The ARM architecture is the basis of the world's most widely available 32-bit microprocessor.

ARM Powered microprocessors are being routinely designed into a wider range of products than any other 32-bit processor. This diversity of applicability is made possible by the ARM architecture, resulting in optimal system solutions at the crossroads of high performance, low power consumption, and low cost.

In November 1990, ARM was formed to develop and promote the ARM architecture. With initial investment from:

- Apple Computer, the world's third largest manufacturer of personal computers
- Acorn, the United Kingdom's leading supplier of Information Technology for education
- VLSI Technology, the world leading ASIC supplier

ARM Ltd. began work to establish the ARM architecture as the 32-bit standard microprocessor architecture.

Initially, ARM devices were made by VLSI Technology, but ARM's business model is to license microprocessor designs to a broad range of semiconductor manufacturers, allowing focus on a variety of end-user applications. To date, this licensing strategy has resulted in twelve companies manufacturing ARM-based designs:

<table>
<thead>
<tr>
<th>VLSI Technology</th>
<th>GEC Plessey Semiconductors (GPS)</th>
<th>Sharp Corporation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Texas Instruments (TI)</td>
<td>Cirrus Logic</td>
<td>Asahi Kasai Microsystems (AKM)</td>
</tr>
<tr>
<td>Samsung Corporation</td>
<td>Digital Equipment Corporation</td>
<td>European Silicon Structures (ES2)</td>
</tr>
<tr>
<td>NEC Corporation</td>
<td>Symbios Logic</td>
<td>Lucky Goldstar Corporation</td>
</tr>
</tbody>
</table>

Through these twelve semiconductor partners, the ARM processor is currently being used in a wider range of applications than any other 32-bit microprocessor. ARM chooses new partners carefully; each new partner is judged on their ability to extend the applicability of ARM processors into new market areas, broadening the total ARM product range by adding their unique expertise.

Customers using ARM processors benefit not only from the ARM architecture, but the ability to select the most appropriate silicon manufacturer. Furthermore, the worldwide awareness of the ARM processor has attracted third-party developers to the ARM architecture, yielding a huge variety of ARM support products:

- ISI, Microtec, Accelerated Technology, Cygnus, Eonic Systems and Perihelion all offer Operating Systems for the ARM.
- Yokogawa Digital and Lauterbach provide In-Circuit Emulators (ICE).
- Hewlett Packard Logic analysers support ARM processors.

In ARM Ltd's five-year history, it has delivered over 30 unique microprocessor and support-chip designs. ARM microprocessors are routinely being designed into hundreds of products including:

- cellular telephones
- interactive game consoles
- organisers
- disk drives
- modems
- high performance workstations
- graphics accelerators
- car navigation systems
- video phones
- digital broadcast set-top decoders
- cameras
- smart cards
- telephone switchboards
- laser printers
Foreword

This product range includes microprocessors designed as macrocell cores, with around 4mm\(^2\) of silicon, drawing less than fifty milliWatts of power when executing over 30 MIPS of sustained performance. For very high performance applications, ARM implementations deliver over 200 MIPS sustained performance (more than most high-performance computer workstations), while still only consuming one half of one Watt.

No other processor architecture can offer implementations in these extremes; no other processor has ARM's broad applicability across an entire product range.

The *ARM Architecture Reference Manual* is the definitive description of the programmers' model of all ARM microprocessors, and is ARM's commitment to users of the ARM processor for compatibility and interworking across products designed and manufactured by many different companies.

Through this commitment, the benefits of an ARM processor can be harnessed to establish and maintain an industry lead that only ARM Powered products can achieve.

Dave Jaggar. July 1996
Contents

Preface
   Introduction ix
   Using this Manual x
   Conventions xi
   Terminology xi

1 Architecture Overview 1-1
  1.1 Overview 1-2
  1.2 Exceptions 1-3
  1.3 ARM Instruction Set 1-4
  1.4 Branch Instructions 1-4
  1.5 Data-processing Instructions 1-4
  1.6 Load and Store Instructions 1-5
  1.7 Coprocessor Instructions 1-6

2 Programmer’s Model 2-1
  2.1 Data Types 2-2
  2.2 Processor Modes 2-2
  2.3 Registers 2-3
  2.4 Program Status Registers 2-3
  2.5 Exceptions 2-6
Contents

3 The ARM Instruction Set 3-1
  3.1 Using this Chapter 3-2
  3.2 Instruction Set Overview 3-3
  3.3 The Condition Field 3-4
  3.4 Branch Instructions 3-6
  3.5 Data Processing 3-7
  3.6 Multiply Instructions 3-10
  3.7 Status Register Access 3-12
  3.8 Load and Store Instructions 3-13
  3.9 Load and Store Word or Unsigned Byte Instructions 3-15
  3.10 Load and Store Halfword and Load Signed Byte Instructions 3-16
  3.11 Load and Store Multiple Instructions 3-17
  3.12 Semaphore Instructions 3-19
  3.13 Coprocessor Instructions 3-20
  3.14 Extending the Instruction Set 3-22
  3.15 Alphabetical List of ARM Instructions 3-30
  3.16 Data-processing Operands 3-84
  3.17 Load and Store Word or Unsigned Byte Addressing Modes 3-98
  3.18 Load and Store Halfword or Load Signed Byte Addressing Modes 3-109
  3.19 Load and Store Multiple Addressing Modes 3-116
  3.20 Load and Store Multiple Addressing Modes (Alternative names) 3-121
  3.21 Load and Store Coprocessor Addressing Modes 3-123

4 ARM Code Sequences 4-1
  4.1 Arithmetic Instructions 4-2
  4.2 Branch Instructions 4-4
  4.3 Load and Store Instructions 4-6
  4.4 Load and Store Multiple Instructions 4-8
  4.5 Semaphore Instructions 4-9
  4.6 Other Code Examples 4-10

5 The 26-bit Architectures 5-1
  5.1 Introduction 5-2
  5.2 Format of Register 15 5-3
  5.3 Writing just the PSR in 26-bit architectures 5-4
  5.4 26-bit PSR Update Instructions 5-5
  5.5 Address Exceptions 5-6
  5.6 Backwards Compatibility from 32-bit Architectures 5-7
This preface describes the ARM Architecture Version 4, also known as ARMv4, and lists the conventions and terminology used in this manual.

- Introduction  x
- Using this Manual  xi
- Conventions  xi
- Terminology  xi
The ARM Architecture exists in 4 major versions:

**Version 1** was implemented only by ARM1, and was never used in a commercial product.

**Version 2** extended Version 1 by adding a multiply instruction, multiply accumulate instruction, coprocessor support, two more banked registers for FIQ mode and later (Version 2a) an Atomic Load and Store instruction (called SWP) and the use of Coprocessor 15 as a system control coprocessor. Version 1, 2 and 2a all support a 26-bit address bus and combine in register 15 a 24-bit Program Counter (PC) and 8 bits of processor status. Version 2 has just three privileged processing modes, Supervisor, IRQ and FIQ.

**Version 3** extended the addressing range to 32 bits, defining just a 30-bit Program Counter value in register 15, and added a separate 11-bit status register (the Current Program Status Register or CPSR) to contain the status information that was previously in register 15. Version 3 added two new privileged processing modes to allow coprocessor emulation and virtual memory support in supervisor mode:

- Undefined
- Abort

Five more status registers (the Saved Program Status Registers, SPSRs) were defined, one for each privileged processor mode, to preserve the CPSR contents when the exception occurs corresponding to each privileged processor mode.

Version 3 supports both hardware and software emulation of Version 2a. The processor can be forced to be Version 2a only in hardware, allowing full backwards compatibility, but no version 3 advantages. Version 3 can also be switched in software into a Version 2 execution model to support Version 2a on a process by process basis to allow a smooth upgrade path from Version 2 to Version 3. See Chapter 5, *The 26-bit Architectures* for information on the differences between the 26-bit architectures (version 1, 2 and 2a) and the 32-bit architectures (Version 3, 3M, 4 and 4T).

**Version 3G** is the same as version 3, without the backwards compatibility support for version 2 architectures.

**Version 3M** adds signed and unsigned multiply and multiply accumulate instructions that produce a 64-bit result to Version 3. The new ARMv3M instructions are marked as such in the text.

**Version 4** adds halfword load and store instructions, sign-extending byte and halfword load instructions, adds a new privileged processor mode (that uses the user mode registers) and defines several new undefined instructions.

**Version 4T** incorporates an instruction decoder for a 16-bit subset of the ARM instruction set (called THUMB).
Using this Manual

The information in this manual is organised as follows:

- **Chapter 1** gives an overview of the ARM architecture
- **Chapter 2** describes the programmer’s model
- **Chapter 3** lists and describes the ARM instruction set
- **Chapter 4** gives examples of coding algorithms
- **Chapter 5** describes the differences between 32-bit and 26-bit architectures
- **Chapter 6** lists and describes the Thumb instruction set
- **Chapter 7** describes ARM system architecture, and the system control processor

Conventions

This manual employs typographic conventions intended to improve its ease of use.

**code**

A monospaced typewriter font shows code which you need to enter, or code which is provided as an example.

Terminology

This manual uses the following terminology:

- **Abort**
  - is caused by an illegal memory access. Abort can be caused by the external memory system or the MMU

- **AND**
  - performs a bitwise AND

- **Arithmetic_Shift_Right**
  - performs a right shift, repeatedly inserting the original left-most bit (the sign bit) in the vacated bit positions on the left

- **ARM instruction**
  - is a word that is word-aligned

- **assert statements**
  - are used to ensure a certain condition has been met

- **Assignment**
  - is signified by =

- **Binary numbers**
  - are preceded by 0b

- **Boolean AND**
  - is signified by the “and” operator
Boolean OR

is signified by the “or” operator

BorrowFrom

returns 1 if the following subtract causes a borrow (the true unsigned result is less than 0)

returns 0 in all other cases

Byte

is an 8-bit data item

CarryFrom

returns 1 if the following addition causes a carry (the result is bigger than $2^{31} - 1$)

returns 0 in all other cases

case ... endcase statements

are used to indicate a one of many execution option. Indentation indicates the range of statements in each option

Comments

are enclosed in /* */

ConditionPassed(cond)

returns true if the state of the N, Z, C and Z flags fulfils the condition encoded in the cond argument

returns false in all other cases

CPSR

is the Current Program Status Register

CurrentModeHasSPSR()

returns true if the current processor mode is not User mode or System mode

returns false if the current mode is User mode or System mode

Do-not-modify fields (DNM)

means the value must not be altered by software. DNM fields read as UNPREDICTABLE values, and may only be written with the same value read from the same field as the same processor.

Elements

are separated by | in a list of possible values for a variable

EOR

performs an Exclusive OR

Exception

is a mechanism to handle an event; for example, an external interrupt or an undefined instruction

External abort

is an abort that is generated by the external memory system
Fault
is an abort that is generated by the MMU

General-purpose register
is one of the 32-bit general-purpose integer registers, R0 to R14

Halfword
is a 16-bit data item

Hexadecimal numbers
are preceded by 0x

if ... else if ... else statements
are used to signify conditional execution. Indentation indicates the range of statements in each condition

IGNORE fields (IGN)
must ignore writes

Immediate and offset fields
are unsigned unless otherwise stated

IMPLEMENTATION-DEPENDENT fields (IMP)
are not architecturally specified, but must be defined and documented by individual implementations

InAPrivilegedMode()
returns true if the current processor mode is not User mode;
returns false if the current mode is User mode

Logical_Shift_Left
performs a left shift, inserting zeroes in the vacated bit positions on the right.
<< is used as a short form for Logical_Shift_Left

Logical_Shift_Right
performs a right shift, inserting zeroes in the vacated bit positions on the left

LR (Link Register)
is integer register R14

Memory[<address>,<size>]
refers to a data item in memory of length <size>, at address <address>, aligned on a <size> byte boundary.
The data item is zero-extended to 32 bits.
Currently defined sizes are:
1 for bytes
2 for halfwords
4 for words
To align on a <size> boundary, halfword accesses ignore address[0] and word accesses ignore address[1:0]
NOT
  performs a Complement

\textbf{NotFinished(CP\_number)}
  returns true if the coprocessor signified by the CP\_number argument has
  signalled that the current operation is complete
  returns false in all other cases

\textbf{NumberOfSetBitsIn(bitfield)}
  performs a population count on (counts the set bits in) the bitfield argument

\textbf{Object[from:to]}
  indicates the bit field extracted from Object starting at bit “from”, ending with bit “to”
  (inclusive)

\textbf{Optional parts of instructions}
  are surrounded by \{ and \}

\textbf{OR}
  performs an Inclusive OR

\textbf{OverflowFrom}
  returns 1 if the following addition (or subtraction) causes a carry (or borrow) to
  (from) bit\[31\]. Addition generates a carry if both operands have the same sign
  (bit\[31\]), and the sign of the result is different to the sign of both operands.
  Subtraction causes an overflow if the operands have different signs, and the first
  operand and the result have different signs

\textbf{PC (Program Counter)}
  is integer register R15 (or bits\[25:2\] of register 15 on 26-bit architectures)

\textbf{PSR}
  is the CPSR or one of the SPSRs (or bits\[31:26\] and bits\[1:0\] of register 15 on 26-bit
  architectures)

\textbf{Read-as-zero fields (RAZ)}
  appear as zero when read

\textbf{Read-Modify-Write fields (RMW)}
  should be read to a general-purpose register, the relevant fields updated in
  the register, and the register value written back.

\textbf{Rotate\_Right}
  performs a right rotate, where each bit that is shifted off the right is inserted on
  the left

\textbf{Security hole}
  is an illegal mechanism that bypasses system protection

\textbf{Should-be-one fields (SBO)}
  should be written as one (or all 1s for bit fields) by software.
  Values other than one values produce unpredictable results
Should-be-one-or-preserved fields (SBOP)
should be written as one (or all 1s for bit fields) or preserved by writing the same
value that has been previously read from the same fields on the same processor.

Should-be-zero fields (SBZ)
should be written as zero (or all 0s for bit fields) by software.
Non-zero values produce unpredictable results

Should-be-zero-or-preserved fields (SBZP)
should be written as zero (or all 0s for bit fields) or preserved by writing the same
value that has been previously read from the same fields on the same processor.

Signed immediate and offset fields
are encoded in two’s complement notation unless otherwise stated

SignExtend(arg)
sign-extends (propagates the sign bit) its argument to 32 bits

SPSR
is the Saved Program Status Register

Test for equality
is signified by ==

THUMB instruction
is a halfword that is halfword-aligned

Unaffected items
are not changed by a particular operation

UNDEFINED
indicates an instruction that generates an undefined instruction trap. See 2.5
Exceptions on page 2-6 for information on undefined instruction traps

UNPREDICTABLE
means the result of an instruction cannot be relied upon.
unpredictable instructions or results must not represent security holes.
UNPREDICTABLE instructions must not halt or hang the processor, or any parts of
the system

UNPREDICTABLE fields (UNP)
do not contain valid data, and a value may vary from moment to moment,
instruction to instruction, and implementation to implementation

Variable name
(a symbolic name for values) is surrounded by < and >

while .... statements
are used to indicate a loop. Indentation indicates the range of statements in
the loop

Word
is a 32-bit data item
Architecture Overview
The ARM architecture has been designed to allow very small, yet high-performance implementations. It is the architectural simplicity of ARM which leads to very small implementations, and small implementations allow devices with very low power consumption.

1.1 Overview
1.2 Exceptions
1.3 ARM Instruction Set
1.4 Branch Instructions
1.5 Data-processing Instructions
1.6 Load and Store Instructions
1.7 Coprocessor Instructions
1.1 Overview

The ARM is a RISC (Reduced Instruction Set Computer), as it incorporates all the features of a typical RISC architecture:

- a large uniform register file
- a load-store architecture (data-processing operations only operate on register contents)
- simple addressing modes (data loaded and stored from an address specified in registers and instruction fields)
- uniform and fixed length instruction fields (which simplify instruction decode)

In addition, the ARM architecture provides these features:

- control over both the ALU and shifter in every data-processing instruction to maximise the use of a shifter and an ALU
- auto-increment and auto-decrement addressing modes to optimise program loops
- load and store multiple instructions to maximise data throughput
- conditional execution of all instructions to maximise execution throughput

Together, these architectural enhancements to a basic RISC architecture allow implementations that can balance high performance, low power consumption and minimal die size in every implementation.

1.1.1 ARM registers

ARM has thirty-one, 32-bit registers. At any one time, sixteen 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 registers.

The main bank of sixteen registers is used by all non-privileged code; these are the User mode registers. User mode is different from all other modes, as it is non-privileged, which means that user mode is the only mode which cannot switch to another processor mode (without generating an exception).

Program counter

Register 15 is the Program Counter (or PC), and 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 4 bytes long (one 32-bit word), and are always aligned on a word boundary, so the PC contains just 30 bits; the bottom two bits are always zero.

Link register

Register 14 is called the Link Register (or LR). Its special purpose is to hold the address of the next instruction after a Branch with 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.

Other registers

The remaining 14 registers have no special hardware purpose - their uses are defined purely by software. Software will normally use R13 as a stack pointer (or SP).
1.2 Exceptions

ARM supports 5 types of exception, and a privileged processing mode for each type. The 5 types of exceptions are:

- two levels of interrupt (fast and normal)
- memory aborts (used to implement memory protection or virtual memory)
- attempted execution of an undefined instruction
- software interrupts (SWIs) (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 exceptions have replacement (or banked) registers for R14 and R13, and one interrupt mode has more registers for fast interrupt processing.

After an exception, R14 holds the return address for exception processing, which is used both to return after the exception is processed and to address the instruction that caused the exception.

R13 is banked across exception modes to provide each exception handler with private stack pointer (SP). 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 seventh processing mode, System mode, that does not have any banked registers (it uses the User mode registers), which is used to run normal (non-exception) tasks that require a privileged processor mode.

CPSR and SPSR

All other processor state is held in status registers. The current operating processor status is in the Current Program Status Register or 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

All 5 exception modes also have a Saved Program Status Register (SPSR) which holds the CPSR of the task immediately before the exception occurred. Both the CPSR and SPSR are accessed with special instructions.

The exception process

When an exception occurs, ARM halts execution after the current instruction and begins execution at a fixed address in low memory, known as the exception vectors. There is a separate vector location for each exception (and two for memory aborts to distinguish between data and instruction accesses).

An operating system will install an handler on every exception at initialisation. Privileged operating system tasks are normally run in System mode to allow exceptions to occur within the operating system without state loss (exceptions overwrite their R14 when an exception occurs, and System mode is the only privileged mode that cannot be entered by an exception).
1.3 ARM Instruction Set

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

- branch
- data-processing
- load and store
- coprocessor

Conditional execution

All ARM instructions may be conditionally executed. 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. Subsequent instructions can be conditionally executed according to the status of the condition code flags. Fifteen conditions are implemented, depending on particular values of the condition code flags; one condition actually ignores the condition code flag so that normal (unconditional) instructions always execute.

1.4 Branch Instructions

As well as allowing any data-processing or load instruction to change control flow (by writing the Program Counter) a standard branch instruction is provided with 24-bit signed offset, allowing forward and backward branches of up to 32Mbytes.

There is a Branch with Link option that also preserves the address of the instruction after the branch in R14 (the Link Register or LR), allowing a move instruction to put the LR in to PC and return to the instruction after the branch, providing a subroutine call.

There also a special type of branch instruction called software interrupt (SWI). This makes a call to the Operating System (to request an OS-defined service). SWI also changes the processor mode, allowing an unprivileged task to gain OS privilege (access to which is controlled by the OS).

On processors that implement the THUMB instruction set there is a branch instruction that jumps to an address specified in a register, and optionally switches instruction set, allowing ARM code to call THUMB code and THUMB code to call ARM code. An overview of the THUMB instruction is provided in Chapter 6, The Thumb Instruction Set.

1.5 Data-processing Instructions

The data-processing instructions perform some operation on the general-purpose registers. There are three types of data-processing instructions:

- data-processing instructions proper
- multiply instructions
- status register transfer instructions

Arithmetic/logic instructions

There are 16 arithmetic/logic instructions which share a common instruction format. This format takes up to two source operands, performs an arithmetic/logic operation on those operands, and then most store the result into a register, and optionally update the condition code flags according to that result.
There are four data-processing instructions which don't store their result in a register. They compare the values of the source operands and then update the condition code flags.

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 may an immediate value or the value of another register, and four types of shift can be specified. So, every data-processing instruction can perform a data-processing and a shift operation. As a result, ARM does not have dedicated shift instructions.

Because the Program Counter (PC) is a general-purpose register, this class of data-processing instruction may write their results directly to the PC, allowing a variety of jump instructions to be easily implemented.

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

- (normal) 32-bit result store the 32-bit result in a register
- (long) 64-bit result store the 64-bit result in two separate registers

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

Status register transfer instructions
The status register transfer instructions transfer the contents of the CPSR or a SPSR to or from a general-purpose register. Writing to the CPSR is one way to set the value of the condition code flags and interrupt enable flags and to set the processor mode.

1.6 Load and Store Instructions
Load and store instruction come in three types:

1. load or store the value of a single register
2. load and store multiple register values
3. swap a register value with the value of a memory location

Load and store single 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 may be automatically zero- 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 instructions have three primary addressing modes that are formed by adding or subtracting an immediate or register-based offset to or from a base register (register-based offsets may also be scaled with shift operations):

1. offset
2. pre-indexed
3. post-indexed

Pre- and post-indexed addressing modes update the base register with the base plus offset calculation. As the Program Counter (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 4Gbyte memory space.

Load and store multiple registers
Load and Store multiple instructions perform a block transfer of any number of the general-purpose registers to or from memory. Four addressing modes are provided:

1. pre-increment
2. post-increment
3. pre-decrement
4. post-decrement

The base address is specified by a register value (which may be optionally updated after the transfer). As the subroutine return address and Program Counter (PC) values are in general-purpose registers, very efficient subroutine call and return can be constructed with Load and Store Multiple; register contents and the return address can be stacked and the stack pointer updated with single store multiple instruction at procedure entry and then register contents restored, the PC loaded with the return address and the stack pointer updated on procedure return with a single load multiple).

Of course, load and store multiple also allow very efficient data movement (for example, block copy).

Swap a register value with the value of a memory location
Swap can load a value from a register-specified memory location, store the contents of a register to the same memory location, then write the loaded value to a register.

By specifying the same register as the load and store value, 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.

1.7 Coprocessor Instructions

There are three types of coprocessor instructions:

- data-processing instructions: start a coprocessor-specific internal operation
- register transfers: allow a coprocessor value to be transferred to or from an ARM register
- data-transfer instructions: transfer coprocessor data to or from memory, where the address of the transfer is calculated by the ARM
2

Programmer’s Model
This chapter introduces the ARM Programmer’s Model.

2.1 Data Types
2.2 Processor Modes
2.3 Registers
2.4 Program Status Registers
2.5 Exceptions
2.1 Data Types

ARM Architecture Version 4 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

ARM instructions are exactly one word (and therefore aligned on a four-byte boundary).

THUMB instructions are exactly one halfword (and therefore aligned on a two-byte boundary).

All data operations (e.g. 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. Signed operands are in two's complement format.

2.2 Processor Modes

ARM Version 4 supports seven processor modes:

<table>
<thead>
<tr>
<th>Processor mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 User</td>
<td>(usr) normal program execution mode</td>
</tr>
<tr>
<td>2 FIQ</td>
<td>(fiq) supports a high-speed data transfer or channel process</td>
</tr>
<tr>
<td>3 IRQ</td>
<td>(irq) used for general purpose interrupt handling</td>
</tr>
<tr>
<td>4 Supervisor</td>
<td>(svc) a protected mode for the operating system</td>
</tr>
<tr>
<td>5 Abort</td>
<td>(abt) implements virtual memory and/or memory protection</td>
</tr>
<tr>
<td>6 Undefined</td>
<td>(und) supports software emulation of hardware coprocessors</td>
</tr>
<tr>
<td>7 System</td>
<td>(sys) runs privileged operating system tasks (Architecture Version 4 only)</td>
</tr>
</tbody>
</table>

**Table 2-1: ARM Version 4 processor modes**

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

The processor has a total of 37 registers:

- 30 general-purpose registers
- 6 status registers
- a program counter

The registers are arranged in partially overlapping banks: a different register bank for each processor mode. At any one time, 15 general-purpose registers (R0 to R14), one or two status registers and the program counter are visible. The general-purpose registers and status registers currently visible depend on the current processor mode. The register bank organisation is shown in Figure 2-1: Register organisation on page 2-4. The banked registers are shaded in the diagram.

The general-purpose registers are 32 bits wide.

Register 13 (the Stack Pointer or SP) is banked across all modes to provide a private Stack Pointer for each mode (except system mode which shares the user mode R13).

Register 14 (the Link Register or LR) is used as the subroutine return address link register. R14 is also banked across all modes (except system mode which shares the user mode R14). When a Subroutine call (Branch and Link instruction) is executed, R14 is set to the subroutine return address; R14_svc, R14_irq, R14_fiq, R14_abort and R14_undef are used similarly to hold the return address when exceptions occur (or a subroutine return address if subroutine calls are executed within interrupt or exception routines). R14 may be treated as a general-purpose register at all other times.

FIQ mode also has banked registers R8 to R12 (as well as R13 and R14). R8_fiq, R9_fiq, R10_fiq, R11_fiq and R12_fiq are provided to allow very fast interrupt processing (without the need to preserve register contents by storing them to memory), and to preserve values across interrupt calls (so that register contents do not need to be restored from memory).

Register R15 holds the Program Counter (PC). When R15 is read, bits [1:0] are zero and bits [31:2] contain the PC. When R15 is written, n bits[1:0] are ignored and bits[31:2] are written to the PC. Depending on how it is used, the value of the PC is either the address of the instruction plus 8 or is UNPREDICTABLE.

The Current Program Status Register (CPSR) is also accessible in all processor modes. It contains condition code flags, interrupt enable flags and the current mode. Each privileged mode (except system mode) has a Saved Program Status Register (SPSR) which is used to preserve the value of the CPSR when an exception occurs.

2.4 Program Status Registers

The format of the Current Program Status Register (CPSR) and the Saved Program Status registers (SPSR) are shown in Figure 2-2: Format of the program status registers on page 2-4. The N, Z, C and V (Negative, Zero, Carry and oVerflow) bits are collectively known as the condition code flags. The condition code flags in the CPSR can be changed as a result of arithmetic and logical operations in the processor and can be tested by all instructions to determine if the instruction is to be executed.
### Mode

<table>
<thead>
<tr>
<th>Mode</th>
<th>User/System</th>
<th>Supervisor</th>
<th>Abort</th>
<th>Undefined</th>
<th>Interrupt</th>
<th>Fast interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
</tr>
<tr>
<td>R1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
</tr>
<tr>
<td>R2</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
</tr>
<tr>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
</tr>
<tr>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
</tr>
<tr>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
</tr>
<tr>
<td>R6</td>
<td>R6</td>
<td>R6</td>
<td>R6</td>
<td>R6</td>
<td>R6</td>
<td>R6</td>
</tr>
<tr>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
</tr>
<tr>
<td>R8</td>
<td>R8</td>
<td>R8</td>
<td>R8</td>
<td>R8</td>
<td>R8</td>
<td>R8</td>
</tr>
<tr>
<td>R9</td>
<td>R9</td>
<td>R9</td>
<td>R9</td>
<td>R9</td>
<td>R9</td>
<td>R9</td>
</tr>
<tr>
<td>R10</td>
<td>R10</td>
<td>R10</td>
<td>R10</td>
<td>R10</td>
<td>R10</td>
<td>R10_FIQ</td>
</tr>
<tr>
<td>R11</td>
<td>R11</td>
<td>R11</td>
<td>R11</td>
<td>R11</td>
<td>R11</td>
<td>R11_FIQ</td>
</tr>
<tr>
<td>R12</td>
<td>R12</td>
<td>R12</td>
<td>R12</td>
<td>R12</td>
<td>R12</td>
<td>R12_FIQ</td>
</tr>
<tr>
<td>R13</td>
<td>R13_SVC</td>
<td>R13_ABORT</td>
<td>R13</td>
<td>R13_UNDEF</td>
<td>R13_IRQ</td>
<td>R13_FIQ</td>
</tr>
<tr>
<td>R14</td>
<td>R14_SVC</td>
<td>R14_ABORT</td>
<td>R14</td>
<td>R14_UNDEF</td>
<td>R14_IRQ</td>
<td>R14_FIQ</td>
</tr>
<tr>
<td>PC</td>
<td>PC</td>
<td>PC</td>
<td>PC</td>
<td>PC</td>
<td>PC</td>
<td>PC</td>
</tr>
</tbody>
</table>

CPSR = banked register

---

#### Figure 2-1: Register organisation

![Register organisation diagram](image)

#### Figure 2-2: Format of the program status registers

![Program status register format](image)
2.4.1 The control bits

The bottom 8 bits of a 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. The I and F bits are the interrupt disable bits:

- I bit
  - disables IRQ interrupts when it is set
  - disables FIQ interrupts when it is set

The T flag is only implemented on Architecture Version 4T (THUMB):

- 0 indicates ARM execution
- 1 indicates THUMB execution

On all other version of the architecture the T flag should be zero (S8Z).

2.4.2 The mode bits

The M0, M1, M2, M3 and M4 bits (M[4:0]) are the mode bits, and these determine the mode in which the processor operates. The interpretation of the mode bits is shown in Table 2-2: **The mode bits**. Not all combinations of the mode bits define a valid processor mode. Only those explicitly described can be used; if any other value is programmed into the mode bits M[4:0], the result is unpredictable.

<table>
<thead>
<tr>
<th>M[4:0]</th>
<th>Mode</th>
<th>Accessible Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10000</td>
<td>User</td>
<td>PC, R14 to R0, CPSR</td>
</tr>
<tr>
<td>0b10001</td>
<td>FIQ</td>
<td>PC, R14_fiq to R8_fiq, R7 to R0, CPSR, SPSR_fiq</td>
</tr>
<tr>
<td>0b10010</td>
<td>IRQ</td>
<td>PC, R14_irq, R13_irq, R12 to R0, CPSR, SPSR_irq</td>
</tr>
<tr>
<td>0b10011</td>
<td>SVC</td>
<td>PC, R14_svc, R13_svc, R12 to R0, CPSR, SPSR_svc</td>
</tr>
<tr>
<td>0b10111</td>
<td>Abort</td>
<td>PC, R14_abt, R13_abt, R12 to R0, CPSR, SPSR_abt</td>
</tr>
<tr>
<td>0b11011</td>
<td>Undef</td>
<td>PC, R14_und, R13_und, R12 to R0, CPSR, SPSR_und</td>
</tr>
<tr>
<td>0b11111</td>
<td>System</td>
<td>PC, R14 to R0, CPSR (Architecture Version 4 only)</td>
</tr>
</tbody>
</table>

*Table 2-2: The mode bits*

User mode and system mode do not have an SPSR, as these modes are not entered on any exception, so a register to preserve the CPSR is not required. In User mode or System mode any reads to the SPSR will read an unpredictable value, and any writes to the SPSR will be ignored.
2.5 Exceptions

Exceptions are generated by internal and external sources to cause the processor to handle an event; for example, 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 may arise at the same time.

ARM supports 7 types of exception and has a privileged processor mode for each type of exception. Table 2-3: Exception processing modes 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 Hard Vectors.

The reserved entry at address 0x14 is for an Address Exception vector used when the processor is configured for a 26-bit address space. See Chapter 5, The 26-bit Architectures for more information.

<table>
<thead>
<tr>
<th>Exception type</th>
<th>Mode</th>
<th>Vector address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>SVC</td>
<td>0x00000000</td>
</tr>
<tr>
<td>Undefined instructions</td>
<td>UNDEF</td>
<td>0x00000004</td>
</tr>
<tr>
<td>Software Interrupt (SWi)</td>
<td>SVC</td>
<td>0x00000008</td>
</tr>
<tr>
<td>Prefetch Abort (Instruction fetch memory abort)</td>
<td>ABORT</td>
<td>0x0000000c</td>
</tr>
<tr>
<td>Data Abort (Data Access memory abort)</td>
<td>ABORT</td>
<td>0x00000010</td>
</tr>
<tr>
<td>IRQ (Interrupt)</td>
<td>IRQ</td>
<td>0x00000018</td>
</tr>
<tr>
<td>FIQ (Fast Interrupt)</td>
<td>FIQ</td>
<td>0x0000001c</td>
</tr>
</tbody>
</table>

Table 2-3: Exception processing modes

When taking an exception, the banked registers are used to save state. When an exception occurs, these actions are performed:

- R14_<exception_mode> = PC
- SPSR_<exception_mode> = CPSR
- CPSR[5:0] = Exception mode number
- CPSR[6] = if <exception_mode> == Reset or FIQ then = 1 else unchanged
- CPSR[7] = 1; Interrupt disabled
- 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:

1. Using a data-processing instruction with the S bit set, and the PC as the destination.
2. Using the Load Multiple and Restore PSR instruction.

The following sections show the recommended way of returning from each exception.
2.5.1  Reset

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

R14_svc = unpredictable value
SPSR_svc = CPSR
CPSR[5:0] = 0b010011 ; Supervisor mode
CPSR[6] = 1 ; Fast Interrupts disabled
CPSR[7] = 1 ; Interrupts disabled
PC = 0x0

Therefore, after reset, ARM begins execution at address 0x0 in supervisor mode with interrupts disabled. See 7.6 Memory Management Unit (MMU) Architecture on page 7-14 for more information on the effects of Reset.

2.5.2  Undefined instruction exception

If ARM 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 3.14.5 Undefined instruction Space on page 3-27).

The undefined instruction exception may 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 undefined instruction + 4
SPSR_und = CPSR
CPSR[5:0] = 0b011011 ; Undefined mode
CPSR[6] = unchanged ; Fast Interrupt status is unchanged
CPSR[7] = 1 ; (Normal) Interrupts disabled
PC = 0x4

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.

2.5.3  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 are performed:
R14_svc = address of SWI instruction + 4
SPSR_svc = CPSR
CPSR[5:0] = 0b010011 ; Supervisor mode
CPSR[6] = unchanged ; Fast Interrupt status is unchanged
CPSR[7] = 1 ; (Normal) Interrupts disabled
PC = 0x8

To return after performing the SWI operation, use:

```
MOVS PC, R14
```

This restores the PC (from R14_svc) and CPSR (from SPSR_svc) and returns to the instruction following the SWI.

2.5.4 Prefetch Abort (Instruction Fetch Memory Abort)

A memory abort is signalled by the memory system. Activating an abort in response to an instruction fetch marks the fetched instruction as invalid. An abort will take place if the processor attempts 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 will occur.

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[5:0] = 0b010111 ; Abort mode
CPSR[6] = unchanged ; Fast Interrupt status is unchanged
CPSR[7] = 1 ; (Normal) Interrupts disabled
PC = 0xc

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.

2.5.5 Data Abort (Data Access Memory Abort)

A memory abort is signalled 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 will occur before any following instructions or exceptions have altered the state of the CPU, and the following actions are performed:

R14_abt = address of the aborted instruction + 8
SPSR_abt = CPSR
CPSR[5:0] = 0b010111 ; Abort mode
CPSR[6] = unchanged ; Fast Interrupt status is unchanged
CPSR[7] = 1 ; (Normal) Interrupts disabled
PC = 0x10
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
```

The final value left in the base register used in memory access instructions which specify writeback and generate a data abort (LDR, LDRH, LDRSH, LDRB, LDRSB, STR, STRH, STRB, LDM, STM, LDC, STC) is IMPLEMENTATION DEFINED.

An implementation can choose to leave either the original value or the updated value in the base register, but the same behaviour must be implemented for all memory access instructions.

### 2.5.6 IRQ (Interrupt Request) exception

The IRQ (Interrupt ReQuest) exception is externally generated by asserting the processor’s IRQ input. It has a lower priority than FIQ (see below), and is masked out when a FIQ sequence is entered. Interrupts are disabled when the I bit in the CPSR is set (but note that the I bit can only be altered from a privileged mode). If the I flag is clear, ARM checks for an IRQ at instruction boundaries.

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

- R14_irq = address of next instruction to be executed + 4
- SPSR_irq = CPSR
- CPSR[5:0] = 0b010010 ; Interrupt mode
- CPSR[6] = unchanged ; Fast Interrupt status is unchanged
- CPSR[7] = 1 ; (Normal) Interrupts disabled
- PC = 0x18

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.

### 2.5.7 FIQ (Fast Interrupt Request) exception

The FIQ (Fast Interrupt reQuest) exception is externally generated by asserting the processor’s FIQ input. FIQ is designed to support a data transfer or channel process, and has sufficient private registers to remove the need for register saving in such applications (thus minimising the overhead of context switching).

Fast interrupts are disabled when the F bit in the CPSR is set (but note that the F bit can only be altered from a privileged mode). If the F flag is clear, ARM checks for a FIQ at instruction boundaries.
When a FIQ is detected, the following actions are performed:

\[
\begin{align*}
\text{R14\_fiq} &= \text{address of next instruction to be executed} + 4 \\
\text{SPSR\_fiq} &= \text{CPSR} \\
\text{CPSR}[5:0] &= 0b010001 \quad ; \text{FIQ mode} \\
\text{CPSR}[6] &= \text{unchanged} \quad ; \text{Fast Interrupt disabled} \\
\text{CPSR}[7] &= 1 \quad ; \text{Interrupts disabled} \\
\text{PC} &= 0x1c
\end{align*}
\]

To return after servicing the interrupt, use:

\[
\text{SUBS PC, R14,}\#4
\]

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 0x1c, and not require a branch instruction from the vector.

### 2.5.8 Exception priorities

The Reset exception has the highest priority. FIQ has higher priority than IRQ. IRQ has higher priority than prefetch abort.

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 data abort is higher than FIQ and lower priority than Reset, which ensures that the data-abort handler is entered before the FIQ handler is entered (so that the data abort will be resolved after the FIQ handler has completed).

<table>
<thead>
<tr>
<th>Exception</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>1 (Highest)</td>
</tr>
<tr>
<td>Data Abort</td>
<td>2</td>
</tr>
<tr>
<td>FIQ</td>
<td>3</td>
</tr>
<tr>
<td>IRQ</td>
<td>4</td>
</tr>
<tr>
<td>Prefetch Abort</td>
<td>5</td>
</tr>
<tr>
<td>Undefined Instruction, SWI</td>
<td>6 (Lowest)</td>
</tr>
</tbody>
</table>

*Table 2-4: Exception priorities*
The ARM Instruction Set
This chapter describes the ARM instruction set.

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Using this Chapter</td>
<td>3-2</td>
</tr>
<tr>
<td>3.2</td>
<td>Instruction Set Overview</td>
<td>3-3</td>
</tr>
<tr>
<td>3.3</td>
<td>The Condition Field</td>
<td>3-4</td>
</tr>
<tr>
<td>3.4</td>
<td>Branch Instructions</td>
<td>3-6</td>
</tr>
<tr>
<td>3.5</td>
<td>Data Processing</td>
<td>3-7</td>
</tr>
<tr>
<td>3.6</td>
<td>Multiply Instructions</td>
<td>3-10</td>
</tr>
<tr>
<td>3.7</td>
<td>Status Register Access</td>
<td>3-12</td>
</tr>
<tr>
<td>3.8</td>
<td>Load and Store Instructions</td>
<td>3-13</td>
</tr>
<tr>
<td>3.9</td>
<td>Load and Store Word or Unsigned Byte Instructions</td>
<td>3-15</td>
</tr>
<tr>
<td>3.10</td>
<td>Load and Store Halfword and Load Signed Byte Instructions</td>
<td>3-16</td>
</tr>
<tr>
<td>3.11</td>
<td>Load and Store Multiple Instructions</td>
<td>3-17</td>
</tr>
<tr>
<td>3.12</td>
<td>Semaphore Instructions</td>
<td>3-19</td>
</tr>
<tr>
<td>3.13</td>
<td>Coprocessor Instructions</td>
<td>3-20</td>
</tr>
<tr>
<td>3.14</td>
<td>Extending the Instruction Set</td>
<td>3-22</td>
</tr>
<tr>
<td>3.15</td>
<td>Alphabetical List of ARM Instructions</td>
<td>3-30</td>
</tr>
<tr>
<td>3.16</td>
<td>Data-processing Operands</td>
<td>3-84</td>
</tr>
<tr>
<td>3.17</td>
<td>Load and Store Word or Unsigned Byte Addressing Modes</td>
<td>3-98</td>
</tr>
<tr>
<td>3.18</td>
<td>Load and Store Halfword or Load Signed Byte Addressing Modes</td>
<td>3-109</td>
</tr>
<tr>
<td>3.19</td>
<td>Load and Store Multiple Addressing Modes</td>
<td>3-116</td>
</tr>
<tr>
<td>3.20</td>
<td>Load and Store Multiple Addressing Modes (Alternative names)</td>
<td>3-121</td>
</tr>
<tr>
<td>3.21</td>
<td>Load and Store Coprocessor Addressing Modes</td>
<td>3-123</td>
</tr>
</tbody>
</table>
The ARM Instruction Set

3.1 Using this Chapter

This chapter is divided into three parts:
1 Overview of the ARM instruction types
2 Alphabetical list of instructions
3 Addressing modes

3.1.1 Overview of the ARM instruction types (page 3-3 through 3-27)

This part describes the functional groups within the instruction set, and shows relevant examples and encodings. Each functional group lists all its instructions, which you can then find in the alphabetical section. The functional groups are:
1 Branch
2 Data processing
3 Multiply
4 Status register access
5 Load and store:
   - load and store word or unsigned byte
   - load and store halfword and load signed byte
   - load and store multiple
6 Semaphore
7 Coprocessor

3.1.2 Alphabetical list of instructions (page 3-30 through 3-81)

This part lists every ARM instruction, and gives:
• instruction syntax and functional group
• encoding and operation
• relevant exceptions and qualifiers
• notes on usage
• restrictions on availability in versions of the ARM architecture
• a cross-reference to the relevant addressing modes

3.1.3 Addressing modes (page 3-84 through 3-126)

This part lists the addressing modes for the functional groups of instructions:
Mode 1 Shifter operands for data processing instructions
Mode 2 Load and store word or unsigned byte addressing modes
Mode 3 Load and store halfword or load signed byte addressing modes
Mode 4 Load and store multiple addressing modes
Mode 5 Load and store coprocessor addressing modes
### 3.2 Instruction Set Overview

*Table 3-1: ARM instruction set overview (expanded)* shows the instruction set encoding. All other bit patterns are UNPREDICTABLE.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cond</th>
<th>Opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Shift Immed</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data processing Immediate</td>
<td>0 0 1</td>
<td>op</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>rotate</td>
<td>immediate</td>
</tr>
<tr>
<td>Data processing Immediate shift</td>
<td>0 0 0</td>
<td>opcode</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>shift 0</td>
<td>Rm</td>
</tr>
<tr>
<td>Data processing register shift</td>
<td>0 0 0</td>
<td>opcode</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>Rs 0</td>
<td>Rm</td>
</tr>
<tr>
<td>Multiply</td>
<td>0 0 0</td>
<td>0 0 0 0 0 0 A</td>
<td>S</td>
<td>Rd</td>
<td>Rn</td>
<td>Rs 1 0 0 1</td>
<td>Rm</td>
</tr>
<tr>
<td>Multiply long</td>
<td>0 0 0</td>
<td>0 0 0 0 1 U</td>
<td>A</td>
<td>S</td>
<td>RdHi RdLo</td>
<td>Rs 1 0 0 1</td>
<td>Rm</td>
</tr>
<tr>
<td>Move from Status register</td>
<td>0 0 0</td>
<td>0 1 0 0 R 0 0 SBO</td>
<td>Rd</td>
<td></td>
<td>SBZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move immediate to Status register</td>
<td>0 0 1</td>
<td>0 1 0 0 R 1 0 Mask</td>
<td>SBO</td>
<td>rotate</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move register to Status register</td>
<td>0 0 0</td>
<td>0 1 0 0 R 1 0 Mask</td>
<td>SBO</td>
<td></td>
<td>SBZ 0 Rm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch/Exchange instruction set</td>
<td>0 0 0</td>
<td>0 1 0 0 0 1 0</td>
<td>SBO</td>
<td>SBO</td>
<td>0 0 0 1</td>
<td>Rm</td>
<td></td>
</tr>
<tr>
<td>Load/Store immediate offset</td>
<td>0 1 0</td>
<td>P U B W L</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td>immediate</td>
<td></td>
</tr>
<tr>
<td>Load/Store register offset</td>
<td>0 1 1</td>
<td>P U B W L</td>
<td>Rn</td>
<td>Rd</td>
<td>shift immed</td>
<td>shift 0 Rm</td>
<td></td>
</tr>
<tr>
<td>Load/Store halfword/signed byte</td>
<td>0 0 0</td>
<td>P U 1 W L</td>
<td>Rn</td>
<td>Rd</td>
<td>Hi Offset 1 S H 1</td>
<td>Lo Offset</td>
<td></td>
</tr>
<tr>
<td>Load/Store halfword/signed byte</td>
<td>0 0 0</td>
<td>P U 0 W L</td>
<td>Rn</td>
<td>Rd</td>
<td>SBZ 1 S H 1</td>
<td>Rm</td>
<td></td>
</tr>
<tr>
<td>Swap/Swap byte</td>
<td>0 0 0</td>
<td>0 1 0 0 B 0 0</td>
<td>Rn</td>
<td>Rd</td>
<td>SBZ 1 0 0 1</td>
<td>Rm</td>
<td></td>
</tr>
<tr>
<td>Load/Store multiple</td>
<td>1 0 0</td>
<td>P U S W L</td>
<td>Rn</td>
<td></td>
<td></td>
<td>Register List</td>
<td></td>
</tr>
<tr>
<td>Coprocessor data processing</td>
<td>1 1 1</td>
<td>0 op1</td>
<td>CRn</td>
<td>CRd</td>
<td>cp_num</td>
<td>op2 0</td>
<td>CRm</td>
</tr>
<tr>
<td>Coprocessor register transfers</td>
<td>1 1 1</td>
<td>op1 L</td>
<td>CRn</td>
<td>Rd</td>
<td>cp_num</td>
<td>op2 1</td>
<td>CRm</td>
</tr>
<tr>
<td>Coprocessor load and store</td>
<td>1 1 0</td>
<td>P U N W L</td>
<td>Rd</td>
<td>CRd</td>
<td>cp_num 8 bit_offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch and Branch with link</td>
<td>1 0 1</td>
<td>L</td>
<td></td>
<td></td>
<td>24 bit_offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software interrupt</td>
<td>1 1 1</td>
<td>swi_number</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Undefined instruction</td>
<td>0 1 1</td>
<td>x x x x x x x x x x x x x x x x x</td>
<td>1 x x x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
All ARM instructions can be conditionally executed, which means that their execution may or may not take place depending on the values of values of the N, Z, C and V flags in the CPSR. Every instruction contains a 4-bit condition code field in bits 31 to 28, as shown in Figure 3-1: Condition code fields.

3.3.1 Condition codes

This field specifies one of 16 conditions as described in Table 3-2: Condition codes on page 3-5. Every instruction mnemonic may be extended with the letters defined in the mnemonic extension field.

If the always (AL) condition is specified, the instruction will be executed irrespective of the value of the condition code flags. Any instruction that uses the never (NV) condition is UNPREDICTABLE. The absence of a condition code on an instruction mnemonic implies the always (AL) condition code.
<table>
<thead>
<tr>
<th>Opcode [31:28]</th>
<th>Mnemonic Extension</th>
<th>Meaning</th>
<th>Status flag state</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>EQ</td>
<td>Equal</td>
<td>Z set</td>
</tr>
<tr>
<td>0001</td>
<td>NE</td>
<td>Not Equal</td>
<td>Z clear</td>
</tr>
<tr>
<td>0010</td>
<td>CS/HS</td>
<td>Carry Set /Unsigned Higher or Same</td>
<td>C set</td>
</tr>
<tr>
<td>0011</td>
<td>CC/LO</td>
<td>Carry Clear /Unsigned Lower</td>
<td>C clear</td>
</tr>
<tr>
<td>0100</td>
<td>MI</td>
<td>Minus / Negative</td>
<td>N set</td>
</tr>
<tr>
<td>0101</td>
<td>PL</td>
<td>Plus /Positive or Zero</td>
<td>N clear</td>
</tr>
<tr>
<td>0110</td>
<td>VS</td>
<td>Overflow</td>
<td>V set</td>
</tr>
<tr>
<td>0111</td>
<td>VC</td>
<td>No Overflow</td>
<td>V clear</td>
</tr>
<tr>
<td>1000</td>
<td>HI</td>
<td>Unsigned Higher</td>
<td>C set and Z clear</td>
</tr>
<tr>
<td>1001</td>
<td>LS</td>
<td>Unsigned Lower or Same</td>
<td>C clear or Z set</td>
</tr>
<tr>
<td>1010</td>
<td>GE</td>
<td>Signed Greater Than or Equal</td>
<td>N set and V set, or N clear and V clear (N = V)</td>
</tr>
<tr>
<td>1011</td>
<td>LT</td>
<td>Signed Less Than</td>
<td>N set and V clear, or N clear and V set (N = V)</td>
</tr>
<tr>
<td>1100</td>
<td>GT</td>
<td>Signed Greater Than</td>
<td>Z set, and either N set and V set, or N clear and V clear (Z = 0, N = V)</td>
</tr>
<tr>
<td>1101</td>
<td>LE</td>
<td>Signed Less Than or Equal</td>
<td>Z set, or N set and V clear, or N clear and V set (Z = 1, N = V)</td>
</tr>
<tr>
<td>1110</td>
<td>AL</td>
<td>Always (unconditional)</td>
<td>-</td>
</tr>
<tr>
<td>1111</td>
<td>NV</td>
<td>Never</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 3-2: Condition codes
The ARM Instruction Set

3.4 Branch Instructions

All ARM processors support a branch instruction that allows a conditional branch forwards or backwards up to 32 Mbytes. As the Program Counter (PC) is one of the general-purpose registers (register 15), a branch or jump can also be generated by writing a value to register 15.

A subroutine call is a variant of the standard branch; as well as allowing a branch forward or backward up to 32 Mbytes, the Branch with Link instruction preserves the address of the instruction after the branch (the return address) in register 14 (the link register or LR).

Lastly, a load instruction provides a way to branch any where in the 4Gbyte address space (known as a long branch). A 32-bit value is loaded directly from memory into the PC, causing a branch. The load instruction may be preceded with a Move instruction to store a return address in the link register (LR or R14)).

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 .

Processors that support the Thumb instruction set (Architecture v4T) also support a branch instruction (BX) that jumps to a given address, and optionally switches executing Thumb instructions.

3.4.1 List of branch instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B, BL</td>
<td>Branch, and branch with link</td>
<td>3-33</td>
</tr>
<tr>
<td>BX</td>
<td>Branch and exchange instruction set</td>
<td>3-35</td>
</tr>
</tbody>
</table>
ARM has 16 data-processing instructions. Most data-processing instructions take two source operands (Move and Move Not have only one operand) and store a result in a register (except for the Compare and Test instructions which only update the condition codes). 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 may have a shift applied to it before it is used as the operand to the ALU.

### Table 3-3: Data-processing instructions

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

\[
\begin{align*}
\text{<opcode1>}{\{<\text{cond}>\}}{\{S\}} \text{ Rd, } & \text{<shifter_operand>} \text{ } \\
\text{<opcode1> := MOV | MVN} \\
\text{<opcode2>}{\{<\text{cond}>\}} \text{ Rn, } & \text{<shifter_operand>} \text{ } \\
\text{<opcode2> := CMP | CMN | TST | TEQ} \\
\text{<opcode3>}{\{<\text{cond}>\}}{\{S\}} \text{ Rd, Rn, } & \text{<shifter_operand>} \text{ } \\
\text{<opcode3> := ADD | SUB | RSB | ADC | SBC | RSC | AND | BIC | EOR | ORR}
\end{align*}
\]

- **Rd** specifies the destination register
- **Rn** specifies the first source operand register
- **<shifter_operand>** specifies the second source operand.

See 3.16 Data-processing Operands on page 3-84 for details of the shifter operands.

### Notes
- **The I bit:** Bit 25 is used to distinguish between the immediate and register forms of **<shifter_operand>**.
- **The S bit:** Bit 20 is used to signify that the instruction updates the condition codes.

### 3.5.2 Condition code flags

Data-processing instructions can update the four condition code flags. CMP, CMN, TST and TEQ always update the condition code flags; the remaining instructions will update the flags if an S is appended to the instruction mnemonic (which sets the S bit in the instruction).

Bits are set as follows:

- **N (Negative) flag**: is set if the result of a data-processing instruction is negative
- **Z (Zero) flag**: is set if the result is equal to zero
- **C (Carry) flag**: is set if an add, subtract or compare causes a carry (result bigger than 32 bits), or is set from the output of the shifter for move and logical instructions
- **V (Overflow) flag**: is set if an Add or Subtract, or compare overflows (signed result bigger than 31 bits); unaffected by move or conditional instructions
### List of data-processing instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Add with Carry</td>
<td>3-30</td>
</tr>
<tr>
<td>ADD</td>
<td>Add</td>
<td>3-31</td>
</tr>
<tr>
<td>AND</td>
<td>Logical AND</td>
<td>3-32</td>
</tr>
<tr>
<td>BIC</td>
<td>Logical Bit Clear</td>
<td>3-34</td>
</tr>
<tr>
<td>CMN</td>
<td>Compare Negative</td>
<td>3-37</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare</td>
<td>3-38</td>
</tr>
<tr>
<td>EOR</td>
<td>Logical EOR</td>
<td>3-39</td>
</tr>
<tr>
<td>MOV</td>
<td>Move</td>
<td>3-53</td>
</tr>
<tr>
<td>MVN</td>
<td>Move negative</td>
<td>3-59</td>
</tr>
<tr>
<td>ORR</td>
<td>Logical OR</td>
<td>3-60</td>
</tr>
<tr>
<td>RSB</td>
<td>Reverse Subtract</td>
<td>3-61</td>
</tr>
<tr>
<td>RSC</td>
<td>Reverse Subtract with Carry</td>
<td>3-62</td>
</tr>
<tr>
<td>SBC</td>
<td>Subtract with Carry</td>
<td>3-63</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
<td>3-74</td>
</tr>
<tr>
<td>TEQ</td>
<td>Test Equivalence</td>
<td>3-78</td>
</tr>
<tr>
<td>TST</td>
<td>Test</td>
<td>3-79</td>
</tr>
</tbody>
</table>
The ARM Instruction Set

3.6 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. ARM does not directly support a multiply-by-constant instruction due to the efficiency of shift and add, or shift and reverse subtract instructions.

3.6.1 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 into a fourth register (i.e. performs multiply and accumulate).

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.

