Cortex-M3/M4F Instruction Set

TECHNICAL USER'S MANUAL

UM-COREISM-7703

Copyright

Texas Instruments Incorporated
108 Wild Basin, Suite 350
Austin, TX 78746
http://www.ti.com/stellaris
http://www-k.ext.ti.com/sc/technical-support/product-information-centers.htm

Table of Contents

List of Figures

List of Tables

List of Examples

1 Introduction

Each of the following chapters describes a functional group of Cortex-M3/M4F instructions. Together they describe all the instructions supported by the Cortex-M3/M4F processor:

1.1 Instruction Set Summary

The processor implements a version of the Thumb instruction set. Table 1-1 on page 19 lists the supported instructions.

In Table 1-1 on page 19:

- 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. Figure 1-1 on page 26 shows the Cortex-M3/M4F instructions by category.

Table 1-1. Cortex-M3/M4F Instructions

M3	M4	M4F	Mnemonic	Operands	Brief Description	Flags	See Page
✓	✓	✓	ADC, ADCS	{Rd,} Rn, Op2	Add with carry	N,Z,C,V	55
✓	✓	✓	ADD, ADDS	{Rd,} Rn, Op2	Add	N,Z,C,V	55
✓	✓	✓	ADD, ADDW	{Rd,} Rn, #imm12	Add	N,Z,C,V	55
✓	✓	✓	ADR	Rd, label	Load PC-relative address	-	36
✓	✓	✓	AND, ANDS	{Rd,} Rn, Op2	Logical AND	N,Z,C	58
✓	✓	✓	ASR, ASRS	Rd, Rm, <Rs|#n>	Arithmetic shift right	N,Z,C	60
✓	✓	✓	B	label	Branch	-	200
✓	✓	✓	BFC	Rd, #lsb, #width	Bit field clear	-	157
✓	✓	✓	BFI	Rd, Rn, #lsb, #width	Bit field insert	-	157
✓	✓	✓	BIC, BICS	{Rd,} Rn, Op2	Bit clear	N,Z,C	55
✓	✓	✓	BKPT	#imm	Breakpoint	-	209
✓	✓	✓	BL	label	Branch with link	-	200
✓	✓	✓	BLX	Rm	Branch indirect with link	-	200

Table 1-1. Cortex-M3/M4F Instructions *(continued)*

M3	M4	M4F	Mnemonic	Operands	Brief Description	Flags	See Page
✓	✓	✓	BX	Rm	Branch indirect	-	200
✓	✓	✓	CBNZ	Rn, label	Compare and branch if non-zero	-	202
✓	✓	✓	CBZ	Rn, label	Compare and branch if zero	-	202
✓	✓	✓	CLREX	-	Clear exclusive	-	52
✓	✓	✓	CLZ	Rd, Rm	Count leading zeros	-	63
✓	✓	✓	CMN	Rn, Op2	Compare negative	N,Z,C,V	64
✓	✓	✓	CMP	Rn, Op2	Compare	N,Z,C,V	64
✓	✓	✓	CPSID	iflags	Change processor state, disable interrupts	-	210
✓	✓	✓	CPSIE	iflags	Change processor state, enable interrupts	-	210
✓	✓	✓	DMB	-	Data memory barrier	-	211
✓	✓	✓	DSB	-	Data synchronization barrier	-	211
✓	✓	✓	EOR, EORS	{Rd,} Rn, Op2	Exclusive OR	N,Z,C	55
✓	✓	✓	ISB	-	Instruction synchronization barrier	-	213
✓	✓	✓	IT	-	If-Then condition block	-	203
✓	✓	✓	LDM	Rn{!}, reglist	Load multiple registers, increment after	-	46
✓	✓	✓	LDMDB, LDMEA	Rn{!}, reglist	Load multiple registers, decrement before	-	46
✓	✓	✓	LDMFD, LDMIA	Rn{!}, reglist	Load multiple registers, increment after	-	46
✓	✓	✓	LDR	Rt, [Rn{, #offset}]	Load register with word	-	37
✓	✓	✓	LDRB, LDRBT	Rt, [Rn{, #offset}]	Load register with byte	-	37
✓	✓	✓	LDRD	Rt, Rt2, [Rn{, #offset}]	Load register with two words	-	37
✓	✓	✓	LDREX	Rt, [Rn, #offset]	Load register exclusive	-	50
✓	✓	✓	LDREXB	Rt, [Rn]	Load register exclusive with byte	-	50
✓	✓	✓	LDREXH	Rt, [Rn]	Load register exclusive with halfword	-	50
✓	✓	✓	LDRH, LDRHT	Rt, [Rn{, #offset}]	Load register with halfword	-	37
✓	✓	✓	LDRSB, LDRSBT	Rt, [Rn{, #offset}]	Load register with signed byte	-	37
✓	✓	✓	LDRSH, LDRSHT	Rt, [Rn{, #offset}]	Load register with signed halfword	-	37
✓	✓	✓	LDRT	Rt, [Rn{, #offset}]	Load register with word	-	42
✓	✓	✓	LSL, LSLS	Rd, Rm, <Rs|#n>	Logical shift left	N,Z,C	60
✓	✓	✓	LSR, LSRS	Rd, Rm, <Rs|#n>	Logical shift right	N,Z,C	60
✓	✓	✓	MLA	Rd, Rn, Rm, Ra	Multiply with accumulate, 32-bit result	-	106
✓	✓	✓	MLS	Rd, Rn, Rm, Ra	Multiply and subtract, 32-bit result	-	106
✓	✓	✓	MOV, MOVS	Rd, Op2	Move	N,Z,C	65
✓	✓	✓	MOV, MOVW	Rd, #imm16	Move 16-bit constant	N,Z,C	65
	✓	✓	MOVT	Rd, #imm16	Move top	-	67
✓	✓	✓	MRS	Rd, spec_reg	Move from special register to general register	-	214

Table 1-1. Cortex-M3/M4F Instructions *(continued)*

M3	M4	M4F	Mnemonic	Operands	Brief Description	Flags	See Page
✓	✓	✓	MSR	spec_reg, Rn	Move from general register to special register	N,Z,C,V	215
✓	✓	✓	MUL, MULS	{Rd,} Rn, Rm	Multiply, 32-bit result	N,Z	106
✓	✓	✓	MVN, MVNS	Rd, Op2	Move NOT	N,Z,C	65
✓	✓	✓	NOP	-	No operation	-	216
✓	✓	✓	ORN, ORNS	{Rd,} Rn, Op2	Logical OR NOT	N,Z,C	55
✓	✓	✓	ORR, ORRS	{Rd,} Rn, Op2	Logical OR	N,Z,C	55
✓	✓	✓	PKHTB, PKHBT	{Rd,} Rn, Rm, Op2	Pack halfword	-	148
✓	✓	✓	POP	reglist	Pop registers from stack	-	48
✓	✓	✓	PUSH	reglist	Push registers onto stack	-	48
	✓	✓	QADD	{Rd,} Rn, Rm	Saturating add	Q	136
	✓	✓	QADD16	{Rd,} Rn, Rm	Saturating add 16	-	136
	✓	✓	QADD8	{Rd,} Rn, Rm	Saturating add 8	-	136
	✓	✓	QASX	{Rd,} Rn, Rm	Saturating add and subtract with exchange	-	138
	✓	✓	QDADD	{Rd,} Rn, Rm	Saturating double and add	Q	140
	✓	✓	QDSUB	{Rd,} Rn, Rm	Saturating double and subtract	Q	140
	✓	✓	QSAX	{Rd,} Rn, Rm	Saturating subtract and add with exchange	-	138
	✓	✓	QSUB	{Rd,} Rn, Rm	Saturating subtract	Q	136
	✓	✓	QSUB16	{Rd,} Rn, Rm	Saturating subtract 16	-	136
	✓	✓	QSUB8	{Rd,} Rn, Rm	Saturating subtract 8	-	136
✓	✓	✓	RBIT	Rd, Rn	Reverse bits	-	68
✓	✓	✓	REV	Rd, Rn	Reverse byte order in a word	-	68
✓	✓	✓	REV16	Rd, Rn	Reverse byte order in each halfword	-	68
✓	✓	✓	REVSH	Rd, Rn	Reverse byte order in bottom halfword and sign extend	-	68
✓	✓	✓	ROR, RORS	Rd, Rm, <Rs|#n>	Rotate right	N,Z,C	60
✓	✓	✓	RRX, RRXS	Rd, Rm	Rotate right with extend	N,Z,C	60
✓	✓	✓	RSB, RSBS	{Rd,} Rn, Op2	Reverse subtract	N,Z,C,V	55
	✓	✓	SADD16	{Rd,} Rn, Rm	Signed add 16	GE	70
	✓	✓	SADD8	{Rd,} Rn, Rm	Signed add 8	GE	70
	✓	✓	SASX	{Rd,} Rn, Rm	Signed add and subtract with exchange	GE	70
✓	✓	✓	SBC, SBCS	{Rd,} Rn, Op2	Subtract with carry	N,Z,C,V	55
✓	✓	✓	SBFX	Rd, Rn, #lsb, #width	Signed bit field extract	-	158
✓	✓	✓	SDIV	{Rd,} Rn, Rm	Signed divide	-	130
	✓	✓	SEL	{Rd,} Rn, Rm	Select bytes	-	98
✓	✓	✓	SEV	-	Send event	-	217
	✓	✓	SHADD16	{Rd,} Rn, Rm	Signed halving add 16	-	72
	✓	✓	SHADD8	{Rd,} Rn, Rm	Signed halving add 8	-	72
	✓	✓	SHASX	{Rd,} Rn, Rm	Signed halving add and subtract with exchange	-	74

Table 1-1. Cortex-M3/M4F Instructions *(continued)*

M3	M4	M4F	Mnemonic	Operands	Brief Description	Flags	See Page
	✓	✓	SHSAX	{Rd,} Rn, Rm	Signed halving add and subtract with exchange	-	74
	✓	✓	SHSUB16	{Rd,} Rn, Rm	Signed halving subtract 16	-	77
	✓	✓	SHSUB8	{Rd,} Rn, Rm	Signed halving subtract 8	-	77
	✓	✓	SMLABB, SMLABT, SMLATB, SMLATT	Rd, Rn, Rm, Ra	Signed multiply accumulate long (halfwords)	Q	108
	✓	✓	SMLAD, SMLADX	Rd, Rn, Rm, Ra	Signed multiply accumulate dual	Q	111
✓	✓	✓	SMLAL	RdLo, RdHi, Rn, Rm	Signed multiply with accumulate (32x32+64), 64-bit result	-	128
✓	✓	✓	SMLALBB, SMLALBT, SMLALTB, SMLALTT	RdLo, RdHi, Rn, Rm	Signed multiply accumulate long (halfwords)	-	113
	✓	✓	SMLALD, SMLALDX	RdLo, RdHi, Rn, Rm	Signed multiply accumulate long dual	-	113
	✓	✓	SMLAWB,SMLAWT	Rd, Rn, Rm, Ra	Signed multiply accumulate, word by halfword	Q	108
	✓	✓	SMLSD SMLSDX	Rd, Rn, Rm, Ra	Signed multiply subtract dual	Q	116
	✓	✓	SMLSLD SMLSLDX	RdLo, RdHi, Rn, Rm	Signed multiply subtract long dual		116
	✓	✓	SMMLA	Rd, Rn, Rm, Ra	Signed most significant word multiply accumulate	-	119
	✓	✓	SMMLS, SMMLR	Rd, Rn, Rm, Ra	Signed most significant word multiply subtract	-	119
	✓	✓	SMMUL, SMMULR	{Rd,} Rn, Rm	Signed most significant word multiply	-	121
	✓	✓	SMUAD SMUADX	{Rd,} Rn, Rm	Signed dual multiply add	Q	123
	✓	✓	SMULBB, SMULBT, SMULTB, SMULTT	{Rd,} Rn, Rm	Signed multiply halfwords	-	125
✓	✓	✓	SMULL	RdLo, RdHi, Rn, Rm	Signed multiply (32x32), 64-bit result	-	128
	✓	✓	SMULWB, SMULWT	{Rd,} Rn, Rm	Signed multiply by halfword	-	125
	✓	✓	SMUSD, SMUSDX	{Rd,} Rn, Rm	Signed dual multiply subtract	-	123
✓	✓	✓	SSAT	Rd, #n, Rm {,shift #s}	Signed saturate	Q	132
	✓	✓	SSAT16	Rd, #n, Rm	Signed saturate 16	Q	134

Table 1-1. Cortex-M3/M4F Instructions *(continued)*

M3	M4	M4F	Mnemonic	Operands	Brief Description	Flags	See Page
	✓	✓	SSAX	{Rd,} Rn, Rm	Saturating subtract and add with exchange	GE	81
	✓	✓	SSUB16	{Rd,} Rn, Rm	Signed subtract 16	-	79
	✓	✓	SSUB8	{Rd,} Rn, Rm	Signed subtract 8	-	79
✓	✓	✓	STM	Rn{!}, reglist	Store multiple registers, increment after	-	46
✓	✓	✓	STMDB, STMEA	Rn{!}, reglist	Store multiple registers, decrement before	-	46
✓	✓	✓	STMFD, STMIA	Rn{!}, reglist	Store multiple registers, increment after	-	46
✓	✓	✓	STR	Rt, [Rn{, #offset}]	Store register word	-	37
✓	✓	✓	STRB, STRBT	Rt, [Rn{, #offset}]	Store register byte	-	37
✓	✓	✓	STRD	Rt, Rt2, [Rn{, #offset}]	Store register two words	-	37
✓	✓	✓	STREX	Rd, Rt, [Rn, #offset]	Store register exclusive	-	50
✓	✓	✓	STREXB	Rd, Rt, [Rn]	Store register exclusive byte	-	50
✓	✓	✓	STREXH	Rd, Rt, [Rn]	Store register exclusive halfword	-	50
✓	✓	✓	STRH, STRHT	Rt, [Rn{, #offset}]	Store register halfword	-	37
✓	✓	✓	STRSB, STRSBT	Rt, [Rn{, #offset}]	Store register signed byte	-	37
✓	✓	✓	STRSH, STRSHT	Rt, [Rn{, #offset}]	Store register signed halfword	-	37
✓	✓	✓	STRT	Rt, [Rn{, #offset}]	Store register word	-	42
✓	✓	✓	SUB, SUBS	{Rd,} Rn, Op2	Subtract	N,Z,C,V	55
✓	✓	✓	SUB, SUBW	{Rd,} Rn, #imm12	Subtract 12-bit constant	N,Z,C,V	55
✓	✓	✓	SVC	#imm	Supervisor call	-	218
	✓	✓	SXTAB	{Rd,} Rn, Rm, {,ROR #}	Extend 8 bits to 32 and add	-	154
	✓	✓	SXTAB16	{Rd,} Rn, Rm,{,ROR #}	Dual extend 8 bits to 16 and add	-	154
	✓	✓	SXTAH	{Rd,} Rn, Rm,{,ROR #}	Extend 16 bits to 32 and add	-	154
	✓	✓	SXTB16	{Rd,} Rm {,ROR #n}	Signed extend byte 16	-	150
✓	✓	✓	SXTB	{Rd,} Rm {,ROR #n}	Sign extend a byte	-	150
✓	✓	✓	SXTH	{Rd,} Rm {,ROR #n}	Sign extend a halfword	-	150
✓	✓	✓	TBB	[Rn, Rm]	Table branch byte	-	206
✓	✓	✓	TBH	[Rn, Rm, LSL #1]	Table branch halfword	-	206
✓	✓	✓	TEQ	Rn, Op2	Test equivalence	N,Z,C	84
✓	✓	✓	TST	Rn, Op2	Test	N,Z,C	84
	✓	✓	UADD16	{Rd,} Rn, Rm	Unsigned add 16	GE	86
	✓	✓	UADD8	{Rd,} Rn, Rm	Unsigned add 8	GE	86
	✓	✓	UASX	{Rd,} Rn, Rm	Unsigned add and subtract with exchange	GE	88
	✓	✓	UHADD16	{Rd,} Rn, Rm	Unsigned halving add 16	-	91
	✓	✓	UHADD8	{Rd,} Rn, Rm	Unsigned halving add 8	-	91
	✓	✓	UHASX	{Rd,} Rn, Rm	Unsigned halving add and subtract with exchange	-	93
	✓	✓	UHSAX	{Rd,} Rn, Rm	Unsigned halving subtract and add with exchange	-	93

Table 1-1. Cortex-M3/M4F Instructions *(continued)*

M3	M4	M4F	Mnemonic	Operands	Brief Description	Flags	See Page
	✓	✓	UHSUB16	{Rd,} Rn, Rm	Unsigned halving subtract 16	-	96
	✓	✓	UHSUB8	{Rd,} Rn, Rm	Unsigned halving subtract 8	-	96
✓	✓	✓	UBFX	Rd, Rn, #lsb, #width	Unsigned bit field extract	-	158
✓	✓	✓	UDIV	{Rd,} Rn, Rm	Unsigned divide	-	130
	✓	✓	UMAAL	RdLo, RdHi, Rn, Rm	Unsigned multiply accumulate accumulate long (32x32+64), 64-bit result	-	128
✓	✓	✓	UMLAL	RdLo, RdHi, Rn, Rm	Unsigned multiply with accumulate (32x32+64), 64-bit result	-	128
✓	✓	✓	UMULL	RdLo, RdHi, Rn, Rm	Unsigned multiply (32x32), 64-bit result	-	128
	✓	✓	UQADD16	{Rd,} Rn, Rm	Unsigned saturating add 16	-	144
	✓	✓	UQADD8	{Rd,} Rn, Rm	Unsigned saturating add 8	-	144
	✓	✓	UQASX	{Rd,} Rn, Rm	Unsigned saturating add and subtract with exchange	-	142
	✓	✓	UQSAX	{Rd,} Rn, Rm	Unsigned saturating subtract and add with exchange	-	142
	✓	✓	UQSUB16	{Rd,} Rn, Rm	Unsigned saturating subtract 16	-	144
	✓	✓	UQSUB8	{Rd,} Rn, Rm	Unsigned saturating subtract 8	-	144
	✓	✓	USAD8	{Rd,} Rn, Rm	Unsigned sum of absolute differences	-	99
	✓	✓	USADA8	{Rd,} Rn, Rm, Ra	Unsigned sum of absolute differences and accumulate	-	101
✓	✓	✓	USAT	Rd, #n, Rm {,shift #s}	Unsigned saturate	Q	132
	✓	✓	USAT16	Rd, #n, Rm	Unsigned saturate 16	Q	134
	✓	✓	USAX	{Rd,} Rn, Rm	Unsigned subtract and add with exchange	GE	93
	✓	✓	USUB16	{Rd,} Rn, Rm	Unsigned subtract 16	GE	96
	✓	✓	USUB8	{Rd,} Rn, Rm	Unsigned subtract 8	GE	96
	✓	✓	UXTAB	{Rd,} Rn, Rm, {,ROR #}	Rotate, extend 8 bits to 32 and add	-	154
	✓	✓	UXTAB16	{Rd,} Rn, Rm, {,ROR #}	Rotate, dual extend 8 bits to 16 and add	-	154
	✓	✓	UXTAH	{Rd,} Rn, Rm, {,ROR #}	Rotate, unsigned extend and add halfword	-	154
✓	✓	✓	UXTB	{Rd,} Rm {,ROR #n}	Zero extend a byte	-	150
	✓	✓	UXTB16	{Rd,} Rm, {,ROR #n}	Unsigned extend byte 16	-	150
✓	✓	✓	UXTH	{Rd,} Rm {,ROR #n}	Zero extend a halfword	-	150
		✓	VABS.F32	Sd, Sm	Floating-point absolute	-	161
		✓	VADD.F32	{Sd,} Sn, Sm	Floating-point add	-	162
		✓	VCMP.F32	Sd, <Sm | #0.0>	Compare two floating-point registers, or one floating-point register and zero	FPSCR	163
		✓	VCMPE.F32	Sd, <Sm | #0.0>	Compare two floating-point registers, or one floating-point register and zero with extend byte check	FPSCR	163

