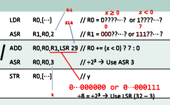
Quick reference card for ARMv7 (Cortex-M4)

Group	Operation	Syntax LSL	Semantic	Flags[1]
Arithmetic	Addition	`add{s}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}`	*Rd(n)* := *Rn* + *Rm*$^{\{shifted\}}$	NZCV
		`adc{s}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}`	*Rd(n)* := *Rn* + Rm$^{\{shifted\}}$ + C	NZCV
		`add{s}<c><q> {<Rd>,} <Rn>, #<const>`	*Rd(n)* := *Rn* + *const*	NZCV
		`adc{s}<c><q> {<Rd>,} <Rn>, #<const>`	*Rd(n)* := *Rn* + *const* + C	NZCV
		`qadd<c><q>    {<Rd>,} <Rn>, <Rm>`	*Rd(n)* := saturated (*Rn* + Rm)	Q
	Subtraction	`sub{s}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}`	*Rd(n)* := *Rn* - *Rm*$^{\{shifted\}}$	NZCV
		`sbc{s}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}`	*Rd(n)* := *Rn* - *Rm*$^{\{shifted\}}$ - not (C)	NZCV
		`rsb{s}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}`	*Rd(n)* := Rm$^{\{shifted\}}$ - *Rn* (÷2)+1 → LSR RD ← Multiplier × R1 Multiplier k=1 k=2 k=3 k=4 k=5	NZCV
		`sub{s}<c><q> {<Rd>,} <Rn>, #<const>`	*Rd(n)* := *Rn* - *const* ADD R0,R1,R1,LSL k $+(2^k+1)$ 3 5 9 17 33 LSL R0,R1,k $+2^k$ 2 4 8 16 32	NZCV
		`sbc{s}<c><q> {<Rd>,} <Rn>, #<const>`	*Rd(n)* := *Rn* - *const* - not (C) RSB R0,R1,R1,LSL k $+(2^k-1$ 1 3 7 15 31	NZCV
		`rsb{s}<c><q> {<Rd>,} <Rn>, #<const>`	*Rd(n)* := *const* - *Rn* SUB R0,R1,R1,LSL k $-(2^k-1)$ −1 −3 −7 −15 −31 MVN R0,R1,LSL k $-(2^k+1)$ −3 −5 −9 −17 −33	NZCV
		`qsub<c><q>    {<Rd>,} <Rn>, <Rm>`	*Rd(n)* := saturated (*Rn* - *Rm*) 32 ← 32x32 (single length) 64 ← 32x32 (double length) 64 ← 64x64 (single length)	Q
	Multiplication	`mul<c><q>     {<Rd>,} <Rn>, <Rm>`	*Rd(n)* := (*Rn*Rm*) MUL R3,R1,R2 UMULL R2,R3,R0,R1 (unsigned) MUL R1,R1,R2 MLA R1,R0,R3,R1	
		`mla<c>          <Rd>,  <Rn>, <Rm>, <Ra>`	*Rd* := *Ra* + (*Rn*Rm*) (signed or unsigned) SMULL R2,R3,R0,R1 (signed) UMULL R1,R2,R0,R2 ADD R1,R1,R2 (signed or unsigned)	
		`mls<c>          <Rd>,  <Rn>, <Rm>, <Ra>`	*Rd* := *Ra* - (*Rn*Rm*) R3←R1 × R2 R3.R2←R0 × R1 R1.R0←R1.R0 × R3.R2	
		`umull<c>     <RdLo>, <RdHi>, <Rn>, <Rm>`	*RdHi:RdLo* := unsigned_64_bit (*Rn*Rm*)	
		`umlal<c><q> <RdLo>, <RdHi>, <Rn>, <Rm>`	*RdHi:RdLo* := unsigned_64_bit (*RdHi:RdLo* + (*Rn*Rm*))	
		`smull<c>     <RdLo>, <RdHi>, <Rn>, <Rm>`	*RdHi:RdLo* := signed_64_bit (*Rn*Rm*) LDR R0,[…] // R0 <-- x LDR R1,=42949673 // 2^{32}/100 = 42949673 (Rounded) SMULL R0,R1,R0,R1 // R1.R0 = 64-bit product STR R1,[…] // y = x/100 (approx.)	
		`smlal<c>     <RdLo>, <RdHi>, <Rn>, <Rm>`	*RdHi:RdLo* := signed_64_bit (*RdHi:RdLo* + (*Rn*Rm*))	
	Division	`udiv<c>         <Rd>,  <Rn>, <Rm>`	*Rd* := unsigned_32_bit (*Rn/Rm*); rounded towards 0	
		`sdiv<c>         <Rd>,  <Rn>, <Rm>`	*Rd* := signed_32_bit (*Rn/Rm*); rounded towards 0	
Bit operations	Logic	`and{s}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}`	*Rd(n)* := *Rn* ∧ *Rm*$^{\{shifted\}}$ Bit Field Clear BFC R_d,lsb,width R_d<bits> ← 0 Bit Field Insert BFI R_d,R_n,lsb,width R_d<bits> ← R_n<lsb's>	NZCV
		`bic{s}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}`	*Rd(n)* := *Rn* ∧ ¬*Rm*$^{\{shifted\}}$ Signed Bit Field Extract SBFX R_d,R_n,lsb,width R_d ← R_n<bits> Sign extends	NZCV
		`orr{s}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}`	*Rd(n)* := *Rn* ∨ *Rm*$^{\{shifted\}}$ Unsigned Bit Field Extract UBFX R_d,R_n,lsb,width R_d ← R_n<bits> Zero extends	NZCV
		`orn{s}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}`	*Rd(n)* := *Rn* ∨ ¬*Rm*$^{\{shifted\}}$	NZCV
		`eor{s}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}`	*Rd(n)* := *Rn* ⊕ *Rm*$^{\{shifted\}}$	NZCV
