Quick reference card for ARMv7 (Cortex-M4)

Group	Operation	Syntax	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*	NZCV
		`sub{s}<c><q> {<Rd>,} <Rn>, #<const>`	*Rd(n)* := *Rn* - *const*	NZCV
		`sbc{s}<c><q> {<Rd>,} <Rn>, #<const>`	*Rd(n)* := *Rn* - *const* - not (C)	NZCV
		`rsb{s}<c><q> {<Rd>,} <Rn>, #<const>`	*Rd(n)* := *const* - *Rn*	NZCV
		`qsub<c><q> {<Rd>,} <Rn>, <Rm>`	*Rd(n)* := saturated (*Rn* - *Rm*)	Q
	Multiplication	`mul<c><q> {<Rd>,} <Rn>, <Rm>`	*Rd(n)* := (*Rn***Rm*)	
		`mla<c> <Rd>, <Rn>, <Rm>, <Ra>`	*Rd* := *Ra* + (*Rn***Rm*)	
		`mls<c> <Rd>, <Rn>, <Rm>, <Ra>`	*Rd* := *Ra* - (*Rn***Rm*)	
		`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*)	
		`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}	NZCV
		`bic{s}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}`	*Rd(n)* := *Rn* ∧ ¬*Rm*{shifted}	NZCV
		`orr{s}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}`	*Rd(n)* := *Rn* ∨ *Rm*{shifted}	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*	NZCV
		`tst<c><q> <Rn>, #<const>`	*Rn* ∧ *const*	NZCV
		`teq<c><q> <Rn>, #<const>`	*Rn* ⊕ *const*	NZCV
Register moves	Move	`mov{s}<c><q> <Rd>, <Rm>`	*Rd* := *Rm*	NZ
		`mov{s}<c><q> <Rd>, #<const>`	*Rd* := *const*	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*]	
		`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>`	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	

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, $0 \leq n \leq 31$
LSR #<n>	logical shift right by <*n*> bits, $1 \leq n \leq 32$
ASR #<n>	arithmetic shift right by <*n*> bits, $1 \leq n \leq 32$
ROR #<n>	rotate right by <*n*> bits, $1 \leq n \leq 31$
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)