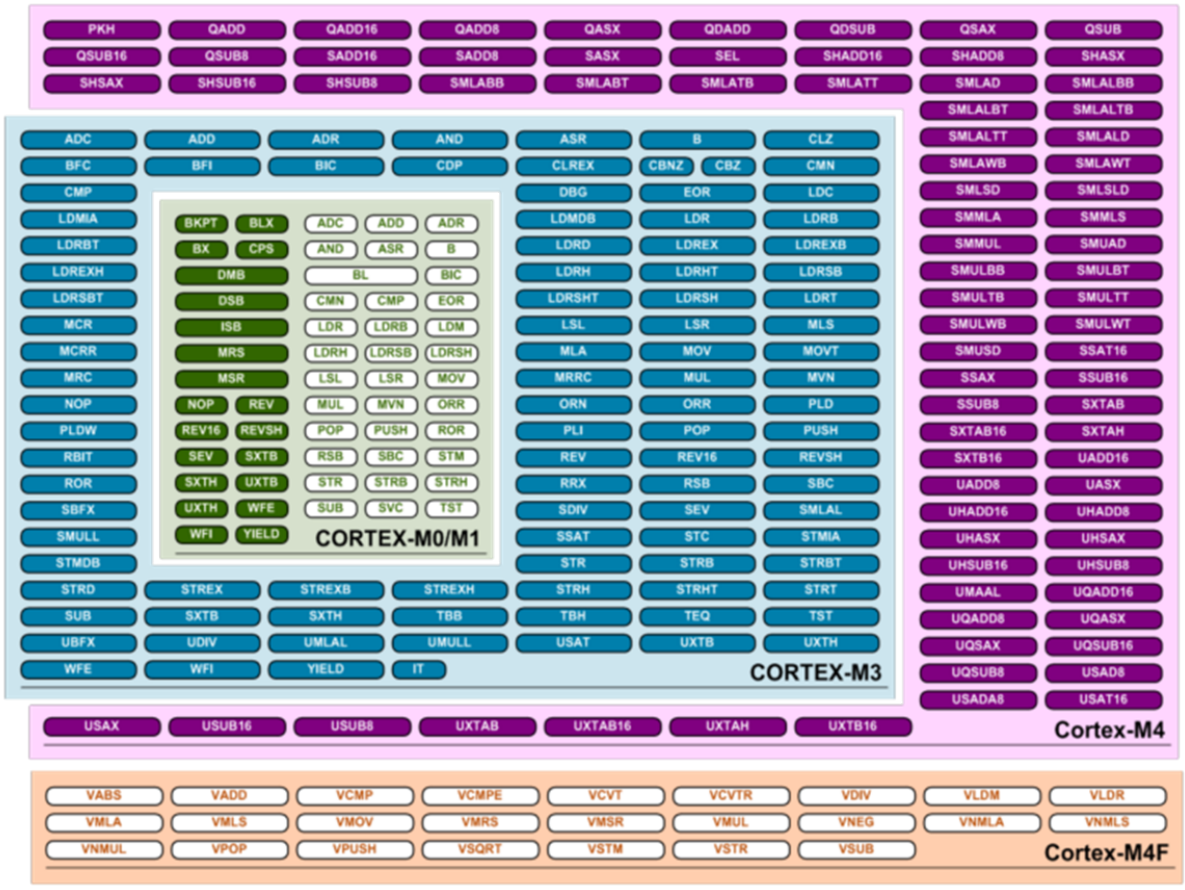
Table 1-1. Cortex-M3/M4F Instructions *(continued)*

M3	M4	M4F	Mnemonic	Operands	Brief Description	Flags	See Page
		✓	VCVT.S32.F32	Sd, Sm	Convert between floating-point and integer	-	167
		✓	VCVT.S16.F32	Sd, Sd, #fbits	Convert between floating-point and fixed point	-	167
		✓	VCVTR.S32.F32	Sd, Sm	Convert between floating-point and integer with rounding	-	167
		✓	VCVT<B|H>.F32.F16	Sd, Sm	Converts half-precision value to single-precision	-	169
		✓	VCVTT<B|T>.F32.F16	Sd, Sm	Converts single-precision register to half-precision	-	169
		✓	VDIV.F32	{Sd,} Sn, Sm	Floating-point divide	-	171
		✓	VFMA.F32	{Sd,} Sn, Sm	Floating-point fused multiply accumulate	-	172
		✓	VFNMA.F32	{Sd,} Sn, Sm	Floating-point fused negate multiply accumulate	-	173
		✓	VFMS.F32	{Sd,} Sn, Sm	Floating-point fused multiply subtract	-	172
		✓	VFNMS.F32	{Sd,} Sn, Sm	Floating-point fused negate multiply subtract	-	173
		✓	VLDM.F<32|64>	Rn{!}, list	Load multiple extension registers	-	174
		✓	VLDR.F<32|64>	<Dd|Sd>, [Rn]	Load an extension register from memory	-	176
		✓	VLMA.F32	{Sd,} Sn, Sm	Floating-point multiply accumulate	-	178
		✓	VLMS.F32	{Sd,} Sn, Sm	Floating-point multiply subtract	-	178
		✓	VMOV.F32	Sd, #imm	Floating-point move immediate	-	179
		✓	VMOV	Sd, Sm	Floating-point move register	-	180
		✓	VMOV	Sn, Rt	Copy ARM core register to single precision	-	182
		✓	VMOV	Sm, Sm1, Rt, Rt2	Copy 2 ARM core registers to 2 single precision	-	183
		✓	VMOV	Dd[x], Rt	Copy ARM core register to scalar	-	184
		✓	VMOV	Rt, Dn[x]	Copy scalar to ARM core register	-	181
		✓	VMRS	Rt, FPSCR	Move FPSCR to ARM core register or APSR	N,Z,C,V	185
		✓	VMSR	FPSCR, Rt	Move to FPSCR from ARM core register	FPSCR	186
		✓	VMUL.F32	{Sd,} Sn, Sm	Floating-point multiply	-	187
		✓	VNEG.F32	Sd, Sm	Floating-point negate	-	188
		✓	VNMLA.F32	{Sd,} Sn, Sm	Floating-point multiply and add	-	189
		✓	VNMLS.F32	{Sd,} Sn, Sm	Floating-point multiply and subtract	-	189
		✓	VNMUL	{Sd,} Sn, Sm	Floating-point multiply	-	189
		✓	VPOP	list	Pop extension registers	-	191
		✓	VPUSH	list	Push extension registers	-	192
		✓	VSQRT.F32	Sd, Sm	Calculates floating-point square root	-	193
		✓	VSTM	Rn{!}, list	Floating-point register store multiple	-	194

Table 1-1. Cortex-M3/M4F Instructions *(continued)*

M3	M4	M4F	Mnemonic	Operands	Brief Description	Flags	See Page
		✓	VSTR.F<32|64>	Sd, [Rn]	Stores an extension register to memory	-	194
		✓	VSUB.F<32|64>	{Sd,} Sn, Sm	Floating-point subtract	-	196
✓	✓	✓	WFE	-	Wait for event	-	219
✓	✓	✓	WFI	-	Wait for interrupt	-	220

Figure 1-1. Cortex-M Extensions



1.2 About the Instruction Descriptions

The following sections give more information about using the instructions:

1.2.1 Operands

An instruction operand can be an ARM Cortex-M3/M4F 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” on page 27.

See the *Stellaris® Data Sheet* for more information on the ARM Cortex-M3/M4F registers.

1.2.2 Restrictions When Using the 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 the instruction descriptions for more information.

Important: Bit[0] of any address you write 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/M4F processor only supports Thumb instructions.

1.2.3 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 or a register with optional shift.

1.2.3.1 Constant

You specify an *Operand2* constant in the form:

`#constant`

where `constant` can be (`X` and `Y` are hexadecimal digits):

- 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 `0xXYXYXYXY`.

Note: In the constants listed 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 `MOVS`, `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.

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*, #0xFFFFFFFE as the equivalent instruction `CMN` *Rd*, #0x2.

1.2.3.2 Register With Optional Shift

You specify an *Operand2* register 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 ≤ *n* ≤ 32.

LSL #*n*
Logical shift left *n* bits, 1 ≤ *n* ≤ 31.

LSR #*n*
Logical shift right *n* bits, 1 ≤ *n* ≤ 32.

ROR #*n*
Rotate right *n* bits, 1 ≤ *n* ≤ 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* remain 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” on page 28.

1.2.4 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`, and 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 27). 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 see “Flexible Second Operand” on page 27). 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.

1.2.4.1 ASR

An arithmetic shift right (ASR) 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 1-2 on page 29.

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

Figure 1-2. ASR #3

1.2.4.2 LSR

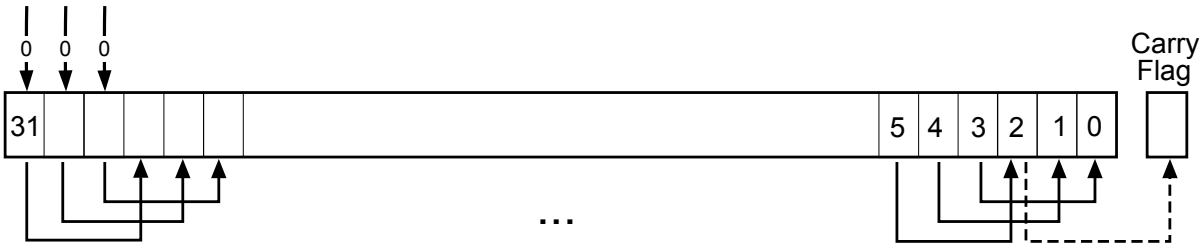
A logical shift right (LSR) by *n* bits moves the left-hand 32-*n* bits of the register *Rm*, to the right by *n* places, into the right-hand 32-*n* bits of the result. And it sets the left-hand *n* bits of the result to 0. See Figure 1-3 on page 30.

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

Figure 1-3. LSR #3



1.2.4.3 LSL

A logical shift left (LSL) 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 1-4 on page 30.

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

Figure 1-4. LSL #3

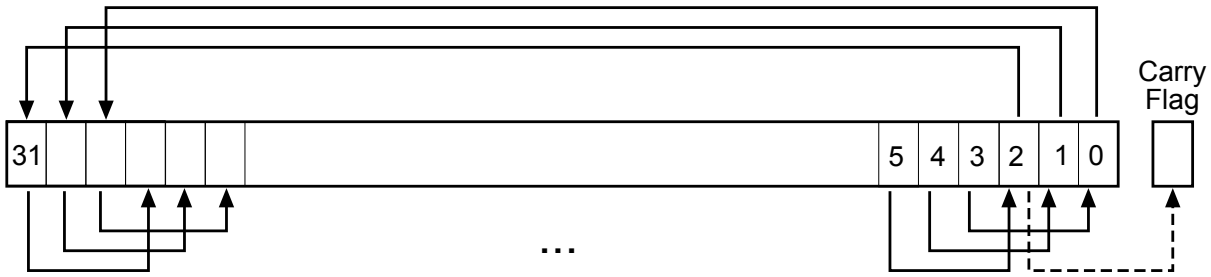
1.2.4.4 ROR

A rotate right (ROR) 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 moves the right-hand *n* bits of the register into the left-hand *n* bits of the result. See Figure 1-5 on page 31.

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

Figure 1-5. ROR #3

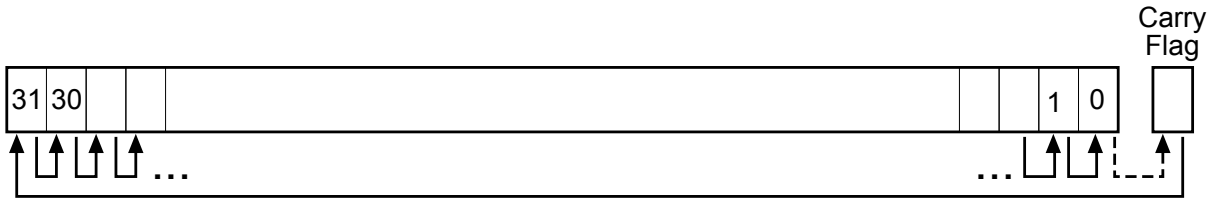


1.2.4.5 RRX

A rotate right with extend (RRX) 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 1-6 on page 31.

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 1-6. RRX



1.2.5 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/M4F processors support 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" in the *Stellaris® Data Sheet*.

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 UNALIGNED bit in the **Configuration and Control (CFGCTRL)** register to trap all unaligned accesses (see **CFGCTRL** in the *Stellaris® Data Sheet*).

1.2.6 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.

Note:
- For `B`, `BL`, `CBNZ`, and `CBZ` instructions, the value of the **PC** is the address of the current instruction plus 4 bytes.
- For all other instructions that use labels, the value of the **PC** is the address of the current instruction plus 4 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].

1.2.7 Conditional Execution

Most data processing instructions can optionally update the condition flags in the **Application Program Status Register (APSR)** register according to the result of the operation (see **APSR** in the *Stellaris® Data Sheet*). 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, either immediately after the instruction that updated the flags, or 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 1-2 on page 33 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 203 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. See “IT” on page 203 for more on the `IT` block.

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

1.2.7.1 Condition Flags

The **Application Program Status Register (APSR)** contains the following condition flags:

- **N.** Set to 1 when the result of the operation was negative; cleared to 0 otherwise.
- **Z.** Set to 1 when the result of the operation was zero; cleared to 0 otherwise.

- **C.** Set to 1 when the operation resulted in a carry; cleared to 0 otherwise.
- **V.** Set to 1 when the operation caused overflow; cleared to 0 otherwise.

For more information about **APSR**, see the *Stellaris® Data Sheet*.

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}.

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

1.2.7.2 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 **APSR** meet the specified condition. Table 1-2 on page 33 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 1-2 on page 33 also shows the relationship between condition code suffixes and the `N`, `Z`, `C`, and `V` flags.

Table 1-2. Condition Code Suffixes

Suffix	Flags	Meaning
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or HS	C = 1	Higher or same, unsigned ≥
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 ≤
GE	N = V	Greater than or equal, signed ≥
LT	N != V	Less than, signed <
GT	Z = 0 and N = V	Greater than, signed >
LE	Z = 1 and N != V	Less than or equal, signed ≤
AL	Can have any value	Always. This is the default when no suffix is specified.

Example 1-1, “Absolute Value” on page 34 shows the use of a conditional instruction to find the absolute value of a number. R0 = ABS(R1).

Example 1-1. Absolute Value

```
MOVS    R0, R1          ; R0 = R1, setting flags.
IT      MI              ; IT instruction for the negative condition.
RSBMI   R0, R1, #0      ; If negative, R0 = -R1.
```

Example 1-2, “Compare and Update Value” on page 34 shows the use of conditional instructions to update the value of R4 if the signed value R0 is greater than R1 and R2 is greater than R3.

Example 1-2. 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
```

1.2.8 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.

Note: 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. Example 1-3, “Instruction Width Selection” on page 34 shows instructions with the instruction width suffix.

Example 1-3. 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
```

2 Memory Access Instructions

Table 2-1 on page 35 shows the memory access instructions:

Table 2-1. Memory Access Instructions

Mnemonic	Brief Description	See Page
ADR	Load PC-relative address	36
CLREX	Clear exclusive	52
LDM{mode}	Load multiple registers	46
LDR{type}	Load register using immediate offset	37
LDR{type}	Load register using register offset	40
LDR{type}T	Load register with unprivileged access	42
LDR{type}	Load register using PC-relative address	44
LDRD	Load register using PC-relative address (two words)	44
LDREX{type}	Load register exclusive	50
POP	Pop registers from stack	48
PUSH	Push registers onto stack	48
STM{mode}	Store multiple registers	46
STR{type}	Store register using immediate offset	37
STR{type}	Store register using register offset	40
STR{type}T	Store register with unprivileged access	42
STREX{type}	Store register exclusive	50

2.1 ADR

Generate PC-relative address.

Applies to...	M3	M4	M4F
	✓	✓	✓

2.1.1 Syntax

ADR{*cond*} *Rd*, *label*

where:

cond
Is an optional condition code. See Table 1-2 on page 33.

Rd
Is the destination register.

label
Is a PC-relative expression. See “PC-Relative Expressions” on page 32.

2.1.2 Operation

ADR determines the address by adding an immediate value to the PC, and 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 of −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 34.

2.1.3 Restrictions

Rd must not be **SP** and must not be **PC**.

2.1.4 Condition Flags

This instruction does not change the flags.

2.1.5 Examples

```
ADR     R1, TextMessage   ; Write address value of a location labeled as
                          ; TextMessage to R1.
```

2.2 LDR and STR (Immediate Offset)

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

Applies to...	M3	M4	M4F
	✓	✓	✓

2.2.1 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 one of:

LDR
Load Register.

STR
Store Register.

type
Is one of:

B
Unsigned byte, zero extend to 32 bits on loads.

SB
Signed byte, sign extend to 32 bits (LDR only).

H
Unsigned halfword, zero extend to 32 bits on loads.

SH
Signed halfword, sign extend to 32 bits (LDR only).

-
Omit, for word.

cond
Is an optional condition code. See Table 1-2 on page 33.

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.

2.2.2 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 31.

Table 2-2 on page 38 shows the ranges of offset for immediate, pre-indexed and post-indexed forms.

Table 2-2. 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

2.2.3 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.

2.2.4 Condition Flags

These instructions do not change the flags.

2.2.5 Examples

```
LDR     R8, [R10]              ; Loads R8 from the address in R10.
LDRNE   R2, [R5, #960]!        ; Loads (conditionally) R2 from a word
                               ; 960 bytes above the address in R5, and
                               ; increments R5 by 960.
STR     R2, [R9,#const-struc]  ; const-struc is an expression evaluating
                               ; to a constant in the range 0-4095.
STRH    R3, [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.
STRD    R0, 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.
```

2.3 LDR and STR (Register Offset)

Load and Store with register offset.

Applies to...	M3	M4	M4F
	✓	✓	✓

2.3.1 Syntax

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

where:

op
Is one of:

LDR
Load Register.

STR
Store Register.

type
Is one of:

B
Unsigned byte, zero extend to 32 bits on loads.

SB
Signed byte, sign extend to 32 bits (LDR only).

H
Unsigned halfword, zero extend to 32 bits on loads.

SH
Signed halfword, sign extend to 32 bits (LDR only).

-
Omit, for word.

cond
Is an optional condition code. See Table 1-2 on page 33.

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.

2.3.2 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 31.

2.3.3 Restrictions

In these instructions:

- *Rn* must not be **PC**.
- *Rm* must not be **SP** and must not be **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.

2.3.4 Condition Flags

These instructions do not change the flags.

2.3.5 Examples

```
STR    R0, [R5, R1]         ; Store value of R0 into an address equal to
                            ; sum of R5 and R1.
LDRSB  R0, [R5, R1, LSL #1] ; Read byte value from an address equal to
                            ; sum of R5 and two times R1, sign extend it
                            ; to a word value and put it in R0.
STR    R0, [R1, R2, LSL #2] ; Store R0 to an address equal to sum of R1
                            ; and four times R2.
```

2.4 LDR and STR (Unprivileged Access)

Load and Store with unprivileged access.

Applies to...	M3	M4	M4F
	✓	✓	✓

2.4.1 Syntax

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

where:

op
Is one of:

LDR
Load Register.

STR
Store Register.

type
Is one of:

B
Unsigned byte, zero extend to 32 bits on loads.

SB
Signed byte, sign extend to 32 bits (LDR only).

H
Unsigned halfword, zero extend to 32 bits on loads.

SH
Signed halfword, sign extend to 32 bits (LDR only).

-
Omit, for word.

cond
Is an optional condition code. See Table 1-2 on page 33.

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*.

2.4.2 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 37). 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.

2.4.3 Restrictions

In these instructions:

- *Rn* must not be **PC**.
- *Rt* must not be **SP** and must not be **PC**.

2.4.4 Condition Flags

These instructions do not change the flags.

2.4.5 Examples

```
STRBTEQ  R4, [R7]       ; Conditionally store least significant byte in
                        ; R4 to an address in R7, with unprivileged access.
LDRHT    R2, [R2, #8]   ; Load halfword value from an address equal to
                        ; sum of R2 and 8 into R2, with unprivileged access.
```

2.5 LDR (PC-Relative)

Load register from memory.

Applies to...	M3	M4	M4F
	✓	✓	✓

2.5.1 Syntax

```
LDR{type}{cond} Rt, label

LDRD{cond} Rt, Rt2, label          ; Load two words
```

where:

type
Is one of:

B
Unsigned byte, zero extend to 32 bits.

SB
Signed byte, sign extend to 32 bits.

H
Unsigned halfword, zero extend to 32 bits.

SH
Signed halfword, sign extend to 32 bits.

-
Omit, for word.

cond
Is an optional condition code. See Table 1-2 on page 33.

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 32.

2.5.2 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 31.

label must be within a limited range of the current instruction. Table 2-3 on page 45 shows the possible offsets between *label* and the PC.

Table 2-3. Offset Ranges

Instruction Type	Offset Range[a]
Word, halfword, signed halfword, byte, signed byte	−4095 to 4095
Two words	−1020 to 1020

a. You might have to use the `.W` suffix to get the maximum offset range. See “Instruction Width Selection” on page 34.

2.5.3 Restrictions

In these instructions:

- *Rt* can be **SP** or **PC** only for word loads.
- *Rt2* must not be **SP** and must not be **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.

2.5.4 Condition Flags

These instructions do not change the flags.

2.5.5 Examples

```
LDR     R0, LookUpTable   ; Load R0 with a word of data from an address
                          ; labeled as LookUpTable.
LDRSB   R7, localdata     ; Load a byte value from an address labeled
                          ; as localdata, sign extend it to a word
                          ; value, and put it in R7.
```

2.6 LDM and STM

Load and Store Multiple registers.

Applies to...	M3	M4	M4F
	✓	✓	✓

2.6.1 Syntax

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

where:

op
Is one of:

LDM
Load Multiple registers.