```
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
```

3.6.2 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 a third and fourth register. There are a 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 and third and fourth register and store the 64-bit result back into those registers (third and fourth). There are also signed (SMLAL) and unsigned (UMLAL) variants. These instruction perform a long multiply and accumulate.

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

```
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 ; R6, R8 = R0 x R1
UMLAL R5, R8, R0, R1 ; R5, R8 = R0 x R1 + R5, R8
```
### 3.6.3 List of multiply instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLA</td>
<td>Multiply accumulate</td>
<td>3-52</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply</td>
<td>3-58</td>
</tr>
<tr>
<td>SMLAL</td>
<td>Signed multiply accumulate long</td>
<td>3-64</td>
</tr>
<tr>
<td>SMULL</td>
<td>Signed multiply long</td>
<td>3-65</td>
</tr>
<tr>
<td>UMLAL</td>
<td>Unsigned multiply accumulate long</td>
<td>3-80</td>
</tr>
<tr>
<td>UMULL</td>
<td>Unsigned multiply long</td>
<td>3-81</td>
</tr>
</tbody>
</table>
The ARM Instruction Set

3.7 Status Register Access

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 than can be individually written:

- bits 31 to 24: the flags field
- bits 23 to 16: the status field
- bits 15 to 8: the extension field
- bits 7 to 0: the control field

ARMv4 does not use the status and extension field, and 4 bits are unused in the flags field. The four condition code flags occupy the remaining four bits of the flags field, and the control field contains two interrupt disable bits, 5 processor mode bits, and the Thumb bit on ARMv4T (See 2.4 Program Status Registers on page 2-3).

The unused bits of the status registers may be used in future ARM architectures, and should not be modified by software. Therefore, a read-modify-write strategy should 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 writeable to allow the write part of the read-modify-write operation, and allow all processor state to be restored.

3.7.1 CPSR value

Alter the value of the CPSR has three uses:
1. Sets the value of the condition code flags to a known value.
2. Enables or disables interrupts.
3. Changes processor mode (for instance to initialise stack pointers).

3.7.2 Examples

- **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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRS</td>
<td>Move SR to general-purpose register</td>
<td>3-55</td>
</tr>
<tr>
<td>MSR</td>
<td>Move general-purpose register to SR</td>
<td>3-56</td>
</tr>
</tbody>
</table>
3.8 Load and Store Instructions

ARMv4 supports two broad classes of instruction which load or store the value of a single register from or to memory.

- The first form can load or store a 32-bit word or an 8-bit unsigned byte
- The second form can load or store a 16-bit unsigned halfword, and can load and sign extend a 16-bit halfword or an 8-bit byte

The first form (word and unsigned byte) allows a wider range of addressing modes than the second (halfword and signed byte). The Word and Unsigned Byte addressing mode comes in two parts:

- the base register
- the offset

The base register is 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 forms:

1. Immediate Offset:
   The offset is a 12-bit unsigned number, that may 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 IO registers.

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

3. Scaled Register Offset:
   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 size of each array element.

As well as the three forms of offset, the offset and base register are used in three different ways to form the memory address.

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

2. Pre-indexed addressing:
   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.

3. Post-indexed addressing:
   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.
The ARM Instruction Set

3.8.1 Examples

```
LDR R1, [R0]          ; Load register 1 from the address in register 0
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 a 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 the address 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 word from R2 to R5, then R5 = R5 + 8
LDR R0, [PC, #40]     ; Load R0 from PC + 8 + 0x40
LDR R0, [R1], R2      ; Load R0 from R1, then R1 = R1 + R2
```

3.8.2 Examples of halfword and signed byte addressing modes

The Halfword and Signed Byte addressing modes are a subset of the above addressing modes. The scaled register offset is not supported, and the immediate offset contains 8 bits, not 12.

```
LDRH R1, [R0]         ; Load a halfword to R1 from R0 (zero top bytes)
LDRH R8, [R3, #2]     ; Load a halfword into R8 from R3 + 2
LDRH R12, [R13, #-6]  ; Load a halfword 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 R11 from the 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
```
3.9 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}  Rd, <addressing_mode>

The I, P, U and W bits: These bits distinguish between different types of <addressing_mode>.

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

3.9.1 List of load and store word or unsigned byte instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR</td>
<td>Load word</td>
<td>3-44</td>
</tr>
<tr>
<td>LDRB</td>
<td>Load byte</td>
<td>3-45</td>
</tr>
<tr>
<td>LDRBT</td>
<td>Load byte with user mode privilege</td>
<td>3-46</td>
</tr>
<tr>
<td>LDRT</td>
<td>Load word with user mode privilege</td>
<td>3-50</td>
</tr>
<tr>
<td>STR</td>
<td>Store word</td>
<td>3-69</td>
</tr>
<tr>
<td>STRB</td>
<td>Store byte</td>
<td>3-70</td>
</tr>
<tr>
<td>STRBT</td>
<td>Store byte with user mode privilege</td>
<td>3-71</td>
</tr>
<tr>
<td>STRT</td>
<td>Store word with user mode privilege</td>
<td>3-73</td>
</tr>
</tbody>
</table>
The ARM Instruction Set

3.10 Load and Store Halfword and Load Signed 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 halfword and load signed byte instructions have a single instruction format:

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

- **cond**: These bits are addressing mode specific.
- **I, P, U and W bits**: These bits specify the type of `<addressing_mode>` (see section 3.18 Load and Store Halfword or Load Signed Byte Addressing Modes on page 3-109).
- **L bit**: This bit distinguishes between a Load (L==1) and a Store instruction (L==0).
- **S bit**: This bit distinguishes between a signed (S==1) and an unsigned (S==0) halfword access.
- **H bit**: This bit distinguishes between a halfword (H==1) and a signed byte (H==0) access.

3.10.1 List of load and store halfword and load signed byte instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDRH</td>
<td>Load unsigned halfword</td>
<td>3-47</td>
</tr>
<tr>
<td>LDRSB</td>
<td>Load signed byte</td>
<td>3-48</td>
</tr>
<tr>
<td>LDRSH</td>
<td>Load signed halfword</td>
<td>3-49</td>
</tr>
<tr>
<td>STRH</td>
<td>Store halfword</td>
<td>3-72</td>
</tr>
</tbody>
</table>
3.11 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{!}, <register_list>{^}
STM{<cond>}<addressing_mode>  Rn{!}, <registers>{^}

where:

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

The register list: The <register_list> has one bit for each general-purpose register; bit 0 is for register zero, and bit 15 is for register 15 (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 may be specified by separating the first and last registers in the range with a minus sign.

The P, U and W bits: These bits distinguish between the different types of addressing mode. See 3.19 Load and Store Multiple Addressing Modes on page 3-116.

The S bit: For LDMs that load the PC, the S bit indicates that the CPSR is loaded from the SPSR. For LDMs that do not load the PC and all STMs, 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.

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

Addressing modes

For full details of the addressing modes for these instructions, refer to 3.19 Load and Store Multiple Addressing Modes on page 3-116, and 3.20 Load and Store Multiple Addressing Modes (Alternative names) on page 3-121.

3.11.1 Examples

STMFD  R13!, {R0 - R12, LR}
LDMFD  R13!, {R0 - R12, PC}
LD Mia  R0,  {R5 - R8}
STMD A  R1!,  {R2, R5, R7 - R9, R11}
### List of load and store multiple instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDM</td>
<td>Load multiple</td>
<td>3-41</td>
</tr>
<tr>
<td>LDM</td>
<td>User registers load multiple</td>
<td>3-42</td>
</tr>
<tr>
<td>LDM</td>
<td>Load multiple and restore CSPR</td>
<td>3-43</td>
</tr>
<tr>
<td>STM</td>
<td>Store multiple</td>
<td>3-67</td>
</tr>
<tr>
<td>STM</td>
<td>User registers store multiple</td>
<td>3-68</td>
</tr>
</tbody>
</table>
3.12 Semaphore Instructions

The ARM instruction set has two semaphore instructions:

- Swap (SWP)
- Swap Byte (SWPB)

These instructions are provided for process synchronisation. 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; the 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 SWP exchanges the value in the register and the value in memory.

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

Examples

```assembly
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
```

3.12.1 List of semaphore instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWP</td>
<td>Swap</td>
<td>3-76</td>
</tr>
<tr>
<td>SWPB</td>
<td>Swap Byte</td>
<td>3-77</td>
</tr>
</tbody>
</table>
3.13 Coprocessor Instructions

**Note:** Coprocessor instructions are not implemented in Architecture version 1.

The ARM instruction set provides 3 types of instruction for communicating with coprocessors. 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 (one coprocessor can use more than one of the 16 numbers if a large coprocessor instruction set is required).

The three classes of coprocessor instruction allow:

- ARM to initiate a coprocessor data processing operation
- ARM registers to be transferred to and from coprocessor registers
- ARM to generate addresses for the coprocessor load and store instructions

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).

Not all fields in coprocessor instructions are used by ARM; 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).

**Examples**

- **CDP** p5, 2, c12, c10, c3, 4; Coprocessor 5 data operation
  
  \- opcode 1 = 2, opcode 2 = 4
  \- destination register 12
  \- source registers are 10 and 3

- **MRC** p15, 5, R4, c0, c2, 3; Coprocessor 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 Coprocessor 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
The ARM Instruction Set

### 3.13.1 List of coprocessor instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDP</td>
<td>Coprocessor data operations</td>
<td>3-36</td>
</tr>
<tr>
<td>LDC</td>
<td>Load coprocessor register</td>
<td>3-40</td>
</tr>
<tr>
<td>MCR</td>
<td>Move to coprocessor from ARM register</td>
<td>3-51</td>
</tr>
<tr>
<td>MRC</td>
<td>Move to ARM register from coprocessor</td>
<td>3-54</td>
</tr>
<tr>
<td>STC</td>
<td>Store coprocessor register</td>
<td>3-66</td>
</tr>
</tbody>
</table>
The ARM Instruction Set

3.14 Extending the Instruction Set

The ARM instruction set can be extended in four areas:
- Arithmetic instruction extension space
- Control instruction extension space
- Load/store instruction extension space
- Coprocessor instruction extension space

Currently, these instructions are UNDEFINED (they cause an undefined instruction exception). These parts of the address space will be used in the future to add new instructions.

3.14.1 Arithmetic instruction extension space

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

\[
\text{opcode}[27:24] = 0 \\
\text{opcode}[7:4] = 0b1001
\]

The field names given are only guidelines, which are likely to simplify implementation.

\[
\begin{array}{ccccccccccc}
31 & 28 & 27 & 26 & 25 & 24 & 23 & 20 & 19 & 16 & 15 & 12 & 11 & 8 & 7 & 6 & 5 & 4 & 3 & 0 \\
\hline
\text{cond} & 0 & 0 & 0 & 0 & \text{op1} & \text{Rn} & \text{Rd} & \text{Rs} & 1 & 0 & 0 & 1 & \text{Rm} \\
\end{array}
\]

MUL and MLA

Multiply and Multiply Accumulate (MUL and MLA) instructions use op1 values 0 to 3.
- \text{Rn} specifies the destination register
- \text{Rm and Rs} specify source registers
- \text{Rd} specifies the accumulated value

UMULL, UMLAL, SMULL, SMLAL

The Signed and Unsigned Multiple Long and Multiply Accumulate Long (UMULL, UMLAL, SMULL, SMLAL) instructions use op1 values 8 to 15;
- \text{Rn and Rd} specify the destination registers
- \text{Rm and Rs} specify the source registers

Other opcodes

The meaning of all other opcodes in the arithmetic instruction extension space is:
- UNDEFINED on ARM Architecture 4
- UNPREDICTABLE on earlier versions of the architecture.
3.14.2 Control instruction extension space

Instructions with:

\[
\begin{align*}
\text{opcode}[27:26] & = \text{0b00} \\
\text{opcode}[24:23] & = \text{0b10} \\
\text{opcode}[20] & = 0
\end{align*}
\]

and not:

\[
\begin{align*}
\text{opcode}[25] & = 0 \\
\text{opcode}[7] & = 1 \\
\text{opcode}[4] & = 1
\end{align*}
\]

are the control instruction extension space. The field names given are only guidelines, which are likely to simplify implementation.

\[
\begin{align*}
\text{cond} & \quad 0 \quad 0 \quad 1 \quad 0 \quad \text{op1} \quad 0 \quad \text{Rn} \quad \text{Rd} \quad \text{Rs} \quad \text{op2} \quad 0 \quad \text{Rm} \\
\text{cond} & \quad 0 \quad 0 \quad 1 \quad 0 \quad \text{op1} \quad 0 \quad \text{Rn} \quad \text{Rd} \quad \text{Rs} \quad 0 \quad \text{op2} \quad 1 \quad \text{Rm} \\
\text{cond} & \quad 0 \quad 0 \quad 1 \quad 0 \quad \text{op1} \quad 0 \quad \text{Rn} \quad \text{Rd} \quad \text{rotate_imm} \quad 8\text{-bit Immediate}
\end{align*}
\]

MRS

The Move status register to general-purpose register (MRS) instruction sets:

\[
\begin{align*}
\text{opcode}[25] & = 0 \\
\text{opcode}[19:16] & = \text{0b1111} \\
\text{opcode}[11:0] & = 0
\end{align*}
\]

and uses both \text{op1} = \text{0b00} and \text{op1} = \text{0b10}.

\text{Rd} is used to specify the destination register.

MSR

The Move general-purpose register to status register (MSR) instruction sets:

\[
\begin{align*}
\text{opcode}[25] & = 0 \\
\text{opcode}[15:12] & = \text{0b1111} \\
\text{opcode}[11:4] & = 0
\end{align*}
\]

and uses both \text{op1} = \text{0b01} and \text{op1} = \text{0b11}.

\text{Rm} specifies the source register

\text{opcode}[19:16] holds the write mask
The ARM Instruction Set

The Move immediate value to status register (MSR) instruction sets:

\[
\text{opcode}[25] = 1 \\
\text{opcode}[15:12] = 0b1111
\]

and uses both \( \text{op1} = 0b01 \) and \( \text{op1} = 0b11 \).

\( \text{Rm} \) specifies the source register

\( \text{opcode}[11:0] \) specify the immediate operand

\( \text{opcode}[19:16] \) hold the write mask

**BX**

The Branch and Exchange Instruction Set (BX) instruction sets:

\[
\begin{align*}
\text{opcode}[25] & = 0 \\
\text{opcode}[19:8] & = 0b111111111111 \\
\text{opcode}[7:4] & = 0b0001
\end{align*}
\]

and uses \( \text{op1} = 0b01 \).

\( \text{Rm} \) is used to specify the source register.

**Other opcodes**

The meaning of all other opcodes in the control instruction extension space is:

- UNDEFINED on ARM Architecture 4
- UNPREDICTABLE on earlier versions of the architecture

### 3.14.3 Load/Store instruction extension space

Instructions with

\[
\begin{align*}
\text{opcode}[27:25] & = 0b000 \\
\text{opcode}[7] & = 1 \\
\text{opcode}[4] & = 1
\end{align*}
\]

and not

\[
\begin{align*}
\text{opcode}[24] & = 0 \\
\text{opcode}[6:5] & = 0
\end{align*}
\]

are the load/store instruction extension space.

The field names given are only guidelines, which are likely to simplify implementation.
The ARM Instruction Set

SWP and SWPB

The Swap and Swap Byte (SWP and SWPB) instructions set:

\[
\text{opcode}[24:23] = 0b10 \\
\text{opcode}[21:20] = 0b00 \\
\text{opcode}[11:4] = 0b00001001
\]

where:
- \( \text{Rn} \) specifies the base address
- \( \text{Rd} \) specifies the destination register
- \( \text{Rm} \) specifies the source register
- \( \text{Opcode}[22] \) indicates a byte transfer

LDRH

The Load Halfword (LDRH) instruction sets:

\[
\text{opcode}[20] = 1 \\
\text{op1} = 0b01
\]

where:
- the B bit distinguishes between a register and an immediate offset
- the P, U and W bits specify the addressing mode
- \( \text{Rn} \) specifies the base register
- \( \text{Rd} \) specifies the destination register
- \( \text{Rm} \) specifies a register offset
- \( \text{Rs} \) and \( \text{Rm} \) specify an immediate offset

LDRSH

The Load Signed Halfword (LDRSH) instruction sets:

\[
\text{opcode}[20] = 1 \\
\text{op1} = 0b11
\]

where:
- the B bit distinguishes between a register and an immediate offset
- the P, U and W bits specify the addressing mode
- \( \text{Rn} \) specifies the base register
- \( \text{Rd} \) specifies the destination register
- \( \text{Rm} \) specifies a register offset
- \( \text{Rs} \) and \( \text{Rm} \) specify an immediate offset
The ARM Instruction Set

LDRSB
The Load Signed Byte (LDRSB) instruction sets:

\[
\begin{align*}
\text{opcode}[20] &= 1 \\
op1 &= 0b10
\end{align*}
\]

where:
- the B bit distinguishes between a register and an immediate offset
- the P, U and W bits specify the addressing mode
- Rn specifies the base register
- Rd specifies the destination register
- Rm specifies a register offset
- Rs and Rm specify an immediate offset

STRH
The Store Halfword (STRH) instruction sets:

\[
\begin{align*}
\text{opcode}[20] &= 0 \\
op1 &= 0b01
\end{align*}
\]

where:
- the B bit distinguishes between a register and an immediate offset
- the P, U and W bits specify the addressing mode
- Rn specifies the base register
- Rd specifies the source register
- Rm specifies a register offset
- Rs and Rm specify an immediate offset

Other opcodes
The meaning of all other opcodes in the Load/Store instruction extension space is:
- UNDEFINED on ARM Architecture 4
- UNPREDICTABLE on earlier versions of the architecture
3.14.4 Coprocessor instruction extension space

Instructions with

\[ \text{opcode}[27:24] = 0b1100 \]
\[ \text{opcode}[21] = 0 \]

are the coprocessor instruction extension space. The names given to fields are only guidelines, which if followed are likely to simplify implementation.

The meaning of instructions in the coprocessor instruction extension space is:

- UNDEFINED on ARM Architecture 4
- UNPREDICTABLE on earlier versions of the architecture

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

3.14.5 Undefined instruction Space

Instructions with

\[ \text{opcode}[27:25] = 0b011 \]
\[ \text{opcode}[4] = 1 \]

are UNDEFINED instruction space.

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

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
ARM Instructions

**Description**
Combined with a suitable addressing mode, the STRH (Store Register Half) instruction stores 16-bit data from a general-purpose register to be stored to memory. Using a suitable addressing mode allows PC-relative addressing, to facilitate position-independent code. 

**Operation**

```
if ConditionPassed(<cond>) then
    if <address>[0] == 0
        <data> = Rd[15:0]
    else /* <address>[0] == 1 */
        <data> = UNPREDICTABLE
    Memory[<address>,2] = <data>
```

**Syntax**

```
STR{<cond>}H Rd, <addressing_mode>
```

**Description**

Combined with a suitable addressing mode, the STRH instruction stores a halfword from the least-significant halfword of register Rd to the address calculated by `<addressing_mode>`. If the address is not halfword-aligned, the result is UNPREDICTABLE.

The instruction is only executed if the condition specified in the instruction makes it ConditionPassed() status.

**Exceptions**
- Data Abort

**Qualifier**
- Condition Code

**Notes**
- Addressing modes: The I, P, U and W bits specify the type of addressing used.

- Mode 3 starting on page -108.

- The addr_mode bits: These bits are addressing mode specific.

- The register Rn specifies the base register used by `<addressing_mode>`.

- Use of Rd: If register 15 is specified for Rd, the result is UNPREDICTABLE.

- Operands restrictions: If `<addressing_mode>` uses pre-indexed or post-indexed addressing, the same register is specified for Rd and Rn, the result is UNPREDICTABLE.

- Data Abort: If a data abort is signalled and `<addressing_mode>` uses pre-indexed or post-indexed addressing, the value left in Rn is IMPLEMENTATION DEFINED, but is either the UNPREDICTABLE value or the updated base register value (even if the same register is used).

- Non-half-word aligned addresses: If the store address is not halfword-aligned, the result is UNPREDICTABLE.

- Alignment: If an implementation includes a System Control Coprocessor (see page -108), checking is enabled, an address with bit[0] != 0 will cause an alignment exception.

- For more details, see the ARM Architecture Reference Manual.
ADC{<cond>}{S}  Rd, Rn, <shifter_operand>

Description
The ADC (Add with Carry) instruction adds the value of <shifter_operand> and the value of the Carry flag 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).

ADC is used to synthesize multi-word addition. If register pairs R0,R1 and R2,R3 hold 64-bit values (where 0 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

The instruction:

- ADCS R0,R0,R0

produces a single-bit Rotate Left with Extend operation (33-bit rotate though the carry flag) on R0. See 3-97 for more information.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

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)

Exceptions
None

Qualifiers
Condition Code
S updates condition code flags N,Z,C,V

Notes
Shifter operand: The shifter operands for this instruction are given in Addressing Mode 1 starting on page 3-84.

The I bit: Bit 25 is used to distinguish between the immediate and register forms of <shifter_operand>.

Writing to R15: When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.
ADD{<cond>}{S}  Rd, Rn, <shifter_operand>

Description
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).
ADD is used to add two values together to produce a third.
To increment a register value (in Rx), use:

\[
\text{ADD Rx, Rx, } \#1
\]

Constant multiplication (of Rx) by \(2^{n+1}\) (into Rd) can be performed with:

\[
\text{ADD Rd, Rx, Rx LSL } \#n
\]

To form a PC-relative address, use:

\[
\text{ADD Rs, PC, } \#\text{offset}
\]

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

Operation
if ConditionPassed(<cond>) then
\[
\text{Rd} = \text{Rn} + \text{<shifter_operand>}
\]

if S == 1 and Rd == R15 then
\[
\text{CPSR} = \text{SPSR}
\]
else if S == 1 then
\[
\text{N Flag} = \text{Rd}[31]
\]
\[
\text{Z Flag} = \text{if Rd} == 0 \text{ then 1 else 0}
\]
\[
\text{C Flag} = \text{CarryFrom(Rn + <shifter_operand>)}
\]
\[
\text{V Flag} = \text{OverflowFrom(Rn + <shifter_operand>)}
\]

Exceptions
None

Qualifiers
Condition Code
S updates condition code flags N,Z,C,V

Notes
Shifter operand: The shifter operands for this instruction are given in Addressing Mode 1 starting on page 3-84.
The I bit: Bit 25 is used to distinguish between the immediate and register forms of <shifter_operand>.
Writing to R15: When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.
AND<cond>{<S>} Rd, Rn, <shifter_operand>

**Description**
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).

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

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

<table>
<thead>
<tr>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ConditionPassed(&lt;cond&gt;) then</td>
</tr>
<tr>
<td>Rd = Rn AND &lt;shifter_operand&gt;</td>
</tr>
<tr>
<td>if S == 1 and Rd == R15 then</td>
</tr>
<tr>
<td>CPSR = SPSR</td>
</tr>
<tr>
<td>else if S == 1 then</td>
</tr>
<tr>
<td>N Flag = Rd[31]</td>
</tr>
<tr>
<td>Z Flag = if Rd == 0 then 1 else 0</td>
</tr>
<tr>
<td>C Flag = &lt;shifter_carry_out&gt;</td>
</tr>
<tr>
<td>V Flag = unaffected</td>
</tr>
</tbody>
</table>

**Exceptions**
None

**Qualifiers**
Condition Code
S updates condition code flags N,Z,C

**Notes**
Shifter operand: The shifter operands for this instruction are given in Addressing Mode 1 starting on page 3-84.

The I bit: Bit 25 is used to distinguish between the immediate and register forms of <shifter_operand>.

Writing to R15: When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.
The B (Branch) and BL (Branch and Link) instructions provide both conditional and unconditional changes to program flow. The Branch with Link instruction is used to perform a subroutine call; the return from subroutine is achieved by copying the LR to the PC.

B and BL cause a branch to a target address. The branch target address is calculated by:
1. shifting the 24-bit signed (two’s complement) offset left two bits
2. sign-extending the result to 32 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 +/- 32Mbytes.

In the BL variant of the instruction, the L (link) bit is set, and the address of the instruction following the branch is copied into the link register (R14).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

```
if ConditionPassed(<cond>) then
  if L == 1 then
    LR = address of the instruction after the branch instruction
    PC = PC + (SignExtend(<24_bit_signed_offset>) << 2)
```

Exceptions
None

Qualifiers
Condition Code
L (Link) stores a return address in the LR (R14) register

Notes
Offset calculation: An assembler will calculate the branch offset address from the difference between the address of the current instruction and the address of the target (given as a program label) minus eight (because the PC holds the address of the current instruction plus eight).

Memory bounds: Branching backwards past location zero and forwards over the end of the 32-bit address space is UNPREDICTABLE.
BIC<cond>{<S>} Rd, Rn, <shifter_operand>

Description
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).

BIC can be used to clear selected bits in a register; for each bit, BIC with 1 will clear the bit, BIC with 0 will leave it unchanged.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

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

Exceptions
None

Qualifiers
Condition Code
S updates condition code flags N,Z,C

Notes
Shifter operand: The shifter operands for this instruction are given in Addressing Mode 1 starting on page 3-84.

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

The I bit: Bit 25 is used to distinguish between the immediate and register forms of <shifter_operand>.
The BX (Branch and Exchange instructions set) is UNDEFINED on ARM Architecture Version 4. On ARM Architecture Version 4T, this instruction branches and selects the instruction set decoder to use to decode the instructions at the branch destination. The branch target address is the value of register Rm. The T flag is updated with bit 0 of the value of register Rm.

BX is used to branch between ARM code and THUMB code. On ARM Architecture 4, it causes an UNDEFINED instruction exception to allow the THUMB instruction set to be emulated.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Condition Passed(&lt;cond&gt;) then</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T Flag = Rm[0]</td>
</tr>
<tr>
<td></td>
<td>PC = Rm[31:1] &lt;&lt; 1</td>
</tr>
</tbody>
</table>

Exceptions

None

Operation

Condition Code

Notes

Transferring to THUMB: When transferring to the THUMB instruction set, bit[0] of PC will be cleared (set to zero), and bits[31:1] will be copied from Rm to the PC.

Transferring to ARM: When transferring to the ARM instruction set, bit[0] of PC will be cleared (set to zero), and bits[31:1] will be copied from Rm to the PC. If bit[1] of Rm is set, the result is UNPREDICTABLE.
CDP\{<cond>\} p<cp#>, <opcode_1>, CRd, CRn, CRm, <opcode_2>

Description
The CDP (Coprocessor Data Processing) instruction tells the coprocessor specified by \(<cp#>\) 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.

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

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

Operation
\[
\text{if } \text{ConditionPassed}(\text{<cond>}) \text{ then} \\
\quad \text{Coprocessor}[\text{<cp_num>}] \text{ dependent operation}
\]

Exceptions
Undefined Instruction

Qualifiers
Condition Code

Notes
Coprocessor fields: Only instruction bits[31:24], bits[11:8] and bit[4] are which architecture defined; the remaining fields are only recommendations, which if followed, will be compatible with ARM Development Systems.
CMN(<cond>)  Rn,  <shifter_operand>

**Description**  
The CMN (Compare Negative) instruction compares an arithmetic value and the negative of an arithmetic value (an immediate or the value of a register) and sets the condition code flags so that subsequent instructions can be conditionally executed.

CMN performs a comparison by adding (or subtracting the negative of) the value of <shifter_operand> to (from) the value of register Rn, and updates the condition code flags (based on the result). The comparison is the subtraction of the negative of the second operand from the first operand (which is the same as adding the two operands).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

**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>)

**Exceptions**  
None

**Qualifiers**  
Condition Code

**Notes**  
Shifter operand: The shifter operands for this instruction are given in Addressing Mode 1 starting on page 3-84.

The I bit: Bit 25 is used to distinguish between the immediate and register forms of <shifter_operand>. 

31 28 27 26 25 24 23 22 21 20 19 16 15 12 11 0
cond 0 0 1 1 1 1 0 0 Rn SBZ shifter_operand
The CMP (Compare) instruction compares two arithmetic values and sets the condition code flags so that subsequent instructions can be conditionally executed. The comparison is a subtraction of the second operand from the first operand.

CMP performs a comparison by subtracting the value of `<shifter_operand>` from the value of register Rn and updates the condition code flags (based on the result).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

If `ConditionPassed(<cond>)` then

\[
\text{N Flag} = \text{alu_out}[31]
\]

\[
\text{Z Flag} = \text{if } \text{alu_out} == 0 \text{ then } 1 \text{ else } 0
\]

\[
\text{C Flag} = \text{NOT BorrowFrom(Rn - <shifter_operand>)}
\]

\[
\text{V Flag} = \text{OverflowFrom (Rn - <shifter_operand>)}
\]

**Shifter operand:** The shifter operands for this instruction are given in Addressing Mode 1 starting on page 3-84.

**The I bit:** Bit 25 is used to distinguish between the immediate and register forms of `<shifter_operand>`. 
**EOR**(<cond>){S} Rd, Rn, <shifter_operand>

**Description**
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).

EOR can be used to invert selected bits in a register; for each bit, EOR with 1 will invert that bit, and EOR with 0 will leave it unchanged.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

**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

**Exceptions**
None

**Qualifiers**
Condition Code
S updates condition code flags N.Z.C

**Notes**
Shifter operand: The shifter operands for this instruction are given in Addressing Mode 1 starting on page 3-84.

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

The I bit: Bit 25 is used to distinguish between the immediate and register forms of <shifter_operand>. 

---

**ARM Architecture Reference Manual**
ARM DUI 0100B
LDC{<cond>}  p<cp_num>, CRd, <addressing_mode>

Description
The LDC (Load Coprocessor) instruction is useful to load coprocessor data from memory. The N bit could be used to distinguish between a single- and double-precision transfer for a floating-point load instruction.

LDC loads memory data from the sequence of consecutive memory addresses calculated by \(<addressing_mode>\) to the coprocessor specified by \(<cp_num>\). If no coprocessors indicate that they can execute the instruction, an UNDEFINED instruction exception is generated.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

Operation
if ConditionPassed\(<cond>\) then
  \(<address> = <start_address>\n  while (NotFinished\(coprocessor[<cp_num>]\))
    \(coprocessor[<cp_num>] = Memory[<address>,4]\n    <address> = <address> + 4
  assert <address> == <end_address>

Exceptions
Undefined Instruction; Data Abort

Qualifiers
Condition Code

Notes
Addressing mode: The P, U and W bits specify the \(<addressing_mode>\). See Addressing Mode 5 starting on page 3-123.

The N bit: This bit is coprocessor-dependent. It can be used to distinguish between two sizes of data to transfer.

Register Rn: Specifies the base register used by \(<addressing_mode>\).

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.

Data Abort: If a data abort is signalled and \(<addressing_mode>\) uses pre-indexed or post-indexed addressing, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value.

Non-word-aligned addresses: Load coprocessor register 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 7), and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.
LDM(<cond>)<addressing_mode>  Rn(!), <registers>

Description
This form of the LDM (Load Multiple) instruction is useful as a block load instruction (combined with store multiple it allows efficient block copy) and for stack operations, including procedure exit, to restore saved registers, load the PC with the return address, and update the stack pointer.

In this case, LDM loads a non-empty subset (or possibly all) of the general-purpose registers from sequential memory locations. The registers are loaded in sequence, the lowest-numbered register first, from the lowest memory address (<start_addr>); the highest-numbered register last, from the highest memory address (<end_addr>). 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.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

Operation
if ConditionPassed(<cond>) then
  <address> = <start_addr>
  for i = 0 to 15
    if <register_list>[i] == 1
      Ri = Memory[<address>,4]
      <address> = <address> + 4
  assert <end_add> == <address> - 4

Exceptions
Data Abort

Qualifiers
Condition Code
! sets the W bit, causing base register update

Notes
Addressing mode: The P, U and W bits distinguish between the different types of addressing mode. See Addressing Mode 4 starting on page 3-116.

Register Rn: Specifies the base register used by <addressing_mode>.
Use of R15: Using R15 as the base register Rn gives an UNPREDICTABLE result.

Operand restrictions: If the base register Rn is specified in <register_list>, and writeback is specified, the final value of Rn is UNPREDICTABLE.

Data Abort: If a data abort is signalled and <addressing_mode> specifies writeback, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if Rn is specified in <register_list>). If register 15 is specified in <register_list>, it must not be overwritten if a data abort occurs.

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 7), and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.
This form of the LDM (Load Multiple) instruction loads user mode registers when the processor is in a privileged mode (useful when performing process swaps).

In this case, LDM instruction loads a non-empty subset (or possibly all except the PC) of the user mode general-purpose registers (which are also the system mode general-purpose registers) from sequential memory locations. The registers are loaded in sequence, the lowest-numbered register first, from the lowest memory address (<start_address>); the highest-numbered register last, from the highest memory address (<end_address>).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

