

3 The Cortex-M3 instruction set

3.1 Instruction set summary

The processor implements a version of the thumb instruction set. [Table 20](#) lists the supported instructions.

In [Table 20](#):

- Angle brackets, <>, enclose alternative forms of the operand
- Braces, {}, enclose optional operands
- The operands column is not exhaustive
- Op2 is a flexible second operand that can be either a register or a constant
- Most instructions can use an optional condition code suffix

For more information on the instructions and operands, see the instruction descriptions.

Table 20. Cortex-M3 instructions

Mnemonic	Operands	Brief description	Flags	Section
ADC, ADCS	{Rd,} Rn, Op2	Add with carry	N,Z,C,V	3.5.1 on page 73
ADD, ADDS	{Rd,} Rn, Op2	Add	N,Z,C,V	3.5.1 on page 73
ADD, ADDW	{Rd,} Rn, #imm12	Add	N,Z,C,V	3.5.1 on page 73
ADR	Rd, label	Load PC-relative address	—	3.4.1 on page 60
AND, ANDS	{Rd,} Rn, Op2	Logical AND	N,Z,C	3.5.2 on page 75
ASR, ASRS	Rd, Rm, <Rs #n>	Arithmetic shift right	N,Z,C	3.5.3 on page 76
B	label	Branch	—	3.8.5 on page 92
BFC	Rd, #lsb, #width	Bit field clear	—	3.8.1 on page 89
BFI	Rd, Rn, #lsb, #width	Bit field insert	—	3.8.1 on page 89
BIC, BICS	{Rd,} Rn, Op2	Bit clear	N,Z,C	3.5.2 on page 75
BKPT	#imm	Breakpoint	—	3.9.1 on page 98
BL	label	Branch with link	—	3.8.5 on page 92
BLX	Rm	Branch indirect with link	—	3.8.5 on page 92

Table 20. Cortex-M3 instructions (continued)

Mnemonic	Operands	Brief description	Flags	Section
BX	Rm	Branch indirect	—	3.8.5 on page 92
CBNZ	Rn, label	Compare and branch if non zero	—	3.8.6 on page 93
CBZ	Rn, label	Compare and branch if zero	—	3.8.6 on page 93
CLREX	—	Clear exclusive	—	3.4.9 on page 71
CLZ	Rd, Rm	Count leading zeros	—	3.5.4 on page 77
CMN, CMNS	Rn, Op2	Compare negative	N,Z,C,V	3.5.5 on page 78
CMP, CMPS	Rn, Op2	Compare	N,Z,C,V	3.5.5 on page 78
CPSID	iflags	Change processor state, disable interrupts	—	3.9.2 on page 98
CPSIE	iflags	Change processor state, enable interrupts	—	3.9.2 on page 98
DMB	—	Data memory barrier	—	3.9.4 on page 100
DSB	—	Data synchronization barrier	—	3.9.4 on page 100
EOR, EORS	{Rd,} Rn, Op2	Exclusive OR	N,Z,C	3.5.2 on page 75
ISB	—	Instruction synchronization barrier	—	3.9.5 on page 100
IT	—	If-then condition block	—	3.8.7 on page 94
LDM	Rn{!}, reglist	Load multiple registers, increment after	—	3.4.6 on page 67
LDMDB, LDMEA	Rn{!}, reglist	Load multiple registers, decrement before	—	3.4.6 on page 67
LDMFD, LDMIA	Rn{!}, reglist	Load multiple registers, increment after	—	3.4.6 on page 67
LDR	Rt, [Rn, #offset]	Load register with word	—	3.4 on page 60
LDRB, LDRBT	Rt, [Rn, #offset]	Load register with byte	—	3.4 on page 60
LDRD	Rt, Rt2, [Rn, #offset]	Load register with two bytes	—	3.4.2 on page 61
LDREX	Rt, [Rn, #offset]	Load register exclusive	—	3.4.8 on page 70

Table 20. Cortex-M3 instructions (continued)

Mnemonic	Operands	Brief description	Flags	Section
LDREXB	Rt, [Rn]	Load register exclusive with byte	—	3.4.8 on page 70
LDREXH	Rt, [Rn]	Load register exclusive with halfword	—	3.4.8 on page 70
LDRH, LDRHT	Rt, [Rn, #offset]	Load register with halfword	—	3.4 on page 60
LDRSB, LDRSBT	Rt, [Rn, #offset]	Load register with signed byte	—	3.4 on page 60
LDRSH, LDRSHT	Rt, [Rn, #offset]	Load register with signed halfword	—	3.4 on page 60
LDRT	Rt, [Rn, #offset]	Load register with word	—	3.4 on page 60
LSL, LSLS	Rd, Rm, <Rs #n>	Logical shift left	N,Z,C	3.5.3 on page 76
LSR, LSRS	Rd, Rm, <Rs #n>	Logical shift right	N,Z,C	3.5.3 on page 76
MLA	Rd, Rn, Rm, Ra	Multiply with accumulate, 32-bit result	—	3.6.1 on page 83
MLS	Rd, Rn, Rm, Ra	Multiply and subtract, 32-bit result	—	3.6.1 on page 83
MOV, MOVS	Rd, Op2	Move	N,Z,C	3.5.6 on page 79
MOVT	Rd, #imm16	Move top	—	3.5.7 on page 80
MOVW, MOV	Rd, #imm16	Move 16-bit constant	N,Z,C	3.5.6 on page 79
MRS	Rd, spec_reg	Move from special register to general register	—	3.9.6 on page 100
MSR	spec_reg, Rm	Move from general register to special register	N,Z,C,V	3.9.7 on page 101
MUL, MULS	{Rd,} Rn, Rm	Multiply, 32-bit result	N,Z	3.6.1 on page 83
MVN, MVNS	Rd, Op2	Move NOT	N,Z,C	3.5.6 on page 79
NOP	—	No operation	—	3.9.8 on page 102
ORN, ORNS	{Rd,} Rn, Op2	Logical OR NOT	N,Z,C	3.5.2 on page 75
ORR, ORRS	{Rd,} Rn, Op2	Logical OR	N,Z,C	3.5.2 on page 75
POP	reglist	Pop registers from stack	—	3.4.7 on page 68

Table 20. Cortex-M3 instructions (continued)

Mnemonic	Operands	Brief description	Flags	Section
PUSH	reglist	Push registers onto stack	—	3.4.7 on page 68
RBIT	Rd, Rn	Reverse bits	—	3.5.8 on page 81
REV	Rd, Rn	Reverse byte order in a word	—	3.5.8 on page 81
REV16	Rd, Rn	Reverse byte order in each halfword	—	3.5.8 on page 81
REVSH	Rd, Rn	Reverse byte order in bottom halfword and sign extend	—	3.5.8 on page 81
ROR, RORS	Rd, Rm, <Rs #n>	Rotate right	N,Z,C	3.5.3 on page 76
RRX, RRXS	Rd, Rm	Rotate right with extend	N,Z,C	3.5.3 on page 76
RSB, RSBS	{Rd,} Rn, Op2	Reverse subtract	N,Z,C,V	3.5.1 on page 73
SBC, SBCS	{Rd,} Rn, Op2	Subtract with carry	N,Z,C,V	3.5.1 on page 73
SBFX	Rd, Rn, #lsb, #width	Signed bit field extract	—	3.8.2 on page 89
SDIV	{Rd,} Rn, Rm	Signed divide	—	3.6.3 on page 86
SEV	—	Send event	—	3.9.9 on page 102
SMLAL	RdLo, RdHi, Rn, Rm	Signed multiply with accumulate (32 x 32 + 64), 64-bit result	—	3.6.2 on page 85
SMULL	RdLo, RdHi, Rn, Rm	Signed multiply (32 x 32), 64-bit result	—	3.6.2 on page 85
SSAT	Rd, #n, Rm {,shift #s}	Signed saturate	Q	3.7.1 on page 87
STM	Rn{!}, reglist	Store multiple registers, increment after	—	3.4.6 on page 67
STMDB, STMEA	Rn{!}, reglist	Store multiple registers, decrement before	—	3.4.6 on page 67
STMFD, STMIA	Rn{!}, reglist	Store multiple registers, increment after	—	3.4.6 on page 67
STR	Rt, [Rn, #offset]	Store register word	—	3.4 on page 60
STRB, STRBT	Rt, [Rn, #offset]	Store register byte	—	3.4 on page 60
STRD	Rt, Rt2, [Rn, #offset]	Store register two words	—	3.4.2 on page 61

Table 20. Cortex-M3 instructions (continued)

Mnemonic	Operands	Brief description	Flags	Section
STREX	Rd, Rt, [Rn, #offset]	Store register exclusive	—	3.4.8 on page 70
STREXB	Rd, Rt, [Rn]	Store register exclusive byte	—	3.4.8 on page 70
STREXH	Rd, Rt, [Rn]	Store register exclusive halfword	—	3.4.8 on page 70
STRH, STRHT	Rt, [Rn, #offset]	Store register halfword	—	3.4 on page 60
STRT	Rt, [Rn, #offset]	Store register word	—	3.4 on page 60
SUB, SUBS	{Rd,} Rn, Op2	Subtract	N,Z,C,V	3.5.1 on page 73
SUB, SUBW	{Rd,} Rn, #imm12	Subtract	N,Z,C,V	3.5.1 on page 73
SVC	#imm	Supervisor call	—	3.9.10 on page 103
SXTB	{Rd,} Rm {,ROR #n}	Sign extend a byte	—	3.8.3 on page 90
SXTH	{Rd,} Rm {,ROR #n}	Sign extend a halfword	—	3.8.3 on page 90
TBB	[Rn, Rm]	Table branch byte	—	3.8.8 on page 96
TBH	[Rn, Rm, LSL #1]	Table branch halfword	—	3.8.8 on page 96
TEQ	Rn, Op2	Test equivalence	N,Z,C	3.5.9 on page 82
TST	Rn, Op2	Test	N,Z,C	3.5.9 on page 82
UBFX	Rd, Rn, #lsb, #width	Unsigned bit field extract	—	3.8.2 on page 89
UDIV	{Rd,} Rn, Rm	Unsigned divide	—	3.6.3 on page 86
UMLAL	RdLo, RdHi, Rn, Rm	Unsigned multiply with accumulate (32 x 32 + 64), 64-bit result	—	3.6.2 on page 85
UMULL	RdLo, RdHi, Rn, Rm	Unsigned multiply (32 x 32), 64-bit result	—	3.6.2 on page 85
USAT	Rd, #n, Rm {,shift #s}	Unsigned saturate	Q	3.7.1 on page 87
UXTB	{Rd,} Rm {,ROR #n}	Zero extend a byte	—	3.8.3 on page 90
UXTH	{Rd,} Rm {,ROR #n}	Zero extend a halfword	—	3.8.3 on page 90

Table 20. Cortex-M3 instructions (continued)

Mnemonic	Operands	Brief description	Flags	Section
WFE	—	Wait for event	—	3.9.11 on page 103
WFI	—	Wait for interrupt	—	3.9.12 on page 104

3.2 Intrinsic functions

ANSI cannot directly access some Cortex-M3 instructions. This section describes intrinsic functions that can generate these instructions, provided by the CMSIS and that might be provided by a C compiler. If a C compiler does not support an appropriate intrinsic function, you might have to use an inline assembler to access some instructions.

The CMSIS provides the intrinsic functions listed in [Table 21](#) to generate instructions that ANSI cannot directly access.

Table 21. CMSIS intrinsic functions to generate some Cortex-M3 instructions

Instruction	CMSIS intrinsic function
CPSIE I	void __enable_irq(void)
CPSID I	void __disable_irq(void)
CPSIE F	void __enable_fault_irq(void)
CPSID F	void __disable_fault_irq(void)
ISB	void __ISB(void)
DSB	void __DSB(void)
DMB	void __DMB(void)
REV	uint32_t __REV(uint32_t int value)
REV16	uint32_t __REV16(uint32_t int value)
REVSH	uint32_t __REVSH(uint32_t int value)
RBIT	uint32_t __RBIT(uint32_t int value)
SEV	void __SEV(void)
WFE	void __WFE(void)
WFI	void __WFI(void)

The CMSIS also provides a number of functions for accessing the special registers using MRS and MSR instructions (see [Table 22](#)).

Table 22. CMSIS intrinsic functions to access the special registers

Special register	Access	CMSIS function
PRIMASK	Read	uint32_t __get_PRIMASK (void)
	Write	void __set_PRIMASK (uint32_t value)
FAULTMASK	Read	uint32_t __get_FAULTMASK (void)
	Write	void __set_FAULTMASK (uint32_t value)
BASEPRI	Read	uint32_t __get_BASEPRI (void)
	Write	void __set_BASEPRI (uint32_t value)
CONTROL	Read	uint32_t __get_CONTROL (void)
	Write	void __set_CONTROL (uint32_t value)
MSP	Read	uint32_t __get_MSP (void)
	Write	void __set_MSP (uint32_t TopOfMainStack)
PSP	Read	uint32_t __get_PSP (void)
	Write	void __set_PSP (uint32_t TopOfProcStack)

3.3 About the instruction descriptions

The following sections give more information about using the instructions:

- [Operands on page 51](#)
- [Restrictions when using PC or SP on page 52](#)
- [Flexible second operand on page 52](#)
- [Shift operations on page 53](#)
- [Address alignment on page 56](#)
- [PC-relative expressions on page 56](#)
- [Conditional execution on page 57](#)
- [Instruction width selection on page 59](#).

