offset

The `STC` (Store Coprocessor) instruction stores data from the coprocessor whose name is cp_num to the sequence of consecutive memory addresses calculated by `<addressing_mode>`. If no coprocessors indicate that they can execute the instruction, an Undefined Instruction exception is generated.

Syntax

```
STC{<cond>}{L}   <coproc>, <CRd>, <addressing_mode>
STC2{L}          <coproc>, <CRd>, <addressing_mode>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`STC2` Causes the condition field of the instruction to be set to 0b1111. This provides additional opcode space for coprocessor designers. The resulting instructions can only be executed unconditionally.

`L` Sets the N bit (bit[22]) in the instruction to 1 and specifies a long store (for example, double-precision instead of single-precision data transfer). If L is omitted, the N bit is 0 and the instruction specifies a short store.

`<coproc>` Specifies the name of the coprocessor, and causes the corresponding coprocessor number to be placed in the cp_num field of the instruction. The standard generic coprocessor names are p0, p1, ..., p15.

`<CRd>` Specifies the coprocessor source register of the instruction.

`<addressing_mode>`

Is described in *Addressing Mode 5 - Load and Store Coprocessor* on page A5-56. It determines the P, U, Rn, W and 8_bit_word_offset bits of the instruction.

The syntax of all forms of `<addressing_mode>` includes a *base register* `<Rn>`. Some forms also specify that the instruction modifies the base register value (this is known as *base register writeback*).

Architecture version

`STC` is in version 2 and above.

`STC2` is in version 5 and above.

Exceptions

Undefined Instruction, Data Abort

Operation

```
if ConditionPassed(cond) then
    address = start_address
    Memory[address,4] = value from Coprocessor[cp_num]
    while (NotFinished(coprocessor[cp_num]))
        address = address + 4
        Memory[address,4] = value from Coprocessor[cp_num]
    assert address == end_address
```

Usage

`STC` is useful for storing coprocessor data to memory. The L (long) option controls the N bit and could be used to distinguish between a single- and double-precision transfer for a floating-point store instruction.

Notes

Coprocessor fields Only instruction bits[31:23], bits[21:16} and bits[11:0] are defined by the ARM architecture. The remaining fields (bit[22] and bits[15:12]) are recommendations, for compatibility with ARM Development Systems.

In the case of the Unindexed addressing mode (P==0, U==1, W==0), instruction bits[7:0] are also not ARM architecture-defined, and can be used to specify additional coprocessor options.

Data abort For details of the effects of the instruction if a data abort occurs, see *Effects of data-aborted instructions* on page A2-17.

Non word-aligned addresses

Store coprocessor register instructions ignore the least significant two bits of `address`.

Alignment If an implementation includes a System Control coprocessor (see Chapter B2 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

Unimplemented coprocessor instructions

Hardware coprocessor support is optional, regardless of the architecture version. An implementation can choose to implement a subset of the coprocessor instructions, or no coprocessor instructions at all. Any coprocessor instructions that are not implemented instead cause an undefined instruction trap.

4.1.42 STM(1)

31 28	27	26	25	24	23	22	21	20	19 16	15 0
cond	1	0	0	P	U	0	W	0	Rn	register_list

This form of the `STM` (Store Multiple) instruction stores a non-empty subset (or possibly all) of the general-purpose registers to sequential memory locations.

Syntax

```
STM{<cond>}<addressing_mode>  <Rn>{!}, <registers>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`<addressing_mode>`

Is described in *Addressing Mode 4 - Load and Store Multiple* on page A5-48. It determines the P, U, and W bits of the instruction.

`<Rn>` Specifies the base register used by `<addressing_mode>`. If R15 is specified as `<Rn>`, the result is UNPREDICTABLE.

`!` Sets the W bit, causing the instruction to write a modified value back to its base register Rn as specified in *Addressing Mode 4 - Load and Store Multiple* on page A5-48. If ! is omitted, the W bit is 0 and the instruction does not change its base register in this way.

`<registers>` Is a list of registers, separated by commas and surrounded by { and }. It specifies the set of registers to be stored by the `STM` instruction.

The registers are stored in sequence, the lowest-numbered register to the lowest memory address (`start_address`), through to the highest-numbered register to the highest memory address (`end_address`).

For each of i=0 to 15, bit[i] in the register_list field of the instruction is 1 if Ri is in the list and 0 otherwise. If bits[15:0] are all zero, the result is UNPREDICTABLE.

If R15 is specified in `<registers>,` the value stored is IMPLEMENTATION DEFINED. For more details, see *Reading the program counter* on page A2-7.

Architecture version

All

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    address = start_address
    for i = 0 to 15
        if register_list[i] == 1
            Memory[address,4] = Ri
            address = address + 4
    assert end_address == address - 4
```

Usage

`STM` is useful as a block store instruction (combined with `LDM` it allows efficient block copy) and for stack operations. A single `STM` used in the sequence of a procedure can push the return address and general-purpose register values on to the stack, updating the stack pointer in the process.

Notes

Operand restrictions

If `<Rn>` is specified as `<registers>` and base register writeback is specified:

- If `<Rn>` is the lowest-numbered register specified in `<register_list>`, the original value of `<Rn>` is stored.
- Otherwise, the stored value of `<Rn>` is UNPREDICTABLE.

Data abort For details of the effects of the instruction if a data abort occurs, see *Effects of data-aborted instructions* on page A2-17.

Non word-aligned addresses

`STM` instructions ignore the least significant two bits of `address`.

Alignment If an implementation includes a System Control coprocessor (see Chapter B2 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

Time order The time order of the accesses to individual words of memory generated by this instruction is only defined in some circumstances. See *Data accesses to memory-mapped I/O* on page A2-32 for details.

4.1.43 STM (2)

31 28	27 26 25	24	23	22	21	20	19 16	15 0
cond	1 0 0	P	U	1	0	0	Rn	register_list

This form of `STM` stores a subset (or possibly all) of the User mode general-purpose registers to sequential memory locations.

Syntax

```
STM{<cond>}<addressing_mode>  <Rn>, <registers>^
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`<addressing_mode>`

Is described in *Addressing Mode 4 - Load and Store Multiple* on page A5-48. It determines the P and U bits of the instruction. Only the forms of this addressing mode with W == 0 are available for this form of the `STM` instruction.

`<Rn>` Specifies the base register used by `<addressing_mode>`. If R15 is specified as the base register `<Rn>`, the result is UNPREDICTABLE.

`<registers>` Is a list of registers, separated by commas and surrounded by { and }. It specifies the set of registers to be stored by the `STM` instruction.

The registers are stored in sequence, the lowest-numbered register to the lowest memory address (`start_address`), through to the highest-numbered register to the highest memory address (`end_address`).

For each of i=0 to 15, bit[i] in the register_list field of the instruction is 1 if Ri is in the list and 0 otherwise. If bits[15:0] are all zero, the result is UNPREDICTABLE.

If R15 is specified in `<registers>` the value stored is IMPLEMENTATION DEFINED. For more details, see *Reading the program counter* on page A2-7.

^ For an `STM` instruction, indicates that User mode registers are to be stored.

Architecture version

All

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    address = start_address
    for i = 0 to 15
        if register_list[i] == 1
            Memory[address,4] = Ri_usr
            address = address + 4
    assert end_address == address - 4
```

Usage

`STM` is used to store the User mode registers when the processor is in a privileged mode (useful when performing process swaps, and in instruction emulators).

Notes

Banked registers This instruction must not be followed by an instruction which accesses banked registers (a following `NOP` is a good way to ensure this).

Writeback Setting bit 21 (the W bit) has UNPREDICTABLE results.

User and System mode

This instruction is UNPREDICTABLE in User or System mode.

Base register mode For the purpose of address calculation, the base register is read from the current processor mode registers, not the User mode registers.

Data abort For details of the effects of the instruction if a data abort occurs, see *Effects of data-aborted instructions* on page A2-17.

Non word-aligned addresses

`STM` instructions ignore the least significant two bits of `address`.

Alignment If an implementation includes a System Control coprocessor (see Chapter B2 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

Time order The time order of the accesses to individual words of memory generated by this instruction is only defined in some circumstances. See *Data accesses to memory-mapped I/O* on page A2-32 for details.

4.1.44 STR

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	P	U	0	W	0	Rn	Rd	addr_mode

The `STR` (Store Register) instruction stores a word from register `<Rd>` to the memory address calculated by `<addressing_mode>`.

Syntax

```
STR{<cond>}  <Rd>, <addressing_mode>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rd>` Specifies the source register for the operation. If R15 is specified for `<Rd>`, the value stored is IMPLEMENTATION DEFINED. For more details, see *Reading the program counter* on page A2-7.

`<addressing_mode>`

Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-18. It determines the I, P, U, W, Rn and addr_mode bits of the instruction.

The syntax of all forms of `<addressing_mode>` includes a *base register* `<Rn>`. Some forms also specify that the instruction modifies the base register value (this is known as *base register writeback*).

Architecture version

All

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    Memory[address,4] = Rd
```

Usage

Combined with a suitable addressing mode, `STR` stores 32-bit data from a general-purpose register into memory. Using the PC as the base register allows PC-relative addressing, which facilitates position-independent code.

Notes

Operand restrictions

If `<addressing_mode>` specifies base register writeback, and the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.

Data abort For details of the effects of the instruction if a data abort occurs, see *Effects of data-aborted instructions* on page A2-17.

Non word-aligned addresses

`STR` instructions ignore the least significant two bits of `address`. So if these bits are not 0b00, the effects of `STR` are not precisely opposite to those of `LDR`.

Alignment If an implementation includes a System Control coprocessor (see Chapter B2 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

4.1.45 STRB

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	P	U	1	W	0	Rn	Rd	addr_mode

The `STRB` (Store Register Byte) instruction stores a byte from the least significant byte of register `<Rd>` to the memory address calculated by `<addressing_mode>`.

Syntax

```
STR{<cond>}B  <Rd>, <addressing_mode>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rd>` Specifies the source register for the operation. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.

`<addressing_mode>`

Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-18. It determines the I, P, U, W, Rn and addr_mode bits of the instruction.

The syntax of all forms of `<addressing_mode>` includes a *base register* `<Rn>`. Some forms also specify that the instruction modifies the base register value (this is known as *base register writeback*).

Architecture version

All

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    Memory[address,1] = Rd[7:0]
```

Usage

Combined with a suitable addressing mode, `STRB` writes the least significant byte of a general-purpose register to memory. Using the PC as the base register allows PC-relative addressing, which facilitates position-independent code.

Notes

Operand restrictions

If `<addressing_mode>` specifies base register writeback, and the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.

Data abort For details of the effects of the instruction if a data abort occurs, see *Effects of data-aborted instructions* on page A2-17.

4.1.46 STRBT

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	0	U	1	1	0	Rn	Rd	addr_mode

The `STRBT` (Store Register Byte with Translation) instruction stores a byte from the least significant byte of register `<Rd>` to the memory address calculated by `<post_indexed_addressing_mode>`. If the instruction is executed when the processor is in a privileged mode, the memory system is signaled to treat the access as if the processor were in User mode.

Syntax

```
STR{<cond>}BT  <Rd>, <post_indexed_addressing_mode>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rd>` Specifies the source register for the operation. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.

`<post_indexed_addressing_mode>`
Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-18. It determines the I, U, Rn and addr_mode bits of the instruction. Only post-indexed forms of Addressing Mode 2 are available for this instruction. These forms have P == 0 and W == 0, where P and W are bit[24] and bit[21] respectively. This instruction uses P == 0 and W == 1 instead, but the addressing mode is the same in all other respects.

The syntax of all forms of `<post_indexed_addressing_mode>` includes a *base register* `<Rn>`. All forms also specify that the instruction modifies the base register value (this is known as *base register writeback*).

Architecture version

All

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    Memory[address,1] = Rd[7:0]
```

Usage

`STRBT` can be used by a (privileged) exception handler that is emulating a memory access instruction which would normally execute in User mode. The access is restricted as if it had User mode privilege.

Notes

User mode If this instruction is executed in User mode, an ordinary User mode access is performed.

Operand restrictions

If the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.

Data abort For details of the effects of the instruction if a data abort occurs, see *Effects of data-aborted instructions* on page A2-17.

4.1.47 STRH

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0	P	U	I	W	0	Rn	Rd	addr_mode	1 0 1 1	addr_mode

The `STRH` (Store Register Halfword) instruction stores a halfword from the least significant halfword of register `<Rd>` to the memory address calculated by `<addressing_mode>`. If the address is not halfword-aligned, the result is UNPREDICTABLE.

Syntax

```
STR{<cond>}H  <Rd>, <addressing_mode>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the source register for the operation. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.
`<addressing_mode>`	Is described in *Addressing Mode 3 - Miscellaneous Loads and Stores* on page A5-34. It determines the P, U, I, W, Rn and addr_mode bits of the instruction. The syntax of all forms of `<addressing_mode>` includes a *base register* `<Rn>`. Some forms also specify that the instruction modifies the base register value (this is known as *base register writeback*).

Architecture version

Version 4 and above

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    if address[0] == 0
        data = Rd[15:0]
    else /* address[0] == 1 */
        data = UNPREDICTABLE
    Memory[address,2] = data
```

Usage

Combined with a suitable addressing mode, `STRH` allows 16-bit data from a general-purpose register to be stored to memory. Using the PC as the base register allows PC-relative addressing, to facilitate position-independent code.

Notes

Operand restrictions If `<addressing_mode>` specifies base register writeback, and the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.

Data abort For details of the effects of the instruction if a data abort occurs, see *Effects of data-aborted instructions* on page A2-17.

Non halfword-aligned addresses

If the store address is not halfword-aligned, the stored value is UNPREDICTABLE.

Alignment If an implementation includes a System Control coprocessor (see Chapter B2 *The System Control Coprocessor*), and alignment checking is enabled, an address with bit[0] != 0 causes an alignment exception.

4.1.48 STRT

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	0	U	0	1	0	Rn	Rd	addr_mode

The `STRT` (Store Register with Translation) instruction stores a word from register `<Rd>` to the memory address calculated by `<post_indexed_addressing_mode>`. If the instruction is executed when the processor is in a privileged mode, the memory system is signaled to treat the access as if the processor was in User mode.

Syntax

```
STR{<cond>}T  <Rd>, <post_indexed_addressing_mode>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rd>` Specifies the source register for the operation. If R15 is specified for `<Rd>`, the value stored is IMPLEMENTATION DEFINED. For more details, see *Reading the program counter* on page A2-7.

`<post_indexed_addressing_mode>`

Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-18. It determines the I, U, Rn and addr_mode bits of the instruction. Only post-indexed forms of Addressing Mode 2 are available for this instruction. These forms have P == 0 and W == 0, where P and W are bit[24] and bit[21] respectively. This instruction uses P == 0 and W == 1 instead, but the addressing mode is the same in all other respects.

The syntax of all forms of `<post_indexed_addressing_mode>` includes a *base register* `<Rn>`. All forms also specify that the instruction modifies the base register value (this is known as *base register writeback*).

Architecture version

All

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    Memory[address,4] = Rd
```

Usage

`STRT` can be used by a (privileged) exception handler that is emulating a memory access instruction that would normally execute in User mode. The access is restricted as if it had User mode privilege.

Notes

User mode If this instruction is executed in User mode, an ordinary User mode access is performed.

Operand restrictions
If the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.

Data abort For details of the effects of the instruction if a data abort occurs, see *Effects of data-aborted instructions* on page A2-17.

Alignment If an implementation includes a System Control coprocessor (see Chapter B2 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

4.1.49 SUB

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	0 0 1 0	S	Rn	Rd	shifter_operand

The `SUB` (Subtract) instruction subtracts the value of `<shifter_operand>` from the value of register `<Rn>`, and stores the result in the destination register `<Rd>`. The condition code flags are optionally updated, based on the result.

Syntax

```
SUB{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.

`S` Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the `N` and `Z` flags are set according to the result of the subtraction, and the `C` and `V` flags are set according to whether the subtraction generated a borrow (unsigned underflow) and a signed overflow, respectively. The rest of the CPSR is unchanged.
- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>` Specifies the destination register of the instruction.

`<Rn>` Specifies the register that contains the first operand for the subtraction.

`<shifter_operand>`

Specifies the second operand for the subtraction. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `SUB`. Instead, see *Extending the instruction set* on page A3-27 to determine which instruction it is.