STM
Store Multiple registers.

addr_mode
Is any one 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 Table 1-2 on page 33.

Rn
Is the register on which the memory addresses are based.

!
Is an optional writeback suffix. If ! is present then 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 47.

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 s synonym for STMDB, and refers to its use for pushing data onto Full Descending stacks

2.6.2 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 of *Rn* - 4 * (*n*-1) is written back to *Rn*.

The PUSH and POP instructions can be expressed in this form. See “PUSH and POP” on page 48 for details.

2.6.3 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.

2.6.4 Condition Flags

These instructions do not change the flags.

2.6.5 Examples

```
LDM     R8,{R0,R2,R9}      ; LDMIA is a synonym for LDM.
STMDB   R1!,{R3-R6,R11,R12}
```

2.6.6 Incorrect Examples

```
STM     R5!,{R5,R4,R9} ; Value stored for R5 is unpredictable.
LDM     R2, {}         ; There must be at least one register in the list.
```

2.7 PUSH and POP

Push registers on and pop registers off a full-descending stack.

Applies to...	M3	M4	M4F
	✓	✓	✓

2.7.1 Syntax

`PUSH{`*`cond`*`} `*`reglist`*

`POP{`*`cond`*`} `*`reglist`*

where:

cond
Is an optional condition code. See Table 1-2 on page 33.

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.

`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.

2.7.2 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 46 for more information.

2.7.3 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.

2.7.4 Condition Flags

These instructions do not change the flags.

2.7.5 Examples

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

2.8 LDREX and STREX

Load and Store Register Exclusive.

Applies to...	M3	M4	M4F
	✓	✓	✓

2.8.1 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 Table 1-2 on page 33.

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*.

2.8.2 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" in the *Stellaris® Data Sheet*).

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.

Important: 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.

2.8.3 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.

2.8.4 Condition Flags

These instructions do not change the flags.

2.8.5 Examples

```
MOV     R1, #0x1            ; Initialize the 'lock taken' value.
try
LDREX   R0, [LockAddr]      ; Load the lock value.
CMP     R0, #0              ; Is the lock free?
ITT     EQ                  ; IT instruction for STREXEQ and CMPEQ.
STREXEQ R0, R1, [LockAddr]  ; Try and claim the lock.
CMPEQ   R0, #0              ; Did this succeed?
BNE     try                 ; No – try again.
....                        ; Yes – we have the lock.
```

2.9 CLREX

Clear Exclusive.

Applies to...	M3	M4	M4F
	✓	✓	✓

2.9.1 Syntax

CLREX{*cond*}

where:

cond
Is an optional condition code. See Table 1-2 on page 33.

2.9.2 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" in the *Stellaris® Data Sheet*).

2.9.3 Condition Flags

This instruction does not change the flags.

2.9.4 Examples

```
CLREX
```

3 General Data Processing Instructions

Table 3-1 on page 53 shows the data processing instructions:

Table 3-1. General Data Processing Instructions

Mnemonic	Brief Description	See Page
ADC	Add with carry	55
ADD	Add	55
ADDW	Add	55
AND	Logical AND	58
ASR	Arithmetic shift right	60
BIC	Bit clear	58
CLZ	Count leading zeros	63
CMN	Compare negative	64
CMP	Compare	64
EOR	Exclusive OR	58
LSL	Logical shift left	60
LSR	Logical shift right	60
MOV	Move	65
MOVT	Move top	67
MOVW	Move 16-bit constant	65
MVN	Move NOT	65
ORN	Logical OR NOT	58
ORR	Logical OR	58
RBIT	Reverse bits	68
REV	Reverse byte order in a word	68
REV16	Reverse byte order in each halfword	68
REVSH	Reverse byte order in bottom halfword and sign extend	68
ROR	Rotate right	60
RRX	Rotate right with extend	60
RSB	Reverse subtract	55
SADD16	Signed add 16	70
SADD8	Signed add 8	70
SASX	Signed add and subtract with exchange	81
SSAX	Signed subtract and add with exchange	81
SBC	Subtract with carry	55
SEL	Select bytes	98
SHADD16	Signed halving add 16	72
SHADD8	Signed halving add 8	72
SHASX	Signed halving add and subtract with exchange	74
SHSAX	Signed halving subtract and add with exchange	74
SHSUB16	Signed halving subtract 16	77
SHSUB8	Signed halving subtract 8	77

Table 3-1. General Data Processing Instructions *(continued)*

Mnemonic	Brief Description	See Page
SSUB16	Signed subtract 16	79
SSUB8	Signed subtract 8	79
SUB	Subtract	55
SUBW	Subtract	55
TEQ	Test equivalence	84
TST	Test	84
UADD16	Unsigned add 16	86
UADD8	Unsigned add 8	86
UASX	Unsigned add and subtract with exchange	88
USAX	Unsigned subtract and add with exchange	88
UHADD16	Unsigned halving add 16	91
UHADD8	Unsigned halving add 8	91
UHASX	Unsigned halving add and subtract with exchange	93
UHSAX	Unsigned halving subtract and add with exchange	93
UHSUB16	Unsigned halving subtract 16	96
UHSUB8	Unsigned halving subtract 8	96
USAD8	Unsigned sum of absolute differences	99
USADA8	Unsigned sum of absolute differences and accumulate	101
USUB16	Unsigned subtract 16	103
USUB8	Unsigned subtract 8	103

3.1 ADD, ADC, SUB, SBC, and RSB

Add, Add with carry, Subtract, Subtract with carry, and Reverse Subtract.

Applies to...	M3	M4	M4F
	✓	✓	✓

3.1.1 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 32.

cond
Is an optional condition code. See Table 1-2 on page 33.

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 27 for details of the options.

imm12
Is any value in the range 0-4095.

3.1.2 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 57.

See also 36.

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

3.1.3 Restrictions

In these instructions:

- *Operand2* must not be **SP** and must not be **PC**.
- *Rd* can be **SP** only in ADD and SUB, and only with the additional restrictions:
  - *Rn* must also be **SP**.
  - any shift in *Operand2* must be limited to a maximum of 3 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 not be **PC** and must not be **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 additional restrictions:
  - You must not specify the S suffix.
  - The second operand must be a constant in the range 0 to 4095.

  Note:
  - 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.
  - 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.

3.1.4 Condition Flags

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

3.1.5 Examples

```
ADD     R2, R1, R3
SUBS    R8, R6, #240        ; Sets the flags on the result.
RSB     R4, R4, #1280       ; Subtracts contents of R4 from 1280.
ADCHI   R11, R0, R3         ; Only executed if C flag set and Z
                            ; flag clear.
```

3.1.6 Multiword Arithmetic Examples

Example 3-1, “64-Bit Addition” on page 57 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.

Example 3-1. 64-Bit Addition

```
ADDS    R4, R0, R2    ; Add the least significant words.
ADC     R5, R1, R3    ; Add the most significant words with carry.
```

Multiword values do not have to use consecutive registers. Example 3-2, “96-Bit Subtraction” on page 57 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.

Example 3-2. 96-Bit Subtraction

```
SUBS    R6, R6, R9     ; Subtract the least significant words.
SBCS    R9, R2, R1     ; Subtract the middle words with carry.
SBC     R2, R8, R11    ; Subtract the most significant words with carry.
```

3.2 AND, ORR, EOR, BIC, and ORN

Logical AND, OR, Exclusive OR, Bit Clear, and OR NOT.

Applies to...	M3	M4	M4F
	✓	✓	✓

3.2.1 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 32.

cond
Is an optional condition code. See Table 1-2 on page 33.

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 27 for details of the options.

3.2.2 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*.

3.2.3 Restrictions

Do not use **SP** and do not use **PC**.

3.2.4 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 27.
- Do not affect the V flag.

3.2.5 Examples

```
AND     R9, R2, #0xFF00
ORREQ   R2, R0, R5
ANDS    R9, R8, #0x19
EORS    R7, R11, #0x18181818
BIC     R0, R1, #0xab
ORN     R7, R11, R14, ROR #4
ORNS    R7, R11, R14, ASR #32
```

3.3 ASR, LSL, LSR, ROR, and RRX

Arithmetic Shift Right, Logical Shift Left, Logical Shift Right, Rotate Right, and Rotate Right with Extend.

Applies to...	M3	M4	M4F
	✓	✓	✓

3.3.1 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 32.

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 in *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 length depends on the instruction:

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`.

3.3.2 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 28.

3.3.3 Restrictions

Do not use **SP** and do not use **PC**.

3.3.4 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 28.

3.3.5 Examples

```
ASR    R7, R8, #9  ; Arithmetic shift right by 9 bits.
LSLS   R1, R2, #3  ; Logical shift left by 3 bits with flag update.
LSR    R4, R5, #6  ; Logical shift right by 6 bits.
ROR    R4, R5, R6  ; Rotate right by the value in the bottom byte of R6.
RRX    R4, R5      ; Rotate right with extend.
```

3.4 CLZ

Count Leading Zeros.

Applies to...	M3	M4	M4F
	✓	✓	✓

3.4.1 Syntax

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

where:

cond
Is an optional condition code. See Table 1-2 on page 33.

Rd
Is the destination register.

Rm
Is the operand register.

3.4.2 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.

3.4.3 Restrictions

Do not use **SP** and do not use **PC**.

3.4.4 Condition Flags

This instruction does not change the flags.

3.4.5 Examples

```
CLZ     R4,R9
CLZNE   R2,R3
```

3.5 CMP and CMN

Compare and Compare Negative.

Applies to...	M3	M4	M4F
	✓	✓	✓

3.5.1 Syntax

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

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

where:

cond
Is an optional condition code. See Table 1-2 on page 33.

Rn
Is the register holding the first operand.

Operand2
Is a flexible second operand. See “Flexible Second Operand” on page 27 for details of the options.

3.5.2 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.

3.5.3 Restrictions

In these instructions:

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

3.5.4 Condition Flags

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

3.5.5 Examples

```
CMP     R2, R9
CMN     R0, #6400
CMPGT   SP, R7, LSL #2
```

3.6 MOV and MVN

Move and Move NOT.

Applies to...	M3	M4	M4F
	✓	✓	✓

3.6.1 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 32.

cond
: Is an optional condition code. See Table 1-2 on page 33.

Rd
: Is the destination register.

Operand2
: Is a flexible second operand. See “Flexible Second Operand” on page 27 for details of the options.

imm16
: Is any value in the range 0-65535.

3.6.2 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:

MOV Instruction	Preferred Syntax using Shift Instruction
`MOV{S}{cond} Rd, Rm, ASR #n`	`ASR{S}{cond} Rd, Rm, #n`
`MOV{S}{cond} Rd, Rm, LSL #n` (if *n* != 0)	`LSL{S}{cond} Rd, Rm, #n`
`MOV{S}{cond} Rd, Rm, LSR #n`	`LSR{S}{cond} Rd, Rm, #n`
`MOV{S}{cond} Rd, Rm, ROR #n`	`ROR{S}{cond} Rd, Rm, #n`
`MOV{S}{cond} Rd, Rm, RRX`	`RRX{S}{cond} Rd, Rm`

Also, the `MOV` instruction permits additional forms of *Operand2* as synonyms for shift instructions. See “ASR, LSL, LSR, ROR, and RRX” on page 60.

Shift Instruction	MOV Instruction Synonym
`ASR{S}{cond} Rd, Rm, Rs`	`MOV{S}{cond} Rd, Rm, ASR Rs`

Shift Instruction	MOV Instruction Synonym
LSL{S}{cond} Rd, Rm, Rs	MOV{S}{cond} Rd, Rm, LSL Rs
LSR{S}{cond} Rd, Rm, Rs	MOV{S}{cond} Rd, Rm, LSR Rs
ROR{S}{cond} Rd, Rm, Rs	MOV{S}{cond} Rd, Rm, ROR Rs

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.

3.6.3 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, Texas Instruments strongly recommends the use of a BX or BLX instruction to branch for software portability to the ARM Cortex-M3/M4F instruction set.

3.6.4 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 27.
- Do not affect the V flag.

3.6.5 Example

```
MOVS  R11, #0x000B    ; Write value of 0x000B to R11, flags get updated.
MOV   R1, #0xFA05     ; Write value of 0xFA05 to R1, flags are not updated.
MOVS  R10, R12        ; Write value in R12 to R10, flags get updated.
MOV   R3, #23         ; Write value of 23 to R3.
MOV   R8, SP          ; Write value of stack pointer to R8.
MVNS  R2, #0xF        ; Write value of 0xFFFFFFF0 (bitwise inverse of 0xF)
                      ; to R2 and update flags.
```

3.7 MOVT

Move Top.

Applies to...	M3	M4	M4F
		✓	✓

3.7.1 Syntax

`MOVT{cond} Rd, #imm16`

where:

cond
Is an optional condition code. See Table 1-2 on page 33.

Rd
Is the destination register.

imm16
Is a 16-bit immediate constant.

3.7.2 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.

3.7.3 Restrictions

Rd must not be **SP** and must not be **PC**.

3.7.4 Condition Flags

This instruction does not change the flags.

3.7.5 Examples

```
MOVT   R3, #0xF123 ; Write 0xF123 to upper halfword of R3, lower halfword
                   ; and APSR are unchanged.
```

3.8 REV, REV16, REVSH, and RBIT

Reverse bytes and Reverse bits.

Applies to...	M3	M4	M4F
	✓	✓	✓

3.8.1 Syntax

`op{cond} Rd, Rn`

where:

op
Is any 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 extend to 32 bits.

`RBIT`
Reverse the bit order in a 32-bit word.

cond
Is an optional condition code. See Table 1-2 on page 33.

Rd
Is the destination register.

Rn
Is the register holding the operand.

3.8.2 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.

3.8.3 Restrictions

Do not use **SP** and do not use **PC**.

3.8.4 Condition Flags

These instructions do not change the flags.

3.8.5 Examples

```
REV    R3, 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.
```

3.9 SADD16 and SADD8

Signed Add 16 and Signed Add 8.

Applies to...	M3	M4	M4F
		✓	✓

3.9.1 Syntax

op{*cond*}{*Rd*,} *Rn*, *Rm*

where:

op
is any of:

SADD16
Performs two 16-bit signed integer additions.

SADD8
Performs four 8-bit signed integer additions.

cond
is an optional condition code, see 33.

Rd
is the destination register.

Rn
is the first register holding the operand.

Rm
is the second register holding the operand.

3.9.2 Operation

Use these instructions to perform a halfword or byte add in parallel:

The SADD16 instruction:

1. Adds each halfword from the first operand to the corresponding halfword of the second operand.
2. Writes the result in the corresponding halfwords of the destination register.

The SADD8 instruction:

1. Adds each byte of the first operand to the corresponding byte of the second operand.
2. Writes the result in the corresponding bytes of the destination register.

3.9.3 Restrictions

Do not use SP and do not use PC.

3.9.4 Condition flags

These instructions do not change the flags.

3.9.5 Examples

```
SADD16 R1, R0      ; Adds the halfwords in R0 to the corresponding halfwords of
                   ; R1 and writes to corresponding halfword of R1.

SADD8  R4, R0, R5  ; Adds bytes of R0 to the corresponding byte in R5 and writes

                   ; to the corresponding byte in R4.
```

3.10 SHADD16 and SHADD8

Signed Halving Add 16 and Signed Halving Add 8.

Applies to...	M3	M4	M4F
		✓	✓

3.10.1 Syntax

op{*cond*}{*Rd*,} *Rn*, *Rm*

where:

op
is any of:

SHADD16
Signed Halving Add 16

SHADD8
Signed Halving Add 8

cond
is an optional condition code, see 33.

Rd
is the destination register.

Rn
is the first operand register.

Rm
is the second operand register.

3.10.2 Operation

Use these instructions to add 16-bit and 8-bit data and then to halve the result before writing the result to the destination register:

The SHADD16 instruction:

1. Adds each halfword from the first operand to the corresponding halfword of the second operand.
2. Shuffles the result by one bit to the right, halving the data.
3. Writes the halfword results in the destination register.

The SHADDB8 instruction:

1. Adds each byte of the first operand to the corresponding byte of the second operand.
2. Shuffles the result by one bit to the right, halving the data.

3. Writes the byte results in the destination register.

3.10.3 Restrictions

Do not use SP and do not use PC.

3.10.4 Condition flags

These instructions do not change the flags.

3.10.5 Examples

```
SHADD16 R1, R0     ; Adds halfwords in R0 to corresponding halfword of R1 and
                   ; writes halved result to corresponding halfword in R1

SHADD8  R4, R0, R5 ; Adds bytes of R0 to corresponding byte in R5 and

                   ; writes halved result to corresponding byte in R4
```

3.11 SHASX and SHSAX

Signed Halving Add and Subtract with Exchange and Signed Halving Subtract and Add with Exchange.

Applies to...	M3	M4	M4F
		✓	✓

3.11.1 Syntax

op{*cond*} {*Rd*}, *Rn*, *Rm*

where:

op
is any of:

SHASX
Add and Subtract with Exchange and Halving.

SHSAX
Subtract and Add with Exchange and Halving.

cond
is an optional condition code, see 33.

Rd
is the destination register.

Rn, *Rm*
are registers holding the first and second operands.

3.11.2 Operation

The SHASX instruction:

1. Adds the top halfword of the first operand with the bottom halfword of the second operand.
2. Writes the halfword result of the addition to the top halfword of the destination register, shifted by one bit to the right causing a divide by two, or halving.
3. Subtracts the top halfword of the second operand from the bottom highword of the first operand.
4. Writes the halfword result of the division in the bottom halfword of the destination register, shifted by one bit to the right causing a divide by two, or halving.

The SHSAX instruction:

1. Subtracts the bottom halfword of the second operand from the top highword of the first operand.
2. Writes the halfword result of the addition to the bottom halfword of the destination register, shifted by one bit to the right causing a divide by two, or halving.
3. Adds the bottom halfword of the first operand with the top halfword of the second operand.

4. Writes the halfword result of the division in the top halfword of the destination register, shifted by one bit to the right causing a divide by two, or halving.

3.11.3 Restrictions

Do not use SP and do not use PC.

3.11.4 Condition flags

These instructions do not affect the condition code flags.

3.11.5 Examples

```
SHASX    R7, R4, R2      ; Adds top halfword of R4 to bottom halfword of R2
                         ; and writes halved result to top halfword of R7
                         ; Subtracts top halfword of R2 from bottom halfword of
                         ; R4 and writes halved result to bottom halfword of R7
SHSAX    R0, R3, R5      ; Subtracts bottom halfword of R5 from top halfword
                         ; of R3 and writes halved result to top halfword of R0
                         ; Adds top halfword of R5 to bottom halfword of R3 and
                         ; writes halved result to bottom halfword of R0
```

3.12 SHSUB16 and SHSUB8

Signed Halving Subtract 16 and Signed Halving Subtract 8.

Applies to...	M3	M4	M4F
		✓	✓

3.12.1 Syntax

op{*cond*}{*Rd*,} *Rn*, *Rm*

where:

op
is any of:

SHSUB16
Signed Halving Subtract 16

SHSUB8
Signed Halving Subtract 8

cond
is an optional condition code, see 33.

Rd
is the destination register.

Rn
is the first operand register.

Rm
is the second operand register

3.12.2 Operation

Use these instructions to add 16-bit and 8-bit data and then to halve the result before writing the result to the destination register:

The SHSUB16 instruction:

1. Subtracts each halfword of the second operand from the corresponding halfwords of the first operand.
2. Shuffles the result by one bit to the right, halving the data.
3. Writes the halved halfword results in the destination register.

The SHSUBB8 instruction:

1. Subtracts each byte of the second operand from the corresponding byte of the first operand,
2. Shuffles the result by one bit to the right, halving the data,

3. Writes the corresponding signed byte results in the destination register.

3.12.3 Restrictions

Do not use SP and do not use PC.

3.12.4 Condition flags

These instructions do not change the flags.

3.12.5 Examples

```
SHSUB16 R1, R0      ; Subtracts halfwords in R0 from corresponding halfword
                    ; of R1 and writes to corresponding halfword of R1

SHSUB8  R4, R0, R5  ; Subtracts bytes of R0 from corresponding byte in R5,

                    ; and writes to corresponding byte in R4
```

3.13 SSUB16 and SSUB8

Signed Subtract 16 and Signed Subtract 8.

Applies to...	M3	M4	M4F
		✓	✓

3.13.1 Syntax

op{*cond*}{*Rd*,} *Rn*, *Rm*

where:

op
is any of:

SSUB16
Performs two 16-bit signed integer subtractions.

SSUB8
Performs four 8-bit signed integer subtractions.

cond
is an optional condition code, see 33.

Rd
is the destination register.

Rn
is the first operand register.

Rm
is the second operand register.

3.13.2 Operation

Use these instructions to change endianness of data:

The SSUB16 instruction:

1. Subtracts each halfword from the second operand from the corresponding halfword of the first operand
2. Writes the difference result of two signed halfwords in the corresponding halfword of the destination register.