```
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
```

### Exceptions
- Data Abort

### Qualifiers
- Condition Code

### Notes
- **Addressing mode**: The P and U bits distinguish between the different types of addressing mode. See Addressing Mode 4 starting on page 3-116.
- **Banked registers**: 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**: LDM is UNPREDICTABLE in user mode or system mode.
- **Register Rn**: Specifies the base register used by <addressing_mode>.
- **Use of R15**: If register 15 if specified as the base register Rn, the result is UNPREDICTABLE.
- **Base register mode**: The base register is read from the current processor mode registers, not the user mode registers.
- **Data Abort**: If a data abort is signalled, the value left in Rn is the original base register value.
- **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 7), and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.
LDM(<cond>)<addressing_mode>  Rn(!), <registers_and_pc>^  

Description  
This form of the LDM (Load Multiple) instruction is useful for returning from 
an exception, to restore saved registers, load the PC with the return address, 
update the stack pointer, and restore the CPSR from the SPSR.  
In this case, LDM loads a non-empty subset (or possibly all) of the general-
purpose registers and the PC from sequential memory locations. The registers are 
loaded in sequence, the lowest-numbered register first, from the lowest memory 
address; the highest-numbered register last, from the highest memory address. 
The SPSR of the current mode is copied to the CPSR.  
The instruction is only executed if the condition specified in the instruction matches 
the condition code status. The conditions are defined in 3.3 The Condition Field on 
page 3-4.  

Operation  
if ConditionPassed(<cond>) then 
  <address> = <start_address>  
  for i = 0 to 15 
    if <register_list>[i] == 1  
      Ri = Memory[<address>,4] 
      <address> = <address> + 4  
assert <end_address> == <address> - 4  
CPSR = SPSR  

Exceptions  
Data Abort  

Qualifiers  
Condition Code  
! sets the W bit, causing base register update  

Notes  
Addressing mode: The P, U and W bits distinguish between the different types of 
addressing mode. See Addressing Mode 4 starting on page 3-116.  
Register Rn: Specifies the base register used by <addressing_mode>.  
Use of R15: Using R15 as the base register Rn gives an UNPREDICTABLE result.  
User and System mode: This instruction is UNPREDICTABLE in user or system mode.  
Operand restrictions: If the base register Rn is specified in <register_list>, 
and writeback is specified, the final value of Rn is UNPREDICTABLE.  
Data Abort: If a data abort is signalled and <addressing_mode> specifies 
writeback, the value left in Rn is IMPLEMENTATION DEFINED, but is either 
the original base register value or the updated base register value (even if Rn 
is specified in <register_list>). If register 15 is specified in <register_list>, it must not be overwritten if a data abort occurs.  
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 7), and alignment checking is enabled, an address with 
bits[1:0] != 0b00 will cause an alignment exception.
LDR(<cond>) Rd, <addressing_mode>

Description
Combined with a suitable addressing mode, the LDR (Load register) instruction 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 (precede the LDR instruction with MOV LR, PC to synthesize a branch and link).

Using the PC as the base register allows PC-relative addressing, to facilitate position-independent code.

LDR 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 so that the addressed byte occupies the least-significant byte of the register. If the PC is specified as register Rd, the instruction loads a branch to the address (data) into the PC.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

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

Exceptions
Data Abort

Qualifiers
Condition Code

Notes
Addressing modes: The I, P, U and W bits specify the type of <addressing_mode> (see Addressing Mode 2 starting on page 3-98).

Register Rn: Specifies the base register used by <addressing_mode>.

Data Abort: If a data abort is signalled and <addressing_mode> uses pre-indexed or post-indexed addressing, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if the same register is specified for Rd and Rn).

Operand restrictions: If <addressing_mode> uses pre-indexed or post-indexed addressing, 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 7), and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.
LDR{<cond>}B Rd, <addressing_mode>

**Description**
Combined with a suitable addressing mode, the LDRB (Load Register Byte) instruction 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.

LDRB 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.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

**Operation**
\[
\text{if ConditionPassed(<cond>) then}\n\quad\text{Rd} = \text{Memory[<address>,1]}
\]

**Exceptions**
- Data Abort

**Qualifiers**
- Condition Code

**Notes**
**Addressing modes:** The I, P, U and W bits specify the type of `<addressing_mode>` (see Addressing Mode 2 starting on page 3-98).

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

**Use of R15:** If register 15 is specified for Rd the result is UNPREDICTABLE.

**Operand restrictions:** If `<addressing_mode>` uses pre-indexed or post-indexed addressing, and the same register is specified for Rd and Rn, the results are UNPREDICTABLE.

**Non-word-aligned addresses:** Store Word instructions ignore the least-significant two bits of `<address>` (the words are not rotated as for Load Word).

**Data Abort:** If a data abort is signalled and `<addressing_mode>` uses pre-indexed or post-indexed addressing, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if the same register is specified for Rd and Rn).
The LDRBT (Load Register Byte with Translation) instruction 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 has User Mode privilege.

LDRBT loads a byte from the memory address calculated by \(<\text{post\_indexed\_addressing\_mode}>\), zero-extends the byte to a 32-bit word, and writes the word to register \(\text{Rd}\). If the instruction is executed when the processor is in a privileged mode, the memory system is signalled to treat the access as if the processor was in user mode.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in §3.3 The Condition Field on page 3-4.

### Operation

\[
\text{if ConditionPassed}(<\text{cond}>) \text{ then } \\
\quad \text{Rd} = \text{Memory}[<\text{address}>,1]
\]

### Exceptions

Data Abort

### Qualifiers

Condition Code

### Notes

**Addressing modes:** The I, P, and U bits specify the type of \(<\text{addressing\_mode}>\) (see Addressing Mode 2 starting on page 3-98).

**Register \(\text{Rn}\):** Specifies the base register used by \(<\text{post\_indexed\_addressing\_mode}>\).

**User mode:** If this instruction is executed in user mode, an ordinary user mode access is performed.

**Use of \(\text{R15}\):** If register 15 is specified for \(\text{Rd}\), the result is UNPREDICTABLE.

**Operand restrictions:** If the same register is specified for \(\text{Rd}\) and \(\text{Rn}\), the results are UNPREDICTABLE.

**Data Abort:** If a data abort is signalled, the value left in \(\text{Rn}\) is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if the same register is specified for \(\text{Rd}\) and \(\text{Rn}\)).
LDR({cond})H Rd, <addressing_mode>

Description
Used with a suitable addressing mode, the LDRH (Load Register Halfword) instruction 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.

LDRH 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.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

Operation
if ConditionPassed({cond}) then
  if <address>[0] == 0
    <data> = Memory[{address},2]
  else /* <address>[0] == 1 */
    <data> = UNPREDICTABLE
  Rd = <data>

Exceptions
Data Abort

Qualifiers
Condition Code

Notes
Addressing modes: The I, P, U and W bits specify the type of <addressing_mode> (see Addressing Mode 3 starting on page 3-109).

The addr_mode bits: These bits are addressing-mode specific.

Register Rn: Specifies the base register used by <addressing_mode>.

Use of R15: If register 15 is specified for Rd, the result is UNPREDICTABLE.

Operand restrictions: If <addressing_mode> uses pre-indexed or post-indexed addressing, and the same register is specified for Rd and Rn, the results are UNPREDICTABLE.

Data Abort: If a data abort is signalled and <addressing_mode> uses pre-indexed or post-indexed addressing, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if the same register is specified for Rd and Rn).

Non-half-word 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 7), and alignment checking is enabled, an address with bit[0] != 0 will cause an alignment exception.
LDR(\textit{cond})SB \ Rd, <addressing\_mode>

**Description**

Used with a suitable addressing mode, the LDRSB (Load Register Signed Byte) instruction 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, to facilitate position-independent code.

LDRSB 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.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

**Operation**

\[
\begin{align*}
\text{if ConditionPassed(\textit{cond}) then} \\
<\text{data}> &= \text{Memory}[\text{address},1] \\
\text{Rd} &= \text{SignExtend}(<\text{data}>)
\end{align*}
\]

**Exceptions**

Data Abort

**Qualifiers**

Condition Code

**Notes**

**Addressing modes:** The I, P, U and W bits specify the type of <addressing\_mode> (see Addressing Mode 3 starting on page 3-109).

**The addr\_mode bits:** These bits are addressing mode specific.

**Register Rn:** Specifies the base register used by <addressing\_mode>.

**Use of R15:** If register 15 is specified for Rd, the result is UNPREDICTABLE.

**Operand restrictions:** If <addressing\_mode> uses pre-indexed or post-indexed addressing, and the same register is specified for Rd and Rn, the results are UNPREDICTABLE.

**Data Abort:** If a data abort is signalled and <addressing\_mode> uses pre-indexed or post-indexed addressing, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if the same register is specified for Rd and Rn).
LDR{<cond>}SH Rd, <addressing_mode>

Description
Used with a suitable addressing mode, the LDRSH (Load Register Signed Halfword) instruction 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, to facilitate position-independent code.

LDRSH 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.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

Operation
if ConditionPassed(<cond>) then
  if <address>[0] == 0
    <data> = Memory[<address>,2]
  else /* <address>[0] == 1 */
    <data> = UNPREDICTABLE
    Rd = SignExtend(<data>)

Exceptions
Data Abort

Qualifiers
Condition Code

Notes
Addressing modes: The I, P, U and W bits specify the type of <addressing_mode> (see Addressing Mode 3 starting on page 3-109).

The addr_mode bits: These bits are addressing mode specific.

Register Rn: Specifies the base register used by <addressing_mode>.

Use of R15: If register 15 is specified for Rd, the result is UNPREDICTABLE.

Operand restrictions: If <addressing_mode> uses pre-indexed or post-indexed addressing, and the same register is specified for Rd and Rn, the results are UNPREDICTABLE.

Data Abort: If a data abort is signalled and <addressing_mode> uses pre-indexed or post-indexed addressing, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if the same register is specified for Rd and Rn).

Non-half-word 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 7), and alignment checking is enabled, an address with bit[0] != 0 causes an alignment exception.
LDR{<cond>}T Rd, <post_indexed_addressing_mode>

Description
The LDRT (Load Register with Translation) instruction 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 has User Mode privilege.

LDRT loads a word from the memory address and writes it to register Rd. If the instruction is executed when the processor is in a privileged mode, the memory system is signalled to treat the access as if the processor was in user mode.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

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

Exceptions
Data Abort

Qualifiers
Condition Code

Notes
Addressing modes: The I, P, and U bits specify the type of <addressing_mode> (see Addressing Mode 2 starting on page 3-98).

Register Rn: Specifies the base register used by <post_indexed_addressing_mode>.

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: If data abort is signalled, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if the same register is specified for Rd and Rn).

Alignment: If an implementation includes a System Control Coprocessor (See Chapter 7), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.
MCR{<cond>} p<cp#>, <opcode_1>, Rd, CRn, CRm, <opcode_2>

Description
The MCR (Move to Coprocessor from ARM Register) instruction is used to initiate coprocessor instructions that operate on values in ARM registers, for example a fixed-point to floating-point conversion instruction for a floating-point accelerator coprocessor.

MCR passes the value of register Rd to the coprocessor specified by <cp_num>. If no coprocessors indicate that they can execute the instruction, an UNDEFINED instruction exception is generated.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

Operation
if ConditionPassed(<cond>) then
    Coprocessor[<cp_num>] = Rd

Exceptions
Undefined Instruction

Qualifiers
Condition Code

Notes
Coprocessor fields: Only instruction bits[31:24], bit[20], bits[15:8] and bit[4] are ARM architecture-defined; the remaining fields are only recommendations, for compatibility with ARM Development Systems.
MLA\(<\text{cond}>\){<S>} Rd, Rm, Rs, Rn

Description
The MLA (Multiply Accumulate) instruction 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.

MLA multiplies the value of register Rm with the value of register Rs, adds 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).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

Operation
\[
\text{if ConditionPassed(<cond>) then} \\
\quad \text{Rd} = (\text{Rm} \times \text{Rs} + \text{Rn})[31:0] \\
\quad \text{if } S = 1 \text{ then} \\
\qquad \text{N Flag} = \text{Rd}[31] \\
\qquad \text{Z Flag} = \text{if Rd} = 0 \text{ then } 1 \text{ else } 0 \\
\qquad \text{C Flag} = \text{UNPREDICTABLE} \\
\qquad \text{V Flag} = \text{unaffected}
\]

Exceptions
None

Qualifiers
Condition Code
S updates condition code flags N and Z

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.
MOV(<cond>){S} Rd, <shifter_operand>

**Description**
The MOV (Move) instruction 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

When the PC is the destination of the instruction, a branch occurs, and
MOV PC, LR can be used to return from a subroutine call (see the B and BL instructions) and to return from some types of exception (See 2.5 Exceptions on page 2-6).

MOV moves the value of <shifter_operand> to the destination register Rd, and optionally updates the condition code flags (based on the result).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

```
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
```

**Exceptions**
None

**Qualifiers**
Condition Code
S updates condition code flags N, Z and C

**Notes**
Shifter operand: The shifter operands for this instruction are given in Addressing Mode 1 starting on page 3-84.

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

The I bit: Bit 25 is used to distinguish between the immediate and register forms of <shifter_operand>.
The MRC (Move to ARM Register from Coprocessor) instruction is used to initiate coprocessor instructions that return values to ARM registers, for example a floating-point to fixed-point conversion instruction for a floating-point accelerator coprocessor.

Specifying R15 as the destination register is useful for operations like a floating-point compare instruction.

MRC has two uses:
1. If Rd specifies register 15, the condition code flags bits are updated from the top four bits of the value from the coprocessor specified by \(<cp_num>\) (to allow conditional branching on the status of a coprocessor) and the other 28 bits are IGNORED.
2. Otherwise the instruction writes into register Rd a value from the coprocessor specified by \(<cp#>\).

If no coprocessors indicate that they can execute the instruction an UNDEFINED instruction exception is generated.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

**Operation**

\[
\text{if ConditionPassed}\left(\text{<cond>}\right) \text{ then } \\
\quad \text{if Rd == 15 then } \\
\quad \quad N \text{ flag } = (\text{value from Coprocessor}[<cp_num>])[31] \\
\quad \quad Z \text{ flag } = (\text{value from Coprocessor}[<cp_num>])[30] \\
\quad \quad C \text{ flag } = (\text{value from Coprocessor}[<cp_num>])[29] \\
\quad \quad V \text{ flag } = (\text{value from Coprocessor}[<cp_num>])[28] \\
\quad \text{else } /* Rd != 15 */ \\
\quad \quad \text{Rd } = \text{ value from Coprocessor}[<cp_num>] \\
\]

**Exceptions**

Undefined Instruction

**Qualifiers**

Condition Code

**Notes**

Coprocessor fields: Only instruction bits[31:24], bit[20], bits[15:8] and bit[4] are ARM architecture-defined; the remaining fields are only recommendations, that if followed, will be compatible with ARM Development Systems.
The MRS 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.

The MRS moves the value of the CPSR, or the value of the SPSR corresponding to the current mode, to a general-purpose register.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

### Operation

```plaintext
if ConditionPassed(<cond>) then
  if R == 1 then
    Rd = SPSR
  else
    Rd = CPSR
```

### Exceptions

None

### Qualifiers

Condition Code

### Notes

- ** Opcode [11:0]:** Execution of MRS instructions with any non-zero bits in opcode[11:0] is UNPREDICTABLE.
- ** Opcode [19:16]:** Execution of MRS instructions with any non-one bits in opcode[19:16] is UNPREDICTABLE.
- ** User mode SPSR:** Accessing the SPSR when in user mode or system mode is UNPREDICTABLE.
### Description
The MSR (Move to Status register from ARM Register) instruction transfers the value of a general-purpose register to the CPSR or the SPSR of the current mode. This is used to update the value of the condition code flags, interrupt enables, or the processor mode.

MSR moves the value of Rm or the value of the 32-bit immediate (encoded as an 8-bit value with rotate) to the CPSR or the SPSR corresponding to the current mode.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

#### Immediate operand

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>Condition</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>Register</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>field_mask</td>
<td>Mask for registers</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>SBO</td>
<td></td>
</tr>
<tr>
<td>rotate_imm</td>
<td>Rotate value</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8_bit_immediate</td>
<td>Immediate value</td>
</tr>
</tbody>
</table>

#### Register operand

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>Condition</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>Register</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>field_mask</td>
<td>Mask for registers</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>SBO</td>
<td></td>
</tr>
<tr>
<td>SBZ</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Rm</td>
<td></td>
</tr>
</tbody>
</table>

#### Operation

if ConditionPassed(<cond>) then
  if opcode[25] == 1
    <operand> = <8_bit_immediate> Rotate_Right (<rotate_imm> * 2)
  else /* opcode[25] == 0 */
    <operand>
  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
    if <field_mask>[3] == 1 then
  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
    if <field_mask>[3] == 1 and CurrentModeHasSPSR() then
### Exceptions

None

### Qualifiers

Condition Code

<fields> is one of

- \_c sets the control field mask bit (bit 0)
- \_x sets the extension field mask bit (bit 1)
- \_s sets the status field mask bit (bit 2)
- \_f sets the flags field mask bit (bit 3)

### Notes

**Immediate Operand:** The immediate form of this instruction can only be used to set the flag bits (PSR bits 31:24). Using the immediate form on any other fields has UNPREDICTABLE results.

**PSR Update:** The value of a PSR must 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).

**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.

**Deprecated 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 above.

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.
MUL\{<\text{cond}>\}\{<S>\} \ Rd, \ Rm, \ Rs

**Description**
The MUL (Multiply) instruction is used to multiply signed or unsigned variables to produce a 32-bit result.

MUL multiplies the value of register Rm with the value of register Rs, and stores the result in the destination register Rd. The condition code flags are optionally updated (based on the result).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

**Operation**

\[
\text{if ConditionPassed(<cond>) then}
\]
\[
\begin{align*}
\text{Rd} &= (Rm \times Rs)[31:0] \\
\text{if } S &= 1 \text{ then} \\
N \text{ Flag} &= Rd[31] \\
Z \text{ Flag} &= \text{ if } Rd = 0 \text{ then } 1 \text{ else } 0 \\
C \text{ Flag} &= \text{UNPREDICTABLE} \\
V \text{ Flag} &= \text{unaffected}
\end{align*}
\]

**Exceptions**
None

**Qualifiers**
Condition Code
S update condition code flags N,Z

**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.
The MVN (Move negative) instruction is used to:

- write a negative value into a register
- form a bit mask
- take the one’s complement of a value

MVN moves the logical one's compliment of the value of `<shifter_operand>` to the destination register Rd, and optionally updates the condition code flags (based on the result).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

**Operation**

\[
\begin{align*}
\text{if ConditionPassed}(\text{<cond>}) & \text{ then} \\
\text{Rd} &= \text{NOT} <\text{shifter_operand}> \\
\text{if S} &= 1 \text{ and Rd} = \text{R15 then} \\
\text{CPSR} &= \text{SPSR} \\
\text{else if S} &= 1 \text{ then} \\
\text{N Flag} &= \text{Rd}[31] \\
\text{Z Flag} &= \text{if Rd} = 0 \text{ then 1 else 0} \\
\text{C Flag} &= <\text{shifter_carry_out}> \\
\text{V Flag} &= \text{unaffected}
\end{align*}
\]

**Exceptions**
None

**Qualifiers**

- **Condition Code**
  - S: Update condition code flags N,Z,C

**Notes**

- **Shifter operand**: The shifter operands for this instruction are given in Addressing Mode 1 starting on page 3-84.
- **Writing to R15**: When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.
- **The I bit**: Bit 25 is used to distinguish between the immediate and register forms of `<shifter_operand>`. 
ORR{<cond>}{S} Rd, Rn, <shifter_operand>

**Description**
The ORR (Logical OR) instruction can be used to set selected bits in a register; for each bit OR with 1 will set the bit, OR with 0 will leave it unchanged.

ORR 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).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

**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
```

**Exceptions**
None

**Qualifiers**
Condition Code
S updates condition code flags N, Z and C

**Notes**
*Shifter operand:* The shifter operands for this instruction are given in Addressing Mode 1 starting on page 3-84.

*Writing to R15:* When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.

*I bit:* Bit 25 is used to distinguish between the immediate and register forms of <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).

The following instruction stores the negative (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
```

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

### Operation

If `ConditionPassed(<cond>)` then

- \(Rd = <\text{shifter_operand}> - \text{Rn}\)
- 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)

### Exceptions
None

### Qualifiers
Condition Code
S updates condition code flags N,Z,C,V

### Notes

**Shifter operand:** The shifter operands for this instruction are given in *Addressing Mode 1* starting on page 3-84.

**Writing to R15:** When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.

**The I bit:** Bit 25 is used to distinguish between the immediate and register forms of <shifter_operand>. 

---

**Example:**

```
RSB Rd, Rx, #0
```
RSC<cond>{S} Rd, Rn, <shifter_operand>

**Description**
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).

To negate the 64-bit value in R0, R1, use the following sequence (R0 holds the least-significant word) and store the result in R2,R3:

\[
\begin{align*}
&\text{RSBS R2,R0,#0} \\
&\text{RSC R3,R1,#0}
\end{align*}
\]

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

**Operation**
if ConditionPassed<cond> then
\[
\text{Rd} = \text{<shifter_operand>} - \text{Rn} - \text{NOT(C Flag)}
\]
if S == 1 and Rd == R15 then
\[
\text{CPSR} = \text{SPSR}
\]
else if S == 1 then
\[
\begin{align*}
\text{N Flag} &= \text{Rd}[31] \\
\text{Z Flag} &= \text{if Rd} == 0 \text{ then 1 else 0} \\
\text{C Flag} &= \text{NOT BorrowFrom}<shifter_operand> - \text{Rn} - \text{NOT(C Flag)} \\
\text{V Flag} &= \text{OverflowFrom}<shifter_operand> - \text{Rn} - \text{NOT(C Flag)}
\end{align*}
\]

**Exceptions**
None

**Qualifiers**
Condition Code
S updates condition code flags N,Z,C and V

**Notes**
**Shifter operand:** The shifter operands for this instruction are given in Addressing Mode 1 starting on page 3-84.

**Writing to R15:** When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.

**The I bit:** Bit 25 is used to distinguish between the immediate and register forms of <shifter_operand>.
SBC(<cond>) S, Rd, Rn, <shifter_operand>

**Description**  
The SBC (Subtract with Carry) instruction is used to synthesize multi-word subtraction. 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

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).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

```plaintext
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))
```

**Exceptions**  
None

**Qualifiers**  
Condition Code

- S Update condition code flags N,Z,C,V

**Notes**  
**Shifter operand:** The shifter operands for this instruction are given in Addressing Mode 1 starting on page 3-84.

**Writing to R15:** When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.

**The I bit:** Bit 25 is used to distinguish between the immediate and register forms of <shifter_operand>. 
The SMLAL (Signed Multiply Accumulate Long) instruction 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.

SMLAL multiplies the signed value of register \( Rm \) with the signed value of register \( Rs \) to produce a 64-bit result. The lower 32 bits of the result are added to \( RdLo \) and stored in \( RdLo \); the upper 32 bits, and the carry from the addition to \( RdLo \), are added to \( RdHi \) and stored in \( RdHi \). The condition code flags are optionally updated (based on the 64-bit result).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

### Operation

\[
\text{if ConditionPassed(<cond>) then} \\
\text{RdLo} = (Rm \times Rs)[31:0] + RdLo \\
\text{RdHi} = (Rm \times Rs)[63:32] + RdHi + \text{CarryFrom}((Rm \times Rs)[31:0] + RdLo) \\
\text{if } S == 1 \text{ then} \\
\text{N Flag} = RdHi[31] \\
\text{Z Flag} = \text{if } (RdHi == 0) \text{ and } (RdLo == 0) \text{ then } 1 \text{ else } 0 \\
\text{C Flag} = \text{UNPREDICTABLE} \\
\text{V Flag} = \text{UNPREDICTABLE}
\]

### Exceptions
None

### Qualifiers
Condition Code
S updates condition code flags N,Z

### Notes
Use of R15: Specifying R15 for register RdHi, RdLo, Rm or Rs has UNPREDICTABLE results.

Operand restriction: Specifying the same register for RdHi and Rm has UNPREDICTABLE results.
Specifying the same register for RdLo and Rm has UNPREDICTABLE results.
Specifying the same register for RdHi and RdLo 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.
The SMULL (Signed Multiply Long) instruction multiplies signed variables to produce a 64-bit result in two general-purpose registers.

SMULL 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).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

If ConditionPassed(<cond>) then

\[
\begin{align*}
\text{RdHi} &= (\text{Rm} \times \text{Rs})[63:32] \\
\text{RdLo} &= (\text{Rm} \times \text{Rs})[31:0]
\end{align*}
\]

if S == 1 then

\[
\begin{align*}
\text{N Flag} &= \text{RdHi}[31] \\
\text{Z Flag} &= \text{if (RdHi == 0) and (RdLo == 0) then 1 else 0} \\
\text{C Flag} &= \text{UNPREDICTABLE} \\
\text{V Flag} &= \text{UNPREDICTABLE}
\end{align*}
\]

None

Condition Code

S updates condition code flags N, Z

Use of R15: Specifying R15 for register RdHi, RdLo, Rm or Rs has UNPREDICTABLE results.

Operand restriction: Specifying the same register for RdHi and Rm has UNPREDICTABLE results.

Specifying the same register for RdLo and Rm has UNPREDICTABLE results.

Specifying the same register for RdHi and RdLo 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.
STC(<cond>)  p<cp_num>, CRd, <addressing_mode>

Description
The STC (Store Coprocessor) instruction is useful for storing coprocessor data to memory. The N bit could be used to distinguish between a single- and double-precision transfer for a floating-point store instruction.

STC stores data from the coprocessor specified by <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.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

Operation
if ConditionPassed(<cond>) then
    <address> = <start_address>
    while (NotFinished(coprocessor[<cp_num>]))
        Memory[<address>,4] = value from Coprocessor[<cp_num>]
        <address> = <address> + 4
    assert <address> == <end_address>

Operation
Undefined Instruction; Data Abort

Qualifiers
Condition Code

Notes
Addressing mode: The P, U and W bits specify the <addressing_mode>. See Addressing Mode 5 starting on page 3-123.

The N bit: This bit is coprocessor-dependent. It can be used to distinguish between two sizes of data to transfer.

Register Rn: Specifies the base register used by <addressing_mode>.

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 only recommendations, for compatibility with ARM Development Systems.

Data Abort: If a data abort is signalled and <addressing_mode> uses pre-indexed or post-indexed addressing, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value.

Non-word-aligned addresses: Store coprocessor register 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 7), and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.
Load and store multiple

Addressing mode 4

STM<cond><addressing_mode>  Rn(!), <registers>

Description
The STM (Store Multiple) instruction is useful as a block store instruction (combined with load multiple it allows efficient block copy) and for stack operations, including procedure entry to save general-purpose registers and the return address, and for updating the stack pointer.

STM stores a non-empty subset (or possibly all) of the general-purpose registers to sequential memory locations. The registers are stored in sequence, the lowest-numbered register first, to the lowest memory address (<start_address>); the highest-numbered register last, to the highest memory address (<end_address>).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

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

Exceptions
Data Abort

Qualifiers
Condition Code
! sets the W bit, causing base register update

Notes
Addressing mode: The R, U and W bits distinguish between the different types of addressing mode. See Addressing Mode 4 starting on page 3-116.
Register Rn: Specifies the base register used by <addressing_mode>.
Use of R15: If register 15 if specified as the base register Rn, the result is UNPREDICTABLE. If register 15 is specified in <register_list>, the value stored is IMPLEMENTATION DEFINED.
Operand restrictions: If Rn is specified in <register_list>, and writeback is specified, the stored value of Rn is UNPREDICTABLE.
Data Abort: If a data abort is signalled and <addressing_mode> specifies writeback, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value.
Non-word-aligned addresses: STM 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 7), and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.
The STM (Store Multiple) instruction is used to store the user mode registers when the processor is in a privileged mode (useful when performing process swaps). This form of STM stores a subset (or possibly all) of the user mode general-purpose registers (which are also the system mode general-purpose registers) to sequential memory locations. The registers are stored in sequence, the lowest-numbered register first, to the lowest memory address (<start_addr>); the highest-numbered register last, to the highest memory address (<end_addr>).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

**Operation**

if ConditionPassed(<cond>) then
```
<address> = <start_addr>
for i = 0 to 15
  if <register_list>[i] == 1
    Memory[<address>,4] = Ri_usr
    <address> = <address> + 4
assert <end_addr> == <address> - 4
```

**Exceptions**

Data Abort

**Qualifiers**

Condition Code

**Notes**

- **Addressing mode**: The P and W bits distinguish between the different types of addressing mode. See Addressing Mode 4 starting on page 3-116.
- **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.
- **Register Rn**: Specifies the base register used by <addressing_mode>.
- **Use of R15**: If register 15 is specified as the base register Rn, the result is UNPREDICTABLE. If register 15 is specified in <register_list> the value stored is IMPLEMENTATION DEFINED.
- **Base register mode**: The base register is read from the current processor mode registers, not the user mode registers.
- **Data Abort**: If a data abort is signalled, the value left in Rn is the original base register value.
- **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 7), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.
The STR (Store register) instruction stores 32-bit data from a general purpose register into memory. Using the PC as the base register allows PC-relative addressing, to facilitate position-independent code.

STR stores a word from register Rd to the memory address calculated by the addressing mode specified in the instruction.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

**Operation**

if ConditionPassed\(<\text{cond}>\) then

\[
\text{Memory}[<\text{address}>,4] = \text{Rd}
\]

**Exceptions**

Data Abort

**Qualifiers**

Condition Code

**Notes**

Addressing modes: The I, P, U and W bits specify the type of the addressing mode (see Addressing Mode 2 starting on page 3-98).

Register Rn: Specifies the base register used by the addressing mode.

Use of R15: If register 15 is specified for Rd, the value stored is IMPLEMENTATION DEFINED.

Operand restrictions: If the addressing mode uses pre-indexed or post-indexed addressing, and the same register is specified for Rd and Rn, the results are UNPREDICTABLE.

Data Abort: If a data abort is signalled and the addressing mode uses preindexed or post-indexed addressing, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if the same register is specified for Rd and Rn).

Alignment: If an implementation includes a System Control Coprocessor (See Chapter 7), and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.
STRB

Description
Combined with a suitable addressing mode, the STRB (Store Register Byte) writes the least-significant byte of a general-purpose register to memory.

Using the PC as the base register allows PC-relative addressing, to facilitate position-independent code.

STRB stores a byte from the least-significant byte of register Rd to the memory address calculated by <addressing_mode>.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

<table>
<thead>
<tr>
<th>cond</th>
<th>0</th>
<th>1</th>
<th>I</th>
<th>P</th>
<th>U</th>
<th>W</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
<th>addressing mode specific</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation
if ConditionPassed(<cond>) then
Memory[<address>,1] = Rd[7:0]

Exceptions
Data Abort

Qualifiers
Condition Code

Notes
Addressing modes: The I, P, U and W bits specify the type of <addressing_mode> (see Addressing Mode 2 starting on page 3-98).

Register Rn: Specifies the base register used by <addressing_mode>.

Use of R15: If register 15 is specified for Rd, the result is UNPREDICTABLE.

Operand restrictions: If <addressing_mode> uses pre-indexed or post-indexed addressing, and the same register is specified for Rd and Rn, the results are UNPREDICTABLE.

Data Abort: If a data abort is signalled and <addressing_mode> uses pre-indexed or post-indexed addressing, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if the same register is specified for Rd and Rn).
**STR{<cond>}BT** Rd, <post_indexed_addressing_mode>

**Description**
The STRBT (Store Register Byte with Translation) instruction 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 has User Mode privilege.

STRBT 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 signalled to treat the access as if the processor were in user mode.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

**Operation**
if ConditionPassed(<cond>) then
    Memory[<address>,1] = Rd[7:0]

**Exceptions**
Data Abort

**Qualifiers**
Condition Code

**Notes**
Addressing modes: The I, P, and U bits specify the type of <addressing_mode> (see Addressing Mode 2 starting on page 3-98).

Register Rn: Specifies the base register used by <post_indexed_addressing_mode>.

User mode: If this instruction is executed in user mode, an ordinary user mode access is performed.

Use of R15: If register 15 is specified for Rd, the result is UNPREDICTABLE.

Operand restrictions: If the same register is specified for Rd and Rn, the results are UNPREDICTABLE.

Data Abort: If a data abort is signalled, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if the same register is specified for Rd and Rn).
STR{<cond>}H Rd, <addressing_mode>

Description
Combined with a suitable addressing mode, the STRH (Store Register Halfword) instruction 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.

STRH 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.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

Operation
if ConditionPassed(<cond>) then
    if <address>[0] == 0
        <data> = Rd[15:0]
    else /* <address>[0] == 1 */
        <data> = UNPREDICTABLE
    Memory[<address>,2] = <data>

Exceptions
Data Abort

Qualifiers
Condition Code

Notes
Addressing modes: The I, P, U and W bits specify the type of <addressing_mode> (see Addressing Mode 3 starting on page 3-109).

The addr_mode bits: These bits are addressing-mode specific.

Register Rn: Specifies the base register used by <addressing_mode>.

Use of R15: If register 15 is specified for Rd, the result is UNPREDICTABLE.

Operand restrictions: If <addressing_mode> uses pre-indexed or post-indexed addressing, and the same register is specified for Rd and Rn, the results are UNPREDICTABLE.

Data Abort: If a data abort is signalled and <addressing_mode> uses pre-indexed or post-indexed addressing, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if the same register is specified for Rd and Rn).

Non-half-word 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 7), and alignment checking is enabled, an address with bit[0] != 0 will cause an alignment exception.
**STRT**

**Description**
The STRT (Store Register with Translation) instruction 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 has User Mode privilege.

STRT 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 signalled to treat the access as if the processor was in user mode.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

**Operation**

\[
\text{if ConditionPassed} (<\text{cond}>) \text{ then}
\]

\[
\text{Memory}[<\text{address}>,4] = \text{Rd}
\]

**Exceptions**
Data Abort

**Qualifiers**
Condition Code

**Notes**

Addressing modes: The I, P, and U bits specify the type of <addressing_mode> (see Addressing Mode 2 starting on page 3-98).

Register Rn: Specifies the base register used by <post_indexed_addressing_mode>.

User mode: If this instruction is executed in user mode, an ordinary user mode access is performed.

Use of R15: If register 15 is specified for Rd, the value stored is IMPLEMENTATION DEFINED.

Operand restrictions: If the same register is specified for Rd and Rn, the results are UNPREDICTABLE.

Data Abort: If a data abort is signalled, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if the same register is specified for Rd and Rn).

Alignment: If an implementation includes a System Control Coprocessor (See Chapter 7), and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.
### SUB{<cond>}{S} Rd, Rn, <shifter_operand>

#### Description
The SUB (Subtract) instruction is used to subtract one value from another to produce a third. To decrement a register value (in Rx) use:

```
SUB Rx, Rx, #1
```

SUB 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).

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 `CMP Rx, #0`.

Use `SUBS PC, LR, #4` to return from an interrupt.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

#### Operation
If `ConditionPassed(<cond>)` then

```
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>)
```

#### Exceptions
None

#### Qualifiers
Condition Code
S updates condition code flags N,Z,C and V

#### Notes
**Shifter operand:** The shifter operands for this instruction are given in **Addressing Mode 1** starting on page 3-84.

**Writing to R15:** When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.

**The I bit:** Bit 25 is used to distinguish between the immediate and register forms of `<shifter_operand>`.

---

The following table shows the encoding of the instruction:

<table>
<thead>
<tr>
<th>Field</th>
<th>R15</th>
<th>R14</th>
<th>R13</th>
<th>R12</th>
<th>R11</th>
<th>R10</th>
<th>R9</th>
<th>R8</th>
<th>R7</th>
<th>R6</th>
<th>R5</th>
<th>R4</th>
<th>R3</th>
<th>R2</th>
<th>R1</th>
<th>R0</th>
<th>Cond</th>
<th>I</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cond</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3-74

ARM Architecture Reference Manual
ARM DUI 0100B
**SWI(<cond>) <24_bit_immediate>**

**Description**

The SWI instruction causes a SWI exception, see 2.5 Exceptions on page 2-6. The SWI instruction is used as an operating system service call. It can be used in two ways:

- to use the 24-bit immediate value to indicate the OS service that is required
- to ignore the 24-bit field and indicate the service required with a general-purpose register

A SWI exception is generated, which is handled by an operating system to provide the requested service.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

**Operation**

if ConditionPassed(<cond>) then

- R14_svc = address of SWI instruction + 4
- SPSR_svc = CPSR
- CPSR[5:0] = 0b010011; enter Supervisor mode
- CPSR[7] = 1; disable IRQ
- PC = 0x08

**Exceptions**

None

**Qualifiers**

Condition Code
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 Rn, this instruction swaps the value of the register and the value at the memory address.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in Section 3.3 The Condition Field on page 3-4.

### Operation

```
if ConditionPassed(<cond>) then
    <temp> = Memory[Rn, 4]
    Memory[Rn, 4] = Rm
    Rd = <temp>
```

### Exceptions

Data Abort

### Qualifiers

Condition Code

### Notes

**Non-word-aligned addresses:** If the address is not word-aligned, the loaded value is rotated right by 8 times the value of `<address>[1:0]`.

**Use of R15:** If register 15 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 signalled on either the load access or the store access (or both), the loaded value is not written to Rd.

**Alignment:** If an implementation includes a System Control Coprocessor (see Chapter 7), and alignment checking is enabled, an address with `bits[1:0] != 0b00` will cause an alignment exception.
SWP(<cond>)B Rd, Rm, [Rn]

Description
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, and 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 Rn, this instruction swaps the value of the least-significant byte of the register
and the byte value at the memory address.

The instruction is only executed if the condition specified in the instruction matches
the condition code status. The conditions are defined in 3.3 The Condition Field on
page 3-4.

Operation
if ConditionPassed(<cond>) then
<temp> = Memory[Rn,1]
Memory[Rn,1] = Rm[7:0]
Rd = <temp>

Exceptions
Data Abort

Qualifiers
Condition Code

Notes
Use of R15: If register 15 is specified for Rd, Rn or Rm, the result is UNPREDICTABLE.
Operands restrictions: If the same register is specified as Rn and Rm, Rn and or Rd,
the result is UNPREDICTABLE.

Data Abort: If a data abort is signalled on either the load access or the store
access (or both), the loaded value is not written to Rd.
TEQ(<cond>)  Rn, <shifter_operand>

**Description**
The TEQ (Test equivalence) instruction is used to test if two values are equal, without affecting the V flag (as CMP does). TEQ is also useful for testing if two values have the same sign.

The comparison is the Logical Exclusive OR of the two operands.

TEQ performs a comparison by logically Exclusive ORing the value of register Rn with the value of <shifter_operand>, and updates the condition code flags (based on the result).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

**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
```

**Exceptions**
None

**Qualifiers**
Condition Code

**Notes**

*Shifter operand:* The shifter operands for this instruction are given in Addressing Mode 1 starting on page 3-84.

*The I bit:* Bit 25 is used to distinguish between the immediate and register forms of <shifter_operand>.
TST(<cond>) Rn, <shifter_operand>

Description
The TST (Test) instruction is used to determine if many bits of a register are all clear, or if at least one bit of a register is set. The comparison is a logical AND of the two operands.

TST performs a comparison by logically ANDing the value of register Rn with the value of <shifter_operand>, and updates the condition code flags (based on the result).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

<table>
<thead>
<tr>
<th>cond</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>Rn</th>
<th>SBZ</th>
<th>shifter_operand</th>
</tr>
</thead>
</table>

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

Exceptions
None

Qualifiers
Condition Code

Notes
- **Shifter operand**: The shifter operands for this instruction are given in Addressing Mode 1 starting on page 3-84.

- **The I bit**: Bit 25 is used to distinguish between the immediate and register forms of <shifter_operand>.
### Description
The UMLAL (Unsigned Multiply Accumulate Long) instruction 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.

UMLAL multiplies the unsigned value of register Rm with the unsigned value of register Rs to produce a 64-bit result. The lower 32 bits of the result are added to RdLo and stored in RdLo; the upper 32 bits and the carry from the addition to RdLo are added to RdHi and stored in RdHi. The condition code flags are optionally updated (based on the 64-bit result).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in [3.3 The Condition Field](#) on page 3-4.

### Operation
\[
\text{if ConditionPassed}(<\text{cond}>) \text{ then}
\begin{align*}
\text{RdLo} &= (\text{Rm} \times \text{Rs})[31:0] + \text{RdLo} \\
\text{RdHi} &= (\text{Rm} \times \text{Rs})[63:32] + \text{RdHi} + \text{CarryFrom}((\text{Rm} \times \text{Rs})[31:0] + \text{RdLo}) \\
\text{if } S &= 1 \text{ then}
\begin{align*}
\text{N Flag} &= \text{RdHi}[31] \\
\text{Z Flag} &= \text{if } (\text{RdHi} == 0) \text{ and } (\text{RdLo} == 0) \text{ then } 1 \text{ else } 0 \\
\text{C Flag} &= \text{UNPREDICTABLE} \\
\text{V Flag} &= \text{UNPREDICTABLE}
\end{align*}
\]

### Exceptions
None

### Qualifiers
- **Condition Code**
  - S updates condition code flags N and Z

### Notes
- **Use of R15:** Specifying R15 for register RdHi, RdLo, Rm or Rs has UNPREDICTABLE results.
- **Operand restriction:** Specifying the same register for RdHi and Rm has UNPREDICTABLE results.
  Specifying the same register for RdLo and Rm has UNPREDICTABLE results.
  Specifying the same register for RdHi and RdLo 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.
The UMULL (Unsigned Multiply Long) instruction multiplies unsigned variables to produce a 64-bit result in two general-purpose registers. UMULL 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).

The instruction is only executed if the condition specified in the instruction matches the condition code status. See 3.3 The Condition Field on page 3-4.

if ConditionPassed(<cond>) then

\[
\begin{align*}
RdHi &= (Rm \times Rs)[63:32] \\
RdLo &= (Rm \times Rs)[31:0]
\end{align*}
\]

if S == 1 then

\[
\begin{align*}
N \text{ Flag} &= RdHi[31] \\
Z \text{ Flag} &= \text{if } (RdHi == 0) \text{ and } (RdLo == 0) \text{ then 1 else 0} \\
C \text{ Flag} &= \text{UNPREDICTABLE} \\
V \text{ Flag} &= \text{UNPREDICTABLE}
\end{align*}
\]

None

Condition Code

S updates condition code flags N and Z

Use of R15: Specifying R15 for register RdHi, RdLo, Rm or Rs has UNPREDICTABLE results.

Specifying the same register for RdHi and Rm has UNPREDICTABLE results.

Specifying the same register for RdLo and Rm has UNPREDICTABLE results.

Specifying the same register for RdHi and RdLo has UNPREDICTABLE results.

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.
ARM Addressing Modes

Addressing Mode
- Mode 1: Shifter operands for data-processing instructions
- Mode 2: Load and store word or unsigned byte
- Mode 3: Load and store halfword or load signed byte
- Mode 4: Load and store multiple
- Mode 5: Load and store coprocessor

Description
If the condition specified in the instruction matches the condition code status register Rm is added to or subtracted from the value of the base register Rn.

Operation
- If ConditionPassed(<cond>) then
  - if U == 1 then
    Rn = Rn + Rm
  - else /* U == 0 */
    Rn = Rn - Rm

Qualifiers
- None

Notes
- The L bit: This bit distinguishes between a Load (L==1) and a Store (L==0).
- The S bit: This bit distinguishes between a signed (S==1) and an unsigned (S==0) word.
- The H bit: This bit distinguishes between a halfword (H==1) and a signed byte (H==0).
- Use of R15: Specifying R15 as register Rm or Rn has UNPREDICTABLE results.
3.16 Data-processing Operands

\[
\text{<opcode>}{\langle\text{cond}\rangle}{\langle S \rangle}{\langle \text{Rd} \rangle}, \ldots, \langle \text{Rn} \rangle, \langle \text{shifter\_operand} \rangle}
\]

### 32-bit immediate

<table>
<thead>
<tr>
<th>cond</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>rotate_imm</th>
<th>8_bit_immediate</th>
</tr>
</thead>
</table>

### Immediate shifts

<table>
<thead>
<tr>
<th>cond</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Rs</th>
<th>shift</th>
<th>0</th>
<th>Rm</th>
</tr>
</thead>
</table>

### Register shifts

<table>
<thead>
<tr>
<th>cond</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Rs</th>
<th>0</th>
<th>shift</th>
<th>1</th>
<th>Rm</th>
</tr>
</thead>
</table>

**Description**

- `<opcode>` Describes 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 below.
- 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>`.

#### 3.16.1 The shifter operand

As well as producing `<shifter\_operand>`, the shifter produces a carry-out which some instructions write into the Carry Flag.

The shifter operand takes one of 3 basic formats:

- Immediate operand value
- Register operand value
- Shifted register operand value

**Format 1: 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). Thus, each instruction contains an 8-bit constant and a 4-bit rotate to be applied to that constant.
Valid constants are:
0xff, 0x104, 0xff0, 0xff00, 0xff000, 0xff000000, 0xf000000f

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

Format 2: 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

Format 3: 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:

<table>
<thead>
<tr>
<th>Shift Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASR</td>
<td>Arithmetic shift right</td>
</tr>
<tr>
<td>LSL</td>
<td>Logical shift left</td>
</tr>
<tr>
<td>LSR</td>
<td>Logical shift right</td>
</tr>
<tr>
<td>ROR</td>
<td>Rotate right</td>
</tr>
<tr>
<td>RRX</td>
<td>Rotate right with extend</td>
</tr>
</tbody>
</table>

The number of bits to shift by is specified either as an immediate or as the value of the register.

For example:
MOV R2, R0 LSL #2 ; Shift R0 left by 2, store in 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

The default shifter operand
The default register operand (register Rm specified with no shift) uses the form register shift left by immediate, with the immediate set to zero.
3.16.2 Shifter Operands

The 11 types of <shifter_operand> are described on the following pages:

- Immediate page 3-87
- Register page 3-88
- Logical shift left by immediate page 3-89
- Logical shift left by register page 3-90
- Logical shift right by immediate page 3-91
- Logical shift right by register page 3-92
- Arithmetic shift right by immediate page 3-93
- Arithmetic shift right by register page 3-94
- Rotate right by immediate page 3-95
- Rotate right by register page 3-96
- Rotate right with extend page 3-97
#<immediate>

**Description**
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>.

This data-processing operand provides a constant (defined in the instruction) operand to a data-processing instruction.

**Operation**
<shifter_operand> = <8_bit_immediate> Rotate_Right
(rotate_imm) * 2)
if <rotate_imm> == 0 then
  <shifter_carry_out> = C flag
else /* <rotate_imm> > 31 */
  <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.

**Alternative assembly specification:** The 32-bit immediate can also be specified by:
#<8_bit_immediate>, <rotate_amount>
where:
<rotate_amount> = <rotate_imm> << 1
### Register Mode 1

#### Description
This data-processing operand provides the value of a register directly. This is an instruction operand produced by the value of register Rm. The carry-out from the shifter is the C flag.

#### Operation

<table>
<thead>
<tr>
<th>cond</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rm</th>
</tr>
</thead>
</table>

- `<shifter_operand> = Rm`
- `<shifter_carry_out> = C Flag`

#### Notes
- **Encoding:** This instruction is encoded as a Logical shift left by immediate (see page 3-89) 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.
**Logical shift left by immediate**

**Description**
This data-processing operand is used to provide either the value of a register directly (lone register operand (see page 3-88), or the value of a register shifted left (multiplied by a constant power of two).

This instruction operand is produced by 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 (lone register operand, see page 3-88).

<table>
<thead>
<tr>
<th>Condition</th>
<th>Opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>shift_imm</th>
<th>0</th>
<th>0</th>
<th>Rm</th>
</tr>
</thead>
</table>

**Operation**
if \(<\text{shift}\_\text{imm}> == 0\) then /* Register Operand */

\(<\text{shifter}\_\text{operand}> = \text{Rm}\>

\(<\text{shifter}\_\text{carry}\_\text{out}> = \text{C Flag}\>

else /* \(<\text{shift}\_\text{imm}> > 0\) */

\(<\text{shifter}\_\text{operand}> = \text{Rm Logical\_Shift\_Left}<\text{shift}\_\text{imm}>\>

\(<\text{shifter}\_\text{carry}\_\text{out}> = \text{Rm}[32 - <\text{shift}\_\text{imm}>]\>

**Notes**
- **Default shift:** If the value of \(<\text{shift}\_\text{imm}> == 0\), the operand may be written as just Rm, (see page 3-88).
- **Use of R15:** If R15 is specified as register Rm or Rn, the value used is the address of the current instruction plus 8.
**Logical shift left by register**

**Description**

This data-processing operand is used to provide the value of a register multiplied by a variable (in a register) power of two.

It is produced by 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.

**Table**

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>Rs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Operation**

if Rs[7:0] == 0 then
  \(\text{shifter\_operand} = Rm\)
  \(\text{shifter\_carry\_out} = C\ Flag\)
else if Rs[7:0] < 32 then
  \(\text{shifter\_operand} = Rm \text{ Logical\_Shift\_Left} Rs[7:0]\)
  \(\text{shifter\_carry\_out} = Rm[32 - Rs[7:0]]\)
else if Rs[7:0] == 32 then
  \(\text{shifter\_operand} = 0\)
  \(\text{shifter\_carry\_out} = Rm[0]\)
else /* Rs[7:0] > 32 */
  \(\text{shifter\_operand} = 0\)
  \(\text{shifter\_carry\_out} = 0\)

**Notes**

Use of R15: Specifying R15 as register Rm, register Rn or register Rs has UNPREDICTABLE results.
This data-processing operand is used to provide the unsigned value of a register shifted right (divided by a constant power of two).

It is produced by 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. A shift by 32 is encoded by \(<\text{shift\_imm}> = 0\).

<table>
<thead>
<tr>
<th>cond</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>shift_imm</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>Rm</th>
</tr>
</thead>
</table>

If \(<\text{shift\_imm}> = 0 \) then

\(<\text{shifter\_operand}> = 0\>
\(<\text{shifter\_carry\_out}> = \text{Rm}[31]\>

else /* \(<\text{shift\_imm}> > 0 \) */

\(<\text{shifter\_operand}> = \text{Rm \_Logical\_Shift\_Right} \ <\text{shift\_imm}>\>
\(<\text{shifter\_carry\_out}> = \text{Rm}[<\text{shift\_imm}> - 1]\>

Use of R15: If R15 is specified as register Rm or Rn, the value used is the address of the current instruction plus 8.
Logical shift right by register

**Rm, LSR Rs**

**Description**

This data-processing operand is used to provide the unsigned value of a register shifted right (divided by a variable power of two (in a register)). 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.