Architecture version

All

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Rd = Rn - shifter_operand
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = NOT BorrowFrom(Rn - shifter_operand)
        V Flag = OverflowFrom(Rn - shifter_operand)
```

Usage

`SUB` is used to subtract one value from another to produce a third. To decrement a register value (in `Rx`) use:

```
    SUBS Ri, Ri, #1
```

`SUBS` is useful as a loop counter decrement, as the loop branch can test the flags for the appropriate termination condition, without the need for a compare instruction:

```
    CMP Rx, #0
```

This both decrements the loop counter in Ri and checks whether it has reached zero.

The form of this instruction with the PC as its destination register and the S bit set can be used to return from interrupts and various other types of exception. See *Exceptions* on page A2-13 for more details.

Notes

C flag If `S` is specified, the C flag is set to:

`1`	if no borrow occurs
`0`	if a borrow does occur.

In other words, the C flag is used as a NOT(borrow) flag. This inversion of the borrow condition is usually compensated for by subsequent instructions. For example:

- The `SBC` and `RSC` instructions use the C flag as a NOT(borrow) operand, performing a normal subtraction if C == 1 and subtracting one more than usual if C == 0.
- The HS (unsigned higher or same) and LO (unsigned lower) conditions are equivalent to CS (carry set) and CC (carry clear) respectively.

4.1.50 SWI

31 28	27	26	25	24	23 0
cond	1	1	1	1	immed_24

The `SWI` (Software Interrupt) instruction causes a SWI exception (see *Exceptions* on page A2-13).

Syntax

`SWI{<cond>}  <immed_24>`

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.
`<immed_24>`	Is a 24-bit immediate value that is put into bits[23:0] of the instruction. This value is ignored by the ARM processor, but can be used by an operating system SWI exception handler to determine what operating system service is being requested (see *Usage* on page A4-101 below for more details).

Architecture version

All

Exceptions

Software interrupt

Operation

```
if ConditionPassed(cond) then
    R14_svc   = address of next instruction after the SWI instruction
    SPSR_svc  = CPSR
    CPSR[4:0] = 0b10011               /* Enter Supervisor mode */
    CPSR[5]   = 0                     /* Execute in ARM state */
    /* CPSR[6] is unchanged */
    CPSR[7]   = 1                     /* Disable normal interrupts */
    if high vectors configured then
        PC    = 0xFFFF0008
    else
        PC    = 0x00000008
```

Usage

The `SWI` instruction is used as an operating system service call. The method used to select which operating system service is required is specified by the operating system, and the SWI exception handler for the operating system determines and provides the requested service. Two typical methods are:

- The 24-bit immediate in the instruction specifies which service is required, and any parameters needed by the selected service are passed in general-purpose registers.
- The 24-bit immediate in the instruction is ignored, general-purpose register R0 is used to select which service is wanted, and any parameters needed by the selected service are passed in other general-purpose registers.

4.1.51 SWP

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	1	0	0	0	0	Rn	Rd	SBZ	1	0	0	1	Rm

The `SWP` (Swap) instruction swaps a word between registers and memory. `SWP` loads a word from the memory address given by the value of register `<Rn>`. The value of register `<Rm>` is then stored to the memory address given by the value of `<Rn>`, and the original loaded value is written to register `<Rd>`. If the same register is specified for `<Rd>` and `<Rm>`, this instruction swaps the value of the register and the value at the memory address.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register for the instruction.
`<Rm>`	Contains the value that is stored to memory.
`<Rn>`	Contains the memory address to load from.

Architecture version

Version 3 and above, plus version 2a

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    if Rn[1:0] == 0b00 then
        temp = Memory[Rn,4]
    else if Rn[1:0] == 0b01 then
        temp = Memory[Rn,4] Rotate_Right 8
    else if Rn[1:0] == 0b10 then
        temp = Memory[Rn,4] Rotate_Right 16
    else /* Rn[1:0] == 0b11 */
        temp = Memory[Rn,4] Rotate_Right 24

    Memory[Rn,4] = Rm
    Rd = temp
```

Usage

The `SWP` instruction can be used to implement semaphores. For sample code, see *Semaphore instructions* on page A9-11.

Notes

Non word-aligned addresses	If the address is not word-aligned, the loaded value is rotated right by 8 times the value of Rn[1:0]. The stored value is not rotated.
Use of R15	If R15 is specified for `<Rd>`, `<Rn>`, or `<Rm>`, the result is UNPREDICTABLE.
Operand restrictions	If the same register is specified as `<Rn>` and `<Rm>`, or `<Rn>` and `<Rd>`, the result is UNPREDICTABLE.
Data abort	If a data abort is signaled on either the load access or the store access, the loaded value is not written to `<Rd>`. If a data abort is signaled on the load access, the store access does not occur.
Alignment	If an implementation includes a System Control coprocessor (see Chapter B2 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

4.1.52 SWPB

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 1 0 1 0 0	Rn	Rd	SBZ	1 0 0 1	Rm

The `SWPB` (Swap Byte) instruction swaps a byte between registers and memory. `SWPB` loads a byte from the memory address given by the value of register `<Rn>`. The value of the least significant byte of register `<Rm>` is stored to the memory address given by `<Rn>`, the original loaded value is zero-extended to a 32-bit word, and the word is written to register `<Rd>`. If the same register is specified for `<Rd>` and `<Rm>`, this instruction swaps the value of the least significant byte of the register and the byte value at the memory address.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register for the instruction.
`<Rm>`	Contains the value that is stored to memory.
`<Rn>`	Contains the memory address to load from.

Architecture version

Version 3 and above, plus version 2a

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    temp = Memory[Rn,1]
    Memory[Rn,1] = Rm[7:0]
    Rd = temp
```

Usage

The `SWPB` instruction can be used to implement semaphores, in a similar manner to that shown for `SWP` instructions in *Semaphore instructions* on page A9-11.

Notes

Use of R15	If R15 is specified for `<Rd>`, `<Rn>`, or `<Rm>`, the result is UNPREDICTABLE.
Operand restrictions	If the same register is specified as `<Rn>` and `<Rm>`, or `<Rn>` and `<Rd>`, the result is UNPREDICTABLE.
Data abort	If a data abort is signaled on either the load access or the store access, the loaded value is not written to `<Rd>`. If a data abort is signaled on the load access, the store access does not occur.

4.1.53 TEQ

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	1	0	0	1	1	Rn	SBZ	shifter_operand

The `TEQ` (Test Equivalence) instruction compares a register value with another arithmetic value. The condition flags are updated, based on the result of logically exclusive-ORing the two values, so that subsequent instructions can be conditionally executed.

Syntax

```
TEQ{<cond>}  <Rn>, <shifter_operand>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rn>`	Specifies the register that contains the first operand for the comparison.
`<shifter_operand>`	Specifies the second operand for the comparison. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option sets the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) in the instruction. If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `TEQ`. Instead, see *Extending the instruction set* on page A3-27 to determine which instruction it is.

Architecture version

All

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    alu_out = Rn EOR shifter_operand
    N Flag = alu_out[31]
    Z Flag = if alu_out == 0 then 1 else 0
    C Flag = shifter_carry_out
    V Flag = unaffected
```

Usage

`TEQ` is used to test if two values are equal, without affecting the V flag (as `CMP` does). The C flag is also unaffected in many cases. `TEQ` is also useful for testing whether two values have the same sign. After the comparison, the N flag is the logical Exclusive OR of the sign bits of the two operands.

4.1.54 TST

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	1 0 0 0	1	Rn	SBZ	shifter_operand

The `TST` (Test) instruction compares a register value with another arithmetic value. The condition flags are updated, based on the result of logically ANDing the two values, so that subsequent instructions can be conditionally executed.

Syntax

```
TST{<cond>}  <Rn>, <shifter_operand>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rn>`	Specifies the register that contains the first operand for the comparison.
`<shifter_operand>`	Specifies the second operand for the comparison. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction. If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `TST`. Instead, see *Extending the instruction set* on page A3-27 to determine which instruction it is.

Architecture version

All

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    alu_out = Rn AND shifter_operand
    N Flag = alu_out[31]
    Z Flag = if alu_out == 0 then 1 else 0
    C Flag = shifter_carry_out
    V Flag = unaffected
```

Usage

`TST` is used to determine whether a particular subset of register bits includes at least one set bit. A very common use for `TST` is to test whether a single bit is set or clear.

4.1.55 UMLAL

31 28	27 26 25 24 23 22 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 0 1 0 1	S	RdHi	RdLo	Rs	1 0 0 1	Rm

The `UMLAL` (Unsigned Multiply Accumulate Long) instruction multiplies the unsigned value of register `<Rm>` with the unsigned value of register `<Rs>` to produce a 64-bit product. This product is added to the 64-bit value held in `<RdHi>` and `<RdLo>`, and the sum is written back to `<RdHi>` and `<RdLo>`. The condition code flags are optionally updated, based on the result.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.
`S`	Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR by setting the N and Z flags according to the result of the multiply-accumulate. If `S` is omitted, the S bit of the instruction is set to 0 and the entire CPSR is unaffected by the instruction.
`<RdLo>`	Supplies the lower 32 bits of the value to be added to the product of `<Rm>` and `<Rs>`, and is the destination register for the lower 32 bits of the result.
`<RdHi>`	Supplies the upper 32 bits of the value to be added to the product of `<Rm>` and `<Rs>`, and is the destination register for the upper 32 bits of the result.
`<Rm>`	Holds the signed value to be multiplied with the value of `<Rs>`.
`<Rs>`	Holds the signed value to be multiplied with the value of `<Rm>`.

Architecture version

All M variants

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    RdLo = (Rm * Rs)[31:0] + RdLo    /* Unsigned multiplication */
    RdHi = (Rm * Rs)[63:32] + RdHi + CarryFrom((Rm * Rs)[31:0] + RdLo)
    if S == 1 then
        N Flag = RdHi[31]
        Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
        C Flag = unaffected    /* See "C and V flags" note */
        V Flag = unaffected    /* See "C and V flags" note */
```

Usage

`UMLAL` multiplies unsigned variables to produce a 64-bit result, which is added to the 64-bit value in the two destination general-purpose registers. The result is written back to the two destination general-purpose registers.

Notes

Use of R15 Specifying R15 for register `<RdHi>`, `<RdLo>`, `<Rm>`, or `<Rs>` has UNPREDICTABLE results.

Operand restriction `<RdHi>`, `<RdLo>`, and `<Rm>` must be three distinct registers, or the results are UNPREDICTABLE.

Early termination If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rs>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

C and V flags The `UMLALS` instruction is defined to leave the C and V flags unchanged in ARM architecture version 5 and above. In earlier versions of the architecture, the values of the C and V flags were UNPREDICTABLE after a `UMLALS` instruction.

4.1.56 UMULL

31 28	27 26 25 24 23 22 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 0 1 0 0	S	RdHi	RdLo	Rs	1 0 0 1	Rm

The `UMULL` (Unsigned Multiply Long) instruction multiplies the unsigned value of register `<Rm>` with the unsigned value of register `<Rs>` to produce a 64-bit result. The upper 32 bits of the result are stored in `<RdHi>`. The lower 32 bits are stored in `<RdLo>`. The condition code flags are optionally updated, based on the 64-bit result.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If `<cond>` is omitted, the `AL` (always) condition is used.
`S`	Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR by setting the N and Z flags according to the result of the multiplication. If `S` is omitted, the S bit of the instruction is set to 0 and the entire CPSR is unaffected by the instruction.
`<RdLo>`	Stores the lower 32 bits of the result.
`<RdHi>`	Stores the upper 32 bits of the result.
`<Rm>`	Holds the signed value to be multiplied with the value of `<Rs>`.
`<Rs>`	Holds the signed value to be multiplied with the value of `<Rm>`.

Architecture version

All M variants

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    RdHi = (Rm * Rs)[63:32]    /* Unsigned multiplication */
    RdLo = (Rm * Rs)[31:0]
    if S == 1 then
        N Flag = RdHi[31]
        Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
        C Flag = unaffected    /* See "C and V flags" note */
        V Flag = unaffected    /* See "C and V flags" note */
```

Usage

`UMULL` multiplies unsigned variables to produce a 64-bit result in two general-purpose registers.

Notes

Use of R15	Specifying R15 for register `<RdHi>`, `<RdLo>`, `<Rm>`, or `<Rs>` has UNPREDICTABLE results.
Operand restriction	`<RdHi>`, `<RdLo>`, and `<Rm>` must be three distinct registers, or the results are UNPREDICTABLE.
Early termination	If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rs>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.
C and V flags	The `UMULLS` instruction is defined to leave the C and V flags unchanged in ARM architecture version 5 and above. In earlier versions of the architecture, the values of the C and V flags were UNPREDICTABLE after a `UMULLS` instruction.

4.2 ARM instructions and architecture versions

Table 4-1 shows which ARM instructions are present in each current ARM architecture version.

Table 4-1 ARM instructions by architecture version

Instruction	v3, v3M	v4, v4xM	v4T, v4TxM	v5, v5xM, v5T, v5TxM	v5TE, v5TExP
`ADC`	Yes	Yes	Yes	Yes	Yes
`ADD`	Yes	Yes	Yes	Yes	Yes
`AND`	Yes	Yes	Yes	Yes	Yes
`B`	Yes	Yes	Yes	Yes	Yes
`BIC`	Yes	Yes	Yes	Yes	Yes
`BKPT`	No	No	No	Yes	Yes
`BL`	Yes	Yes	Yes	Yes	Yes
`BLX` (both forms)	No	No	No	Yes	Yes
`BX`	No	No	Yes	Yes	Yes
`CDP`	Yes	Yes	Yes	Yes	Yes
`CDP2`	No	No	No	Yes	Yes
`CLZ`	No	No	No	Yes	Yes
`CMN`	Yes	Yes	Yes	Yes	Yes
`CMP`	Yes	Yes	Yes	Yes	Yes
`EOR`	Yes	Yes	Yes	Yes	Yes
`LDC`	Yes	Yes	Yes	Yes	Yes
`LDC2`	No	No	No	Yes	Yes
`LDM` (all forms)	Yes	Yes	Yes	Yes	Yes
`LDR`	Yes	Yes	Yes	Yes	Yes
`LDRB`	Yes	Yes	Yes	Yes	Yes
`LDRD`	No	No	No	No	Only v5TE
`LDRBT`	Yes	Yes	Yes	Yes	Yes

Table 4-1 ARM instructions by architecture version *(Continued)*

Instruction	v3, v3M	v4, v4xM	v4T, v4TxM	v5, v5xM, v5T, v5TxM	v5TE, v5TExP
`LDRH`	No	Yes	Yes	Yes	Yes
`LDRSB`	No	Yes	Yes	Yes	Yes
`LDRSH`	No	Yes	Yes	Yes	Yes
`LDRT`	Yes	Yes	Yes	Yes	Yes
`MCR`	Yes	Yes	Yes	Yes	Yes
`MCR2`	No	No	No	Yes	Yes
`MCRR`	No	No	No	No	Only v5TE
`MLA`	Yes	Yes	Yes	Yes	Yes
`MOV`	Yes	Yes	Yes	Yes	Yes
`MRC`	Yes	Yes	Yes	Yes	Yes
`MRC2`	No	No	No	Yes	Yes
`MRRC`	No	No	No	No	Only v5TE
`MRS`	Yes	Yes	Yes	Yes	Yes
`MSR`	Yes	Yes	Yes	Yes	Yes
`MUL`	Yes	Yes	Yes	Yes	Yes
`MVN`	Yes	Yes	Yes	Yes	Yes
`ORR`	Yes	Yes	Yes	Yes	Yes
`PLD`	No	No	No	No	Only v5TE
`QADD`	No	No	No	No	Yes
`QDADD`	No	No	No	No	Yes
`QDSUB`	No	No	No	No	Yes
`QSUB`	No	No	No	No	Yes
`RSB`	Yes	Yes	Yes	Yes	Yes
`RSC`	Yes	Yes	Yes	Yes	Yes

Table 4-1 ARM instructions by architecture version *(Continued)*

Instruction	v3, v3M	v4, v4xM	v4T, v4TxM	v5, v5xM, v5T, v5TxM	v5TE, v5TExP
`SBC`	Yes	Yes	Yes	Yes	Yes
`SMLAL`	Only v3M	Only v4	Only v4T	Only v5/v5T	Yes
`SMLA<x><y>`	No	No	No	No	Yes
`SMLAL<x><y>`	No	No	No	No	Yes
`SMLAW<y>`	No	No	No	No	Yes
`SMULL`	Only v3M	Only v4	Only v4T	Only v5/v5T	Yes
`SMUL<x><y>`	No	No	No	No	Yes
`SMULW<y>`	No	No	No	No	Yes
`STC`	Yes	Yes	Yes	Yes	Yes
`STC2`	No	No	No	Yes	Yes
`STM` (both forms)	Yes	Yes	Yes	Yes	Yes
`STR`	Yes	Yes	Yes	Yes	Yes
`STRB`	Yes	Yes	Yes	Yes	Yes
`STRBT`	Yes	Yes	Yes	Yes	Yes
`STRD`	No	No	No	No	Only v5TE
`STRH`	No	Yes	Yes	Yes	Yes
`STRT`	Yes	Yes	Yes	Yes	Yes
`SUB`	Yes	Yes	Yes	Yes	Yes
`SWI`	Yes	Yes	Yes	Yes	Yes
`SWP`	Yes	Yes	Yes	Yes	Yes
`SWPB`	Yes	Yes	Yes	Yes	Yes
`TEQ`	Yes	Yes	Yes	Yes	Yes

Table 4-1 ARM instructions by architecture version *(Continued)*

Instruction	v3, v3M	v4, v4xM	v4T, v4TxM	v5, v5xM, v5T, v5TxM	v5TE, v5TExP
`TST`	Yes	Yes	Yes	Yes	Yes
`UMLAL`	Only v3M	Only v4	Only v4T	Only v5/v5T	Yes
`UMULL`	Only v3M	Only v4	Only v4T	Only v5/v5T	Yes

Chapter A5
ARM Addressing Modes

This chapter describes each of the five addressing modes used with ARM instructions. The chapter contains the following sections:

5.1 Addressing Mode 1 - Data-processing operands

There are 11 addressing modes used to calculate the `<shifter_operand>` in an ARM data-processing instruction. The general instruction syntax is:

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

where `<shifter_operand>` is one of the following 11 options:

1. `#<immediate>`

   See *Data-processing operands - Immediate* on page A5-6.

2. `<Rm>`

   See *Data-processing operands - Register* on page A5-8.

3. `<Rm>, LSL #<shift_imm>`

   See *Data-processing operands - Logical shift left by immediate* on page A5-9.

4. `<Rm>, LSL <Rs>`

   See *Data-processing operands - Logical shift left by register* on page A5-10.

5. `<Rm>, LSR #<shift_imm>`

   See *Data-processing operands - Logical shift right by immediate* on page A5-11.

6. `<Rm>, LSR <Rs>`

   See *Data-processing operands - Logical shift right by register* on page A5-12.

7. `<Rm>, ASR #<shift_imm>`

   See *Data-processing operands - Arithmetic shift right by immediate* on page A5-13.

8. `<Rm>, ASR <Rs>`

   See *Data-processing operands - Arithmetic shift right by register* on page A5-14.

9. `<Rm>, ROR #<shift_imm>`

   See *Data-processing operands - Rotate right by immediate* on page A5-15.

10. `<Rm>, ROR <Rs>`

    See *Data-processing operands - Rotate right by register* on page A5-16.

11. `<Rm>, RRX`

    See *Data-processing operands - Rotate right with extend* on page A5-17.

5.1.1 Encoding

The following diagrams show the encodings for this addressing mode:

32-bit immediate

31 28	27	26	25	24 21	20	19 16	15 12	11 8	7 0
cond	0	0	1	opcode	S	Rn	Rd	rotate_imm	immed_8

Immediate shifts

31 28	27 26	25	24 21	20	19 16	15 12	11 7	6 5	4	3 0
cond	0 0	0	opcode	S	Rn	Rd	shift_imm	shift	0	Rm

Register shifts

31 28	27 26	25	24 21	20	19 16	15 12	11 8	7	6 5	4	3 0
cond	0 0	0	opcode	S	Rn	Rd	Rs	0	shift	1	Rm

opcode	Specifies the operation of the instruction.
S bit	Indicates that the instruction updates the condition codes.
Rd	Specifies the destination register.
Rn	Specifies the first source operand register.
Bits[11:0]	The fields within bits[11:0] are collectively called a *shifter operand*. This is described in *The shifter operand* on page A5-4.
Bit[25]	Is referred to as the I bit, and is used to distinguish between an immediate shifter operand and a register-based shifter operand.

If all three of the following bits have the values shown, the instruction is not a data-processing instruction, but lies in the arithmetic or Load/Store instruction extension space:

```
bit[25]  == 0
bit[4]   == 1
bit[7]   == 1
```

See *Extending the instruction set* on page A3-27 for more information.

5.1.2 The shifter operand

As well as producing the shifter operand, the shifter produces a carry-out which some instructions write into the Carry Flag. The default register operand (register Rm specified with no shift) uses the form register shift left by immediate, with the immediate set to zero.

The shifter operand takes one of the following three basic formats.

Immediate operand value

An immediate operand value is formed by rotating an 8-bit constant (in a 32-bit word) by an even number of bits (0,2,4,8...26,28,30). Therefore, each instruction contains an 8-bit constant and a 4-bit rotate to be applied to that constant.

Some valid constants are:

```
    0xFF,0x104,0xFF0,0xFF00,0xFF000,0xFF000000,0xF000000F
```

Some invalid constants are:

```
    0x101,0x102,0xFF1,0xFF04,0xFF003,0xFFFFFFFF,0xF000001F
```

For example:

```
MOV    R0, #0                      ; Move zero to R0
ADD    R3, R3, #1                  ; Add one to the value of register 3
CMP    R7, #1000                   ; Compare value of R7 with 1000
BIC    R9, R8, #0xFF00             ; Clear bits 8-15 of R8 and store in R9
```

Register operand value

A register operand value is simply the value of a register. The value of the register is used directly as the operand to the data-processing instruction. For example:

```
MOV    R2, R0                      ; Move the value of R0 to R2
ADD    R4, R3, R2                  ; Add R2 to R3, store result in R4
CMP    R7, R8                      ; Compare the value of R7 and R8
```

Shifted register operand value

A shifted register operand value is the value of a register, shifted (or rotated) before it is used as the data-processing operand. There are five types of shift:

`ASR`	Arithmetic shift right
`LSL`	Logical shift left
`LSR`	Logical shift right
`ROR`	Rotate right
`RRX`	Rotate right with extend.

The number of bits to shift by is specified either as an immediate or as the value of a register. For example:

```
MOV    R2,  R0, LSL  #2          ; Shift R0 left by 2, write to R2, (R2=R0x4)
ADD    R9,  R5, R5,  LSL #3      ; R9 = R5 + R5 x 8 or R9 = R5 x 9
RSB    R9,  R5, R5,  LSL #3      ; R9 = R5 x 8 - R5 or R9 = R5 x 7
SUB    R10, R9, R8,  LSR #4      ; R10 = R9 - R8 / 16
MOV    R12, R4, ROR  R3          ; R12 = R4 rotated right by value of R3
```

5.1.3 Data-processing operands - Immediate

31 28	27 26 25	24 21	20	19 16	15 12	11 8	7 0
cond	0 0 1	opcode	S	Rn	Rd	rotate_imm	immed_8

This data-processing operand provides a constant (defined in the instruction) operand to a data-processing instruction.

The `shifter_operand` value is formed by rotating (to the right) an 8-bit immediate value to any even bit position in a 32-bit word. If the rotate immediate is zero, the carry-out from the shifter is the value of the C flag, otherwise, it is set to bit[31] of the value of `<shifter_operand>`.

Syntax

`#<immediate>`

where:

`<immediate>`	Specifies the immediate constant wanted. It is encoded in the instruction as an 8-bit immediate (immed_8) and a 4-bit immediate (rotate_imm), so that `<immediate>` is equal to the result of rotating immed_8 right by (2 * rotate_imm) bits.

Architecture version

All

Operation

```
shifter_operand = immed_8 Rotate_Right (rotate_imm * 2)
if rotate_imm == 0 then
    shifter_carry_out = C flag
else /* rotate_imm != 0 */
    shifter_carry_out = shifter_operand[31]
```

Notes

Legitimate immediates

Not all 32-bit immediates are legitimate. Only those that can be formed by rotating an 8-bit immediate right by an even amount are valid 32-bit immediates for this format.

Encoding Some values of `<immediate>` have more than one possible encoding. For example, a value of `0x3F0` could be encoded as:

immed_8 == `0x3F`, rotate_imm == `0xE`

or as:

immed_8 == `0xFC`, rotate_imm == `0xF`

When more than one encoding is available, an assembler needs to choose the correct one to use, as follows:

- If `<immediate>` lies in the range 0 to `0xFF`, an encoding with rotate_imm == 0 is available. The assembler must choose that encoding. (Choosing another encoding would affect how some instructions set the C flag.)
- Otherwise, it is recommended that the encoding with the smallest value of rotate_imm is chosen. (This choice does not affect instruction functionality.)

For more precise control of the encoding, the instruction fields can be specified directly by using the syntax:

```
#<immed_8>, <rotate_amount>
```

where `<rotate_amount>` = 2 * rotate_imm.

5.1.4 Data-processing operands - Register

31 28	27 26 25	24 21	20	19 16	15 12	11 10 9 8 7	6 5 4	3 0
cond	0 0 0	opcode	S	Rn	Rd	0 0 0 0 0	0 0 0	Rm

This data-processing operand provides the value of a register directly. The carry-out from the shifter is the C flag.

Syntax

`<Rm>`

where:

`<Rm>`	Specifies the register whose value is the instruction operand.

Architecture version

All

Operation

```
shifter_operand = Rm
shifter_carry_out = C Flag
```

Notes

Encoding This instruction is encoded as a logical shift left by immediate (see *Data-processing operands - Logical shift left by immediate* on page A5-9) with a shift of zero (shift_imm == 0).

Use of R15 If R15 is specified as register Rm or Rn, the value used is the address of the current instruction plus 8.

5.1.5 Data-processing operands - Logical shift left by immediate

31 28	27	26	25	24 21	20	19 16	15 12	11 7	6	5	4	3 0
cond	0	0	0	opcode	S	Rn	Rd	shift_imm	0	0	0	Rm

This data-processing operand is used to provide either the value of a register directly (lone register operand, as described in *Data-processing operands - Register* on page A5-8), or the value of a register shifted left (multiplied by a constant power of two).

This instruction operand is the value of register Rm, logically shifted left by an immediate value in the range 0 to 31. Zeros are inserted into the vacated bit positions. The carry-out from the shifter is the last bit shifted out, or the C flag if no shift is specified.

Syntax

```
<Rm>, LSL #<shift_imm>
```

where:

`<Rm>`	Specifies the register whose value is to be shifted.
`LSL`	Indicates a logical shift left.
`<shift_imm>`	Specifies the shift. This is a value between 0 and 31.

Architecture version

All

Operation

```
if shift_imm == 0 then /* Register Operand */
    shifter_operand = Rm
    shifter_carry_out = C Flag
else /* shift_imm > 0 */
    shifter_operand = Rm Logical_Shift_Left shift_imm
    shifter_carry_out = Rm[32 - shift_imm]
```

Notes

Default shift If the value of `<shift_imm> == 0`, the operand can be written as just `<Rm>` (see *Data-processing operands - Register* on page A5-8).

Use of R15 If R15 is specified as register Rm or Rn, the value used is the address of the current instruction plus 8.

5.1.6 Data-processing operands - Logical shift left by register

31 28	27 26 25	24 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0	opcode	S	Rn	Rd	Rs	0 0 0 1	Rm

This data-processing operand is used to provide the value of a register multiplied by a variable power of two.

This instruction operand is the value of register Rm, logically shifted left by the value in the least significant byte of register Rs. Zeros are inserted into the vacated bit positions. The carry-out from the shifter is the last bit shifted out, which is zero if the shift amount is more than 32, or the C flag if the shift amount is zero.

Syntax

`<Rm>, LSL <Rs>`

where:

`<Rm>`	Specifies the register whose value is to be shifted.
`LSL`	Indicates a logical shift left.
`<Rs>`	Is the register containing the value of the shift.

Architecture version

All

Operation

```
if Rs[7:0] == 0 then
    shifter_operand = Rm
    shifter_carry_out = C Flag
else if Rs[7:0] < 32 then
    shifter_operand = Rm Logical_Shift_Left Rs[7:0]
    shifter_carry_out = Rm[32 - Rs[7:0]]
else if Rs[7:0] == 32 then
    shifter_operand = 0
    shifter_carry_out = Rm[0]
else /* Rs[7:0] > 32 */
    shifter_operand = 0
    shifter_carry_out = 0
```

Notes

Use of R15	Specifying R15 as register Rd, register Rm, register Rn, or register Rs has UNPREDICTABLE results.

5.1.7 Data-processing operands - Logical shift right by immediate

31 28	27 26 25	24 21	20	19 16	15 12	11 7	6 5 4	3 0
cond	0 0 0	opcode	S	Rn	Rd	shift_imm	0 1 0	Rm

This data-processing operand is used to provide the unsigned value of a register shifted right (divided by a constant power of two).

This instruction operand is the value of register Rm, logically shifted right by an immediate value in the range 1 to 32. Zeros are inserted into the vacated bit positions. The carry-out from the shifter is the last bit shifted out.

Syntax

```
<Rm>, LSR #<shift_imm>
```

where:

`<Rm>`	Specifies the register whose value is to be shifted.
`LSR`	Indicates a logical shift right.
`<shift_imm>`	Specifies the shift. This is an immediate value between 1 and 32. (A shift by 32 is encoded by shift_imm == 0.)

Architecture version

All

Operation

```
if shift_imm == 0 then
    shifter_operand = 0
    shifter_carry_out = Rm[31]
else /* shift_imm > 0 */
    shifter_operand = Rm Logical_Shift_Right shift_imm
    shifter_carry_out = Rm[shift_imm - 1]
```

Notes

Use of R15	If R15 is specified as register Rm or Rn, the value used is the address of the current instruction plus 8.

5.1.8 Data-processing operands - Logical shift right by register

31 28	27 26 25	24 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0	opcode	S	Rn	Rd	Rs	0 0 1 1	Rm

This data-processing operand is used to provide the unsigned value of a register shifted right (divided by a variable power of two).

It is produced by the value of register Rm, logically shifted right by the value in the least significant byte of register Rs. Zeros are inserted into the vacated bit positions. The carry-out from the shifter is the last bit shifted out, which is zero if the shift amount is more than 32, or the C flag if the shift amount is zero.

Syntax

`<Rm>, LSR <Rs>`

where:

`<Rm>`	Specifies the register whose value is to be shifted.
`LSR`	Indicates a logical shift right.
`<Rs>`	Is the register containing the value of the shift.

Architecture version

All

Operation

```
if Rs[7:0] == 0 then
    shifter_operand = Rm
    shifter_carry_out = C Flag
else if Rs[7:0] < 32 then
    shifter_operand = Rm Logical_Shift_Right Rs[7:0]
    shifter_carry_out = Rm[Rs[7:0] - 1]
else if Rs[7:0] == 32 then
    shifter_operand = 0
    shifter_carry_out = Rm[31]
else /* Rs[7:0] > 32 */
    shifter_operand = 0
    shifter_carry_out = 0
```

Notes

Use of R15	Specifying R15 as register Rd, register Rm, register Rn, or register Rs has UNPREDICTABLE results.

5.1.9 Data-processing operands - Arithmetic shift right by immediate

31 28	27 26 25	24 21	20	19 16	15 12	11 7	6 5 4	3 0
cond	0 0 0	opcode	S	Rn	Rd	shift_imm	1 0 0	Rm

This data-processing operand is used to provide the signed value of a register arithmetically shifted right (divided by a constant power of two).

This instruction operand is the value of register Rm, arithmetically shifted right by an immediate value in the range 1 to 32. The sign bit of Rm (Rm[31]) is inserted into the vacated bit positions. The carry-out from the shifter is the last bit shifted out.

Syntax

```
<Rm>, ASR #<shift_imm>
```

where:

`<Rm>`	Specifies the register whose value is to be shifted.
`ASR`	Indicates an arithmetic shift right.
`<shift_imm>`	Specifies the shift. This is an immediate value between 1 and 32. (A shift by 32 is encoded by shift_imm == 0.)

Architecture version

All

Operation

```
if shift_imm == 0 then
    if Rm[31] == 0 then
        shifter_operand = 0
        shifter_carry_out = Rm[31]
    else /* Rm[31] == 1 */
        shifter_operand = 0xFFFFFFFF
        shifter_carry_out = Rm[31]
else /* shift_imm > 0 */
    shifter_operand = Rm Arithmetic_Shift_Right <shift_imm>
    shifter_carry_out = Rm[shift_imm - 1]
```

Notes

Use of R15	If R15 is specified as register Rm or Rn, the value used is the address of the current instruction plus 8.

5.1.10 Data-processing operands - Arithmetic shift right by register

31 28	27 26 25	24 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0	opcode	S	Rn	Rd	Rs	0 1 0 1	Rm

This data-processing operand is used to provide the signed value of a register arithmetically shifted right (divided by a variable power of two).

This instruction operand is the value of register Rm arithmetically shifted right by the value in the least significant byte of register Rs. The sign bit of Rm (Rm[31]) is inserted into the vacated bit positions. The carry-out from the shifter is the last bit shifted out, which is the sign bit of Rm if the shift amount is more than 32, or the C flag if the shift amount is zero.

Syntax

`<Rm>, ASR <Rs>`

where:

`<Rm>`	Specifies the register whose value is to be shifted.
`ASR`	Indicates an arithmetic shift right.
`<Rs>`	Is the register containing the value of the shift.

Architecture version

All

Operation

```
if Rs[7:0] == 0 then
    shifter_operand = Rm
    shifter_carry_out = C Flag
else if Rs[7:0] < 32 then
    shifter_operand = Rm Arithmetic_Shift_Right Rs[7:0]
    shifter_carry_out = Rm[Rs[7:0] - 1]
else /* Rs[7:0] >= 32 */
    if Rm[31] == 0 then
        shifter_operand = 0
        shifter_carry_out = Rm[31]
    else /* Rm[31] == 1 */
        shifter_operand = 0xFFFFFFFF
        shifter_carry_out = Rm[31]
```

Notes

Use of R15 Specifying R15 as register Rd, register Rm, register Rn, or register Rs has UNPREDICTABLE results.

5.1.11 Data-processing operands - Rotate right by immediate

31 28	27	26	25	24 21	20	19 16	15 12	11 7	6	5	4	3 0
cond	0	0	0	opcode	S	Rn	Rd	shift_imm	1	1	0	Rm

This data-processing operand is used to provide the value of a register rotated by a constant value.

This instruction operand is the value of register Rm rotated right by an immediate value in the range 1 to 31. As bits are rotated off the right end, they are inserted into the vacated bit positions on the left. The carry-out from the shifter is the last bit rotated off the right end.

Syntax

```
<Rm>, ROR #<shift_imm>
```

where:

`<Rm>`	Specifies the register whose value is to be rotated.
`ROR`	Indicates a rotate right.
`<shift_imm>`	Specifies the rotation. This is an immediate value between 1 and 31. When shift_imm == 0, an `RRX` operation (rotate right with extend) is performed. This is described in *Data-processing operands - Rotate right with extend* on page A5-17.

Architecture version

All

Operation