The SSUB8 instruction:

1. Subtracts each byte of the second operand from the corresponding byte of the first operand
2. Writes the difference result of four signed bytes in the corresponding byte of the destination register.

3.13.3 Restrictions

Do not use SP and do not use PC.

3.13.4 Condition flags

These instructions do not change the flags.

3.13.5 Examples

```
SSUB16 R1, R0       ; Subtracts halfwords in R0 from corresponding halfword of R1
                    ; and writes to corresponding halfword of R1

SSUB8  R4, R0, R5   ; Subtracts bytes of R5 from corresponding byte in
                    ; R0, and writes to corresponding byte of R4
```

3.14 SASX and SSAX

Signed Add and Subtract with Exchange and Signed Subtract and Add with Exchange.

Applies to...	M3	M4	M4F
		✓	✓

3.14.1 Syntax

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

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

where:

op
is any of:

SASX
Signed Add and Subtract with Exchange.

SSAX
Signed Subtract and Add with Exchange.

cond
is an optional condition code, see 33.

Rd
is the destination register.

Rn, *Rm*
are registers holding the first and second operands.

3.14.2 Operation

The SASX instruction:

1. Adds the signed top halfword of the first operand with the signed bottom halfword of the second operand.
2. Writes the signed result of the addition to the top halfword of the destination register.
3. Subtracts the signed bottom halfword of the second operand from the top signed highword of the first operand.
4. Writes the signed result of the subtraction to the bottom halfword of the destination register.

The SSAX instruction:

1. Subtracts the signed bottom halfword of the second operand from the top signed highword of the first operand.
2. Writes the signed result of the addition to the bottom halfword of the destination register.

3. Adds the signed top halfword of the first operand with the signed bottom halfword of the second operand.
4. Writes the signed result of the subtraction to the top halfword of the destination register.

3.14.3 Restrictions

Do not use SP and do not use PC.

3.14.4 Condition flags

These instructions do not affect the condition code flags.

3.14.5 Examples

```
SASX    R0, R4, R5     ; Adds top halfword of R4 to bottom halfword of R5 and
                       ; writes to top halfword of R0
                       ; Subtracts bottom halfword of R5 from top halfword of R4
                       ; and writes to bottom halfword of R0
SSAX    R7, R3, R2     ; Subtracts top halfword of R2 from bottom halfword of R3
                       ; and writes to bottom halfword of R7
                       ; Adds top halfword of R3 with bottom halfword of R2 and
                       ; writes to top halfword of R7
```

3.15 TST and TEQ

Test bits and Test Equivalence.

Applies to...	M3	M4	M4F
	✓	✓	✓

3.15.1 Syntax

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

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

where:

cond
Is an optional condition code. See Table 1-2 on page 33.

Rn
Is the register holding the first operand.

Operand2
Is a flexible second operand. See “Flexible Second Operand” on page 27 for details of the options.

3.15.2 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.

3.15.3 Restrictions

Do not use **SP** and do not use **PC**.

3.15.4 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 27.
- Do not affect the V flag.

3.15.5 Examples

```
TST     R0, #0x3F8  ; Perform bitwise AND of R0 value to 0x3F8;
                    ; APSR is updated but result is discarded.
TEQEQ   R10, R9     ; Conditionally test if value in R10 is equal to
                    ; value in R9; APSR is updated but result is discarded.
```

3.16 UADD16 and UADD8

Unsigned Add 16 and Unsigned Add 8.

Applies to...	M3	M4	M4F
		✓	✓

3.16.1 Syntax

op{*cond*}{*Rd*,} *Rn*, *Rm*

where:

op
is any of:

UADD16
Performs two 16-bit unsigned integer additions.

UADD8
Performs four 8-bit unsigned integer additions.

cond
is an optional condition code, see 33.

Rd
is the destination register.

Rn
is the first register holding the operand.

Rm
is the second register holding the operand.

3.16.2 Operation

Use these instructions to add 16- and 8-bit unsigned data:

The UADD16 instruction:

1. adds each halfword from the first operand to the corresponding halfword of the second operand.
2. writes the unsigned result in the corresponding halfwords of the destination register.

The UADD16 instruction:

1. adds each byte of the first operand to the corresponding byte of the second operand.
2. writes the unsigned result in the corresponding byte of the destination register.

3.16.3 Restrictions

Do not use SP and do not use PC.

3.16.4 Condition flags

These instructions do not change the flags.

3.16.5 Examples

```
UADD16 R1, R0      ; Adds halfwords in R0 to corresponding halfword of R1,
                   ; writes to corresponding halfword of R1

UADD8  R4, R0, R5  ; Adds bytes of R0 to corresponding byte in R5 and writes
                   ; to corresponding byte in R4
```

3.17 UASX and USAX

Add and Subtract with Exchange and Subtract and Add with Exchange.

Applies to...	M3	M4	M4F
		✓	✓

3.17.1 Syntax

op{*cond*} {*Rd*}, *Rn*, *Rm*

where:

op
is one of:

UASX
Add and Subtract with Exchange.

USAX
Subtract and Add with Exchange.

cond
is an optional condition code, see 33.

Rd
is the destination register.

Rn, *Rm*
are registers holding the first and second operands.

3.17.2 Operation

The UASX instruction:

1. Subtracts the top halfword of the second operand from the bottom halfword of the first operand.
2. Writes the unsigned result from the subtraction to the bottom halfword of the destination register.
3. Adds the top halfword of the first operand with the bottom halfword of the second operand.
4. Writes the unsigned result of the addition to the top halfword of the destination register.

The USAX instruction:

1. Adds the bottom halfword of the first operand with the top halfword of the second operand.
2. Writes the unsigned result of the addition to the bottom halfword of the destination register.
3. Subtracts the bottom halfword of the second operand from the top halfword of the first operand.
4. Writes the unsigned result from the subtraction to the top halfword of the destination register.

3.17.3 Restrictions

Do not use SP and do not use PC.

3.17.4 Condition flags

These instructions do not affect the condition code flags.

3.17.5 Examples

```
UASX    R0, R4, R5     ; Adds top halfword of R4 to bottom halfword of R5 and
                       ; writes to top halfword of R0
                       ; Subtracts bottom halfword of R5 from top halfword of R0
                       ; and writes to bottom halfword of R0
USAX    R7, R3, R2     ; Subtracts top halfword of R2 from bottom halfword of R3
                       ; and writes to bottom halfword of R7
                       ; Adds top halfword of R3 to bottom halfword of R2 and
                       ; writes to top halfword of R7
```

3.18 UHADD16 and UHADD8

Unsigned Halving Add 16 and Unsigned Halving Add 8.

Applies to...	M3	M4	M4F
		✓	✓

3.18.1 Syntax

op{*cond*}{*Rd*,} *Rn*, *Rm*

where:

op
is any of:

UHADD16
Unsigned Halving Add 16.

UHADD8
Unsigned Halving Add 8.

cond
is an optional condition code, see 33.

Rd
is the destination register.

Rn
is the register holding the first operand.

Rm
is the register holding the second operand.

3.18.2 Operation

Use these instructions to add 16- and 8-bit data and then to halve the result before writing the result to the destination register:

The UHADD16 instruction:

1. Adds each halfword from the first operand to the corresponding halfword of the second operand.
2. Shuffles the halfword result by one bit to the right, halving the data.
3. Writes the unsigned results to the corresponding halfword in the destination register.

The UHADD8 instruction:

1. Adds each byte of the first operand to the corresponding byte of the second operand.
2. Shuffles the byte result by one bit to the right, halving the data.
3. Writes the unsigned results in the corresponding byte in the destination register.

3.18.3 Restrictions

Do not use SP and do not use PC.

3.18.4 Condition flags

These instructions do not change the flags.

3.18.5 Examples

```
UHADD16 R7, R3      ; Adds halfwords in R7 to corresponding halfword of R3
                    ; and writes halved result to corresponding halfword in R7

UHADD8  R4, R0, R5  ; Adds bytes of R0 to corresponding byte in R5 and writes

                    ; halved result to corresponding byte in R4
```

3.19 UHASX and UHSAX

Unsigned Halving Add and Subtract with Exchange and Unsigned Halving Subtract and Add with Exchange.

Applies to...	M3	M4	M4F
		✓	✓

3.19.1 Syntax

op{*cond*} {*Rd*}, *Rn*, *Rm*

where:

op
is one of:

UHASX
Add and Subtract with Exchange and Halving.

UHSAX
Subtract and Add with Exchange and Halving.

cond
is an optional condition code, see 33.

Rd
is the destination register.

Rn, *Rm*
are registers holding the first and second operands.

3.19.2 Operation

The UHASX instruction:

1. Adds the top halfword of the first operand with the bottom halfword of the second operand.
2. Shifts the result by one bit to the right causing a divide by two, or halving.
3. Writes the halfword result of the addition to the top halfword of the destination register.
4. Subtracts the top halfword of the second operand from the bottom highword of the first operand.
5. Shifts the result by one bit to the right causing a divide by two, or halving.
6. Writes the halfword result of the division in the bottom halfword of the destination register.

The UHSAX instruction:

1. Subtracts the bottom halfword of the second operand from the top highword of the first operand.
2. Shifts the result by one bit to the right causing a divide by two, or halving.
3. Writes the halfword result of the subtraction in the top halfword of the destination register.

4. Adds the bottom halfword of the first operand with the top halfword of the second operand.
5. Shifts the result by one bit to the right causing a divide by two, or halving.
6. Writes the halfword result of the addition to the bottom halfword of the destination register.

3.19.3 Restrictions

Do not use SP and do not use PC.

3.19.4 Condition flags

These instructions do not affect the condition code flags.

3.19.5 Examples

```
UHASX    R7, R4, R2     ; Adds top halfword of R4 with bottom halfword of R2
                        ; and writes halved result to top halfword of R7
                        ; Subtracts top halfword of R2 from bottom halfword of
                        ; R7 and writes halved result to bottom halfword of R7
UHSAX    R0, R3, R5     ; Subtracts bottom halfword of R5 from top halfword of
                        ; R3 and writes halved result to top halfword of R0
                        ; Adds top halfword of R5 to bottom halfword of R3 and
                        ; writes halved result to bottom halfword of R0
```

3.20 UHSUB16 and UHSUB8

Unsigned Halving Subtract 16 and Unsigned Halving Subtract 8.

Applies to...	M3	M4	M4F
		✓	✓

3.20.1 Syntax

op{*cond*}{*Rd*,} *Rn*, *Rm*

where:

op
is any of:

UHSUB16
Performs two unsigned 16-bit integer additions, halves the results, and writes the results to the destination register.

UHSUB8
Performs four unsigned 8-bit integer additions, halves the results, and writes the results to the destination register.

cond
is an optional condition code, see 33.

Rd
is the destination register.

Rn
is the first register holding the operand.

Rm
is the second register holding the operand.

3.20.2 Operation

Use these instructions to add 16-bit and 8-bit data and then to halve the result before writing the result to the destination register:

The UHSUB16 instruction:

1. Subtracts each halfword of the second operand from the corresponding halfword of the first operand.
2. Shuffles each halfword result to the right by one bit, halving the data.
3. Writes each unsigned halfword result to the corresponding halfwords in the destination register.

The UHSUB8 instruction:

1. Subtracts each byte of second operand from the corresponding byte of the first operand.
2. Shuffles each byte result by one bit to the right, halving the data.

3. Writes the unsigned byte results to the corresponding byte of the destination register.

3.20.3 Restrictions

Do not use SP and do not use PC.

3.20.4 Condition flags

These instructions do not change the flags.

3.20.5 Examples

```
UHSUB16  R1, R0      ; Subtracts halfwords in R0 from corresponding halfword of
                     ; R1 and writes halved result to corresponding halfword in

                     ; R1

UHSUB8  R4, R0, R5   ; Subtracts bytes of R5 from corresponding byte in R0 and

                     ; writes halved result to corresponding byte in R4
```

3.21 SEL

Select Bytes. Selects each byte of its result from either its first operand or its second operand, according to the values of the GE flags.

Applies to...	M3	M4	M4F
		✓	✓

3.21.1 Syntax

```
SEL{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

where:

<c>, <q>
is a standard assembler syntax fields.

<Rd>
is the destination register.

<Rn>
is the first operand register.

<Rm>
is the second operand register.

3.21.2 Operation

The SEL instruction:

1. Reads the value of each bit of APSR.GE.
2. Depending on the value of APSR.GE, assigns the destination register the value of either the first or second operand register.

3.21.3 Restrictions

None.

3.21.4 Condition flags

These instructions do not change the flags.

3.21.5 Examples

```
SADD16 R0, R1, R2     ; Set GE bits based on result

SEL R0, R0, R3        ; Select bytes from R0 or R3, based on GE
```

3.22 USAD8

Unsigned Sum of Absolute Differences.

Applies to...	M3	M4	M4F
		✓	✓

3.22.1 Syntax

USAD8{*cond*}{*Rd*,} *Rn*, *Rm*

where:

cond
is an optional condition code, see 33.

Rd
is the destination register.

Rn
is the first operand register.

Rm
is the second operand register.

3.22.2 Operation

The USAD8 instruction:

1. Subtracts each byte of the second operand register from the corresponding byte of the first operand register.
2. Adds the absolute values of the differences together.
3. Writes the result to the destination register.

3.22.3 Restrictions

Do not use SP and do not use PC.

3.22.4 Condition flags

These instructions do not change the flags.

3.22.5 Examples

```
USAD8 R1, R4, R0      ; Subtracts each byte in R0 from corresponding byte of R4
                      ; adds the differences and writes to R1

USAD8  R0, R5         ; Subtracts bytes of R5 from corresponding byte in R0,

                      ; adds the differences and writes to R0
```

3.23 USADA8

Unsigned Sum of Absolute Differences and Accumulate.

Applies to...	M3	M4	M4F
		✓	✓

3.23.1 Syntax

USADA8{*cond*}{*Rd*,} *Rn*, *Rm*, *Ra*

where:

cond
is an optional condition code, see 33.

Rd
is the destination register.

Rn
is the first operand register.

Rm
is the second operand register.

Ra
is the register that contains the accumulation value.

3.23.2 Operation

The USADA8 instruction:

1. Subtracts each byte of the second operand register from the corresponding byte of the first operand register.
2. Adds the unsigned absolute differences together.
3. Adds the accumulation value to the sum of the absolute differences.
4. Writes the result to the destination register.

3.23.3 Restrictions

Do not use SP and do not use PC.

3.23.4 Condition flags

These instructions do not change the flags.

3.23.5 Examples

```
USADA8 R1, R0, R6      ; Subtracts bytes in R0 from corresponding halfword of R1
                       ; adds differences, adds value of R6, writes to R1
```

```
USADA8  R4, R0, R5, R2 ; Subtracts bytes of R5 from corresponding byte in R0

                       ; adds differences, adds value of R2 writes to R4
```

3.24 USUB16 and USUB8

Unsigned Subtract 16 and Unsigned Subtract 8.

Applies to...	M3	M4	M4F
		✓	✓

3.24.1 Syntax

op{*cond*}{*Rd*,} *Rn*, *Rm*

where:

op
is any of:

USUB16
Unsigned Subtract 16.

USUB8
Unsigned Subtract 8.

cond
is an optional condition code, see 33.

Rd
is the destination register.

Rn
is the first operand register.

Rm
is the second operand register.

3.24.2 Operation

Use these instructions to subtract 16-bit and 8-bit data before writing the result to the destination register:

The USUB16 instruction:

1. Subtracts each halfword from the second operand register from the corresponding halfword of the first operand register.
2. Writes the unsigned result in the corresponding halfwords of the destination register.

The USUB8 instruction:

1. Subtracts each byte of the second operand register from the corresponding byte of the first operand register.
2. Writes the unsigned byte result in the corresponding byte of the destination register.

3.24.3 Restrictions

Do not use SP and do not use PC.

3.24.4 Condition flags

These instructions do not change the flags.

3.24.5 Examples

```
USUB16 R1, R0      ; Subtracts halfwords in R0 from corresponding halfword of R1
                   ; and writes to corresponding halfword in R1USUB8  R4, R0, R5  ;


                   ; writes to the corresponding byte in R4
```

4 Multiply and Divide Instructions

Table 4-1 on page 105 shows the multiply and divide instructions:

Table 4-1. Multiply and Divide Instructions

Mnemonic	Brief Description	See Page
MLA	Multiply with accumulate, 32-bit result	106
MLS	Multiply and subtract, 32-bit result	106
MUL	Multiply, 32-bit result	106
SDIV	Signed divide	130
SMLA[B,T]	Signed multiply accumulate (halfwords)	108
SMLAD, SMLADX	Signed multiply accumulate dual	111
SMLAL	Signed multiply with accumulate (32x32+64), 64-bit result	128
SMLAL[B,T]	Signed multiply accumulate long (halfwords)	113
SMLALD, SMLALDX	Signed multiply accumulate long dual	113
SMLAW[B|T]	Signed multiply accumulate (word by halfword)	108
SMLSD	Signed multiply subtract dual	116
SMLSLD	Signed multiply subtract long dual	116
SMMLA	Signed most significant word multiply accumulate	119
SMMLS, SMMLSR	Signed most significant word multiply subtract	119
SMUAD, SMUADX	Signed dual multiply add	123
SMUL[B,T]	Signed multiply (word by halfword)	125
SMMUL, SMMULR	Signed most significant word multiply	121
SMULL	Signed multiply (32x32), 64-bit result	128
SMULWB, SMULWT	Signed multiply (word by halfword)	125
SMUSD, SMUSDX	Signed dual multiply subtract	123
UDIV	Unsigned divide	130
UMAAL	Unsigned multiply accumulate accumulate long (32x32+32+32), 64-bit result	128
UMLAL	Unsigned multiply with accumulate (32x32+64), 64-bit result	128
UMULL	Unsigned multiply (32x32), 64-bit result	128

4.1 MUL, MLA, and MLS

Multiply, Multiply with Accumulate, and Multiply with Subtract, using 32-bit operands, and producing a 32-bit result.

Applies to...	M3	M4	M4F
	✓	✓	✓

4.1.1 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 Table 1-2 on page 33.

S
Is an optional suffix. If S is specified, the condition code flags are updated on the result of the operation. See 33.

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 or subtracted from.

4.1.2 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 *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.

4.1.3 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.

4.1.4 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.

4.1.5 Examples

```
MUL     R10, R2, R5     ; Multiply, R10 = R2 x R5.
MLA     R10, R2, R1, R5 ; Multiply with accumulate, R10 = (R2 x R1) + R5.
MULS    R0, R2, R2      ; Multiply with flag update, R0 = R2 x R2.
MULLT   R2, R3, R2      ; Conditionally multiply, R2 = R3 x R2.
MLS     R4, R5, R6, R7  ; Multiply with subtract, R4 = R7 - (R5 x R6).
```

4.2 SMLA and SMLAW

Signed Multiply Accumulate (halfwords).

Applies to...	M3	M4	M4F
		✓	✓

4.2.1 Syntax

`op{X}{Y}{cond} Rd, Rn, Rm`

`op{Y}{cond} Rd, Rn, Rm, Ra`

where

op
is one of:

SMLAXY
Signed Multiply Accumulate Long (halfwords)

X and *Y* specifies which half of the source registers *Rn* and *Rm* are used as the first and second multiply operand.

If *X* is *B*, then the bottom halfword, bits [15:0], of *Rn* is used. If *X* is *T*, then the top halfword, bits [31:16], of *Rn* is used.

If *Y* is *B*, then the bottom halfword, bits [15:0], of *Rm* is used. If *Y* is *T*, then the top halfword, bits [31:16], of *Rm* is used.

SMLAWY
Signed Multiply Accumulate (word by halfword)

Y specifies which half of the source register *Rm* is used as the second multiply operand.

If *Y* is T, then the top halfword, bits [31:16] of *Rm* is used.

If *Y* is B, then the bottom halfword, bits [15:0] of *Rm* is used.

cond
is an optional condition code, see Table 1-2 on page 33.

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 or subtracted from.

4.2.2 Operation

The SMALBB, SMLABT, SMLATB, SMLATT instructions:

- Multiplies the specified signed halfword, top or bottom, values from *Rn* and *Rm*.
- Adds the value in *Ra* to the resulting 32-bit product.
- Writes the result of the multiplication and addition in *Rd*.

The non-specified halfwords of the source registers are ignored.

The SMLAWB and SMLAWT instructions:

- Multiply the 32-bit signed values in *Rn* with:
  - The top signed halfword of *Rm*, *T* instruction suffix.
  - The bottom signed halfword of *Rm*, *B* instruction suffix.
- Add the 32-bit signed value in *Ra* to the top 32 bits of the 48-bit product
- Writes the result of the multiplication and addition in *Rd*.

The bottom 16 bits of the 48-bit product are ignored.

If overflow occurs during the addition of the accumulate value, the instruction sets the Q flag in the APSR. No overflow can occur during the multiplication.

4.2.3 Restrictions

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

4.2.4 Condition flags

If an overflow is detected, the Q flag is set.

4.2.5 Examples