		`and{s}<c><q> {<Rd>,} <Rn>, #<const>`	*Rd(n)* := *Rn* ∧ *const*	NZCV
		`bic{s}<c><q> {<Rd>,} <Rn>, #<const>`	*Rd(n)* := *Rn* ∧ ¬*const*	NZCV
		`orr{s}<c><q> {<Rd>,} <Rn>, #<const>`	*Rd(n)* := *Rn* ∨ *const*	NZCV
		`orn{s}<c><q> {<Rd>,} <Rn>, #<const>`	*Rd(n)* := *Rn* ∨ ¬*const*	NZCV
		`eor{s}<c><q> {<Rd>,} <Rn>, #<const>`	*Rd(n)* := *Rn* ⊕ *const*	NZCV
	Tests	`cmp<c><q> <Rn>, <Rm> {,<shift>}`	*Rn* - *Rm*$^{\{shifted\}}$	NZCV
		`cmn<c><q> <Rn>, <Rm> {,<shift>}`	*Rn* + *Rm*$^{\{shifted\}}$	NZCV
		`tst<c><q> <Rn>, <Rm> {,<shift>}`	*Rn* ∧ *Rm*$^{\{shifted\}}$	NZCV
		`teq<c><q> <Rn>, <Rm> {,<shift>}`	*Rn* ⊕ *Rm*$^{\{shifted\}}$	NZCV
		`cmp<c><q> <Rn>, #<const>`	*Rn* - *const*	NZCV
		`cmn<c><q> <Rn>, #<const>`	*Rn* + *const* ADD for neg	NZCV
		`tst<c><q> <Rn>, #<const>`	*Rn* ∧ *const* by shift	NZCV
		`teq<c><q> <Rn>, #<const>`	*Rn* ⊕ *const* 16863_{10} = 0100 0001 1101 1111_2 = A + 0000 0000 0100 0001 + (2^6 + 2^0)	NZCV
Register moves	Move	`mov{s}<c><q> <Rd>, <Rm>`	*Rd* := *Rm* = 0100 0010 0010 0000 = $2^{14} + 2^9 + 2^5$ − 0000 0000 0100 0001 - (2^6 + 2^0)	NZ
		`mov{s}<c><q> <Rd>, #<const>`	*Rd* := *const* = $2^{14} + 2^9 + 2^5 - (2^6 + 2^0)$ = A = $2^{14} + 2^9 - 2^6 + 2^5 - 2^0$ = A	NZC
	Shift / Rotate	`lsr{s}<c><q> <Rd>, <Rm>, #<n>`	*Rd* := *Rm*$^{\{shifted-right by <n>\}}$; filled with 0's, C := last shifted-out	NZC
		`lsr{s}<c><q> <Rd>, <Rm>, <Rs>`	*Rd* := *Rm*$^{\{shifted-right by Rs\}}$; filled with 0's, C := last shifted-out	NZC
		`asr{s}<c><q> <Rd>, <Rm>, #<n>`	*Rd* := *Rm*$^{\{shifted-right by <n>\}}$; filled with MSB[2], C := last shifted-out	NZC
		`asr{s}<c><q> <Rd>, <Rm>, <Rs>`	*Rd* := *Rm*$^{\{shifted-right by Rs\}}$; filled with MSB[2], C := last shifted-out	NZC
		`lsl{s}<c><q> <Rd>, <Rm>, #<n>`	*Rd* := *Rm*$^{\{shifted-left by <n>\}}$; filled with 0's, C := last shifted-out	NZC
		`lsl{s}<c><q> <Rd>, <Rm>, <Rs>`	*Rd* := *Rm*$^{\{shifted-left by Rs\}}$; filled with 0's, C := last shifted-out	NZC
		`ror{s}<c><q> <Rd>, <Rm>, #<n>`	*Rd* := *Rm*$^{\{rotated-right by <n>\}}$; C := MSB[2] of result	NZC
		`ror{s}<c><q> <Rd>, <Rm>, <Rs>`	*Rd* := *Rm*$^{\{rotated-right by Rs\}}$; C := MSB[2] of result	NZC
		`rrx{s}<c><q> <Rd>, <Rm>`	*Rd* := *Rm*$^{\{rotated-right by 1 including carry bit\}}$	NZC
Load & Store	Offset	`ldr<c><q> <Rd>, [<Rb> {, #+/-<offset>}]`	*Rd* := [*Rb* ± *offset*] SBFX/BFX Dest, R0, lsb, width	
		`str<c><q> <Rs>, [<Rb> {, #+/-<offset>}]`	[*Rb* ± *offset*] := *Rs*	
	Pre-offset	`ldr<c><q> <Rd>, [<Rb>, #+/-<offset>]!`	*Rb* := *Rb* ± *offset; Rd* := [*Rb*];	
		`str<c><q> <Rs>, [<Rb>, #+/-<offset>]!`	*Rb* := *Rb* ± *offset;* [*Rb*] := *Rs;*	
	Post-offset	`ldr<c><q> <Rd>, [<Rb>], #+/-<offset>`	*Rd* := [*Rb*]; *Rb* := *Rb* ± *offset*	
		`str<c><q> <Rs>, [<Rb>], #+/-<offset>`	[*Rb*] := *Rs; Rb* := *Rb* ± *offset*	
	Indexed	`ldr<c><q> <Rd>, [<Rb>, <Ri> {, lsl #<shift>}]`	*Rd* := [*Rb* + *Ri*$^{\{shifted- left\}}$]	
		`str<c><q> <Rs>, [<Rb>, <Ri> {, lsl #<shift>}]`	[*Rb* + *Ri*$^{\{shifted- left\}}$] := *Rs*	
	Literal	`ldr<c><q> <Rd>, <label>`	*Rd* := [label]	
		`ldr<c><q> <Rd>, [PC, #+/-<offset>]`	*Rd* := [*PC* ± *offset*]	
	Positive stack	`stmia<c><q> <Rs>!, <registers>`	for *Ri* in *registers*: [*Rs*] := *Ri; Rs* := *Rs* + 4	
		`ldmdb<c><q> <Rs>!, <registers>`	for *Ri* in reverse *registers*: *Rs* := *Rs* - 4; Ri := [*Rs*]	
	Negative stack	`stmdb<c><q> <Rs>!, <registers>`	for *Ri* in reverse *registers*: *Rs* := *Rs* - 4; [*Rs*] := *Ri*	
		`ldmia<c><q> <Rs>!, <registers>`	for *Ri* in *registers*: *Ri* := [*Rs*]; *Rs* := *Rs* + 4	