```
if shift_imm == 0 then
    See "Data-processing operands - Rotate right with extend" on page A5-17
else /* shift_imm > 0 */
    shifter_operand = Rm Rotate_Right shift_imm
    shifter_carry_out = Rm[shift_imm - 1]
```

Notes

Use of R15	If R15 is specified as register Rm or Rn, the value used is the address of the current instruction plus 8.

5.1.12 Data-processing operands - Rotate right by register

31 28	27	26	25	24 21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	opcode	S	Rn	Rd	Rs	0	1	1	1	Rm

This data-processing operand is used to provide the value of a register rotated by a variable value.

This instruction operand is produced by the value of register Rm rotated right by the value in the least significant byte of register Rs. As bits are rotated off the right end, they are inserted into the vacated bit positions on the left. The carry-out from the shifter is the last bit rotated off the right end, or the C flag if the shift amount is zero.

Syntax

`<Rm>, ROR <Rs>`

where:

`<Rm>`	Specifies the register whose value is to be rotated.
`ROR`	Indicates a rotate right.
`<Rs>`	Is the register containing the value of the rotation.

Architecture version

All

Operation

```
if Rs[7:0] == 0 then
    shifter_operand = Rm
    shifter_carry_out = C Flag
else if Rs[4:0] == 0 then
    shifter_operand = Rm
    shifter_carry_out = Rm[31]
else /* Rs[4:0] > 0 */
    shifter_operand = Rm Rotate_Right Rs[4:0]
    shifter_carry_out = Rm[Rs[4:0] - 1]
```

Notes

Use of R15	Specifying R15 as register Rd, register Rm, register Rn, or register Rs has UNPREDICTABLE results.

5.1.13 Data-processing operands - Rotate right with extend

31 28	27 26 25	24 21	20	19 16	15 12	11 10 9 8 7 6 5 4	3 0
cond	0 0 0	opcode	S	Rn	Rd	0 0 0 0 0 1 1 0	Rm

This data-processing operand can be used to perform a 33-bit rotate right using the Carry Flag as the 33rd bit.

This instruction operand is the value of register Rm shifted right by one bit, with the Carry Flag replacing the vacated bit position. The carry-out from the shifter is the bit shifted off the right end.

Syntax

`<Rm>, RRX`

where:

`<Rm>`	Specifies the register whose value is shifted right by one bit.
`RRX`	Indicates a rotate right with extend.

Architecture version

All

Operation

```
shifter_operand = (C Flag Logical_Shift_Left 31) OR (Rm Logical_Shift_Right 1)
shifter_carry_out = Rm[0]
```

Notes

Encoding	The instruction encoding is in the space that would be used for `ROR #0`.
Use of R15	If R15 is specified as register Rm or Rn, the value used is the address of the current instruction plus 8.
ADC instruction	A rotate left with extend can be performed with an `ADC` instruction.

5.2 Addressing Mode 2 - Load and Store Word or Unsigned Byte

There are nine addressing modes used to calculate the address for a Load and Store Word or Unsigned Byte instruction. The general instruction syntax is:

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

where `<addressing_mode>` is one of the nine options listed below.

All nine of the following options are available for `LDR`, `LDRB`, `STR` and `STRB`. For `LDRBT`, `LDRT`, `STRBT` and `STRBT`, only the *post-indexed* options (the last three in the list) are available. For the `PLD` instruction described in *PLD* on page A10-14, only the *offset* options (the first three in the list) are available.

1. `[<Rn>, #+/-<offset_12>]`

   See *Load and Store Word or Unsigned Byte - Immediate offset* on page A5-20.

2. `[<Rn>, +/-<Rm>]`

   See *Load and Store Word or Unsigned Byte - Register offset* on page A5-21.

3. `[<Rn>, +/-<Rm>, <shift> #<shift_imm>]`

   See *Load and Store Word or Unsigned Byte - Scaled register offset* on page A5-22.

4. `[<Rn>, #+/-<offset_12>]!`

   See *Load and Store Word or Unsigned Byte - Immediate pre-indexed* on page A5-24.

5. `[<Rn>, +/-<Rm>]!`

   See *Load and Store Word or Unsigned Byte - Register pre-indexed* on page A5-25.

6. `[<Rn>, +/-<Rm>, <shift> #<shift_imm>]!`

   See *Load and Store Word or Unsigned Byte - Scaled register pre-indexed* on page A5-26.

7. `[<Rn>], #+/-<offset_12>`

   See *Load and Store Word or Unsigned Byte - Immediate post-indexed* on page A5-28.

8. `[<Rn>], +/-<Rm>`

   See *Load and Store Word or Unsigned Byte - Register post-indexed* on page A5-30.

9. `[<Rn>], +/-<Rm>, <shift> #<shift_imm>`

   See *Load and Store Word or Unsigned Byte - Scaled register post-indexed* on page A5-32.

5.2.1 Encoding

The following three diagrams show the encodings for this addressing mode:

Immediate offset/index

31 28	27 26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0 1	0	P	U	B	W	L	Rn	Rd	offset_12

Register offset/index

31 28	27 26	25	24	23	22	21	20	19 16	15 12	11 10 9 8 7 6 5 4	3 0
cond	0 1	1	P	U	B	W	L	Rn	Rd	0 0 0 0 0 0 0 0	Rm

Scaled register offset/index

31 28	27 26	25	24	23	22	21	20	19 16	15 12	11 7	6 5	4	3 0
cond	0 1	1	P	U	B	W	L	Rn	Rd	shift_imm	shift	0	Rm

The P bit Has two meanings:

- **P == 0** Indicates the use of *post-indexed addressing*. The base register value is used for the memory address, and the offset is then applied to the base register value and written back to the base register.
- **P == 1** Indicates the use of *offset addressing* or *pre-indexed addressing* (the W bit determines which). The memory address is generated by applying the offset to the base register value.

The U bit Indicates whether the offset is added to the base (U == 1) or is subtracted from the base (U == 0).

The B bit Distinguishes between an unsigned byte (B == 1) and a word (B == 0) access.

The W bit Has two meanings:

- **P == 0** If W == 0, the instruction is `LDR`, `LDRB`, `STR` or `STRB` and a normal memory access is performed. If W == 1, the instruction is `LDRBT`, `LDRT`, `STRBT` or `STRT` and an unprivileged (User mode) memory access is performed.
- **P == 1** If W == 0, the base register is not updated (offset addressing). If W == 1, the calculated memory address is written back to the base register (pre-indexed addressing).

The L bit Distinguishes between a Load (L == 1) and a Store (L == 0).

5.2.2 Load and Store Word or Unsigned Byte - Immediate offset

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	0	1	U	B	0	L	Rn	Rd	offset_12

This addressing mode calculates an address by adding or subtracting the value of an immediate offset to or from the value of the base register Rn.

Syntax

```
[<Rn>, #+/-<offset_12>]
```

where:

`<Rn>`	Specifies the register containing the base address.
`<offset_12>`	Specifies the immediate offset used with the value of Rn to form the address.

Architecture version

All

Operation

```
if U == 1 then
    address = Rn + offset_12
else /* U == 0 */
    address = Rn - offset_12
```

Usage

This addressing mode is useful for accessing structure (record) fields, and accessing parameters and local variables in a stack frame. With an offset of zero, the address produced is the unaltered value of the base register Rn.

Notes

Offset of zero	The syntax `[<Rn>]` is treated as an abbreviation for `[<Rn>, #0]`, unless the instruction is one that only allows post-indexed addressing modes (`LDRBT`, `LDRT`, `STRBT` or `STRT`).
The B bit	This bit distinguishes between an unsigned byte (B==1) and a word (B==0) access.
The L bit	This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.
Use of R15	If R15 is specified as register Rn, the value used is the address of the instruction plus 8.

5.2.3 Load and Store Word or Unsigned Byte - Register offset

31 28	27 26	25	24	23	22	21	20	19 16	15 12	11 10 9 8 7 6 5 4	3 0
cond	0 1	1	1	U	B	0	L	Rn	Rd	0 0 0 0 0 0 0 0	Rm

This addressing mode calculates an address by adding or subtracting the value of the index register Rm to or from the value of the base register Rn.

Syntax

```
[<Rn>, +/-<Rm>]
```

where:

`<Rn>`	Specifies the register containing the base address.
`<Rm>`	Specifies the register containing the value to add to or subtract from Rn.

Architecture version

All

Operation

```
if U == 1 then
    address = Rn + Rm
else /* U == 0 */
    address = Rn - Rm
```

Usage

This addressing mode is used for pointer plus offset arithmetic, and accessing a single element of an array of bytes.

Notes

Encoding	This addressing mode is encoded as an `LSL` scaled register offset, scaled by zero.
The B bit	This bit distinguishes between an unsigned byte (B==1) and a word (B==0) access.
The L bit	This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.
Use of R15	If R15 is specified as register Rn, the value used is the address of the instruction plus 8. Specifying R15 as register Rm has UNPREDICTABLE results.

5.2.4 Load and Store Word or Unsigned Byte - Scaled register offset

31 28	27 26	25	24	23	22	21	20	19 16	15 12	11 7	6 5	4	3 0
cond	0 1	1	1	U	B	0	L	Rn	Rd	shift_imm	shift	0	Rm

These five addressing modes calculate an address by adding or subtracting the shifted or rotated value of the index register Rm to or from the value of the base register Rn.

Syntax

One of:

```
[<Rn>, +/-<Rm>, LSL #<shift_imm>]
[<Rn>, +/-<Rm>, LSR #<shift_imm>]
[<Rn>, +/-<Rm>, ASR #<shift_imm>]
[<Rn>, +/-<Rm>, ROR #<shift_imm>]
[<Rn>, +/-<Rm>, RRX]
```

where:

`<Rn>`	Specifies the register containing the base address.
`<Rm>`	Specifies the register containing the offset to add to or subtract from Rn.
`LSL`	Specifies a logical shift left.
`LSR`	Specifies a logical shift right.
`ASR`	Specifies an arithmetic shift right.
`ROR`	Specifies a rotate right.
`RRX`	Specifies a rotate right with extend.
`<shift_imm>`	Specifies the shift or rotation.

`LSL`	0 to 31, encoded directly in the shift_imm field.
`LSR`	1 to 32. A shift amount of 32 is encoded as shift_imm == 0. Other shift amounts are encoded directly.
`ASR`	1 to 32. A shift amount of 32 is encoded as shift_imm == 0. Other shift amounts are encoded directly.
`ROR`	1 to 31, encoded directly in the shift_imm field. (The shift_imm == 0 encoding is used to specify the `RRX` option.)

Architecture version

All

Operation

```
case shift of
    0b00 /* LSL */
        index = Rm Logical_Shift_Left shift_imm
    0b01 /* LSR */
        if shift_imm == 0 then /* LSR #32 */
            index = 0
        else
            index = Rm Logical_Shift_Right shift_imm
    0b10 /* ASR */
        if shift_imm == 0 then /* ASR #32 */
            if Rm[31] == 1 then
                index = 0xFFFFFFFF
            else
                index = 0
        else
            index = Rm Arithmetic_Shift_Right shift_imm
    0b11 /* ROR or RRX */
        if shift_imm == 0 then /* RRX */
            index = (C Flag Logical_Shift_Left 31) OR
                    (Rm Logical_Shift_Right 1)
        else /* ROR */
            index = Rm Rotate_Right shift_imm
endcase
if U == 1 then
    address = Rn + index
else /* U == 0 */
    address = Rn - index
```

Usage

These addressing modes are used for accessing a single element of an array of values larger than a byte.

Notes

The B bit	This bit distinguishes between an unsigned byte (B==1) and a word (B==0) access.
The L bit	This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.
Use of R15	If R15 is specified as register Rn, the value used is the address of the instruction plus 8. Specifying R15 as register Rm has UNPREDICTABLE results.

5.2.5 Load and Store Word or Unsigned Byte - Immediate pre-indexed

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	0	1	U	B	1	L	Rn	Rd	offset_12

This addressing mode calculates an address by adding or subtracting the value of an immediate offset to or from the value of the base register Rn.

If the condition specified in the instruction matches the condition code status, the calculated address is written back to the base register Rn. The conditions are defined in *The condition field* on page A3-5.

Syntax

```
[<Rn>, #+/-<offset_12>]!
```

where:

`<Rn>`	Specifies the register containing the base address.
`<offset_12>`	Specifies the immediate offset used with the value of Rn to form the address.
`!`	Sets the W bit, causing base register update.

Architecture version

All

Operation

```
if U == 1 then
    address = Rn + offset_12
else /* if U == 0 */
    address = Rn - offset_12
if ConditionPassed(cond) then
    Rn = address
```

Usage

This addressing mode is used for pointer access to arrays with automatic update of the pointer value.

Notes

Offset of zero	The syntax `[<Rn>]` must never be treated as an abbreviation for `[<Rn>, #0]!`.
The B bit	This bit distinguishes between an unsigned byte (B==1) and a word (B==0) access.
The L bit	This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.
Use of R15	Specifying R15 as register Rn has UNPREDICTABLE results.

5.2.6 Load and Store Word or Unsigned Byte - Register pre-indexed

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
cond	0	1	1	1	U	B	1	L	Rn	Rd	0	0	0	0	0	0	0	0	Rm

This addressing mode calculates an address by adding or subtracting the value of an index register Rm to or from the value of the base register Rn.

If the condition specified in the instruction matches the condition code status, the calculated address is written back to the base register Rn. The conditions are defined in *The condition field* on page A3-5.

Syntax

```
[<Rn>, +/-<Rm>]!
```

where:

`<Rn>`	Specifies the register containing the base address.
`<Rm>`	Specifies the register containing the offset to add to or subtract from Rn.
`!`	Sets the W bit, causing base register update.

Architecture version

All

Operation

```
if U == 1 then
    address = Rn + Rm
else /* U == 0 */
    address = Rn - Rm
if ConditionPassed(cond) then
    Rn = address
```

Notes

Encoding	This addressing mode is encoded as an LSL scaled register offset, scaled by zero.
The B bit	This bit distinguishes between an unsigned byte (B==1) and a word (B==0) access.
The L bit	This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.
Use of R15	Specifying R15 as register Rm or Rn has UNPREDICTABLE results.
Operand restriction	If the same register is specified for Rn and Rm, the result is UNPREDICTABLE.

5.2.7 Load and Store Word or Unsigned Byte - Scaled register pre-indexed

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 7	6 5	4	3 0
cond	0	1	1	1	U	B	1	L	Rn	Rd	shift_imm	shift	0	Rm

These five addressing modes calculate an address by adding or subtracting the shifted or rotated value of the index register Rm to or from the value of the base register Rn.

If the condition specified in the instruction matches the condition code status, the calculated address is written back to the base register Rn. The conditions are defined in *The condition field* on page A3-5.

Syntax

One of:

```
[<Rn>, +/-<Rm>, LSL #<shift_imm>]!
[<Rn>, +/-<Rm>, LSR #<shift_imm>]!
[<Rn>, +/-<Rm>, ASR #<shift_imm>]!
[<Rn>, +/-<Rm>, ROR #<shift_imm>]!
[<Rn>, +/-<Rm>, RRX]!
```

where:

`<Rn>`	Specifies the register containing the base address.
`<Rm>`	Specifies the register containing the offset to add to or subtract from Rn.
`LSL`	Specifies a logical shift left.
`LSR`	Specifies a logical shift right.
`ASR`	Specifies an arithmetic shift right.
`ROR`	Specifies a rotate right.
`RRX`	Specifies a rotate right with extend.
`<shift_imm>`	Specifies the shift or rotation.

`LSL`	0 to 31, encoded directly in the shift_imm field.
`LSR`	1 to 32. A shift amount of 32 is encoded as shift_imm == 0. Other shift amounts are encoded directly.
`ASR`	1 to 32. A shift amount of 32 is encoded as shift_imm == 0. Other shift amounts are encoded directly.
`ROR`	1 to 31, encoded directly in the shift_imm field. (The shift_imm == 0 encoding is used to specify the `RRX` option.)

!	Sets the W bit, causing base register update.

Architecture version

All

Operation

```
case shift of
    0b00 /* LSL */
        index = Rm Logical_Shift_Left shift_imm
    0b01 /* LSR */
        if shift_imm == 0 then /* LSR #32 */
            index = 0
        else
            index = Rm Logical_Shift_Right shift_imm
    0b10 /* ASR */
        if shift_imm == 0 then /* ASR #32 */
            if Rm[31] == 1 then
                index = 0xFFFFFFFF
            else
                index = 0
        else
            index = Rm Arithmetic_Shift_Right shift_imm
    0b11 /* ROR or RRX */
        if shift_imm == 0 then /* RRX */
            index = (C Flag Logical_Shift_Left 31) OR
                    (Rm Logical_Shift_Right 1)
        else /* ROR */
            index = Rm Rotate_Right shift_imm
endcase
if U == 1 then
    address = Rn + index
else /* U == 0 */
    address = Rn - index
if ConditionPassed(cond) then
    Rn = address
```

Notes

The B bit	This bit distinguishes between an unsigned byte (B==1) and a word (B==0) access.
The L bit	This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.
Use of R15	Specifying R15 as register Rm or Rn has UNPREDICTABLE results.
Operand restriction	If the same register is specified for Rn and Rm, the result is UNPREDICTABLE.

5.2.8 Load and Store Word or Unsigned Byte - Immediate post-indexed

31 28	27 26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0 1	0	0	U	B	0	L	Rn	Rd	offset_12

This addressing mode uses the value of the base register Rn as the address for the memory access.

If the condition specified in the instruction matches the condition code status, the value of the immediate offset is added to or subtracted from the value of the base register Rn and written back to the base register Rn. The conditions are defined in *The condition field* on page A3-5.

Syntax

`[<Rn>], #+/-<offset_12>`

where:

`<Rn>`	Specifies the register containing the base address.
`<offset_12>`	Specifies the immediate offset used with the value of Rn to form the address.

Architecture version

All

Operation