```
SMLABB  R5, R6, R4, R1  ; Multiplies bottom halfwords of R6 and R4, adds
                        ; R1 and writes to R5
SMLATB  R5, R6, R4, R1  ; Multiplies top halfword of R6 with bottom halfword
                        ; of R4, adds R1 and writes to R5
SMLATT  R5, R6, R4, R1  ; Multiplies top halfwords of R6 and R4, adds
                        ; R1 and writes the sum to R5
SMLABT  R5, R6, R4, R1  ; Multiplies bottom halfword of R6 with top halfword
                        ; of R4, adds R1 and writes to R5
```

```
SMLABT  R4, R3, R2      ; Multiplies bottom halfword of R4 with top halfword of
                        ; R3, adds R2 and writes to R4
SMLAWB  R10, R2, R5, R3 ; Multiplies R2 with bottom halfword of R5, adds
                        ; R3 to the result and writes top 32-bits to R10
SMLAWT  R10, R2, R1, R5 ; Multiplies R2 with top halfword of R1, adds R5
                        ; and writes top 32-bits to R10
```

4.3 SMLAD

Signed Multiply Accumulate Long Dual.

Applies to...	M3	M4	M4F
		✓	✓

4.3.1 Syntax

`SMLAD{X}{cond} Rd, Rn, Rm, Ra  ;`

where:

SMLAD
Signed Multiply Accumulate Dual

SMLADX
Signed Multiply Accumulate Dual Reverse

X
Specifies which halfword of the source register *Rn* is used as the multiply operand. If *X* is omitted, the multiplications are bottom × bottom and top × top. If *X* is present, the multiplications are bottom × top and top × bottom.

cond
is an optional condition code, see Table 1-2 on page 33.

Rd
is the destination register.

Rn
is the first operand register holding the values to be multiplied.

Rm
the second operand register.

Ra
is the accumulate value.

4.3.2 Operation

The `SMLAD` and `SMLADX` instructions regard the two operands as four halfword 16-bit values. The `SMLAD` and `SMLADX` instructions:

- If *X* is not present, multiply the top signed halfword value in *Rn* with the top signed halfword of *Rm* and the bottom signed halfword values in *Rn* with the bottom signed halfword of *Rm*.
- Or if *X* is present, multiply the top signed halfword value in *Rn* with the bottom signed halfword of *Rm* and the bottom signed halfword values in *Rn* with the top signed halfword of *Rm*.
- Add both multiplication results to the signed 32-bit value in *Ra*.
- Writes the 32-bit signed result of the multiplication and addition to *Rd*.

4.3.3 Restrictions

Do not use SP and do not use PC.

4.3.4 Condition flags

These instructions do not change the flags.

4.3.5 Examples

```
SMLAD   R10, R2, R1, R5 ; Multiplies two halfword values in R2 with

                        ; corresponding halfwords in R1, adds R5 and writes to
                        ; R10

SMLALDX R0, R2, R4, R6  ; Multiplies top halfword of R2 with bottom halfword

                        ; of R4, multiplies bottom halfword of R2 with top

                        ; halfword of R4, adds R6 and writes to R0
```

4.4 SMLAL and SMLALD

Signed Multiply Accumulate Long and Signed Multiply Accumulate Long Dual and Signed Multiply Accumulate Long (halfwords).

Applies to...	SMLAL			SMLALD		
	M3	M4	M4F	M3	M4	M4F
	✓	✓	✓		✓	✓

4.4.1 Syntax

SMLAL{*cond*} *RdLo*, *RdHi*, *Rn*, *Rm*

SMLAL{X}{Y}{*cond*} *RdLo*, *RdHi*, *Rn*, *Rm*

SMLALD{*X*}{*cond*} *RdLo*, *RdHi*, *Rn*, *Rm*

where:

SMLAL
Signed Multiply Accumulate Long

SMLAL{X}{Y}
Signed Multiply Accumulate Long (halfwords, X and Y)

{X}{Y}
Specifies which halfword of the source registers *Rn* and *Rm* are used as the first and second multiply operand:

If *X* is B, then the bottom halfword, bits [15:0], of *Rn* is used. If *X* is T, then the top halfword, bits [31:16], of *Rn* is used.

If *Y* is B, then the bottom halfword, bits [15:0], of *Rm* is used. If *Y* is T, then the top halfword, bits [31:16], of *Rm* is used.

SMLALD
Signed Multiply Accumulate Long Dual

SMLALDX
Signed Multiply Accumulate Long Dual Reversed

If the *X* is omitted, the multiplications are bottom × bottom and top × top.

If *X* is present, the multiplications are bottom × top and top × bottom.

cond
is an optional condition code, see Table 1-2 on page 33.

RdHi, *RdLo*
are the destination registers. *RdLo* is the lower 32 bits and *RdHi* is the upper 32 bits of the 64-bit integer. For SMLAL, SMLALBB, SMLALBT, SMLALTB, SMLALTT, SMLALD and SMLALDX, they also hold the accumulating value.

Rn, Rm
are registers holding the first and second operands.

4.4.2 Operation

The SMLAL instruction:

- Multiplies the two's complement signed word values from *Rn* and *Rm*.
- Adds the 64-bit value in *RdLo* and *RdHi* to the resulting 64-bit product.
- Writes the 64-bit result of the multiplication and addition in *RdLo* and *RdHi*.

The SMLALBB, SMLALBT, SMLALTB and SMLALTT instructions:

- Multiplies the specified signed halfword, Top or Bottom, values from *Rn* and *Rm*.
- Adds the resulting sign-extended 32-bit product to the 64-bit value in *RdLo* and *RdHi*.
- Writes the 64-bit result of the multiplication and addition in *RdLo* and *RdHi*.

The non-specified halfwords of the source registers are ignored.

The SMLALD and SMLALDX instructions interpret the values from *Rn* and *Rm* as four halfword two's complement signed 16-bit integers. These instructions:

- if *X* is not present, multiply the top signed halfword value of *Rn* with the top signed halfword of *Rm* and the bottom signed halfword values of *Rn* with the bottom signed halfword of *Rm*.
- Or if *X* is present, multiply the top signed halfword value of *Rn* with the bottom signed halfword of *Rm* and the bottom signed halfword values of *Rn* with the top signed halfword of *Rm*.
- Add the two multiplication results to the signed 64-bit value in *RdLo* and *RdHi* to create the resulting 64-bit product.
- Write the 64-bit product in *RdLo* and *RdHi*.

4.4.3 Restrictions

In these instructions:

- do not use SP and do not use PC.
- *RdHi* and *RdLo* must be different registers.

4.4.4 Condition flags

These instructions do not affect the condition code flags.

4.4.5 Examples

```
SMLAL       R4, R5, R3, R8   ; Multiplies R3 and R8, adds R5:R4 and writes to

                             ; R5:R4

SMLALBT     R2, R1, R6, R7   ; Multiplies bottom halfword of R6 with top

                             ; halfword of R7, sign extends to 32-bit, adds
```

```
                                ; R1:R2 and writes to R1:R2
SMLALTB     R2, R1, R6, R7      ; Multiplies top halfword of R6 with bottom
                                ; halfword of R7,sign extends to 32-bit, adds R1:R2
                                ; and writes to R1:R2
SMLALD      R6, R8, R5, R1      ; Multiplies top halfwords in R5 and R1 and bottom
                                ; halfwords of R5 and R1, adds R8:R6 and writes to
                                ; R8:R6
SMLALDX     R6, R8, R5, R1      ; Multiplies top halfword in R5 with bottom
                                ; halfword of R1, and bottom halfword of R5 with
                                ; top halfword of R1, adds R8:R6 and writes to
                                ; R8:R6
```

4.5 SMLSD and SMLSLD

Signed Multiply Subtract Dual and Signed Multiply Subtract Long Dual.

Applies to...	M3	M4	M4F
		✓	✓

4.5.1 Syntax

`op{X}{cond} Rd, Rn, Rm, Ra`

where:

op
is one of:

`SMLSD`
Signed Multiply Subtract Dual.

`SMLSLD`
Signed Multiply Subtract Long Dual.

X
If *X* is present, the multiplications are bottom × top and top × bottom. If the *X* is omitted, the multiplications are bottom × bottom and top × top.

cond
is an optional condition code, see Table 1-2 on page 33.

Rd
is the destination register.

Rn, Rm
are registers holding the first and second operands.

Ra
is the register holding the accumulate value.

4.5.2 Operation

The `SMLSD` instruction interprets the values from the first and second operands as four signed halfwords. This instruction:

- Optionally rotates the halfwords of the second operand.
- Performs two signed 16 × 16-bit halfword multiplications.
- Subtracts the result of the upper halfword multiplication from the result of the lower halfword multiplication.
- Adds the signed accumulate value to the result of the subtraction.
- Writes the result of the addition to the destination register.

The `SMLSLD` instruction interprets the values from *Rn* and *Rm* as four signed halfwords. This instruction:

- Optionally rotates the halfwords of the second operand.

- Performs two signed 16 × 16-bit halfword multiplications.
- Subtracts the result of the upper halfword multiplication from the result of the lower halfword multiplication.
- Adds the 64-bit value in *RdHi* and *RdLo* to the result of the subtraction.
- Writes the 64-bit result of the addition to the *RdHi* and *RdLo*.

4.5.3 Restrictions

In these instructions:

- Do not use SP and do not use PC.

4.5.4 Condition flags

This instruction sets the Q flag if the accumulate operation overflows. Overflow cannot occur during the multiplications or subtraction.

For the Thumb instruction set, these instructions do not affect the condition code flags.

4.5.5 Examples

```
SMLSD       R0, R4, R5, R6   ; Multiplies bottom halfword of R4 with bottom
                             ; halfword of R5, multiplies top halfword of R4
                             ; with top halfword of R5, subtracts second from
                             ; first, adds R6, writes to R0
SMLSDX      R1, R3, R2, R0   ; Multiplies bottom halfword of R3 with top
                             ; halfword of R2, multiplies top halfword of R3
                             ; with bottom halfword of R2, subtracts second from
                             ; first, adds R0, writes to R1
SMLSLD      R3, R6, R2, R7   ; Multiplies bottom halfword of R6 with bottom
                             ; halfword of R2, multiplies top halfword of R6
                             ; with top halfword of R2, subtracts second from
                             ; first, adds R6:R3, writes to R6:R3
```

```
SMLSLDX     R3, R6, R2, R7   ; Multiplies bottom halfword of R6 with top
                             ; halfword of R2, multiplies top halfword of R6
                             ; with bottom halfword of R2, subtracts second from
                             ; first, adds R6:R3, writes to R6:R3
```

4.6 SMMLA and SMMLS

Signed Most Significant Word Multiply Accumulate and Signed Most Significant Word Multiply Subtract.

Applies to...	M3	M4	M4F
		✓	✓

4.6.1 Syntax

`op{R}{cond} Rd, Rn, Rm, Ra`

where:

op
is one of:

`SMMLA`
Signed Most Significant Word Multiply Accumulate.

`SMMLS`
Signed Most Significant Word Multiply Subtract.

R
is a rounding error flag. If *R* is specified, the result is rounded instead of being truncated. In this case the constant `0x80000000` is added to the product before the high word is extracted.

cond
is an optional condition code, see Table 1-2 on page 33.

Rd
is the destination register.

Rn, Rm
are registers holding the first and second multiply operands.

Ra
is the register holding the accumulate value.

4.6.2 Operation

The `SMMLA` instruction interprets the values from *Rn* and *Rm* as signed 32-bit words.

The `SMMLA` instruction:

- Multiplies the values in *Rn* and *Rm*.
- Optionally rounds the result by adding `0x80000000`.
- Extracts the most significant 32 bits of the result.
- Adds the value of *Ra* to the signed extracted value.
- Writes the result of the addition in *Rd*.

The `SMMLS` instruction interprets the values from *Rn* and *Rm* as signed 32-bit words.

The `SMMLS` instruction:

- Multiplies the values in *Rn* and *Rm*.
- Optionally rounds the result by adding `0x80000000`.
- Extracts the most significant 32 bits of the result.
- Subtracts the extracted value of the result from the value in *Ra*.
- Writes the result of the subtraction in *Rd*.

4.6.3 Restrictions

In these instructions:

- Do not use SP and do not use PC.

4.6.4 Condition flags

These instructions do not affect the condition code flags.

4.6.5 Examples

```
SMMLA      R0, R4, R5, R6   ; Multiplies R4 and R5, extracts top 32 bits, adds
                            ; R6, truncates and writes to R0
SMMLAR     R6, R2, R1, R4   ; Multiplies R2 and R1, extracts top 32 bits, adds
                            ; R4, rounds and writes to R6
SMMLSR     R3, R6, R2, R7   ; Multiplies R6 and R2, extracts top 32 bits,
                            ; subtracts R7, rounds and writes to R3
SMMLS      R4, R5, R3, R8   ; Multiplies R5 and R3, extracts top 32 bits,
                            ; subtracts R8, truncates and writes to R4
```

4.7 SMMUL

Signed Most Significant Word Multiply.

Applies to...	M3	M4	M4F
		✓	✓

4.7.1 Syntax

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

where:

op
is one of:

SMMUL
Signed Most Significant Word Multiply

R
is a rounding error flag. If *R* is specified, the result is rounded instead of being truncated. In this case the constant `0x80000000` is added to the product before the high word is extracted.

cond
is an optional condition code, see Table 1-2 on page 33.

Rd
is the destination register.

Rn, Rm
are registers holding the first and second operands.

4.7.2 Operation

The `SMMUL` instruction interprets the values from *Rn* and *Rm* as two's complement 32-bit signed integers. The `SMMUL` instruction:

- Multiplies the values from *Rn* and *Rm*.
- Optionally rounds the result, otherwise truncates the result.
- Writes the most significant signed 32 bits of the result in *Rd*.

4.7.3 Restrictions

In this instruction:

- do not use SP and do not use PC.

4.7.4 Condition flags

This instruction does not affect the condition code flags.

4.7.5 Examples

```
SMULL       R0, R4, R5       ; Multiplies R4 and R5, truncates top 32 bits

                             ; and writes to R0

SMULLR      R6, R2           ; Multiplies R6 and R2, rounds the top 32 bits
                             ; and writes to R6
```

4.8 SMUAD and SMUSD

Signed Dual Multiply Add and Signed Dual Multiply Subtract.

Applies to...	M3	M4	M4F
		✓	✓

4.8.1 Syntax

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

where:

op
is one of:

`SMUAD`
Signed Dual Multiply Add.

`SMUSD`
Signed Dual Multiply Subtract.

X
If *X* is present, the multiplications are bottom × top and top × bottom. If the *X* is omitted, the multiplications are bottom × bottom and top × top.

cond
is an optional condition code, see Table 1-2 on page 33.

Rd
is the destination register.

Rn, Rm
are registers holding the first and the second operands.

4.8.2 Operation

The `SMUAD` instruction interprets the values from the first and second operands as two signed halfwords in each operand. This instruction:

- Optionally rotates the halfwords of the second operand.
- Performs two signed 16 × 16-bit multiplications.
- Adds the two multiplication results together.
- Writes the result of the addition to the destination register.

The `SMUSD` instruction interprets the values from the first and second operands as two's complement signed integers. This instruction:

- Optionally rotates the halfwords of the second operand.
- Performs two signed 16 × 16-bit multiplications.
- Subtracts the result of the top halfword multiplication from the result of the bottom halfword multiplication.
- Writes the result of the subtraction to the destination register.

4.8.3 Restrictions

In these instructions:

- Do not use SP and do not use PC.

4.8.4 Condition flags

Sets the Q flag if the addition overflows. The multiplications cannot overflow.

4.8.5 Examples

```
SMUAD    R0, R4, R5      ; Multiplies bottom halfword of R4 with the bottom
                         ; halfword of R5, adds multiplication of top halfword
                         ; of R4 with top halfword of R5, writes to R0
SMUADX   R3, R7, R4      ; Multiplies bottom halfword of R7 with top halfword
                         ; of R4, adds multiplication of top halfword of R7
                         ; with bottom halfword of R4, writes to R3
SMUSD    R3, R6, R2      ; Multiplies bottom halfword of R4 with bottom halfword
                         ; of R6, subtracts multiplication of top halfword of R6
                         ; with top halfword of R3, writes to R3
SMUSDX   R4, R5, R3      ; Multiplies bottom halfword of R5 with top halfword of
                         ; R3, subtracts multiplication of top halfword of R5
                         ; with bottom halfword of R3, writes to R4
```

4.9 SMUL and SMULW

Signed Multiply (halfwords) and Signed Multiply (word by halfword).

Applies to...	M3	M4	M4F
		✓	✓

4.9.1 Syntax

`SMUL{X}{Y}{cond} Rd,Rn, Rm`

`SMULW{Y}{cond} Rd. Rn, Rm`

For *SMULXY* only:

op
is one of:

SMUL{*X*}{*Y*} Signed Multiply (halfwords)

X and *Y* specify which halfword of the source registers *Rn* and *Rm* is used as the first and second multiply operand. If *X* is B, then the bottom halfword, bits [15:0] of *Rn* is used. If *X* is T, then the top halfword, bits [31:16] of *Rn* is used.If *Y* is B, then the bottom halfword, bits [15:0], of *Rm* is used. If *Y* is T, then the top halfword, bits [31:16], of *Rm* is used.

SMULW{Y} Signed Multiply (word by halfword)

Y specifies which halfword of the source register *Rm* is used as the second multiply operand. If *Y* is B, then the bottom halfword (bits [15:0]) of *Rm* is used. If *Y* is T, then the top halfword (bits [31:16]) of *Rm* is used.

cond
is an optional condition code, see Table 1-2 on page 33.

Rd
is the destination register.

Rn, Rm
are registers holding the first and second operands.

4.9.2 Operation

The SMULBB, SMULTB, SMULBT and SMULTT instructions interprets the values from *Rn* and *Rm* as four signed 16-bit integers. These instructions:

- Multiplies the specified signed halfword, Top or Bottom, values from *Rn* and *Rm*.
- Writes the 32-bit result of the multiplication in *Rd.*

The SMULWT and SMULWB instructions interprets the values from *Rn* as a 32-bit signed integer and *Rm* as two halfword 16-bit signed integers. These instructions:

- Multiplies the first operand and the top, T suffix, or the bottom, B suffix, halfword of the second operand.

- Writes the signed most significant 32 bits of the 48-bit result in the destination register.

4.9.3 Restrictions

In these instructions:

- Do not use SP and do not use PC.
- *RdHi* and *RdLo* must be different registers.

4.9.4 Examples

```
SMULBT       R0, R4, R5       ; Multiplies the bottom halfword of R4 with the
                              ; top halfword of R5, multiplies results and
                              ; writes to R0
SMULBB       R0, R4, R5       ; Multiplies the bottom halfword of R4 with the
                              ; bottom halfword of R5, multiplies results and
                              ; writes to R0
SMULTT       R0, R4, R5       ; Multiplies the top halfword of R4 with the top
                              ; halfword of R5, multiplies results and writes
                              ; to R0
SMULTB       R0, R4, R5       ; Multiplies the top halfword of R4 with the
                              ; bottom halfword of R5, multiplies results and
                              ; and writes to R0
SMULWT       R4, R5, R3       ; Multiplies R5 with the top halfword of R3,
                              ; extracts top 32 bits and writes to R4
SMULWB       R4, R5, R3       ; Multiplies R5 with the bottom halfword of R3,
```

```
                                        ; extracts top 32 bits and writes to R4
```

4.10 UMULL, UMAAL, UMLAL, SMULL, and SMLAL

Signed and Unsigned Long Multiply, with optional Accumulate, using 32-bit operands and producing a 64-bit result.

Applies to...	UMULL, UMLAL, SMULL, SMLAL			UMAAL		
	M3	M4	M4F	M3	M4	M4F
	✓	✓	✓		✓	✓

4.10.1 Syntax

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

where:

op
is one of:

`UMULL`
Unsigned Long Multiply.

`UMAAL`
Unsigned Long Multiply with Accumulate Accumulate.

`UMLAL`
Unsigned Long Multiply, with Accumulate.

`SMULL`
Signed Long Multiply.

`SMLAL`
Signed Long Multiply, with Accumulate.

cond
is an optional condition code, see Table 1-2 on page 33.

RdHi, *RdLo*
are the destination registers. For `UMLAL` and `SMLAL` they also hold the accumulating value.

Rn, Rm
are registers holding the operands.

4.10.2 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 `UMAAL` instruction interprets the values from *Rn* and *Rm* as unsigned integers. It multiplies the two unsigned integers in the first and second operands and adds the unsigned 32-bit integer in *RdHi* to the 64-bit result of the multiplication. It adds the unsigned 32-bit integer in *RdLo* to the 64-bit result of the addition, writes the top 32-bits of the result to *RdHi*, and writes the lower 32-bits of the result to *RdLo*.

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*.

4.10.3 Restrictions

In these instructions:

- do not use SP and do not use PC
- *RdHi* and *RdLo* must be different registers.

4.10.4 Condition flags

These instructions do not affect the condition code flags.

4.10.5 Examples