**Operation**

<table>
<thead>
<tr>
<th>Condition</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Rs</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>opcode</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>Rs</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Rm</td>
</tr>
</tbody>
</table>

if Rs[7:0] == 0 then
  \(<\text{shifter_operand}> = \text{Rm}\)
  \(<\text{shifter_carry_out}> = \text{C Flag}\)
else if Rs[7:0] < 32 then
  \(<\text{shifter_operand}> = \text{Rm Logical_Shift_Right Rs[7:0]}\)
  \(<\text{shifter_carry_out}> = \text{Rm[Rs[7:0] - 1]}\)
else if Rs[7:0] == 32 then
  \(<\text{shifter Operand}> = 0\)
  \(<\text{shifter_carry_out}> = \text{Rm[31]}\)
else /* Rs[7:0] > 32 */
  \(<\text{shifter_operand}> = 0\)
  \(<\text{shifter_carry_out}> = 0\)

**Notes**

Use of R15: Specifying R15 as register Rm, register Rn or register Rs has UNPREDICTABLE results.
Arithmetic shift right
by immediate

This data-processing operand is used to provide the signed value of a register arithmetically shifted right (divided by a constant power of two). It is produced by 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. A shift by 32 is encoded by \(<\text{shift\_imm}> = 0\).

<table>
<thead>
<tr>
<th>Addressing Mode 1</th>
<th>ARM Architecture Reference Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rm, ASR #&lt;shift_imm&gt;</td>
<td>Arithmetic shift right by immediate</td>
</tr>
</tbody>
</table>

### Description
This data-processing operand is used to provide the signed value of a register arithmetically shifted right (divided by a constant power of two).

### Operation
if \(<\text{shift\_imm}> = 0\) then
  if \(\text{Rm}[31] = 0\) then
    \(<\text{shifter\_operand}> = 0\)
    \(<\text{shifter\_carry\_out}> = \text{Rm}[31]\)
  else /* \(\text{Rm}[31] = 1\) */
    \(<\text{shifter\_operand}> = \text{0xffffffff}\)
    \(<\text{shifter\_carry\_out}> = \text{Rm}[31]\)
  else /* \(<\text{shift\_imm}> > 0\) */
    \(<\text{shifter\_operand}> = \text{Rm\ Arithmetic\_Shift\_Right}\)
    \(<\text{shift\_imm}> = \text{Rm}[<\text{shift\_imm}> - 1]\)
### 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.
Arithmetic shift right by register

**Description**

This data-processing operand is used to provide the signed value of a register arithmetically shifted right (divided by a variable power of two (in a register)).

It is produced by 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.

**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 Rm, register Rn or register Rs has UNPREDICTABLE results.
This data-processing operand is used to provide the value of a register rotated by a constant value. An instruction operand produced by 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.

When \(<shift\_imm> = 0\), a Rotate right with extend operation is performed; see page 3-97.

\[
\begin{array}{cccccccccccc}
31 & 28 & 27 & 26 & 25 & 24 & 21 & 20 & 19 & 16 & 15 & 12 & 11 & 7 & 6 & 5 & 4 & 3 & 0 \\
\hline
\text{cond} & 0 & 0 & 0 & \text{opcode} & S & Rn & Rd & \text{shift\_imm} & 1 & 1 & 0 & Rm
\end{array}
\]

\begin{enumerate}
\item \textbf{Operation} 
\begin{enumerate}
\item if \(<shift\_imm> = 0\) then
\begin{enumerate}
\item See Section , Rm, RRX, on page 3-97
\end{enumerate}
\item else /* \(<shift\_imm> > 0\) */
\begin{enumerate}
\item <shifter\_operand> = Rm Rotate\_Right \(<shift\_imm>\)
\item <shifter\_carry\_out> = Rm[<shift\_imm> - 1]
\end{enumerate}
\end{enumerate}
\item \textbf{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.
**Description**

This data-processing operand is used to provide the value of a register rotated by a variable value (in a register).

It 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.

**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 \text{ Rotate\_Right} Rs[4:0]
  <shifter\_carry\_out> = Rm[Rs[4:0] - 1]

**Notes**

Use of R15: Specifying R15 as register Rm, register Rn or register Rs has UNPREDICTABLE results.
**Rm, RRX**

**Description**
This data-processing operand can be used to perform a 33-bit rotate right using the Carry Flag as the 33rd bit.

It is produced by the value of register Rm shifted right by one bit, with the Carry Flag replacing the vacated bit position.

**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 right with extend can be performed with an ADC instruction.
3.17 Load and Store Word or Unsigned Byte Addressing Modes

There are nine addressing modes used to calculate the address for a load and store word or unsigned byte instruction. Each addressing mode is described in detail on the following pages.

Immediate offset page 3-100
LDR|STR{<cond>}{B} Rd, [Rn, #+/-<12_bit_offset>]

Register offset page 3-101
LDR|STR{<cond>}{B} Rd, [Rn, +/-Rm]

Scaled register offsets page 3-102
LDR|STR{<cond>}{B} Rd, [Rn, +/-Rm, <shift> #<shift_imm>]

Immediate pre-indexed page 3-103
LDR|STR{<cond>}{B} Rd, [Rn, #+/-<12_bit_offset>]

Register pre-indexed page 3-104
LDR|STR{<cond>}{B} Rd, [Rn, +/-Rm]!

Scaled register pre-indexed page 3-105
LDR|STR{<cond>}{B} Rd, [Rn, +/-Rm, <shift> #<shift_imm>]!

Immediate post-indexed page 3-106
LDR|STR{<cond>}{B}{T}Rd, [Rn], #+/-<12_bit_offset>

Register post-indexed page 3-107
LDR|STR{<cond>}{B}{T}Rd, [Rn], +/-Rm

Scaled register post-indexed page 3-108
LDR|STR{<cond>}{B}{T}Rd, [Rn], +/-Rm, <shift> #<shift_imm>

Immediate offset/index

<table>
<thead>
<tr>
<th>cond</th>
<th>0</th>
<th>1</th>
<th>P</th>
<th>U</th>
<th>B</th>
<th>W</th>
<th>L</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12_bit_offset</td>
</tr>
</tbody>
</table>

Register offset/index

| cond | 0 | 1 | P | U | B | W | L | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|------|---|---|---|---|---|---|---|----|----|----|----|----|---|---|---|---|---|---|---|---|---|
| Rd   |   |   |   |   |   |   |   |    |    |    |    |    |    |    | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Rm |

Scaled register offset/index

| cond | 0 | 1 | P | U | B | W | L | 16 | 15 | 12 | 11 | 7 | 6 | 5 | 4 | 3 | 0 |
|------|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|---|---|---|---|
| Rd   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    | 0 | shift_imm | shift | 0 | Rm |
Notes

The P bit: Pre/Post indexing:
Pre-indexing (P==1) indicates the offset is applied to the base register, and the result is used as the address.
Post-indexing (P==0) indicates the base register value is used for the address; the offset is then applied to the base register and written back to the base register.

The U bit: Indicates whether the offset is added to the base (U == 1) or subtracted from the base (U == 0).

The B bit: This bit distinguishes between an unsigned byte (B == 1) and a word (B == 0) access.

The W bit: This bit has two meanings:
if P == 1  if W == 1, the calculated address will be written back to the base register. (If W == 0, the base register will not be updated.)
if P == 0  if W == 1, the current access is treated (by the protection and memory system) as a user mode access (if W == 0, a normal access is performed)

The L bit: This bit distinguishes between a Load (L == 1) and a Store (L == 0).
Immediate offset

**Description**  
This addressing mode is useful 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.

It calculates an address by adding or subtracting the value of an immediate offset to or from the value of the base register Rn.

**Operation**

if U == 1 then  
<address> = Rn + <12_bit_offset>
else /* U == 0 */  
<address> = Rn - <12_bit_offset>

**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.
[Rn, +/- Rm]

Description
This addressing mode is used for pointer + offset arithmetic, and accessing a single element of an array. It calculates an address by adding or subtracting the value of the index register Rm to or from the value of the base register Rn.

<table>
<thead>
<tr>
<th>cond</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>U</th>
<th>B</th>
<th>0</th>
<th>L</th>
<th>Rn</th>
<th>Rd</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rm</th>
</tr>
</thead>
</table>

Operation
if U == 1 then
  \langle address\rangle = Rn + Rm
else /* U == 0 */
  \langle address\rangle = Rn - Rm

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.
These addressing modes are used for accessing a single element of an array of values larger than a byte. They 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.

**Operation**

```plaintext
case <shift> of
  00 /* LSL */
    <index> = Rm Logical_Shift_Left <shift_imm>
  01 /* LSR */
    <index> = Rm Logical_Shift_Right <shift_imm>
  10 /* ASR */
    <index> = Rm Arithmetic_Shift_Right <shift_imm>
  11 /* 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>
```

**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.
[Rn, #+/-<12_bit_offset>]

**Description**
This addressing mode is used for pointer access to arrays with automatic update of the pointer value.

It 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 3.3 The Condition Field on page 3-4.

**Operation**

```plaintext
if U == 1 then
    <address> = Rn + <12_bit_offset>
else /* if U == 0 */
    <address> = Rn - <12_bit_offset>
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 Rn has UNPREDICTABLE results.
Addressing Mode 2

Register pre-indexed

Description
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 3.3 The Condition Field on page 3-4.

Operation
\[
\begin{align*}
\text{if } U == 1 \text{ then} \\
& \quad \langle\text{address}\rangle = Rn + Rm \\
\text{else } /* U == 0 */ \\
& \quad \langle\text{address}\rangle = Rn - Rm \\
& \quad \text{if ConditionPassed}\langle\text{cond}\rangle \text{ then} \\
& \quad \quad \text{Rn} = \langle\text{address}\rangle
\end{align*}
\]

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 Restrictions: If the same register is specified for Rn and Rm, the result is UNPREDICTABLE.
[Rn, +/-Rm, LSL #<shift_imm>]!
[Rn, +/-Rm, LSR #<shift_imm>]!
[Rn, +/-Rm, ASR #<shift_imm>]!
[Rn, +/-Rm, ROR #<shift_imm>]!
[Rn, +/-Rm, RRX]!

Description
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 3.3 The Condition Field on page 3-4.

<table>
<thead>
<tr>
<th>cond</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>U</th>
<th>B</th>
<th>1</th>
<th>L</th>
<th>Rn</th>
<th>Rd</th>
<th>shift_imm</th>
<th>shift</th>
<th>0</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>16</td>
<td>15</td>
</tr>
</tbody>
</table>

Operation
```
case <shift> of
  00 /* LSL */
    <index> = Rm Logical_Shift_Left <shift_imm>
  01 /* LSR */
    <index> = Rm Logical_Shift_Right <shift_imm>
  10 /* ASR */
    <index> = Rm Arithmetic_Shift_Right <shift_imm>
  11 /* 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 Restrictions: If the same register is specified for Rn and Rm, the result is UNPREDICTABLE.
[Rn], #/−<12 bit_offset>

**Description**

This addressing mode is used for pointer access to arrays with automatic update of the pointer value.

It calculates an address from the value of base register Rn.

If the condition specified in the instruction matches the condition code status, the value of the immediate offset 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 3.3 The Condition Field on page 3-4

**Operation**

<address> = Rn

if ConditionPassed(<cond>) then

if U == 1 then

Rn = Rn + <12 bit_offset>
else /* U == 0 */

Rn = Rn - <12 bit_offset>

**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 Rn has UNPREDICTABLE results.
This addressing mode calculates its address from the value of base register Rn. 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 3.3 The Condition Field on page 3-4.

\[
\text{Operation} \\
\begin{align*}
<\text{address}> &= Rn \\
\text{if } &\text{ConditionPassed(<cond>) then} \\
&\quad \text{if } U == 1 \text{ then} \\
&\quad \quad Rn = Rn + Rm \\
&\quad \text{else } /* U == 0 */ \\
&\quad \quad Rn = Rn - Rm
\end{align*}
\]

\textbf{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 Rn or Rm has UNPREDICTABLE results.
- Operand Restrictions: If the same register is specified for Rn and Rm, the result is UNPREDICTABLE.
### Description

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 3.3 The Condition Field on page 3-4.

### Operation

\[
\text{<address>} = \text{Rn}
\]

\[
\text{case } \text{<shift>} \text{ of}
\]

- 00 /* LSL */
  \[
  \text{<index>} = \text{Rm Logical_Shift_Left } \text{<shift_imm>}
  \]

- 01 /* LSR */
  \[
  \text{<index>} = \text{Rm Logical_Shift_Right } \text{<shift_imm>}
  \]

- 10 /* ASR */
  \[
  \text{<index>} = \text{Rm Arithmetic_Shift_Right } \text{<shift_imm>}
  \]

- 11 /* ROR or RRX */
  \[
  \text{if } \text{<shift_imm>} = 0 \text{ then } /* RRX */
  \]
  \[
  \text{<index>} = (C \text{ Flag Logical_Shift_Left 31})
  \]
  \[
  \text{OR (Rm Logical_Shift_Right 1)}
  \]
  \[
  \text{else } /* \text{ROR} */
  \]
  \[
  \text{<index>} = \text{Rm Rotate_Right } \text{<shift_imm>}
  \]

endcase

\[
\text{if ConditionPassed(<cond>) then}
\]

\[
\text{if } U = 1 \text{ then}
\]

\[
\text{Rn} = \text{Rn} + \text{<index>}
\]

\[
\text{else } /* U = 0 */
\]

\[
\text{Rn} = \text{Rn} - \text{<index>}
\]

### 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 Restrictions:** If the same register is specified for Rn and Rm, the result is UNPREDICTABLE.
Addressing
Mode 3

3.18 Load and Store Halfword or Load Signed Byte Addressing Modes

There are six addressing modes which are used to calculate the address for a load and store (signed or unsigned) halfword or load signed byte instructions.

1 Immediate offset
   LDR|STR{<cond>}H|SH|SB  Rd, [Rn, #/+<8_bit_offset>]

2 Register offset
   LDR|STR{<cond>}H|SH|SB  Rd, [Rn, +/-Rm]

3 Immediate pre-indexed
   LDR|STR{<cond>}H|SH|SB  Rd, [Rn, #/+<8_bit_offset>]!

4 Register pre-indexed
   LDR|STR{<cond>}H|SH|SB  Rd, [Rn, +/-Rm]!

5 Immediate post-indexed
   LDR|STR{<cond>}H|SH|SB  Rd, [Rn], #/+<8_bit_offset>

6 Register post-indexed
   LDR|STR{<cond>}H|SH|SB  Rd, [Rn], +/-Rm

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 |

Notes

The P bit: Pre-indexing (P==1) indicates the offset is applied to the base register, and the result is used as the address. Post-indexing (P==0) indicates the base register value is used for the address; the offset is then applied to the base register and written back to the base register.

The U bit: Indicates whether the offset is added to the base (U==1) or subtracted from the base(U==0).

The W bit: If P is set, W indicates that the calculated address will be written back to the base register; if P is clear, the W bit must be clear or the instruction is UNPREDICTABLE.

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.
Addressing Mode 3

Immediate offset

Architecture v4 only

[Rn, #/+/<8_bit_offset>]

**Description**
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. It calculates an address by adding or subtracting the value of an immediate offset to or from the value of the base register Rn.

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>U</td>
<td>1</td>
<td>0</td>
<td>L</td>
<td>Rn</td>
<td>Rd</td>
<td>ImmedH</td>
<td>1</td>
<td>S</td>
<td>H</td>
<td>1</td>
<td>ImmedL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Operation**

<8_bit_offset> = (immedH) << 4) OR immedL
if U == 1 then
  address = Rn + <8_bit_offset>
else /* U == 0 */
  address = Rn - <8_bit_offset>

**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.

**Use of R15:** If R15 is specified as register Rn, the value used is the address of the instruction plus 8.
[Rn, +/- Rm]

**Description**
This addressing mode is useful for pointer + offset arithmetic, and for accessing a single element of an array.
It calculates an address by adding or subtracting the value of the index register Rm to or from the value of the base register Rn.

**Operation**
\[
\text{if } U == 1 \text{ then} \\
\text{<address> } = \text{Rn} + \text{Rm} \\
\text{else } /* U == 0 */ \\
\text{<address> } = \text{Rn} - \text{Rm}
\]

**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.

**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.
**Addressing Mode 3**

**Immediate pre-indexed**

**Architecture v4 only**

### Description

This addressing mode gives pointer access to arrays, with automatic update of the pointer value.

It 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 **3.3 The Condition Field** on page 3-4.

### Operation

\[
\text{<8_bit_offset>} = (<\text{immedH}> \ll 4) \text{ OR } <\text{immedL}>
\]

- if \( U == 1 \) then
  \[
  \text{<address>} = Rn + <\text{8_bit_offset}>
  \]
- else /* \( U == 0 */
  \[
  \text{<address>} = Rn - <\text{8_bit_offset}>
  \]

if ConditionPassed(<cond>) then

\[
Rn = <\text{address}>
\]

### 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.

**Use of R15:** Specifying R15 as register \( Rn \) has UNPREDICTABLE results.
Addressing Mode 3

Description

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 3.3 The Condition Field on page 3-4.

Operation

\[
\begin{align*}
\text{if } U &= 1 \text{ then } \\
\langle \text{address} \rangle &= Rn + Rm \\
\text{else } /* U &= 0 */ & \\
\langle \text{address} \rangle &= Rn - Rm \\
\text{if } \text{ConditionPassed}(<\text{cond}>) \text{ then } \\
Rn &= \langle \text{address} \rangle
\end{align*}
\]

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.

Use of R15: Specifying R15 as register Rm or Rn has UNPREDICTABLE results.
Immediate post-indexed

Architecture v4 only

### Addressing Mode 3

[<Rn>, #/+<8_bit_offset>]

#### Description

This addressing mode gives pointer access to arrays, with automatic update of the pointer value.

It calculates an address from the value of base register Rn.

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 3.3 The Condition Field on page 3-4.

#### Operation

\[
\text{<address> = Rn}
\]

\[
\text{<8_bit_offset> = (immedH} \ll 4) \text{ OR immedL}
\]

if ConditionPassed(<cond>)) then

\[
\text{if } \text{U} = 1 \text{ then}
\]

\[
\text{Rn} = \text{Rn} + \text{<8_bit_offset>}
\]

else /* U == 0 */

\[
\text{Rn} = \text{Rn} - \text{<8_bit_offset>}
\]

#### 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.

**Use of R15:** Specifying R15 as register Rn has UNPREDICTABLE results.
**Addressing Mode 3**

**Register post-indexed**

**Architecture v4 only**

---

**Addressing Mode 3**

**Register post-indexed**

**Architecture v4 only**

---

### [Rn], +/- Rm

#### Description

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 3.3 The Condition Field on page 3-4.

#### Operation

\[
\begin{align*}
<\text{address}> &= Rn \\
\text{if ConditionPassed(<cond>)} &\text{ then} \\
\quad &\text{if } U == 1 \text{ then} \\
\quad &\quad Rn = Rn + Rm \\
\quad &\text{else /* } U == 0 */ \\
\quad &\quad Rn = Rn - Rm
\end{align*}
\]

#### 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.
- **Use of R15:** Specifying R15 as register Rm or Rn has UNPREDICTABLE results.
3.19 Load and Store Multiple Addressing Modes

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. These instructions have a single instruction format.

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.

There are four Load and Store Multiple addressing modes:

1. Increment After
   \[ \text{LDM|STM}<\text{cond}>IA \ Rn[!], \ <\text{registers}>^{\text{^}} \]
2. Increment Before
   \[ \text{LDM|STM}<\text{cond}>IB \ Rn[!], \ <\text{registers}>^{\text{^}} \]
3. Decrement After
   \[ \text{LDM|STM}<\text{cond}>DA \ Rn[!], \ <\text{registers}>^{\text{^}} \]
4. Decrement Before
   \[ \text{LDM|STM}<\text{cond}>DB \ Rn[!], \ <\text{registers}>^{\text{^}} \]

Notes

- **The register list:** The \(<\text{register_list}>\) has 1 bit for each general-purpose register; bit 0 is for register zero, bit 15 is for register 15 (the PC). The \(<\text{register_list}>\) is specified in the instruction mnemonic using a comma-separated list of registers, surrounded by brackets. If no bits are set, the result is UNPREDICTABLE.
- **The U bit:** Indicates that the transfer is made upwards (U==1) or downwards (U==0) from base register.
- **The P bit:** Pre-indexing or post-indexing:
  - P==1 indicates that each address in the range is incremented (U==1) or decremented (U==0) before it is used to access memory.
  - P==0 indicates that each address in the range is incremented (U==1) or decremented (U==0) after it is used to access memory.
- **The W bit:** Indicates that the base register will be 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 S bit:** For LDMs that load the PC, the S bit indicates that the CPSR is loaded from the SPSR. For LDMs that do not load the PC and all STMs, the S bit 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.
- **The L bit:** Distinguishes between Load (L==1) and Store (L==0) instructions.
Addressing Mode 4

**Description**

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 `<register_list>`.

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 `<register_list>`.

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 numbers of registers in `<register_list>`. The conditions are defined in 3.3 The Condition Field on page 3-4.

**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)
```

**Qualifiers**

- `!` sets the W bit, causing base register update
- `^` is used to set the S bit, see below.

**Notes**

- The **L** bit: This bit distinguishes between a load multiple and a store multiple.
- The **S** bit: The action of the **S** bit is described in section 3.19 Load and Store Multiple Addressing Modes on page 3-116.
LDM|STM{<cond>}IB Rd(!), <registers>{^}

**Description**
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 `<register_list>`.

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 `<register_list>`.

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 numbers of registers in `<register_list>`. The conditions are defined in 3.3 The Condition Field on page 3-4.

- **Operation**
  
  \[
  \begin{align*}
  \text{<start_address>} &= \text{Rn} + 4 \\
  \text{<end_address>} &= \text{Rn} + (\text{Number Of Set Bits In}<\text{register_list}> \times 4) \\
  \text{if ConditionPassed}<\text{cond}> \text{ and } W == 1 \text{ then} \\
  & \quad \text{Rn} = \text{Rn} + (\text{Number Of Set Bits In}<\text{register_list}> \times 4)
  \end{align*}
  \]

- **Qualifiers**
  
  ! sets the W bit, causing base register update
  
  ^ is used to set the S bit, see below.

- **Notes**
  
  The L bit: This bit distinguishes between a load multiple and a store multiple.
  
  The S bit: The action of the S bit is described in section 3.19 Load and Store Multiple Addressing Modes on page 3-116.
### Addressing Mode 4

#### LDM|STM(<cond>)DA  Rd(!), <registers>{^}

**Description**

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 `<register_list>`, plus 4. Subsequent addresses are formed by incrementing the previous address by four. One address is produced for each register that is specified in `<register_list>`.

The last address produced is the `<end_address>`; its value is the value of 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 numbers of registers in `<register_list>`. The conditions are defined in 3.3 The Condition Field on page 3-4.

#### 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)`

#### Qualifiers

- `!` sets the W bit, causing base register update
- `^` is used to set the S bit, see below.

#### Notes

- **The L bit**: This bit distinguishes between a load multiple and a store multiple.
- **The S bit**: The action of the S bit is described in section 3.19 Load and Store Multiple Addressing Modes on page 3-116.
**Description**

This addressing mode is for Load and Store multiple instructions which form 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 `<register_list>`. Subsequent addresses are formed by incrementing the previous address by four. One address is produced for each register that is specified in `<register_list>`.

The last address produced is the `<end_address>`; its value is the value of 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 numbers of registers in `<register_list>`. The conditions are defined in 3.3 The Condition Field on page 3-4.

**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)
  ```

**Qualifiers**

- `!` sets the W bit, causing base register update
- `^` is used to set the S bit, see below.

**Notes**

- **The L bit:** This bit distinguishes between a load multiple and a store multiple.
- **The S bit:** The action of the S bit is described in section 3.19 Load and Store Multiple Addressing Modes on page 3-116.
3.20 Load and Store Multiple Addressing Modes
(Alternative names)

3.20.1 Block data transfer

The four addressing mode names given in 3.18 Load and Store Halfword or Load Signed Byte Addressing Modes on page 3-109 (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 will 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 will not be 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.

3.20.2 Stack operations

Load Multiple and Store Multiple addressing modes may be specified with an alternative syntax, which is more applicable to stack operations. Two attributes are used to describe the stack.

Full or Empty

- Full: is defined to have the stack pointer pointing to the last used (full) location in the stack
- Empty: is defined to have the stack pointer pointing to the first unused (empty) location in the stack

Ascending or Descending

- Descending: grows towards decreasing memory address (towards the bottom of memory)
- Ascending: grows towards increasing memory address (towards the top of memory)

This allows four types of stack to be defined:
1. Full Descending (FD)
2. Empty Descending (ED)
3. Full Ascending (FA)
4. Empty Ascending (EA)

Table 3-1: LDM/STM addressing modes shows 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 3-1: LDM/STM addressing modes

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Addressing Mode</th>
<th>Stack Type</th>
<th>L bit</th>
<th>P Bit</th>
<th>U bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDM (Load)</td>
<td>IA (Increment After)</td>
<td>FD (Full Descending)</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>STM (Store)</td>
<td>IA (Increment After)</td>
<td>EA (Empty Ascending)</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>LDM (Load)</td>
<td>IB (Increment Before)</td>
<td>ED (Empty Descending)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>STM (Store)</td>
<td>IB (Increment Before)</td>
<td>FA (Full Ascending)</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>LDM (Load)</td>
<td>DA (Decrement After)</td>
<td>FA (Full Ascending)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>STM (Store)</td>
<td>DA (Decrement After)</td>
<td>ED (Empty Descending)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LDM (Load)</td>
<td>DB (Decrement Before)</td>
<td>EA (Empty Ascending)</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>STM (Store)</td>
<td>DB (Decrement Before)</td>
<td>FD (Full Descending)</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
3.21 Load and Store Coprocessor Addressing Modes

There are three addressing modes which are used to calculate the address of a load or store coprocessor instruction:

1. Immediate offset
   \[
   \text{opcode}\{\text{cond}\}\{L\}p<\text{cp}\#>, \text{CRd}, [Rn, \#+/-(<8\_bit\_offset>*4)]
   \]

2. Immediate pre-indexed
   \[
   \text{opcode}\{\text{cond}\}\{L\}p<\text{cp}\#>, \text{CRd}, [Rn, \#+/-(<8\_bit\_offset>*4)]!
   \]

3. Immediate post-indexed
   \[
   \text{opcode}\{\text{cond}\}\{L\}p<\text{cp}\#>, \text{CRd}, [Rn], \#+/-(<8\_bit\_offset>*4)
   \]

**Notes**

- **The P bit:** Pre-indexing (P==1) or post-indexing (P==0):
  - (P==1) indicates that the offset is added to the base register, and the result is used as the address.
  - (P==0) indicates that the base register value is used for the address; the offset is then added to the base register and written back to the base register (because W will equal 1, see below).

- **The U bit:** Indicates that the offset is added to the base (U==1) or that the offset is subtracted from the base (U==0).

- **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:** This indicates that the calculated address will be written back to the base register. If P is 0, W must equal 1 or the result is UNPREDICTABLE.

- **The L bit:** Distinguishes between Load (L==1) and Store (L==0) instructions.
Addressing Mode 5

Immediate offset

[Rn, #+/-(<8_bit_offset>*4)]

Description
This addressing mode produces a sequence of consecutive addresses. The first address is calculated by adding or subtracting 4 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.

Operation

if ConditionPassed(<cond>) then
  if U == 1 then
    <address> = Rn + <8_bit_offset> * 4
  else /* U == 0 */
    <address> = Rn - <8_bit_offset> * 4
  <start_address> = <address>
  while (NotFinished(coprocessor[<cp_num>]))
    <address> = <address> + 4
    <end_address> = <address>

Notes

The N bit: This 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.
Immediate pre-indexed

**Description**
This addressing mode produces a sequence of consecutive addresses. The first address is calculated by adding or subtracting 4 times the value of an immediate offset to or from the value of the base register Rn. 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.

**Operation**
if ConditionPassed(<cond>) then
  if U == 1 then
    Rn = Rn + <8_bit_offset> * 4
  else /* U == 0 */
    Rn = Rn - <8_bit_offset> * 4
  <start_address> = Rn
  <address> = <start_address>
  while (NotFinished(coprocessor[{cp_num}]))
    <address> = <address> + 4
    <end_address> = <address>

**Notes**
The **N** bit: This 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.
Immediate post-indexed

### ARM Addressing Mode 5

#### [Rn], #/+-(<8_bit_offset>*4)

**Description**

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 updated by adding or subtracting 4 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.

**Operation**

```
if ConditionPassed(<cond>) then
    <start_address> = Rn
    if U == 1 then
        Rn = Rn + <8_bit_offset> * 4
    else /* U == 0 */
        Rn = Rn - <8_bit_offset> * 4
    <address> = <start_address>
    while (NotFinished(coprocessor[<cp_num>]))
        <address> = <address> + 4
    <end_address> = <address>
```

**Notes**

- **The N bit:** This 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 W bit:** If bit 21 (the Writeback bit) is not set, the result is UNPREDICTABLE.
ARM Code Sequences
The ARM instruction set is a powerful tool for generating high-performance microprocessor systems. Used to full extent, the ARM instruction set allows very compact and efficient algorithms to be coded. This chapter contains some sample routines that provide insight into the ARM instruction set.

4.1 Arithmetic Instructions 4-2
4.2 Branch Instructions 4-4
4.3 Load and Store Instructions 4-6
4.4 Load and Store Multiple Instructions 4-8
4.5 Semaphore Instructions 4-9
4.6 Other Code Examples 4-10
4.1 Arithmetic Instructions

The following code sequences illustrate some ways of using ARM's data-processing instructions.

4.1.1 Bit field manipulation

ARM shift and logical operations are very useful for bit manipulation:

; Extract 8 bits from the top of R2 and insert them into the bottom of R3
; R0 is a temporary value
MOV R0, R2, LSR #24 ; extract top bits from R2 into R0
ORR R3, R0, R3, LSL #8; shift up R3 and insert R0

4.1.2 Multiplication by constant

Combinations of shifts, add with shifts, and reverse subtract with shift can be used to perform multiplication by constant:

; multiplication of R0 by 2^n
MOV R0, R0, LSL #n ; R0 = R0 << n

; multiplication of R0 by 2^n + 1
ADD R0, R0, R0, LSL #n ; R0 = R0 + R0 << n

; multiplication of R0 by 2^n - 1
RSB R0, R0, LSL #n ; R0 = R0 << n - R0

; R0 = R0 * 10 + R1
ADD R0, R0, R0, LSL #4 ; R0 = R0 * 5
ADD R0, R1, R0, LSL #1 ; R0 = R1 + R0 * 2

; R0 = R0 * 100 + R1, R2 is destroyed
ADD R2, R0, R0, LSL #3 ; R2 = R0 * 9
ADD R0, R2, R0, LSL #4 ; R0 = R2 + R0 * 16 (R0 = R0 * 25)
ADD R0, R1, R0, LSL #2 ; R0 = R1 + R0 * 4
4.1.3 Multi-precision arithmetic

Arithmetic instructions allow efficient arithmetic on 64-bit or larger objects:

Add, and Add with Carry — perform multi-precision addition
Subtract, and Subtract with Carry — perform subtraction
Compare — can be used for comparison

; On entry: R0 and R1 hold a 64-bit number
; (R0 is least significant)
; : R2 and R3 hold a second 64-bit number
; On exit: R0 and R1 hold the 64-bit sum (or difference) of the 2 numbers
add64 ADDSR0, R0, R2 ; add lower halves and update Carry flag
ADC R1, R1, R3 ; add the high halves and Carry flag

sub64 SUBSR0, R0, R2 ; subtract lower halves, update Carry
SBC R1, R1, R3 ; subtract high halves and Carry

; This routine compares two 64-bit numbers
; On entry: As above
; On exit: N, Z, C and V flags updated correctly
cmpeq CMP R1, R3 ; compare high halves, if they are
CMPEQRA, R2 ; equal, then compare lower halves

4.1.4 Swapping endianness

Swapping the order of bytes in a word (the endianness) can be performed in two ways.

1 The first method is best for single words:

; On entry: R0 holds the word to be swapped
; On exit: R0 holds the swapped word, R1 is destroyed
byteswap
EOR R1, R0, R0, ROR #16 ; R1 = A^C, B^D, C^A, D^B
BIC R1, R1, #0xff0000 ; R1 = A^C, 0, C^A, D^B
MOV R0, R0, ROR #8 ; R0 = D, A, B, C
EOR R0, R0, R1, LSR #8 ; R0 = D, C, B, A

2 The second method is best for swapping the endianness of a large number of words:

; On entry: R0 holds the word to be swapped
; On exit: R0 holds the swapped word,
; : R1, R2 and R3 are destroyed
byteswap
MOV R2, #0xff ; first the three instruction initialisation
ORR R2, R2, #0xff0000 ; R2 = 0xff0000
MOV R3, R2, LSL #8 ; R3 = 0xff0000
; repeat the following code for each word to swap
ADD R1, R2, R0, ROR #16 ; R1 = 0, C, 0, A
ADD R0, R3, R0, ROR #16 ; R0 = D, 0, B, 0
ADD R0, R0, R1 ; R0 = D, C, B, A
4.2 Branch Instructions

The following code sequences show some different ways of controlling the flow of execution in ARM code.

4.2.1 Procedure call and return

The BL (Branch and Link) instruction makes a procedure call by preserving the address of the instruction after the BL in R14 (the link register or LR), and then branching to the target address. Returning from a procedure is achieved by moving R14 to the PC:

```assembly
BL function ; call ‘function’
.... ; procedure returns to here
....
function ; function body
....
....
MOV PC, LR ; Put R14 into PC to return
```

4.2.2 Conditional execution

Conditional execution allow if-then-else statements to be collapsed into sequences that do not require forward branches:

```c
/* C code for Euclid’s Greatest Common Divisor (GCD)*/
/* Returns the GCD of its two parameters */
int gcd(int a, int b)
{ while (a != b)
    if (a > b )
        a = a - b ;
    else
        b = b - a ;
    return a ;
}
```

```assembly
; ARM assembler code for Euclid’s Greatest Common Divisor
; On entry: R0 holds ‘a’, R1 holds ‘b’
; On exit : R0 hold GCD of A and B
gcd  CMP R0, R1 ; compare ‘a’ and ‘b’
    SUBGT R0, R0, R1; if (a>b) a=a-b (if a==b do nothing)
    SUBLT R1, R1, R0; if (b>a) b=b-a (if a==b do nothing)
    BNE gcd ; if (a!=b) then keep going
MOV PC, LR ; return to caller
```
4.2.3 Conditional compare instructions

Compare instructions can be conditionally executed to implement more complicated expressions:

```
if (a==0 || b==1)
    c = d + e;
CMP R0, #0 ; compare a with 0
CMPNE R1, #1 ; if a is not 0, compare b to 1
ADDEQ R2, R3, R4 ; if either was true c = d + e
```

4.2.4 Loop variables

The subtract instruction can be used to both decrement a loop counter and set the condition codes to test for a zero.

```
MOV R0, #loopcount ; initialise the loop counter
loop
    ....
    ....
SUBS R0, R0, #1 ; subtract 1 from counter
    ; set condition codes
    BNE loop ; if not zero, continue looping
```

4.2.5 Multi-way branch

A very simple multi-way branch can be implemented with a single instruction. The following code dispatches the control of execution to any number of routines, with the restriction that the code to handle each case of the multi-way branch is the same size, and that size is a power of two bytes.

```
; Multi-way branch
; On entry: R0 holds the branch index
gcd CMP R0, #maxindex ; (optional) checks the index is in range
ADDLT PC, PC, R0, LSL #RoutineSizeLog2
    ; scale index by the log of the size of each handler; add to the PC; jump there
    B IndexOutOfRange; jump to the error handler
Index0Handler
    ....
Index1Handler
    ....
Index2Handler
    ....
Index3Handler
    ....
```
4.3 Load and Store Instructions

Load and Store instructions are the best way to load or store a single word. They are also the only instructions that can load or store a byte or halfword.

4.3.1 Simple string compare

The following code performs a very simple string compare on two zero-terminated strings.

```
; String compare
; On entry : R0 points to the first string
; : R1 points to the second string
; : Call this code with a BL
; On exit : R0 is < 0 if the first string is less than the second
; : R0 is = 0 if the first string is equal to the second
; : R0 is > 0 if the first string is greater than the second
; : R1, R2 and R3 are destroyed

strcmp
    LDRB    R2, [R0] #1 ; get a byte from the first string
    LDRB    R3, [R1] #1; get a byte from the second string
    CMP     R2, #0; reached the end?
    BEQ    return; go to return code; calculate return value
    SUBS    R0, R2, R3; compare the two bytes
    BEQ    strcmp; if they are equal, keep looking

return
    MOV     PC, LR; put R14 (LR) into PC to return
```

Much faster implementations of this code are possible by loading a word of each string at a time and comparing all four bytes.

4.3.2 Linked lists

The following code searches for an element in a linked list. The linked list has two elements in each record; a single byte value and a pointer to the next record. A null next pointer indicates this is the last element in the list.

```
; Linked list search
; On entry: R0 holds a pointer to the first record in the list
; R1 holds the byte we are searching for
; : Call this code with a BL
; On exit : R0 hold the address of the first record matched
; : R2 is destroyed
; or a null pointer if no match was found

llsearch
    CMP     R0, #0 ; null pointer?
    LDRNEB2, [R0] ; load the byte value from this record
    CMPNER1, R2 ; compare with the looked-for value
    LDRNER0, [R0, #4] ; if not found, follow the link to the
    BNE llsearch ; next record and then keep looking
    MOV     PC, LR ; return with pointer in R0
```
4.3.3 Long branch

A load instruction can be used to generate a branch to anywhere in the 4Gbyte address space. By manually setting the value of the link register (R14), a subroutine call can be made to anywhere in the address space.

; Long branch (and link)
ADD LR, PC, #4 ; set the return address to be 8 byte after the next instruction
LDR PC, [PC, #-4] ; get the address from the next word
DCD function ; store the address of the function
; (DCD is an assembler directive)
; return to here

This code uses the location after the load to hold the address of the function to call. In practice, this location can be accessing as long as it is within 4Kbytes of the load instruction. Notice also, that this code is position-independent except for the address of the function to call. Full position-independence can be achieved by storing the offset of the branch target after the load, and using an ADD instruction to add it to the PC.

4.3.4 Multi-way branches

The following code improves on the multi-way branch code shown above by using a table of addresses of functions to call.

; Multi-way branch
; On entry: R0 holds the branch index
CMP R0, #maxindex ; (optional) checks the index is in range
LDRLT PC, [PC, LSL #2] ; convert the index to a word offset
; do a look up in the table put the loaded value into the PC and jump there
B IndexOutOfRange ; jump to the error handler
DCD Handler0 ; DCD is an assembler directive to DCD Handler1
; store a word (in this case an address in memory.
DCD Handler2
DCD Handler3
4.4 Load and Store Multiple Instructions

Load and Store Multiple instructions are the most efficient way to manipulate blocks of data.

4.4.1 Simple block copy

This code performs very simple block copy, 48 bytes at a time, and will approach the maximum throughput for a particular machine. The source and destination must be word-aligned, and objects with less than 48 bytes must be handled separately.

; Simple block copy function
; R12 points to the start of the source block
; R13 points to the start of the destination block
; R14 points to the end of the source block
loopLDMIA R12!, (R0-R11) ; load 48 bytes
STMIA R13!, {R0-R11} ; store 48 bytes
CMP R12, R14 ; reached the end yet?
BLT loop ; branch to the top of the loop

4.4.2 Procedure entry and exit

This code uses load and store multiple to preserve and restore the processor state during a procedure. The code assumes the register r0 to r3 are argument registers, preserved by the caller of the function, and therefore do not need to be preserved. R13 is also assumed to point to a full descending stack.

function
STEMFD R13!, {R4 - R12, R14}; preserve all the local registers
; and the return address, and
; update the stack pointer.

; function body
LDMFD R13!, {R4 - R12, PC}; restore the local register, load
; the PC from the saved return
; update the stack pointer.

Notice that this code restores all saved registers, updates the stack pointer, and returns the caller (by loading the PC value) all in with single instruction. This allows very efficient conditional return for exceptional cases from a procedure (by checking the condition with a compare instruction and the conditionally executing the load multiple).
4.5 Semaphore Instructions

This code controls the entry and exit from a critical section of code. The semaphore instructions do not provide a compare and conditional write facility; this must be done explicitly. The following code achieves this by using a semaphore value to indicate that the lock is being inspected.

The code below causes the calling process to busy-wait until the lock is free; to ensure progress, three OS calls need to be made (one before each loop branch) to sleep the process if the lock cannot be accessed.

```assembly
; Critical section entry and exit
; The code uses a process ID to identify the lock owner
; An ID of zero indicates the lock is free
; An ID of -1 indicates the lock is being inspected
; On entry: R0 holds the address of the semaphore
; R1 holds the ID of the process requesting the lock

MVN R2, #0 ; load the 'looking' value (-1) in R2
spinin SWP R3, R2, [R0] ; look at the lock, and lock others out
CMN R3, #1 ; anyone else trying to look?
: conditional OS call to sleep process
BEQ spinin ; yes, so wait our turn
CMP R3, #0 ; no-one looking, is the lock free?
STRNE R3, [R0] ; no, then restore the previous owner
: conditional OS call to sleep process
BNE spinin ; and wait again
STR R1, [R0] ; otherwise grab the lock
.....
Insert critical code here
.....
spinout SWP R3, R2, [R0] ; look at the lock, and lock others out
CMN R3, #1 ; anyone else trying to look?
: conditional OS call to sleep process
BEQ spinout ; yes, so wait our turn
CMP R3, R1 ; check we own it
BNE CorruptSemaphore ; we should have been the owner!
MOV R2, #0 ; load the 'free' value
STR R2, [R0] ; and open the lock
```
4.6 Other Code Examples

The following sequences illustrate some other applications of ARM assembly language.

4.6.1 Software Interrupt dispatch

This code segment dispatches software interrupts (SWIs) to individual handlers. The SWI instruction has a 24-bit field that can be used to encode specific SWI functions.

```assembly
STMFD SP!, {R12} ; save some registers
LDR R12, [R14, #-4] ; load the SWI instruction
BIC R12, R12, #0xff000000; preserve the SWI number
CMP R12, #MaximumSWI ; check the SWI number is in range, if so
LDRLE PC, [PC, R12, LSL #2]; branch through a table to the handler
B UnkownSWI ; this SWI number is not supported
DCD SWI0Handler ; address of handler for SWI 0
DCD SWI1Handler ; address of handler for SWI 1
DCD SWI2Handler ; address of handler for SWI 2
....
```

4.6.2 Single-channel DMA transfer

The following code is an interrupt handler to perform interrupt driven IO to memory transfers (soft DMA). The code is especially useful as a FIQ handler, as it uses the banked FIQ registers to maintain state between interrupts. Therefore this code is best situated at location 0x1c.

```assembly
R8 points to the base address of the IO device that data is read from
IOData is the offset from the base address to the 32-bit data register that is read. Reading this register disables the interrupt
R9 points to the memory location where data is being transferred
R10 points to the last address to transfer to

The entire sequence to handle a normal transfer is just 4 instructions; code situated after the conditional return is used to signal that the transfer is complete.

LDR r11, [r8, #IOData] ; load port data from the IO device
STR r11, [r9], #4 ; store it to memory: update the pointer
CMP r9, r10 ; reached the end?
SUBLTS pc, lr, #4 ; no, so return
; insert transfer complete code here

Of course, byte transfers can be made by replacing the load instructions with load byte instructions, and transfers from memory to and IO device are made by swapping the addressing modes between the load instruction and the store instruction.

```
4.6.3 Dual-channel DMA transfer

This code is similar to the example in 4.6.2 Single-channel DMA transfer on page 4-10, except that it handles two channels (which may be the input and output side of the same channel). Again this code is especially useful as a FIQ handler, as it uses the banked FIQ registers to maintain state between interrupts. Therefore, this code is best situated at location 0x1c.

The entire sequence to handle a normal transfer is just 9 instructions; code situated after the conditional return is used to signal that the transfer is complete.

LDR r13, [r8, #IOStat] ; load status register to find ....
TST r13, #IOPort1Active ; .... which port caused the interrupt?
LDREQ r13, [r8, #IOPort1] ; load port 1 data
LDNE r13, [r8, #IOPort2] ; load port 2 data
STREQ r13, [r9], #4 ; store to buffer 1
STRNE r13, [r10], #4 ; store to buffer 2
CMP r9, r11 ; reached the end?
CMPNE r10, r12 ; on either channel?
SUBNES pc, lr, #4 ; return
; Insert transfer complete code here

where:

R8 points to the base address of the IO device that data is read from
IOStat is the offset from the base address to a register indicating which of two ports caused the interrupt
IOPort1Active is a bit mask indicating if the first port caused the interrupt (otherwise it is assumed that the second port caused the interrupt)
IOPort, IOPort2 are offsets to the two data registers to be read. Reading a data register disables the interrupt for that port
R9 points to the memory location that data from the first port is being transferred to
R10 points to the memory location that data from the second port is being transferred to
R11 and R12 point to the last address to transfer to (R11 for the first port, R12 for the second)

Again, byte transfers can be made by replacing the load instructions with load byte instructions, and transfers from memory to and IO device are made by swapping the addressing modes between the conditional load instructions and the conditional store instructions.
4.6.4 Interrupt prioritisation

This code dispatches up to 32 interrupt source to their appropriate handler routines. This code is intended to use the normal interrupt vector, and so should be branched to from location 0x18.