3.3.1 Operands

An instruction operand can be an ARM register, a constant, or another instruction-specific parameter. Instructions act on the operands and often store the result in a destination register. When there is a destination register in the instruction, it is usually specified before the operands.

Operands in some instructions are flexible in that they can either be a register or a constant (see [Flexible second operand](#)).

3.3.2 Restrictions when using PC or SP

Many instructions have restrictions on whether you can use the *program counter* (PC) or *stack pointer* (SP) for the operands or destination register. See instruction descriptions for more information.

Bit[0] of any address written to the PC with a BX, BLX, LDM, LDR, or POP instruction must be 1 for correct execution, because this bit indicates the required instruction set, and the Cortex-M3 processor only supports thumb instructions.

3.3.3 Flexible second operand

Many general data processing instructions have a flexible second operand. This is shown as *operand2* in the descriptions of the syntax of each instruction.

Operand2 can be a:

- [Constant](#)
- [Register with optional shift](#)

Constant

You specify an *operand2* constant in the form *#constant*, where *constant* can be:

- Any constant that can be produced by shifting an 8-bit value left by any number of bits within a 32-bit word.
- Any constant of the form 0x00XY00XY
- Any constant of the form 0xXY00XY00
- Any constant of the form 0xYXYXYXY

In the constants shown above, X and Y are hexadecimal digits.

In addition, in a small number of instructions, *constant* can take a wider range of values. These are described in the individual instruction descriptions.

When an *operand2* constant is used with the instructions MOVNS, MVNS, ANDS, ORRS, ORNS, EORS, BICS, TEQ or TST, the carry flag is updated to bit[31] of the constant, if the constant is greater than 255 and can be produced by shifting an 8-bit value. These instructions do not affect the carry flag if *operand2* is any other constant.

Instruction substitution

Your assembler might be able to produce an equivalent instruction in cases where you specify a constant that is not permitted. For example, an assembler might assemble the instruction `CMP Rd, #0xFFFFFFFF` as the equivalent instruction `CMN Rd, #0x2`.

Register with optional shift

An operand2 register is specified in the form $Rm \{, shift\}$, where:

- Rm is the register holding the data for the second operand
- $Shift$ is an optional shift to be applied to Rm . It can be one of:
 - ASR $\#n$: Arithmetic shift right n bits, $1 \leq n \leq 32$
 - LSL $\#n$: Logical shift left n bits, $1 \leq n \leq 31$
 - LSR $\#n$: Logical shift right n bits, $1 \leq n \leq 32$
 - ROR $\#n$: Rotate right n bits, $1 \leq n \leq 31$
 - RRX: Rotate right one bit, with extend
 - : If omitted, no shift occurs, equivalent to LSL #0

If you omit the shift, or specify LSL #0, the instruction uses the value in Rm .

If you specify a shift, the shift is applied to the value in Rm , and the resulting 32-bit value is used by the instruction. However, the contents in the register Rm remains unchanged. Specifying a register with shift also updates the carry flag when used with certain instructions. For information on the shift operations and how they affect the carry flag, see [Shift operations](#).

3.3.4 Shift operations

Register shift operations move the bits in a register left or right by a specified number of bits, the *shift length*. Register shift can be performed:

- Directly by the instructions ASR, LSR, LSL, ROR, and RRX. The result is written to a destination register.
- During the calculation of operand2 by the instructions that specify the second operand as a register with shift (see [Flexible second operand on page 52](#)). The result is used by the instruction.

The permitted shift lengths depend on the shift type and the instruction (see the individual instruction description or [Flexible second operand](#)). If the shift length is 0, no shift occurs. Register shift operations update the carry flag except when the specified shift length is 0. The following sub-sections describe the various shift operations and how they affect the carry flag. In these descriptions, Rm is the register containing the value to be shifted, and n is the shift length.

ASR

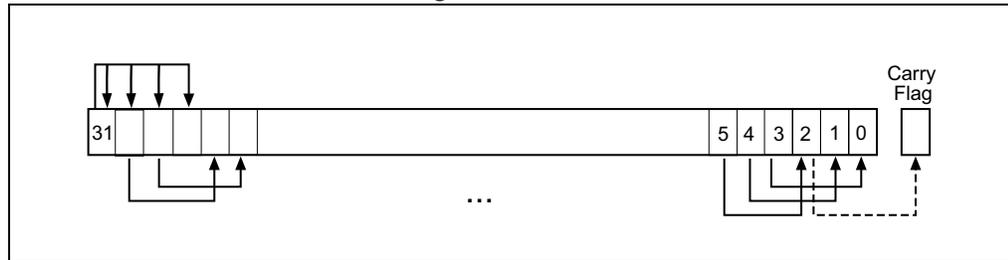
Arithmetic shift right by n bits moves the left-hand $32-n$ bits of the register Rm , to the right by n places, into the right-hand $32-n$ bits of the result. And it copies the original bit[31] of the register into the left-hand n bits of the result (see [Figure 13: ASR#3 on page 54](#)).

You can use the ASR $\#n$ operation to divide the value in the register Rm by 2^n , with the result being rounded towards negative-infinity.

When the instruction is ASRS or when ASR $\#n$ is used in operand2 with the instructions MOVs, MVNS, ANDs, ORRS, ORNS, EORS, BICS, TEQ or TST, the carry flag is updated to the last bit shifted out, bit[$n-1$], of the register Rm .

- Note:*
- 1 If n is 32 or more, all the bits in the result are set to the value of bit[31] of Rm .
 - 2 If n is 32 or more and the carry flag is updated, it is updated to the value of bit[31] of Rm .

Figure 13. ASR#3



LSR

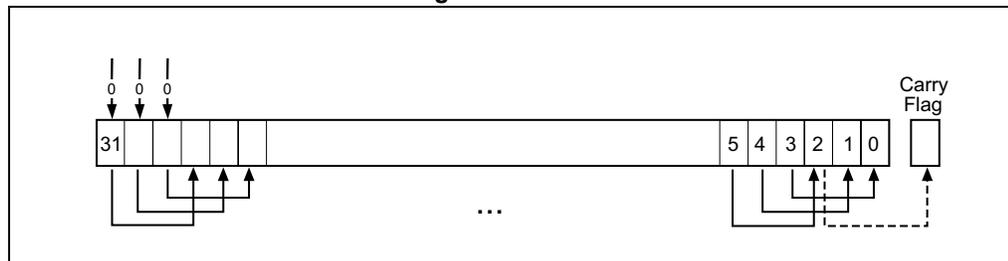
Logical shift right by n bits moves the left-hand $32-n$ bits of the register Rm , to the right by n places, into the right-hand $32-n$ bits of the result. And it sets the left-hand n bits of the result to 0 (see [Figure 14](#)).

You can use the LSR $\#n$ operation to divide the value in the register Rm by 2^n , if the value is regarded as an unsigned integer.

When the instruction is LSRS or when LSR $\#n$ is used in *operand2* with the instructions MOVS, MVNS, ANDS, ORRS, ORNS, EORS, BICS, TEQ or TST, the carry flag is updated to the last bit shifted out, bit[$n-1$], of the register Rm .

- Note:
- 1 If n is 32 or more, then all the bits in the result are cleared to 0.
 - 2 If n is 33 or more and the carry flag is updated, it is updated to 0.

Figure 14. LSR#3



LSL

Logical shift left by n bits moves the right-hand $32-n$ bits of the register Rm , to the left by n places, into the left-hand $32-n$ bits of the result. And it sets the right-hand n bits of the result to 0 (see [Figure 15: LSL#3 on page 55](#)).

You can use the LSL $\#n$ operation to multiply the value in the register Rm by 2^n , if the value is regarded as an unsigned integer or a two's complement signed integer. Overflow can occur without warning.

When the instruction is LLSL or when LSL $\#n$, with non-zero n , is used in *operand2* with the instructions MOVS, MVNS, ANDS, ORRS, ORNS, EORS, BICS, TEQ or TST, the carry flag is updated to the last bit shifted out, bit[$32-n$], of the register Rm . These instructions do not affect the carry flag when used with LSL $\#0$.

- Note: 1 If n is 32 or more, then all the bits in the result are cleared to 0.
 2 If n is 33 or more and the carry flag is updated, it is updated to 0.

Figure 15. LSL#3

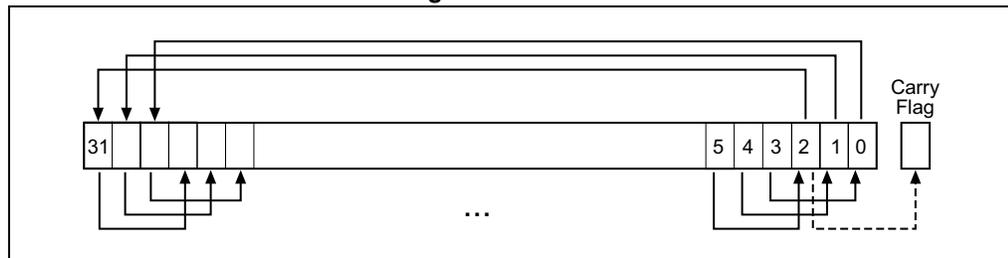
**ROR**

Rotate right by n bits moves the left-hand $32-n$ bits of the register Rm , to the right by n places, into the right-hand $32-n$ bits of the result. It also moves the right-hand n bits of the register into the left-hand n bits of the result (see [Figure 16](#)).

When the instruction is RORS or when ROR # n is used in *operand2* with the instructions MOVS, MVNS, ANDS, ORRS, ORNS, EORS, BICS, TEQ or TST, the carry flag is updated to the last bit rotation, bit[$n-1$], of the register Rm .

- Note: 1 If n is 32, then the value of the result is same as the value in Rm , and if the carry flag is updated, it is updated to bit[31] of Rm .
 2 ROR with shift length, n , more than 32 is the same as ROR with shift length $n-32$.

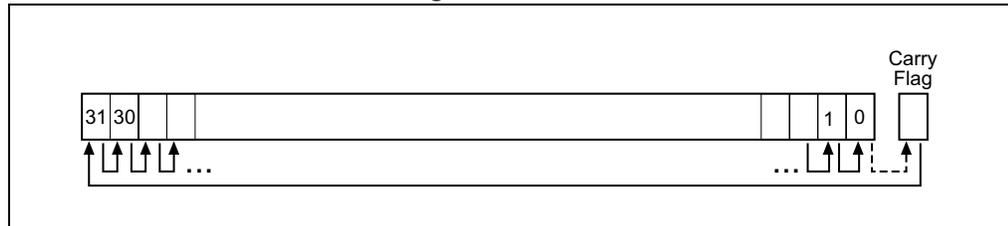
Figure 16. ROR #3

**RRX**

Rotate right with extend moves the bits of the register Rm to the right by one bit. And it copies the carry flag into bit[31] of the result (see [Figure 17](#)).

When the instruction is RRXS or when RRX is used in *operand2* with the instructions MOVS, MVNS, ANDS, ORRS, ORNS, EORS, BICS, TEQ or TST, the carry flag is updated to bit[0] of the register Rm .

Figure 17. RRX #3



3.3.5 Address alignment

An aligned access is an operation where a word-aligned address is used for a word, dual word, or multiple word access, or where a halfword-aligned address is used for a halfword access. Byte accesses are always aligned.

The Cortex-M3 processor supports unaligned access only for the following instructions:

- LDR, LDRT
- LDRH, LDRHT
- LDRSH, LDRSHT
- STR, STRT
- STRH, STRHT

All other load and store instructions generate a usage fault exception if they perform an unaligned access, and therefore their accesses must be address aligned. For more information about usage faults see [Fault handling on page 40](#).

Unaligned accesses are usually slower than aligned accesses. In addition, some memory regions might not support unaligned accesses. Therefore, ARM recommends that programmers ensure that accesses are aligned. To avoid accidental generation of unaligned accesses, use the UNALIGN_TRP bit in the configuration and control register to trap all unaligned accesses, see [Configuration and control register \(SCB_CCR\) on page 137](#).

3.3.6 PC-relative expressions

A PC-relative expression or *label* is a symbol that represents the address of an instruction or literal data. It is represented in the instruction as the PC value plus or minus a numeric offset. The assembler calculates the required offset from the label and the address of the current instruction. If the offset is too big, the assembler produces an error.

- For the B, BL, CBNZ, and CBZ instructions, the value of the PC is the address of the current instruction plus four bytes.
- For all other instructions that use labels, the value of the PC is the address of the current instruction plus four bytes, with bit[1] of the result cleared to 0 to make it word-aligned.
- Your assembler might permit other syntaxes for PC-relative expressions, such as a label plus or minus a number, or an expression of the form [PC, #number].

3.3.7 Conditional execution

Most data processing instructions can optionally update the condition flags in the *application program status register* (APSR) according to the result of the operation (see [Application program status register on page 17](#)). Some instructions update all flags, and some only update a subset. If a flag is not updated, the original value is preserved. See the instruction descriptions for the flags they affect.