```
UMULL       R0, R4, R5, R6   ; Unsigned (R4,R0) = R5 x R6
SMLAL       R4, R5, R3, R8   ; Signed (R5,R4) = (R5,R4) + R3 x R8
UMAAL       R3, R6, R2, R7   ; Multiplies R2 and R7, adds R6, adds R3, writes t
                             ; top 32 bits to R6, and the bottom 32 bits to R3.
```

4.11 SDIV and UDIV

Signed Divide and Unsigned Divide.

Applies to...	M3	M4	M4F
	✓	✓	✓

4.11.1 Syntax

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

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

where:

cond
Is an optional condition code. See Table 1-2 on page 33.

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.

4.11.2 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.

4.11.3 Restrictions

Do not use **SP** and do not use **PC**.

4.11.4 Condition Flags

These instructions do not change the flags.

4.11.5 Examples

```
SDIV  R0, R2, R4  ; Signed divide, R0 = R2/R4.
UDIV  R8, R8, R1  ; Unsigned divide, R8 = R8/R1.
```

5 Saturating Instructions

Table 5-1 on page 131 shows the saturating instructions:

Table 5-1. Saturating Instructions

Mnemonic	Brief Description	See Page
SSAT	Signed saturate	132
SSAT16	Signed saturate halfword	134
USAT	Unsigned saturate	132
USAT16	Unsigned saturate halfword	134
QADD	Saturating add	136
QADD8	Saturating add 8	136
QADD16	Saturating add 16	136
QSUB	Saturating subtract	136
QSUB8	Saturating subtract 8	136
QSUB16	Saturating subtract 16	136
QASX	Saturating add and subtract with exchange	138
QSAX	Saturating subtract and add with exchange	138
QDADD	Saturating double and add	140
QDSUB	Saturating double and subtract	140
UQADD16	Unsigned saturating add 16	144
UQADD8	Unsigned saturating add 8	144
UQASX	Unsigned saturating add and subtract with exchange	142
UQSAX	Unsigned saturating subtract and add with exchange	142
UQSUB16	Unsigned saturating subtract 16	144
UQSUB8	Unsigned saturating subtract 8	144

5.1 SSAT and USAT

Signed Saturate and Unsigned Saturate to any bit position, with optional shift before saturating.

Applies to...	M3	M4	M4F
	✓	✓	✓

5.1.1 Syntax

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

where:

op
Is one of:

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 Table 1-2 on page 33.

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.

5.1.2 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 $-2^{n-1} \le x \le 2^{n-1}-1$.

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

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 215.

To read the state of the Q flag, use the MRS instruction. See “MRS” on page 214.

5.1.3 Restrictions

Do not use **SP** and do not use **PC**.

5.1.4 Condition Flags

These instructions do not affect the condition code flags.

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

5.1.5 Examples

```
SSAT    R7, #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.
USATNE  R0, #7, R5           ; Conditionally saturate value in R5 as an
                             ; unsigned 7 bit value and write it to R0.
```

5.2 SSAT16 and USAT16

Signed Saturate and Unsigned Saturate to any bit position for two halfwords.

Applies to...	M3	M4	M4F
		✓	✓

5.2.1 Syntax

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

where:

op
is one of:

SSAT16
Saturates a signed halfword value to a signed range.

USAT16
Saturates a signed halfword value to an unsigned range.

cond
is an optional condition code, see Table 1-2 on page 33.

Rd
is the destination register.

n
specifies the bit position to saturate to:
- *n* ranges from 1 to 16 for SSAT.
- *n* ranges from 0 to 15 for USAT.

Rm
is the register containing the value to saturate.

5.2.2 Operation

The SSAT16 instruction:

1. Saturates two signed 16-bit halfword values of the register with the value to saturate from selected by the bit position in *n*.
2. Writes the results as two signed 16-bit halfwords to the destination register.

The USAT16 instruction:

1. Saturates two unsigned 16-bit halfword values of the register with the value to saturate from selected by the bit position in *n*.
2. Writes the results as two unsigned halfwords in the destination register.

5.2.3 Restrictions

Do not use SP and do not use PC.

5.2.4 Condition flags

These instructions do not affect the condition code flags.

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

5.2.5 Examples

```
SSAT16    R7, #9, R2       ; Saturates the top and bottom highwords of R2
                           ; as 9-bit values, writes to corresponding halfword
                           ; of R7
USAT16NE  R0, #13, R5      ; Conditionally saturates the top and bottom
                           ; halfwords of R5 as 13-bit values, writes to
                           ; corresponding halfword of R0
```

5.3 QADD and QSUB

Saturating Add and Saturating Subtract, signed.

Applies to...	M3	M4	M4F
		✓	✓

5.3.1 Syntax

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

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

where:

op
is one of:

`QADD`
Saturating 32-bit add.

`QADD8`
Saturating four 8-bit integer additions.

`QADD16`
Saturating two 16-bit integer additions.

`QSUB`
Saturating 32-bit subtraction.

`QSUB8`
Saturating four 8-bit integer subtraction.

`QSUB16`
Saturating two 16-bit integer subtraction.

cond
is an optional condition code, see Table 1-2 on page 33.

Rd
is the destination register.

Rn, Rm
are registers holding the first and second operands.

5.3.2 Operation

These instructions add or subtract two, four or eight values from the first and second operands and then writes a signed saturated value in the destination register.

The `QADD` and `QSUB` instructions apply the specified add or subtract, and then saturate the result to the signed range $-2^{n-1} \le x \le 2^{n-1}-1$, where x is given by the number of bits applied in the instruction, 32, 16 or 8.

If the returned result is different from the value to be saturated, it is called *saturation*. If saturation occurs, the QADD and QSUB instructions set the Q flag to 1 in the APSR. Otherwise, it leaves the Q flag unchanged. The 8-bit and 16-bit QADD and QSUB instructions always leave the Q flag unchanged.

To clear the Q flag to 0, you must use the MSR instruction, see 215.

To read the state of the Q flag, use the MRS instruction, see 214.

5.3.3 Restrictions

Do not use SP and do not use PC.

5.3.4 Condition flags

These instructions do not affect the condition code flags.

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

5.3.5 Examples

```
QADD16   R7, R4, R2  ; Adds halfwords of R4 with corresponding halfword of
                     ; R2, saturates to 16 bits and writes to corresponding
                     ; halfword of R7
QADD8    R3, R1, R6  ; Adds bytes of R1 to the corresponding bytes of R6,
                     ; saturates to 8 bits and writes to corresponding byte of
                     ; R3
QSUB16   R4, R2, R3  ; Subtracts halfwords of R3 from corresponding halfword
                     ; of R2, saturates to 16 bits, writes to corresponding
                     ; halfword of R4
QSUB8    R4, R2, R5  ; Subtracts bytes of R5 from the corresponding byte in
                     ; R2, saturates to 8 bits, writes to corresponding byte of
                     ; R4
```

5.4 QASX and QSAX

Saturating Add and Subtract with Exchange and Saturating Subtract and Add with Exchange, signed.

Applies to...	M3	M4	M4F
		✓	✓

5.4.1 Syntax

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

where:

op
is one of:

QASX
Add and Subtract with Exchange and Saturate.

QSAX
Subtract and Add with Exchange and Saturate.

cond
is an optional condition code, see Table 1-2 on page 33.

Rd
is the destination register.

Rn, *Rm*
are registers holding the first and second operands.

5.4.2 Operation

The QASX instruction:

1. Adds the top halfword of the source operand with the bottom halfword of the second operand.
2. Subtracts the top halfword of the second operand from the bottom highword of the first operand.
3. Saturates the result of the subtraction and writes a 16-bit signed integer in the range –215 ≤ x ≤ 215 – 1, where *x* equals 16, to the bottom halfword of the destination register.
4. Saturates the results of the sum and writes a 16-bit signed integer in the range –215 ≤ x ≤ 215 – 1, where *x* equals 16, to the top halfword of the destination register.

The QSAX instruction:

1. Subtracts the bottom halfword of the second operand from the top highword of the first operand.
2. Adds the bottom halfword of the source operand with the top halfword of the second operand.
3. Saturates the results of the sum and writes a 16-bit signed integer in the range –215 ≤ x ≤ 215 – 1, where *x* equals 16, to the bottom halfword of the destination register.
4. Saturates the result of the subtraction and writes a 16-bit signed integer in the range –215 ≤ x ≤ 215 – 1, where *x* equals 16, to the top halfword of the destination register.

5.4.3 Restrictions

Do not use SP and do not use PC.

5.4.4 Condition flags

These instructions do not affect the condition code flags.

5.4.5 Examples

```
QASX    R7, R4, R2      ; Adds top halfword of R4 to bottom halfword of R2,
                        ; saturates to 16 bits, writes to top halfword of R7
                        ; Subtracts top highword of R2 from bottom halfword of
                        ; R4, saturates to 16 bits and writes to bottom halfword
                        ; of R7
QSAX    R0, R3, R5      ; Subtracts bottom halfword of R5 from top halfword of
                        ; R3, saturates to 16 bits, writes to top halfword of R0
                        ; Adds bottom halfword of R3 to top halfword of R5,
                        ; saturates to 16 bits, writes to bottom halfword of R0
```

5.5 QDADD and QDSUB

Saturating Double and Add and Saturating Double and Subtract, signed.

Applies to...	M3	M4	M4F
		✓	✓

5.5.1 Syntax

op{*cond*} {*Rd*}, *Rm*, *Rn*

where:

op
is one of:

QDADD
Saturating Double and Add.

QDSUB
Saturating Double and Subtract.

cond
is an optional condition code, see Table 1-2 on page 33.

Rd
is the destination register.

Rm, *Rn*
are registers holding the first and second operands.

5.5.2 Operation

The QDADD instruction:

- Doubles the second operand value.
- Adds the result of the doubling to the signed saturated value in the first operand.
- Writes the result to the destination register.

The QDSUB instruction:

- Doubles the second operand value.
- Subtracts the doubled value from the signed saturated value in the first operand.
- Writes the result to the destination register.

Both the doubling and the addition or subtraction have their results saturated to the 32-bit signed integer range –231 ≤ x ≤ 231– 1. If saturation occurs in either operation, it sets the Q flag in the APSR.

5.5.3 Restrictions

Do not use SP and do not use PC.

5.5.4 Condition flags

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

5.5.5 Examples

```
QDADD     R7, R4, R2          ; Doubles and saturates R4 to 32 bits, adds R2,

                              ; saturates to 32 bits, writes to R7.

QDSUB     R0, R3, R5          ; Subtracts R3 doubled and saturated to 32 bits

                              ; from R5, saturates to 32 bits, writes to R0.
```

5.6 UQASX and UQSAX

Saturating Add and Subtract with Exchange and Saturating Subtract and Add with Exchange, unsigned.

Applies to...	M3	M4	M4F
		✓	✓

5.6.1 Syntax

op{*cond*} {*Rd*}, *Rm*, *Rn*

where:

type
is one of:

UQASX
Add and Subtract with Exchange and Saturate.

UQSAX
Subtract and Add with Exchange and Saturate.

cond
is an optional condition code, see Table 1-2 on page 33.

Rd
is the destination register.

Rn, *Rm*
are registers holding the first and second operands.

5.6.2 Operation

The UQASX instruction:

1. Adds the bottom halfword of the source operand with the top halfword of the second operand.
2. Subtracts the bottom halfword of the second operand from the top highword of the first operand.
3. Saturates the results of the sum and writes a 16-bit unsigned integer in the range $0 \le x \le 216 - 1$, where x equals 16, to the top halfword of the destination register.
4. Saturates the result of the subtraction and writes a 16-bit unsigned integer in the range $0 \le x \le 216 - 1$, where x equals 16, to the bottom halfword of the destination register.

The UQSAX instruction:

1. Subtracts the bottom halfword of the second operand from the top highword of the first operand.
2. Adds the bottom halfword of the first operand with the top halfword of the second operand.
3. Saturates the result of the subtraction and writes a 16-bit unsigned integer in the range $0 \le x \le 216 - 1$, where x equals 16, to the top halfword of the destination register.
4. Saturates the results of the addition and writes a 16-bit unsigned integer in the range $0 \le x \le 216 - 1$, where x equals 16, to the bottom halfword of the destination register.

5.6.3 Restrictions

Do not use SP and do not use PC.

5.6.4 Condition flags

These instructions do not affect the condition code flags.

5.6.5 Examples

```
UQASX   R7, R4, R2  ; Adds top halfword of R4 with bottom halfword of R2,
                    ; saturates to 16 bits, writes to top halfword of R7
                    ; Subtracts top halfword of R2 from bottom halfword of
                    ; R4, saturates to 16 bits, writes to bottom halfword of R7
UQSAX   R0, R3, R5  ; Subtracts bottom halfword of R5 from top halfword of R3,
                    ; saturates to 16 bits, writes to top halfword of R0
                    ; Adds bottom halfword of R4 to top halfword of R5
                    ; saturates to 16 bits, writes to bottom halfword of R0.
```

5.7 UQADD and UQSUB

Saturating Add and Saturating Subtract Unsigned.

Applies to...	M3	M4	M4F
		✓	✓

5.7.1 Syntax

op{*cond*} {*Rd*}, *Rn*, *Rm*

op{*cond*} {*Rd*}, *Rn*, *Rm*

where:

op
is one of:

UQADD8
Saturating four unsigned 8-bit integer additions.

UQADD16
Saturating two unsigned 16-bit integer additions.

UDSUB8
Saturating four unsigned 8-bit integer subtractions.

UQSUB16
Saturating two unsigned 16-bit integer subtractions.

cond
is an optional condition code, see Table 1-2 on page 33.

Rd
is the destination register.

Rn, Rm
are registers holding the first and second operands.

5.7.2 Operation

These instructions add or subtract two or four values and then writes an unsigned saturated value in the destination register.

The UQADD16 instruction:

- Adds the respective top and bottom halfwords of the first and second operands.
- Saturates the result of the additions for each halfword in the destination register to the unsigned range $0 \le x \le 2^{16}-1$, where x is 16.

The UQADD8 instruction:

- Adds each respective byte of the first and second operands.

- Saturates the result of the addition for each byte in the destination register to the unsigned range $0 \le x \le 2^8-1$, where x is 8.

The UQSUB16 instruction:

- Subtracts both halfwords of the second operand from the respective halfwords of the first operand.
- Saturates the result of the differences in the destination register to the unsigned range $0 \le x \le 2^{16}-1$, where x is 16.

The UQSUB8 instructions:

- Subtracts the respective bytes of the second operand from the respective bytes of the first operand.
- Saturates the results of the differences for each byte in the destination register to the unsigned range $0 \le x \le 2^8-1$, where x is 8.

5.7.3 Restrictions

Do not use SP and do not use PC.

5.7.4 Condition flags

These instructions do not affect the condition code flags.

5.7.5 Examples

```
UQADD16  R7, R4, R2   ; Adds halfwords in R4 to corresponding halfword in R2,
                      ; saturates to 16 bits, writes to corresponding halfword
                      ; of R7
UQADD8   R4, R2, R5   ; Adds bytes of R2 to corresponding byte of R5, saturates
                      ; to 8 bits, writes to corresponding bytes of R4
UQSUB16  R6, R3, R0   ; Subtracts halfwords in R0 from corresponding halfword
                      ; in R3, saturates to 16 bits, writes to corresponding
                      ; halfword in R6
UQSUB8   R1, R5, R6   ; Subtracts bytes in R6 from corresponding byte of R5,
                      ; saturates to 8 bits, writes to corresponding byte of R1
```

6 Packing and Unpacking Instructions

Table 6-1 on page 147 shows the packing and unpacking instructions:

Table 6-1. Packing and Unpacking Instructions

Mnemonic	Brief Description	See Page
PKH	Pack halfword	148
SXTAB	Extend 8 bits to 32 and add	154
SXTAB16	Dual extend 8 bits to 16 and add	154
SXTAH	Extend 16 bits to 32 and add	154
SXTB	Sign extend a byte	150
SXTB16	Dual extend 8 bits to 16 and add	150
SXTH	Sign extend a halfword	150
UXTAB	Extend 8 bits to 32 and add	154
UXTAB16	Dual extend 8 bits to 16 and add	154
UXTAH	Extend 16 bits to 32 and add	154
UXTB	Zero extend a byte	150
UXTB16	Dual zero extend 8 bits to 16 and add	150
UXTH	Zero extend a halfword	150

6.1 PKHBT and PKHTB

Pack Halfword.

Applies to...	M3	M4	M4F
	✓	✓	✓

6.1.1 Syntax

`op{cond} {Rd}, Rn, Rm {, LSL #imm}`

`op{cond} {Rd}, Rn, Rm {, ASR #imm}`

where:

op
is one of:

PKHBT
Pack Halfword, bottom and top with shift.

PKHTB
Pack Halfword, top and bottom with shift.

cond
is an optional condition code, see 33.

Rd
is the destination register.

Rn
is the first operand register.

Rm
is the second operand register holding the value to be optionally shifted.

imm
is the shift length. The type of shift length depends on the instruction:For PKHBT
:

LSL
a left shift with a shift length from 1 to 31, 0 means no shift.
For PKHTB:

ASR
an arithmetic shift right with a shift length from 1 to 32, a shift of 32-bits is encoded as `0b00000`.

6.1.2 Operation

The PKHBT instruction:

1. Writes the value of the bottom halfword of the first operand to the bottom halfword of the destination register.
2. If shifted, the shifted value of the second operand is written to the top halfword of the destination register.

The PKHTB instruction:

1. Writes the value of the top halfword of the first operand to the top halfword of the destination register.
2. If shifted, the shifted value of the second operand is written to the bottom halfword of the destination register.

6.1.3 Restrictions

Rd must not be SP and must not be PC.

6.1.4 Condition flags

This instruction does not change the flags.

6.1.5 Examples

```
PKHBT   R3, R4, R5 LSL #0  ; Writes bottom halfword of R4 to bottom halfword of

                           ; R3, writes top halfword of R5, unshifted, to top
                           ; halfword of R3

PKHTB   R4, R0, R2 ASR #1  ; Writes R2 shifted right by 1 bit to bottom halfword

                           ; of R4, and writes top halfword of R0 to top

                           ; halfword of R4
```

6.2 SXT and UXT

Sign extend and Zero extend.

Applies to...	M3	M4	M4F
	✓	✓	✓

6.2.1 Syntax

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

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

where:

op
is one of:

SXTB
Sign extends an 8-bit value to a 32-bit value.

SXTH
Sign extends a 16-bit value to a 32-bit value.

SXTB16
Sign extends two 8-bit values to two 16-bit values.

UXTB
Zero extends an 8-bit value to a 32-bit value.

UXTH
Zero extends a 16-bit value to a 32-bit value.

UXTB16
Zero extends two 8-bit values to two 16-bit values.

cond
is an optional condition code, see 33.

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.

6.2.2 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.
   - `SXTB16` extracts bits[7:0] and sign extends to 16 bits, and extracts bits [23:16] and sign extends to 16 bits.
   - `UXTB16` extracts bits[7:0] and zero extends to 16 bits, and extracts bits [23:16] and zero extends to 16 bits.

6.2.3 Restrictions

Do not use SP and do not use PC.

6.2.4 Condition flags

These instructions do not affect the flags.

6.2.5 Examples

```
SXTH  R4, R6, ROR #16  ; Rotates R6 right by 16 bits, obtains bottom halfword of
                       ; of result, sign extends to 32 bits and writes to R4
UXTB  R3, R10          ; Extracts lowest byte of value in R10, zero extends, and
                       ; writes to R3
```

6.3 SXTB16 and UXTB16

Sign extend and Zero extend byte 16.

Applies to...	M3	M4	M4F
		✓	✓

6.3.1 Syntax

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

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

where:

op
is one of:

SXTB16
Sign extends two 8-bit values to two 16-bit values.

UXTB16
Zero extends two 8-bit values to two 16-bit values.

cond
is an optional condition code, see 33.

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.

6.3.2 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:
   - SXTB16 extracts bits[7:0] and sign extends to 16 bits, and extracts bits [23:16] and sign extends to 16 bits.
   - UXTB16 extracts bits[7:0] and zero extends to 16 bits, and extracts bits [23:16] and zero extends to 16 bits.

6.3.3 Restrictions

Do not use SP and do not use PC.

6.3.4 Condition flags

These instructions do not affect the flags.

6.3.5 Examples

```
SXTH  R4, R6, ROR #16  ; Rotates R6 right by 16 bits, obtains bottom halfword of
                       ; of result, sign extends to 32 bits and writes to R4
UXTB  R3, R10          ; Extracts lowest byte of value in R10, zero extends, and
                       ; writes to R3
```

6.4 SXTA and UXTA

Signed and Unsigned Extend and Add.

Applies to...	M3	M4	M4F
		✓	✓

6.4.1 Syntax

op{*cond*} {*Rd*,} *Rn*, *Rm* {, ROR #*n*}

op{*cond*} {*Rd*,} *Rn*, *Rm* {, ROR #*n*}

where:

op
is one of:

SXTAB
Sign extends an 8-bit value to a 32-bit value and add.

SXTAH
Sign extends a 16-bit value to a 32-bit value and add.

SXTAB16
Sign extends two 8-bit values to two 16-bit values and add.

UXTAB
Zero extends an 8-bit value to a 32-bit value and add.

UXTAH
Zero extends a 16-bit value to a 32-bit value and add.

UXTAB16
Zero extends two 8-bit values to two 16-bit values and add.

cond
is an optional condition code, see 33.

Rd
is the destination register.

Rn
is the first operand register.

Rm
is the register holding the value to rotate and 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.

6.4.2 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:
   - SXTAB extracts bits[7:0] from *Rm* and sign extends to 32 bits.
   - UXTAB extracts bits[7:0] from *Rm* and zero extends to 32 bits.
   - SXTAH extracts bits[15:0] from *Rm* and sign extends to 32 bits.
   - UXTAH extracts bits[15:0] from *Rm* and zero extends to 32 bits.
   - SXTAB16 extracts bits[7:0] from *Rm* and sign extends to 16 bits, and extracts bits [23:16] from *Rm* and sign extends to 16 bits.
   - UXTAB16 extracts bits[7:0] from *Rm* and zero extends to 16 bits, and extracts bits [23:16] from *Rm* and zero extends to 16 bits.
3. Adds the signed or zero extended value to the word or corresponding halfword of *Rn* and writes the result in *Rd*.

6.4.3 Restrictions

Do not use SP and do not use PC.

6.4.4 Condition flags

These instructions do not affect the flags.

6.4.5 Examples

```
SXTAH  R4, R8, R6, ROR #16 ; Rotates R6 right by 16 bits, obtains bottom
                           ; halfword, sign extends to 32 bits, adds R8,and
                           ; writes to R4
UXTAB  R3, R4, R10         ; Extracts bottom byte of R10 and zero extends to 32
                           ; bits, adds R4, and writes to R3
```

7 Bitfield Instructions

Table 7-1 on page 156 shows the instructions that operate on adjacent sets of bits in registers or bitfields:

Table 7-1. Bitfield Instructions

Mnemonic	Brief Description	See Page
BFC	Bit field clear	157
BFI	Bit field insert	157
SBFX	Signed bit field extract	158
UBFX	Unsigned bit field extract	158

7.1 BFC and BFI

Bit Field Clear and Bit Field Insert.

Applies to...	M3	M4	M4F
	✓	✓	✓

7.1.1 Syntax

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

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

where:

cond
Is an optional condition code. See Table 1-2 on page 33.

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*.

7.1.2 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.

7.1.3 Restrictions

Do not use **SP** and do not use **PC**.

7.1.4 Condition Flags

These instructions do not affect the flags.

7.1.5 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.
```

7.2 SBFX and UBFX

Signed Bit Field Extract and Unsigned Bit Field Extract.