External hardware is used to prioritise the interrupt and present the number of the highest-priority active interrupt in an IO register.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IntBase</td>
<td>holds the base address of the interrupt handler</td>
</tr>
<tr>
<td>IntLevel</td>
<td>holds the offset (from IntBase) of the register containing the highest-priority active interrupt</td>
</tr>
<tr>
<td>R13</td>
<td>is assumed to point to a small (60 byte) full descending stack</td>
</tr>
</tbody>
</table>

Interrupts are enabled after 10 instructions (including the branch to this code)

```assembly
; first save the critical state
SUB r14, r14, #4 ; adjust return address before saving it
STMFD r13!, {r12, r14} ; stack return address and working register
MRS r12, SPSR ; get the SPSR ...
STMFD r13!, {r12} ; ... and stack that too
; now get the priority level of the highest priority active interrupt
MOV r12, #IntBase ; get interrupt controller's base address
LDR r12, [r12, #IntLevel] ; get the interrupt level (0 to 31)
; now read-modify-write the CPSR to enable interrupts
MRS r14, CPSR ; read the status register
BIC r14, r14, #0x40 ; clear the F bit (use 0x80 for the I bit)
MSR CPSR, r14 ; write it back to re-enable interrupts
; jump to the correct handler
LDR PC, [PC, r12, LSL #2] ; and jump to the correct handler
; PC base address points to this
; instruction + 8
NOP ; pad so the PC indexes this table
; table of handler start addresses
DCD Priority0Handler
DCD Priority1Handler ........
Priority0Handler
STMFD r13!, {r0 - r11} ; save working registers
; insert handler code here
 ..........
LDMFD r13!, {r0 - r12} ; restore the working registers and the SPSR
orr r14, r14, #0x40 ; set the F bit (use 0x80 for the I bit)
msr CPSR, r14 ; write it back to disable interrupts
msr SPSR, r12 ; stick the SPSR back
ldmfd r13!, {r12, PC}^ ; restore last working register and return
Priority1Handler
 ..........
4.6.5 Context switch

This code performs a context switch on the user mode process. The code is based around a list of pointers to process control blocks (PCBs) of processes that are ready to run.

The pointer to the PCB of the next process to run is pointed to by R12, and the end of the list has a zero pointer.

R13 is a pointer to the PCB, and is preserved between timeslices (so that on entry R13 points to the PCB of the currently running process).

The code assumes the layout of the PCBs, as shown in Figure 4-1: PCB layout.

```
STMIA r13, {r0 - r14}^ ; dump user registers above r13
MSR r0, SPSR ; pick up the user status
STMDB r13, {r0, r14} ; and dump with return address below
LDR r13, [r12], #4 ; load next process info pointer
CMP r13, #0 ; if it is zero, it is invalid
LDMNEDB r13, {r0, r14} ; pick up status and return address
MRSNE SPSR, r0 ; restore the status
LDMNEIA r13, {r0 - r14}^ ; get the rest of the registers
MOVNES pc, r14 ; and return and restore CPSR
; insert "no next process code" here
```
The 26-bit Architectures
This chapter describes the differences between 32-bit and 26-bit architectures.

5.1 Introduction 5-2
5.2 Format of Register 15 5-3
5.3 Writing just the PSR in 26-bit architectures 5-4
5.5 Address Exceptions 5-6
5.4 26-bit PSR Update Instructions 5-5
5.6 Backwards Compatibility from 32-bit Architectures 5-7
5.1 Introduction

ARM architecture versions 1, 2 and 2a are earlier versions of the ARM architecture which implement only a 26-bit address bus, and are known as 26-bit architectures. ARM architecture versions 3 and 4 implement a 32-bit address space (and are known as 32-bit architectures). For backwards compatibility, they also implement the 26-bit address space (except Version 3G). Implementation of a backwards-compatible 26-bit address space on ARM architecture version 4 is optional.

There are several differences between the 26-bit and the 32-bit architectures:

- **Program counter**: The 26-bit architectures implement only a 24-bit program counter in register 15, which allows 64Mbytes of program space. The 32-bit architectures have a 30-bit program counter in register 15 which allows 4Gbytes of program space on 32-bit architectures.

- **Processor modes**: Only four processor modes are supported on 26-bit architectures:
  - User (0b00)
  - FIQ (0b01)
  - IRQ (0b10)
  - Supervisor (0b11)

- **Register 15**: In the 26-bit architectures, the following are also stored in register 15:
  - Four condition flags (N, Z, C and V)
  - The interrupt disable flags (I and F)
  - Two processor mode bits (M1 and M0)

- **CPSR/SPSR**: The 26-bit architectures do not have a CPSR or any SPSRs.

- **Exceptions**: An exception (called an address exception) is raised if a memory access instruction uses an address that is greater than $2^{26} - 1$ bytes.

Together, these differences make up the fundamental distinction between 26-bit and 32-bit architectures:

- **26-bit architectures**: all process status (namely the condition flags, interrupt status and processor mode) can be preserved across subroutine calls and nested exceptions without adding any instructions to the entry or exit sequence.

- **32-bit architectures**: give up this functionality to allow 32-bit instruction addresses to be used.
### 5.2 Format of Register 15

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Z</td>
<td>C</td>
<td>V</td>
<td>I</td>
<td>F</td>
<td></td>
<td></td>
<td>M</td>
</tr>
</tbody>
</table>

Bits[25:2] are collectively known as the Program Counter. Because the Program Counter occupies only 24 bits of register 15, only \(2^{24}\) instructions (\(2^{26}\) bytes) can be addressed, giving a maximum addressable program size of 64Mbytes.

Bits[31:26] and bits[1:0] are collectively known as the Program Status Register or PSR.

The N, Z, C, V, I, and F bits have the same meaning in both 26-bit and 32-bit architectures.

M[1:0] also have the same meaning in both architectures.

Abort mode and Undef mode are not supported in 26-bit architectures.

Aborts and undefined instruction exceptions have exactly the same actions in both modes, except that in 26-bit architectures, Supervisor mode is entered instead of Abort or Undef mode.

The I, F and M[1:0] bits cannot be written directly when the processor is in User mode; in User mode they are only changed by an exception occurring.

#### 5.2.1 Reading register 15

In 26-bit architectures, the value of register 15 is read in five different ways.

1. Most importantly, if register 15 has an unpredictable value in the 32-bit architecture, it also has an unpredictable value when used in the same way in the 26-bit architecture.
2. If register 15 is specified in bits[19:16] of an instruction (and its value is not unpredictable), only the Program Counter (bits[25:2]) are used; all other bits read as zero.
3. If register 15 is specified in bits[3:0] of an instruction (and its value is not unpredictable), all 32 bits are used.
4. If register 15 is stored using STR or STM, the value of the program counter (bits[25:0]) is IMPLEMENTATION DEFINED, but all 32 bits of register are stored.
5. All 32 bits are stored in the Link register (R14) after a Branch with Link instruction or an exception entry.
5.2.2 Writing register 15

In 26-bit architectures the value of register 15, is written in three different ways:

1. Data-processing instructions without the S bit set, Load, and Load Multiple only write the PC part of register 15, and leave the PSR part unchanged.

2. Data-processing instructions with the S bit set and Load Multiple with restore PSR write the PC and the PSR part of register 15.

3. Variants of the CMP, CMN, TST and TEQ instructions write just the PSR part of register 15, and leave the PC part unchanged. These instruction variants are described below.

These read/write rules mean that register 15 is used in three basic ways. It is used as:

- The Rn specifier in data-processing instructions, and as the base address for load and store instructions; only the value of the program counter is used, to simplify PC-relative addressing and position-independent code.

- The Rm specifier in data-processing operands to allow all process status to be restored after a subroutine call or exception by subroutine-return instructions like MOVS PC, LR and LDM..., PC\. These instructions are unpredictable in user mode on 32-bit architectures, but are legal on 26-bit architectures, as they are used preserve the condition code values across procedure calls.

- The value saved in the Link register to preserve the Program Counter and the PSR across subroutine calls and exceptions.

5.3 Writing just the PSR in 26-bit architectures

On 26-bit architectures, the MSR and MRS instructions are not supported. Instead, variants of the CMP, CMN, TST and TEQ instructions are used to write just the PSR part of register 15. These variants are called CMPP, CMNP, TSTP and TEQP, and are distinguished by having instruction bits[15:12] equal to 0b1111 (these bits are usually set to zero for these instructions).

These instructions write their ALU result directly to the PSR part of register 15 (only N, Z, C and V are affected in User mode).
5.4 26-bit PSR Update Instructions

TST{<cond>}P Rn, <shifter_operand>
TEQ{<cond>}P Rn, <shifter_operand>
CMP{<cond>}P Rn, <shifter_operand>
CMN{<cond>}P Rn, <shifter_operand>

Description
The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

The TSTP, TEQP, CMPP and CMNP are 26-bit-only instructions and are used to write the PSR part of register 15 without affecting the PC part of register 15. When the processor is in user mode, only the condition codes will be affected; all other modes allow all PSR bits to be altered.

<table>
<thead>
<tr>
<th>31</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>0 0</td>
<td>I</td>
<td>0</td>
<td>opc</td>
<td>1</td>
<td>Rn</td>
<td>SBO</td>
<td>shifter_operand</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation
if ConditionPassed(<cond>) then
    case <opc> of
        0b00 /* TSTP */
        <alu_out> = Rn AND <shifter_operand>
        0b01 /* TEQP */
        <alu_out> = Rn EOR <shifter_operand>
        0b10 /* CMP */
        <alu_out> = Rn - <shifter_operand>
        0b11 /* CMN */
        <alu_out> = Rn + <shifter_operand>
    endcase
    if R15[1:0] == 0b00 then /* M[1:0] == 0b00, user mode */
    else /* a privileged mode */
        R15[31:26] = <alu_out>[31:26] /* update NZCVIF and ... */
        R15[1:0] = <alu_out>[1:0] /* ... update M[1:0] */

Exceptions
None

Qualifiers
Condition Code

Notes
The I bit: Bit 25 is used to distinguish the immediate and register forms of <shifter_operand>. See the data processing instructions for the types of shifter operand.
5.5 Address Exceptions

On 26-bit architectures, all data addresses are checked to ensure that they are between 0 and 64 Megabytes (26-bit). If a data address is produced with a 1 in any of the top 6 bits, an address exception is generated. When an address exception is generated, the following actions are performed:

\[
\begin{align*}
R14\_svc &= \text{address of address exception generating instruction} + 4 \\
\text{CPSR}[4:0] &= 0b00011 ; \text{Supervisor mode} \\
\text{CPSR}[7] &= 1 ; \text{(Normal) Interrupts disabled} \\
\text{PC} &= 0x14
\end{align*}
\]

The address of the instruction which caused the address exception is the value in register 14 minus 8.

Returning from an address exception

As this exception implies a programming error, it is not usual to return from address exceptions, but if a return is required, use:

\[
\text{SUBS PC, R14, #8}
\]

This restores both the PC and PSR (from R14\_svc) and returns to the instruction that generated the address exception.
5.6 Backwards Compatibility from 32-bit Architectures

As well as the six (seven in Version 4) 32-bit processor modes, ARM Architecture Version 3 (but not 3G), and 4 (optionally on 4T) implement the four 26-bit processor modes described above, including the register 15 format shown. This allows backwards compatibility for the older 26-bit programs by executing those programs in a 26-bit mode. If the backwards-compatibility support is not implemented, CPSR bit 4 (M[4]) always reads as 1, and all writes are ignored.

The complete list of processor modes is shown in Table 5-1: 32-bit and 26-bit modes.

<table>
<thead>
<tr>
<th>M[4:0]</th>
<th>Mode</th>
<th>Accessible Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00000</td>
<td>User_26</td>
<td>R0 to R14, PC, (CPSR)</td>
</tr>
<tr>
<td>0b00001</td>
<td>FIQ_26</td>
<td>R0 to R7, R8_fiq to R14_fiq, PC, (CPSR, SPSR_fiq)</td>
</tr>
<tr>
<td>0b00010</td>
<td>IRQ_26</td>
<td>R0 to R12, R13_irq, R14_irq, PC, (CPSR, SPSR_irq)</td>
</tr>
<tr>
<td>0b00011</td>
<td>SVC_26</td>
<td>R0 to R12, R13_svc, R14_svc, PC, (CPSR, SPSR_svc)</td>
</tr>
<tr>
<td>0b10000</td>
<td>User_32</td>
<td>R0 to R14, PC, CPSR</td>
</tr>
<tr>
<td>0b10001</td>
<td>FIQ_32</td>
<td>R0 to R7, R8_fiq to R14_fiq, PC, CPSR, SPSR_fiq</td>
</tr>
<tr>
<td>0b10010</td>
<td>IRQ_32</td>
<td>R0 to R12, R13_irq, R14_irq, PC, CPSR, SPSR_irq</td>
</tr>
<tr>
<td>0b10011</td>
<td>SVC_32</td>
<td>R0 to R12, R13_svc, R14_svc, PC, CPSR, SPSR_svc</td>
</tr>
<tr>
<td>0b10111</td>
<td>Abort_32</td>
<td>R0 to R12, R13_abt, R14_abt, PC, CPSR, SPSR_abt</td>
</tr>
<tr>
<td>0b11011</td>
<td>Undef_32</td>
<td>R0 to R12, R13_und, R14_und, PC, CPSR, SPSR_und</td>
</tr>
<tr>
<td>0b11111</td>
<td>System_32</td>
<td>R0 to R14, PC, CPSR, (Architecture Version 4 only)</td>
</tr>
</tbody>
</table>

*Table 5-1: 32-bit and 26-bit modes*
5.6.1 32-bit and 26-bit configuration

ARM Architecture Version 3, 3M and 4 (but not 3G or 4T) optionally incorporate two signals that control 32-bit instruction accesses and 32-bit data accesses. The signals are mapped to two bits in register 1 of the system control coprocessor. These signals are:

- PROG32
- DATA32

32-bit configuration

1 If PROG32 is active, the processor switches to a 32-bit mode when processing exceptions (including Reset), using the _32 modes for handling all exceptions. This is called a 32-bit configuration. Abort_32 mode is used for handling memory aborts, and Undef_32 for handling undefined instruction exceptions. A 26-bit mode can be selected by putting a 26-bit mode number into the M[4:0] bits of the CPSR (either using MSR or an exception return sequence). A 32-bit mode can also be entered from 26-bit mode using the MSR instruction. Once in a 26-bit mode, another 26-bit mode can be entered using one of the TEQP, TSTP, CMPP and CMNP instructions, or the MSR instruction.

If an exception occurs when the processor is in a 26-bit mode, only the PC bits from R15[25:2] are copied to the link register; the remaining bits in the link register are zeroed. The PSR bits from R15[31:26] and R15[1:0] are copied into the SPSR, ready for a normal 32-bit return sequence.

2 If PROG32 is active, and DATA32 is not active (32-bit programs with 26-bit data), the result is unpredictable.

26-bit configuration

1 If PROG32 is not active, the processor is locked into 26-bit modes (cannot be placed into a 32-bit mode by any means) and handles exceptions in 26-bit modes. This is called a 26-bit configuration. In this configuration, TEQP, TSTP, CMPP and CMNP instructions, or the MSR instruction can be used to switch to 26-bit mode. Attempts to write CPSR bits[4:2] (M[4:2]) are ignored, stopping any attempts to switch to a 32-bit mode, and SVC_26 mode is used to handle memory aborts and undefined instruction exceptions. The program counter is limited to 24 bits, limiting the addressable program memory to 64 Megabytes.

2 If PROG32 is not active, DATA32 has the following actions.

a) If DATA32 is not active, all data addresses are checked to ensure that they are between 0 and 64 Megabytes (26 bit). If a data address is produced with a 1 in any of the top 6 bits, an address exception is generated.

b) If DATA32 is active, full 32-bit addresses may be produced and are not checked for address exceptions. This allows 26-bit programs to access 32-bit data.
5.6.2 Vector exceptions

When the processor is in a 32-bit configuration (PROG32 is active) and in a 26-bit mode (CPSR[4] == 0), data access (but not instruction fetches) to the hard vectors (address 0x0 to 0x1f) cause a data abort, known as a vector exception.

Vector exceptions are always produced if the hard vectors are written in a 32-bit configuration and a 26-bit mode, and it is IMPLEMENTATION DEFINED whether reading the hard vectors in a 32-bit configuration and a 26-bit mode also causes a vector exception.

Vector exceptions are provided to support 26-bit backwards compatibility. When a vector exception is generated, it indicates that a 26-bit mode process is trying to install a (26-bit) vector handler. Because the processor is in a 32-bit configuration, exceptions will be handled in a 32-bit mode, so a veneer must be used to change from the 32-bit exception mode to a 26-bit mode before calling the 26-bit exception handler. This veneer may be installed on each vector, and can switch to a 26-bit mode before calling any 26-bit handlers.

The 26-bit exception handler's return may also need to be veneered. Some SWI handlers return status information in the processor flags, and this information will need to be transferred from the link register to the SPSR with a return veneer for the SWI handler.
The Thumb Instruction Set
The Thumb Instruction Set

The Thumb instruction set is a subset of the ARM instruction set. Thumb is designed to increase the performance of ARM implementations that use a memory data bus, and to allow better code density than ARM. The ARMv4T architecture incorporates both a full 32-bit ARM instruction set and the 16-bit Thumb instruction set. Every Thumb instruction can be encoded in 16 bits.

This chapter lists every Thumb instruction, and gives information on its format and encoding.

6.1 Using this Chapter 6-2
6.2 Introduction to Thumb 6-3
6.3 Instruction Set Overview 6-4
6.4 Branch Instructions 6-5
6.5 Data-processing Instructions 6-7
6.6 Load and Store Register Instructions 6-12
6.7 Load and Store Multiple Instructions 6-14
6.8 Alphabetical List of Thumb Instructions 6-19
The Thumb Instruction Set

6.1 Using this Chapter

This chapter is divided into three parts:
1 Introduction to Thumb
2 Overview of the Thumb instruction types
3 Alphabetical list of instructions

6.1.1 Introduction to Thumb (page 6-3 through 6-4)

This part describes the Thumb concepts and how it fits in with ARM instruction execution.

6.1.2 Overview of the Thumb instruction types (page 6-5 through 6-15)

This part describes the functional groups within the Thumb instruction set, and shows relevant examples and encodings. Each functional group lists all its instructions, which you can then find in the alphabetical section. The functional groups are:
1 Branch Instructions
2 Data-processing Instructions
3 Load and Store Register Instructions
4 Load and Store Multiple Instructions

6.1.3 Alphabetical list of instructions (page 6-19 through 6-82)

This part lists every Thumb instruction in alphabetical order, and gives:
• instruction syntax and functional group
• encoding and operation
• relevant exceptions and qualifiers
• notes on usage

Where relevant, the instruction descriptions show the equivalent ARM instruction and encoding.
6.2 Introduction to Thumb

Thumb does not alter the underlying structure of the ARM architecture; it merely presents restricted access to the ARM architecture. All Thumb data-processing instructions operate on full 32-bit values, and full 32-bit bit addresses are produced by both data-access instructions and for instruction fetches.

When the processor is executing Thumb, eight general-purpose integer registers are available, R0 to R7, which are the same physical registers as R0 to R7 when executing ARM. 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 (known 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 in the ARM MSR and MRS instructions). 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 4) and are executed as ARM instructions
- T == 1 16-bit instructions are fetched from memory (and the PC is incremented by two) and are executed as Thumb instructions

6.2.1 Entering Thumb state

Thumb execution is normally entered by executing an ARM BX instruction (Branch and eXchange instruction set). 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.

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 and Restore CPSR instruction). This allows an operating system to automatically restart a process independently 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.

6.2.2 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 same return instruction performs correctly, regardless of whether the exception occurred during ARM or Thumb execution.

Table 6-1: Exception return instructions lists the values of the exception mode LR for exceptions generated during Thumb execution.
The Thumb Instruction Set

### Exception Link Register Value

<table>
<thead>
<tr>
<th>Exception</th>
<th>Exception Link Register Value</th>
<th>Return Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>Unpredictable value</td>
<td>MOVPC, R14</td>
</tr>
<tr>
<td>Undefined</td>
<td>Address of Undefined instruction + 2</td>
<td>MOVPC, R14</td>
</tr>
<tr>
<td>SWI</td>
<td>Address of SWI instruction + 2</td>
<td>MOVPC, R14</td>
</tr>
<tr>
<td>Prefetch Abort</td>
<td>Address of aborted instruction fetch + 4</td>
<td>SUBS PC, R14, #4</td>
</tr>
<tr>
<td>Data Abort</td>
<td>Address of the instruction that generated the abort + 8</td>
<td>SUBS PC, R14, #8</td>
</tr>
<tr>
<td>IRQ</td>
<td>Address of the next instruction to be executed + 4</td>
<td>SUBS PC, R14, #4</td>
</tr>
<tr>
<td>FIQ</td>
<td>Address of the next instruction to be executed + 4</td>
<td>SUBS PC, R14, #4</td>
</tr>
</tbody>
</table>

#### Table 6-1: Exception return instructions

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>op</td>
<td>Rm</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>op</td>
<td>immediate</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>op</td>
<td>immediate</td>
<td>Rd</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>opcode</td>
<td>Rm</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>opcode</td>
<td>H1</td>
<td>H2</td>
<td>Rm</td>
<td>Rd</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rd</td>
<td>PC-relative offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>L B</td>
<td>0</td>
<td>Rm</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>H S</td>
<td>1</td>
<td>Rm</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>B L</td>
<td>immediate</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>L</td>
<td>immediate</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>L</td>
<td>Rd</td>
<td>SP-relative offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>SP</td>
<td>Rd</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>SBZ</td>
<td>0</td>
<td>SBZ</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>L 1</td>
<td>SBZ</td>
<td>R</td>
<td>register_list</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>L</td>
<td>Rn</td>
<td>register_list</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>cond</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>SWI number</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x x x x x x x x x x x x x x</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Figure 6-1: The Thumb instruction set (expanded)

6.3 Instruction Set Overview

---

ARM Architecture Reference Manual
ARM DUI 0100B
6.4 Branch Instructions

Thumb supports four types of branch instruction:

- an unconditional branch that allows a forward or backward branch of up to 2 Kbytes
- a conditional branch to allow forward and backward branches of up to 256 bytes
- a branch and link (subroutine call) is supported with a pair of instructions that allow forward and backwards branches of up to 4 Mbytes
- a branch and exchange instruction branches to an address in a register and optionally switches to ARM code execution.

6.4.1 Encoding

The encoding for these formats is given below:

**Format 1**

\[
\text{B<cond> <target_address>}
\]

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>con</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8 bit signed offset

**Format 2**

\[
\text{B <target_address>}
\]

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

11 bit signed offset

**Format 3**

\[
\text{BL <target_address>}
\]

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>H</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

11 bit signed offset

**Format 4**

\[
\text{BX Rm}
\]

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>H2</td>
<td>Rm</td>
<td>SBZ</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Thumb Instruction Set

6.4.2 Examples

B label ; unconditionally branch to label
BCC label ; branch to label if carry flags is clear
BEQ label ; branch to label if zero flag is set
BL func ; subroutine call to function

func .

MOV PC, LR ; R15=R14, return to instruction after the BL

BX R12 ; branch to address in R12; begin Thumb execution if
; bit 0 of R12 is zero; otherwise continue executing
; Thumb code

6.4.3 List of branch instructions

The following instructions follow the formats shown above.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Conditional branch</td>
<td>6-30</td>
</tr>
<tr>
<td>B</td>
<td>Unconditional branch</td>
<td>6-31</td>
</tr>
<tr>
<td>BL</td>
<td>Branch with Link</td>
<td>6-33</td>
</tr>
<tr>
<td>BX</td>
<td>Branch and exchange instruction set</td>
<td>6-34</td>
</tr>
</tbody>
</table>
The Thumb Instruction Set

6.5 Data-processing Instructions

Thumb data-processing instructions are a subset of the ARM data-processing instructions, as shown below.

All Thumb data-processing instructions set the condition codes.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV Rd, #0 to 255</td>
<td>Move</td>
<td>Rd := 8-bit immediate</td>
</tr>
<tr>
<td>MVN Rd, Rm</td>
<td>Move Not</td>
<td>Rd := NOT Rm</td>
</tr>
<tr>
<td>ADD Rd, Rn, Rm</td>
<td>Add</td>
<td>Rd := Rn + Rm</td>
</tr>
<tr>
<td>ADD Rd, Rn, #0 to 7</td>
<td>Add</td>
<td>Rd := Rn + 3-bit immediate</td>
</tr>
<tr>
<td>ADD Rd, #0 to 255</td>
<td>Add</td>
<td>Rd := Rd + 8-bit immediate</td>
</tr>
<tr>
<td>ADC Rd, Rm</td>
<td>Add with Carry</td>
<td>Rd := Rd + Rm + Carry Flag</td>
</tr>
<tr>
<td>SUB Rd, Rn, Rm</td>
<td>Subtract</td>
<td>Rd := Rn - Rm</td>
</tr>
<tr>
<td>SUB Rd, Rn, #0 to 7</td>
<td>Subtract</td>
<td>Rd := Rn - 3-bit immediate</td>
</tr>
<tr>
<td>SUB Rd, #0 to 255</td>
<td>Subtract</td>
<td>Rd := Rd - 8-bit immediate</td>
</tr>
<tr>
<td>SBC Rd, Rm</td>
<td>Subtract with Carry</td>
<td>Rd := Rd - Rm - NOT(Carry Flag)</td>
</tr>
<tr>
<td>NEG Rd, Rm</td>
<td>Negate</td>
<td>Rd := 0 - Rm</td>
</tr>
<tr>
<td>AND Rd, Rm</td>
<td>Logical AND</td>
<td>Rd := Rd AND Rm</td>
</tr>
<tr>
<td>EOR Rd, Rm</td>
<td>Logical Exclusive OR</td>
<td>Rd := Rd EOR Rm</td>
</tr>
<tr>
<td>ORR Rd, Rm</td>
<td>Logical (inclusive) OR</td>
<td>Rd := Rd OR Rm</td>
</tr>
<tr>
<td>BIC Rd, Rm</td>
<td>Bit Clear</td>
<td>Rd := Rd AND NOT Rm</td>
</tr>
<tr>
<td>CMP Rn, #0 to 255</td>
<td>Compare</td>
<td>update flags after Rn - 8-bit immediate</td>
</tr>
<tr>
<td>CMP Rn, Rm</td>
<td>Compare</td>
<td>update flags after Rn - Rm</td>
</tr>
<tr>
<td>CMN Rn, Rm</td>
<td>Compare Negated</td>
<td>update flags after Rn + Rm</td>
</tr>
<tr>
<td>TST Rn, Rm</td>
<td>Test</td>
<td>update flags after Rn AND Rm</td>
</tr>
<tr>
<td>MUL Rd, Rs</td>
<td>Multiply</td>
<td>Rd := Rs x Rd</td>
</tr>
<tr>
<td>LSL Rd, Rm, #0 to 31</td>
<td>Logical Shift Left</td>
<td>Rd := Rm LSL 5-bit immediate</td>
</tr>
<tr>
<td>LSL Rd, Rs</td>
<td>Logical Shift Left</td>
<td>Rd := Rd LSL Rs</td>
</tr>
<tr>
<td>LSR Rd, Rm, #0 to 31</td>
<td>Logical Shift Right</td>
<td>Rd := Rm LSR 5-bit immediate</td>
</tr>
<tr>
<td>LSR Rd, Rs</td>
<td>Logical Shift Right</td>
<td>Rd := Rd LSR Rs</td>
</tr>
<tr>
<td>ASR Rd, Rm, #0 to 31</td>
<td>Arithmetic Shift Right</td>
<td>Rd := Rm ASR 5-bit immediate</td>
</tr>
<tr>
<td>ASR Rd, Rs</td>
<td>Arithmetic Shift Right</td>
<td>Rd := Rd ASR Rs</td>
</tr>
<tr>
<td>ROR Rd, Rs</td>
<td>Rotate Right</td>
<td>Rd := Rd ROR Rs</td>
</tr>
</tbody>
</table>

Table 6-2: Thumb data-processing instructions
The Thumb Instruction Set

Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R0, R4, R7</td>
<td>; R0 = R4 + R7</td>
<td></td>
</tr>
<tr>
<td>SUB R6, R1, R2</td>
<td>; R6 = R1 - R2</td>
<td></td>
</tr>
<tr>
<td>ADD R0, #255</td>
<td>; R0 = R0 + 255</td>
<td></td>
</tr>
<tr>
<td>ADD R1, R4, #4</td>
<td>; R1 = R4 + 4</td>
<td></td>
</tr>
<tr>
<td>NEG R3, R1</td>
<td>; R3 = 0 - R1</td>
<td></td>
</tr>
<tr>
<td>AND R2, R5</td>
<td>; R2 = R2 AND R5</td>
<td></td>
</tr>
<tr>
<td>EOR R1, R6</td>
<td>; R1 = R1 EOR R6</td>
<td></td>
</tr>
<tr>
<td>CMP R2, R3</td>
<td>; update flags after R2 - R3</td>
<td></td>
</tr>
<tr>
<td>CMP R7, #100</td>
<td>; update flags after R7 - 100</td>
<td></td>
</tr>
<tr>
<td>MOV R0, #200</td>
<td>; R0 = 200</td>
<td></td>
</tr>
</tbody>
</table>

6.5.1 High registers

There are seven types of data-processing instruction which operate on ARM registers 8 to 14 and the PC (called the high registers).

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV Rd, Rn</td>
<td>Move</td>
<td>Rd := Rn</td>
</tr>
<tr>
<td>ADD Rd, Rn</td>
<td>Add</td>
<td>Rd := Rd + Rm</td>
</tr>
<tr>
<td>CMP Rn, Rm</td>
<td>Compare</td>
<td>update flags after Rn - Rm</td>
</tr>
<tr>
<td>ADD SP, SP, #0 to 511</td>
<td>Increment stack pointer</td>
<td>R13 = R13 + #9-bit immediate</td>
</tr>
<tr>
<td>SUB SP, SP, #0 to 511</td>
<td>Decrement stack pointer</td>
<td>R13 = R13 - #9-bit immediate</td>
</tr>
<tr>
<td>ADD Rd, SP, #0 to 1020</td>
<td>Form Stack address</td>
<td>Rd = R13 + #10-bit immediate</td>
</tr>
<tr>
<td>ADD Rd, PC, #0 to 1020</td>
<td>Form PC address</td>
<td>Rd = PC + #10-bit immediate</td>
</tr>
</tbody>
</table>

Table 6-3: High register data-processing instructions

Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0, R12</td>
<td>; R0 = R12</td>
<td></td>
</tr>
<tr>
<td>ADD R10, R1, R2</td>
<td>; R6 = R1 - R2</td>
<td></td>
</tr>
<tr>
<td>MOV PC, LR</td>
<td>; PC = R14</td>
<td></td>
</tr>
<tr>
<td>CMP R10, R11</td>
<td>; update flags after R10 - R11</td>
<td></td>
</tr>
<tr>
<td>SUB SP, SP, #10</td>
<td>; increase stack size by 100 bytes</td>
<td></td>
</tr>
<tr>
<td>ADD SP, SP, #16</td>
<td>; decrease stack size by 16 bytes</td>
<td></td>
</tr>
<tr>
<td>ADD R2, SP, #20</td>
<td>; R2 = SP + 20</td>
<td></td>
</tr>
<tr>
<td>ADD R0, PC, #500</td>
<td>; R0 = PC + 500</td>
<td></td>
</tr>
</tbody>
</table>
6.5.2 Formats

Data-processing instructions perform an operation on the general processor registers:

**Format 1**

<opcode1> Rd, Rn, Rm
<opcode1> := ADD | SUB

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>op_1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rd</td>
</tr>
</tbody>
</table>
```

**Format 2**

<opcode2> Rd, Rn, <3_bit_immed>
<opcode2> := ADD | SUB

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>op_2</td>
<td>3_bit_immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rd</td>
</tr>
</tbody>
</table>
```

**Format 3**

<opcode3> Rd|Rn, #<8_bit_immed>
<opcode3> := ADD | SUB | MOV | CMP

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>op_3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8_bit_immediate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rd</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Format 4**

<opcode4> Rd, Rn #<shift_imm>
<opcode4> := LSL | LSR | ASR

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>op_4</td>
<td>shift_imm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Format 5**

<opcode5> Rd | Rn, Rm | Rs
<opcode5> := MVN| CMP| CMN| TST| ADC| SBC| NEG| MUL| LSL| LSR| ASR| ROR| AND| EOR| ORR| BIC

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>op_5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rm</td>
<td>Rs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
The Thumb Instruction Set

Format 6

\[ \text{ADD } R_d, \langle \text{reg} \rangle, \#<8\text{-bit_immed}> \]
\[ \langle \text{reg} \rangle := \text{SP} | \text{PC} \]

\[\begin{array}{cccccc}
1 & 0 & 1 & 0 & \text{reg} & \text{Rd} & 8\_\text{bit_immed} \\
\end{array}\]

Format 7

\[\langle \text{opcode6} \rangle \text{ SP, SP, } \#<7\_\text{bit_immed}> \]
\[\langle \text{opcode6} \rangle := \text{ADD} | \text{SUB} \]

\[\begin{array}{ccccccccc}
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & \text{op}_6 & 7\_\text{bit_immed} \\
\end{array}\]

6.5.3 List of data-processing instructions

The following instructions follow the formats shown above.

- **ADC**: Add with Carry (register)  
  - Page 6-19
- **ADD**: Add (immediate)  
  - Page 6-20
- **ADD**: Add (large immediate)  
  - Page 6-21
- **ADD**: Add (register)  
  - Page 6-22
- **ADD**: Add high registers  
  - Page 6-23
- **ADD**: Add (immediate to program counter)  
  - Page 6-24
- **ADD**: Add (immediate to stack pointer)  
  - Page 6-25
- **ADD**: Increment stack pointer  
  - Page 6-26
- **AND**: Logical AND  
  - Page 6-27
- **ASR**: Arithmetic shift right (immediate)  
  - Page 6-28
- **ASR**: Arithmetic shift right (register)  
  - Page 6-29
- **BIC**: Bit clear  
  - Page 6-32
- **CMN**: Compare negative (register)  
  - Page 6-35
- **CMP**: Compare (immediate)  
  - Page 6-36
- **CMP**: Compare (register)  
  - Page 6-37
- **CMP**: Compare high registers  
  - Page 6-38
- **EOR**: Exclusive OR  
  - Page 6-39
- **LSL**: Logical shift left (immediate)  
  - Page 6-52
- **LSL**: Logical shift left (register)  
  - Page 6-53
- **LSR**: Logical shift right (immediate)  
  - Page 6-54
- **LSR**: Logical shift right (register)  
  - Page 6-55
## The Thumb Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>Move (immediate)</td>
<td>6-56</td>
</tr>
<tr>
<td>MOV</td>
<td>Move high registers</td>
<td>6-57</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply</td>
<td>6-58</td>
</tr>
<tr>
<td>MVN</td>
<td>Move NOT (register)</td>
<td>6-59</td>
</tr>
<tr>
<td>NEG</td>
<td>Negate (register)</td>
<td>6-60</td>
</tr>
<tr>
<td>ORR</td>
<td>Logical OR</td>
<td>6-61</td>
</tr>
<tr>
<td>ROR</td>
<td>Rotate right (register)</td>
<td>6-66</td>
</tr>
<tr>
<td>SBC</td>
<td>Subtract with Carry (register)</td>
<td>6-67</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract (immediate)</td>
<td>6-77</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract (large immediate)</td>
<td>6-78</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract (register)</td>
<td>6-79</td>
</tr>
<tr>
<td>SUB</td>
<td>Decrement stack pointer</td>
<td>6-80</td>
</tr>
<tr>
<td>TST</td>
<td>Test (register)</td>
<td>6-82</td>
</tr>
</tbody>
</table>
The Thumb Instruction Set

6.6 Load and Store Register Instructions

Thumb supports 8 types of load and store register instructions. Two basic addressing modes are available:

- register plus register
- register plus 5-bit immediate

If an immediate offset is used, it is scaled by 4 for word access and 2 for halfword accesses. Three special instructions allow a load using the PC as a base with a 1 Kbyte (word-aligned) immediate offset, and a load and store instructions with the stack pointer (R13) as the base and a 1Kbyte (word-aligned) immediate offset.

6.6.1 Formats

Load and Store instructions perform an operation on the general processor registers. Load and store instructions have the following formats:

**Format 1**

\[
\text{<opcode1> Rd, [Rn, #<5_bit_offset>]}
\]

\[
\text{<opcode1> := LDR|LDRH|LDRB|STR|STRH|STRB}
\]

**Format 2**

\[
\text{<opcode2> Rd, [Rn, Rm]}
\]

\[
\text{<opcode2> := LDR|LDRH|LDRSH|LDRB|LDRSB|STR|STRH|STRB}
\]

**Format 3**

\[
\text{LDR Rd, [PC, #<8_bit_offset>]}
\]

**Format 4**

\[
\text{<opcode3> Rd, [SP, #<8_bit_offset>]}
\]

\[
\text{<opcode3> := LDR | STR}
\]
The Thumb Instruction Set

6.6.2 Examples

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
LDRSB R5, [R0, #0]  ; Load signed byte into R5 byte from R0
LDRSH R1, [R2, #10]  ; Load signed halfword to R1 from R2 + 10
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

6.6.3 List of load and store register instructions

The following instructions follow the formats shown above.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR</td>
<td>Load word (immediate offset)</td>
<td>6-42</td>
</tr>
<tr>
<td>LDR</td>
<td>Load word (register offset)</td>
<td>6-43</td>
</tr>
<tr>
<td>LDR</td>
<td>Load word (PC-relative)</td>
<td>6-44</td>
</tr>
<tr>
<td>LDR</td>
<td>Load word (SP-relative)</td>
<td>6-45</td>
</tr>
<tr>
<td>LDRB</td>
<td>Load unsigned byte (immediate offset)</td>
<td>6-46</td>
</tr>
<tr>
<td>LDRB</td>
<td>Load unsigned byte (register offset)</td>
<td>6-47</td>
</tr>
<tr>
<td>LDRH</td>
<td>Load unsigned halfword (immediate offset)</td>
<td>6-48</td>
</tr>
<tr>
<td>LDRH</td>
<td>Load unsigned halfword (register offset)</td>
<td>6-49</td>
</tr>
<tr>
<td>LDRSB</td>
<td>Load signed byte (register offset)</td>
<td>6-50</td>
</tr>
<tr>
<td>LDRSH</td>
<td>Load signed halfword (register offset)</td>
<td>6-51</td>
</tr>
<tr>
<td>STR</td>
<td>Store word (immediate offset)</td>
<td>6-70</td>
</tr>
<tr>
<td>STR</td>
<td>Store word (register offset)</td>
<td>6-71</td>
</tr>
<tr>
<td>STR</td>
<td>Store word (SP-relative)</td>
<td>6-72</td>
</tr>
<tr>
<td>STRB</td>
<td>Store byte (immediate offset)</td>
<td>6-73</td>
</tr>
<tr>
<td>STRB</td>
<td>Store byte (register offset)</td>
<td>6-74</td>
</tr>
<tr>
<td>STRH</td>
<td>Store halfword (immediate offset)</td>
<td>6-75</td>
</tr>
<tr>
<td>STRH</td>
<td>Store halfword (register offset)</td>
<td>6-76</td>
</tr>
</tbody>
</table>
The Thumb Instruction Set

6.7 Load and Store Multiple Instructions

Thumb supports four types of load and store multiple instructions. Two (a load and a store) are designed to support block copy; they have a fixed increment-after addressing mode from a base register.

The other two instructions (called 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 can transfer any or all of the lower 8 registers. PUSH can also stack the return address and POP can load the PC. All four instructions update the base register after the transfer.

6.7.1 Formats

Format 1

\[
\langle \text{opcode1} \rangle \ Rn!, \ <\text{register}\_\text{list}>
\]

\[
\langle \text{opcode1} \rangle \ := \ \text{LDMIA} \mid \text{STMIA}
\]

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>L</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

register_list

Format 2

\[
\text{POP} \ \{<\text{register}\_\text{list}>,<\text{PC}>\}
\]

\[
\text{PUSH} \ \{<\text{register}\_\text{list}>,<\text{LR}>\}
\]

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>L</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

register_list

6.7.2 Examples

LDMIA R7!, (R0 - R3, R5) ; Load R0 to R3 and R5 from R7
; then add 20 to R7
STMIA R0!, (R3, R4, R5) ; Store R3-R5 to R0: add 12 to R0 function
STMFD R13!, (R0-R7, LR) ; save all regs and return address
; code of the function body
.
STMFD R13!, (R0-R7, PC) ; restore all register and return
.

PUSH (R0 - R7, LR) ; push onto the stack (R13) R0 - R7 and
; the return address

POP (R0 - R7, PC) ; restore R0 - R7 from the stack
; and the program counter, and return
6.7.3 List of load and store multiple instructions

The following instructions follow the formats shown above.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDM</td>
<td>Load multiple</td>
<td>6-40</td>
</tr>
<tr>
<td>POP</td>
<td>Pop multiple</td>
<td>6-62</td>
</tr>
<tr>
<td>PUSH</td>
<td>Push multiple</td>
<td>6-64</td>
</tr>
<tr>
<td>STM</td>
<td>Store multiple</td>
<td>6-68</td>
</tr>
</tbody>
</table>
**B (1)**  
**Conditional branch**

**Architecture availability**  
Thumb instructions are available in Architecture v4 only

**Description**  
This form of the B (Branch) instruction provides conditional changes to program flow. In this case, B causes a conditional branch to a target address by shifting the 8-bit signed offset left by one bit, sign-extending the result to a 9-bit value. The instruction therefore specifies a branch of +/- 256 bytes.

The instruction is only executed if the condition specified in the instruction meets the boolean status. The conditions are defined in 3.2 The Condition Field on page 3-3.

**Operation**  
\[
\text{if ConditionPassed}(\text{<cond>}) \text{ then } \text{PC} = \text{PC} + \text{SignExtend}(8\text{-bit signed offset})
\]

**Exceptions**  
None

**Qualifiers and flag settings**  
Condition Code

**Notes**  
Offset calculation: An assembler will calculate the branch target address from the address of the current instruction and the address of the target. When the PC holds the address of the current instruction plus four (because the PC holds the address of the current instruction), the address space is UNPREDICTABLE.

**Memory bounds**: Branching backwards past location zero and forwards over four address space is UNPREDICTABLE.

**Equivalent ARM syntax and encoding**  
\[
\text{B}<\text{cond}>\:<\text{target_address}>
\]

**Encoding**  
\[
\begin{array}{cccccc}
1 & 1 & 0 & 1 & L & \text{<8-bit signed offset>}
\end{array}
\]

**Thumb instructions are available in Architecture v4 only**
ADC Rd, Rm

Description
The ADC (Add with Carry) instruction is used to synthesize 64-bit addition. If register pairs R0,R1 and R2,R3 hold 64-bit values (R0 and R2 hold the least-significant word), the following instructions leave the 64-bit sum in R0,R1:

\[
\begin{align*}
\text{ADD} & \quad R0,R2 \\
\text{ADC} & \quad R1,R3
\end{align*}
\]

The instruction ADC R0,R0 produces a single-bit Rotate Left with Extend operation (33-bit rotate though the carry flag) on R0.

ADC adds the value of register Rd, and the value of the Carry flag, and the value of register Rm, and stores the result in register Rd. The condition code flags are updated (based on the result).

\[
Rd = Rd + Rm + C \text{ Flag}
\]

N Flag = Rd[31]

Z Flag = if Rd == 0 then 1 else 0

C Flag = CarryFrom(Rd + Rm + C Flag)

V Flag = OverflowFrom(Rd + Rm + C Flag)

Exceptions
None

Qualifiers
None

Equivalent ARM syntax and encoding

\[
\begin{array}{c}
\text{ADCS} \quad \text{Rd}, \text{Rd}, \text{Rm} \\
\end{array}
\]

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ARM Architecture Reference Manual
ARM DUI 0100B
ADD Rd, Rn, #<3_bit_immediate>

**Description**

This form of the ADD instruction adds a small constant value to the value of a register and stores the result in a second register.

In this case, ADD adds the value of register Rn and the value of the 3-bit immediate (values 0 to 7), and stores the result in the destination register Rd. The condition code flags are updated (based on the result).

<table>
<thead>
<tr>
<th>Rd</th>
<th>Rn</th>
<th>3_bit_immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Operation**

Rd = Rn + <3_bit_immediate>
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = CarryFrom(Rn + <3_bit_immed>)
V Flag = OverflowFrom(Rn + <3_bit_immed>)

**Exceptions**

None

**Qualifiers**

None

---

**Equivalent ARM syntax and encoding**

ADDS Rd, Rn, #<3_bit_immediate>

<table>
<thead>
<tr>
<th>Rd</th>
<th>Rn</th>
<th>3_bit_immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

---
ADD Rd, #<8_bit_immediate>

**Description**
This form of the ADD instruction is also used to add a large constant value to the value of a register and to store the result back in the same register.

In this case, ADD instruction adds the value of register Rd and the value of the 8-bit immediate (values 0 to 255), and stores the result back in register Rd. The condition code flags are updated (based on the result).

<table>
<thead>
<tr>
<th>Rd</th>
<th>8_bit_immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>15</td>
<td>14 13 12 11 10 9 8 7 0</td>
</tr>
</tbody>
</table>

**Operation**
- \( \text{Rd} = \text{Rd} + \text{<8_bit_immed>} \)
- \( \text{N Flag} = \text{Rd}[31] \)
- \( \text{Z Flag} = \text{if Rd == 0 then 1 else 0} \)
- \( \text{C Flag} = \text{CarryFrom(Rd + <8_bit_immed>)} \)
- \( \text{V Flag} = \text{OverflowFrom(Rd + <8_bit_immed>)} \)

**Exceptions**
None

**Qualifiers**
None

---

**Equivalent ARM syntax and encoding**

\[
\text{ADDS Rd, Rd, #<8_bit_immediate>}
\]

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 12 11 10 9 8 7 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

ARM Architecture Reference Manual
ARM DUI 0100B
ADD Rd, Rn, Rm

**Description**

This form of the ADD instruction adds the value of one register to the value of a second register, and stores the result in a third register.

In this case, ADD adds the value of register Rn and the value of register Rm, and stores the result in the destination register Rd. The condition code flags are updated (based on the result).

\[
Rd = Rn + Rm
\]

N Flag = Rd[31]

Z Flag = if Rd == 0 then 1 else 0

C Flag = CarryFrom(Rn + Rm)

V Flag = OverflowFrom(Rn + Rm)

**Operation**

Rd = Rn + Rm

N Flag = Rd[31]

Z Flag = if Rd == 0 then 1 else 0

C Flag = CarryFrom(Rn + Rm)

V Flag = OverflowFrom(Rn + Rm)

**Exceptions**

None

**Qualifiers**

None

---

**Equivalent ARM syntax and encoding**

ADDS Rd, Rn, Rm

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 16  | 15  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1   | 1   | 0   | 0   | 0   | 0   | 1   | 0   | 0   | 1   | Rn  | Rn  | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | Rm  |

---
ADD Rd, Rm

Description
This form of the ADD instruction is used for addition of values in the high registers. In this case, ADD:
• adds the value of a low register to a high register (H1=1, H2=0), or
• adds the value of a high register to a low register (H1=0, H2=1), or
• adds the value of a high register to another high register (H1=1, H2=1)

The condition code flags are not affected.

Operation
Rd = Rd + Rm

Exceptions
None

Qualifiers
None

Notes
Operand restriction: If a low register is specified for Rd and Rm (H1=0 and H2=0), the result is UNPREDICTABLE.

Equivalent ARM syntax and encoding
ADD Rd, Rd, Rm

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>H1</td>
<td>Rd</td>
<td>H1</td>
<td>Rd</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>H2</td>
<td>Rm</td>
</tr>
</tbody>
</table>

ADD (4) Thumb
Add high registers
Architecture v4T only
ADD Rd, PC, #<8_bit_immediate>)

**Description**
This form of the ADD instruction is used to address a PC-relative (word-sized) variable.

In this case, ADD clears the bottom two bits of the value of the PC and adds the result to the value of the 8-bit immediate (values 0 to 255), and stores the result in register Rd.

**Operation**
Rd = (PC AND 0xfffffffc) + <8_bit_immed>

**Exceptions**
None

**Qualifiers**
None

---

**Equivalent ARM syntax and encoding**

ADD Rd, PC, #<8_bit_immediate>

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 1 1 0 | 0 0 1 0 1 0 0 0 | 1 1 1 1 | Rd | 0 0 0 0 | 8_bit_immediate |
ADD Rd, SP, #<8_bit_immediate>

**Description**
This form of the ADD instruction is used to address an SP-relative (word-sized) variable.

In this case, ADD adds the value of the SP and the value of the 8-bit immediate (values 0 to 255), and stores the result in register Rd.

**Operation**
Rd = SP + <8_bit_immed>

**Exceptions**
None

**Qualifiers**
None

---

**Equivalent ARM syntax and encoding**