You can execute an instruction conditionally, based on the condition flags set in another instruction:

- Immediately after the instruction that updated the flags
- After any number of intervening instructions that have not updated the flags

Conditional execution is available by using conditional branches or by adding condition code suffixes to instructions. See [Table 23: Condition code suffixes on page 58](#) for a list of the suffixes to add to instructions to make them conditional instructions. The condition code suffix enables the processor to test a condition based on the flags. If the condition test of a conditional instruction fails, the instruction:

- Does not execute
- Does not write any value to its destination register
- Does not affect any of the flags
- Does not generate any exception

Conditional instructions, except for conditional branches, must be inside an If-then instruction block. See [IT on page 94](#) for more information and restrictions when using the IT instruction. Depending on the vendor, the assembler might automatically insert an IT instruction if you have conditional instructions outside the IT block.

Use the CBZ and CBNZ instructions to compare the value of a register against zero and branch on the result.

This section describes:

- [The condition flags](#)
- [Condition code suffixes on page 58](#)

The condition flags

The APSR contains the following condition flags:

- N: Set to 1 when the result of the operation is negative, otherwise cleared to 0
- Z: Set to 1 when the result of the operation is zero, otherwise cleared to 0
- C: Set to 1 when the operation results in a carry, otherwise cleared to 0.
- V: Set to 1 when the operation causes an overflow, otherwise cleared to 0.

For more information about the APSR see [Program status register on page 16](#).

A carry occurs:

- If the result of an addition is greater than or equal to 2^{32}
- If the result of a subtraction is positive or zero
- As the result of an inline barrel shifter operation in a move or logical instruction

Overflow occurs if the result of an add, subtract, or compare is greater than or equal to 2^{31} , or less than -2^{31} .

Most instructions update the status flags only if the S suffix is specified. See the instruction descriptions for more information.

Condition code suffixes

The instructions that can be conditional have an optional condition code, shown in syntax descriptions as {*cond*}. Conditional execution requires a preceding IT instruction. An instruction with a condition code is only executed if the condition code flags in the APSR meet the specified condition. [Table 23](#) shows the condition codes to use.

You can use conditional execution with the IT instruction to reduce the number of branch instructions in code.

[Table 23](#) also shows the relationship between condition code suffixes and the N, Z, C, and V flags.

Table 23. Condition code suffixes

Suffix	Flags	Meaning
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or HS	C = 1	Higher or same, unsigned \geq
CC or LO	C = 0	Lower, unsigned $<$
MI	N = 1	Negative
PL	N = 0	Positive or zero
VS	V = 1	Overflow
VC	V = 0	No overflow
HI	C = 1 and Z = 0	Higher, unsigned $>$
LS	C = 0 or Z = 1	Lower or same, unsigned \leq
GE	N = V	Greater than or equal, signed \geq
LT	N \neq V	Less than, signed $<$
GT	Z = 0 and N = V	Greater than, signed $>$
LE	Z = 1 and N \neq V	Less than or equal, signed \leq
AL	Can have any value	Always. This is the default when no suffix is specified.

[Specific example 1: Absolute value](#) shows the use of a conditional instruction to find the absolute value of a number. $R0 = ABS(R1)$.

Specific example 1: Absolute value

```
MOVSR0, R1; R0 = R1, setting flags
IT MI; IT instruction for the negative condition
RSBMR0, R1, #0; If negative, R0 = -R1
```

Specific example 2: Compare and update value shows the use of conditional instructions to update the value of R4 if the signed value R0 and R2 are greater than R1 and R3 respectively.

Specific example 2: Compare and update value

```
CMP R0, R1 ; compare R0 and R1, setting flags
ITT GT ; IT instruction for the two GT conditions
CMPGT R2, R3; if 'greater than', compare R2 and R3, setting flags
MOVGT R4, R5 ; if still 'greater than', do R4 = R5
```

3.3.8 Instruction width selection

There are many instructions that can generate either a 16-bit encoding or a 32-bit encoding depending on the operands and destination register specified. For some of these instructions, you can force a specific instruction size by using an instruction width suffix. The `.W` suffix forces a 32-bit instruction encoding. The `.N` suffix forces a 16-bit instruction encoding.

If you specify an instruction width suffix and the assembler cannot generate an instruction encoding of the requested width, it generates an error.

In some cases it might be necessary to specify the `.W` suffix, for example if the operand is the label of an instruction or literal data, as in the case of branch instructions. This is because the assembler might not automatically generate the right size encoding.

To use an instruction width suffix, place it immediately after the instruction mnemonic and condition code, if any. *Specific example 3: Instruction width selection* shows instructions with the instruction width suffix.

Specific example 3: Instruction width selection

```
BCS.W label; creates a 32-bit instruction even for a short branch
ADDS.W R0, R0, R1; creates a 32-bit instruction even though the same
; operation can be done by a 16-bit instruction
```

3.4 Memory access instructions

Table 24 shows the memory access instructions:

Table 24. Memory access instructions

Mnemonic	Brief description	Section
ADR	Load PC-relative address	ADR on page 60
CLREX	Clear exclusive	CLREX on page 71
LDM{mode}	Load multiple registers	LDM and STM on page 67
LDR{type}	Load register using immediate offset	LDR and STR, immediate offset on page 61
LDR{type}	Load register using register offset	LDR and STR, register offset on page 63
LDR{type}T	Load register with unprivileged access	LDR and STR, unprivileged on page 64
LDR	Load register using PC-relative address	LDR, PC-relative on page 65
LDREX{type}	Load register exclusive	LDREX and STREX on page 70
POP	Pop registers from stack	PUSH and POP on page 68
PUSH	Push registers onto stack	PUSH and POP on page 68
STM{mode}	Store multiple registers	LDM and STM on page 67
STR{type}	Store register using immediate offset	LDR and STR, immediate offset on page 61
STR{type}	Store register using register offset	LDR and STR, register offset on page 63
STR{type}T	Store register with unprivileged access	LDR and STR, unprivileged on page 64
STREX{type}	Store register exclusive	LDREX and STREX on page 70

3.4.1 ADR

Load PC-relative address.

Syntax

```
ADR{cond} Rd, label
```

where:

- '*cond*' is an optional condition code (see [Conditional execution on page 57](#))
- '*Rd*' is the destination register
- '*label*' is a PC-relative expression (see [PC-relative expressions on page 56](#))

Operation

ADR determines the address by adding an immediate value to the PC. It writes the result to the destination register.

ADR produces position-independent code, because the address is PC-relative.

If you use ADR to generate a target address for a BX or BLX instruction, you must ensure that bit[0] of the address you generate is set to 1 for correct execution.

Values of *label* must be within the range -4095 to 4095 from the address in the PC.

Note: You might have to use the *.W* suffix to get the maximum offset range or to generate addresses that are not word-aligned (see [Instruction width selection on page 59](#)).

Restrictions

Rd must be neither SP nor PC.

Condition flags

This instruction does not change the flags.

Examples

```
ADR R1, TextMessage; write address value of a location labelled as
; TextMessage to R1
```

3.4.2 LDR and STR, immediate offset

Load and store with immediate offset, pre-indexed immediate offset, or post-indexed immediate offset.

Syntax

```
op{type}{cond} Rt, [Rn {, #offset}]; immediate offset
op{type}{cond} Rt, [Rn, #offset]!; pre-indexed
op{type}{cond} Rt, [Rn], #offset; post-indexed
opD{cond} Rt, Rt2, [Rn {, #offset}]; immediate offset, two words
opD{cond} Rt, Rt2, [Rn, #offset]!; pre-indexed, two words
opD{cond} Rt, Rt2, [Rn], #offset; post-indexed, two words
```

where:

- '*op*' is either LDR (load register) or STR (store register)
- '*type*' is one of the following:
 - B: Unsigned byte, zero extends to 32 bits on loads
 - SB: Signed byte, sign extends to 32 bits (LDR only)
 - H: Unsigned halfword, zero extends to 32 bits on loads
 - SH: Signed halfword, sign extends to 32 bits (LDR only)
 - : Omit, for word
- '*cond*' is an optional condition code (see [Conditional execution on page 57](#))
- '*Rt*' is the register to load or store
- '*Rn*' is the register on which the memory address is based
- '*offset*' is an offset from *Rn*. If *offset* is omitted, the address is the contents of *Rn*
- '*Rt2*' is the additional register to load or store for two-word operations

Operation

LDR instructions load one or two registers with a value from memory. STR instructions store one or two register values to memory.

Load and store instructions with immediate offset can use the following addressing modes:

- Offset addressing

The offset value is added to or subtracted from the address obtained from the register *Rn*. The result is used as the address for the memory access. The register *Rn* is unaltered. The assembly language syntax for this mode is: [*Rn*, #*offset*].
- Pre-indexed addressing

The offset value is added to or subtracted from the address obtained from the register *Rn*. The result is used as the address for the memory access and written back into the register *Rn*. The assembly language syntax for this mode is: [*Rn*, #*offset*]!
- Post-indexed addressing

The address obtained from the register *Rn* is used as the address for the memory access. The offset value is added to or subtracted from the address, and written back into the register *Rn*. The assembly language syntax for this mode is: [*Rn*], #*offset*.

The value to load or store can be a byte, halfword, word, or two words. Bytes and halfwords can either be signed or unsigned (see [Address alignment on page 56](#)).

[Table 25](#) shows the range of offsets for immediate, pre-indexed and post-indexed forms.

Table 25. Immediate, pre-indexed and post-indexed offset ranges

Instruction type	Immediate offset	Pre-indexed	Post-indexed
Word, halfword, signed halfword, byte, or signed byte	-255 to 4095	-255 to 255	-255 to 255
Two words	Multiple of 4 in the range -1020 to 1020	Multiple of 4 in the range -1020 to 1020	Multiple of 4 in the range -1020 to 1020

Restrictions

- For load instructions
 - *Rt* can be SP or PC for word loads only
 - *Rt* must be different from *Rt2* for two-word loads
 - *Rn* must be different from *Rt* and *Rt2* in the pre-indexed or post-indexed forms
- When *Rt* is PC in a word load instruction
 - bit[0] of the loaded value must be 1 for correct execution
 - A branch occurs to the address created by changing bit[0] of the loaded value to 0
 - If the instruction is conditional, it must be the last instruction in the IT block
- For store instructions
 - *Rt* can be SP for word stores only
 - *Rt* must not be PC
 - *Rn* must not be PC
 - *Rn* must be different from *Rt* and *Rt2* in the pre-indexed or post-indexed forms

Condition flags

These instructions do not change the flags.

Examples

```
LDRR8, [R10]; loads R8 from the address in R10.
LDRNER2, [R5, #960]!; loads (conditionally) R2 from a word
; 960 bytes above the address in R5, and
; increments R5 by 960.
STRR2, [R9,#const-struct]; const-struct is an expression evaluating
; to a constant in the range 0-4095.
STRHR3, [R4], #4; Store R3 as halfword data into address in
; R4, then increment R4 by 4
LDRD R8, R9, [R3, #0x20]; Load R8 from a word 32 bytes above the
; address in R3, and load R9 from a word 36
; bytes above the address in R3
STRDR0, R1, [R8], #-16; Store R0 to address in R8, and store R1 to
; a word 4 bytes above the address in R8,
; and then decrement R8 by 16.
```

3.4.3 LDR and STR, register offset

Load and store with register offset.

Syntax

```
op{type}{cond} Rt, [Rn, Rm {, LSL #n}]
```

where:

- 'op' is either LDR (load register) or STR (store register)
- 'type' is one of the following:
 - B: Unsigned byte, zero extends to 32 bits on loads
 - SB: Signed byte, sign extends to 32 bits (LDR only)
 - H: Unsigned halfword, zero extends to 32 bits on loads
 - SH: Signed halfword, sign extends to 32 bits (LDR only)
 - : Omit, for word
- 'cond' is an optional condition code (see [Conditional execution on page 57](#))
- 'Rt' is the register to load or store
- 'Rn' is the register on which the memory address is based
- 'Rm' is a register containing a value to be used as the offset
- 'LSL #n' is an optional shift, with *n* in the range 0 to 3

Operation

LDR instructions load a register with a value from memory. STR instructions store a register value into memory.

The memory address to load from or store to is at an offset from the register *Rn*. The offset is specified by the register *Rm* and can be shifted left by up to 3 bits using LSL.

The value to load or store can be a byte, halfword, or word. For load instructions, bytes and halfwords can either be signed or unsigned (see [Address alignment on page 56](#)).

Restrictions

In these instructions:

- *Rn* must not be PC
- *Rm* must be neither SP nor PC
- *Rt* can be SP only for word loads and word stores
- *Rt* can be PC only for word loads

When *Rt* is PC in a word load instruction:

- bit[0] of the loaded value must be 1 for correct execution, and a branch occurs to this halfword-aligned address
- If the instruction is conditional, it must be the last instruction in the IT block.

Condition flags

These instructions do not change the flags.

Examples

```
STRRO, [R5, R1]; store value of R0 into an address equal to
; sum of R5 and R1
LDRSBR0, [R5, R1, LSL #1]; read byte value from an address equal to
; sum of R5 and two times R1, sign extended it
; to a word value and put it in R0
STRRO, [R1, R2, LSL #2]; stores R0 to an address equal to sum of R1
; and four times R2
```

3.4.4 LDR and STR, unprivileged

Load and store with unprivileged access.