Group	Operation	Syntax	Semantic	Flags[1]
Branch	Branch on flags	`b<c><q> <label>`	if c then *PC* := *label*	
		`bl<c>    <label>`	if c then *LR* := *PC_next*; *PC* := *label*	
		`bx<c>       <Rm>`	if c then *PC* := *Rm*	
		`blx<c><q> <Rm>`	if c then LR := *PC_next*; *PC* := *Rm*	
	Test & branch	`cbz<q>  <Rn>, <label>` ← location	if *Rn* = 0 then *PC* := *label*	
		`cbnz<q> <Rn>, <label>`	if *Rn* ≠ 0 then *PC* := *label*	
	Table based	`tbb<c><q> [<Rn>, <Rm>]`	branch to [PC + *Rm*'s byte in the table starting at *Rn*)];	
		`tbh<c><q> [<Rn>, <Rm>, lsl #1]`	branch to [PC + *Rm*'s halfword in the table starting at *Rn*)];	
Synchronization		`ldrex<c><q> <Rt>, [<Rn> {,#<offset>}]`	*Rt* := [*Rn* + *offset*]; mark (*Rn* + *offset*) as exclusive memory	
		`strex<c><q> <Rd>, <Rt>, [<Rn> {,#<offset>}]`	if exclusive then [*Rn* + *offset*] := *Rt*; *Rd* := 0 else *Rd* := 1	



```
LDR   R0,[···]            // R0 = 0????···? or 1????···?
ASR   R1,R0,2             // R1 = 000??···? or 111??··· ?
ADD   R0,R0,R1,LSR 29     // R0 += (x < 0) ? 7 : 0
ASR   R0,R0,3             // ÷2³ → Use ASR 3
STR   R0,[···]            // y
                          0···000000 or 0···000111
                          ÷8 = ÷2³ → Use LSR (32 – 3)
```