```assembly
ADD Rd, SP, #<8_bit_immediate>
```

```
  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 12 11 10 9 8 7 0
0111 0010 0101 0000 1101 0000 0000 0000
```

---

**Description**
This form of the ADD instruction is used to address an SP-relative (word-sized) variable.

In this case, ADD adds the value of the SP and the value of the 8-bit immediate (values 0 to 255), and stores the result in register Rd.

**Operation**
Rd = SP + <8_bit_immed>

**Exceptions**
None

**Qualifiers**
None
**ADD SP, SP, #<7_bit_immediate>**

**Description**
This form of the ADD instruction is used to decrease the size of the stack. In this case, ADD adds the value of the SP and the value of the 7-bit immediate (values 0 to 127), and stores the result back in the SP.

<table>
<thead>
<tr>
<th>Position</th>
<th>Bit Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>7_bit_immediate</td>
</tr>
</tbody>
</table>

**Operation**
SP = SP + <7_bit_immed>

**Exceptions**
None

**Qualifiers**
None

---

**Equivalent ARM syntax and encoding**

<table>
<thead>
<tr>
<th>Position</th>
<th>Bit Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>7_bit_immediate</td>
</tr>
</tbody>
</table>

---

**6-26**

**ARM Architecture Reference Manual**

ARM DUI 0100B
The AND (Logical AND) instruction is most useful for extracting a field from a register, by ANDing the register with a mask value that has 1's in the field to be extracted, and 0's elsewhere.

AND performs a bitwise AND of the value of register Rm with the value of register Rd, and stores the result back in register Rd. The condition code flags are updated (based on the result).

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>The AND (Logical AND) instruction is most useful for extracting a field from a register, by ANDing the register with a mask value that has 1's in the field to be extracted, and 0's elsewhere. AND performs a bitwise AND of the value of register Rm with the value of register Rd, and stores the result back in register Rd. The condition code flags are updated (based on the result).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd = Rd AND Rm</td>
</tr>
<tr>
<td>N Flag = Rd[31]</td>
</tr>
<tr>
<td>Z Flag = if Rd == 0 then 1 else 0</td>
</tr>
<tr>
<td>C Flag = &lt;shifter_carry_out&gt;</td>
</tr>
<tr>
<td>V Flag = unaffected</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Qualifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
</tr>
</tbody>
</table>

Equivalent ARM syntax and encoding

**ANDS Rd, Rd, Rm**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>The AND (Logical AND) instruction is most useful for extracting a field from a register, by ANDing the register with a mask value that has 1's in the field to be extracted, and 0's elsewhere. AND performs a bitwise AND of the value of register Rm with the value of register Rd, and stores the result back in register Rd. The condition code flags are updated (based on the result).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd = Rd AND Rm</td>
</tr>
<tr>
<td>N Flag = Rd[31]</td>
</tr>
<tr>
<td>Z Flag = if Rd == 0 then 1 else 0</td>
</tr>
<tr>
<td>C Flag = &lt;shifter_carry_out&gt;</td>
</tr>
<tr>
<td>V Flag = unaffected</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Qualifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
</tr>
</tbody>
</table>
ASR Rd, Rm, #<shift_imm>

The ASR (Arithmetic Shift Right) instruction is used to provide the signed value of a register divided by a constant power of 2.

In this case, ASR performs an arithmetic shift right of the value of register Rm by an immediate value in the range 1 to 32, and stores the result into the destination register Rd. The sign bit of Rm (Rm[31]) is inserted into the vacated bit positions.

A shift by 32 is encoded by:

<shift_imm> = 0

The condition code flags are updated (based on the result).

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation

if <shift_imm> == 0

  if Rd[31] == 0 then
    C Flag = Rd[31]
    Rd = 0
  else /* Rd[31] == 1 */
    C Flag = Rm[31]
    Rd = 0xffffffff

else /* <shift_imm> > 0 */

  Rd = Rd Arithmetic_Shift_Right <shift_imm>

  N Flag = Rd[31]
  Z Flag = if Rd == 0 then 1 else 0
  V Flag = unaffected

Exceptions  None
Qualifiers   None

Equivalent ARM syntax and encoding

MOVS Rd, Rm, ASR #<shift_imm>
ASR Rd, Rs

Description
This form of the ASR (Arithmetic Shift Right) instruction is used to provide the signed value of a register divided by a constant power of 2.

In this case, ASR performs an arithmetic shift right of the value of register Rd by the value in the least-significant byte of register Rs, and stores the result back into the register Rd. The sign bit of the original Rd (Rd[31]) is inserted into the vacated bit positions. The condition code flags are updated (based on the result).

Operation
if Rs[7:0] == 0 then
  C Flag = unaffected
  Rd = unaffected
else if Rs[7:0] < 32 then
  C Flag = Rd[Rs[7:0] - 1]
  Rd = Rd Arithmetic_Shift_Right Rs[7:0]
else /* Rs[7:0] >= 32 */
  if Rd[31] == 0 then
    C Flag = Rd[31]
    Rd = 0
  else /* Rd[31] == 1 */
    C Flag = Rd[31]
    Rd = 0xffffffff

Exceptions
None

Qualifiers
None

Equivalent ARM syntax and encoding

MOVS Rd, Rd, ASR Rs
Conditional branch

Architecture v4T only

Description

This form of the B (Branch) instruction provides conditional changes to program flow.

In this case, B causes a conditional branch to a target address. The branch target address is calculated by shifting the 8-bit signed offset left by one bit, sign-extending the result to 32 bits, and adding this to the contents of the PC (which contains the address of the branch instruction plus 4). The instruction can therefore specify a branch of +/- 256 bytes.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in 3.3 The Condition Field on page 3-4.

Operation

\[
\text{if ConditionPassed(<cond>) then}
\]
\[
PC = PC + (\text{SignExtend(<8_bit_signed_offset>) \ll 1})
\]

Exceptions

None

Qualifiers

Condition Code

Notes

Offset calculation: An assembler will calculate the branch offset address from the difference between the address of the current instruction and the address of the target (given as a program label) minus four (because the PC holds the address of the current instruction plus four).

Memory bounds: Branching backwards past location zero and forwards over the end of the 32-bit address space is UNPREDICTABLE.
B \textlangle target\_address\rangle

**Description**
This form of the B (Branch) instruction provides unconditional changes to program flow.

In this case, B causes an unconditional branch to a target address. The branch target address is calculated by shifting the 11-bit signed (two’s complement) offset left one bit, sign-extending the result to 32 bits, and adding this to the contents of the PC (which contains the address of the branch instruction plus 4).

The instruction can therefore specify a branch of +/- 2048 bytes.

$$PC = PC + (\text{SignExtend}(<11\text{bit\_signed\_offset}>) \ll 1)$$

**Exceptions**
None

**Qualifiers**
None

**Notes**
- **Offset calculation**: An assembler will calculate the branch offset address from the difference between the address of the current instruction and the address of the target (given as a program label) minus four (because the PC holds the address of the current instruction plus four).
- **Memory bounds**: Branching backwards past location zero and forwards over the end of the 32-bit address space is UNPREDICTABLE.

---

**Equivalent ARM syntax and encoding**

\text{B \langle target\_address\rangle}

\begin{array}{cccccc}
\text{cond} & 1 & 0 & 1 & 0 & \text{sign extension of 11\_bit\_signed\_offset} \\
& & & & & 11\_bit\_signed\_offset \\
\end{array}
The BIC (Bit Clear) instruction 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.

BIC performs a bitwise AND of the complement of the value of register Rm with the value of register Rd, and stores the result back in register Rd. The condition code flags are updated (based on the result).

**Operation**

\[
\text{Rd} = \text{Rd} \text{ AND NOT } \text{Rm}
\]

- N Flag = Rd[31]
- Z Flag = if Rd == 0 then 1 else 0
- C Flag = <shifter_carry_out>
- V Flag = unaffected

**Exceptions**

None

**Qualifiers**

None

**Equivalent ARM syntax and encoding**

\[
\text{BICS Rd, Rd, Rm}
\]
BL <target_address>

Description
The BL (Branch with Link) instruction provides an unconditional subroutine call; the return from subroutine is achieved by copying the LR to the PC (see page 6-57). BL causes an unconditional subroutine call to a target address, and stores the return address into the LR (link register or R14). Thumb subroutine calls are made using a two-instruction sequence:

1. The first instruction (H==0) sign-extends the value of <11_bit_signed_offset>, shifts the result left by 12 bits, adds the value of the PC (the address of the branch instruction plus 4), and stores the result in LR.

2. The second instruction shifts the value of <11_bit_signed_offset> left by one bit, adds the value of LR (that was calculated by the first instruction), stores the result in the PC, places the address of the instruction after the second BL in LR.

The instruction can therefore specify a branch of +/- 4 Mbytes.

Operation
if H==0
   LR = PC + (SignExtend(<11_bit_signed_offset>) << 12)
else /* H==1 */
   <return_address> = PC + 2 | 1
   PC = LR + (<11_bit_signed_offset> << 1)
   LR = <return_address>

Exceptions
None

Qualifiers
None

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

Offset calculation: An assembler will calculate the branch offset address from the difference between the address of the current instruction and the address of the target (given as a program label) minus four (because the PC holds the address of the current instruction plus four).

Equivalent ARM syntax and encoding
BL <target_address>

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>offset</td>
<td>sign</td>
<td>22_bit_signed_offset</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Description
The BX (Branch and Exchange) instruction is used to branch between ARM code and Thumb code.

BX branches and selects the instruction set decoder to use to decode the instructions at the branch destination. The branch target address is the value of register Rm. The T flag is updated with bit 0 of the value of register Rm.

<table>
<thead>
<tr>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>T Flag = Rm[0]</td>
</tr>
<tr>
<td>PC = Rm[31:1] &lt;&lt; 1</td>
</tr>
</tbody>
</table>

Exceptions
None

Qualifiers
Condition Code

Notes
The H2 bit: This bit is the high register specifier:
H2 == 0 indicates that Rm specifies a low register
H2 == 1 indicates Rm specifies a high register

Transferring to Thumb: When transferring to the Thumb instruction set, bit[0] of Rm will be set to zero when transferred to the PC.

Transferring to ARM: When transferring to the ARM instruction set, bit[1:0] of Rm will be set to zero when transferred to the PC.

Equivalent ARM syntax and encoding

<table>
<thead>
<tr>
<th>Equivalent ARM syntax and encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>BX Rm</td>
</tr>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 12 11 8 7 6 5 4 3 0</td>
</tr>
<tr>
<td>1 1 1 0 0 0 1 0 0 1 0 SBO SBO SBO 0 0 1 Rm</td>
</tr>
</tbody>
</table>
The CMN (Compare Negative) instruction compares an arithmetic value and the negative of an arithmetic value and sets the condition code flags so that subsequent instructions can be conditionally executed (using a conditional branch instruction).

CMN performs a comparison by adding (or subtracting the negative of) the value of register Rm to (from) the value of register Rd. The condition code flags are updated (based on the result).

**Operation**

\[ <\text{alu\_out}> = Rn + Rm \]

- N Flag = \(<\text{alu\_out}>[31]\)
- Z Flag = if \(<\text{alu\_out}> == 0 \) then 1 else 0
- C Flag = NOT BorrowFrom(Rn + Rm)
- V Flag = OverflowFrom (Rn + Rm)

**Exceptions** None

**Qualifiers** None

---

**Equivalent ARM syntax and encoding**

```
CMN Rn, Rm
```

```
0 1 0 0 0 0 1 0 1 1 Rm 1 0 0 0 0 0 0 0 0 0 0
```

---

ARM Architecture Reference Manual
ARM DUI 0100B
### CMP (1)

#### Compare immediate

**Description**
This form of the CMP (Compare) instruction compares two arithmetic values and sets the condition code flags so that subsequent instructions can be conditionally executed (using a conditional branch instruction).

In this case, CMP performs a comparison by subtracting the value of the 8-bit immediate (values 0 to 255) from the value of register Rd. The condition code flags are updated (based on the result).

#### Operation

\[
\text{<alu_out>} = \text{Rn} - \text{<8_bit_immed>}
\]

- **N Flag** = \text{<alu_out>}[31]
- **Z Flag** = if \text{<alu_out>} == 0 then 1 else 0
- **C Flag** = NOT \text{BorrowFrom(Rn - <8_bit_immed>)}
- **V Flag** = \text{OverflowFrom (Rn - <8_bit_immed>)}

#### Exceptions
None

#### Qualifiers
None

---

### Equivalent ARM syntax and encoding

**CMP Rn, #<8_bit_immediate>**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>Rn</td>
</tr>
<tr>
<td>16</td>
<td>SBO</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>8_bit_immediate</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
CMP Rm, Rm

Description
This form of the CMP (Compare) instruction compares two arithmetic values and sets the condition code flags so that subsequent instructions can be conditionally executed (using a conditional branch instruction).

In this case, CMP performs a comparison by subtracting the value of register Rm from the value of register Rd. The condition code flags are updated (based on the result).

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td>Rm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation
\[ \text{alu_out} = Rn - Rm \]
N Flag = \(<\text{alu_out}>[31]\)
Z Flag = if \(<\text{alu_out}> == 0 \) then 1 else 0
C Flag = NOT BorrowFrom(Rn - Rm)
V Flag = OverflowFrom (Rn - Rm)

Exceptions
None

Qualifiers
None
**Thumb**

**CMP (3)**

**Compare high registers**

**Architecture v4T only**

---

**CMP Rn, Rm**

**Description**

This form of the CMP (Compare) instruction compares two arithmetic values in the high registers.

In this case, CMP:

- compares the value of a low register and a high register (H1=1, H2=0), or
- compares the value of a high register and a low register (H1=0, H2=1), or
- compares the value of a high register and another high register (H1=1, H2=1) and sets the condition code flags so that subsequent instructions can be conditionally executed (using a conditional branch instruction).

---

**Operation**

\[
\text{<alu\_out>} = Rn - Rm
\]

- N Flag = \text{<alu\_out>}[31]
- Z Flag = if \text{<alu\_out>} == 0 then 1 else 0
- C Flag = NOT BorrowFrom(Rn - Rm)
- V Flag = OverflowFrom(Rn - Rm)

---

**Exceptions**

None

---

**Qualifiers**

None

---

**Notes**

**Operand restriction:** If a low register is specified for Rd and Rm (H1=0 and H2=0), the result is **UNPREDICTABLE**.

---

**Equivalent ARM syntax and encoding**

**CMP Rn, Rm**

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | H1| H2| Rm| Rd|

---

**ARM Architecture Reference Manual**

ARM DUI 0100B
**EOR Rd, Rm**

**Description**  
The EOR (Exclusive OR) instruction can be used to invert selected bits in a register. For each bit, EOR with 1 will invert that bit, and EOR with 0 will leave it unchanged.  
EOR performs a bitwise Exclusive OR of the value of register Rm with the value of register Rd, and stores the result back in register Rd. The condition code flags are updated (based on the result).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Rd = Rd EOR Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>N Flag</td>
<td>Rd[31]</td>
</tr>
<tr>
<td>Z Flag</td>
<td>if Rd == 0 then 1 else 0</td>
</tr>
<tr>
<td>C Flag</td>
<td>&lt;shifter_carry_out&gt;</td>
</tr>
<tr>
<td>V Flag</td>
<td>unaffected</td>
</tr>
</tbody>
</table>

**Exceptions**  
None

**Qualifiers**  
None

---

**Equivalent ARM syntax and encoding**

<table>
<thead>
<tr>
<th>Equivalent ARM syntax and encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EORS Rd, Rd, Rm</strong></td>
</tr>
</tbody>
</table>

| 31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 12 11 10 9 8 7 6 5 4 3 0 |
|--------------------------|--------------------------|
| 1 1 1 0 0 0 0 0 0 0 0 0 1 1 | Rd | Rd | 0 0 0 0 0 0 0 0 0 | Rm |
The LDM instruction is useful as a block load instruction. Combined with STM (store multiple), it allows efficient block copy.

The LDMIA (Load Multiple Increment After) instruction loads a subset (or possibly all) of the general-purpose registers from sequential memory locations.

The registers are loaded in sequence:

- the lowest-numbered register first, from the lowest memory address
- the highest-numbered register last, from the highest memory address

The `<start_address>` 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 `<register_list>`.

The `<end_address>` value is four less than the sum of the value of the base register and four times the number of registers specified in `<register_list>`.

Finally, the base register Rn is incremented by four times the numbers of registers in `<register_list>`.

### Operation

```
Rn = `<start_address`
<end_address> = Rn + (Number_Of_Set_Bits_In(<register_list>) * 4) - 4
<address> = `<start_address`
for i = 0 to 7
  if <register_list>[i] == 1
    Ri = Memory[<address>,4]
    <address> = <address> + 4
assert <end_address> == <address> - 4
Rn = Rn + (Number_Of_Set_Bits_In(<register_list>) * 4)
```

### Equivalent ARM syntax and encoding

```
LDMIA Rn!, <register_list>
```

```
0 31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 14 13 12 11 10 9 8 7 0
1 1 1 0 1 0 0 0 1 0 1 1 Rn 0 0 0 0 0 0 0 0 register_list
```
Exceptions: Data Abort

Qualifiers: None

Notes:

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

Operand restrictions: If the base register Rn is specified in `<register_list>`, the final value of Rn is the loaded value (not the written-back value).

Data Abort: If a data abort is signalled, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if Rn is specified in `<register_list>`).

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 7, System Architecture and System Control Coprocessor), and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.
Load word
immediate offset

Architecture v4T only

**LDR Rd, [Rn, #5_bit_offset])**

**Description**
This form of the LDR (Load Register) instruction allows 32-bit memory data to be loaded into a general-purpose register where its value can be manipulated. The addressing mode is useful for accessing structure (record) fields. With an offset of zero, the address produced is the unaltered value of the base register Rn.

In this case, LDR loads a word from memory and writes it to register Rd. The memory address is calculated by adding 4 times the value of `<5_bit_offset>` to the value of register Rn. If the address is not word-aligned, the result is UNPREDICTABLE.

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5_bit_offset</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Operation**

```
<address> = Rn + (5_bit_offset * 4)
if <address>[1:0] == 0b00
   <data> = Memory[<address>,4]
else
   <data> = UNPREDICTABLE
Rd = <data>
```

**Exceptions**
Data Abort

**Qualifiers**
None

**Notes**
Alignment: If an implementation includes a System Control Coprocessor (see Chapter 7, System Architecture and System Control Coprocessor), and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.
LDR Rd, [Rn, Rm]

**Description**
This form of the LDR (Load Register) instruction allows 32-bit memory data to be loaded into a general-purpose register where its value can be manipulated. The addressing mode is useful for pointer + large offset arithmetic (use MOV immediate to set the offset), and accessing a single element of an array. In this case, LDR loads a word from memory and writes it to register Rd. The memory address is calculated by adding the value of register Rm to the value of register Rn. If the address is not word aligned, the result is UNPREDICTABLE.

```
<address> = Rn + Rm
if <address>[1:0] == 0b00
    <data> = Memory[<address>, 4]
else
    <data> = UNPREDICTABLE
Rd = <data>
```

**Exceptions**
Data Abort

**Qualifiers**
None

**Notes**
Alignment: If an implementation includes a System Control Coprocessor (see Chapter 7, System Architecture and System Control Coprocessor), and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.
Load word
PC-relative

Architecture v4T only

Description
This form of the LDR (Load Register) instruction allows 32-bit memory data to be loaded into a general-purpose register where its value can be manipulated. The addressing mode is useful for accessing PC relative data.

In this case, LDR loads a word from memory and writes it to register Rd. The memory address is calculated by adding 4 times the value of <8_bit_offset> to the value of the PC. If the address is not word-aligned, the result is UNPREDICTABLE.

Operation
<address> = (PC[31:1] << 1) + (8_bit_offset * 4)
if <address>[1:0] == 0b00
   <data> = Memory[<address>,4]
else
   <data> = UNPREDICTABLE
Rd = <data>

Exceptions
Data Abort

Qualifiers
None

Notes
Alignment: If an implementation includes a System Control Coprocessor (see Chapter 7, System Architecture and System Control Coprocessor), and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.

Equivalent ARM syntax and encoding
LDR Rd, [PC, #8_bit_offset]
LDR Rd, [SP, #8_bit_offset]

**Description**  
This form of the LDR (Load Register) instruction allows 32-bit memory data to be loaded into a general-purpose register where its value can be manipulated. The addressing mode is useful for accessing stack data.

In this case, LDR loads a word from memory and writes it to register Rd. The memory address is calculated by adding 4 times the value of `<8_bit_offset>` to the value of SP. If the address is not word-aligned, the result is UNPREDICTABLE.

$$<\text{address}> = \text{SP} + (8\text{_bit_offset} \times 4)$$

if $$<\text{address}>[1:0] == 0b00$$

$$<\text{data}> = \text{Memory}(<\text{address}>, 4)$$

else

$$<\text{data}> = \text{UNPREDICTABLE}$$

Rd = `<data`

**Exceptions**  
Data Abort

**Qualifiers**  
None

**Notes**  
Alignment: If an implementation includes a System Control Coprocessor (see Chapter 7, System Architecture and System Control Coprocessor), and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.

**Equivalent ARM syntax and encoding**

LDR Rd, [SP, #8_bit_offset]
LDRB (1)

Load unsigned byte immediate offset

Architecture v4T only

**LDRB Rd, [Rn, #5_bit_offset]**

**Description**
This form of the LDRB (Load Register Byte) instruction allows 8-bit memory data to be loaded into a general-purpose register where its value can be manipulated. The addressing mode is useful for accessing structure (record) fields. With an offset of zero, the address produced is the unaltered value of the base register Rn.

This form of the LDRB:
1. loads a byte from memory
2. zero-extends the byte to a 32-bit word
3. writes the word to register Rd

The memory address is calculated by adding the value of \(<5\_bit\_offset>\) to the value of register Rn.

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5_bit_offset</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rd</td>
</tr>
</tbody>
</table>

**Operation**

\(<address> = Rn + 5\_bit\_offset >

Rd = Memory[<address>,1]

**Exceptions**
Data Abort

**Qualifiers**
None

**Equivalent ARM syntax and encoding**

LDRB Rd, [Rn, #5_bit_offset]

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5_bit_offset</td>
</tr>
</tbody>
</table>

ARM Architecture Reference Manual
ARM DUI 0100B
LDRB Rd, [Rn, Rm]

Description
This form of the LDRB (Load Register Byte) instruction allows 8-bit memory data to be loaded into a general-purpose register where its value can be manipulated. The addressing mode is useful for pointer + large offset arithmetic (use the MOV immediate to set the offset), and accessing a single element of an array.

In this case, LDRB:
1. loads a byte from memory
2. zero-extends the byte to a 32-bit word
3. writes the word to register Rd

The memory address is calculated by adding the value register Rm to the value of register Rn.

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rm</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation
<address> = Rn + Rm
Rd = Memory<address>,1

Exceptions
Data Abort

Qualifiers
None

Equivalent ARM syntax and encoding
LDRB Rd, [Rn, Rm]

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rn | Rd |    | 0  | 0  | 0  | 0  | 0  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
**LDRH (Load unsigned halfword)**

**Immediate offset**

**Architecture v4T only**

**Description**
This form of the LDRH (Load Register Halfword) instruction allows 16-bit memory data to be loaded into a general-purpose register where its value can be manipulated. The addressing mode is useful for accessing structure (record) fields. With an offset of zero, the address produced is the unaltered value of the base register Rn.

In this case, LDRH:
1. loads a halfword from memory
2. zero-extends the halfword to a 32-bit word
3. writes the word to register Rd

The memory address is calculated by adding 2 times the value of <5_bit_offset> to the value of register Rn. If the address is not halfword-aligned, the result is UNPREDICTABLE.

**Operation**

```
<address> = Rn + (5_bit_offset * 2)
if <address>[1:0] == 0
    <data> = Memory[<address>,2]
else
    <data> = UNPREDICTABLE
Rd = <data>
```

**Exceptions**
Data Abort

**Qualifiers**
None

**Notes**
Alignment: If an implementation includes a System Control Coprocessor (see Chapter 7, System Architecture and System Control Coprocessor), and alignment checking is enabled, an address with bit[0] != 0 will cause an alignment exception.
LDRH Rd, [Rn, Rm]

Description
This form of the LDRH (Load Register Halfword) instruction allows 16-bit memory data to be loaded into a general-purpose register where its value can be manipulated. The addressing mode is useful for pointer + large offset arithmetic (use MOV immediate to set the offset), and accessing a single element of an array.

In this case, LDRH:
1. loads a halfword from memory
2. zero-extends the halfword to a 32-bit word
3. writes the word to register Rd

The memory address is calculated by adding the value of register Rm to the value of register Rn. If the address is not halfword-aligned, the result is UNPREDICTABLE.

```
Operation
<address> = Rn + Rm
if <address>[0] == 0
    <data> = Memory[<address>,2]
else
    <data> = UNPREDICTABLE
Rd = <data>
```

Exceptions
Data Abort

Qualifiers
None

Notes
Alignment: If an implementation includes a System Control Coprocessor (see Chapter 7, System Architecture and System Control Coprocessor), and alignment checking is enabled, an address with bit[0] != 0 will cause an alignment exception.

Equivalent ARM syntax and encoding
LDRH Rd, [Rn, Rm]

```
Equivalent ARM syntax and encoding
LDRH Rd, [Rn, Rm]
```

```
1 1 1 0 0 0 1 1 0 0 1 1
```

ARM Architecture Reference Manual
ARM DUI 0100B
The LDRSB (Load Register Signed Byte) instruction allows 8-bit signed memory data to be loaded into a general-purpose register where its value can be manipulated. The addressing mode is useful for pointer + large offset arithmetic (use the MOV immediate to set the offset), and accessing a single element of an array.

LDRSB:
1. loads a byte from memory
2. sign-extends the byte to a 32-bit word
3. writes it to register Rd

The memory address is calculated by adding the value register Rm to the value of register Rn.

<address> = Rn + Rm
Rd = SignExtend(Memory[<address>,1])

Exceptions: Data Abort
Qualifiers: None
LDRSH Rd, [Rn, Rm]

Description
The LDRSH (Load Register Signed Halfword) instruction allows 16-bit signed memory data to be loaded into a general-purpose register where its value can be manipulated. The addressing mode is useful for pointer + large offset arithmetic (use the MOV immediate to set the offset), and accessing a single element of an array.

LDRSH:
1. loads a halfword from memory
2. sign-extends the halfword to a 32-bit word
3. writes the word to register Rd

The memory address is calculated by adding the value register Rm to the value of register Rn. If the address is not halfword-aligned, the result is UNPREDICTABLE.

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rm</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation
\[
\text{if} \ \langle \text{address} \rangle[1:0] == 0 \\
\quad \langle \text{data} \rangle = \text{Memory}[\langle \text{address} \rangle, 2] \\
\text{else} \\
\quad \langle \text{data} \rangle = \text{UNPREDICTABLE} \\
Rd = \text{SignExtend}(\langle \text{data} \rangle)
\]

Exceptions
Data Abort

Qualifiers
None

Notes
Alignment: If an implementation includes a System Control Coprocessor (see Chapter 7, System Architecture and System Control Coprocessor), and alignment checking is enabled, an address with bit[0] != 0 will cause an alignment exception.
Logical shift left (immediate)

Description
This form of the LSL (Logical Shift Left) instruction is used to provide either the value of a register directly (LSL #0), or the value of a register multiplied by a constant power of two.

In this case, LSL performs a logical shift left of the value of register Rm by an immediate value in the range 0 to 31 and stores the result into the destination register Rd. Zeros are inserted into the vacated bit positions. The condition code flags are updated (based on the result).

Operation
if <shift_imm> == 0
   C Flag = UNAFFECTED
   Rd = UNAFFECTED
else /* <shift_imm> > 0 */
   C Flag = Rm[32 - <shift_imm>]
   Rd = Rd Logical_Shift_Left <shift_imm>
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
V Flag = unaffected

Exceptions
None

Qualifiers
None

Equivalent ARM syntax and encoding

\[
\text{MOV}$ S, \text{Rm, LSL #}<\text{shift_imm}>$
\]
This form of the LSL (Logical Shift Left) instruction is used to provide the unsigned value of a register multiplied by a variable (in a register) power of two.

In this case, LSL instruction performs a logical shift left of the value of register $Rd$ by the value in the least-significant byte of register $Rs$ and stores the result back into the register $Rd$. Zeros are inserted into the vacated bit positions. The condition code flags are updated (based on the result).

```
Operation
if Rs[7:0] == 0
    C Flag = UNAFFECTED
    Rd = UNAFFECTED
else if Rs[7:0] < 32 then
    C Flag = Rd[32 - Rs[7:0]]
    Rd = Rd Logical_Shift_Left Rs[7:0]
else if Rs[7:0] == 32 then
    C Flag = Rd[0]
    Rd = 0
else /* Rs[7:0] > 32 */
    C Flag = 0
    Rd = 0
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
V Flag = unaffected
```

Exceptions
None

Qualifiers
None

Equivalent ARM syntax and encoding
```
MOVZ Rd, Rd, LSL Rs
```

![ARM Architecture Reference Manual](image)
Logical shift right (immediate)

Architecture v4T only

### Description

This form of the LSR (Logical Shift Right) instruction is used to provide the value of a register, divided by a constant power of two.

In this case, LSR performs a logical shift right of the value of register Rm by an immediate value in the range 1 to 32, and stores the result into the destination register Rd. Zeros are inserted into the vacated bit positions.

A shift by 32 is encoded by:

<shift_imm> = 0

The condition code flags are updated (based on the result).

### Operation

```plaintext
if <shift_imm> == 0
  C Flag = Rd[31]
  Rd = 0
else /* <shift_imm> > 0 */
  C Flag = Rd[<shift_imm> - 1]
  Rd = Rd Logical_Shift_Right <shift_imm>

N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
V Flag = unaffected
```

### Exceptions
None

### Qualifiers
None

### Equivalent ARM syntax and encoding

```
MOVS Rd, Rm, LSR #<shift_imm>
```

```plaintext
               31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 12 11 7 6 5 4 3 0
SBZ Rd shift_imm 0 1 0 Rd
```

---

---

---

---
**LSR Rd, Rs**

**Description**
This form of the LSR (Logical Shift Right) instruction is used to provide the unsigned value of a register divided by a variable (in a register) power of two.

In this case, LSR instruction performs a logical shift right of the value of register Rd by the value in the least-significant byte of register Rs, and stores the result back into the register Rd. Zeros are inserted into the vacated bit positions. The condition code flags are updated (based on the result).

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Operation**

if Rs[7:0] == 0 then
  C Flag = unaffected
  Rd = unaffected
else if Rs[7:0] < 32 then
  C Flag = Rd[Rs[7:0] - 1]
  Rd = Rd Logical_Shift_Right Rs[7:0]
else if Rs[7:0] == 32 then
  C Flag = Rd[31]
  Rd = 0
else /* Rs[7:0] > 32 */
  C Flag = 0
  Rd = 0
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
V Flag = unaffected

**Exceptions**
None

**Qualifiers**
None

**Equivalent ARM syntax and encoding**

MOVS Rd, Rd, LSR Rs
MOV Rd, #<8_bit_immediate>

**Description**

This form of the MOV (Move) instruction moves a large constant value to a register.

In this case, MOV writes the value of the 8-bit immediate (values 0 to 255) to the destination register Rd. The condition code flags are updated (based on the result).

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rd</td>
<td>8_bit_immediate</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Operation**

- \( \text{Rd} = <8\text{ _bit_immed}> \)
- \( N \text{ Flag} = \text{Rd}[31] \)
- \( Z \text{ Flag} = \text{if Rd} == 0 \text{ then 1 else 0} \)
- \( C \text{ Flag} = \text{unaffected} \)
- \( V \text{ Flag} = \text{unaffected} \)

**Exceptions**

None

**Qualifiers**

None

---

**Equivalent ARM syntax and encoding**

MOV Rn, #<8_bit_immediate>

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9  | 8  | 7  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 8_bit_immediate |
MOV Rd, Rm

**Description**  
This form of the MOV (Move) instruction is used to move a value to, from, or between high registers.

In this case, MOV:

- moves the value of a low register to a high register (H1=1, H2=0), or
- moves the value of a high register to a low register (H1=0, H2=1), or
- moves the value of a high register and another high register (H1=1, H2=1).

The subroutine return instruction is:

```assembly
MOV PC, LR
```

(after executing a BL sequence; see page 6-33).

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>H1</td>
<td>H2</td>
<td>Rm</td>
<td>Rd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Operation**  
Rd = Rm

**Exceptions**  
None

**Qualifiers**  
None

**Notes**  
**Operand restriction:** If a low register is specified for Rd and Rm (H1=0 and H2=0), the result is UNPREDICTABLE.

**Equivalent ARM syntax and encoding**

```
MOV Rd, Rm
```

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 16 | 15 | 14 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | H1 | Rd | H1 | Rd | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | H2 | Rm |
Multiply

MUL Rd, Rm

Description
The MUL (Multiply) instruction multiplies signed or unsigned variables to produce a 32-bit result.

MUL multiplies the value of register Rm with the value of register Rd, and stores the result back in the register Rd. The condition code flags are updated (based on the result).

\[
\text{Rd} = (\text{Rm} \times \text{Rd})[31:0]
\]

- N Flag = Rd[31]
- Z Flag = if Rd == 0 then 1 else 0
- C Flag = UNPREDICTABLE
- V Flag = UNAFFECTED

Exceptions
None

Qualifiers
None

Notes
**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 Rd 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.

---

Equivalent ARM syntax and encoding

MULS Rd, Rm, Rd

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \text{Rd} & \text{SBZ} & \text{Rd} & 1 & 0 & 0 & 1 & \text{Rm} \\
\end{array}
\]
The MVN (Move NOT) instruction is used to compliment a register value, often to form a bit mask.

MVN writes the logical one’s compliment value of register Rn to the destination register Rd. The condition code flags are updated (based on the result).

**Operation**

- \( R_d = \text{NOT } R_m \)
- \( N \text{ Flag} = R_d[31] \)
- \( Z \text{ Flag} = \text{if } R_d == 0 \text{ then } 1 \text{ else } 0 \)
- \( C \text{ Flag} = \text{unaffected} \)
- \( V \text{ Flag} = \text{unaffected} \)

**Exceptions**

None

**Qualifiers**

None

---

**Equivalent ARM syntax and encoding**

**MVNS Rd, Rm**

- \( R_d = \text{NOT } R_m \)
- \( N \text{ Flag} = R_d[31] \)
- \( Z \text{ Flag} = \text{if } R_d == 0 \text{ then } 1 \text{ else } 0 \)
- \( C \text{ Flag} = \text{unaffected} \)
- \( V \text{ Flag} = \text{unaffected} \)
**NEG**

### Description

The NEG (Negate) instruction negates the value of one register and stores the result in a second register.

NEG subtracts the value of register Rn from zero, and stores the result in the destination register Rd. The condition code flags are updated (based on the result).

### Operation

\[
Rd = 0 - Rn
\]

- **N Flag**: \(Rd[31]\)
- **Z Flag**: if \(Rd = 0\) then 1 else 0
- **C Flag**: \(\text{NOT BorrowFrom}(0 - Rn)\)
- **V Flag**: \(\text{OverflowFrom}(0 - Rn)\)

### Exceptions

None

### Qualifiers

None

---

**Equivalent ARM syntax and encoding**

\[
\text{RSBS Rd, Rn, #0}
\]
ORR Rd, Rm

Description
The ORR (Logical OR) instruction can be used to set selected bits in a register; for each bit, ORR with 1 will set the bit, and ORR with 0 will leave it unchanged.

ORR performs a bitwise (inclusive) OR of the value of register Rm with the value of register Rd, and stores the result back in register Rd. The condition code flags are updated (based on the result).

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rm</td>
<td>Rd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation
Rd = Rd OR Rm
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = <shifter_carry_out>
V Flag = unaffected

Exceptions
None

Qualifiers
None

Equivalent ARM syntax and encoding
ORRS Rd, Rd, Rm

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 3  | 2  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | Rd | Rd | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Rm |
POP {<register_list>, <PC>}

**Description**

The POP (Pop Multiple Registers) instruction is useful for stack operations, including procedure exit, to restore saved registers, load the PC with the return address, and update the stack pointer.

POP loads a subset (or possibly all) of the general-purpose registers and optionally the PC from sequential memory locations. Registers are loaded in sequence:

- the lowest-numbered register first, from the lowest memory address
  (<start_address>)
- the highest-numbered register last, from the highest memory address
  (<end_address>)

The <start_address> is the value of the SP.

Subsequent addresses are formed by incrementing the previous address by four. One address is produced for each register that is specified in <register_list>.

The <end_address> value is four less than the sum of the value of the SP and four times the number of registers specified in <register_list> (including the R bit).

Finally, the base register Rn is incremented by four times the numbers of registers in <register_list> (plus the R bit).

```
<start_address> = Rn
<end_address> = Rn + (Number_Of_Set_Bits_In(<register_list> + R) * 4) - 4
<address> = <start_address>
for i = 0 to 7
    if <register_list>[i] == 1
        Ri = Memory[<address>,4]
        <address> = <address> + 4
    if R == 1
        PC = Memory[<address>,4]
        <address> = <address> + 4
assert <end_address> == <address> - 4
Rn = Rn + (Number_Of_Set_Bits_In(<register_list> + R) * 4)
```

**Equivalent ARM syntax and encoding**

```
LDMIA SP!, <register_list>, {PC}
```

```
1 1 1 0 1 0 0 0 1 0 0 1 1 1 0 1 | R 0 0 0 0 0 0 0 |
```
**POP**

**Thumb**

**Pop multiple registers**

**Architecture v4T only**

<table>
<thead>
<tr>
<th>Exceptions</th>
<th>Data Abort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qualifiers</td>
<td>None</td>
</tr>
<tr>
<td>Notes</td>
<td>The R bit: If R == 1, the PC is also loaded from the stack; if R == 0, the PC is not loaded. Data Abort: If a data abort is signalled, the value left in SP is IMPLEMENTATION DEFINED, but is either the original SP value or the updated SP value. Non-word aligned addresses: Pop multiple instructions ignore the least-significant two bits of &lt;address&gt; (the words are not rotated as for load word). Alignment: If an implementation includes a System Control Coprocessor (see Chapter 7, System Architecture and System Control Coprocessor), and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.</td>
</tr>
</tbody>
</table>
The PUSH (Push Multiple Registers) instruction is useful for stack operations, including procedure entry, to save registers (optionally including the return address), and to update the stack pointer.

PUSH stores a subset (or possibly all) of the general-purpose registers and optionally the LR to sequential memory locations. The registers are stored in sequence:

- the lowest-numbered register first, to the lowest memory address
- the highest-numbered register last, to the highest memory address

The <start_address> is the value of the SP. Subsequent addresses are formed by incrementing the previous address by four. One address is produced for each register that is specified in <register_list>.

The <end_address> value is four less than the sum of the value of the SP and four times the number of registers specified in <register_list> (including the R bit).

Finally, the base register Rn is incremented by four times the numbers of registers in <register_list> (plus the R bit).

**Operation**

<start_address> = Rn
<end_address> = Rn + (Number_Of_Set_Bits_In(<register_list> + R) * 4) - 4
<address> = <start_address>
for i = 0 to 7
    if <register_list>[i] == 1
        Memory[<address>,4] = Ri
        <address> = <address> + 4
    if R == 1
        Memory[<address>,4] = LR
        <address> = <address> + 4
assert <end_address> == <address> - 4
Rn = Rn + (Number_Of_Set_Bits_In(<register_list> + R) * 4)

**Equivalent ARM syntax and encoding**

LDMDB SP!, <register_list>, {LR}

**ARM Architecture Reference Manual**

ARM DUI 0100B
<table>
<thead>
<tr>
<th>Exceptions</th>
<th>Data Abort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qualifiers</td>
<td>None</td>
</tr>
<tr>
<td>Notes</td>
<td>The R bit: If R == 1, the LR is also stored to the stack; if R == 0, the LR is not stored. Data Abort: If a data abort is signalled, the value left in SP is IMPLEMENTATION DEFINED, but is either the original SP value or the updated SP value. Non-word aligned addresses: Push multiple instructions ignore the least-significant two bits of &lt;address&gt; (the words are not rotated as for load word). Alignment: If an implementation includes a System Control Coprocessor (see Chapter 7, System Architecture and System Control Coprocessor), and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.</td>
</tr>
</tbody>
</table>
**ROR Rd, Rs**

**Description**
The ROR (Rotate Right Register) instruction is used to provide the value of a register rotated by a variable value (in a register).

ROR performs a rotate right of the value of register Rd by the value in the least-significant byte of register Rs, and stores the result back into register Rd. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The condition code flags are updated (based on the result).

**Operation**

```
if Rs[7:0] == 0 then
  C Flag = unaffected
  Rd = unaffected
else if Rs[4:0] == 0 then
  C Flag = Rd[31]
  Rd = unaffected
else /* Rs[4:0] > 0 */
  C Flag = Rd[Rs[4:0] - 1]
  Rd = Rd Rotate_Right Rs[4:0]
```

**Exceptions**
None

**Qualifiers**
None

**Equivalent ARM syntax and encoding**

```
MOVS Rd, Rd, ROR Rs
```

```
1 1 1 0 0 0 1 1 0 1 1 11 0 1 1 1 1 0 1 1 0 1 1 3 0 29 28 27 26 25 24 23 22 21 20 19
```

**ARM Architecture Reference Manual**

ARM DUI 0100B
The SBC (Subtract with Carry) instruction is used to synthesize 64-bit subtraction. If register pairs R0,R1 and R2,R3 hold 64-bit values (R0 and R2 hold the least-significant word), the following instructions leave the 64-bit sum in R0,R1.

\[
\begin{align*}
    &\text{SUB} & R0, & R2 \\
    &\text{SBC} & R1, & R3 \\
\end{align*}
\]

SBC subtracts the value of register Rm and the value of NOT(Carry Flag) from the value of register Rd, and stores the result in register Rd. The condition code flags are updated (based on the result).

\[
\begin{align*}
    \text{Rd} &= \text{Rd} - \text{Rm} - \text{NOT}(C \text{ Flag}) \\
    \text{N Flag} &= \text{Rd}[31] \\
    \text{Z Flag} &= \text{if } \text{Rd} == 0 \text{ then } 1 \text{ else } 0 \\
    \text{C Flag} &= \text{CarryFrom}(\text{Rd} - \text{Rm} - \text{NOT}(C \text{ Flag})) \\
    \text{V Flag} &= \text{OverflowFrom}(\text{Rd} - \text{Rm} - \text{NOT}(C \text{ Flag}))
\end{align*}
\]

Exceptions: None

Qualifiers: None

Equivalent ARM syntax and encoding

\[
\begin{align*}
    &\text{SBCS} & Rd, & Rd, & Rm \\
\end{align*}
\]
**STMIA Rn!, <register_list>**

**Description**
The STM (Store Multiple) instruction is useful as a block store instruction. Combined with LDM (load multiple), it allows efficient block copy.

The STMIA (Store Multiple Increment After) instruction stores a subset (or possibly all) of the general-purpose registers to sequential memory locations. The registers are stored in sequence:

- the lowest-numbered register first, to the lowest memory address
  
- the highest-numbered register last, to the highest memory address

The `<start_address>` 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 `<register_list>`.

The `<end_address>` value is four less than the sum of the value of the base register and four times the number of registers specified in `<register_list>`.

Finally, the base register Rn is incremented by 4 times the numbers of registers in `<register_list>`.

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|    |    |    |    |
| 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | Rn |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

**Operation**

- `<start_address>` = Rn
- `<end_address>` = Rn + (Number_of_Set_Bits_In(<register_list>) * 4) - 4
- `<address>` = `<start_address>`
  
  for i = 0 to 7
  
  if `<register_list>[i]` == 1
    Memory[<address>,4] = Ri
  
  `<address>` = `<address>` + 4
  
  assert `<end_address>` == `<address>` - 4
  
  Rn = Rn + (Number_of_Set_Bits_In(<register_list>) * 4)

**Equivalent ARM syntax and encoding**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|    |    |    |    |
| 1  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |   |   |   |   |

ARM Architecture Reference Manual

ARM DUI 0100B
**Exceptions**  Data Abort

**Qualifiers**  None

**Notes**

- **Register Rn**: Specifies the base register used by `<addressing_mode>`, and writeback is specified, the value of Rn stored for Rn is UNPREDICTABLE.

- **Operand restrictions**: If the base register Rn is specified in `<register_list>`, and writeback is specified, the value of Rn stored for Rn is UNPREDICTABLE.

- **Data Abort**: If a data abort is signalled, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value.

- **Non-word-aligned addresses**: Store 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 7, System Architecture and System Control Coprocessor), and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.
The STR (Store Register) instruction allows 32-bit data from a general-purpose register to be stored to memory. The addressing mode is useful for accessing structure (record) fields. With an offset of zero, the address produced is the unaltered value of the base register Rn.

STR stores a word from register Rd to memory. The memory address is calculated by adding 4 times the value of `<5_bit_offset>` to the value of register Rn. If the address is not word-aligned, the result is UNPREDICTABLE.