Syntax

```
op{type}T{cond} Rt, [Rn {, #offset}]; immediate offset
```

where:

- '*op*' is either LDR (load register) or STR (store register)
- '*type*' is one of the following:
 - B: Unsigned byte, zero extends to 32 bits on loads
 - SB: Signed byte, sign extends to 32 bits (LDR only)
 - H: Unsigned halfword, zero extends to 32 bits on loads
 - SH: Signed halfword, sign extends to 32 bits (LDR only)
 - : Omit, for word
- '*cond*' is an optional condition code (see [Conditional execution on page 57](#))
- '*Rt*' is the register to load or store
- '*Rn*' is the register on which the memory address is based
- '*offset*' is an offset from *Rn* and can be 0 to 255. If *offset* is omitted, the address is the value in *Rn*.

Operation

These load and store instructions perform the same function as the memory access instructions with immediate offset (see [LDR and STR, immediate offset on page 61](#)). The difference is that these instructions have only unprivileged access even when used in privileged software.

When used in unprivileged software, these instructions behave in exactly the same way as normal memory access instructions with immediate offset.

Restrictions

In these instructions:

- *Rn* must not be PC
- *Rt* must be neither SP nor PC.

Condition flags

These instructions do not change the flags.

Examples

```
STRBTEQR4, [R7]; conditionally store least significant byte in
; R4 to an address in R7, with unprivileged access
LDRHTR2, [R2, #8]; load halfword value from an address equal to
; sum of R2 and 8 into R2, with unprivileged access
```

3.4.5 LDR, PC-relative

Load register from memory.

Syntax

```
LDR{type}{cond} Rt, label
LDRD{cond} Rt, Rt2, label; load two words
```

where:

- '*type*' is one of the following:
 - B: Unsigned byte, zero extends to 32 bits
 - SB: Signed byte, sign extends to 32 bits
 - H: Unsigned halfword, sign extends to 32 bits
 - SH: Signed halfword, sign extends to 32 bits
 - : Omit, for word
- '*cond*' is an optional condition code (see [Conditional execution on page 57](#))
- '*Rt*' is the register to load or store
- '*Rt2*' is the second register to load or store
- '*label*' is a PC-relative expression (see [PC-relative expressions on page 56](#))

Operation

LDR loads a register with a value from a PC-relative memory address. The memory address is specified by a label or by an offset from the PC.

The value to load or store can be a byte, halfword, or word. For load instructions, bytes and halfwords can either be signed or unsigned (see [Address alignment on page 56](#)).

'label' must be within a limited range of the current instruction. [Table 26](#) shows the possible offsets between label and the PC.

Table 26. label-PC offset ranges

Instruction type	Offset range
Word, halfword, signed halfword, byte, signed byte	–4095 to 4095
Two words	–1020 to 1020

You might have to use the .W suffix to get the maximum offset range (see [Instruction width selection on page 59](#)).

Restrictions

In these instructions:

- *Rt* can be SP or PC only for word loads
- *Rt2* must be neither SP nor PC
- *Rt* must be different from *Rt2*

When *Rt* is PC in a word load instruction:

- bit[0] of the loaded value must be 1 for correct execution, and a branch occurs to this halfword-aligned address
- If the instruction is conditional, it must be the last instruction in the IT block.

Condition flags

These instructions do not change the flags.

Examples

```
LDRR0, LookUpTable; load R0 with a word of data from an address
; labelled as LookUpTable
LDRSBR7, localdata; load a byte value from an address labelled
; as localdata, sign extend it to a word
; value, and put it in R7
```

3.4.6 LDM and STM

Load and store multiple registers.

Syntax

```
op{addr_mode}{cond} Rn{!}, reglist
```

where:

- 'op' is either LDM (load multiple register) or STM (store multiple register)
- 'addr_mode' is any of the following:
 - IA: Increment address after each access (this is the default)
 - DB: Decrement address before each access
- 'cond' is an optional condition code (see [Conditional execution on page 57](#))
- 'Rn' is the register on which the memory addresses are based
- '!' is an optional writeback suffix. If '!' is present, the final address that is loaded from or stored to is written back into Rn.
- 'reglist' is a list of one or more registers to be loaded or stored, enclosed in braces. It can contain register ranges. It must be comma-separated if it contains more than one register or register range (see [Examples on page 68](#)).

LDM and LDMFD are synonyms for LDMIA. LDMFD refers to its use for popping data from full descending stacks.

LDMEA is a synonym for LDMDB, and refers to its use for popping data from empty ascending stacks.

STM and STMEA are synonyms for STMIA. STMEA refers to its use for pushing data onto empty ascending stacks.

STMFD is a synonym for STMDB, and refers to its use for pushing data onto full descending stacks

Operation

LDM instructions load the registers in *reglist* with word values from memory addresses based on *Rn*.

STM instructions store the word values in the registers in *reglist* to memory addresses based on *Rn*.

For LDM, LDMIA, LDMFD, STM, STMIA, and STMEA the memory addresses used for the accesses are at 4-byte intervals ranging from Rn to $Rn + 4 * (n-1)$, where n is the number of registers in *reglist*. The accesses happen in order of increasing register numbers, with the lowest numbered register using the lowest memory address and the highest number register using the highest memory address. If the writeback suffix is specified, the value of $Rn + 4 * (n-1)$ is written back to *Rn*.

For LDMDB, LDMEA, STMDB, and STMFD the memory addresses used for the accesses are at 4-byte intervals ranging from Rn to $Rn - 4 * (n-1)$, where n is the number of registers in *reglist*. The accesses happen in order of decreasing register numbers, with the highest numbered register using the highest memory address and the lowest number register using the lowest memory address. If the writeback suffix is specified, the value $Rn - 4 * (n-1)$ is written back to *Rn*.

The PUSH and POP instructions can be expressed in this form (see [PUSH and POP](#) for details).

Restrictions

In these instructions:

- *Rn* must not be PC
- *reglist* must not contain SP
- In any STM instruction, *reglist* must not contain PC
- In any LDM instruction, *reglist* must not contain PC if it contains LR
- *reglist* must not contain *Rn* if you specify the writeback suffix

When PC is in *reglist* in an LDM instruction:

- bit[0] of the value loaded to the PC must be 1 for correct execution, and a branch occurs to this halfword-aligned address.
- If the instruction is conditional, it must be the last instruction in the IT block

Condition flags

These instructions do not change the flags.

Examples

```
LDMR8, {R0,R2,R9}; LDMIA is a synonym for LDM
STMDBR1!, {R3-R6,R11,R12}
```

Incorrect examples

```
STM5!, {R5,R4,R9}; value stored for R5 is unpredictable
LDMR2, {}; there must be at least one register in the list
```

3.4.7 PUSH and POP

Push registers onto, and pop registers off a full-descending stack.

Syntax

```
PUSH{cond} reglist
```

```
POP{cond} reglist
```

where:

- '*cond*' is an optional condition code (see [Conditional execution on page 57](#))
- '*reglist*' is a non-empty list of registers, enclosed in braces. It can contain register ranges. It must be comma-separated if it contains more than one register or register range (see [Examples on page 68](#)).

PUSH and POP are synonyms for STMDB and LDM (or LDMIA) with the memory addresses for the access based on SP, and with the final address for the access written back to the SP. PUSH and POP are the preferred mnemonics in these cases.

Operation

PUSH stores registers on the stack in order of decreasing register numbers, with the highest numbered register using the highest memory address and the lowest numbered register using the lowest memory address.

POP loads registers from the stack in order of increasing register numbers, with the lowest numbered register using the lowest memory address and the highest numbered register using the highest memory address.

See [LDM and STM on page 67](#) for more information.

Restrictions

In these instructions:

- '*reglist*' must not contain SP
- For the PUSH instruction, *reglist* must not contain PC
- For the POP instruction, *reglist* must not contain PC if it contains LR

When PC is in *reglist* in a POP instruction:

- bit[0] of the value loaded to the PC must be 1 for correct execution, and a branch occurs to this halfword-aligned address.
- If the instruction is conditional, it must be the last instruction in the IT block.

Condition flags

These instructions do not change the flags.

Examples

```
PUSH{R0, R4-R7}
PUSH{R2, LR}
POP{R0, R10, PC}
```

3.4.8 LDREX and STREX

Load and store register exclusive.

Syntax

```
LDREX{cond} Rt, [Rn {, #offset}]
```

```
STREX{cond} Rd, Rt, [Rn {, #offset}]
```

```
LDREXB{cond} Rt, [Rn]
```

```
STREXB{cond} Rd, Rt, [Rn]
```

```
LDREXH{cond} Rt, [Rn]
```

```
STREXH{cond} Rd, Rt, [Rn]
```

where:

- ‘*cond*’ is an optional condition code (see [Conditional execution on page 57](#))
- ‘*Rd*’ is the destination register for the returned status
- ‘*Rt*’ is the register to load or store
- ‘*Rn*’ is the register on which the memory address is based
- ‘*offset*’ is an optional offset applied to the value in *Rn*. If *offset* is omitted, the address is the value in *Rn*.

Operation

LDREX, LDREXB, and LDREXH load a word, byte, and halfword respectively from a memory address.

STREX, STREXB, and STREXH attempt to store a word, byte, and halfword respectively to a memory address. The address used in any store-exclusive instruction must be the same as the address in the most recently executed load-exclusive instruction. The value stored by the Store-exclusive instruction must also have the same data size as the value loaded by the preceding load-exclusive instruction. This means software must always use a load-exclusive instruction and a matching store-exclusive instruction to perform a synchronization operation, see [Synchronization primitives on page 31](#).

If a store-exclusive instruction performs the store, it writes 0 to its destination register. If it does not perform the store, it writes 1 to its destination register. If the store-exclusive instruction writes 0 to the destination register, it is guaranteed that no other process in the system has accessed the memory location between the load-exclusive and store-exclusive instructions.

For reasons of performance, keep the number of instructions between corresponding load-exclusive and store-exclusive instruction to a minimum.

Note: *The result of executing a store-exclusive instruction to an address that is different from that used in the preceding load-exclusive instruction is unpredictable.*

Restrictions

In these instructions:

- Do not use PC
- Do not use SP for *Rd* and *Rt*
- For STREX, *Rd* must be different from both *Rt* and *Rn*
- The value of offset must be a multiple of four in the range 0-1020

Condition flags

These instructions do not change the flags.

Examples

```

MOVR1, #0x1; initialize the 'lock taken' value
LDREXR0, [LockAddr]; load the lock value
CMPR0, #0; is the lock free?
ITTEQ; IT instruction for STREXEQ and CMPEQ
STREXEQR0, R1, [LockAddr]; try and claim the lock
CMPEQR0, #0; did this succeed?
BNEtry; no - try again
; yes - we have the lock

```

3.4.9 CLREX

Clear exclusive.

Syntax

```
CLREX{cond}
```

where:

'*cond*' is an optional condition code (see [Conditional execution on page 57](#))

Operation

Use CLREX to make the next STREX, STREXB, or STREXH instruction write 1 to its destination register and fail to perform the store. It is useful in exception handler code to force the failure of the store exclusive if the exception occurs between a load exclusive instruction and the matching store exclusive instruction in a synchronization operation.

See [Synchronization primitives on page 31](#) for more information.

Condition flags

These instructions do not change the flags.

Examples

```
CLREX
```

3.5 General data processing instructions

Table 27 shows the data processing instructions.

Table 27. Data processing instructions

Mnemonic	Brief description	See
ADC	Add with carry	<i>ADD, ADC, SUB, SBC, and RSB on page 73</i>
ADD	Add	<i>ADD, ADC, SUB, SBC, and RSB on page 73</i>
ADDW	Add	<i>ADD, ADC, SUB, SBC, and RSB on page 73</i>
AND	Logical AND	<i>AND, ORR, EOR, BIC, and ORN on page 75</i>
ASR	Arithmetic shift right	<i>ASR, LSL, LSR, ROR, and RRX on page 76</i>
BIC	Bit clear	<i>AND, ORR, EOR, BIC, and ORN on page 75</i>
CLZ	Count leading zeros	<i>CLZ on page 77</i>
CMN	Compare negative	<i>CMP and CMN on page 78</i>
CMP	Compare	<i>CMP and CMN on page 78</i>
EOR	Exclusive OR	<i>AND, ORR, EOR, BIC, and ORN on page 75</i>
LSL	Logical shift left	<i>ASR, LSL, LSR, ROR, and RRX on page 76</i>
LSR	Logical shift right	<i>ASR, LSL, LSR, ROR, and RRX on page 76</i>
MOV	Move	<i>MOV and MVN on page 79</i>
MOVT	Move top	<i>MOVT on page 80</i>
MOVW	Move 16-bit constant	<i>MOV and MVN on page 79</i>
MVN	Move NOT	<i>MOV and MVN on page 79</i>
ORN	Logical OR NOT	<i>AND, ORR, EOR, BIC, and ORN on page 75</i>
ORR	Logical OR	<i>AND, ORR, EOR, BIC, and ORN on page 75</i>
RBIT	Reverse bits	<i>REV, REV16, REVSH, and RBIT on page 81</i>
REV	Reverse byte order in a word	<i>REV, REV16, REVSH, and RBIT on page 81</i>
REV16	Reverse byte order in each halfword	<i>REV, REV16, REVSH, and RBIT on page 81</i>
REVSH	Reverse byte order in bottom halfword and sign extend	<i>REV, REV16, REVSH, and RBIT on page 81</i>
ROR	Rotate right	<i>ASR, LSL, LSR, ROR, and RRX on page 76</i>
RRX	Rotate right with extend	<i>ASR, LSL, LSR, ROR, and RRX on page 76</i>
RSB	Reverse subtract	<i>ADD, ADC, SUB, SBC, and RSB on page 73</i>
SBC	Subtract with carry	<i>ADD, ADC, SUB, SBC, and RSB on page 73</i>
SUB	Subtract	<i>ADD, ADC, SUB, SBC, and RSB on page 73</i>
SUBW	Subtract	<i>ADD, ADC, SUB, SBC, and RSB on page 73</i>
TEQ	Test equivalence	<i>TST and TEQ on page 82</i>
TST	Test	<i>TST and TEQ on page 82</i>

3.5.1 ADD, ADC, SUB, SBC, and RSB

Add, add with carry, subtract, subtract with carry, and reverse subtract.