Applies to...	M3	M4	M4F
	✓	✓	✓

7.2.1 Syntax

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

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

where:

cond
Is an optional condition code. See Table 1-2 on page 33.

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*.

7.2.2 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.

7.2.3 Restrictions

Do not use **SP** and do not use **PC**.

7.2.4 Condition Flags

These instructions do not affect the flags.

7.2.5 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.
```

8 Floating-Point

Table 8-1 on page 159 shows the floating-point instructions:[1]

Table 8-1. Floating-Point Instructions

Mnemonic	Brief Description	See Page
VABS	Floating-point absolute	161
VADD	Floating-point add	162
VCMP	Compare two floating-point registers, or one floating-point register and zero	163
VCMPE	Compare two floating-point registers, or one floating-point register and zero with Invalid Operation check	163
VCVT	Convert between floating-point and integer	167
VCVT	Convert between floating-point and fixed point	167
VCVTR	Convert between floating-point and integer with rounding	167
VCVTB	Converts half-precision value to single-precision	169
VCVTT	Converts single-precision register to half-precision	169
VDIV	Floating-point divide	171
VFMA	Floating-point fused multiply accumulate	172
VFNMA	Floating-point fused negate multiply accumulate	173
VFMS	Floating-point fused multiply subtract	172
VFNMS	Floating-point fused negate multiply subtract	173
VLDM	Load multiple extension registers	174
VLDR	Loads an extension register from memory	176
VLMA	Floating-point multiply accumulate	178
VLMS	Floating-point multiply subtract	178
VMOV	Floating-point move immediate	179
VMOV	Floating-point move register	180
VMOV	Copy ARM core register to single precision	182
VMOV	Copy 2 ARM core registers to 2 single precision	183
VMOV	Copies between ARM core register to scalar	182
VMOV	Copies between scalar to ARM core register	182
VMRS	Move to ARM core register from floating-point system register	185
VMSR	Move to floating-point system register from ARM Core register	186
VMUL	Multiply floating-point	187
VNEG	Floating-point negate	188
VNMLA	Floating-point multiply and add	189
VNMLS	Floating-point multiply and subtract	189
VNMUL	Floating-point multiply	189
VPOP	Pop extension registers	191
VPUSH	Push extension registers	192
VSQRT	Floating-point square root	193

[1]These instructions are only available if the FPU is included, and enabled, in the system.

Table 8-1. Floating-Point Instructions *(continued)*

Mnemonic	Brief Description	See Page
VSTM	Store multiple extension registers	194
VSTR	Stores an extension register to memory	196
VSUB	Floating-point subtract	197

8.1 VABS

Floating-point Absolute.

Applies to...	M3	M4	M4F
			✓

8.1.1 Syntax

VABS{*cond*}.F32 *Sd*, *Sm*

where:

cond
is an optional condition code, see “Conditional Execution” on page 32.

Sd, *Sm*
are the destination floating-point value and the operand floating-point value.

8.1.2 Operation

This instruction:

1. Takes the absolute value of the operand floating-point register.
2. Places the results in the destination floating-point register.

8.1.3 Restrictions

There are no restrictions.

8.1.4 Condition flags

The floating-point instruction clears the sign bit.

8.1.5 Examples

```
VABS.F32 S4, S6
```

8.2 VADD

Floating-point Add.

Applies to...	M3	M4	M4F
			✓

8.2.1 Syntax

`VADD{`*`cond`*`}.F32 {`*`Sd`*`,} `*`Sn`*`, `*`Sm`*

where:

cond
is an optional condition code, see “Conditional Execution” on page 32.

Sd
is the destination floating-point value.

Sn, *Sm*
are the operand floating-point values.

8.2.2 Operation

This instruction:

1. Adds the values in the two floating-point operand registers.
2. Places the results in the destination floating-point register.

8.2.3 Restrictions

There are no restrictions.

8.2.4 Condition flags

This instruction does not change the flags.

8.2.5 Examples

```
VADD.F32 S4, S6, S7
```

8.3 VCMP, VCMPE

Compares two floating-point registers, or one floating-point register and zero.

Applies to...	M3	M4	M4F
			✓

8.3.1 Syntax

`VCMP{E}{cond}.F32 Sd, Sm`

`VCMP{E}{cond}.F32 Sd, #0.0`

where:

cond
is an optional condition code, see “Conditional Execution” on page 32.

E
If present, any `NaN` operand causes an `Invalid Operation` exception. Otherwise, only a signaling `NaN` causes the exception.

Sd
is the floating-point operand to compare.

Sm
is the floating-point operand that is compared with.

8.3.2 Operation

This instruction:

1. Compares:
   - Two floating-point registers.
   - One floating-point register and zero.
2. Writes the result to the `FPSCR` flags.

8.3.3 Restrictions

This instruction can optionally raise an `Invalid Operation` exception if either operand is any type of `NaN`. It always raises an `Invalid Operation` exception if either operand is a signaling `NaN`.

8.3.4 Condition flags

When this instruction writes the result to the `FPSCR` flags, the values are normally transferred to the ARM flags by a subsequent `VMRS` instruction, see “VMRS” on page 185.

8.3.5 Examples

```
VCMP.F32    S4, #0.0VCMP.F32    S4, S2
```

8.4 VCVT, VCVTR between floating-point and integer

Converts a value in a register from floating-point to a 32-bit integer.

Applies to...	M3	M4	M4F
			✓

8.4.1 Syntax

VCVT{*R*}{*cond*}.*Tm*.F32 *Sd, Sm*

VCVT{*cond*}.F32.*Tm Sd, Sm*

where:

R
If *R* is specified, the operation uses the rounding mode specified by the FPSCR. If *R* is omitted. the operation uses the `Round towards Zero` rounding mode.

cond
is an optional condition code, see “Conditional Execution” on page 32.

Tm
is the data type for the operand. It must be one of:

- S32 signed 32-bit value.
- U32 unsigned 32-bit value.

Sd, *Sm*
are the destination register and the operand register.

8.4.2 Operation

These instructions:

1. Either
   - Converts a value in a register from floating-point value to a 32-bit integer.
   - Converts from a 32-bit integer to floating-point value.
2. Places the result in a second register.

The floating-point to integer operation normally uses the `Round towards Zero` rounding mode, but can optionally use the rounding mode specified by the FPSCR.

The integer to floating-point operation uses the rounding mode specified by the FPSCR.

8.4.3 Restrictions

There are no restrictions.

8.4.4 Condition flags

These instructions do not change the flags.

8.5 VCVT between floating-point and fixed-point

Converts a value in a register from floating-point to and from fixed-point.

Applies to...	M3	M4	M4F
			✓

8.5.1 Syntax

`VCVT{`*`cond`*`}.`*`Td`*`.F32 `*`Sd`*`, `*`Sd`*`, #`*`fbits`*

`VCVT{`*`cond`*`}.F32.`*`Td Sd`*`, `*`Sd`*`, #`*`fbits`*

where:

cond
is an optional condition code, see “Conditional Execution” on page 32.

Td
is the data type for the fixed-point number. It must be one of:

- S16 signed 16-bit value.
- U16 unsigned 16-bit value.
- S32 signed 32-bit value.
- U32 unsigned 32-bit value.

Sd
is the destination register and the operand register.

fbits
is the number of fraction bits in the fixed-point number:

- If *Td* is S16 or U16, *fbits* must be in the range 0-16.
- I f *Td* is S32 or U32, *fbits* must be in the range 1-32.

8.5.2 Operation

These instructions:

1. Either
   - Converts a value in a register from floating-point to fixed-point.
   - Converts a value in a register from fixed-point to floating-point.
2. Places the result in a second register.

The floating-point values are single-precision.

The fixed-point value can be 16-bit or 32-bit. Conversions from fixed-point values take their operand from the low-order bits of the source register and ignore any remaining bits.

Signed conversions to fixed-point values sign-extend the result value to the destination register width.

Unsigned conversions to fixed-point values zero-extend the result value to the destination register width.

The floating-point to fixed-point operation uses the `Round towards Zero` rounding mode. The fixed-point to floating-point operation uses the `Round to Nearest` rounding mode.

8.5.3 Restrictions

There are no restrictions.

8.5.4 Condition flags

These instructions do not change the flags.

8.6 VCVTB, VCVTT

Converts between a half-precision value and a single-precision value.

Applies to...	M3	M4	M4F
			✓

8.6.1 Syntax

`VCVT{y}{cond}.F32.F16 Sd, Sm`

`VCVT{y}{cond}.F16.F32 Sd, Sm`

where:

y
Specifies which half of the operand register *Sm* or destination register *Sd* is used for the operand or destination:

- If y is `B`, then the bottom half, bits [15:0], of *Sm* or *Sd* is used.
- If *y* is `T`, then the top half, bits [31:16], of *Sm* or *Sd* is used.

cond
is an optional condition code, see “Conditional Execution” on page 32.

Sd
is the destination register.

Sm
is the operand register.

8.6.2 Operation

This instruction with the `.F16.32` suffix:

1. Converts the half-precision value in the top or bottom half of a single-precision. register to single-precision.
2. Writes the result to a single-precision register.

This instruction with the `.F32.F16` suffix:

1. Converts the value in a single-precision register to half-precision.
2. Writes the result into the top or bottom half of a single-precision register, preserving the other half of the target register.

8.6.3 Restrictions

There are no restrictions.

8.6.4 Condition flags

These instructions do not change the flags.

8.7 VDIV

Divides floating-point values.

Applies to...	M3	M4	M4F
			✓

8.7.1 Syntax

`VDIV{`*`cond`*`}.F32 {`*`Sd`*`,} `*`Sn`*`, `*`Sm`*

where:

cond
is an optional condition code, see “Conditional Execution” on page 32.

Sd
is the destination register.

Sn, *Sm*
are the operand registers.

8.7.2 Operation

This instruction:

1. Divides one floating-point value by another floating-point value.
2. Writes the result to the floating-point destination register.

8.7.3 Restrictions

There are no restrictions.

8.7.4 Condition flags

These instructions do not change the flags.

8.8 VFMA, VFMS

Floating-point Fused Multiply Accumulate and Subtract.

Applies to...	M3	M4	M4F
			✓

8.8.1 Syntax

`VFMA{`*`cond`*`}.F32 {`*`Sd,`*`} `*`Sn, Sm`*

`VFMS{`*`cond`*`}.F32 {`*`Sd,`*`} `*`Sn, Sm`*

where:

cond
is an optional condition code, see “Conditional Execution” on page 32.

Sd
is the destination register.

Sn, *Sm*
are the operand registers.

8.8.2 Operation

The VFMA instruction:

1. Multiplies the floating-point values in the operand registers.
2. Accumulates the results into the destination register.

The result of the multiply is not rounded before the accumulation.

The VFMS instruction:

1. Negates the first operand register.
2. Multiplies the floating-point values of the first and second operand registers.
3. Adds the products to the destination register.
4. Places the results in the destination register.

The result of the multiply is not rounded before the addition.

8.8.3 Restrictions

There are no restrictions.

8.8.4 Condition flags

These instructions do not change the flags.

8.9 VFNMA, VFNMS

Floating-point Fused Negate Multiply Accumulate and Subtract.

Applies to...	M3	M4	M4F
			✓

8.9.1 Syntax

`VFNMA{`*`cond`*`}.F32 {`*`Sd,`*`} `*`Sn, Sm`*

`VFNMS{`*`cond`*`}.F32 {`*`Sd,`*`} `*`Sn, Sm`*

where:

cond
is an optional condition code, see “Conditional Execution” on page 32.

Sd
is the destination register.

Sn, *Sm*
are the operand registers.

8.9.2 Operation

The VFNMA instruction:

1. Negates the first floating-point operand register.
2. Multiplies the first floating-point operand with second floating-point operand.
3. Adds the negation of the floating -point destination register to the product
4. Places the result into the destination register.

The result of the multiply is not rounded before the addition.

The VFNMS instruction:

1. Multiplies the first floating-point operand with second floating-point operand.
2. Adds the negation of the floating-point value in the destination register to the product.
3. Places the result in the destination register.

The result of the multiply is not rounded before the addition.

8.9.3 Restrictions

There are no restrictions.

8.9.4 Condition flags

These instructions do not change the flags.

8.10 VLDM

Floating-point Load Multiple.

Applies to...	M3	M4	M4F
			✓

8.10.1 Syntax

`VLDM{`*`mode`*`}{`*`cond`*`}{.`*`size`*`} `*`Rn`*`{`*`!`*`}, `*`list`*

where:

mode
is the addressing mode:

- *IA* Increment After. The consecutive addresses start at the address specified in *Rn*.
- *DB* Decrement Before. The consecutive addresses end just before the address specified in *Rn*.

cond
is an optional condition code, see “Conditional Execution” on page 32.

size
is an optional data size specifier.

Rn
is the base register. The SP can be used.

!
is the command to the instruction to write a modified value back to *Rn*. This is required if `mode == DB`, and is optional if `mode == IA`.

list
is the list of extension registers to be loaded, as a list of consecutively numbered doubleword or singleword registers, separated by commas and surrounded by brackets.

8.10.2 Operation

This instruction loads:

- Multiple extension registers from consecutive memory locations using an address from an ARM core register as the base address.

8.10.3 Restrictions

The restrictions are:

- If *size* is present, it must be equal to the size in bits, 32 or 64, of the registers in *list*.
- For the base address, the SP can be used. In the ARM instruction set, if *!* is not specified the PC can be used.

- *list* must contain at least one register. If it contains doubleword registers, it must not contain more than 16 registers.
- If using the `Decrement Before` addressing mode, the write back flag, *!*, must be appended to the base register specification.

8.10.4 Condition flags

These instructions do not change the flags.

8.11 VLDR

Loads a single extension register from memory.

Applies to...	M3	M4	M4F
			✓

8.11.1 Syntax

```
VLDR{cond}{.64} Dd, [Rn{#imm}]

VLDR{cond}{.64} Dd, label

VLDR{cond}{.64} Dd, [PC, #imm}]

VLDR{cond}{.32} Sd, [Rn {, #imm}]