```
<address> = Rn + (5_bit_offset * 4)
<data> = Rd
if <address>[1:0] == 0b00
    Memory[<address>,4] = <data>
else
    Memory[<address>,4] = UNPREDICTABLE
```

**Description**
The STR (Store Register) instruction allows 32-bit data from a general-purpose register to be stored to memory. The addressing mode is useful for accessing structure (record) fields. With an offset of zero, the address produced is the unaltered value of the base register Rn.

**Operation**
```
<address> = Rn + (5_bit_offset * 4)
<data> = Rd
if <address>[1:0] == 0b00
    Memory[<address>,4] = <data>
else
    Memory[<address>,4] = UNPREDICTABLE
```

**Exceptions**
Data Abort

**Qualifiers**
None

**Notes**
Alignment: If an implementation includes a System Control Coprocessor (see Chapter 7, System Architecture and System Control Coprocessor), and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.
STR Rd, [Rn, Rm]

Description
This form of the STR (Store Register) instruction allows 32-bit data from a general-purpose register to be stored to memory. The addressing mode is useful for pointer + large offset arithmetic (use the MOV immediate to set the offset), and for accessing a single element of an array.

In this case, STR stores a word from register Rd to memory. The memory address is calculated by adding the value of register Rm to the value of register Rn. If the address is not word-aligned, the result is UNPREDICTABLE.

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rm</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation
<address> = Rn + Rm
<data> = Rd
if <address>[1:0] == 0b00
   Memory[<address>,4] = <data>
else
   Memory[<address>,4] = UNPREDICTABLE

Exceptions
Data Abort

Qualifiers
None

Notes
Alignment: If an implementation includes a System Control Coprocessor (see Chapter 7, System Architecture and System Control Coprocessor), and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.
**STR Rd, [SP, #8_bit_offset]**

### Description
This form of the STR (Store Register) instruction allows 32-bit data from a general-purpose register to be stored to memory. The addressing mode is useful for accessing stack data.

In this case, STR stores a word from register Rd to memory. The memory address is calculated by adding 4 times the value of `<8_bit_offset>` to the value of the SP. If the address is not word-aligned, the result is UNPREDICTABLE.

### Operation

<table>
<thead>
<tr>
<th>Bit</th>
<th>(&lt;address&gt;)</th>
<th>(&lt;data&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>(\text{SP} + (8\text{_{bit_offset}} \times 4))</td>
<td>(\text{Rd})</td>
</tr>
<tr>
<td>0</td>
<td>[&lt;address&gt;,4]</td>
<td>(\text{Rd})</td>
</tr>
</tbody>
</table>

- if \(<address>[1:0] == 0b00\)
  - \(\text{Memory}[<address>,4] = \text{Rd}\)
- else
  - \(\text{Memory}[<address>,4] = \text{UNPREDICTABLE}\)

### Exceptions
Data Abort

### Qualifiers
None

### Notes
Alignment: If an implementation includes a System Control Coprocessor ((see Chapter 7, System Architecture and System Control Coprocessor), and alignment checking is enabled, an address with \(\text{bits}[1:0] \neq 0b00\) will cause an alignment exception.

### Equivalent ARM syntax and encoding

<table>
<thead>
<tr>
<th>Bit</th>
<th>(\text{STR Rd, [SP, #8_bit_offset]})</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>1 1 1 0 0 1 0 1 1 0 0 0 1 1 0 1 Rd 0 0 8_bit_offset 0 0</td>
</tr>
</tbody>
</table>
STRB Rd, [Rn, #5_bit_offset]

Description  This form of the STRB (Store Register Byte) instruction allows 8-bit data from a general-purpose register to be stored to memory. The addressing mode is useful for accessing structure (record) fields. With an offset of zero, the address produced is the unaltered value of the base register Rn.

In this case, STRB stores a byte from the least-significant byte of register Rd to memory. The memory address is calculated by adding the value of <5_bit_offset> to the value of register Rn.

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>5_bit_offset</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation  
\[
<\text{address}> = \text{Rn} + 5\_\text{bit\_offset} \\
\text{Memory}[<\text{address}>,1] = \text{Rd}[7:0]
\]

Exceptions  Data Abort

Qualifiers  None

Equivalent ARM syntax and encoding

STRB Rd, [Rn, #5_bit_offset]

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | Rn | Rd | 0  | 0  | 0  | 0  | 0  | 5_bit_offset | 0  | 0  |
This form of the STRB (Store Register Byte) instruction allows 8-bit data from a general-purpose register to be stored to memory. The addressing mode is useful for pointer + large offset arithmetic (use the MOV immediate to set the offset), and for accessing a single element of an array.

In this case, STRB stores a byte from the least-significant byte of register Rd to memory. The memory address is calculated by adding the value register Rm to the value of register Rn.

\[
<\text{address}> = Rn + Rm \\
\text{Memory}[<\text{address}>,1] = \text{Rd}[7:0]
\]

**Description**

**Operation**

**Exceptions**

**Qualifiers**

None
This form of the STRH (Store Register Halfword) instruction allows 16-bit data from a general-purpose register to be stored to memory. The addressing mode is useful for accessing structure (record) fields. With an offset of zero, the address produced is the unaltered value of the base register Rn.

In this case, STRH stores a halfword from the least-significant halfword of register Rd to memory. The memory address is calculated by adding 2 times the value of <5_bit_offset> to the value of register Rn. If the address is not halfword-aligned, the result is UNPREDICTABLE.

\[
\text{<address>} = \text{Rn} + (5\text{\_bit\_offset} \times 2)
\]

\[
\text{<data>} = \text{Rd}
\]

\[
\text{if } \text{<address>}[1:0] = 0 \\
\text{ Memory[<address>,2] = <data>[15:0]} \\
\text{else} \\
\text{ Memory[<address>,2] = UNPREDICTABLE}
\]

**Equivalence ARM syntax and encoding**

\[
\text{STRH Rd, [Rn, #5\_bit\_offset]}
\]

Alignment: If an implementation includes a System Control Coprocessor (see Chapter 7, System Architecture and System Control Coprocessor), and alignment checking is enabled, an address with bit[0] != 0 will cause an alignment exception.
**STRH Rd, [Rn, Rm]**

### Description
This form of the STRH (Store Register Halfword) instruction allows 16-bit data from a general-purpose register to be stored to memory. The addressing mode is useful for pointer + large offset arithmetic (use the MOV immediate to set the offset), and accessing a single element of an array.

In this case, STRH stores a halfword from the least-significant halfword of register Rd to memory. The memory address is calculated by adding the value of register Rm to the value of register Rn. If the address is not halfword-aligned, the result is UNPREDICTABLE.

### Operation
- \( <\text{address}> = Rn + Rm \)
- \( <\text{data}> = Rd \)
- if \( <\text{address}>[1:0] == 0 \)
  - \( \text{Memory}(<\text{address}>, 2) = <\text{data}>[15:0] \)
- else
  - \( \text{Memory}(<\text{address}>, 2) = \text{UNPREDICTABLE} \)

### Exceptions
Data Abort

### Qualifiers
None

### Notes
**Alignment:** If an implementation includes a System Control Coprocessor (see Chapter 7, System Architecture and System Control Coprocessor), and alignment checking is enabled, an address with bit[0] != 0 will cause an alignment exception.

### Equivalent ARM syntax and encoding

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rm</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | Rn | Rd | SBZ | 1  | 0  | 1  | 1  | Rm |
SUB Rd, Rn, #<3_bit immediate>

Description
This form of the SUB (Subtract) instruction subtracts a small constant value from the value of a register and stores the result in a second register.

In this case, SUB subtracts the value of the 3-bit immediate (values 0 to 7) from the value of register Rn, and stores the result in the destination register Rd. The condition code flags are updated (based on the result).

Operation
Rd = Rn - <3_bit_immed>
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = NOT BorrowFrom(Rn - <3_bit_immed>)
V Flag = OverflowFrom(Rn - <3_bit_immed>)

Exceptions
None

Qualifiers
None

Equivalent ARM syntax and encoding
SUBS Rd, Rn, #<3_bit immediate>

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19</th>
<th>16 15 12 11 10 9 8 7 6 5 4 3 2 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0</td>
<td>0 0 1 0 0 1 0 1</td>
</tr>
</tbody>
</table>
SUB Rd, #<8_bit_immediate>

Description
This form of the SUB (Subtract) instruction subtracts a large constant value from the value of a register and stores the result back in the same register.

In this case, SUB subtracts the value of the 8-bit immediate (values 0 to 255) from the value of register Rd, and stores the result back in the register Rd. The condition code flags are updated (based on the result).

Operation
Rd = Rd - <8_bit_immed>
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = NOT BorrowFrom(Rd - <8_bit_immed>)
V Flag = OverflowFrom(Rd - <8_bit_immed>)

Exceptions
None

Qualifiers
None

Equivalent ARM syntax and encoding
SUBS Rd, Rd, #<8_bit_immediate>
This form of the SUB (Subtract) instruction subtracts the value of one register from
the value of a second register and stores the result in a third register.

In this case, SUB subtracts the value of register Rm from the value of register Rn,
and stores the result in the destination register Rd. The condition code flags are
updated (based on the result).

**Operation**

\[
Rd = Rn - Rm
\]

- **N Flag** = \(Rd[31]\)
- **Z Flag** = if \(Rd = 0\) then 1 else 0
- **C Flag** = \(\text{NOT BorrowFrom}(Rn - Rm)\)
- **V Flag** = \(\text{OverflowFrom}(Rn - Rm)\)

**Exceptions** None

**Qualifiers** None

---

**Equivalent ARM syntax and encoding**

**SUBS Rd, Rn, Rm**
This form of the SUB (Subtract) instruction is used to increase the size of the stack. In this case, SUB subtracts the value of the 7-bit immediate (values 0 to 127) and the value of the SP, and stores the result back in the SP.

\[ SP = SP - \text{7-bit_immediate} \]

**Exceptions**
None

**Qualifiers**
None

**Equivalent ARM syntax and encoding**

```
SUB SP, SP, #<7_bit_immediate>
```

```
1 1 1 0 0 1 0 1 0 0 1 1 0 1 1 0 0 1 0 0 0 0 7_bit_immediate
```
**SWI <8_bit_immediate>**

**Description**
The SWI instruction is used as an operating system service call. It can be used in two ways:

1. Uses the 8-bit offset to indicate the OS service that is required.
2. Ignores the 8-bit field and indicates the service required with a general-purpose register.

A SWI exception is generated, which is handled by an operating system to provide the requested service; see 2.5 Exceptions on page 2-6.

**Operation**

- R14_svc = PC
- SPSR_svc = CPSR
- CPSR[7] = 0 ; begin ARM execution
- CPSR[4:0] = 0b10011 ; enter Supervisor mode
- CPSR[7] = 1 ; disable IRQ
- PC = 0x08

**Exceptions**
None

**Qualifiers**
None

---

**Equivalent ARM syntax and encoding**

```
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 0
1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8_bit_immediate
```

---

ARM Architecture Reference Manual
ARM DUI 0100B

6-81
The TST (Test) instruction is used to determine if many bits of a register are all clear, or if at least one bit of a register is set.

TST performs a comparison by logically ANDing the value of register Rm from the value of register Rd. The condition code flags are updated (based on the result).

\[
\text{Operation} \quad <\text{alu\_out}> = \text{Rn} \text{ AND Rm}
\]

\[
\begin{align*}
\text{N Flag} &= <\text{alu\_out}>[31] \\
\text{Z Flag} &= \text{if } <\text{alu\_out}> == 0 \text{ then 1 else 0} \\
\text{C Flag} &= \text{UNAFFECTED} \\
\text{V Flag} &= \text{UNAFFECTED}
\end{align*}
\]

None

None
System Architecture and System Control Coprocessor
This chapter describes ARM system architecture, and the system control processor.

7.1 Introduction 7-2
7.2 CP15 Access 7-2
7.3 CP15 Architectures 7-2
7.4 ARMv4 System Control Coprocessor 7-3
7.5 ARMv3 System Control Coprocessor 7-10
7.6 Memory Management Unit (MMU) Architecture 7-14
7.7 Cache and Write Buffer Control 7-22
7.8 Access Permissions 7-22
7.10 Aborts 7-24
7.11 MMU Faults 7-24
7.9 Domains 7-23
7.12 External Aborts 7-28
7.13 System-level Issues 7-29
7.14 Semaphores 7-31
7.1 Introduction

Implementations of the ARM architecture optionally incorporate:

- on-chip Memory Management Unit (MMU) (including Translation Lookaside Buffer(s) (TLB))
- Instruction and/or Data Cache (IDC)
- Write Buffer (WB)

If these functions are implemented, coprocessor 15 is used to control them. Coprocessor 15 is called the System Control Coprocessor or just CP15.

The MMU incorporates a two-level page table for virtual to physical address translation, and access permission attributes for each virtual to physical translation. The MMU page tables also contain cache and write buffer enables; therefore, if a cache or a write buffer is implemented, the MMU must also be implemented.

7.2 CP15 Access

CP15 defines 16 registers. CP15 registers can only be accessed with MRC and MCR instructions (CDP, LDC and STC instructions to CP15 will cause an undefined instruction trap). The CRn field of MRC and MCR instructions specify the coprocessor register to access, and the CRm field and opcode_2 field are used to specify a particular action when addressing some registers.

```
   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 1 1 1 1    opcode_2 1 CRm
```

Opcode_1 should be zero (sbz) for all CP15 instructions.

If a cache, MMU and Write Buffer are not implemented, CP15 will not be implemented, and all accesses to CP15 will cause undefined exceptions.

Any access to CP15 while the processor is in User mode will cause an undefined instruction exception.

An MRC instruction from coprocessor 15 to register 15 is UNPREDICTABLE.

7.3 CP15 Architectures

If a cache, MMU and Write Buffer are implemented, CP15 register 0 contains an architecture field that specifies a particular layout and functionality for the remaining registers.

Reading from CP15 register 0 returns an architecture and implementation-defined identification from the processor:

```
   31 24 23 16 15 12 11 4 3 0
Implementor Architecture version Part number Revision
```
7.4 ARMv4 System Control Coprocessor

ARM Architecture Version 4 System Control Coprocessor is designed to control a single combined instruction and data cache, or separate instructions and data caches, a write buffer, a prefetch buffer, and a virtual to physical address translator including combined instruction and data TLB or separate instruction and data TLB. CP15 also controls various system configuration signals.

<table>
<thead>
<tr>
<th>Register</th>
<th>Reads</th>
<th>Writes</th>
<th>Update Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ID Register</td>
<td>UNPREDICTABLE</td>
<td>No update</td>
</tr>
<tr>
<td>1</td>
<td>Control</td>
<td>Control</td>
<td>Read Modify Write</td>
</tr>
<tr>
<td>2</td>
<td>Translation Table Base</td>
<td>Translation Table Base</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Domain Access Control</td>
<td>Domain Access Control</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>UNPREDICTABLE</td>
<td>UNPREDICTABLE</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Fault Status</td>
<td>Fault Status</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Fault Address</td>
<td>Fault Address</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Cache Operations</td>
<td>Cache operations</td>
<td>Write only</td>
</tr>
<tr>
<td>8</td>
<td>TLB operations</td>
<td>TLB operations</td>
<td>Write only</td>
</tr>
<tr>
<td>9 to 15</td>
<td>UNPREDICTABLE</td>
<td>UNPREDICTABLE</td>
<td></td>
</tr>
</tbody>
</table>

*Table 7-1: ARMv4 CP15 register summary*
7.4.1 Register 0: ID register

Reading from CP15 register 0 returns the implementation-defined identification for the processor. The CRm and opcode_2 fields are ignored when reading CP15 register 0, and SHOULD BE ZERO.

- Bits[3:0] contain the revision number for the processor
- Bits[15:4] contain a 3-digit part number in binary-coded decimal format (for example, 0x810 for ARM810)
- Bits[23:16] contain the architecture version (for example, 0x01 = Version 4)
- Bits[31:24] contain the ASCII code of an implementation trademark (0x41 = A = ARM Ltd.)

Note: Writing to CP15 register 0 is unpredictable.

7.4.2 Register 1: Control register

Reading from CP15 register 1 reads the control bits. The CRm and opcode_2 fields are IGNORED when reading CP15 register 1, and should be zero.

Writing to CP15 register 1 sets the control bits. The CRm and opcode_2 fields are not used when writing CP15 register 1, and should be zero.

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

- **M** Bit 0  Memory Management Unit (MMU) Enable/Disable
  0 = MMU disabled
  1 = MMU enabled

- **A** Bit 1  Alignment Fault Enable/Disable
  0 = Address alignment fault checking disabled
  1 = Address alignment fault checking enabled

- **C** Bit 2  Instruction and data cache Enable/Disable
  If separate instruction and data caches are implemented, this bit controls only the data cache enable/disable, and the I bit (bit 12) controls the instruction cache enable/disable.
  0 = Instruction and data cache (IDC) disabled
  1 = Instruction and data cache (IDC) enabled

- **W** Bit 3  Write buffer Enable/Disable
  0 = Write buffer disabled
  1 = Write buffer enabled
System Architecture and System Control Coprocessor

P Bit 4  32-bit/26-bit Exception handlers
Implementations that support 26-bit configurations (see Chapter 5, The 26-bit Architectures) use this bit to control the PROG32 signal.
0 = 26-bit exception handlers
1 = 32-bit exception handlers
This bit is UNPREDICTABLE on implementations that do not support
26-bit configurations, and should be 1.

D Bit 5  32-bit/26-bit data address range
Implementations that support 26-bit data spaces use this bit to control
the DATA32 signal (see Chapter 5, The 26-bit Architectures).
0 = 26-bit data address checking enabled
1 = 26-bit data address checking disabled (32-bit data addresses)
This bit is UNPREDICTABLE on implementations that do not support
26-bit data spaces, and should be 1.

L Bit 6  IMPLEMENTATION DEFINED.

B Bit 7  Big endian/Little endian
0 = Little endian operation
1 = Big endian operation

S Bit 8  System protection
This bit modifies the MMU protection system.

R Bit 9  ROM protection
This bit modifies the MMU protection system.

F Bit 10 IMPLEMENTATION DEFINED

Z Bit 11 IMPLEMENTATION DEFINED

I Bit 12  Instruction cache enable/disable
0 = Instruction cache disabled
1 = Instruction cache enabled
If separate instruction and data caches are implemented, this bit
controls the instruction cache enable/disable, and the C bit (bit 2)
controls the data cache enable/disable. If a combined instruction and
data cache is implemented, any writes to this bit are IGNORED, and
reads return an UNPREDICTABLE value.

Bits 31:13  When read return an UNPREDICTABLE value, and when written SHOULD
BE ZERO.

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

If the cache and/or write buffer are enabled when the MMU is not enabled, the results
are UNPREDICTABLE.
7.4.3 Register 2: Translation table base register

![Translation Table Base UNP/SBZP](image)

Reading from CP15 register 2 returns the pointer to the currently active first-level translation table in bits[31:14] and an unpredictable value in bits[13:0]. The CRm and opcode_2 fields are ignored when reading CP15 register 2, and should be zero.

Writing to CP15 register 2 updates the pointer to the currently active first-level translation table from the value in bits[31:14] of the written value. Bits[13:0] must be written as zero. The CRm and opcode_2 fields are ignored when writing CP15 register 2, and should be zero.

7.4.4 Register 3: Domain access control register

![Domain Access Control Register](image)

Reading from CP15 register 3 returns the value of the Domain Access Control Register. The CRm and opcode_2 fields are ignored when reading CP15 register 3, and should be zero.

Writing to CP15 register 3 writes the value of the Domain Access Control Register. The CRm and opcode_2 fields are ignored when writing CP15 register 3, and should be zero.

The Domain Access Control Register consists of sixteen 2-bit fields, each defining the access permissions for one of the 16 Domains (D15-D0). For the meaning of each field, see 7.9 Domains on page 7-23.

7.4.5 Register 4: Reserved

Reading and writing CP15 register 4 is unpredictable.
7.4.6 Register 5: Fault status register

Reading CP15 register 5 returns the value of the Fault Status Register (FSR). The FSR contains the source of the last data fault. Note that only the bottom 9 bits are returned. The upper 23 bits are unpredictable. The FSR indicates the domain and type of access being attempted when an abort occurred.

- Bit 8 returns zero
- Bits 7:4 specify which of the 16 domains (D15-D0) was being accessed when a fault occurred
- Bits 3:0 indicate the type of access being attempted. The encoding of these bits is shown in Table 7-9: Priority encoding of fault status on page 7-25.

The FSR is only updated for data faults, not for prefetch faults. The CRm and opcode_2 fields are ignored when reading CP15 register 5, and should be zero.

Writing CP15 register 5 sets the Fault Status Register to the value of the data written. This is useful for a debugger to restore the value of the FSR. The upper 24 bits written should be zero (SBZ). The CRm and opcode_2 fields are ignored when writing CP15 register 5, and should be zero.

7.4.7 Register 6: Fault address register

Reading CP15 register 6 returns the value of the Fault Address Register (FAR). The FAR holds the virtual address of the access which was attempted when a fault occurred. The FAR is only updated for data faults, not for prefetch faults. The CRm and opcode_2 fields are ignored when reading CP15 register 6, and should be zero.

Writing CP15 register 6 sets the Fault Address Register to the value of the data written. This is useful for a debugger to restore the value of the FAR. The CRm and opcode_2 fields are ignored when writing CP15 register 6, and should be zero.
7.4.8 Register 7: Cache functions

Writing to CP15 register 7 is used to control caches and buffers.

An ARM implementation may include a combined instruction and data cache, or separate instruction and data caches. A write buffer, prefetch buffer and branch target cache may also be implemented and are controlled by this register. Several cache functions are defined, and the function to be performed is selected by the opcode_2 and CRm fields in the MCR instruction used to write CP15 register 7.

The Flush ID functions flush (invalidate) all cache data.

The Flush Entry functions may be implemented to flush more than a single entry, up to the entire cache.

The Clean D cache functions write out dirty data held in a writeback cache. They do not invalidate any cached data.

Any functions that are not relevant to a particular implementation are UNPREDICTABLE. All unused values of opcode_2 and CRm are UNPREDICTABLE.

Not all functions are provided by all implementations. Reading CP15 register 7 is UNPREDICTABLE.

<table>
<thead>
<tr>
<th>Function</th>
<th>opcode_2 value</th>
<th>CRm value</th>
<th>Data</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flush ID cache(s)</td>
<td>0b000</td>
<td>0b0111</td>
<td>SBZ</td>
<td>MCR p15, 0, Rd, c7, c7, 0</td>
</tr>
<tr>
<td>Flush ID single entry</td>
<td>0b001</td>
<td>0b0111</td>
<td>IMP</td>
<td>MCR p15, 0, Rd, c7, c7, 1</td>
</tr>
<tr>
<td>Flush I cache</td>
<td>0b000</td>
<td>0b0101</td>
<td>SBZ</td>
<td>MCR p15, 0, Rd, c7, c5, 0</td>
</tr>
<tr>
<td>Flush I single entry</td>
<td>0b001</td>
<td>0b0101</td>
<td>IMP</td>
<td>MCR p15, 0, Rd, c7, c5, 1</td>
</tr>
<tr>
<td>Flush D cache</td>
<td>0b000</td>
<td>0b0110</td>
<td>SBZ</td>
<td>MCR p15, 0, Rd, c7, c6, 0</td>
</tr>
<tr>
<td>Flush D single entry</td>
<td>0b001</td>
<td>0b0110</td>
<td>IMP</td>
<td>MCR p15, 0, Rd, c7, c6, 1</td>
</tr>
<tr>
<td>Clean ID cache</td>
<td>0b000</td>
<td>0b1011</td>
<td>SBZ</td>
<td>MCR p15, 0, Rd, c7, c11, 0</td>
</tr>
<tr>
<td>Clean ID cache entry</td>
<td>0b001</td>
<td>0b1011</td>
<td>IMP</td>
<td>MCR p15, 0, Rd, c7, c11, 1</td>
</tr>
<tr>
<td>Clean D cache</td>
<td>0b000</td>
<td>0b1010</td>
<td>SBZ</td>
<td>MCR p15, 0, Rd, c7, c10, 0</td>
</tr>
<tr>
<td>Clean D cache entry</td>
<td>0b001</td>
<td>0b1010</td>
<td>IMP</td>
<td>MCR p15, 0, Rd, c7, c10, 1</td>
</tr>
<tr>
<td>Clean and Flush ID cache</td>
<td>0b000</td>
<td>0b1111</td>
<td>SBZ</td>
<td>MCR p15, 0, Rd, c7, c15, 0</td>
</tr>
<tr>
<td>Clean and Flush ID entry</td>
<td>0b001</td>
<td>0b1111</td>
<td>IMP</td>
<td>MCR p15, 0, Rd, c7, c15, 1</td>
</tr>
<tr>
<td>Clean and Flush D cache</td>
<td>0b000</td>
<td>0b1110</td>
<td>SBZ</td>
<td>MCR p15, 0, Rd, c7, c14, 0</td>
</tr>
<tr>
<td>Clean and Flush D entry</td>
<td>0b001</td>
<td>0b1110</td>
<td>IMP</td>
<td>MCR p15, 0, Rd, c7, c14, 1</td>
</tr>
<tr>
<td>Flush Prefetch Buffer</td>
<td>0b100</td>
<td>0b0101</td>
<td>SBZ</td>
<td>MCR p15, 0, Rd, c7, c5, 0</td>
</tr>
<tr>
<td>Drain Write Buffer</td>
<td>0b100</td>
<td>0b1010</td>
<td>SBZ</td>
<td>MCR p15, 0, Rd, c7, c10, 0</td>
</tr>
<tr>
<td>Flush Branch Target Cache</td>
<td>0b110</td>
<td>0b0101</td>
<td>SBZ</td>
<td>MCR p15, 0, Rd, c7, c5, 0</td>
</tr>
<tr>
<td>Flush Branch Target Entry</td>
<td>0b111</td>
<td>0b0101</td>
<td>IMP</td>
<td>MCR p15, 0, Rd, c7, c5, 0</td>
</tr>
</tbody>
</table>

*Table 7-2: Cache functions*
7.4.9 Register 8: TLB functions

Writing to CP15 register 8 is used to control Translation Lookaside Buffers (TLBs). An ARM implementation may include a combined instruction and data TLB, or separate instruction and data TLBs. Several TLB functions are defined, and the function to be performed is selected by the opcode_2 and CRm fields in the MCR instruction used to write CP15 register 6.

Not all functions are provided by all implementations.

The Flush ID functions flush (invalidate) all TLB data.

The Flush I and Flush D functions are intended for use on implementations with split instruction and data TLBs; if used on an implementation with a combined TLB, the behaviour is as if a Flush ID function was used.

The Flush Entry functions may be implemented to flush from more than a single entry to the entire TLB.

Any functions that are not relevant to a particular implementation are UNPREDICTABLE. All unused values of opcode_2 and CRm are UNPREDICTABLE.

Reading CP15 register 8 is UNPREDICTABLE.

<table>
<thead>
<tr>
<th>Function</th>
<th>opcode_2</th>
<th>CRm</th>
<th>Data</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flush ID TLB(s)</td>
<td>0b000</td>
<td>0b0111</td>
<td>SBZ</td>
<td>MCR p15, 0, Rd, c8, c7, 0</td>
</tr>
<tr>
<td>Flush ID single entry</td>
<td>0b001</td>
<td>0b0111</td>
<td>Virtual Address</td>
<td>MCR p15, 0, Rd, c8, c7, 1</td>
</tr>
<tr>
<td>Flush I TLB</td>
<td>0b000</td>
<td>0b0101</td>
<td>SBZ</td>
<td>MCR p15, 0, Rd, c8, c5, 0</td>
</tr>
<tr>
<td>Flush I single entry</td>
<td>0b001</td>
<td>0b0101</td>
<td>Virtual Address</td>
<td>MCR p15, 0, Rd, c8, c5, 1</td>
</tr>
<tr>
<td>Flush D TLB</td>
<td>0b000</td>
<td>0b0110</td>
<td>SBZ</td>
<td>MCR p15, 0, Rd, c8, c6, 0</td>
</tr>
<tr>
<td>Flush D single entry</td>
<td>0b001</td>
<td>0b0110</td>
<td>Virtual Address</td>
<td>MCR p15, 0, Rd, c8, c6, 1</td>
</tr>
</tbody>
</table>

Table 7-3: TLB functions

7.4.10 11-15: Reserved

Accessing (reading or writing) any of these registers is unpredictable.
7.5 ARMv3 System Control Coprocessor

The ARM Architecture Version 3 System Control Coprocessor is designed to control a single combined instruction and data cache, a write buffer, and a virtual to physical address translator including combined instruction and data TLB.

CP15 also controls various system configuration signals:

<table>
<thead>
<tr>
<th>Register</th>
<th>Reads</th>
<th>Writes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ID Register</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>1</td>
<td>UNPREDICTABLE</td>
<td>Control</td>
</tr>
<tr>
<td>2</td>
<td>UNPREDICTABLE</td>
<td>Translation Table Base</td>
</tr>
<tr>
<td>3</td>
<td>UNPREDICTABLE</td>
<td>Domain Access Control</td>
</tr>
<tr>
<td>4</td>
<td>UNPREDICTABLE</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>5</td>
<td>Fault Status</td>
<td>Flush TLB</td>
</tr>
<tr>
<td>6</td>
<td>Fault Address</td>
<td>Flush TLB Entry</td>
</tr>
<tr>
<td>7</td>
<td>UNPREDICTABLE</td>
<td>Flush Cache</td>
</tr>
<tr>
<td>8 to 15</td>
<td>UNDEFINED</td>
<td>UNDEFINED</td>
</tr>
</tbody>
</table>

Table 7-4: ARMv3 CP15 register summary

7.5.1 Register 0: ID register

Reading from CP15 register 0 returns an architecture and IMPLEMENTATION DEFINED identification for the processor. The CRm and opcode_2 fields are IGNORED when reading CP15 register 0, and SHOULD BE ZERO.

- Bits[3:0] contain the revision number for the processor
- Bits[15:4] contain a 3 digit part number in binary coded decimal format (for example, 0x700 for ARM700)
- Bits[23:16] contain the architecture version (for example, 0x00 = Version 3)
- Bits[31:24] contain the ASCII code of an implementor’s trademark (for example, 0x41 = A = ARM Ltd.)

Writing to CP15 register 0 is unpredictable.
7.5.2 Register 1: Control Register

Reading from CP15 register 1 is UNPREDICTABLE.

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

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

- **M Bit 0**: Memory Management Unit (MMU) Enable/Disable
  - 0 = MMU disabled
  - 1 = MMU enabled

- **A Bit 1**: Alignment Fault Enable/Disable
  - 0 = Address alignment fault checking disabled
  - 1 = Address alignment fault checking enabled

- **C Bit 2**: Instruction and data cache Enable/Disable
  - 0 = Instruction and data cache (IDC) disabled
  - 1 = Instruction and data cache (IDC) enabled

- **W Bit 3**: Write buffer Enable/Disable
  - 0 = Write buffer disabled
  - 1 = Write buffer enabled

- **P Bit 4**: 32-bit/26-bit Exception handlers
  - Implementations that support 26-bit configurations use this bit to control the PROG32 signal (see Chapter 5, The 26-bit Architectures)
  - 0 = 26-bit exception handlers
  - 1 = 32-bit exception handlers

- **D Bit 5**: 32-bit/26-bit data address range
  - Implementations that support 26-bit data spaces use this bit to control the DATA32 signal (Chapter 5, The 26-bit Architectures)
  - 0 = 26-bit data address checking enabled
  - 1 = 26-bit data address checking disabled (32-bit data addresses)

- **L Bit 6**: IMPLEMENTATION DEFINED.

- **B Bit 7**: Big endian/Little endian
  - 0 = Little endian operation
  - 1 = Big endian operation

- **S Bit 8**: System protection
  - This bit modifies the MMU protection system.

- **R Bit 9**: ROM protection
  - This bit modifies the MMU protection system.

- **F Bit 10**: IMPLEMENTATION DEFINED

- **Bits 31:11**: When read, these bits return an UNPREDICTABLE value; when written, SHOULD BE ZERO.
Enabling the MMU

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

The correct code sequence for enabling and disabling the MMU is IMPLEMENTATION DEFINED.

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

7.5.3 Register 2: Translation Table Base Register

Reading from CP15 register 2 is UNPREDICTABLE.

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

7.5.4 Register 3: Domain Access Control Register

Reading from CP15 register 3 is UNPREDICTABLE.

Writing to CP15 register 3 writes the value of the Domain Access Control Register. The CRm and opcode_2 fields are IGNORED when writing CP15 register 3, and SHOULD BE ZERO.

The Domain Access Control Register consists of sixteen 2-bit fields, each of which defines the access permissions for one of the sixteen Domains (D15-D0). For the meaning of each field, see 7.9 Domains on page 7-23.

7.5.5 Register 4: Reserved

Reading and writing CP15 register 4 is UNPREDICTABLE.
7.5.6 Register 5: Fault Status Register and Flush TLB

Reading CP15 register 5 returns the value of the Fault Status Register (FSR). The FSR contains the source of the last data fault. Note that only the bottom 9 bits are returned. The upper 23 bits are UNPREDICTABLE. The FSR indicates the domain and type of access being attempted when an abort occurred:

- Bit 8 returns zero
- Bits 7:4 specify which of the 16 domains (D15-D0) was being accessed when a fault occurred
- Bits 3:0 indicate the type of access being attempted

The encoding is shown in Table 7-9: Priority encoding of fault status on page 7-25. The FSR is only updated for data faults, not for prefetch faults. The CRm and opcode_2 fields are IGNORED when reading CP15 register 5, and SHOULD BE ZERO.

Writing CP15 register 5 flushes the TLB. An ARMv3 implementation may only include a combined instruction and data TLB, and not separate instruction and data TLBs. The data written to the register is IGNORED, and SHOULD BE ZERO.

7.5.7 Register 6: Fault Address Register and Flush TLB Entry

Reading CP15 register 6 returns the value of the Fault Address Register (FAR). The FAR holds the virtual address of the access which was attempted when a fault occurred. The FAR is only updated for data faults, not for prefetch faults. The CRm and opcode_2 fields are IGNORED when reading CP15 register 6, and SHOULD BE ZERO.

Writing CP15 register 6 flushes a single entry from the TLB. An ARMv3 implementation may only include a combined instruction and data TLB, and not separate instruction and data TLBs. The data written to the register is the virtual address to be flushed.

7.5.8 Register 7: Flush Cache

Reading CP15 register 7 is UNPREDICTABLE.

Writing to CP15 register 7 is used to flush the instruction and data cache. An ARMv3 implementation may only include a combined instruction and data cache, and not separate instruction and data caches. The data written to the register is IGNORED, and SHOULD BE ZERO.

7.5.9 Registers 8-15: Reserved

Accessing (reading or writing) any of these registers will cause an undefined instruction exception.
7.6 Memory Management Unit (MMU) Architecture

7.6.1 Overview

The ARM MMU performs two primary functions:

- it translates virtual addresses into physical addresses
- it controls memory access permissions

The MMU hardware required to perform these functions consists of:

- at least one Translation Lookaside Buffer (TLB)
- access control logic
- translation-table-walking logic

For implementations with separate Instruction and Data caches, separate TLBs for instruction and data are also likely.

The translation lookaside buffer

The TLB caches virtual to physical address translations and access permissions for each translation. If the TLB contains a translated entry for the virtual address, the access control logic determines whether access is permitted. If access is permitted, the MMU outputs the appropriate physical address corresponding to the virtual address. If access is not permitted, the MMU signals the CPU to abort.

If the TLB misses (it does not contain a translated entry for the virtual address), the translation table walk hardware is invoked to retrieve the translation and access permission information from a translation table in physical memory. Once retrieved, the information is placed into the TLB, possibly overwriting an existing entry.

Memory accesses

The MMU supports memory accesses based on sections or pages:

- **Sections** are comprised of 1MB blocks of memory
- **Pages** Two different page sizes are supported:
  - **small pages** consist of 4kB blocks of memory
  - **large pages** consist of 64kB blocks of memory

Sections and large pages are supported to allow mapping of a large region of memory while using only a single entry in the TLB. Additional access control mechanisms are extended within small pages to 1kB sub-pages and within large pages to 16kB sub-pages.

Translation table

The translation table held in main memory has two levels:

- **first-level table** holds both section translations and pointers to second-level tables
- **second-level tables** hold both large and small page translations
Domains

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

When the MMU is turned off (as happens on reset), the virtual address is output directly as the physical address, and no memory access permission checks are performed.

It is **UNPREDICTABLE** if two TLB entries address overlapping areas of memory. This can occur if the TLB is not flushed after memory is re-mapped with different-sized pages (leaving an old mapping with different sizes in the TLB, and a new mapping gets loaded into a different TLB location).

7.6.2 Translation process

The MMU translates virtual addresses generated by the CPU into physical addresses to access external memory, and also derivates and checks the access permission. There are three routes by which the address translation (and hence permission check) takes place. The route taken depends on whether the address in question has been marked as a section-mapped access or a page-mapped access; and there are two sizes of page-mapped access (large pages and small pages).

However, the translation process always starts out in the same way, as described below, with a first-level fetch. A section-mapped access only requires a first level fetch, but a page-mapped access also requires a *second-level* fetch.

7.6.3 Translation table base

The translation process is initiated when the on-chip TLB does not contain an entry for the requested virtual address. The Translation Table Base Register points to the base of the first-level table. Only bits 31 to 14 of the Translation Table Base Register are significant; bits 13 to 0 should be zero. Therefore, the first-level page table must reside on a 16Kbyte boundary.

7.6.4 First-level fetch

Bits 31:14 of the Translation Table Base register are concatenated with bits 31:20 of the virtual address to produce a 30-bit address as illustrated in *Figure 7-1: Accessing the translation table first-level descriptors* on page 7-16. This address selects a four-byte translation table entry which is a first-level descriptor for a section or a pointer to a second-level page table.
7.6.5 First-level descriptors

The first-level descriptor may define either a section descriptor or a pointer to a second-level page table and its format varies accordingly. Figure 7-2: First-level descriptor format shows the format, bits[1:0] indicate the descriptor type and validity.

Accessing a descriptor that has bits[1:0] = 0b00 generates a translation fault (see 7.10 Aborts on page 7-24).

Accessing a descriptor that has bits[1:0] = 0b11 is UNPREDICTABLE.
7.6.6 Section descriptor and translating section references

If the first-level descriptor is a section descriptor, the fields have the following meanings:

- **Bits 1:0** Identify the type of descriptor (0b10 marks a section descriptor).
- **Bit 3:2** The cachable and bufferable bits. See 7.7 Cache and Write Buffer Control on page 7-22.
- **Bit 4** The meaning of this bit is IMPLEMENTATION DEPENDENT.
- **Bits 8:5** The domain field specifies one of the sixteen possible domains for all the pages controlled by this descriptor.
- **Bits 9** This bit is not currently used, and SHOULD BE ZERO.
- **Bits 11:10** Access permissions. These bits control the access to the section. See Table 7-7: Access permissions on page 7-23 for the interpretation of these bits.
- **Bits 19:12** These bits are not currently used, and SHOULD BE ZERO.
- **Bits 31:20** The Section Base Address forms the top 12 bits of the physical address.

*Figure 7-3: Section translation* illustrates the complete section translation sequence. Note that the access permissions contained in the first-level descriptor must be checked before the physical address is generated. The sequence for checking access permissions is described in 7.8 Access Permissions on page 7-22.
7.6.7 Page table descriptor

If the first-level descriptor is a page table descriptor, the fields have the following meanings:

- **Bits 1:0** Identify the type of descriptor (0b01 marks a page table descriptor).
- **Bit 4:2** The meaning of these bits is IMPLEMENTATION DEPENDENT.
- **Bits 8:5** The domain field specifies one of the sixteen possible domains for all the pages controlled by this descriptor.
- **Bits 9** This bit is not currently used, and SHOULD BE ZERO.
- **Bits 31:10** The Page Table Base Address is a pointer to a second-level page table, giving the base address for a second level fetch to be performed. Second level page tables must be aligned on a 1Kbyte boundary.

If a page table descriptor is returned from the first-level fetch, a second-level fetch is initiated to retrieve a second level descriptor, as shown in Figure 7-4: Accessing the translation table second-level descriptors.

![Figure 7-4: Accessing the translation table second-level descriptors](image-url)
7.6.8 Second-level descriptor

The second-level descriptor may define either a large page or a small page access. Table 7-5: Second-level descriptor format shows the format; bits[1:0] indicate the descriptor type and validity.

Accessing a descriptor that has bits[1:0] = 0b00 generates a translation fault (see 7.10 Aborts on page 7-24).

Accessing a descriptor that has bits[1:0] = 0b11 is UNPREDICTABLE.

<table>
<thead>
<tr>
<th>Field</th>
<th>Large page base address</th>
<th>Small page base address</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault</td>
<td>SBZ</td>
<td>SBZ</td>
<td>1 1</td>
</tr>
<tr>
<td>Access permissions</td>
<td>0 0</td>
<td>AP3 AP2 AP1 AP0 C B 0 1</td>
<td></td>
</tr>
<tr>
<td>Large page</td>
<td>Large page base address</td>
<td>AP3 AP2 AP1 AP0 C B 0 1</td>
<td></td>
</tr>
<tr>
<td>Small page</td>
<td>Small page base address</td>
<td>AP3 AP2 AP1 AP0 C B 1 0</td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td></td>
<td>0 0</td>
</tr>
</tbody>
</table>

Table 7-5: Second-level descriptor format

The fields in both large and small pages have the following meanings:

- Bits 1:0 identifies the type of descriptor
- Bits 2:3 The cachable and bufferable bits. See 7.7 Cache and Write Buffer Control on page 7-22.
- Bits 11:4 Access permissions
  - These bits control access to the page. See Table 7-7: Access permissions on page 7-23 for the interpretation of these bits.
  - Both large and small pages are split into four sub-pages:
    - AP0 encodes the access permissions for the first sub-page
    - AP1 encodes the access permissions for the second sub-page
    - AP2 encodes the access permissions for the third sub-page
    - AP3 encodes the access permissions for the fourth (last) sub-page
- Bits 15:12 are not currently used for large pages, and must be zero
- Bits 31:12 are used to form the corresponding bits of the physical address (small pages)
- Bits 31:16 are used to form the corresponding bits of the physical address (large pages)
7.6.9 Translating large page references

Figure 7-5: Large page translation shows the complete translation sequence for a 64Kbyte large page.

Note As the upper four bits of the Page Index and low-order four bits of the Second-level Table Index overlap, each page table entry for a large page must be duplicated 16 times (in consecutive memory locations) in the page table.
7.6.10 Translating small page references

Figure 7-6: Small page translation shows the complete translation sequence for a 4Kbyte small page.

![Diagram of small page translation](image_url)
7.7 Cache and Write Buffer Control

The ARM memory system is controlled by two attributes which are individually selectable for each virtual page:

**Cacheable**
This attribute indicates that data in the page may be cached, so that subsequent accesses may not access main memory. Cacheable also indicates that instruction speculative prefetching beyond the current point of execution may be performed. The cache implementation may use a write-back or a write-through policy (or a choice of either for individual virtual pages).

**Bufferable**
This attribute indicates that data in the page may be stored in the write buffer, allowing faster write operations for processors that operate faster than main memory. The write buffer may not preserve strict write ordering, and may not ensure that multiple writes to the same location result in multiple off-chip writes.

The Cacheable and Bufferable bits in the Section and Page descriptors control caching and buffering.

<table>
<thead>
<tr>
<th>C</th>
<th>B</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Uncached, Unbuffered</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Uncached, Buffered</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Cached, Unbuffered or Writethrough cached, Buffered</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Cached, Buffered or Writeback cached, Buffered</td>
</tr>
</tbody>
</table>

Table 7-6: Cache and bufferable bit meanings

Implementations that offer both writethrough or writeback caching use the 0b10 value to specify writethrough caching, and the 0b11 value to specify writeback caching.

Implementations that only offer one type of cache behaviour (writeback or writethrough) use the C and B bits strictly as cache enable and write buffer enable respectively.

Note that writeback cache implementations that do not also support writethrough caching, may not provide cached, unbuffered memory (as the writeback cache effectively buffers writes).

7.8 Access Permissions

The access permission bits in section and page descriptors control access to the corresponding section or page. The access permissions are modified by the System (S) and ROM (R) control bits. Table 7-7: Access permissions on page 7-23 describes the meaning of the access permission bits in conjunction with the S and R bits. If an access is made to an area of memory without the required permission, a Permission Fault is raised; see 7.10 Aborts on page 7-24.
7.9 Domains

A domain is a collection of sections, large pages and small pages. The ARM architecture supports 16 domains; access to each domain is controlled by a 2-bit field in the Domain Access Control Register. Each field allows the access to an entire domain to be enabled and disabled very quickly, so that whole memory areas can be swapped in and out of virtual memory very efficiently.

Two kinds of domain access are supported:

Clients are users of domains (execute programs, access data), and are guarded by the access permissions of the individual sections and pages that make up the domain.

Managers control the behaviour of the domain (the current sections and pages in the domain, and the domain access), and are not guarded by the access permissions of individual sections and pages in the domain.

One program can be a client of some domains, and a manager of some other domains, and have no access to the remaining domains. This allows very flexible memory protection for programs that access different memory resources. Table 7-8: Domain Access Values illustrates the encoding of the bits in the Domain Access Control Register.

<table>
<thead>
<tr>
<th>Value</th>
<th>Access Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>No Access</td>
<td>Any access will generate a domain fault</td>
</tr>
<tr>
<td>0b01</td>
<td>Client</td>
<td>Accesses are checked against the access permission bits in the section or page descriptor</td>
</tr>
<tr>
<td>0b10</td>
<td>Reserved</td>
<td>Using this value has unpredictable results</td>
</tr>
<tr>
<td>0b11</td>
<td>Manager</td>
<td>Accesses are not checked against the access permission bits in the section or page descriptor, so a permission fault cannot be generated</td>
</tr>
</tbody>
</table>

Table 7-8: Domain Access Values
The mechanisms that can cause the ARM processor to halt execution because of memory access restrictions are:

- **MMU fault**: The MMU detects the restriction and signals the processor.
- **External abort**: The external memory system signals an illegal memory access.

Collectively, MMU faults and external aborts are just called aborts. Accesses that cause aborts are said to be aborted.

If the memory request that is aborted is an instruction fetch, then a Prefetch Abort Exception is raised if and when the processor attempts to execute the instruction corresponding to the illegal access. If the aborted access is a data access, a Data Abort Exception is raised. See 2.5 Exceptions on page 2-6 for more information about Prefetch and Data Aborts.

### 7.11 MMU Faults

The MMU generates four types of faults:

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

The memory system may abort three types of access:

- Line Fetches
- Memory Accesses (uncached or unbuffered accesses)
- Translation Table Accesses

Aborts that are detected by the MMU are stopped before any external memory access takes place. It is the responsibility of the external system to stop external accesses that cause external aborts.

The System Control coprocessor contains two registers which are updated when a data access is aborted. These registers are not updated for prefetch aborts, as the aborted instruction may not be executed due to changes in program flow.
Aborts resulting from data accesses (data aborts) are immediately acted upon by the CPU. The Fault Status Register (FSR) is updated with a 4-bit Fault Status (FS[3:0]) and the domain number of the access. In addition, the virtual address which caused the data abort is written into the Fault Address Register (FAR). If a data access simultaneously generates more than one type of data abort, they are prioritised in the order given in Table 7-9: Priority encoding of fault status.

Aborts arising from instruction fetches are simply flagged as the instruction enters the instruction pipeline. Only when (and if) the instruction is executed does it cause a prefetch abort; a prefetch abort is not acted upon if the instruction is not used (e.g. it is branched around). Because instruction prefetch aborts may or may not be acted upon, the FSR and FAR are not updated (the value of the PC saved in R14_abt after the exception occurs can be used to calculate the fault address).

<table>
<thead>
<tr>
<th>Priority</th>
<th>Sources</th>
<th>FS[3:0]</th>
<th>Domain[3:0]</th>
<th>FAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highest</td>
<td>Terminal Exception</td>
<td>0b0010</td>
<td>invalid</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
<tr>
<td></td>
<td>Vector Exception</td>
<td>0b0000</td>
<td>invalid</td>
<td>valid</td>
</tr>
<tr>
<td></td>
<td>Alignment</td>
<td>0b00x1</td>
<td>invalid</td>
<td>valid</td>
</tr>
<tr>
<td></td>
<td>External Abort on Translation</td>
<td>0b1100</td>
<td>invalid</td>
<td>valid</td>
</tr>
<tr>
<td></td>
<td>First level</td>
<td>0b1110</td>
<td>valid</td>
<td>valid</td>
</tr>
<tr>
<td></td>
<td>Second level</td>
<td>0b1111</td>
<td>valid</td>
<td>valid</td>
</tr>
<tr>
<td></td>
<td>Translation</td>
<td>0b0101</td>
<td>invalid</td>
<td>valid</td>
</tr>
<tr>
<td></td>
<td>Section</td>
<td>0b0111</td>
<td>valid</td>
<td>valid</td>
</tr>
<tr>
<td></td>
<td>Page</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Domain</td>
<td>0b1001</td>
<td>valid</td>
<td>valid</td>
</tr>
<tr>
<td></td>
<td>Section</td>
<td>0b1011</td>
<td>valid</td>
<td>valid</td>
</tr>
<tr>
<td></td>
<td>Page</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Permission</td>
<td>0b1101</td>
<td>valid</td>
<td>valid</td>
</tr>
<tr>
<td></td>
<td>Section</td>
<td>0b1111</td>
<td>valid</td>
<td>valid</td>
</tr>
<tr>
<td></td>
<td>Page</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>External Abort on Linefetch</td>
<td>0b0100</td>
<td>valid</td>
<td>valid</td>
</tr>
<tr>
<td></td>
<td>Section</td>
<td>0b0110</td>
<td>valid</td>
<td>valid</td>
</tr>
<tr>
<td></td>
<td>Page</td>
<td>0b0111</td>
<td>valid</td>
<td>valid</td>
</tr>
<tr>
<td>Lowest</td>
<td>External Abort on Non-linefetch</td>
<td>0b1000</td>
<td>valid</td>
<td>valid</td>
</tr>
<tr>
<td></td>
<td>Section</td>
<td>0b1010</td>
<td>valid</td>
<td>valid</td>
</tr>
<tr>
<td></td>
<td>Page</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7-9: Priority encoding of fault status

Notes

1. Alignment faults may write either 0b0001 or 0b0011 into FS[3:0].
2. Invalid values in Domain[3:0] occur because the fault is raised before a valid domain field has been loaded.
3. Any abort masked by the priority encoding may be regenerated by fixing the primary abort and restarting the instruction.
4. The FS[3:0] encoding for Vector Exception breaks from the pattern that FS[0] is zero for all external aborts.
7.11.2 Fault-checking sequence

The sequence by which the MMU checks for access faults is slightly different for Sections and Pages. Figure 7-7: Sequence for checking faults on page 7-27 illustrates the sequence for both types of access. The sections and figures that follow describe the conditions that generate each of the faults.

7.11.3 Vector Exceptions

When the processor is in a 32-bit configuration (PROG32 is active) and in a 26-bit mode (CPSR[4] == 0), data access (but not instruction fetches) to the hard vectors (address 0x0 to 0x1f) will cause a data abort, known as a vector exception. See 7.11.3 Vector Exceptions on page 7-26 for a full description. It is IMPLEMENTATION DEFINED if vector exceptions are generated when the MMU is not enabled.

7.11.4 Alignment fault

If Alignment Faults are enabled, an alignment fault will be generated on any data word access whose address is not word-aligned (virtual address bits [1:0] != 0b00), or any halfword access that is not halfword-aligned (virtual address bit[0] != 0). Alignment faults will not be generated on any instruction fetch, or on any byte access.

Note that if the access generates an alignment fault, the access will be aborted without reference to further permission checks. It is IMPLEMENTATION DEFINED if alignment exceptions are generated when the MMU is not enabled.

7.11.5 Translation fault

There are two types of translation fault:

- **Section** is generated if the first-level descriptor is marked as invalid. This happens if bits[1:0] of the descriptor are both 0.
- **Page** is generated if the second-level descriptor is marked as invalid. This happens if bits[1:0] of the descriptor are both 0.

7.11.6 Domain fault

There are two types of domain fault:

- **Section**
- **Page**

In both cases, the first descriptor holds the 4-bit Domain field which selects one of the sixteen 2-bit domains in the Domain Access Control Register. The two bits of the specified domain are then checked for access permissions as detailed in Table 7-8: Domain Access Values on page 7-23.

In the case of a section, the domain is checked when the first-level descriptor is returned, and in the case of a page, the domain is checked when the second-level descriptor is returned. If the specified access is marked as No Access in the Domain Access Control Register, either a Section Domain Fault or Page Domain Fault occurs.
Figure 7-7: Sequence for checking faults

- Virtual address
  - Checking alignment?
    - Yes: Check address alignment
    - No: Misaligned?
      - Yes: Alignment fault
      - No: Get first-level descriptor
  - Alignment fault

- Get first-level descriptor
  - Translation external abort
    - Yes: External abort
    - No: Section translation fault
  - Descriptor fault
    - Yes: Section translation fault
    - No: Section or page?

- Section or page?
  - Section
    - Translation external abort
      - Yes: External abort
      - No: Invalid descriptor?
        - Yes: Page translation fault
        - No: Sub-page permission fault
  - Page
    - Translation external abort
      - Yes: Sub-page permission fault
      - No: Invalid descriptor?
        - Yes: Page translation fault
        - No: Access type
          - No: Check access permissions
          - Yes: Check access permissions

- Access type
  - No: Section domain fault
    - No: Access type
      - No: Section domain fault
      - Yes: Section permission fault
        - No: No Access
        - Yes: Access type
          - No: Check access permissions
          - Yes: Check access permissions

- Check access permissions
  - Yes: Violation?
    - No: No
    - Yes: Violation?
      - No: No
      - Yes: Yes

- Physical address
  - Yes: Yes
  - No: No
There are section permission faults and sub-page permission faults.

Permission faults are checked at the same time as domain faults. If the 2-bit domain field returns client (01), the permission access check is invoked as follows:

**Section**
If the first-level descriptor defines a section access, the AP bits of the descriptor define whether or not the access is allowed, according to Table 7-7: Access permissions on page 7-23. If the access is not allowed, a Section Permission fault is generated.

**Sub-page**
If the first-level descriptor defines a page-mapped access, the second-level descriptor specifies four access permission fields (ap3, ap2, ap1, ap0) each corresponding to one quarter of the page. For small pages, ap3 is selected by the top 1kB of the page, and ap0 is selected by the bottom 1kB of the page. For large pages, ap3 is selected by the top 16kB of the page, and ap0 is selected by the bottom 16kB of the page. The selected AP bits are then interpreted in exactly the same way as for a section, (see Table 7-7: Access permissions on page 7-23) the only difference being that the fault generated is a Sub-page Permission fault.

### 7.12 External Aborts

In addition to the MMU faults, the ARM Architecture defines an external abort pin which may be used to flag an error on an external memory access. However, not all accesses can be aborted in this way, so this pin must be used with great care. The following accesses may be externally aborted and restarted safely:

- Reads
- Unbuffered writes
- First-level descriptor fetch
- Second-level descriptor fetch
- Multi-bus master semaphores

A linefetch may be safely aborted on any word in the line transfer. If the abort happens on data that has been requested by the processor (rather than data that is being fetched as the remainder of a cache line), the access will be aborted. Any data transferred that is not immediately accessed (the remainder of the cache line) will only be aborted when it is accessed.

It is IMPLEMENTATION-DEFINED if the FAR points to the start address of the cache line, or the address that generated the abort.

Buffered writes cannot be externally aborted. Therefore, the system must be configured such that it does not do buffered writes to areas of memory which are capable of flagging an external abort, or a different mechanism should be used to signal the abort (an interrupt for example).

The contents of a memory location that causes an abort is UNPREDICTABLE after the abort.
7.13 System-level Issues

This section lists a number of issues that need to be addressed by the system designer and operating systems to provide an ARMv4 compatible system.

7.13.1 Memory systems, write buffers and caches

ARMv4 processors and software expect to be connected to a byte-addressed memory. Word and halfword accesses to the memory will ignore the alignment of the address and return the naturally-aligned value that is addressed (so a memory access will ignore address bits 0 and 1 for word access, and will ignore bit 0 for halfword accesses). ARMv4 processors must implement some method for switching between big-endian and little-endian addressing of the memory system (if CP15 is implemented, bit 7 of register 1 controls endianness). It is IMPLEMENTATION DEFINED if the endianness can be changed dynamically.

Memory that is used to hold programs and data will be marked as follows:

- **Main (RAM) memory** will normally be set as cacheable and bufferable
- **ROM memory** will normally be set as cacheable, and will be marked as read only (so the bufferable attribute is not used, and SHOULD BE ONE)

**Write buffers**

An ARMv4 implementation may incorporate a merging write buffer, that subsumes multiple writes to the same location into a single write to main memory. Furthermore, a write buffer may re-order writes, so that writes are issued to memory in a different order to the order in which they are issued by the processor. Thus IO locations should never be marked as bufferable, to ensure all writes are issued, and in the correct order, to the IO device.

**Caches**

Frame buffers may be cacheable, but frame buffers on writeback cache implementations must be copied back to memory after the frame buffer has been updated. Frame buffers may be bufferable, but again the write buffer must be written back to memory after the frame buffer has been updated.

ARMv4 does not support cache coherency between the ARM and other system bus masters (bus snooping is not supported). Memory data that is shared between multiple bus masters should be mapped as uncacheable to ensure that all reads access main memory, and unbufferable to ensure all writes do access main memory. IO devices that are mapped into the memory map should be marked as uncacheable and unbufferable.

The coherency of data buffers that are read or written by another bus master may be managed in software, by cleaning data from writeback caches and write buffers to memory when the processor has written to the data buffer and before the other bus reads the buffer, and flushing relevant data from caches when the buffer is being read after the other bus master has written the buffer. An uncached, unbuffered semaphore can be used to maintain synchronisation between multiple bus masters. See 7.14 Semaphores on page 7-31.
For implementations with writeback caches, all dirty cache data must be written back before any alterations are made to the MMU page tables, to ensure that cache line write back may use the page tables to form the correct physical address for the transfer.

Caches may be indexed using either virtual or physical addresses. Physical pages must only be mapped into a single virtual page, otherwise the result is unpredictable. ARMv4 does not provide coherency between multiple virtual copies of a single physical page.

Some ARM implementations support separate instruction and data caches. The coherency between the data and instruction cache may not be maintained in hardware, so if the instruction stream is written, the instruction cache and data cache must be made coherent. This may entail cleaning the data cache (storing dirty data to memory), draining the write buffer (completing all buffered writes), and flushing the instruction cache. Instruction and data memory incoherency occurs after a program has been loaded (and thus treated as data) and is about to be executed, or if self-modifying code is used or generated.

### 7.13.2 Interrupts

ARM processors implement fast and normal levels of interrupt.

Both interrupts are signalled externally, and many implementations will synchronise interrupts before an exception is raised. A fast interrupt request (FIQ) will disable subsequent normal and fast interrupts by setting the I and F bits in the CPSR, and a normal interrupt request (IRQ) will disable subsequent normal interrupts by setting the I bit in the CPSR. See 2.5 Exceptions on page 2-6.

#### Cancelling interrupts

It is the responsibility of software (the interrupt handler) to ensure that the cause of an interrupt is cancelled (no longer signalled to the processor) before interrupts are re-enabled (by clearing the I and/or F bit in the CPSR). Interrupts may be cancelled with any instruction that may make an external data bus access; that is, any load or store, a swap, or any coprocessor instruction.

Cancelling an interrupt via an instruction fetch is UNPREDICTABLE.

Cancelling an interrupt with a load multiple that restores the CPSR and re-enables interrupts is UNPREDICTABLE.

Devices that do not instantaneously cancel an interrupt (i.e. they do not cancel the interrupt before letting the access complete) should be probed by software to ensure that interrupts have been cancelled before interrupts are re-enabled. This allows a device connected to a remote IO bus to operate correctly.
7.14 Semaphores

The Swap and Swap Byte instructions have predictable behaviour when used in two ways:

- Multi-bus master systems that use the Swap instructions to implement semaphores to control interaction between different bus masters.
  In this case, the semaphores must be placed in an uncached and unbufferable region of memory. The Swap instruction will then cause a (locked) read-write bus transaction.
  This type of semaphore may be externally aborted.

- Systems with multiple threads running on a uni-processor that use the Swap instructions to implement semaphores to control interaction of the threads.
  In this case, the semaphores may be placed in a cached and bufferable region of memory, and a (locked) read-write bus transaction may not occur.
  This system is likely to have better performance than the multi bus master system above.
  This type of semaphore has unpredictable behaviour if it is externally aborted.

Semaphores placed in non-cacheable/bufferable memory regions have UNPREDICTABLE results. Semaphores placed in cacheable/non-bufferable memory regions have UNPREDICTABLE results.
## Numerics

**16-bit instructions** 6-1  
**26-bit**  
addressing 5-8  
architectures 5-1  
  - address exceptions 5-6  
  - compatibility with 32-bit architectures 5-7  
  - program counter 5-3  
  - program status register 5-4  
configuration 5-8  
**32-bit**  
address space 5-2  
addressing 5-8  
architecture 5-2  
  - compatibility with 26-bit architectures 5-7  
configuration 5-8  
immediate 3-87  
multiply 3-10  
**64-bit**  
multiply 3-10  
objects 4-3  
value 3-10, 3-30, 3-63 to 3-65, 3-80 to 3-81

## A

**aborts** 7-24  
  - external 7-28  
**access permissions** 7-22  
**ADC**  
  - ARM 3-30  
  - Thumb 6-19  
**ADD**  
  - ARM 3-31  
  - Thumb 6-20 to 6-26  
**add (ADD) instruction**  
  - ARM 3-31  
  - Thumb  
    - high registers 6-23  
    - immediate 6-20  
    - immediate to PC 6-24  
    - immediate to SP 6-25  
    - increment stack pointer 6-26  
    - large immediate 6-21  
    - registers 6-22  
**add with carry (ADC) instruction**  
  - ARM 3-30  
  - Thumb 6-19  
**address exceptions, 26-bit architectures** 5-6
<table>
<thead>
<tr>
<th>Addressing Modes</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2, 1-6</td>
<td></td>
</tr>
<tr>
<td>Data Processing</td>
<td></td>
</tr>
<tr>
<td>Arithmetic Shift Right by Immediate</td>
<td>3-93</td>
</tr>
<tr>
<td>Arithmetic Shift Right by Register</td>
<td>3-94</td>
</tr>
<tr>
<td>Immediate 3-87</td>
<td></td>
</tr>
<tr>
<td>Logical Shift Left by Immediate</td>
<td>3-89</td>
</tr>
<tr>
<td>Logical Shift Left by Register</td>
<td>3-90</td>
</tr>
<tr>
<td>Logical Shift Right by Immediate</td>
<td>3-91</td>
</tr>
<tr>
<td>Logical Shift Right by Register</td>
<td>3-92</td>
</tr>
<tr>
<td>Register 3-88</td>
<td></td>
</tr>
<tr>
<td>Rotate Right and Extend 3-97</td>
<td></td>
</tr>
<tr>
<td>Rotate Right by Immediate 3-95</td>
<td></td>
</tr>
<tr>
<td>Rotate Right by Register 3-96</td>
<td></td>
</tr>
<tr>
<td>Load and Store Coprocessor</td>
<td></td>
</tr>
<tr>
<td>Immediate Offset 3-124</td>
<td></td>
</tr>
<tr>
<td>Immediate Post-indexed 3-126</td>
<td></td>
</tr>
<tr>
<td>Immediate Pre-indexed 3-125</td>
<td></td>
</tr>
<tr>
<td>Load and Store Halfword or Load Signed Byte</td>
<td></td>
</tr>
<tr>
<td>Immediate Offset 3-110</td>
<td></td>
</tr>
<tr>
<td>Immediate Post-indexed 3-114</td>
<td></td>
</tr>
<tr>
<td>Immediate Pre-indexed 3-112</td>
<td></td>
</tr>
<tr>
<td>Register Offset 3-111</td>
<td></td>
</tr>
<tr>
<td>Register Post-indexed 3-115</td>
<td></td>
</tr>
<tr>
<td>Register Pre-indexed 3-113</td>
<td></td>
</tr>
<tr>
<td>Load and Store Multiple</td>
<td></td>
</tr>
<tr>
<td>Decrement After 3-119</td>
<td></td>
</tr>
<tr>
<td>Decrement Before 3-120</td>
<td></td>
</tr>
<tr>
<td>Increment After 3-117</td>
<td></td>
</tr>
<tr>
<td>Increment Before 3-118</td>
<td></td>
</tr>
<tr>
<td>Stack Operations 3-121</td>
<td></td>
</tr>
<tr>
<td>Load and Store Word or Unsigned Byte</td>
<td></td>
</tr>
<tr>
<td>Immediate Offset 3-100</td>
<td></td>
</tr>
<tr>
<td>Immediate Post-indexed Offset 3-106</td>
<td></td>
</tr>
<tr>
<td>Immediate Pre-indexed 3-103</td>
<td></td>
</tr>
<tr>
<td>Register Offset 3-101</td>
<td></td>
</tr>
<tr>
<td>Register Post-indexed Offset 3-107</td>
<td></td>
</tr>
<tr>
<td>Register Pre-indexed Offset 3-104</td>
<td></td>
</tr>
<tr>
<td>Scaled Register Offset 3-102</td>
<td></td>
</tr>
<tr>
<td>Scaled Register Post-indexed 3-108</td>
<td></td>
</tr>
<tr>
<td>Scaled Register Pre-indexed 3-105</td>
<td></td>
</tr>
<tr>
<td>Mode 1 3-84 to 3-97</td>
<td></td>
</tr>
<tr>
<td>Mode 2 3-98 to 3-108</td>
<td></td>
</tr>
<tr>
<td>Mode 3 3-109 to 3-115</td>
<td></td>
</tr>
<tr>
<td>Mode 4 3-116 to 3-122</td>
<td></td>
</tr>
<tr>
<td>Mode 5 3-123 to 3-126</td>
<td></td>
</tr>
<tr>
<td>Alignment Fault 7-26</td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td></td>
</tr>
<tr>
<td>ARM 3-32</td>
<td></td>
</tr>
<tr>
<td>Thumb 6-27</td>
<td></td>
</tr>
<tr>
<td>AND (AND) Instruction</td>
<td></td>
</tr>
<tr>
<td>ARM 3-32</td>
<td></td>
</tr>
<tr>
<td>Thumb 6-27</td>
<td></td>
</tr>
<tr>
<td>ARM Architecture</td>
<td></td>
</tr>
<tr>
<td>Architecture Version 4T (Thumb) 6-1</td>
<td></td>
</tr>
<tr>
<td>Arithmetic Instructions</td>
<td></td>
</tr>
<tr>
<td>1-4, 4-3</td>
<td></td>
</tr>
<tr>
<td>Examples 4-2</td>
<td></td>
</tr>
<tr>
<td>Extension Space 3-22</td>
<td></td>
</tr>
<tr>
<td>Arithmetic Shift Right (ASR) Instruction</td>
<td></td>
</tr>
<tr>
<td>ARM 3-93 to 3-94, 3-102, 3-105, 3-108, 3-115</td>
<td></td>
</tr>
<tr>
<td>By Immediate 3-93</td>
<td></td>
</tr>
<tr>
<td>By Register 3-94</td>
<td></td>
</tr>
<tr>
<td>Thumb By Immediate 6-28</td>
<td></td>
</tr>
<tr>
<td>By Register 6-29</td>
<td></td>
</tr>
<tr>
<td>ARM Addressing Modes</td>
<td></td>
</tr>
<tr>
<td>See Addressing Modes 1-2</td>
<td></td>
</tr>
<tr>
<td>Branch Instructions 3-6</td>
<td></td>
</tr>
<tr>
<td>Code 1-4, 3-35</td>
<td></td>
</tr>
<tr>
<td>Coprocessor Instructions 3-20</td>
<td></td>
</tr>
<tr>
<td>Data-processing Instructions 3-7, 3-8</td>
<td></td>
</tr>
<tr>
<td>Instruction Set 3-3</td>
<td></td>
</tr>
<tr>
<td>Extending 3-22</td>
<td></td>
</tr>
<tr>
<td>Load and Store Instructions 3-13</td>
<td></td>
</tr>
<tr>
<td>Multiply Instructions 3-10</td>
<td></td>
</tr>
<tr>
<td>Semaphore Instructions 3-19</td>
<td></td>
</tr>
<tr>
<td>Status Register Access Instructions 3-12</td>
<td></td>
</tr>
<tr>
<td>ARM Architecture 26-bit 5-1</td>
<td></td>
</tr>
<tr>
<td>Coprocessor 15 7-2</td>
<td></td>
</tr>
<tr>
<td>Implementations of 7-2</td>
<td></td>
</tr>
<tr>
<td>Overview 1-2</td>
<td></td>
</tr>
<tr>
<td>Version 4 System Control Coprocessor 7-2, 7-10</td>
<td></td>
</tr>
<tr>
<td>ASR</td>
<td></td>
</tr>
<tr>
<td>ARM 3-93 to 3-94, 3-102, 3-105, 3-108, 3-115</td>
<td></td>
</tr>
<tr>
<td>Thumb 6-28, 6-29</td>
<td></td>
</tr>
<tr>
<td>Atomic Update 1-6</td>
<td></td>
</tr>
</tbody>
</table>

**B**

<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>ARM 3-33</td>
<td></td>
</tr>
<tr>
<td>Thumb 6-30, 6-31</td>
<td></td>
</tr>
<tr>
<td>BIC</td>
<td></td>
</tr>
<tr>
<td>ARM 3-34</td>
<td></td>
</tr>
<tr>
<td>Thumb 6-32</td>
<td></td>
</tr>
<tr>
<td>Bit Clear (BIC) Instruction</td>
<td></td>
</tr>
<tr>
<td>ARM 3-34</td>
<td></td>
</tr>
<tr>
<td>Thumb 6-32</td>
<td></td>
</tr>
<tr>
<td>Bit Manipulation 4-2</td>
<td></td>
</tr>
</tbody>
</table>
BL
  ARM  3-33
  Thumb  6-33
block copy  1-6, 4-8
branch (B) instruction  1-4
  ARM  3-33
  Thumb
    conditional  6-30
    unconditional  6-31
branch instructions
  ARM
    examples  3-6, 4-4
    list of  3-6
    long branch  4-7
    multi-way  4-5, 4-7
  Thumb  6-5
    examples  6-6
    list of  6-6
branch with exchange (BX) instruction
  ARM  3-35
  Thumb  6-34
branch with link (BL) instruction  1-2, 1-4, 3-6
  ARM  3-33
  Thumb  6-33
BX
  ARM  3-35
  Thumb  6-34
byte  1-5
  order (endianness)  4-3

C
  cache  7-22, 7-29
coprocessor 15 functions  7-8
CDP  3-36
CMN
  ARM  3-37
  Thumb  6-35
CMNP (26-bit compare negative) instruction  5-4
CMNP (26-bit compare) negative instruction  5-5, 5-8
CMP
  ARM  3-38
  Thumb  6-36 to 6-38
CMPP (26-bit compare) instruction  5-4 to 5-5, 5-8
code samples  4-1
  arithmetic instructions  4-2
  branch instructions  4-4
  load and store instructions  4-6
  load and store multiple instructions  4-8
  semaphore instructions  4-9
compare (CMP) instruction
  ARM  3-38
  Thumb
    high registers  6-38
    immediate  6-36
    registers  6-37
compare instructions, conditional  4-4
compare negative (CMN) instruction
  ARM  3-37
  Thumb  6-35
comparing strings  4-6
condition code flags  1-3 to 1-5, 3-8, 3-56
  26-bit architectures  5-2
carry (C)  2-3, 3-4, 3-8
negative (N)  2-3, 3-4, 3-8
overflow (V)  2-3, 3-4, 3-8
zero (Z)  2-3, 3-4, 3-8
conditional branch  3-6
conditional execution  1-2, 1-4, 3-4
  examples  4-4
collection
  26-bit  5-8
  32-bit  5-8
constant value. See immediate value
constants
  dividing by  3-91
  multiplying by  4-2
copy of switch  4-13
code instruction extension space  3-23
coprocessor
  domain access control register  3-38
  extension space  3-27
     software emulation  2-7
coprocessor 15
  architecture  7-2
cache functions  7-8
coprocessor 15
  architecture  7-2
cache functions  7-8
  control register
    architecture version 3  7-11
coprocessor 15
  control register
    architecture version 3  7-11
    architecture version 4  7-4
  domain access control register
    architecture version 3  7-12
coprocessor 15
  architecture  7-2
cache functions  7-8
coprocessor 15
  architecture  7-2
cache functions  7-8
  control register
    architecture version 3  7-11
coprocessor 15
  architecture  7-2
cache functions  7-8
  control register
    architecture version 3  7-11
  architecture version 4  7-4
  domain access control register
    architecture version 3  7-12
    architecture version 4  7-6
  fault address register  7-7
coprocessor 15
  architecture  7-2
cache functions  7-8
  control register
    architecture version 3  7-11
  architecture version 4  7-4
  domain access control register
    architecture version 3  7-12
coprocessor 15
  architecture  7-2
cache functions  7-8
  control register
    architecture version 3  7-11
    architecture version 4  7-6
  fault address register  7-7
  fault status register  7-7
## Index

flush cache  
architecture version 3 7-13
ID register  
architecture version 3 7-10
architecture version 4 7-4
registers 7-2
translation lookaside buffers 7-9
translation table base register 7-6
architecture version 3 7-12
coprocessor data processing (CDP) instruction 3-36
CPSR 1-3, 1-5, 2-3, 3-12
26-bit architectures 5-2
condition code flags 3-4
See also condition code flags
control bits 2-5
Current Program Status Register. See CPSR

### D

data abort 2-6  
See also exception, data abort

data types 2-2
DATA32 signal 5-8
data-processing  
addressing modes 3-87 to 3-97
instructions  
ARM 1-4
examples 3-7, 4-2
list of 3-9
Thumb 6-7
examples 6-8
list of 6-10
default shifter operand 3-85
divide by constant 3-91
DMA transfer examples 4-10, 4-11
domain access control register  
architecture version 3 7-12
coprocessor 15 7-6
domain fault 7-26
domains, MMU 7-15, 7-23

### E

endianness, swapping byte order 4-3
EOR  
ARM 3-39
Thumb 6-39
examples of code 4-1, 4-2, 4-4
exceptions 2-6
26-bit  
address 5-6
architectures 5-2
data abort 2-8
fast interrupt request 2-9
handler 1-3
interrupt request 2-9
prefetch abort 2-8
priorities 2-10
return 3-53
software interrupt 2-7, 3-75
undefined instruction 2-7
vector 5-9, 7-26
26-bit architectures 5-8
See also vectors
See also interrupt and fast interrupt
exchanging ARM and THUMB state 6-3
exclusive OR (EOR) instruction  
ARM 3-39
Thumb 6-39
extending the ARM instruction set 3-22
external aborts 7-28

### F

fast interrupt 1-3, 2-6
See also processor mode, fast interrupt mode
fault address register 7-25
coprocessor 15 7-7
fault status register 7-25
coprocessor 15 7-7
faults  
alignment 7-26
domain 7-26
permission 7-28
translation 7-26
FIQ. See processor mode, fast interrupt mode
first-level descriptors 7-16
first-level fetch 7-15
flush functions  
architecture version 3 7-13
architecture version 4 7-9

### H

halfword 1-5
high registers 6-8
Index

ID register
architecture version 3 7-10
architecture version 4 7-4
IDC, instruction and/or data cache 7-2
immediate operand 3-84
value 1-5, 3-7, 3-87
indivisible bus operation 1-6
instruction sets
ARM 3-1
Thumb 6-1
instructions
26-bit
CMNP 5-4 to 5-5, 5-8
CMPP 5-4 to 5-5, 5-8
TEQP 5-4 to 5-5, 5-8
TSTP 5-4 to 5-5, 5-8
ARM
ADC 3-30
ADD 3-31
AND 3-32
ASR 3-93 to 3-94, 3-102, 3-105, 3-108, 3-115
B 3-33
BIC 3-34
BL 3-33
BX 3-35
CDP 3-36
CMN 3-37
CMP 3-38
EOR 3-39
LDC 3-40
LDM 3-41 to 3-43
LDR 3-44
LDRB 3-45
LDRBT 3-46
LDRH 3-47
LDRSB 3-48
LDRSH 3-49
LDRT 3-50
LSL 3-89 to 3-90, 3-102, 3-105, 3-108
LSR 3-91 to 3-92, 3-102, 3-105, 3-108
MCR 3-51
MLA 3-52
MOV 3-53
MRC 3-54
MRS 3-55
MSR 3-56
MUL 3-58
MVN 3-59
ORR 3-60
ROR 3-95 to 3-96, 3-102, 3-105, 3-108
RRX 3-97, 3-102, 3-105, 3-108
RSB 3-61
RSC 3-62
SBC 3-63
SMLAL 3-64
SMULL 3-65
STC 3-66
STM 3-67 to 3-68
STR 3-69
STRB 3-70
STRBT 3-71
STRH 3-72
STRR 3-73
SUB 3-74
SWI 3-75
SWP 3-76
SWPB 3-77
TEQ 3-78
TST 3-79
UMLAL 3-80
UMULL 3-81
Thumb
ADC 6-19
ADD 6-20 to 6-26
AND 6-27
ASR 6-28, 6-29
B 6-30, 6-31
BIC 6-32
BL 6-33
BX 6-34
CMN 6-35
CMP 6-36 to 6-38
EOR 6-39
LDM 6-40
LDR 6-42 to 6-45
LDRB 6-46 to 6-47
LDRH 6-48 to 6-49
LDRRB 6-50
LDRSH 6-51
LSL 6-52 to 6-53
LSR 6-54 to 6-55
MOV 6-56 to 6-57
MUL 6-58
MVN 6-59
NEG 6-60
ORR 6-61
POP 6-62
PUSH 6-64
ROR 6-66
SBC 6-67
STM 6-68
STR 6-70 to 6-72

ARM Architecture Reference Manual
ARM DDI 0100B
Index

STRB  6-73 to 6-74
STRH  6-75 to 6-76
SUB   6-77 to 6-80
SWI   6-81
TST   6-82
interrupt  1-3, 2-6, 7-30
disable  1-3, 2-5
26-bit architectures  5-2
enable  1-5, 3-56
handlers  4-10, 4-12
priority  4-12
IRQ. See processor mode, interrupt

J
jump  1-6

L
LDC  3-40
LDM
   ARM  3-41 to 3-43
   Thumb  6-40
LDR
   ARM  3-44
   Thumb  6-42
LDRB
   ARM  3-45
   Thumb  6-46 to 6-47
LDRBT
   ARM  3-46
LDRH
   ARM  3-47
   Thumb  6-48 to 6-49
LDRSB
   ARM  3-48
   Thumb  6-50
LDRSH
   ARM  3-49
   Thumb  6-51
LDRT  3-50
link register (LR)  1-2, 1-4, 2-3, 3-6
linked lists  4-6
load and store coprocessor
   addressing modes  3-123 to 3-126
   instructions  3-20
   list of  3-21
load and store halfword and load signed byte
   addressing modes  3-109 to 3-115
   instruction list  3-16
load and store instructions  1-5, 3-13, 6-12
   addressing mode
      offset  1-6
      post-indexed  1-6
      pre-indexed  1-6
   examples  4-6
   extension space  3-24
   Thumb  6-12
      examples  6-13
      list of  6-13
See also LDR and STR
load and store multiple
   addressing modes  3-116 to 3-122
      post-decrement  1-6
      post-increment  1-6
      pre-decrement  1-6
      pre-increment  1-6
      instructions  1-2, 3-17
      base address  1-6
      examples  4-8
      list of  1-6, 3-18
See also LDM and STM
   Thumb
      instructions  6-14
      examples  6-14
      list of  6-15
load and store word and unsigned byte
   addressing modes  3-98 to 3-108
   instructions
      list of  3-15
load coprocessor (LDC) instruction  3-40
load instructions
   used as a branch  4-7
load multiple (LDM) instruction
   ARM  3-41 to 3-43
   Thumb  6-40
load register (LDR) instruction
   ARM  3-44
   Thumb
      immediate  6-42
      PC-relative  6-44
      register offset  6-43
      SP-relative  6-45
load register byte (LDRB) instruction
   ARM  3-45
   Thumb
      immediate offset  6-46
      register offset  6-47
load register byte with translation (LDRBT) instruction
   3-46
Index

load register halfword (LDRH) instruction 3-47
  Thumb
    immediate offset 6-48
    register offset 6-49

load register signed byte (LDRSB) instruction 3-48
  Thumb 6-50

load register signed halfword (LDRSH) instruction 3-49
  Thumb 6-51

load register with translation (LDRT) instruction 3-50

load-store architecture 1-2

logical OR (ORR) instruction 3-60
  Thumb 6-61

logical shift left (LSL) instruction 3-90
  as addressing mode 3-102, 3-105, 3-108
    immediate 3-89
  Thumb
    immediate 6-52
    register 6-53

logical shift right (LSR) instruction 3-91
  as addressing mode 3-102, 3-105, 3-108
  Thumb
    immediate 6-54
    register 6-55

long branch 3-6

loop counter 4-5

LR. See link register

LSL
  ARM 3-89 to 3-90, 3-102, 3-105, 3-108
  Thumb 6-52 to 6-53

LSR
  ARM 3-91 to 3-92, 3-102, 3-105, 3-108
  Thumb 6-54 to 6-55

M

MCR 3-51

memory
  aborts 1-3
  access restrictions 7-24
  accesses 7-14
  protection 1-3
  virtual 1-3

memory management unit. See MMU

MLA 3-52

MMU 7-2, 7-5, 7-12
  architecture 7-14
  domains 7-15, 7-23
  faults 7-24

MOV
  ARM 3-53
  Thumb 6-56 to 6-57

move (MOV) instruction 3-53
  Thumb
    immediate 6-56
    register 6-57

move general-purpose register or immediate to status register (MSR) instruction 3-56

move not (MVN) instruction 3-59
  Thumb 6-59

move status register to general-purpose register (MRS) instruction 3-55

move to ARM register from coprocessor (MRC) instruction 3-54

move to coprocessor from ARM register (MCR) instruction 3-51

MRC 3-54

MRS 3-55

MSR 3-56

MUL
  ARM 3-58
  Thumb 6-58

multiply
  32-bit 3-10
  64-bit 3-10
  by constant 3-89
    example 4-2
    instructions 1-5
    list of 3-10, 3-11

multiply (MUL) instruction
  ARM 3-58
  Thumb 6-58

multiply accumulate (MLA) instruction 3-52

multi-way branch 4-5, 4-7

MVN
  ARM 3-59
  Thumb 6-59

N

NEG 6-60

negate (NEG) instruction 6-60
**Index**

**O**
- operands
  - data-processing  3-84
  - immediate  3-84
  - shifted register  3-85
  - shifter  3-86
- operating system  1-3, 1-4, 2-7, 3-75
- ORR
  - ARM  3-60
  - Thumb  6-61

**P**
- page table descriptor  7-18
- PC. See Program Counter
- permission fault  7-28
- POP  6-62
- pop multiple registers (POP) instruction  6-62
- prefetch abort  2-6
- procedure
  - call and return  4-4
  - entry and exit
    - example  4-8
- processor mode  1-3, 1-5
  - 26-bit and 32-bit  5-7
  - 26-bit architectures  5-2
  - abort mode  2-2
  - changing  2-2, 3-56
  - fast interrupt mode  2-2, 2-3
  - interrupt mode  2-2
  - mode bits  2-5
  - privileged  1-3, 2-2, 2-6
  - reset  2-7
  - supervisor mode  2-2
  - system mode  1-3, 2-2
  - undefined mode  2-2
  - unprivileged modes  1-4
  - user mode  2-2
- PROG32 signal  5-8
- program counter (PC)  1-2, 1-4, 2-3, 3-6
  - 26-bit architecture  5-2
- program status register  3-12
  - 26-bit architectures  5-4
  - access instructions  3-12
  - instructions
    - list of  3-12
- PUSH  6-64
- push multiple registers (PUSH) instruction  6-64

**R**
- register 15
  - 26-bit architectures  5-2
  - program counter bits  5-3
- registers
  - banked  1-3, 2-6
  - banks of  2-3
  - high  6-8
  - operand value  3-85
  - overview  1-2
  - shifted operand value  3-85
- reset  2-6
- return address  1-6, 3-6
- reverse subtract (RSB) instruction  3-61
- reverse subtract with carry (RSC) instruction  3-62
- RISC (Reduced Instruction Set Computer)  1-2
- ROR
  - ARM  3-95 to 3-96, 3-102, 3-105, 3-108
  - Thumb  6-66
- rotate left with extend  3-30
- rotate right (ROR) instruction
  - ARM
    - as addressing mode  3-102, 3-105, 3-108
    - immediate  3-96
    - register  3-95
  - Thumb
    - register  6-66
- rotate right with extend (RRX) instruction  3-97
  - ARM
    - as addressing mode  3-105
    - as addressing mode  3-102, 3-105, 3-108
  - RRX  3-97, 3-102, 3-105, 3-108
  - ARM  3-105
  - RSB  3-61
  - RSC  3-62

**S**
- Saved Program Status Register. See SPSR
- SBC
  - ARM  3-63
  - Thumb  6-67
  - second-level descriptor  7-19
  - section descriptor  7-17
  - section references  7-17
  - semaphores  1-6, 7-31
  - examples  4-9
  - instruction list  3-19
shift 1-5, 3-7, 3-53  
  instructions 1-5  
  shifter register 1-5  
shifter operand 3-7, 3-84  
  default 3-85  
  register 3-85  
signals  
  DATA32 5-8  
  PROG32 5-8  
signed multiply accumulate long (SMLAL) instruction 3-64  
signed multiply long (SMULL) instruction 3-65  
sign-extend 1-5, 2-2  
SMLAL 3-64  
SMULL 3-65  
software interrupt (SWI) 1-3, 1-4, 2-6  
  examples 4-10  
  instruction  
    ARM 3-75  
    Thumb 6-81  
SPSR 1-3, 1-5, 2-3, 3-12  
  26-bit architectures 5-2  
stack pointer (SP) 1-2, 1-3, 1-6, 2-3  
  incrementing 6-26  
status register access instructions 3-12  
status register transfer instructions 1-5  
  See also MRS and MSR instructions  
status registers. See program status register  
STC 3-66  
STM  
    ARM 3-67 to 3-68  
    Thumb 6-68  
store coprocessor (STC) instruction 3-66  
store multiple (STM) instruction  
    ARM 3-67 to 3-68  
    Thumb 6-68  
store register (STR) instruction  
    ARM 3-69  
    Thumb  
      immediate 6-70  
      register 6-71  
      SP-relative 6-72  
store register byte (STRB) instruction  
    ARM 3-70  
    Thumb  
      immediate 6-73  
      register 6-74  
store register halfword (STRH) instruction  
    ARM 3-72  
    Thumb  
      immediate 6-75  
      register 6-76  
store register with translation (STRT) instruction 3-73  
STR  
    ARM 3-69  
    Thumb 6-70 to 6-72  
STRB  
    ARM 3-70  
    Thumb 6-73 to 6-74  
STRBT 3-71  
STRH  
    ARM 3-72  
    Thumb 6-75 to 6-76  
string compare 4-6  
STRT 3-73  
SUB  
    ARM 3-74  
    Thumb 6-77 to 6-80  
subroutine  
    call 1-2, 2-3, 3-6, 3-33  
    call and return 1-6  
    return 3-53  
    return address 1-6  
subtract (SUB) instruction  
    ARM 3-74  
    Thumb  
      decrement stack pointer 6-80  
      immediate 6-77  
      large immediate 6-78  
      large register 6-79  
subtract with carry (SBC) instruction  
    ARM 3-63  
    Thumb 6-67  
swap byte (SWPB) instruction 3-77  
swap word (SWP) instruction 3-76  
swapping  
    byte order (endianness) 4-3  
    register and memory values 1-6  
SWI  
    ARM 3-75  
    Thumb 6-81  
SWP 3-76  
SWPB 3-77
system control coprocessor 7-2
  ARM version 3 7-10
  ARM version 4 7-2
  See also coprocessor 15

system mode 1-3
  See processor mode, system mode

T

TEQ 3-78
TEQP (26-bit test equivalence) instruction 5-4 to 5-5, 5-8

test (TST) instruction
  ARM 3-79
  Thumb 6-82
test equivalence (TEQ) instruction 3-78

THUMB 6-12
  architecture 6-1
  ARM code execution 2-5
  branch instructions 6-5
    list of 6-6
code 1-4, 3-35
code execution 2-5
data-processing instructions 6-7
  list of 6-8, 6-10O
instruction set 1-4, 6-1
  overview 6-4
load and store instructions
  examples 6-13
    list of 6-13
load and store multiple instructions 6-14
  examples 6-14
    list of 6-15
T flag 2-5, 3-35

translating
  large page references 7-20
  section references 7-17
  small page references 7-21
translation fault 7-26
translation lookaside buffers 7-14
  architecture version 3 7-13
coprocessor 15 7-9
flush functions 7-9
translation table 7-9
  base 7-15
  base register
  architecture version 3 7-12
coprocessor 15 7-6

TST
  ARM 3-79
  Thumb 6-82
TSTP (26-bit test) instruction 5-4 to 5-5, 5-8

U

UMLAL 3-80
UMULL 3-81
undefined instruction 1-3, 2-6
  extension space 3-27
unsigned multiply accumulate long (UMLAL) instruction 3-80
unsigned multiply long (UMULL) instruction 3-81
user mode 1-2, 1-3
  context switch 4-13
  See also processor mode, user mode

V

variables 4-5
vector exception 5-8, 5-9, 7-26
vectors 1-3, 2-6
virtual memory 1-3

W

word 1-5
write buffer 7-2, 7-22, 7-29

Z

zero-extend 1-5, 2-2