Syntax

```
op{S}{cond} {Rd,} Rn, Operand2
```

```
op{cond} {Rd,} Rn, #imm12; ADD and SUB only
```

where:

- 'op' is one of:
 - ADD: Add
 - ADC: Add with carry
 - SUB: Subtract
 - SBC: Subtract with carry
 - RSB: Reverse subtract
- 'S' is an optional suffix. If S is specified, the condition code flags are updated on the result of the operation (see [Conditional execution on page 57](#))
- 'cond' is an optional condition code (see [Conditional execution on page 57](#))
- 'Rd' is the destination register. If Rd is omitted, the destination register is Rn
- 'Rn' is the register holding the first operand
- 'Operand2' is a flexible second operand (see [Flexible second operand on page 52](#) for details of the options).
- 'imm12' is any value in the range 0—4095

Operation

The ADD instruction adds the value of *operand2* or *imm12* to the value in *Rn*.

The ADC instruction adds the values in *Rn* and *operand2*, together with the carry flag.

The SUB instruction subtracts the value of *operand2* or *imm12* from the value in *Rn*.

The SBC instruction subtracts the value of *operand2* from the value in *Rn*. If the carry flag is clear, the result is reduced by one.

The RSB instruction subtracts the value in *Rn* from the value of *operand2*. This is useful because of the wide range of options for *operand2*.

Use ADC and SBC to synthesize multiword arithmetic (see [Multiword arithmetic examples on page 74](#) and [ADR on page 60](#)).

ADDW is equivalent to the ADD syntax that uses the *imm12* operand. SUBW is equivalent to the SUB syntax that uses the *imm12* operand.

Restrictions

In these instructions:

- *Operand2* must be neither SP nor PC
- *Rd* can be SP only in ADD and SUB, and only with the following additional restrictions:
 - *Rn* must also be SP
 - Any shift in operand2 must be limited to a maximum of three bits using LSL
- *Rn* can be SP only in ADD and SUB
- *Rd* can be PC only in the ADD{cond} PC, PC, *Rm* instruction where:
 - You must not specify the S suffix
 - *Rm* must be neither PC nor SP
 - If the instruction is conditional, it must be the last instruction in the IT block
- With the exception of the ADD{cond} PC, PC, *Rm* instruction, *Rn* can be PC only in ADD and SUB, and only with the following additional restrictions:
 - You must not specify the S suffix
 - The second operand must be a constant in the range 0 to 4095

- Note:*
- 1 When using the PC for an addition or a subtraction, bits[1:0] of the PC are rounded to b00 before performing the calculation, making the base address for the calculation word-aligned.
 - 2 If you want to generate the address of an instruction, you have to adjust the constant based on the value of the PC. ARM recommends that you use the ADR instruction instead of ADD or SUB with *Rn* equal to the PC, because your assembler automatically calculates the correct constant for the ADR instruction.

When *Rd* is PC in the ADD{cond} PC, PC, *Rm* instruction:

- bit[0] of the value written to the PC is ignored
- A branch occurs to the address created by forcing bit[0] of that value to 0

Condition flags

If S is specified, these instructions update the N, Z, C and V flags according to the result.

Examples

```
ADDR2, R1, R3
SUBSR8, R6, #240; sets the flags on the result
RSBR4, R4, #1280; subtracts contents of R4 from 1280
ADCHIR11, R0, R3; only executed if C flag set and Z
; flag clear
```

Multiword arithmetic examples

[Specific example 4: 64-bit addition](#) shows two instructions that add a 64-bit integer contained in R2 and R3 to another 64-bit integer contained in R0 and R1, and place the result in R4 and R5.

Specific example 4: 64-bit addition

```
ADDSR4, R0, R2; add the least significant words
ADCR5, R1, R3; add the most significant words with carry
```

Multiword values do not have to use consecutive registers. [Specific example 5: 96-bit subtraction](#) shows instructions that subtract a 96-bit integer contained in R9, R1, and R11 from another contained in R6, R2, and R8. The example stores the result in R6, R9, and R2.

Specific example 5: 96-bit subtraction

```
SUBSR6, R6, R9; subtract the least significant words
SBCSR9, R2, R1; subtract the middle words with carry
SBCR2, R8, R11; subtract the most significant words with carry
```

3.5.2 AND, ORR, EOR, BIC, and ORN

Logical AND, OR, exclusive OR, bit clear, and OR NOT.

Syntax

```
op{S}{cond} {Rd,} Rn, Operand2
```

where:

- 'op' is one of:
AND: Logical AND
ORR: Logical OR or bit set
EOR: Logical exclusive OR
BIC: Logical AND NOT or bit clear
ORN: Logical OR NOT
- 'S' is an optional suffix. If S is specified, the condition code flags are updated on the result of the operation (see [Conditional execution on page 57](#)).
- 'cond' is an optional condition code (see [Conditional execution on page 57](#))
- 'Rd' is the destination register
- 'Rn' is the register holding the first operand
- 'Operand2' is a flexible second operand (see [Flexible second operand on page 52](#) for details of the options).

Operation

The AND, EOR, and ORR instructions perform bitwise AND, exclusive OR, and OR operations on the values in *Rn* and *operand2*.

The BIC instruction performs an AND operation on the bits in *Rn* with the complements of the corresponding bits in the value of *operand2*.

The ORN instruction performs an OR operation on the bits in *Rn* with the complements of the corresponding bits in the value of *operand2*.

Restrictions

Do not use either SP or PC.

Condition flags

If S is specified, these instructions:

- Update the N and Z flags according to the result
- Can update the C flag during the calculation of *operand2* (see [Flexible second operand on page 52](#))
- Do not affect the V flag

Examples

```
ANDR9, R2, #0xFF00
ORREQR2, R0, R5
ANDSR9, R8, #0x19
EORSR7, R11, #0x18181818
BICR0, R1, #0xab
ORNR7, R11, R14, ROR #4
ORNSR7, R11, R14, ASR #32
```

3.5.3 ASR, LSL, LSR, ROR, and RRX

Arithmetic shift right, logical shift left, logical shift right, rotate right, and rotate right with extend.

Syntax

```
op{S}{cond} Rd, Rm, Rs
```

```
op{S}{cond} Rd, Rm, #n
```

```
RRX{S}{cond} Rd, Rm
```

where:

- 'op' is one of:
 - ASR: Arithmetic shift right
 - LSL: Logical shift left
 - LSR: Logical shift right
 - ROR: Rotate right
- 'S' is an optional suffix. If S is specified, the condition code flags are updated on the result of the operation (see [Conditional execution on page 57](#))
- 'Rd' is the destination register
- 'Rm' is the register holding the value to be shifted
- 'Rs' is the register holding the shift length to apply to the value *Rm*. Only the least significant byte is used and can be in the range 0 to 255.
- 'n' is the shift length. The range of shift lengths depend on the instruction as follows:
 - ASR: Shift length from 1 to 32
 - LSL: Shift length from 0 to 31
 - LSR: Shift length from 1 to 32
 - ROR: Shift length from 1 to 31

Note: `MOV{S}{cond} Rd, Rm` is the preferred syntax for `LSL{S}{cond} Rd, Rm, #0`.

Operation

ASR, LSL, LSR, and ROR move the bits in the register *Rm* to the left or right by the number of places specified by constant *n* or register *Rs*.

RRX moves the bits in register *Rm* to the right by 1.

In all these instructions, the result is written to *Rd*, but the value in register *Rm* remains unchanged. For details on what result is generated by the different instructions (see [Shift operations on page 53](#)).

Restrictions

Do not use either SP or PC.

Condition flags

If S is specified:

- These instructions update the N and Z flags according to the result
- The C flag is updated to the last bit shifted out, except when the shift length is 0 (see [Shift operations on page 53](#)).

Examples

```
ASRR7, R8, #9; arithmetic shift right by 9 bits
LSLSR1, R2, #3; logical shift left by 3 bits with flag update
LSRR4, R5, #6; logical shift right by 6 bits
RORR4, R5, R6; rotate right by the value in the bottom byte of R6
RRXR4, R5; rotate right with extend
```

3.5.4 CLZ

Count leading zeros.

Syntax

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

where:

- '*cond*' is an optional condition code (see [Conditional execution on page 57](#))
- '*Rd*' is the destination register
- '*Rm*' is the operand register

Operation

The CLZ instruction counts the number of leading zeros in the value in *Rm* and returns the result in *Rd*. The result value is 32 if no bits are set in the source register, and zero if bit[31] is set.

Restrictions

Do not use either SP or PC.

Condition flags

This instruction does not change the flags.

Examples

```
CLZR4, R9
CLZNER2, R3
```

3.5.5 CMP and CMN

Compare and compare negative.

Syntax

```
CMP{cond} Rn, Operand2
```

```
CMN{cond} Rn, Operand2
```

where:

- '*cond*' is an optional condition code (see [Conditional execution on page 57](#))
- '*Rn*' is the register holding the first operand
- '*Operand2*' is a flexible second operand (see [Flexible second operand on page 52](#)) for details of the options.

Operation

These instructions compare the value in a register with *operand2*. They update the condition flags on the result, but do not write the result to a register.

The CMP instruction subtracts the value of *operand2* from the value in *Rn*. This is the same as a SUBS instruction, except that the result is discarded.

The CMN instruction adds the value of *operand2* to the value in *Rn*. This is the same as an ADDS instruction, except that the result is discarded.

Restrictions

In these instructions:

- Do not use PC
- *Operand2* must not be SP

Condition flags

These instructions update the N, Z, C and V flags according to the result.

Examples

```
CMPR2, R9
CMNR0, #6400
CMPGTSP, R7, LSL #2
```

3.5.6 MOV and MVN

Move and move NOT.

Syntax

```
MOV{S}{cond} Rd, Operand2
```

```
MOV{cond} Rd, #imm16
```

```
MVN{S}{cond} Rd, Operand2
```

where:

- 'S' is an optional suffix. If S is specified, the condition code flags are updated on the result of the operation (see [Conditional execution on page 57](#)).
- 'cond' is an optional condition code (see [Conditional execution on page 57](#)).
- 'Rd' is the destination register
- 'Operand2' is a flexible second operand (see [Flexible second operand on page 52](#)) for details of the options.
- 'imm16' is any value in the range 0—65535

Operation

The MOV instruction copies the value of *operand2* into *Rd*.

When *operand2* in a MOV instruction is a register with a shift other than LSL #0, the preferred syntax is the corresponding shift instruction:

- ASR{S}{cond} Rd, Rm, #n is the preferred syntax for MOV{S}{cond} Rd, Rm, ASR #n
- LSL{S}{cond} Rd, Rm, #n is the preferred syntax for MOV{S}{cond} Rd, Rm, LSL #n if n != 0
- LSR{S}{cond} Rd, Rm, #n is the preferred syntax for MOV{S}{cond} Rd, Rm, LSR #n
- ROR{S}{cond} Rd, Rm, #n is the preferred syntax for MOV{S}{cond} Rd, Rm, ROR #n
- RRX{S}{cond} Rd, Rm is the preferred syntax for MOV{S}{cond} Rd, Rm, RRX

Also, the MOV instruction permits additional forms of *operand2* as synonyms for shift instructions:

- MOV{S}{cond} Rd, Rm, ASR Rs is a synonym for ASR{S}{cond} Rd, Rm, Rs
- MOV{S}{cond} Rd, Rm, LSL Rs is a synonym for LSL{S}{cond} Rd, Rm, Rs
- MOV{S}{cond} Rd, Rm, LSR Rs is a synonym for LSR{S}{cond} Rd, Rm, Rs
- MOV{S}{cond} Rd, Rm, ROR Rs is a synonym for ROR{S}{cond} Rd, Rm, Rs

See [ASR, LSL, LSR, ROR, and RRX on page 76](#).

The MVN instruction takes the value of *operand2*, performs a bitwise logical NOT operation on the value, and places the result into *Rd*.

Note: The MOVW instruction provides the same function as MOV, but is restricted to using the imm16 operand.

Restrictions

You can use SP and PC only in the MOV instruction, with the following restrictions:

- The second operand must be a register without shift
- You must not specify the S suffix

When *Rd* is PC in a MOV instruction:

- bit[0] of the value written to the PC is ignored
- A branch occurs to the address created by forcing bit[0] of that value to 0.