```
address = Rn
if ConditionPassed(cond) then
    if U == 1 then
        Rn = Rn + offset_12
    else /* U == 0 */
        Rn = Rn - offset_12
```

Usage

This addressing mode is used for pointer access to arrays with automatic update of the pointer value.

Notes

Post-indexed addressing modes

`LDRBT`, `LDRT`, `STRBT`, and `STRT` only support post-indexed addressing modes. They use a minor modification of the above bit pattern, where bit[21] (the W bit) is 1, not 0 as shown.

Offset of zero The syntax `[<Rn>]` is treated as an abbreviation for `[<Rn>],#0` for instructions that only support post-indexed addressing modes (`LDRBT`, `LDRT`, `STRBT`, `STRT`), but not for other instructions.

The B bit This bit distinguishes between an unsigned byte (B==1) and a word (B==0) access.

The L bit This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

Use of R15 Specifying R15 as register Rn has UNPREDICTABLE results.

5.2.9 Load and Store Word or Unsigned Byte - Register post-indexed

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
cond	0	1	1	0	U	B	0	L	Rn	Rd	0	0	0	0	0	0	0	0	Rm

This addressing mode uses the value of the base register Rn as the address for the memory access.

If the condition specified in the instruction matches the condition code status, the value of the index register Rm is added to or subtracted from the value of the base register Rn and written back to the base register Rn. The conditions are defined in *The condition field* on page A3-5.

Syntax

```
[<Rn>], +/-<Rm>
```

where:

`<Rn>`	Specifies the register containing the base address.
`<Rm>`	Specifies the register containing the offset to add to or subtract from Rn.

Architecture version

All

Operation

```
address = Rn
if ConditionPassed(cond) then
    if U == 1 then
        Rn = Rn + Rm
    else /* U == 0 */
        Rn = Rn - Rm
```

Notes

Encoding This addressing mode is encoded as an `LSL` scaled register offset, scaled by zero.

Post-indexed addressing modes

`LDRBT`, `LDRT`, `STRBT`, and `STRT` only support post-indexed addressing modes. They use a minor modification of the above bit pattern, where bit[21] (the W bit) is 1, not 0 as shown.

The B bit This bit distinguishes between an unsigned byte (B==1) and a word (B==0) access.

The L bit This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

Use of R15 Specifying R15 as register Rn or Rm has UNPREDICTABLE results.

Operand restriction If the same register is specified for Rn and Rm, the result is UNPREDICTABLE.

5.2.10 Load and Store Word or Unsigned Byte - Scaled register post-indexed

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–7	6–5	4	3–0
cond	0	1	1	0	U	B	0	L	Rn	Rd	shift_imm	shift	0	Rm

This addressing mode uses the value of the base register Rn as the address for the memory access.

If the condition specified in the instruction matches the condition code status, the shifted or rotated value of index register Rm is added to or subtracted from the value of the base register Rn and written back to the base register Rn. The conditions are defined in *The condition field* on page A3-5.

Syntax

One of:

```
[<Rn>], +/-<Rm>, LSL #<shift_imm>
[<Rn>], +/-<Rm>, LSR #<shift_imm>
[<Rn>], +/-<Rm>, ASR #<shift_imm>
[<Rn>], +/-<Rm>, ROR #<shift_imm>
[<Rn>], +/-<Rm>, RRX
```

where:

`<Rn>`	Specifies the register containing the base address.
`<Rm>`	Specifies the register containing the offset to add to or subtract from Rn.
`LSL`	Specifies a logical shift left.
`LSR`	Specifies a logical shift right.
`ASR`	Specifies an arithmetic shift right.
`ROR`	Specifies a rotate right.
`RRX`	Specifies a rotate right with extend.
`<shift_imm>`	Specifies the shift or rotation.

`LSL`	0 to 31, encoded directly in the shift_imm field.
`LSR`	1 to 32. A shift amount of 32 is encoded as shift_imm == 0. Other shift amounts are encoded directly.
`ASR`	1 to 32. A shift amount of 32 is encoded as shift_imm == 0. Other shift amounts are encoded directly.
`ROR`	1 to 31, encoded directly in the shift_imm field. (The shift_imm == 0 encoding is used to specify the `RRX` option.)

Architecture version

All

Operation

```
address = Rn
case shift of
    0b00 /* LSL */
        index = Rm Logical_Shift_Left shift_imm
    0b01 /* LSR */
        if shift_imm == 0 then /* LSR #32 */
            index = 0
        else
            index = Rm Logical_Shift_Right shift_imm
    0b10 /* ASR */
        if shift_imm == 0 then /* ASR #32 */
            if Rm[31] == 1 then
                index = 0xFFFFFFFF
            else
                index = 0
        else
            index = Rm Arithmetic_Shift_Right shift_imm
    0b11 /* ROR or RRX */
        if shift_imm == 0 then /* RRX */
            index = (C Flag Logical_Shift_Left 31) OR
                    (Rm Logical_Shift_Right 1)
        else /* ROR */
            index = Rm Rotate_Right shift_imm
endcase
if ConditionPassed(cond) then
    if U == 1 then
        Rn = Rn + index
    else /* U == 0 */
        Rn = Rn - index
```

Notes

Post-indexed addressing modes

`LDRBT`, `LDRT`, `STRBT`, and `STRT` only support post-indexed addressing modes. They use a minor modification of the above bit pattern, where bit[21] (the W bit) is 1, not 0 as shown.

The B bit This bit distinguishes between an unsigned byte (B == 1) and a word (B == 0) access.

The L bit This bit distinguishes between a Load (L == 1) and a Store (L == 0) instruction.

Use of R15 Specifying R15 as register Rm or Rn has UNPREDICTABLE results.

Operand restriction If the same register is specified for Rn and Rm, the result is UNPREDICTABLE.

5.3 Addressing Mode 3 - Miscellaneous Loads and Stores

There are six addressing modes used to calculate the address for load and store (signed or unsigned) halfword, load signed byte, or load and store doubleword instructions. The general instruction syntax is:

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

where `<addressing_mode>` is one of the following six options:

1. `[<Rn>, #+/-<offset_8>]`

   See *Miscellaneous Loads and Stores - Immediate offset* on page A5-36.

2. `[<Rn>, +/-<Rm>]`

   See *Miscellaneous Loads and Stores - Register offset* on page A5-38.

3. `[<Rn>, #+/-<offset_8>]!`

   See *Miscellaneous Loads and Stores - Immediate pre-indexed* on page A5-40.

4. `[<Rn>, +/-<Rm>]!`

   See *Miscellaneous Loads and Stores - Register pre-indexed* on page A5-42.

5. `[<Rn>], #+/-<offset_8>`

   See *Miscellaneous Loads and Stores - Immediate post-indexed* on page A5-44.

6. `[<Rn>], +/-<Rm>`

   See *Miscellaneous Loads and Stores - Register post-indexed* on page A5-46.

5.3.1 Encoding

The following diagrams show the encodings for this addressing mode:

Immediate offset/index

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	P	U	1	W	L	Rn	Rd	immedH	1	S	H	1	ImmedL

Register offset/index

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	P	U	0	W	L	Rn	Rd	SBZ	1	S	H	1	Rm

The P bit Has two meanings:

P == 0 Indicates the use of *post-indexed addressing*. The base register value is used for the memory address, and the offset is then applied to the base register value and written back to the base register.

P == 1 Indicates the use of *offset addressing* or *pre-indexed addressing* (the W bit determines which). The memory address is generated by applying the offset to the base register value.

The U bit Indicates whether the offset is added to the base (U == 1) or subtracted from the base (U == 0).

The W bit Has two meanings:

P == 0 The W bit must be 0 or the instruction is UNPREDICTABLE.

P == 1 W == 1 indicates that the memory address is written back to the base register (pre-indexed addressing), and W == 0 that the base register is unchanged (offset addressing).

The L bit This bit distinguishes between a Load (L == 1) and a Store (L == 0) instruction.

The S bit This bit distinguishes between a signed (S == 1) and an unsigned (S == 0) halfword access.

The H bit This bit distinguishes between a halfword (H == 1) and a byte (H == 0) access.

Unsigned bytes

If S == 0 and H == 0, apparently indicating an unsigned byte, the instruction is not one that uses this addressing mode. Instead, it is a multiply instruction, a `SWP` or `SWPB` instruction, or an unallocated instruction in the arithmetic or load/store instruction extension space (see *Extending the instruction set* on page A3-27).

Unsigned bytes are accessed by the `LDRB`, `LDRBT`, `STRB` and `STRBT` instructions, which use addressing mode 2 rather than addressing mode 3.

Signed stores If S == 1 and L == 0, apparently indicating a signed store instruction, the instruction is an unallocated instruction in the load/store extension space (see *Extending the instruction set* on page A3-27).

Signed bytes and halfwords can be stored with the same `STRB` and `STRH` instructions as are used for unsigned quantities, so no separate signed store instructions are provided.

5.3.2 Miscellaneous Loads and Stores - Immediate offset

31 28	27 26 25 24	23	22 21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0 0 0 1	U	1 0	L	Rn	Rd	immedH	1	S	H	1	immedL

This addressing mode calculates an address by adding or subtracting the value of an immediate offset to or from the value of the base register Rn.

Syntax

```
[<Rn>, #+/-<offset_8>]
```

where:

`<Rn>`	Specifies the register containing the base address.
`<offset_8>`	Specifies the immediate offset used with the value of Rn to form the address. The offset is encoded in immedH (top 4 bits) and immedL (bottom 4 bits).

Architecture version

Version 4 and above

Operation

```
offset_8 = (immedH << 4) OR immedL
if U == 1 then
    address = Rn + offset_8
else /* U == 0 */
    address = Rn - offset_8
```

Usage

This addressing mode is used for accessing structure (record) fields, and accessing parameters and locals variable in a stack frame. With an offset of zero, the address produced is the unaltered value of the base register Rn.

Notes

Zero offset	The syntax `[<Rn>]` is treated as an abbreviation for `[<Rn>,#0]`.
The L bit	This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.
The S bit	This bit distinguishes between a signed (S==1) and an unsigned (S==0) halfword access.
The H bit	This bit distinguishes between a halfword (H==1) and a signed byte (H==0) access.
Use of R15	If R15 is specified as register Rn, the value used is the address of the instruction plus 8.

Unsigned bytes

If S == 0 and H == 0, apparently indicating an unsigned byte, the instruction is not one that uses this addressing mode. Instead, it is a multiply instruction, a `SWP` or `SWPB` instruction, or an unallocated instruction in the arithmetic or load/store instruction extension space (see *Extending the instruction set* on page A3-27).

Unsigned bytes are accessed by the `LDRB`, `LDRBT`, `STRB` and `STRBT` instructions, which use addressing mode 2 rather than addressing mode 3.

Signed stores If S == 1 and L == 0, apparently indicating a signed store instruction, the instruction is an unallocated instruction in the load/store extension space (see *Extending the instruction set* on page A3-27).

Signed bytes and halfwords can be stored with the same `STRB` and `STRH` instructions as are used for unsigned quantities, so no separate signed store instructions are provided.

5.3.3 Miscellaneous Loads and Stores - Register offset

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	1	U	0	0	L	Rn	Rd	SBZ	1	S	H	1	Rm

This addressing mode calculates an address by adding or subtracting the value of the index register Rm to or from the value of the base register Rn.

Syntax

```
[<Rn>, +/-<Rm>]
```

where:

`<Rn>`	Specifies the register containing the base address.
`<Rm>`	Specifies the register containing the offset to add to or subtract from Rn.

Architecture version

Version 4 and above

Operation

```
if U == 1 then
    address = Rn + Rm
else /* U == 0 */
    address = Rn - Rm
```

Usage

This addressing mode is useful for pointer plus offset arithmetic and for accessing a single element of an array.

Notes

The L bit	Distinguishes between a Load (L==1) and a Store (L==0) instruction.
The S bit	Distinguishes between a signed (S==1) and an unsigned (S==0) halfword access.
The H bit	This bit distinguishes between a halfword (H==1) and a signed byte (H==0) access.
Unsigned bytes	If S == 0 and H == 0, apparently indicating an unsigned byte, the instruction is not one that uses this addressing mode. Instead, it is a multiply instruction, a `SWP` or `SWPB` instruction, or an unallocated instruction in the arithmetic or load/store instruction extension space (see *Extending the instruction set* on page A3-27). Unsigned bytes are accessed by the `LDRB`, `LDRBT`, `STRB` and `STRBT` instructions, which use addressing mode 2 rather than addressing mode 3.

Signed stores If S == 1 and L == 0, apparently indicating a signed store instruction, the instruction is an unallocated instruction in the load/store extension space (see *Extending the instruction set* on page A3-27).

Signed bytes and halfwords can be stored with the same `STRB` and `STRH` instructions as are used for unsigned quantities, so no separate signed store instructions are provided.

Use of R15 If R15 is specified as register Rn, the value used is the address of the instruction plus 8. Specifying R15 as register Rm has UNPREDICTABLE results.

5.3.4 Miscellaneous Loads and Stores - Immediate pre-indexed

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	1	U	1	1	L	Rn	Rd	immedH	1	S	H	1	ImmedL

This addressing mode calculates an address by adding or subtracting the value of an immediate offset to or from the value of the base register `Rn`.

If the condition specified in the instruction matches the condition code status, the calculated address is written back to the base register Rn. The conditions are defined in *The condition field* on page A3-5.

Syntax

```
[<Rn>, #+/-<offset_8>]!
```

where:

`<Rn>`	Specifies the register containing the base address.
`<offset_8>`	Specifies the immediate offset used with the value of Rn to form the address. The offset is encoded in immedH (top 4 bits) and immedL (bottom 4 bits).
`!`	Sets the W bit, causing base register update.

Architecture version

Version 4 and above

Operation

```
offset_8 = (immedH << 4) OR immedL
if U == 1 then
    address = Rn + offset_8
else /* U == 0 */
    address = Rn - offset_8
if ConditionPassed(cond) then
    Rn = address
```

Usage

This addressing mode gives pointer access to arrays, with automatic update of the pointer value.

Notes

Offset of zero The syntax `[<Rn>]` must not be treated as an abbreviation for `[<Rn>,#0]!`.

The L bit This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

The S bit This bit distinguishes between a signed (S==1) and an unsigned (S==0) halfword access.

The H bit This bit distinguishes between a halfword (H==1) and a signed byte (H==0) access.

Unsigned bytes If S == 0 and H == 0, apparently indicating an unsigned byte, the instruction is not one that uses this addressing mode. Instead, it is a multiply instruction, a `SWP` or `SWPB` instruction, or an unallocated instruction in the arithmetic or load/store instruction extension space (see *Extending the instruction set* on page A3-27).

Unsigned bytes are accessed by the `LDRB`, `LDRBT`, `STRB` and `STRBT` instructions, which use addressing mode 2 rather than addressing mode 3.

Signed stores If S == 1 and L == 0, apparently indicating a signed store instruction, the instruction is an unallocated instruction in the load/store extension space (see *Extending the instruction set* on page A3-27).

Signed bytes and halfwords can be stored with the same `STRB` and `STRH` instructions as are used for unsigned quantities, so no separate signed store instructions are provided.

Use of R15 Specifying R15 as register Rn has UNPREDICTABLE results.

5.3.5 Miscellaneous Loads and Stores - Register pre-indexed

31 28	27 26 25 24	23	22 21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0 0 0 1	U	0 1	L	Rn	Rd	SBZ	1	S	H	1	Rm

This addressing mode calculates an address by adding or subtracting the value of the index register Rm to or from the value of the base register Rn.

If the condition specified in the instruction matches the condition code status, the calculated address is written back to the base register Rn. The conditions are defined in *The condition field* on page A3-5.

Syntax

```
[<Rn>, +/-<Rm>]!
```

where:

`<Rn>`	Specifies the register containing the base address.
`<Rm>`	Specifies the register containing the offset to add to or subtract from Rn.
`!`	Sets the W bit, causing base register update.

Architecture version

Version 4 and above

Operation

```
if U == 1 then
    address = Rn + Rm
else /* U == 0 */
    address = Rn - Rm
if ConditionPassed(cond) then
    Rn = address
```

Notes

The L bit	This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.
The S bit	This bit distinguishes between a signed (S==1) and an unsigned (S==0) halfword access.
The H bit	This bit distinguishes between a halfword (H==1) and a signed byte (H==0) access.
Unsigned bytes	If S == 0 and H == 0, apparently indicating an unsigned byte, the instruction is not one that uses this addressing mode. Instead, it is a multiply instruction, a `SWP` or `SWPB` instruction, or an unallocated instruction in the arithmetic or load/store instruction extension space (see *Extending the instruction set* on page A3-27). Unsigned bytes are accessed by the `LDRB`, `LDRBT`, `STRB` and `STRBT` instructions, which use addressing mode 2 rather than addressing mode 3.
Signed stores	If S == 1 and L == 0, apparently indicating a signed store instruction, the instruction is an unallocated instruction in the load/store extension space (see *Extending the instruction set* on page A3-27). Signed bytes and halfwords can be stored with the same `STRB` and `STRH` instructions as are used for unsigned quantities, so no separate signed store instructions are provided.
Use of R15	Specifying R15 as register Rm or Rn has UNPREDICTABLE results.
Operand restriction	If the same register is specified for Rn and Rm, the result is UNPREDICTABLE.

5.3.6 Miscellaneous Loads and Stores - Immediate post-indexed

31 28	27 26 25 24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0 0 0 0	U	1	0	L	Rn	Rd	immedH	1	S	H	1	ImmedL

This addressing mode uses the value of the base register Rn as the address for the memory access.

If the condition specified in the instruction matches the condition code status, the value of the immediate offset is added to or subtracted from the value of the base register Rn and written back to the base register Rn. The conditions are defined in *The condition field* on page A3-5.

Syntax