VLDR{cond}{.32} Sd, label

VLDR{cond}{.32} Sd, [PC, #imm]
```

where:

cond
is an optional condition code, see “Conditional Execution” on page 32.

64, *32*
are the optional data size specifiers.

Dd
is the destination register for a doubleword load.

Sd
is the destination register for a singleword load.

Rn
is the base register. The SP can be used.

imm
is the + or - immediate offset used to form the address. Permitted address values are multiples of 4 in the range 0 to 1020.

label
is the label of the literal data item to be loaded.

8.11.2 Operation

This instruction:

- Loads a single extension register from memory, using a base address from an ARM core register, with an optional offset.

8.11.3 Restrictions

There are no restrictions.

8.11.4 Condition flags

These instructions do not change the flags.

8.12 VLMA, VLMS

Multiplies two floating-point values, and accumulates or subtracts the results.

Applies to...	M3	M4	M4F
			✓

8.12.1 Syntax

`VLMA{`*cond*`}.F32` *Sd*`,` *Sn*`,` *Sm*

`VLMS{`*cond*`}.F32` *Sd*`,` *Sn*`,` *Sm*

where:

cond
is an optional condition code, see “Conditional Execution” on page 32.

Sd
is the destination floating-point value.

Sn, *Sm*
are the operand floating-point values.

8.12.2 Operation

The floating-point Multiply Accumulate instruction:

1. Multiplies two floating-point values.
2. Adds the results to the destination floating-point value.

The floating-point Multiply Subtract instruction:

1. Multiplies two floating-point values.
2. Subtracts the products from the destination floating-point value.
3. Places the results in the destination register.

8.12.3 Restrictions

There are no restrictions.

8.12.4 Condition flags

These instructions do not change the flags.

8.13 VMOV Immediate

Move floating-point Immediate.

Applies to...	M3	M4	M4F
			✓

8.13.1 Syntax

`VMOV{`*`cond`*`}.F32 `*`Sd, #imm`*

where:

cond
is an optional condition code, see “Conditional Execution” on page 32.

Sd
is the branch destination.

imm
is a floating-point constant.

8.13.2 Operation

This instruction copies a constant value to a floating-point register.

8.13.3 Restrictions

There are no restrictions.

8.13.4 Condition flags

These instructions do not change the flags.

8.14 VMOV Register

Copies the contents of one register to another.

Applies to...	M3	M4	M4F
			✓

8.14.1 Syntax

VMOV{*cond*}.F64 *Dd, Dm*

VMOV{*cond*}.F32 *Sd, Sm*

where:

cond
is an optional condition code, see “Conditional Execution” on page 32.

Dd
is the destination register, for a doubleword operation.

Dm
is the source register, for a doubleword operation.

Sd
is the destination register, for a singleword operation.

Sm
is the source register, for a singleword operation.

8.14.2 Operation

This instruction copies the contents of one floating-point register to another.

8.14.3 Restrictions

There are no restrictions

8.14.4 Condition flags

These instructions do not change the flags.

8.15 VMOV Scalar to ARM Core register

Transfers one word of a doubleword floating-point register to an ARM core register.

Applies to...	M3	M4	M4F
			✓

8.15.1 Syntax

`VMOV{cond} Rt, Dn[x]`

where:

cond
is an optional condition code, see “Conditional Execution” on page 32.

Rt
is the destination ARM core register.

Dn
is the 64-bit doubleword register.

x
Specifies which half of the doubleword register to use:

- If *x* is 0, use lower half of doubleword register
- If *x* is 1, use upper half of doubleword register.

8.15.2 Operation

This instruction transfers:

- one word from the upper or lower half of a doubleword floating-point register to an ARM core register.

8.15.3 Restrictions

Rt cannot be PC or SP.

8.15.4 Condition flags

These instructions do not change the flags.

8.16 VMOV ARM Core register to single precision

Transfers a single-precision register to and from an ARM core register.

Applies to...	M3	M4	M4F
			✓

8.16.1 Syntax

`VMOV{cond} Sn, Rt`

`VMOV{cond} Rt, Sn`

where:

cond
is an optional condition code, see “Conditional Execution” on page 32.

Sn
is the single-precision floating-point register.

Rt
is the ARM core register.

8.16.2 Operation

This instruction transfers:

- The contents of a single-precision register to an ARM core register.
- The contents of an ARM core register to a single-precision register.

8.16.3 Restrictions

Rt cannot be PC or SP.

8.16.4 Condition flags

These instructions do not change the flags.

8.17 VMOV Two ARM Core registers to two single precision

Transfers two consecutively numbered single-precision registers to and from two ARM core registers.

Applies to...	M3	M4	M4F
			✓

8.17.1 Syntax

`VMOV{cond} Sm, Sm1, Rt, Rt2`

`VMOV{cond} Rt, Rt2, Sm, Sm`

where:

cond
is an optional condition code, see “Conditional Execution” on page 32.

Sm
is the first single-precision register.

Sm1
is the second single-precision register. This is the next single-precision register after *Sm*.

Rt
is the ARM core register that *Sm* is transferred to or from.

Rt2
is the The ARM core register that *Sm1* is transferred to or from.

8.17.2 Operation

This instruction transfers:

- The contents of two consecutively numbered single-precision registers to two ARM core registers.
- The contents of two ARM core registers to a pair of single-precision registers.

8.17.3 Restrictions

The restrictions are:

- The floating-point registers must be contiguous, one after the other.
- The ARM core registers do not have to be contiguous.
- *Rt* cannot be PC or SP.

8.17.4 Condition flags

These instructions do not change the flags.

8.18 VMOV ARM Core register to scalar

Transfers one word to a floating-point register from an ARM core register.

Applies to...	M3	M4	M4F
			✓

8.18.1 Syntax

VMOV{*cond*}{*.32*} *Dd[x], Rt*

where:

cond
is an optional condition code, see “Conditional Execution” on page 32.

32
is an optional data size specifier.

Dd[x]
is the destination, where [x] defines which half of the doubleword is transferred, as follows:

- If *x* is 0, the lower half is extracted
- If *x* is 1, the upper half is extracted.

Rt
is the source ARM core register.

8.18.2 Operation

This instruction transfers one word to the upper or lower half of a doubleword floating-point register from an ARM core register.

8.18.3 Restrictions

Rt cannot be PC or SP.

8.18.4 Condition flags

These instructions do not change the flags.

8.19 VMRS

Move to ARM Core register from floating-point System Register.

Applies to...	M3	M4	M4F
			✓

8.19.1 Syntax

VMRS{*cond*} *Rt*, FPSCR

VMRS{*cond*} *APSR_nzcv*, FPSCR

where:

cond
is an optional condition code, see “Conditional Execution” on page 32.

Rt
is the destination ARM core register. This register can be R0-R14.

APSR_nzcv
Transfer floating-point flags to the APSR flags.

8.19.2 Operation

This instruction performs one of the following actions:

- Copies the value of the FPSCR to a general-purpose register.
- Copies the value of the FPSCR flag bits to the APSR N, Z, C, and V flags.

8.19.3 Restrictions

Rt cannot be PC or SP.

8.19.4 Condition flags

These instructions optionally change the flags: N, Z, C, V

8.20 VMSR

Move to floating-point System Register from ARM Core register.

Applies to...	M3	M4	M4F
			✓

8.20.1 Syntax

VMSR{*cond*} FPSCR, *Rt*

where:

cond
is an optional condition code, see “Conditional Execution” on page 32.

Rt
is the general-purpose register to be transferred to the FPSCR.

8.20.2 Operation

This instruction moves the value of a general-purpose register to the FPSCR. See the Floating Point Status Control Register for more information.

8.20.3 Restrictions

The restrictions are:

- *Rt* cannot be PC or SP.

8.20.4 Condition flags

This instruction updates the FPSCR.

8.21 VMUL

Floating-point Multiply.

Applies to...	M3	M4	M4F
			✓

8.21.1 Syntax

`VMUL{`*cond*`}.F32 {`*Sd*`,} `*Sn*`, `*Sm*

where:

cond
is an optional condition code, see “Conditional Execution” on page 32.

Sd
is the destination floating-point value.

Sn, *Sm*
are the operand floating-point values.

8.21.2 Operation

This instruction:

1. Multiplies two floating-point values.
2. Places the results in the destination register.

8.21.3 Restrictions

There are no restrictions.

8.21.4 Condition flags

These instructions do not change the flags.

8.22 VNEG

Floating-point Negate.

Applies to...	M3	M4	M4F
			✓

8.22.1 Syntax

`VNEG{`*`cond`*`}.F32 `*`Sd, Sm`*

where:

cond
is an optional condition code, see “Conditional Execution” on page 32.

Sd
is the destination floating-point value.

Sm
is the operand floating-point value.

8.22.2 Operation

This instruction:

1. Negates a floating-point value.
2. Places the results in a second floating-point register.

The floating-point instruction inverts the sign bit.

8.22.3 Restrictions

There are no restrictions.

8.22.4 Condition flags

These instructions do not change the flags.

8.23 VNMLA, VNMLS, VNMUL

Floating-point multiply with negation followed by add or subtract.

Applies to...	M3	M4	M4F
			✓

8.23.1 Syntax

`VNMLA{`*cond*`}.F32` *Sd*, *Sn*, *Sm*

`VNMLS{`*cond*`}.F32` *Sd*, *Sn*, *Sm*

`VNMUL{`*cond*`}.F32 {`*Sd*`,}` *Sn*, *Sm*

where:

cond
is an optional condition code, see “Conditional Execution” on page 32.

Sd
is the destination floating-point register.

Sn, *Sm*
are the operand floating-point registers.

8.23.2 Operation

The `VNMLA` instruction:

1. Multiplies two floating-point register values.
2. Adds the negation of the floating-point value in the destination register to the negation of the product.
3. Writes the result back to the destination register.

The `VNMLS` instruction:

1. Multiplies two floating-point register values.
2. Adds the negation of the floating-point value in the destination register to the product.
3. writes the result back to the destination register.

The `VNMUL` instruction:

1. Multiplies together two floating-point register values.
2. Writes the negation of the result to the destination register.

8.23.3 Restrictions

There are no restrictions.

8.23.4 Condition flags

These instructions do not change the flags.

8.24 VPOP

Floating-point extension register Pop.

Applies to...	M3	M4	M4F
			✓

8.24.1 Syntax

`VPOP{`*`cond`*`}{.`*`size`*`} `*`list`*

where:

cond
is an optional condition code, see “Conditional Execution” on page 32.

size
is an optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers in *list*.

list
is a list of extension registers to be loaded, as a list of consecutively numbered doubleword or singleword registers, separated by commas and

surrounded by brackets.

8.24.2 Operation

This instruction loads multiple consecutive extension registers from the stack.

8.24.3 Restrictions

The list must contain at least one register, and not more than sixteen registers.

8.24.4 Condition flags

These instructions do not change the flags.

8.25 VPUSH

Floating-point extension register Push.

Applies to...	M3	M4	M4F
			✓

8.25.1 Syntax

VPUSH{*cond*}{.*size*} *list*

where:

cond
is an optional condition code, see “Conditional Execution” on page 32.

size
is an optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers in *list*.

list
is a list of the extension registers to be stored, as a list of consecutively numbered doubleword or singleword registers, separated by commas and surrounded by brackets.

8.25.2 Operation

This instruction:

- Stores multiple consecutive extension registers to the stack.

8.25.3 Restrictions

The restrictions are:

- *list* must contain at least one register, and not more than sixteen.

8.25.4 Condition flags

These instructions do not change the flags.

8.26 VSQRT

Floating-point Square Root.

Applies to...	M3	M4	M4F
			✓

8.26.1 Syntax

VSQRT{*cond*}.F32 *Sd*, *Sm*

where:

cond
is an optional condition code, see “Conditional Execution” on page 32.

Sd
is the destination floating-point value.

Sm
is the operand floating-point value.

8.26.2 Operation

This instruction:

- Calculates the square root of the value in a floating-point register.
- Writes the result to another floating-point register.

8.26.3 Restrictions

There are no restrictions.

8.26.4 Condition flags

These instructions do not change the flags.

8.27 VSTM

Floating-point Store Multiple.

Applies to...	M3	M4	M4F
			✓

8.27.1 Syntax

`VSTM{`*`mode`*`}{`*`cond`*`}{.`*`size`*`} `*`Rn`*`{!}, `*`list`*

where:

mode
is the addressing mode:

- IA *Increment After*. The consecutive addresses start at the address specified in *Rn*. This is the default and can be omitted.
- DB *Decrement Before*. The consecutive addresses end just before the address specified in *Rn*.

cond
is an optional condition code, see “Conditional Execution” on page 32.

size
is an optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of theregisters in *list*.

Rn
is the base register. The SP can be used.

!
is the function that causes the instruction to write a modified value back to *Rn*. Required if `mode == DB`.

list
is a list of the extension registers to be stored, as a list of consecutively numbered doubleword or singleword registers, separated by commas and surrounded by brackets.

8.27.2 Operation

This instruction:

- Stores multiple extension registers to consecutive memory locations using a base address from an ARM core register.

8.27.3 Restrictions

The restrictions are:

- *list* must contain at least one register. If it contains doubleword registers it must not contain more than 16 registers.

- Use of the PC as *Rn* is deprecated.

8.27.4 Condition flags

These instructions do not change the flags.

8.28 VSTR

Floating-point Store.

Applies to...	M3	M4	M4F
			✓

8.28.1 Syntax

VSTR{*cond*}{*.32*} *Sd*, [*Rn*{, *#imm*}]

VSTR{*cond*}{*.64*} *Dd*, [*Rn*{, *#imm*}]

where:

cond
is an optional condition code, see “Conditional Execution” on page 32.

32, *64*
are the optional data size specifiers.

Sd
is the source register for a singleword store.

Dd
is the source register for a doubleword store.

Rn
is the base register. The SP can be used.

imm
is the + or - immediate offset used to form the address. Values are multiples of 4 in the range 0-1020. *imm* can be omitted, meaning an offset of +0.

8.28.2 Operation

This instruction:

- Stores a single extension register to memory, using an address from an ARM core register, with an optional offset, defined in *imm*.

8.28.3 Restrictions

The restrictions are:

- The use of PC for *Rn* is deprecated.

8.28.4 Condition flags

These instructions do not change the flags.

8.29 VSUB

Floating-point Subtract.

Applies to...	M3	M4	M4F
			✓

8.29.1 Syntax

VSUB{*cond*}.F32 {*Sd*,} *Sn*, *Sm*

where:

cond
is an optional condition code, see “Conditional Execution” on page 32.

Sd
is the destination floating-point value.

Sn, *Sm*
are the operand floating-point value.

8.29.2 Operation

This instruction:

1. Subtracts one floating-point value from another floating-point value.
2. Places the results in the destination floating-point register.

8.29.3 Restrictions

There are no restrictions.

8.29.4 Condition flags

These instructions do not change the flags.

8.29.5 Operation

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

- a non-masked interrupt occurs and is taken
- an interrupt masked by PRIMASK becomes pending
- a Debug Entry request.

8.29.6 Condition flags

This instruction does not change the flags.

8.29.7 Examples

```
WFI ; Wait for interrupt
```

9 Branch and Control Instructions

Table 9-1 on page 199 shows the branch and control instructions:

Table 9-1. Branch and Control Instructions

Mnemonic	Brief Description	See Page
B	Branch	200
BL	Branch with link	200
BLX	Branch indirect with link	200
BX	Branch indirect	200
CBNZ	Compare and branch if non-zero	202
CBZ	Compare and branch if zero	202
IT	If-Then	203
TBB	Table branch byte	206
TBH	Table branch halfword	206

9.1 B, BL, BX, and BLX

Branch instructions.

Applies to...	M3	M4	M4F
	✓	✓	✓

9.1.1 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 Table 1-2 on page 33.

label
Is a PC-relative expression. See “PC-Relative Expressions” on page 32.

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.

9.1.2 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 the **Link Register (LR)**, register R14. See the *Stellaris® Data Sheet* for more on **LR**.
- 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 203.

Table 9-2 on page 201 shows the ranges for the various branch instructions.

Table 9-2. Branch Ranges

Instruction	Branch Range[a]
`B label`	−16 MB to +16 MB
`B`*cond* `label` (outside `IT` block)	−1 MB to +1 MB
`B`*cond* `label` (inside `IT` block)	−16 MB to +16 MB
`BL{`*cond*`} label`	−16 MB to +16 MB
`BX{`*cond*`} Rm`	Any value in register
`BLX{`*cond*`} Rm`	Any value in register

a. You might have to use the `.W` suffix to get the maximum branch range. See “Instruction Width Selection” on page 34.

9.1.3 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.

Note: `B` *cond* 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.

9.1.4 Condition Flags

These instructions do not change the flags.

9.1.5 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.
```

9.2 CBZ and CBNZ

Compare and Branch if Zero, Compare and Branch if Non-Zero.

Applies to...	M3	M4	M4F
	✓	✓	✓

9.2.1 Syntax

`CBZ` *`Rn, label`*

`CBNZ` *`Rn, label`*

where:

Rn
Is the register holding the operand.

label
Is the branch destination.

9.2.2 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
```

9.2.3 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.

9.2.4 Condition Flags

These instructions do not change the flags.

9.2.5 Examples

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

9.3 IT

If-Then.

Applies to...	M3	M4	M4F
	✓	✓	✓

9.3.1 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.

Note: 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`.

9.3.2 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.

Note: 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**. See the **PSR** register in the *Stellaris® Data Sheet* for more information.

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.

9.3.3 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 B*cond* must be inside an IT block. B*cond* 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 the logical inverse.

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

9.3.4 Condition Flags

This instruction does not change the flags.

9.3.5 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.
```

9.4 TBB and TBH

Table Branch Byte and Table Branch Halfword.

Applies to...	M3	M4	M4F
	✓	✓	✓

9.4.1 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 the **Program Counter (PC)** register, R15, 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.

9.4.2 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. 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.

9.4.3 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.

9.4.4 Condition Flags

These instructions do not change the flags.

9.4.5 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
```

10 Miscellaneous Instructions

Table 10-1 on page 208 shows the remaining Cortex-M3/M4F instructions:

Table 10-1. Miscellaneous Instructions

Mnemonic	Brief Description	See Page
BKPT	Breakpoint	209
CPSID	Change processor state, disable interrupts	210
CPSIE	Change processor state, enable interrupts	210
DMB	Data memory barrier	211
DSB	Data synchronization barrier	212
ISB	Instruction synchronization barrier	213
MRS	Move from special register to register	214
MSR	Move from register to special register	215
NOP	No operation	216
SEV	Send event	217
SVC	Supervisor call	218
WFE	Wait for event	219
WFI	Wait for interrupt	220

10.1 BKPT

Breakpoint.

Applies to...	M3	M4	M4F
	✓	✓	✓

10.1.1 Syntax

BKPT *#imm*

where:

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

10.1.2 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.

10.1.3 Condition Flags

This instruction does not change the flags.

10.1.4 Examples

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

10.2 CPS

Change Processor State.

Applies to...	M3	M4	M4F
	✓	✓	✓

10.2.1 Syntax

`CPS`*`effect 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 the **Priority Mask Register (PRIMASK)**.

`f`
Set or clear the **Fault Mask Register (FAULTMASK)**.

10.2.2 Operation

`CPS` changes the **PRIMASK** and **FAULTMASK** special register values. See the *Stellaris® Data Sheet* for more information about these registers.

10.2.3 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.

10.2.4 Condition Flags

This instruction does not change the flags.

10.2.5 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).
```

10.3 DMB

Data Memory Barrier.

Applies to...	M3	M4	M4F
	✓	✓	✓

10.3.1 Syntax

`DMB{`*cond*`}`

where:

cond
Is an optional condition code. See Table 1-2 on page 33.

10.3.2 Operation

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

10.3.3 Condition Flags

This instruction does not change the flags.

10.3.4 Examples

```
DMB  ; Data Memory Barrier
```

10.4 DSB

Data Synchronization Barrier.

Applies to...	M3	M4	M4F
	✓	✓	✓

10.4.1 Syntax

`DSB{`*cond*`}`

where:

cond
Is an optional condition code. See Table 1-2 on page 33.

10.4.2 Operation

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

10.4.3 Condition Flags

This instruction does not change the flags.

10.4.4 Examples

```
DSB ; Data Synchronization Barrier
```

10.5 ISB

Instruction Synchronization Barrier.

Applies to...	M3	M4	M4F
	✓	✓	✓

10.5.1 Syntax

`ISB{`*cond*`}`

where:

cond
Is an optional condition code. See Table 1-2 on page 33.

10.5.2 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.

10.5.3 Condition Flags

This instruction does not change the flags.

10.5.4 Examples

```
ISB  ; Instruction Synchronization Barrier
```

10.6 MRS

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

Applies to...	M3	M4	M4F
	✓	✓	✓

10.6.1 Syntax

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

where:

cond
Is an optional condition code. See Table 1-2 on page 33.

Rd
Is the destination register.

spec_reg
Can be any of the following special registers: **APSR**, **IPSR**, **EPSR**, **IEPSR**, **IAPSR**, **EAPSR**, **PSR**, **MSP**, **PSP**, **PRIMASK**, **BASEPRI**, **BASEPRI_MAX**, **FAULTMASK**, or **CONTROL**.

10.6.2 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.

Note: **`BASEPRI_MAX`** is an alias of **`BASEPRI`** when used with the `MRS` instruction.

See also “MSR” on page 215.

10.6.3 Restrictions

Rd must not be **SP** and must not be **PC**.

10.6.4 Condition Flags

This instruction does not change the flags.

10.6.5 Examples

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

10.7 MSR

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

Applies to...	M3	M4	M4F
	✓	✓	✓

10.7.1 Syntax

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

where:

cond
Is an optional condition code. See Table 1-2 on page 33.

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**.

10.7.2 Operation

The register access operation in MSR depends on the privilege level. Unprivileged software can only access the **Application Program Status Register (APSR)** (see **APSR** in the *Stellaris® Data Sheet*). Privileged software can access all special registers.

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

Note: 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 also “MRS” on page 214.

10.7.3 Restrictions

Rn must not be **SP** and must not be **PC**.

10.7.4 Condition Flags

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

10.7.5 Examples

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

10.8 NOP

No Operation.

Applies to...	M3	M4	M4F
	✓	✓	✓

10.8.1 Syntax

`NOP`{*cond*}

where:

cond
Is an optional condition code. See Table 1-2 on page 33.

10.8.2 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.

10.8.3 Condition Flags

This instruction does not change the flags.

10.8.4 Examples

```
NOP  ; No Operation
```

10.9 SEV

Send Event.

Applies to...	M3	M4	M4F
	✓	✓	✓

10.9.1 Syntax

SEV{*cond*}

where:

cond
Is an optional condition code. See Table 1-2 on page 33.

10.9.2 Operation

SEV is a hint instruction that causes an event to be signaled to all processors within a multiprocessor system. It also sets the one-bit event register to 1. See "Power Management" in the *Stellaris® Data Sheet*.

10.9.3 Condition Flags

This instruction does not change the flags.

10.9.4 Examples

```
SEV ; Send Event
```

10.10 SVC

Supervisor Call.

Applies to...	M3	M4	M4F
	✓	✓	✓

10.10.1 Syntax

SVC{*cond*} *#imm*

where:

cond
Is an optional condition code. See Table 1-2 on page 33.

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

10.10.2 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.

10.10.3 Condition Flags

This instruction does not change the flags.

10.10.4 Examples

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

10.11 WFE

Wait For Event.

Applies to...	M3	M4	M4F
	✓	✓	✓

10.11.1 Syntax

`WFE{`*cond*`}`

where:

cond
Is an optional condition code. See Table 1-2 on page 33.

10.11.2 Operation

`WFE` is a hint instruction.

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

- An exception, unless masked by the exception mask registers (**PRIMASK**, **FAULTMASK**, and **BASEPRI**) or the current priority level.
- An exception enters the Pending state, if `SEVONPEND` in the **System Control Register (SCR)** 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" in the *Stellaris® Data Sheet*.

10.11.3 Condition Flags

This instruction does not change the flags.

10.11.4 Examples

```
WFE  ; Wait for Event
```

10.12 WFI

Wait for Interrupt.

Applies to...	M3	M4	M4F
	✓	✓	✓

10.12.1 Syntax

`WFI{`*cond*`}`

where:

cond
Is an optional condition code. See Table 1-2 on page 33.

10.12.2 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.

10.12.3 Condition Flags

This instruction does not change the flags.

10.12.4 Examples

```
WFI ; Wait for Interrupt
```

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Audio	www.ti.com/audio	Communications and Telecom	www.ti.com/communications
Amplifiers	amplifier.ti.com	Computers and Peripherals	www.ti.com/computers
Data Converters	dataconverter.ti.com	Consumer Electronics	www.ti.com/consumer-apps
DLP® Products	www.dlp.com	Energy and Lighting	www.ti.com/energy
DSP	dsp.ti.com	Industrial	www.ti.com/industrial
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Security	www.ti.com/security
Logic	logic.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Power Mgmt	power.ti.com	Transportation and Automotive	www.ti.com/automotive
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Mobile Processors	www.ti.com/omap		
Wireless Connectivity	www.ti.com/wirelessconnectivity		
	TI E2E Community Home Page		e2e.ti.com

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265