Note: Though it is possible to use MOV as a branch instruction, ARM strongly recommends the use of a BX or BLX instruction to branch for software portability to the ARM instruction set.

Condition flags

If S is specified, these instructions:

- Update the N and Z flags according to the result
- Can update the C flag during the calculation of *operand2* (see [Flexible second operand on page 52](#)).
- Do not affect the V flag

Example

```
MOVSR11, #0x000B; write value of 0x000B to R11, flags get updated
MOVR1, #0xFA05; write value of 0xFA05 to R1, flags are not updated
MOVSR10, R12; write value in R12 to R10, flags get updated
MOVR3, #23; write value of 23 to R3
MOVR8, SP; write value of stack pointer to R8
MVNSR2, #0xF; write value of 0xFFFFF0 (bitwise inverse of 0xF)
; to the R2 and update flags
```

3.5.7 MOVT

Move top.

Syntax

```
MOVT{cond} Rd, #imm16
```

where:

- '*cond*' is an optional condition code (see [Conditional execution on page 57](#))
- '*Rd*' is the destination register
- '*imm16*' is a 16-bit immediate constant

Operation

MOVT writes a 16-bit immediate value, *imm16*, to the top halfword, *Rd*[31:16], of its destination register. The write does not affect *Rd*[15:0].

The MOV, MOVT instruction pair enables you to generate any 32-bit constant.

Restrictions

Rd must be neither SP nor PC.

Condition flags

This instruction does not change the flags.

Examples

```
MOVTR3, #0xF123; write 0xF123 to upper halfword of R3, lower halfword
; and APSR are unchanged
```

3.5.8 REV, REV16, REVSH, and RBIT

Reverse bytes and reverse bits.

Syntax

```
op{cond} Rd, Rn
```

where:

- '*op*' is one of:
 - REV: Reverse byte order in a word
 - REV16: Reverse byte order in each halfword independently
 - REVSH: Reverse byte order in the bottom halfword, and sign extends to 32 bits
 - RBIT: Reverse the bit order in a 32-bit word
- '*cond*' is an optional condition code (see [Conditional execution on page 57](#))
- '*Rd*' is the destination register
- '*Rn*' is the register holding the operand

Operation

Use these instructions to change endianness of data:

- REV: Converts 32-bit big-endian data into little-endian data or 32-bit little-endian data into big-endian data.
- REV16: Converts 16-bit big-endian data into little-endian data or 16-bit little-endian data into big-endian data.
- REVSH: Converts either:
 - 16-bit signed big-endian data into 32-bit signed little-endian data
 - 16-bit signed little-endian data into 32-bit signed big-endian data

Restrictions

Do not use either SP or PC.

Condition flags

These instructions do not change the flags.

Examples

```
REVR3, R7; reverse byte order of value in R7 and write it to R3
REV16 R0, R0; reverse byte order of each 16-bit halfword in R0
```

```

REVSH R0, R5 ; reverse Signed Halfword
REVHS R3, R7 ; reverse with Higher or Same condition
RBIT R7, R8 ; reverse bit order of value in R8 and write the result to R7

```

3.5.9 TST and TEQ

Test bits and test equivalence.

Syntax

```
TST{cond} Rn, Operand2
```

```
TEQ{cond} Rn, Operand2
```

where:

- ‘*cond*’ is an optional condition code (see [Conditional execution on page 57](#))
- ‘*Rn*’ is the register holding the first operand
- ‘*Operand2*’ is a flexible second operand (see [Flexible second operand on page 52](#)) for details of the options.

Operation

These instructions test the value in a register against *operand2*. They update the condition flags based on the result, but do not write the result to a register.

The TST instruction performs a bitwise AND operation on the value in *Rn* and the value of *operand2*. This is the same as the ANDS instruction, except that it discards the result.

To test whether a bit of *Rn* is 0 or 1, use the TST instruction with an *operand2* constant that has that bit set to 1 and all other bits cleared to 0.

The TEQ instruction performs a bitwise exclusive OR operation on the value in *Rn* and the value of *operand2*. This is the same as the EORS instruction, except that it discards the result.

Use the TEQ instruction to test if two values are equal without affecting the V or C flags.

TEQ is also useful for testing the sign of a value. After the comparison, the N flag is the logical exclusive OR of the sign bits of the two operands.

Restrictions

Do not use either SP or PC.

Condition flags

These instructions:

- Update the N and Z flags according to the result
- Can update the C flag during the calculation of *operand2* (see [Flexible second operand on page 52](#)).
- Do not affect the V flag

Examples

```

TSTR0, #0x3F8; perform bitwise AND of R0 value to 0x3F8,
; APSR is updated but result is discarded
TEQEQR10, R9; conditionally test if value in R10 is equal to
; value in R9, APSR is updated but result is discarded

```

3.6 Multiply and divide instructions

[Table 28](#) shows the multiply and divide instructions.

Table 28. Multiply and divide instructions

Mnemonic	Brief description	See
MLA	Multiply with accumulate, 32-bit result	MUL, MLA, and MLS on page 83
MLS	Multiply and subtract, 32-bit result	MUL, MLA, and MLS on page 83
MUL	Multiply, 32-bit result	MUL, MLA, and MLS on page 83
SDIV	Signed divide	SDIV and UDIV on page 86
SMLAL	Signed multiply with accumulate (32x32+64), 64-bit result	UMULL, UMLAL, SMULL, and SMLAL on page 85
SMULL	Signed multiply (32x32), 64-bit result	UMULL, UMLAL, SMULL, and SMLAL on page 85
UDIV	Unsigned divide	SDIV and UDIV on page 86
UMLAL	Unsigned multiply with accumulate (32x32+64), 64-bit result	UMULL, UMLAL, SMULL, and SMLAL on page 85
UMULL	Unsigned multiply (32x32), 64-bit result	UMULL, UMLAL, SMULL, and SMLAL on page 85

3.6.1 MUL, MLA, and MLS

Multiply, multiply with accumulate, and multiply with subtract, using 32-bit operands, and producing a 32-bit result.

Syntax

```
MUL{S}{cond} {Rd,} Rn, Rm ; Multiply
```

```
MLA{cond} Rd, Rn, Rm, Ra ; Multiply with accumulate
```

```
MLS{cond} Rd, Rn, Rm, Ra ; Multiply with subtract
```

where:

- '*cond*' is an optional condition code (see [Conditional execution on page 57](#))
- '*S*' is an optional suffix. If *S* is specified, the condition code flags are updated on the result of the operation (see [Conditional execution on page 57](#)).
- '*Rd*' is the destination register. If *Rd* is omitted, the destination register is *Rn*
- '*Rn*', '*Rm*' are registers holding the values to be multiplied
- '*Ra*' is a register holding the value to be added to or subtracted from

Operation

The MUL instruction multiplies the values from *Rn* and *Rm*, and places the least significant 32 bits of the result in *Rd*.

The MLA instruction multiplies the values from *Rn* and *Rm*, adds the value from *Ra*, and places the least significant 32 bits of the result in *Rd*.

The MLS instruction multiplies the values from *Rn* and *Rm*, subtracts the product from the value from *Ra*, and places the least significant 32 bits of the result in *Rd*.

The results of these instructions do not depend on whether the operands are signed or unsigned.

Restrictions

In these instructions, do not use SP and do not use PC.

If you use the S suffix with the MUL instruction:

- *Rd*, *Rn*, and *Rm* must all be in the range R0 to R7
- *Rd* must be the same as *Rm*
- You must not use the *cond* suffix

Condition flags

If S is specified, the MUL instruction:

- Updates the N and Z flags according to the result
- Does not affect the C and V flags

Examples

```
MULR10, R2, R5; multiply, R10 = R2 x R5
MLAR10, R2, R1, R5; multiply with accumulate, R10 = (R2 x R1) + R5
MULSR0, R2, R2; multiply with flag update, R0 = R2 x R2
MULLTR2, R3, R2; conditionally multiply, R2 = R3 x R2
MLSR4, R5, R6, R7; multiply with subtract, R4 = R7 - (R5 x R6)
```

3.6.2 UMULL, UMLAL, SMULL, and SMLAL

Signed and unsigned long multiply, with optional accumulate, using 32-bit operands and producing a 64-bit result.

Syntax

```
op{cond} RdLo, RdHi, Rn, Rm
```

where:

- 'op' is one of:
 UMULL: Unsigned long multiply
 UMLAL: Unsigned long multiply, with accumulate
 SMULL: Signed long multiply
 SMLAL: Signed long multiply, with accumulate
- 'cond' is an optional condition code (see [Conditional execution on page 57](#))
- 'RdHi, RdLo' are the destination registers. For UMLAL and SMLAL, they also hold the accumulating value.
- 'Rn, Rm' are registers holding the operands

Operation

The UMULL instruction interprets the values from *Rn* and *Rm* as unsigned integers. It multiplies these integers and places the least significant 32 bits of the result in *RdLo*, and the most significant 32 bits of the result in *RdHi*.

The UMLAL instruction interprets the values from *Rn* and *Rm* as unsigned integers. It multiplies these integers, adds the 64-bit result to the 64-bit unsigned integer contained in *RdHi* and *RdLo*, and writes the result back to *RdHi* and *RdLo*.

The SMULL instruction interprets the values from *Rn* and *Rm* as two's complement signed integers. It multiplies these integers and places the least significant 32 bits of the result in *RdLo*, and the most significant 32 bits of the result in *RdHi*.

The SMLAL instruction interprets the values from *Rn* and *Rm* as two's complement signed integers. It multiplies these integers, adds the 64-bit result to the 64-bit signed integer contained in *RdHi* and *RdLo*, and writes the result back to *RdHi* and *RdLo*.

Restrictions

In these instructions:

- Do not use either SP or PC
- *RdHi* and *RdLo* must be different registers

Condition flags

These instructions do not affect the condition code flags.

Examples

```
UMULLR0, R4, R5, R6; unsigned (R4,R0) = R5 x R6
SMLALR4, R5, R3, R8; signed (R5,R4) = (R5,R4) + R3 x R8
```

3.6.3 SDIV and UDIV

Signed divide and unsigned divide.

Syntax

```
SDIV{cond} {Rd,} Rn, Rm
```

```
UDIV{cond} {Rd,} Rn, Rm
```

where:

- '*cond*' is an optional condition code (see [Conditional execution on page 57](#))
- '*Rd*' is the destination register. If *Rd* is omitted, the destination register is *Rn*
- '*Rn*' is the register holding the value to be divided
- '*Rm*' is a register holding the divisor

Operation

SDIV performs a signed integer division of the value in *Rn* by the value in *Rm*.

UDIV performs an unsigned integer division of the value in *Rn* by the value in *Rm*.

For both instructions, if the value in *Rn* is not divisible by the value in *Rm*, the result is rounded towards zero.

Restrictions

Do not use either SP or PC.

Condition flags

These instructions do not change the flags.

Examples

```
SDIVR0, R2, R4; signed divide, R0 = R2/R4  
UDIVR8, R8, R1; unsigned divide, R8 = R8/R1
```

3.7 Saturating instructions

This section describes the saturating instructions, SSAT and USAT.

3.7.1 SSAT and USAT

Signed saturate and unsigned saturate to any bit position, with optional shift before saturating.

Syntax

```
op{cond} Rd, #n, Rm [, shift #s]
```

where:

- 'op' is one of the following:
 - SSAT: Saturates a signed value to a signed range
 - USAT: Saturates a signed value to an unsigned range
- 'cond' is an optional condition code (see [Conditional execution on page 57](#))
- 'Rd' is the destination register.
- 'n' specifies the bit position to saturate to:
 - n ranges from 1 to 32 for SSAT
 - n ranges from 0 to 31 for USAT
- 'Rm' is the register containing the value to saturate
- 'shift #s' is an optional shift applied to Rm before saturating. It must be one of the following:
 - ASR #s where s is in the range 1 to 31
 - LSL #s where s is in the range 0 to 31

Operation

These instructions saturate to a signed or unsigned n -bit value.

The SSAT instruction applies the specified shift, then saturates to the signed range: $-2^{n-1} \leq x \leq 2^{n-1}-1$.

The USAT instruction applies the specified shift, then saturates to the unsigned range: $0 \leq x \leq 2^n-1$.

For signed n -bit saturation using SSAT, this means that:

- If the value to be saturated is less than -2^{n-1} , the result returned is -2^{n-1}
- If the value to be saturated is greater than $2^{n-1}-1$, the result returned is $2^{n-1}-1$
- otherwise, the result returned is the same as the value to be saturated.

For unsigned n -bit saturation using USAT, this means that:

- If the value to be saturated is less than 0, the result returned is 0
- If the value to be saturated is greater than 2^n-1 , the result returned is 2^n-1
- Otherwise, the result returned is the same as the value to be saturated.

If the returned result is different from the value to be saturated, it is called *saturation*. If saturation occurs, the instruction sets the Q flag to 1 in the APSR. Otherwise, it leaves the Q flag unchanged. To clear the Q flag to 0, you must use the MSR instruction, see [MSR on page 101](#).

To read the state of the Q flag, use the MRS instruction (see [MRS on page 100](#)).