Flags

Flag	Meaning	Calculated as
`N`	Negative	MSB[2] (Result) = 1
`Z`	Zero	Result = 0
`C`	Carry	Depends on instruction
`V`	Overflow	Signed overflow
`Q`	Saturated	Signed overflow (result saturated)

Opcode size

`<q>`	Meaning
`.N`	Narrow code (16 bit)
`.W`	Wide code (32 bit)
`<omit>`	Let the assembler choose

Shift options

`<shift>`	Meaning
`<omit>`	no shifts or rotations, equivalent to LSL #0
`LSL #<n>`	logical shift left by *<n>* bits, 0 ≤ *n* ≤ 31
`LSR #<n>`	logical shift right by *<n>* bits, 1 ≤ *n* ≤ 32
`ASR #<n>`	arithmetic shift right by *<n>* bits, 1 ≤ *n* ≤ 32
`ROR #<n>`	rotate right by *<n>* bits, 1 ≤ *n* ≤ 31
`RRX`	rotate right by 1 bit through carry flag

Condition codes

`<c>`	Meanings	Flags
`eq`	Equal	`Z = 1`
`ne`	Not equal	`Z = 0`
`cs, hs`	Carry set, Unsigned higher or same	`C = 1`
`cc, lo`	Carry clear, Unsigned lower	`C = 0`
`mi`	Minus, Negative	`N = 1`
`pl`	Plus, Positive or zero	`N = 0`
`vs`	Overflow	`V = 1`
`vc`	No overflow	`V = 0`
`hi`	Unsigned higher	`C = 1 ∧ Z = 0`
`ls`	Unsigned lower or same	`C = 0 ∨ Z = 1`
`ge`	Signed greater or equal	`N = V`
`lt`	Signed less	`N ≠ V`
`gt`	Signed greater	`Z = 0 ∧ N = V`
`le`	Signed less or equal	`Z = 1 ∨ N ≠ V`
`al, <omit>`	Always	`any`

1 If the instruction can be amended by adding an "s" to it, you can choose whether it will set flags or not ("s" means to set flags). If the instruction does not provide this option, then the indicated flags are always set. Other flags are untouched.

2 MSB: Most Significant Bit (left-most bit, which also indicates the sign for a signed integer type)