```
[<Rn>], #+/-<offset_8>
```

where:

`<Rn>`	Specifies the register containing the base address.
`<offset_8>`	Specifies the immediate offset used with the value of Rn to form the address. The offset is encoded in immedH (top 4 bits) and immedL (bottom 4 bits).

Architecture version

Version 4 and above

Operation

```
address = Rn
offset_8 = (immedH << 4) OR immedL
if ConditionPassed(cond) then
    if U == 1 then
        Rn = Rn + offset_8
    else /* U == 0 */
        Rn = Rn - offset_8
```

Usage

This addressing mode gives pointer access to arrays, with automatic update of the pointer value.

Notes

Offset of zero	The syntax `[<Rn>]` must not be treated as an abbreviation for `[<Rn>],#0`.
The L bit	This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.
The S bit	This bit distinguishes between a signed (S==1) and an unsigned (S==0) halfword access.
The H bit	This bit distinguishes between a halfword (H==1) and a signed byte (H==0) access.
Unsigned bytes	If S == 0 and H == 0, apparently indicating an unsigned byte, the instruction is not one that uses this addressing mode. Instead, it is a multiply instruction, a `SWP` or `SWPB` instruction, or an unallocated instruction in the arithmetic or load/store instruction extension space (see *Extending the instruction set* on page A3-27). Unsigned bytes are accessed by the `LDRB`, `LDRBT`, `STRB` and `STRBT` instructions, which use addressing mode 2 rather than addressing mode 3.
Signed stores	If S == 1 and L == 0, apparently indicating a signed store instruction, the instruction is an unallocated instruction in the load/store extension space (see *Extending the instruction set* on page A3-27). Signed bytes and halfwords can be stored with the same `STRB` and `STRH` instructions as are used for unsigned quantities, so no separate signed store instructions are provided.
Use of R15	Specifying R15 as register Rn has UNPREDICTABLE results.

5.3.7 Miscellaneous Loads and Stores - Register post-indexed

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	0	U	0	0	L	Rn	Rd	SBZ	1	S	H	1	Rm

This addressing mode uses the value of the base register Rn as the address for the memory access.

If the condition specified in the instruction matches the condition code status, the value of the index register Rm is added to or subtracted from the value of the base register Rn and written back to the base register Rn. The conditions are defined in *The condition field* on page A3-5.

Syntax

```
[<Rn>], +/-<Rm>
```

where:

`<Rn>`	Specifies the register containing the base address.
`<Rm>`	Specifies the register containing the offset to add to or subtract from Rn.

Architecture version

Version 4 and above

Operation

```
address = Rn
if ConditionPassed(cond) then
    if U == 1 then
        Rn = Rn + Rm
    else /* U == 0 */
        Rn = Rn - Rm
```

Notes

The L bit	This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.
The S bit	This bit distinguishes between a signed (S==1) and an unsigned (S==0) halfword access.
The H bit	This bit distinguishes between a halfword (H==1) and a signed byte (H==0) access.
Unsigned bytes	If S == 0 and H == 0, apparently indicating an unsigned byte, the instruction is not one that uses this addressing mode. Instead, it is a multiply instruction, a `SWP` or `SWPB` instruction, or an unallocated instruction in the arithmetic or load/store instruction extension space (see *Extending the instruction set* on page A3-27). Unsigned bytes are accessed by the `LDRB`, `LDRBT`, `STRB` and `STRBT` instructions, which use addressing mode 2 rather than addressing mode 3.
Signed stores	If S == 1 and L == 0, apparently indicating a signed store instruction, the instruction is an unallocated instruction in the load/store extension space (see *Extending the instruction set* on page A3-27). Signed bytes and halfwords can be stored with the same `STRB` and `STRH` instructions as are used for unsigned quantities, so no separate signed store instructions are provided.
Use of R15	Specifying R15 as register Rm or Rn has UNPREDICTABLE results.
Operand restriction	If the same register is specified for Rn and Rm, the result is UNPREDICTABLE.

5.4 Addressing Mode 4 - Load and Store Multiple

Load Multiple instructions load a subset (possibly all) of the general-purpose registers from memory. Store Multiple instructions store a subset (possibly all) of the general purpose registers to memory.

Load and Store Multiple addressing modes produce a sequential range of addresses. The lowest-numbered register is stored at the lowest memory address and the highest-numbered register at the highest memory address.

The general instruction syntax is:

```
LDM|STM{<cond>}<addressing_mode> <Rn>{!}, <registers>{^}
```

where `<addressing_mode>` is one of the following four addressing modes:

1. `IA` (Increment After)

   See *Load and Store Multiple - Increment after* on page A5-50.

2. `IB` (Increment Before)

   See *Load and Store Multiple - Increment before* on page A5-51.

3. `DA` (Decrement After)

   See *Load and Store Multiple - Decrement after* on page A5-52.

4. `DB` (Decrement Before)

   See *Load and Store Multiple - Decrement before* on page A5-53.

5.4.1 Encoding

The following diagram shows the encoding for this addressing mode:

31 28	27	26	25	24	23	22	21	20	19 16	15 0
cond	1	0	0	P	U	S	W	L	Rn	register list

The P bit	Has two meanings:	
	P==1	indicates that the word addressed by Rn is included in the range of memory locations accessed, lying at the top (U==0) or bottom (U==1) of that range.
	P==0	indicates that the word addressed by Rn is excluded from the range of memory locations accessed, and lies one word beyond the top of the range (U==0) or one word below the bottom of the range (U==1).
The U bit	Indicates that the transfer is made upwards (U==1) or downwards (U==0) from the base register.	

The S bit For `LDM`s that load the PC, the S bit indicates that the CPSR is loaded from the SPSR. For `LDM`s that do not load the PC and all `STM`s, the S bit indicates that when the processor is in a privileged mode, the User mode banked registers are transferred instead of the registers of the current mode.

The W bit Indicates that the base register is updated after the transfer. The base register is incremented (U==1) or decremented (U==0) by four times the number of registers in the register list.

The L bit Distinguishes between Load (L==1) and Store (L==0) instructions.

Register list The register_list field of the instruction has one bit for each general-purpose register: bit[0] for register zero through to bit[15] for register 15 (the PC). If no bits are set, the result is UNPREDICTABLE.

The instruction syntax specifies the registers to load or store in `<registers>`, which is a comma-separated list of registers, surrounded by { and }.

5.4.2 Load and Store Multiple - Increment after

31 28	27 26 25	24	23	22	21	20	19 16	15 0
cond	1 0 0	0	1	S	W	L	Rn	register list

This addressing mode is for Load and Store Multiple instructions, and forms a range of addresses.

The first address formed is the `<start_address>`, and is the value of the base register Rn. Subsequent addresses are formed by incrementing the previous address by four. One address is produced for each register that is specified in `<registers>`.

The last address produced is the `<end_address>`. Its value is four less than the sum of the value of the base register and four times the number of registers specified in `<registers>`.

If the condition specified in the instruction matches the condition code status and the W bit is set, Rn is incremented by four times the number of registers in `<registers>`. The conditions are defined in *The condition field* on page A3-5.

Syntax

```
IA
```

See also the alternative syntax described in *Load and Store Multiple addressing modes (alternative names)* on page A5-54.

Architecture version

All

Operation

```
start_address = Rn
end_address = Rn + (Number_Of_Set_Bits_In(register_list) * 4) - 4
if ConditionPassed(cond) and W == 1 then
    Rn = Rn + (Number_Of_Set_Bits_In(register_list) * 4)
```

Notes

The L bit This bit distinguishes between a Load Multiple and a Store Multiple.

The S bit For `LDM`s that load the PC, the S bit indicates that the CPSR is loaded from the SPSR. For `LDM`s that do not load the PC and all `STM`s, the S bit indicates that when the processor is in a privileged mode, the User mode banked registers are transferred instead of the registers of the current mode.

5.4.3 Load and Store Multiple - Increment before

31 28	27 26 25	24	23	22	21	20	19 16	15 0
cond	1 0 0	1	1	S	W	L	Rn	register list

This addressing mode is for Load and Store Multiple instructions, and forms a range of addresses.

The first address formed is the `<start_address>`, and is the value of the base register Rn plus four. Subsequent addresses are formed by incrementing the previous address by four. One address is produced for each register that is specified in `<registers>`.

The last address produced is the `<end_address>`. Its value is the sum of the value of the base register and four times the number of registers specified in `<registers>`.

If the condition specified in the instruction matches the condition code status and the W bit is set, Rn is incremented by four times the number of registers in `<registers>`. The conditions are defined in *The condition field* on page A3-5.

Syntax

```
IB
```

See also the alternative syntax described in *Load and Store Multiple addressing modes (alternative names)* on page A5-54.

Architecture version

All

Operation

```
start_address = Rn + 4
end_address = Rn + (Number_Of_Set_Bits_In(register_list) * 4)
if ConditionPassed(cond) and W == 1 then
    Rn = Rn + (Number_Of_Set_Bits_In(register_list) * 4)
```

Notes

The L bit This bit distinguishes between a Load Multiple and a Store Multiple.

The S bit For `LDM`s that load the PC, the S bit indicates that the CPSR is loaded from the SPSR. For `LDM`s that do not load the PC and all `STM`s, the S bit indicates that when the processor is in a privileged mode, the User mode banked registers are transferred instead of the registers of the current mode.

5.4.4 Load and Store Multiple - Decrement after

31 28	27 26 25	24	23	22	21	20	19 16	15 0
cond	1 0 0	0	0	S	W	L	Rn	register list

This addressing mode is for Load and Store Multiple instructions, and forms a range of addresses.

The first address formed is the `<start_address>`, and is the value of the base register minus four times the number of registers specified in `<registers>`, plus 4. Subsequent addresses are formed by incrementing the previous address by four. One address is produced for each register that is specified in `<registers>`.

The last address produced is the `<end_address>`. Its value is the value of the base register Rn.

If the condition specified in the instruction matches the condition code status and the W bit is set, Rn is decremented by four times the number of registers in `<registers>`. The conditions are defined in *The condition field* on page A3-5.

Syntax

```
DA
```

See also the alternative syntax described in *Load and Store Multiple addressing modes (alternative names)* on page A5-54.

Architecture version

All

Operation

```
start_address = Rn - (Number_Of_Set_Bits_In(register_list) * 4) + 4
end_address = Rn
if ConditionPassed(cond) and W == 1 then
    Rn = Rn - (Number_Of_Set_Bits_In(register_list) * 4)
```

Notes

The L bit This bit distinguishes between a Load Multiple and a Store Multiple.

The S bit For `LDM`s that load the PC, the S bit indicates that the CPSR is loaded from the SPSR. For `LDM`s that do not load the PC and all `STM`s, the S bit indicates that when the processor is in a privileged mode, the User mode banked registers are transferred instead of the registers of the current mode.

5.4.5 Load and Store Multiple - Decrement before

31 28	27 26 25	24	23	22	21	20	19 16	15 0
cond	1 0 0	1	0	S	W	L	Rn	register list

This addressing mode is for Load and Store multiple instructions, and forms a range of addresses.

The first address formed is the `<start_address>`, and is the value of the base register minus four times the number of registers specified in `<registers>`. Subsequent addresses are formed by incrementing the previous address by four. One address is produced for each register that is specified in `<registers>`.

The last address produced is the `<end_address>`. Its value is the value of the base register Rn minus four.

If the condition specified in the instruction matches the condition code status and the W bit is set, Rn is decremented by four times the number of registers in `<registers>`. The conditions are defined in *The condition field* on page A3-5.

Syntax

`DB`

See also the alternative syntax described in *Load and Store Multiple addressing modes (alternative names)* on page A5-54.

Architecture version

All

Operation

```
start_address = Rn - (Number_Of_Set_Bits_In(register_list) * 4)
end_address = Rn - 4
if ConditionPassed(cond) and W == 1 then
    Rn = Rn - (Number_Of_Set_Bits_In(register_list) * 4)
```

Notes

The L bit This bit distinguishes between a Load Multiple and a Store Multiple.

The S bit For `LDM`s that load the PC, the S bit indicates that the CPSR is loaded from the SPSR. For `LDM`s that do not load the PC and all `STM`s, the S bit indicates that when the processor is in a privileged mode, the User mode banked registers are transferred instead of the registers of the current mode.

5.4.6 Load and Store Multiple addressing modes (alternative names)

The four addressing mode names given in *Addressing Mode 4 - Load and Store Multiple* on page A5-48 (IA, IB, DA, DB) are most useful when a load and Store Multiple instruction is being used for block data transfer, as it is likely that the Load Multiple and Store Multiple have the same addressing mode, so that the data is stored in the same way that it was loaded.

However, if Load Multiple and Store Multiple are being used to access a stack, the data is not loaded with the same addressing mode that was used to store the data, because the load (pop) and store (push) operations must adjust the stack in opposite directions.

Stack operations

Load Multiple and Store Multiple addressing modes can be specified with an alternative syntax, which is more applicable to stack operations:

Full stacks	Have stack pointers that point to the last used (full) location.
Empty stacks	Have stack pointers that point to the first unused (empty) location.
Descending stacks	Grow towards decreasing memory addresses (towards the bottom of memory).
Ascending stacks	Grow towards increasing memory addresses (towards the top of memory).

Two attributes allow four types of stack to be defined:

- Full Descending, with the syntax `FD`
- Empty Descending, with the syntax `ED`
- Full Ascending, with the syntax `FA`
- Empty Ascending, with the syntax `EA`.

Note

When defining stacks on which coprocessor data is to be placed (or might be placed in the future), programmers are advised to use the `FD` or `EA` stack types. This is because coprocessor data can be pushed to these types of stack with a single `STC` instruction and popped from them with a single `LDC` instruction. Multi-instruction sequences are required for coprocessor access to `FA` or `ED` stacks.

Table 5-1 and Table 5-2 on page A5-55 show the relationship between the four types of stack, the four types of addressing mode shown above, and the L, U, and P bits in the instruction format.

Table 5-1 shows the relationship for `LDM` instructions.

Table 5-1 LDM addressing modes

Non-stack addressing mode	Stack addressing mode	L bit	P bit	U bit
`LDMDA` (Decrement After)	`LDMFA` (Full Ascending)	1	0	0
`LDMIA` (Increment After)	`LDMFD` (Full Descending)	1	0	1
`LDMDB` (Decrement Before)	`LDMEA` (Empty Ascending)	1	1	0
`LDMIB` (Increment Before)	`LDMED` (Empty Descending)	1	1	1

Table 5-2 shows the relationship for `STM` instructions.

Table 5-2 STM addressing modes

Non-stack addressing mode	Stack addressing mode	L bit	P bit	U bit
`STMDA` (Decrement After)	`STMED` (Empty Descending)	0	0	0
`STMIA` (Increment After)	`STMEA` (Empty Ascending)	0	0	1
`STMDB` (Decrement Before)	`STMFD` (Full Descending)	0	1	0
`STMIB` (Increment Before)	`STMFA` (Full Ascending)	0	1	1

5.5 Addressing Mode 5 - Load and Store Coprocessor

There are four addressing modes which are used to calculate the address of a Load or Store Coprocessor instruction. The general instruction syntax is:

```
<opcode>{<cond>}{L}  <coproc>,<CRd>,<addressing_mode>
```

where `<addressing_mode>` is one of the following four options:

1. `[<Rn>,#+/-<offset_8>*4]`

   See *Load and Store Coprocessor - Immediate offset* on page A5-58.

2. `[<Rn>,#+/-<offset_8>*4]!`

   See *Load and Store Coprocessor - Immediate pre-indexed* on page A5-60.

3. `[<Rn>],#+/-<offset_8>*4`

   See *Load and Store Coprocessor - Immediate post-indexed* on page A5-62.

4. `[<Rn>],<option>`

   See *Load and Store Coprocessor - Unindexed* on page A5-64.

5.5.1 Encoding

The following diagram shows the encoding for this addressing mode:

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7 0
cond	1	1	0	P	U	N	W	L	Rn	CRd	cp#	offset_8

The P bit Has two meanings:

P == 1 Indicates the use of *post-indexed addressing* or *unindexed addressing* (the W bit determines which). The base register value is used for the memory address.

P == 0 Indicates the use of *offset addressing* or *pre-indexed addressing* (the W bit determines which). The memory address is generated by applying the offset to the base register value.

The U bit Has two meanings:

U == 1 Indicates that the offset is added to the base.

U == 0 Indicates that he offset is subtracted from the base

The N bit The meaning of this bit is coprocessor-dependent. Its recommended use is to distinguish between different-sized values to be transferred.

The W bit Has two meanings:

W == 1 Indicates that the memory address is written back to the base register.

W == 0 Indicates that the base register value is unchanged.

Also:

- If P == 0, this distinguishes unindexed addressing (W == 0) from post-indexed addressing (W == 1). For unindexed addressing, U must equal 1 or the result is either UNDEFINED or UNPREDICTABLE (see *Coprocessor instruction extension space* on page A3-33).
- If P == 1, this distinguishes offset addressing (W == 0) from pre-indexed addressing (W == 1).

The L bit Distinguishes between Load (L == 1) and Store (L == 0) instructions.

5.5.2 Load and Store Coprocessor - Immediate offset

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 8	7 0
cond	1 1 0	1	U	N	0	L	Rn	CRd	cp_num	offset_8

This addressing mode produces a sequence of consecutive addresses. The first address is calculated by adding or subtracting four times the value of an immediate offset to or from the value of the base register Rn. The subsequent addresses in the sequence are produced by incrementing the previous address by four until the coprocessor signals the end of the instruction. This allows a coprocessor to access data whose size is coprocessor-defined.

The coprocessor must not request a transfer of more than 16 words.

Syntax