Restrictions

Do not use either SP or PC.

Condition flags

These instructions do not affect the condition code flags.

If saturation occurs, these instructions set the Q flag to 1.

Examples

```
SSATR7, #16, R7, LSL #4; logical shift left value in R7 by 4, then
; saturate it as a signed 16-bit value and
; write it back to R7
USATNER0, #7, R5; conditionally saturate value in R5 as an
; unsigned 7 bit value and write it to R0
```

3.8 Bitfield instructions

[Table 29](#) shows the instructions that operate on adjacent sets of bits in registers or bitfields.

Table 29. Packing and unpacking instructions

Mnemonic	Brief description	See
BFC	Bit field clear	BFC and BFI on page 89
BFI	Bit field insert	BFC and BFI on page 89
SBFX	Signed bit field extract	SBFX and UBFX on page 89
SXTB	Sign extend a byte	SXT and UXT on page 90
SXTH	Sign extend a halfword	SXT and UXT on page 90
UBFX	Unsigned bit field extract	SBFX and UBFX on page 89
UXTB	Zero extend a byte	SXT and UXT on page 90
UXTH	Zero extend a halfword	SXT and UXT on page 90

3.8.1 BFC and BFI

Bit Field Clear and Bit Field Insert.

Syntax

```
BFC{cond} Rd, #lsb, #width
```

```
BFI{cond} Rd, Rn, #lsb, #width
```

where:

- '*cond*' is an optional condition code, see [Conditional execution on page 57](#).
- '*Rd*' is the destination register.
- '*Rn*' is the source register.
- '*lsb*' is the position of the least significant bit of the bitfield. *lsb* must be in the range 0 to 31.
- '*width*' is the width of the bitfield and must be in the range 1 to 32-*lsb*.

Operation

BFC clears a bitfield in a register. It clears width bits in *Rd*, starting at the low bit position *lsb*. Other bits in *Rd* are unchanged.

BFI copies a bitfield into one register from another register. It replaces width bits in *Rd* starting at the low bit position *lsb*, with width bits from *Rn* starting at bit[0]. Other bits in *Rd* are unchanged.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not affect the flags.

Examples

```
BFC  R4, #8, #12    ; Clear bit 8 to bit 19 (12 bits) of R4 to 0
BFI  R9, R2, #8, #12 ; Replace bit 8 to bit 19 (12 bits) of R9 with
                    ; bit 0 to bit 11 from R2
```

3.8.2 SBFX and UBFX

Signed Bit Field Extract and Unsigned Bit Field Extract.

Syntax

```
SBFX{cond} Rd, Rn, #lsb, #width
```

```
UBFX{cond} Rd, Rn, #lsb, #width
```

where:

- ‘*cond*’ is an optional condition code, see [Conditional execution on page 57](#).
- ‘*Rd*’ is the destination register.
- ‘*Rn*’ is the source register.
- ‘*lsb*’ is the position of the least significant bit of the bitfield. *lsb* must be in the range 0 to 31.
- ‘*width*’ is the width of the bitfield and must be in the range 1 to 32-*lsb*.

Operation

SBFX extracts a bitfield from one register, sign extends it to 32 bits, and writes the result to the destination register.

UBFX extracts a bitfield from one register, zero extends it to 32 bits, and writes the result to the destination register.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not affect the flags.

Examples

```
SBFX R0, R1, #20, #4 ; Extract bit 20 to bit 23 (4 bits) from R1 and sign
                    ; extend to 32 bits and then write the result to R0.
UBFX R8, R11, #9, #10 ; Extract bit 9 to bit 18 (10 bits) from R11 and zero
                    ; extend to 32 bits and then write the result to R8
```

3.8.3 SXT and UXT

Sign extend and Zero extend.

Syntax

```
SXTextend{cond} {Rd}, Rm {, ROR #n}
```

```
UXTextend{cond} {Rd}, Rm {, ROR #n}
```

where:

- ‘extend’ is one of:
 - B: Extends an 8-bit value to a 32-bit value.
 - H: Extends a 16-bit value to a 32-bit value.
- ‘*cond*’ is an optional condition code, see [Conditional execution on page 57](#).
- ‘*Rd*’ is the destination register.
- ‘*Rm*’ is the register holding the value to extend.
- ROR #*n* is one of:
 - ROR #8: Value from *Rm* is rotated right 8 bits.
 - ROR #16: Value from *Rm* is rotated right 16 bits.
 - ROR #24: Value from *Rm* is rotated right 24 bits.
 - If ROR #*n* is omitted, no rotation is performed.

Operation

These instructions do the following:

1. Rotate the value from *Rm* right by 0, 8, 16 or 24 bits.
2. Extract bits from the resulting value:
 - SXTB extracts bits[7:0] and sign extends to 32 bits.
 - UXTB extracts bits[7:0] and zero extends to 32 bits.
 - SXTH extracts bits[15:0] and sign extends to 32 bits.
 - UXTH extracts bits[15:0] and zero extends to 32 bits.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not affect the flags.

Examples

```

SXTB  R4, R6, ROR #16 ; Rotate R6 right by 16 bits, then obtain the lower
                        ; halfword of the result and then sign extend to
                        ; 32 bits and write the result to R4.
UXTB  R3, R10          ; Extract lowest byte of the value in R10 and zero
                        ; extend it, and write the result to R3
  
```

3.8.4 Branch and control instructions

[Table 30](#) shows the branch and control instructions:

Table 30. Branch and control instructions

Mnemonic	Brief description	See
B	Branch	B, BL, BX, and BLX on page 92
BL	Branch with Link	B, BL, BX, and BLX on page 92
BLX	Branch indirect with Link	B, BL, BX, and BLX on page 92
BX	Branch indirect	B, BL, BX, and BLX on page 92
CBNZ	Compare and Branch if Non Zero	CBZ and CBNZ on page 93
CBZ	Compare and Branch if Non Zero	CBZ and CBNZ on page 93
IT	If-Then	IT on page 94
TBB	Table Branch Byte	TBB and TBH on page 96
TBH	Table Branch Halfword	TBB and TBH on page 96

3.8.5 B, BL, BX, and BLX

Branch instructions.

Syntax

`B{cond} label`

`BL{cond} label`

`BX{cond} Rm`

`BLX{cond} Rm`

where:

- 'B' is branch (immediate).
- 'BL' is branch with link (immediate).
- 'BX' is branch indirect (register).
- 'BLX' is branch indirect with link (register).
- '*cond*' is an optional condition code, see [Conditional execution on page 57](#).
- '*label*' is a PC-relative expression. See [PC-relative expressions on page 56](#).
- '*Rm*' is a register that indicates an address to branch to. Bit[0] of the value in *Rm* must be 1, but the address to branch to is created by changing bit[0] to 0.

Operation

All these instructions cause a branch to *label*, or to the address indicated in *Rm*. In addition:

- The BL and BLX instructions write the address of the next instruction to LR (the link register, R14).
- The BX and BLX instructions cause a UsageFault exception if bit[0] of *Rm* is 0.

`B cond label` is the only conditional instruction that can be either inside or outside an IT block. All other branch instructions must be conditional inside an IT block, and must be unconditional outside the IT block, see [IT on page 94](#).

[Table 31](#) shows the ranges for the various branch instructions.

Table 31. Branch ranges

Instruction	Branch range
B label	–16 MB to +16 MB
B <i>cond</i> label (outside IT block)	–1 MB to +1 MB
B <i>cond</i> label (inside IT block)	–16 MB to +16 MB
BL{ <i>cond</i> } label	–16 MB to +16 MB
BX{ <i>cond</i> } Rm	Any value in register
BLX{ <i>cond</i> } Rm	Any value in register

You might have to use the .W suffix to get the maximum branch range. See [Instruction width selection on page 59](#).

Restrictions

The restrictions are:

- Do not use PC in the BLX instruction
- For BX and BLX, bit[0] of *Rm* must be 1 for correct execution but a branch occurs to the target address created by changing bit[0] to 0
- When any of these instructions is inside an IT block, it must be the last instruction of the IT block.

Bcond is the only conditional instruction that is not required to be inside an IT block. However, it has a longer branch range when it is inside an IT block.

Condition flags

These instructions do not change the flags.

Examples

```

B      loopA ; Branch to loopA
BLE   ng    ; Conditionally branch to label ng
B.W   target ; Branch to target within 16MB range
BEQ   target ; Conditionally branch to target
BEQ.W target ; Conditionally branch to target within 1MB
BL    funC  ; Branch with link (Call) to function funC, return address
      ; stored in LR
BX    LR    ; Return from function call
BXNE  R0    ; Conditionally branch to address stored in R0
BLX   R0    ; Branch with link and exchange (Call) to a address stored
      ; in R0

```

3.8.6 CBZ and CBNZ

Compare and branch on zero, compare and branch on non-zero.

Syntax

```
CBZ Rn, label
```

```
CBNZ Rn, label
```

where:

- '*Rn*' is the register holding the operand.
- '*label*' is the branch destination.

Operation

Use the CBZ or CBNZ instructions to avoid changing the condition code flags and to reduce the number of instructions.

CBZ *Rn*, *label* does not change condition flags but is otherwise equivalent to:

```

CMP    Rn, #0
BEQ    label

```

CBNZ *Rn*, *label* does not change condition flags but is otherwise equivalent to:

```

CMP    Rn, #0
BNE    label

```

Restrictions

The restrictions are:

- *Rn* must be in the range of R0 to R7
- The branch destination must be within 4 to 130 bytes after the instruction
- These instructions must not be used inside an IT block.

Condition flags

These instructions do not change the flags.

Examples

```

CBZ    R5, target ; Forward branch if R5 is zero
CBNZ   R0, target ; Forward branch if R0 is not zero

```

3.8.7 IT

If-Then condition instruction.

Syntax

```
IT{x{y{z}}} cond
```

where:

- 'x' specifies the condition switch for the second instruction in the IT block.
- 'y' specifies the condition switch for the third instruction in the IT block.
- 'z' specifies the condition switch for the fourth instruction in the IT block.
- 'cond' specifies the condition for the first instruction in the IT block.

The condition switch for the second, third and fourth instruction in the IT block can be either:

T: Then. Applies the condition *cond* to the instruction.

E: Else. Applies the inverse condition of *cond* to the instruction.

- It is possible to use AL (the *always* condition) for *cond* in an IT instruction. If this is done, all of the instructions in the IT block must be unconditional, and each of *x*, *y*, and *z* must be T or omitted but not E.

Operation

The IT instruction makes up to four following instructions conditional. The conditions can be all the same, or some of them can be the logical inverse of the others. The conditional instructions following the IT instruction form the *IT block*.

The instructions in the IT block, including any branches, must specify the condition in the *{cond}* part of their syntax.

Your assembler might be able to generate the required IT instructions for conditional instructions automatically, so that you do not need to write them yourself. See your assembler documentation for details.

A BKPT instruction in an IT block is always executed, even if its condition fails.

Exceptions can be taken between an IT instruction and the corresponding IT block, or within an IT block. Such an exception results in entry to the appropriate exception handler, with suitable return information in LR and stacked PSR.

Instructions designed for use for exception returns can be used as normal to return from the exception, and execution of the IT block resumes correctly. This is the only way that a PC-modifying instruction is permitted to branch to an instruction in an IT block.

Restrictions

The following instructions are not permitted in an IT block:

- IT
- CBZ and CBNZ
- CPSID and CPSIE.

Other restrictions when using an IT block are:

- a branch or any instruction that modifies the PC must either be outside an IT block or must be the last instruction inside the IT block. These are:
 - ADD PC, PC, Rm
 - MOV PC, Rm
 - B, BL, BX, BLX
 - any LDM, LDR, or POP instruction that writes to the PC
 - TBB and TBH
- Do not branch to any instruction inside an IT block, except when returning from an exception handler
- All conditional instructions except *Bcond* must be inside an IT block. *Bcond* can be either outside or inside an IT block but has a larger branch range if it is inside one
- Each instruction inside the IT block must specify a condition code suffix that is either the same or logical inverse as for the other instructions in the block.

Your assembler might place extra restrictions on the use of IT blocks, such as prohibiting the use of assembler directives within them.

Condition flags

This instruction does not change the flags.

Example

```
ITTE    NE           ; Next 3 instructions are conditional
ANDNE   R0, R0, R1   ; ANDNE does not update condition flags
ADDSNE  R2, R2, #1   ; ADDSNE updates condition flags
MOVEQ   R2, R3       ; Conditional move
```

```

CMP    R0, #9      ; Convert R0 hex value (0 to 15) into ASCII
                    ; ('0'-'9', 'A'-'F')
ITE    GT          ; Next 2 instructions are conditional
ADDGT  R1, R0, #55 ; Convert 0xA -> 'A'
ADDLE  R1, R0, #48 ; Convert 0x0 -> '0'

IT     GT          ; IT block with only one conditional instruction
ADDGT  R1, R1, #1  ; Increment R1 conditionally

ITTEE  EQ          ; Next 4 instructions are conditional
MOVEQ  R0, R1      ; Conditional move
ADDEQ  R2, R2, #10 ; Conditional add
ANDNE  R3, R3, #1  ; Conditional AND
BNE.W  dloop       ; Branch instruction can only be used in the last
                    ; instruction of an IT block

IT     NE          ; Next instruction is conditional
ADD    R0, R0, R1  ; Syntax error: no condition code used in IT block

```

3.8.8 TBB and TBH

Table Branch Byte and Table Branch Halfword.

Syntax

TBB [*Rn*, *Rm*]

TBH [*Rn*, *Rm*, LSL #1]

where:

- '*Rn*' is the register containing the address of the table of branch lengths.
If *Rn* is PC, then the address of the table is the address of the byte immediately following the TBB or TBH instruction.
- '*Rm*' is the index register. This contains an index into the table. For halfword tables, LSL #1 doubles the value in *Rm* to form the right offset into the table.

Operation

These instructions cause a PC-relative forward branch using a table of single byte offsets for TBB, or halfword offsets for TBH. *Rn* provides a pointer to the table, and *Rm* supplies an index into the table. For TBB the branch offset is twice the unsigned value of the byte returned from the table. and for TBH the branch offset is twice the unsigned value of the halfword returned from the table. The branch occurs to the address at that offset from the address of the byte immediately after the TBB or TBH instruction.

Restrictions

The restrictions are:

- *Rn* must not be SP
- *Rm* must not be SP and must not be PC
- When any of these instructions is used inside an IT block, it must be the last instruction of the IT block.

Condition flags

These instructions do not change the flags.

Examples

```

ADR.W R0, BranchTable_Byte
TBB [R0, R1] ; R1 is the index, R0 is the base address of the
; branch table
Case1
; an instruction sequence follows
Case2
; an instruction sequence follows
Case3
; an instruction sequence follows
BranchTable_Byte
DCB 0 ; Case1 offset calculation
DCB ((Case2-Case1)/2) ; Case2 offset calculation
DCB ((Case3-Case1)/2) ; Case3 offset calculation

TBH [PC, R1, LSL #1] ; R1 is the index, PC is used as base of the
; branch table
BranchTable_H
DCI ((CaseA - BranchTable_H)/2) ; CaseA offset calculation
DCI ((CaseB - BranchTable_H)/2) ; CaseB offset calculation
DCI ((CaseC - BranchTable_H)/2) ; CaseC offset calculation

CaseA
; an instruction sequence follows
CaseB
; an instruction sequence follows
CaseC
; an instruction sequence follows

```

3.9 Miscellaneous instructions

[Table 32](#) shows the remaining Cortex-M3 instructions:

Table 32. Miscellaneous instructions

Mnemonic	Brief description	See
BKPT	Breakpoint	BKPT on page 98
CPSID	Change Processor State, Disable Interrupts	CPS on page 98
CPSIE	Change Processor State, Enable Interrupts	CPS on page 98
DMB	Data Memory Barrier	DMB on page 99
DSB	Data Synchronization Barrier	DSB on page 100
ISB	Instruction Synchronization Barrier	ISB on page 100
MRS	Move from special register to register	MRS on page 100
MSR	Move from register to special register	MSR on page 101

Table 32. Miscellaneous instructions (continued)

Mnemonic	Brief description	See
NOP	No Operation	NOP on page 102
SEV	Send Event	SEV on page 102
SVC	Supervisor Call	SVC on page 103
WFE	Wait For Event	WFE on page 103
WFI	Wait For Interrupt	WFI on page 104

3.9.1 BKPT

Breakpoint.

Syntax

```
BKPT #imm
```

where:

- '*imm*' is an expression evaluating to an integer in the range 0-255 (8-bit value).

Operation

The BKPT instruction causes the processor to enter Debug state. Debug tools can use this to investigate system state when the instruction at a particular address is reached.

imm is ignored by the processor. If required, a debugger can use it to store additional information about the breakpoint.

The BKPT instruction can be placed inside an IT block, but it executes unconditionally, unaffected by the condition specified by the IT instruction.

Condition flags

This instruction does not change the flags.

Examples

```
BKPT 0xAB ; Breakpoint with immediate value set to 0xAB (debugger can
           ; extract the immediate value by locating it using the PC)
```

3.9.2 CPS

Change Processor State.

Syntax

```
CPSeffect iflags
```

where:

- ‘*effect*’ is one of:
 - IE: Clears the special purpose register.
 - ID: Sets the special purpose register.
- ‘*iflags*’ is a sequence of one or more flags:
 - i: Set or clear PRIMASK.
 - f: Set or clear FAULTMASK.

Operation

CPS changes the PRIMASK and FAULTMASK special register values. See [Exception mask registers on page 19](#) for more information about these registers.

Restrictions

The restrictions are:

- Use CPS only from privileged software, it has no effect if used in unprivileged software
- CPS cannot be conditional and so must not be used inside an IT block.

Condition flags

This instruction does not change the condition flags.

Examples

```
CPSID i ; Disable interrupts and configurable fault handlers (set PRIMASK)
CPSID f ; Disable interrupts and all fault handlers (set FAULTMASK)
CPSIE i ; Enable interrupts and configurable fault handlers (clear PRIMASK)
CPSIE f ; Enable interrupts and fault handlers (clear FAULTMASK)
```

3.9.3 DMB

Data Memory Barrier.

Syntax

```
DMB{cond}
```

where:

- ‘*cond*’ is an optional condition code, see [Conditional execution on page 57](#).

Operation

DMB acts as a data memory barrier. It ensures that all explicit memory accesses that appear, in program order, before the DMB instruction are completed before any explicit memory accesses that appear, in program order, after the DMB instruction. DMB does not affect the ordering or execution of instructions that do not access memory.

Condition flags

This instruction does not change the flags.

Examples

```
DMB ; Data Memory Barrier
```

3.9.4 DSB

Data Synchronization Barrier.

Syntax

```
DSB{cond}
```

where:

- 'cond' is an optional condition code, see [Conditional execution on page 57](#).

Operation

DSB acts as a special data synchronization memory barrier. Instructions that come after the DSB, in program order, do not execute until the DSB instruction completes. The DSB instruction completes when all explicit memory accesses before it complete.

Condition flags

This instruction does not change the flags.

Examples

```
DSB ; Data Synchronisation Barrier
```

3.9.5 ISB

Instruction Synchronization Barrier.

Syntax

```
ISB{cond}
```

where:

- 'cond' is an optional condition code, see [Conditional execution on page 57](#).

Operation

ISB acts as an instruction synchronization barrier. It flushes the pipeline of the processor, so that all instructions following the ISB are fetched from cache or memory again, after the ISB instruction has been completed.

Condition flags

This instruction does not change the flags.

Examples

```
ISB ; Instruction Synchronisation Barrier
```

3.9.6 MRS

Move the contents of a special register to a general-purpose register.

Syntax

```
MRS{cond} Rd, spec_reg
```

where:

- ‘*cond*’ is an optional condition code, see [Conditional execution on page 57](#).
- ‘*Rd*’ is the destination register.
- ‘*spec_reg*’ can be any of: APSR, IPSR, EPSR, IEPSR, IAPSR, EAPSR, PSR, MSP, PSP, PRIMASK, BASEPRI, BASEPRI_MAX, FAULTMASK, or CONTROL.

Operation

Use MRS in combination with MSR as part of a read-modify-write sequence for updating a PSR, for example to clear the Q flag.

In process swap code, the programmers model state of the process being swapped out must be saved, including relevant PSR contents. Similarly, the state of the process being swapped in must also be restored. These operations use MRS in the state-saving instruction sequence and MSR in the state-restoring instruction sequence.

BASEPRI_MAX is an alias of BASEPRI when used with the MRS instruction.

See [MSR on page 101](#).

Restrictions

Rd must not be SP and must not be PC.

Condition flags

This instruction does not change the flags.

Examples

```
MRS R0, PRIMASK ; Read PRIMASK value and write it to R0
```

3.9.7 MSR

Move the contents of a general-purpose register into the specified special register.

Syntax

```
MSR{cond} spec_reg, Rn
```

where:

- ‘*cond*’ is an optional condition code, see [Conditional execution on page 57](#).
- ‘*Rn*’ is the source register.
- ‘*spec_reg*’ can be any of: APSR, IPSR, EPSR, IEPSR, IAPSR, EAPSR, PSR, MSP, PSP, PRIMASK, BASEPRI, BASEPRI_MAX, FAULTMASK, or CONTROL.

Operation

The register access operation in MSR depends on the privilege level. Unprivileged software can only access the APSR, see [Table 4: APSR bit definitions on page 17](#). Privileged software can access all special registers.

In unprivileged software writes to unallocated or execution state bits in the PSR are ignored.

When you write to BASEPRI_MAX, the instruction writes to BASEPRI only if either:

- *Rn* is non-zero and the current BASEPRI value is 0
- *Rn* is non-zero and less than the current BASEPRI value.

See [MRS on page 100](#).

Restrictions

Rn must not be SP and must not be PC.

Condition flags

This instruction updates the flags explicitly based on the value in *Rn*.

Examples

```
MSR CONTROL, R1 ; Read R1 value and write it to the CONTROL register
```

3.9.8 NOP

No Operation.

Syntax

```
NOP{cond}
```

where:

- '*cond*' is an optional condition code, see [Conditional execution on page 57](#).

Operation

NOP does nothing. NOP is not necessarily a time-consuming NOP. The processor might remove it from the pipeline before it reaches the execution stage.

Use NOP for padding, for example to place the following instruction on a 64-bit boundary.

Condition flags

This instruction does not change the flags.

Examples

```
NOP ; No operation
```

3.9.9 SEV

Send Event.

Syntax

```
SEV{cond}
```

where:

- '*cond*' is an optional condition code, see [Conditional execution on page 57](#).

Operation

SEV is a hint instruction that causes an event to be signaled to all processors within a multiprocessor system. It also sets the local event register to 1, see [Power management on page 42](#).

Condition flags

This instruction does not change the flags.

Examples

```
SEV ; Send Event
```

3.9.10 SVC

Supervisor Call.

Syntax

```
SVC{cond} #imm
```

where:

- '*cond*' is an optional condition code, see [Conditional execution on page 57](#).
- '*imm*' is an expression evaluating to an integer in the range 0-255 (8-bit value).

Operation

The SVC instruction causes the SVC exception.

imm is ignored by the processor. If required, it can be retrieved by the exception handler to determine what service is being requested.

Condition flags

This instruction does not change the flags.

Examples

```
SVC 0x32 ; Supervisor Call (SVC handler can extract the immediate value  
; by locating it via the stacked PC)
```

3.9.11 WFE

Wait For Event.

Syntax

```
WFE{cond}
```

where:

- '*cond*' is an optional condition code, see [Conditional execution on page 57](#).

Operation

WFE is a hint instruction.

If the event register is 0, WFE suspends execution until one of the following events occurs:

- An exception, unless masked by the exception mask registers or the current priority level
- An exception enters the Pending state, if SEVONPEND in the System Control Register is set
- A Debug Entry request, if Debug is enabled
- An event signaled by a peripheral or another processor in a multiprocessor system using the SEV instruction.

If the event register is 1, WFE clears it to 0 and returns immediately.

For more information see [Power management on page 42](#).

Condition flags

This instruction does not change the flags.

Examples

```
WFE ; Wait for event
```

3.9.12 WFI

Wait for Interrupt.

Syntax

```
WFI{cond}
```

where:

- 'cond' is an optional condition code, see [Conditional execution on page 57](#).

Operation

WFI is a hint instruction that suspends execution until one of the following events occurs:

- An exception
- A Debug Entry request, regardless of whether Debug is enabled.

Condition flags

This instruction does not change the flags.

Examples

```
WFI ; Wait for interrupt
```

4 Core peripherals

4.1 About the STM32 core peripherals

The address map of the *Private peripheral bus* (PPB) is:

Table 33. STM32 core peripheral register regions

Address	Core peripheral	Description
0xE000E010-0xE000E01F	System timer	Table 49 on page 154
0xE000E100-0xE000E4EF	Nested vectored interrupt controller	Table 44 on page 128
0xE000ED00-0xE000ED3F	System control block	Table 48 on page 149
0xE000ED90-0xE000ED93	Memory protection unit	Table 40 on page 117 ⁽¹⁾
0xE000EF00-0xE000EF03	Nested vectored interrupt controller	Table 44 on page 128

1. Software can read the MPU Type Register at 0xE000ED90 to test for the presence of a *memory protection unit* (MPU).

In register descriptions:

- The *required privilege* gives the privilege level required to access the register, as follows:
 - Privileged** Only privileged software can access the register.
 - Unprivileged** Both unprivileged and privileged software can access the register.

4.2 Memory protection unit (MPU)

This section describes the Memory protection unit (MPU) which is implemented in some STM32 microcontrollers. Refer to the corresponding device datasheet to see if the MPU is present in the STM32 type you are using.

The MPU divides the memory map into a number of regions, and defines the location, size, access permissions, and memory attributes of each region. It supports:

- Independent attribute settings for each region
- Overlapping regions
- Export of memory attributes to the system.

The memory attributes affect the behavior of memory accesses to the region. The Cortex-M3 MPU defines:

- Eight separate memory regions, 0-7
- A background region.

When memory regions overlap, a memory access is affected by the attributes of the region with the highest number. For example, the attributes for region 7 take precedence over the attributes of any region that overlaps region 7.

The background region has the same memory access attributes as the default memory map, but is accessible from privileged software only.