```
[<Rn>, #+/-<offset_8>*4]
```

where:

`<Rn>`	Specifies the register containing the base address.
`<offset_8>`	Specifies the immediate offset that is multiplied by 4, then added to or subtracted from the value of Rn to form the address.

Architecture version

Version 2 and above

Operation

```
if ConditionPassed(cond) then
    if U == 1 then
        address = Rn + offset_8 * 4
    else /* U == 0 */
        address = Rn - offset_8 * 4
    start_address = address
    while (NotFinished(coprocessor[cp_num]))
        address = address + 4
    end_address = address
```

Notes

The N bit	Is coprocessor-dependent.
The L bit	Distinguishes between Load (L==1) and Store (L==0) instructions.
Use of R15	If R15 is specified as register Rn, the value used is the address of the instruction plus 8.

5.5.3 Load and Store Coprocessor - Immediate pre-indexed

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 8	7 0
cond	1 1 0	1	U	N	1	L	Rn	CRd	cp_num	offset_8

This addressing mode produces a sequence of consecutive addresses. The first address is calculated by adding or subtracting four times the value of an immediate offset to or from the value of the base register Rn. If the condition specified in the instruction matches the condition code status, the first address is written back to the base register Rn. The subsequent addresses in the sequence are produced by incrementing the previous address by four until the coprocessor signals the end of the instruction. This allows a coprocessor to access data whose size is coprocessor-defined.

The coprocessor must not request a transfer of more than 16 words.

Syntax

```
[<Rn>, #+/-<offset_8>*4]!
```

where:

`<Rn>`	Specifies the register containing the base address.
`<offset_8>`	Specifies the immediate offset that is multiplied by 4, then added to or subtracted from the value of Rn to form the address.
!	Sets the W bit, causing base register update.

Architecture version

Version 2 and above

Operation

```
if ConditionPassed(cond) then
    if U == 1 then
        Rn = Rn + offset_8 * 4
    else /* U == 0 */
        Rn = Rn - offset_8 * 4
    start_address = Rn
    address = start_address
    while (NotFinished(coprocessor[cp_num]))
        address = address + 4
    end_address = address
```

Notes

The N bit	Is coprocessor-dependent.
The L bit	Distinguishes between Load (L==1) and Store (L==0) instructions.
Use of R15	Specifying R15 as register Rn has UNPREDICTABLE results.

5.5.4 Load and Store Coprocessor - Immediate post-indexed

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 8	7 0
cond	1 1 0	0	U	N	1	L	Rn	CRd	cp_num	offset_8

This addressing mode produces a sequence of consecutive addresses. The first address is the value of the base register Rn. The subsequent addresses in the sequence are produced by incrementing the previous address by four until the coprocessor signals the end of the instruction. This allows a coprocessor to access data whose size is coprocessor-defined.

If the condition specified in the instruction matches the condition code status, the base register Rn is updated by adding or subtracting four times the value of an immediate offset to or from the value of the base register Rn.

The coprocessor must not request a transfer of more than 16 words.

Syntax

`[<Rn>], #+/-<offset_8>*4`

where:

`<Rn>`	Specifies the register containing the base address.
`<offset_8>`	Specifies the immediate offset that is multiplied by 4, then added to or subtracted from the value of Rn to form the address.

Architecture version

Version 2 and above

Operation

```
if ConditionPassed(cond) then
    start_address = Rn
    if U == 1 then
        Rn = Rn + offset_8 * 4
    else /* U == 0 */
        Rn = Rn - offset_8 * 4
    address = start_address
    while (NotFinished(coprocessor[cp_num]))
        address = address + 4
    end_address = address
```

Notes

The N bit	Is coprocessor-dependent.
The L bit	Distinguishes between Load (L==1) and Store (L==0) instructions.
Use of R15	Specifying R15 as register Rn has UNPREDICTABLE results.

5.5.5 Load and Store Coprocessor - Unindexed

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 8	7 0
cond	1 1 0	0	U	N	0	L	Rn	CRd	cp_num	option

This addressing mode produces a sequence of consecutive addresses. The first address is the value of the base register Rn. The subsequent addresses in the sequence are produced by incrementing the previous address by four until the coprocessor signals the end of the instruction. This allows a coprocessor to access data whose size is coprocessor-defined.

The base register Rn is not updated. Bits[7:0] of the instruction are therefore not used by the ARM, either for the address calculation or to calculate a new value for the base register, and so can be used to specify additional instruction options to the coprocessor.

The coprocessor must not request a transfer of more than 16 words.

Syntax

```
[<Rn>], <option>
```

where:

`<Rn>`	Specifies the register containing the base address.
`<option>`	Specifies additional instruction options to the coprocessor. The `<option>` is specified in the instruction syntax as an integer in the range 0-255, surrounded by { and }.

Architecture version

Version 2 and above

Operation

```
if ConditionPassed(cond) then
    start_address = Rn
    address = start_address
    while (NotFinished(coprocessor[cp_num]))
        address = address + 4
    end_address = address
```

Notes

The N bit	Is coprocessor-dependent.
The L bit	Distinguishes between Load (L==1) and Store (L==0) instructions.
Use of R15	If R15 is specified as register Rn, the value used is the address of the instruction plus 8.
The U bit	If bit[23] (the Up/down bit) is not set, the result is either UNDEFINED or UNPREDICTABLE (see *Coprocessor instruction extension space* on page A3-33).
Option bits	Are unused by the ARM in this addressing mode, and therefore can be used to request additional instruction options in a coprocessor-dependent fashion.

Chapter A6
The Thumb Instruction Set

This chapter introduces the Thumb instruction set and describes how Thumb uses the ARM programmer's model. It contains the following sections:

6.1 About the Thumb instruction set

The Thumb instruction set is a re-encoded subset of the ARM instruction set. Thumb is designed to increase the performance of ARM implementations that use a 16-bit or narrower memory data bus and to allow better code density than ARM. T variants of the ARM architecture incorporate both a full 32-bit ARM instruction set and the 16-bit Thumb instruction set. Every Thumb instruction is encoded in 16 bits.

Thumb does not alter the underlying programmer's model of the ARM architecture. It merely presents restricted access to it. All Thumb data-processing instructions operate on full 32-bit values, and full 32-bit addresses are produced by both data-access instructions and instruction fetches.

When the processor is executing Thumb instructions, eight general-purpose integer registers are available, R0 to R7, which are the same physical registers as R0 to R7 when executing ARM instructions. Some Thumb instructions also access the Program Counter (ARM Register 15), the Link Register (ARM Register 14) and the Stack Pointer (ARM Register 13). Further instructions allow limited access to ARM registers 8 to 15, which are know as the *high registers*.

When R15 is read, bit[0] is zero and bits[31:1] contain the PC. When R15 is written, bit[0] is IGNORED and bits[31:1] are written to the PC. Depending on how it is used, the value of the PC is either the address of the instruction plus 4 or is UNPREDICTABLE.

Thumb does not provide direct access to the CPSR or any SPSR (as the MSR and MRS instructions do in the ARM instruction set)). Thumb execution is flagged by the T bit (bit[5]) in the CPSR:

T == 0	32-bit instructions are fetched (and the PC is incremented by four) and are executed as ARM instructions.
T == 1	16-bit instructions are fetched (and the PC is incremented by two) and are executed as Thumb instructions.

Note

The Thumb instruction set is only compatible with the 32-bit ARM architectures. Thumb is not recommended for use with 26-bit architectures or with 26-bit compatibility options on 32-bit architectures.

6.1.1 Entering Thumb state

Thumb execution is normally entered by executing an ARM `BX` instruction (Branch and Exchange). This instruction branches to the address held in a general-purpose register, and if bit[0] of that register is 1, Thumb execution begins at the branch target address. If bit[0] of the target register is 0, ARM execution continues from the branch target address. On architecture versions 5 and above, `BLX` instructions and `LDR`/`LDM` instructions that load the PC can be used similarly.

Thumb execution can also be initiated by setting the T bit in the SPSR and executing an ARM instruction which restores the CPSR from the SPSR (a data-processing instruction with the S bit set and the PC as the destination, or a Load Multiple with Restore CPSR instruction). This allows an operating system to automatically restart a process independent of whether that process is executing Thumb code or ARM code.

The result is UNPREDICTABLE if the T bit is altered directly by writing the CPSR.

6.1.2 Exceptions

Exceptions generated during Thumb execution switch to ARM execution before executing the exception handler (whose first instruction is at the hardware vector). The state of the T bit is preserved in the SPSR, and the LR of the exception mode is set so that the normal return instruction performs correctly, regardless of whether the exception occurred during ARM or Thumb execution. Table 6-1 lists the values of the exception mode LR for exceptions generated during Thumb execution.

Table 6-1 Exception return instructions

Exception	Exception link register value	Return instruction
Reset	UNPREDICTABLE value	-
Undefined	Address of undefined instruction + 2	`MOVS PC, R14`
SWI	Address of SWI instruction + 2	`MOVS PC, R14`
Prefetch Abort	Address of aborted instruction fetch + 4	`SUBS PC, R14, #4`
Data Abort	Address of the instruction that generated the abort + 8	`SUBS PC, R14, #8`
IRQ	Address of the next instruction to be executed + 4	`SUBS PC, R14, #4`
FIQ	Address of the next instruction to be executed + 4	`SUBS PC, R14, #4`

Note

For each exception, the return instruction indicated by Table 6-1 is the same as the return instruction required if the exception occurred during ARM execution, for the primary or only method of return from that instruction listed in *Exceptions* on page A2-13. However, the following two types of exception have a secondary return method, for which different return instructions are needed depending on whether the exception occurred during ARM or Thumb execution:

- For the Data Abort exception, the primary method of return causes execution to resume at the aborted instruction, which causes it to be re-executed. As described in *Data Abort (data access memory abort)* on page A2-17, it is also possible to return to the next instruction after the aborted instruction, using a `SUBS PC,R14,#4` instruction. If this type of return is required for a data abort caused by a Thumb instruction, use `SUBS PC,R14,#6` for the return instruction.
- For the Undefined Instruction exception, the primary method of return causes execution to resume at the next instruction after the undefined instruction. As described in *Undefined Instruction exception* on page A2-15, it is also possible to return to the undefined instruction itself, using the instruction `SUBS PC,R14,#4`. If this type of return is required for a Thumb undefined instruction, use `SUBS PC,R14,#2` for the return instruction. However, the main use of this type of return is for some types of coprocessor instruction, and as the Thumb instruction set does not contain any coprocessor instructions, you are unlikely to need this secondary method of return for Thumb instructions.

When these secondary methods of return are used, the exception handler code must test the SPSR T bit in order to determine which of the two return instructions to use.

6.2 Instruction set encoding

Figure 6-1 shows the Thumb instruction set encoding. An entry in square brackets, for example [1], indicates a note on the following page.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Shift by immediate	0	0	0	opcode [1]		immediate					Rm			Rd		
Add/subtract register	0	0	0	1	1	0	opc	Rm			Rn			Rd		
Add/subtract immediate	0	0	0	1	1	1	opc	immediate			Rn			Rd		
Add/subtract/compare/move immediate	0	0	1	opcode		Rd / Rn			immediate							
Data-processing register	0	1	0	0	0	0	opcode				Rm / Rs			Rd / Rn		
Special data processing	0	1	0	0	0	1	opcode [1]		H1	H2	Rm			Rd / Rn		
Branch/exchange instruction set [3]	0	1	0	0	0	1	1	1	L	H2	Rm			SBZ		
Load from literal pool	0	1	0	0	1	Rd			PC-relative offset							
Load/store register offset	0	1	0	1	opcode			Rm			Rn			Rd		
Load/store word/byte immediate offset	0	1	1	B	L	offset					Rn			Rd		
Load/store halfword immediate offset	1	0	0	0	L	offset					Rn			Rd		
Load/store to/from stack	1	0	0	1	L	Rd			SP-relative offset							
Add to SP or PC	1	0	1	0	SP	Rd			immediate							
Miscellaneous: See Figure 6-2	1	0	1	1	x	x	x	x	x	x	x	x	x	x	x	x
Load/store multiple	1	1	0	0	L	Rn			register list							
Conditional branch	1	1	0	1	cond [2]				offset							
Undefined instruction	1	1	0	1	1	1	1	0	x	x	x	x	x	x	x	x
Software interrupt	1	1	0	1	1	1	1	1	immediate							
Unconditional branch	1	1	1	0	0	offset										
BLX suffix [4]	1	1	1	0	1	offset										0
Undefined instruction	1	1	1	0	1	x	x	x	x	x	x	x	x	x	x	1
BL/BLX prefix	1	1	1	1	0	offset										
BL suffix	1	1	1	1	1	offset										

Figure 6-1 Thumb instruction set overview

1. The `opc` field is not allowed to be 11 in this line. Other lines deal with the case that the `opc` field is 11.
2. The `cond` field is not allowed to be 1110 or 1111 in this line. Other lines deal with the cases where the `cond` field is 1110 or 1111.
3. The form with `L==1` is UNPREDICTABLE prior to ARM architecture version 5.
4. This is an undefined instruction prior to ARM architecture version 5.

6.2.1 Miscellaneous instructions

Figure 6-2 lists miscellaneous Thumb instructions. An entry in square brackets, for example [1], indicates a note below the figure.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Adjust stack pointer	1	0	1	1	0	0	0	0	opc	immediate						
Push/pop register list	1	0	1	1	L	1	0	R	register list							
Software breakpoint [1]	1	0	1	1	1	1	1	0	immediate							

Figure 6-2 Miscellaneous Thumb instructions

1. This is an undefined instruction prior to ARM architecture version 5.

Note

Any instruction with bits[15:12] = 1011, and which is not shown in Figure 6-2, is an undefined instruction.

6.3 Branch instructions

Thumb supports six types of branch instruction:

- an unconditional branch that allows a forward or backward branch of up to 2KB
- a conditional branch to allow forward and backward branches of up to 256 bytes
- a Branch with Link (subroutine call) is supported with a pair of instructions that allow forward and backward branches of up to 4MB
- a Branch and Exchange instruction branches to an address in a register and optionally switches to ARM code execution
- a Branch with Link and Exchange instruction performs a subroutine call to an address in a register and optionally switches to ARM code execution
- a second form of Branch with Link and Exchange uses a pair of instructions, similar to Branch with Link, but additionally switches to ARM code execution.

The encoding for these instructions is given below.

6.3.1 Conditional branch

```
B<cond>  <target_address>
```

15	14	13	12	11 8	7 0
1	1	0	1	cond	8_bit_signed_offset

6.3.2 Unconditional branch

```
B    <target_address>
BL   <target_address>            ; Produces two 16-bit instructions
BLX  <target_address>            ; Produces two 16-bit instructions
```

15	14	13	12 11	10 0
1	1	1	H	offset_11

6.3.3 Branch with exchange

```
BX    <Rm>
BLX   <Rm>
```

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	1	1	1	L	H2	Rm	SBZ

6.3.4 Examples

```
    B    label        ; unconditionally branch to label

    BCC  label        ; branch to label if carry flag is clear

    BEQ  label        ; branch to label if zero flag is set

    BL   func         ; subroutine call to function

func
...                   ; Include body of function here
...
    MOV  PC, LR       ; R15=R14, return to instruction after the BL

    BX   R12          ; branch to address in R12; begin ARM execution if
                      ; bit 0 of R12 is zero; otherwise continue executing
                      ; Thumb code
```

6.3.5 List of branch instructions

The following instructions follow the formats shown above.

`B`	Conditional Branch. See *B (1)* on page A7-18.
`B`	Unconditional Branch. See *B (2)* on page A7-20.
`BL`	Branch with Link. See *BL, BLX(1)* on page A7-26.
`BX`	Branch and Exchange instruction set. See *BX* on page A7-32.
`BLX`	Branch with Link and Exchange instruction set. See *BL, BLX(1)* on page A7-26 and *BLX(2)* on page A7-30.

6.4 Data-processing instructions

Thumb data-processing instructions are a subset of the ARM data-processing instructions, as shown in Table 6-2. All Thumb data-processing instructions in this table set the condition codes.

Table 6-2 Thumb data-processing instructions

Mnemonic	Operation	Action
`ADC Rd, Rm`	Add with Carry	Rd := Rd + Rm + Carry flag
`ADD Rd, Rn, Rm`	Add	Rd := Rn + Rm
`ADD Rd, Rn, #0 to 7`	Add	Rd := Rn + 3-bit immediate
`ADD Rd, #0 to 255`	Add	Rd := Rd + 8-bit immediate
`AND Rd, Rm`	Logical AND	Rd := Rd AND Rm
`ASR Rd, Rm, #1 to 32`	Arithmetic Shift Right	Rd := Rm ASR 5-bit immediate
`ASR Rd, Rs`	Arithmetic Shift Right	Rd := Rd ASR Rs
`BIC Rd, Rm`	Bit Clear	Rd := Rd AND NOT Rm
`CMN Rn, Rm`	Compare Negated	Update flags after Rn + Rm
`CMP Rn, #0 to 255`	Compare	Update flags after Rn - 8-bit immediate
`CMP Rn, Rm`	Compare	Update flags after Rn - Rm
`EOR Rd, Rm`	Logical Exclusive OR	Rd := Rd EOR Rm
`LSL Rd, Rm, #0 to 31`	Logical Shift Left	Rd := Rm LSL 5-bit immediate
`LSL Rd, Rs`	Logical Shift Left	Rd := Rd LSL Rs
`LSR Rd, Rm, #1 to 32`	Logical Shift Right	Rd := Rm LSR 5-bit immediate
`LSR Rd, Rs`	Logical Shift Right	Rd := Rd LSR Rs
`MOV Rd, #0 to 255`	Move	Rd := 8-bit immediate
`MOV Rd, Rn`	Move	Rd := Rn
`MUL Rd, Rm`	Multiply	Rd := Rm x Rd
`MVN Rd, Rm`	Move Not	Rd := NOT Rm
`NEG Rd, Rm`	Negate	Rd := 0 - Rm
`ORR Rd, Rm`	Logical (inclusive) OR	Rd := Rd OR Rm

Table 6-2 Thumb data-processing instructions (Continued)

Mnemonic	Operation	Action
`ROR Rd, Rs`	Rotate Right	Rd := Rd ROR Rs
`SBC Rd, Rm`	Subtract with Carry	Rd := Rd - Rm - NOT(Carry Flag)
`SUB Rd, Rn, Rm`	Subtract	Rd := Rn - Rm
`SUB Rd, Rn, #0 to 7`	Subtract	Rd := Rn - 3-bit immediate
`SUB Rd, #0 to 255`	Subtract	Rd := Rd - 8-bit immediate
`TST Rn, Rm`	Test	Update flags after Rn AND Rm

For example:

```
ADD    R0, R4, R7        ; R0 = R4 + R7
SUB    R6, R1, R2        ; R6 = R1 - R2
ADD    R0, #255          ; R0 = R0 + 255
ADD    R1, R4, #4        ; R1 = R4 + 4
NEG    R3, R1            ; R3 = 0 - R1
AND    R2, R5            ; R2 = R2 AND R5
EOR    R1, R6            ; R1 = R1 EOR R6
CMP    R2, R3            ; update flags after R2 - R3
CMP    R7, #100          ; update flags after R7 - 100
MOV    R0, #200          ; R0 = 200
```

6.4.1 High registers

There are seven types of data-processing instruction which operate on ARM registers 8 to 14 and the PC as shown in Table 6-3. Apart from `CMP`, instructions in this table do not change the condition code flags.

Table 6-3 High register data-processing instructions

Mnemonic	Operation	Action
`MOV Rd, Rn`	Move	Rd := Rn
`ADD Rd, Rm`	Add	Rd := Rd + Rm
`CMP Rn, Rm`	Compare	Update flags after Rn - Rm
`ADD SP, #0 to 508`	Increment stack pointer	R13 = R13 + 4* (7-bit immediate)
`SUB SP, #0 to 508`	Decrement stack pointer	R13 = R13 - 4* (7-bit immediate)
`ADD Rd, SP, #0 to 1020`	Form Stack address	Rd = R13 + 4* (8-bit immediate)
`ADD Rd, PC, #0 to 1020`	Form PC address	Rd = PC + 4* (8-bit immediate)

For example:

```
MOV    R0, R12              ; R0 = R12
ADD    R10, R1, R2          ; R6 = R1 - R2
MOV    PC, LR               ; PC = R14
CMP    R10, R11             ; update flags after R10 - R11
SUB    SP, #12              ; increase stack size by 12 bytes
ADD    SP, #16              ; decrease stack size by 16 bytes
ADD    R2, SP, #20          ; R2 = SP + 20
ADD    R0, PC, #500         ; R0 = PC + 500
```

6.4.2 Formats

Data-processing instructions use the following eight instruction formats:

Format 1

```
<opcode1>  <Rd>, <Rn>, <Rm>
<opcode1> := ADD | SUB
```

15	14	13	12	11	10	9	8 6	5 3	2 0
0	0	0	1	1	0	op_1	Rm	Rn	Rd

Format 2

```
<opcode2>  <Rd>, <Rn>, #<3_bit_immed>
<opcode2> := ADD | SUB
```

15	14	13	12	11	10	9	8 6	5 3	2 0
0	0	0	1	1	1	op_2	3_bit_immediate	Rn	Rd

Format 3

```
<opcode3>  <Rd>|<Rn>, #<8_bit_immed>
<opcode3> := ADD | SUB | MOV | CMP
```

15	14	13	12 11	10 8	7 0
0	0	1	op_3	Rd|Rn	8_bit_immediate

Format 4

```
<opcode4>  <Rd>, <Rm>, #<shift_imm>
<opcode4> := LSL | LSR | ASR
```

15	14	13	12 11	10 6	5 3	2 0
0	0	0	op_4	shift_immediate	Rm	Rd

Format 5

```
<opcode5>  <Rd>|<Rn>, <Rm>|<Rs>
<opcode5> := MVN | CMP | CMN | TST | ADC | SBC | NEG | MUL |
             LSL | LSR | ASR | ROR | AND | EOR | ORR | BIC
```

15	14	13	12	11	10	9 6	5 3	2 0
0	1	0	0	0	0	op_5	Rm|Rs	Rd|Rn

Format 6

```
ADD  <Rd>, <reg>, #<8_bit_immed>
<reg> := SP | PC
```

15	14	13	12	11	10 8	7 0
1	0	1	0	reg	Rd	8_bit_immediate

Format 7

```
<opcode6>  SP, SP, #<7_bit_immed>
<opcode6> := ADD | SUB
```

15	14	13	12	11	10	9	8	7	6 0
1	0	1	1	0	0	0	0	op_6	7_bit_immediate

Format 8

```
<opcode7>  <Rd>|<Rn>, <Rm>
<opcode7> := MOV | ADD | CMP
```

15	14	13	12	11	10	9 8	7	6	5 3	2 0
0	1	0	0	0	1	opcode	H1	H2	Rm	Rd|Rn

6.4.3 List of data-processing instructions

The following instructions follow the formats shown above.

`ADC`	Add with Carry. See *ADC* on page A7-4.
`ADD`	Add (immediate). See *ADD (1)* on page A7-5.
`ADD`	Add (large immediate). See *ADD (2)* on page A7-6.
`ADD`	Add (register). See *ADD (3)* on page A7-7.
`ADD`	Add high registers. See *ADD (4)* on page A7-8.
`ADD`	Add (immediate to program counter). See *ADD (5)* on page A7-10.
`ADD`	Add (immediate to stack pointer). See *ADD (6)* on page A7-11.
`ADD`	Increment stack pointer. See *ADD (7)* on page A7-12.
`AND`	Logical AND. See *AND* on page A7-13.
`ASR`	Arithmetic Shift Right (immediate). See *ASR (1)* on page A7-14.
`ASR`	Arithmetic Shift Right (register). See *ASR (2)* on page A7-16.
`BIC`	Bit Clear. See *BIC* on page A7-22.
`CMN`	Compare Negative (register). See *CMN* on page A7-34.
`CMP`	Compare (immediate). See *CMP (1)* on page A7-35.
`CMP`	Compare (register). See *CMP (2)* on page A7-36.
`CMP`	Compare high registers. See *CMP (3)* on page A7-37.
`EOR`	Exclusive OR. See *EOR* on page A7-39.
`LSL`	Logical Shift Left (immediate). See *LSL (1)* on page A7-59.
`LSL`	Logical Shift Left (register). See *LSL (2)* on page A7-60.
`LSR`	Logical Shift Right (immediate). See *LSR (1)* on page A7-62.
`LSR`	Logical Shift Right (register). See *LSR (2)* on page A7-64.
`MOV`	Move (immediate). See *MOV (1)* on page A7-66.
`MOV`	Move a low register to another low register. See *MOV (2)* on page A7-67.
`MOV`	Move high registers. See *MOV (3)* on page A7-68.
`MUL`	Multiply. See *MUL* on page A7-70.
`MVN`	Move NOT (register). See *MVN* on page A7-72.
`NEG`	Negate (register). See *NEG* on page A7-73.

`ORR`	Logical OR. See *ORR* on page A7-74.
`ROR`	Rotate Right (register). See *ROR* on page A7-80.
`SBC`	Subtract with Carry (register). See *SBC* on page A7-82.
`SUB`	Subtract (immediate). See *SUB (1)* on page A7-98.
`SUB`	Subtract (large immediate). See *SUB (2)* on page A7-99.
`SUB`	Subtract (register). See *SUB (3)* on page A7-100.
`SUB`	Decrement stack pointer. See *SUB (4)* on page A7-101.
`TST`	Test (register). See *TST* on page A7-103.

6.5 Load and Store Register instructions

Thumb supports eight types of Load and Store Register instructions. Two basic addressing modes are available. These allow the load and store of words, halfwords and bytes, and also the load of signed halfwords and bytes:

- register plus register
- register plus 5-bit immediate (not available for signed halfword and signed byte loads).

If an immediate offset is used, it is scaled by 4 for word access and 2 for halfword accesses.

In addition, three special instructions allow:

- words to be loaded using the PC as a base with a 1KB (word-aligned) immediate offset
- words to be loaded and stored with the stack pointer (R13) as the base and a 1KB (word-aligned) immediate offset.

6.5.1 Formats

Load and Store Register instructions have the following formats:

Format 1

```
<opcode1>  <Rd>, [<Rn>, #<5_bit_offset>]
<opcode1> := LDR|LDRH|LDRB|STR|STRH|STRB
```

15 11	10 6	5 3	2 0
opcode1	5_bit_offset	Rn	Rd

Format 2

```
<opcode2>  <Rd>, [<Rn>, <Rm>]
<opcode2> := LDR|LDRH|LDRSH|LDRB|LDRSB|STR|STRH|STRB
```

15 9	8 6	5 3	2 0
opcode2	Rm	Rn	Rd

Format 3

```
LDR  <Rd>, [PC, #<8_bit_offset>]
```

15	14	13	12	11	10 8	7 0
0	1	0	0	1	Rd	8_bit_immediate

Format 4

```
<opcode3>  <Rd>, [SP, #<8_bit_offset>]
<opcode3>  := LDR | STR
```

15	14	13	12	11	10 8	7 0
1	0	0	1	L	Rd	8_bit_immediate

For example:

```
LDR    R4, [R2, #4]          ; Load word into R4 from address R2 + 4

LDR    R4, [R2, R1]          ; Load word into R4 from address R2 + R1

STR    R0, [R7, #0x7C]       ; Store word from R0 to address R7 + 124

STRB   R1, [R5, #31]         ; Store byte from R1 to address R5 + 31

STRH   R4, [R2, R3]          ; Store halfword from R4 to R2 + R3

LDRH   R3, [R6, R5]          ; Load word into R3 from R6 + R5

LDRB   R2, [R1, #5]          ; Load byte into R2 from R1 + 5

LDR    R6, [PC, #0x3FC]      ; Load R6 from PC + 0x3FC

LDR    R5, [SP, #64]         ; Load R5 from SP + 64

STR    R4, [SP, #0x260]      ; Load R5 from SP + 0x260
```

6.5.2 List of Load and Store Register instructions

The following instructions follow the formats shown above.

`LDR`	Load Word (immediate offset). See *LDR (1)* on page A7-42.
`LDR`	Load Word (register offset). See *LDR (2)* on page A7-44.
`LDR`	Load Word (PC-relative). See *LDR (3)* on page A7-46.
`LDR`	Load Word (SP-relative). See *LDR (4)* on page A7-48.
`LDRB`	Load Unsigned Byte (immediate offset). See *LDRB (1)* on page A7-50.
`LDRB`	Load Unsigned Byte (register offset). See *LDRB (2)* on page A7-51.
`LDRH`	Load Unsigned Halfword (immediate offset). See *LDRH (1)* on page A7-52.
`LDRH`	Load Unsigned Halfword (register offset). See *LDRH (2)* on page A7-54.
`LDRSB`	Load Signed Byte (register offset). See *LDRSB* on page A7-56.
`LDRSH`	Load Signed Halfword (register offset). See *LDRSH* on page A7-57.
`STR`	Store Word (immediate offset). See *STR (1)* on page A7-86.
`STR`	Store Word (register offset). See *STR (2)* on page A7-88.
`STR`	Store Word (SP-relative). See *STR (3)* on page A7-90.
`STRB`	Store Byte (immediate offset). See *STRB (1)* on page A7-92.
`STRB`	Store Byte (register offset). See *STRB (2)* on page A7-93.
`STRH`	Store Halfword (immediate offset). See *STRH (1)* on page A7-94.
`STRH`	Store Halfword (register offset). See *STRH (2)* on page A7-96.

6.6 Load and Store Multiple instructions

Thumb supports four types of Load and Store Multiple instructions:

- Two instructions, `LDMIA` and `STMIA`, are designed to support block copy. They have a fixed Increment After addressing mode from a base register.
- The other two instructions, `PUSH` and `POP`, also have a fixed addressing mode. They implement a full descending stack and the stack pointer (R13) is used as the base register.

All four instructions update the base register after transfer and all can transfer any or all of the lower 8 registers. `PUSH` can also stack the return address, and `POP` can load the PC.

6.6.1 Formats

Load and Store Multiple instructions have the following formats:

Format 1

```
<opcode1>  <Rn>!, <registers>
<opcode1>  := LDMIA | STMIA
```

15	14	13	12	11	10 8	7 0
1	1	0	0	L	Rn	register_list

Format 2

```
PUSH  {<registers>}
POP   {<registers>}
```

15	14	13	12	11	10	9	8	7 0
1	0	1	1	L	1	0	R	register_list

6.6.2 Examples

```
    LDMIA    R7!, {R0-R3, R5}        ; Load R0 to R3-R5 from R7, add 20 to R7
    STMIA    R0!, {R3, R4, R5}       ; Store R3-R5 to R0: add 12 to R0

function
    PUSH     {R0-R7, LR}             ; push onto the stack (R13) R0-R7 and
                                     ; the return address
...                                  ; code of the function body
...
    POP      {R0-R7, PC}             ; restore R0-R7 from the stack
                                     ; and the program counter, and return
```

6.6.3 List of Load and Store Multiple instructions

The following instructions follow the formats shown above.

`LDMIA`	Load Multiple. See *LDMIA* on page A7-40.
`POP`	Pop Multiple. See *POP* on page A7-75.
`PUSH`	Push Multiple. See *PUSH* on page A7-78.
`STMIA`	Store Multiple. See *STMIA* on page A7-84.

6.7 Exception-generating instructions

The Thumb instruction set provides two types of instruction whose main purpose is to cause a processor exception to occur:

- The Software Interrupt (`SWI`) instruction is used to cause a SWI exception to occur (see *Software Interrupt exception* on page A2-16). This is the main mechanism in the Thumb instruction set by which User mode code can make calls to privileged Operating System code.
- The Breakpoint (`BKPT`) instruction is used for software breakpoints in T variants of ARM architecture versions 5 and above. Its default behavior is to cause a Prefetch Abort exception to occur (see *Prefetch Abort (instruction fetch memory abort)* on page A2-16). A debug monitor program that has previously been installed on the Prefetch Abort vector can handle this exception.

  If debug hardware is present in the system, it is allowed to override this default behavior. Details of whether and how this happens are IMPLEMENTATION DEFINED.

6.7.1 Instruction encodings

`SWI <immed_8>`

15	14	13	12	11	10	9	8	7 … 0
1	1	0	1	1	1	1	1	immed_8

`BKPT <immed_8>`

15	14	13	12	11	10	9	8	7 … 0
1	0	1	1	1	1	1	0	immed_8

In both `SWI` and `BKPT`, the immed_8 field of the instruction is ignored by the ARM processor. The SWI or Prefetch Abort handler can optionally be written to load the instruction that caused the exception and extract these fields. This allows them to be used to communicate extra information about the Operating System call or breakpoint to the handler.

6.7.2 List of exception-generating instructions

`BKPT`	Breakpoint. See *BKPT* on page A7-24.
`SWI`	Software Interrupt. See *SWI* on page A7-102.

6.8 Undefined instruction space

The following instructions are UNDEFINED in the Thumb instruction set:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	1	x	x	x	x	x	x	x	x
1	0	1	1	0	x	1	x	x	x	x	x	x	x	x	x
1	0	1	1	1	0	x	x	x	x	x	x	x	x	x	x
1	0	1	1	1	1	1	1	x	x	x	x	x	x	x	x
1	1	0	1	1	1	1	0	x	x	x	x	x	x	x	x

In general, these instructions can be used to extend the Thumb instruction set in the future. However, it is intended that the following group of instructions will not be used in this manner:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	1	0	x	x	x	x	x	x	x	x

Use one of these instructions if you want to use an undefined instruction for software purposes, with minimal risk that future hardware will treat it as a defined instruction.