ARM Instructions

Data movement			
Arithmetic	ADD*cd*S[†]	*reg*, *reg*, *arg*	Âăadd
	SUB*cd*S	*reg*, *reg*, *arg*	Âăsubtract
	RSB*cd*S	*reg*, *reg*, *arg*	Âăsubtract reversed operands
	ADC*cd*S	*reg*, *reg*, *arg*	Âăadd both operands and carry flag
	SBC*cd*S	*reg*, *reg*, *arg*	Âăsubtract both operands and adds carry flag − 1
	RSC*cd*S	*reg*, *reg*, *arg*	Âăreverse subtract both operands and adds carry flag − 1
	MUL*cd*S	reg_d, reg_m, reg_s	Âămultiply reg_m and reg_s, places lower 32 bits into reg_d
	MLA*cd*S	reg_d, reg_m, reg_s, reg_n	Âăplaces lower 32 bits of reg_m Âů reg_s + reg_n into reg_d
	UMULL*cd*S	reg_{lo}, reg_{hi}, reg_m, reg_s	Âămultiply reg_m and reg_s place 64-bit unsigned result into $\{reg_{hi}, reg_{lo}\}$
	UMLAL*cd*S	reg_{lo}, reg_{hi}, reg_m, reg_s	Âăplace unsigned $reg_m \cdot reg_s + \{reg_{hi}, reg_{lo}\}$ into $\{reg_{hi}, reg_{lo}\}$
	SMULL*cd*S	reg_{lo}, reg_{hi}, reg_m, reg_s	Âămultiply reg_m and reg_s, place 64-bit signed result into $\{reg_{hi}, reg_{lo}\}$
	SMLAL*cd*S	reg_{lo}, reg_{hi}, reg_m, reg_s	Âăplace signed $reg_m \cdot reg_s + \{reg_{hi}, reg_{lo}\}$ into $\{reg_{hi}, reg_{lo}\}$
Bitwise logic	AND*cd*S	*reg*, *reg*, *arg*	Âăbitwise AND
	ORR*cd*S	*reg*, *reg*, *arg*	Âăbitwise OR
	EOR*cd*S	*reg*, *reg*, *arg*	Âăbitwise exclusive-OR
	BIC*cd*S	*reg*, reg_a, arg_b	Âăbitwise reg_a AND (NOT arg_b)
Comparison	CMP*cd*	*reg*, *arg*	Âăupdate flags based on subtraction
	CMN*cd*	*reg*, *arg*	Âăupdate flags based on addition
	TST*cd*	*reg*, *arg*	Âăupdate flags based on bitwise AND
	TEQ*cd*	*reg*, *arg*	Âăupdate flags based on bitwise exclusive-OR
Data movement	MOV*cd*S	*reg*, *arg*	Âăcopy argument
	MVN*cd*S	*reg*, *arg*	Âăcopy bitwise NOT of argument
Memory access	LDR*cd*B[‡]	*reg*, *mem*	Âăloads word/ byte/ half from memory into a register
	STR*cd*B	*reg*, *mem*	Âăstores word/ byte/ half to memory from a register
	LDM*cdum*	*reg*!, *mreg*	Âăloads into multiple registers
	STM*cdum*	*reg*!, *mreg*	Âăstores multiple registers
	SWP*cd*B	reg_d, reg_m, $[reg_n]$	Âăcopies reg_m to memory at reg_n, old value at address reg_n to reg_d
Branching	B*cd*	imm_{24}	Âăbranch to imm_{24} words away
	BL*cd*	imm_{24}	Âăcopy PC to LR, then branch
	BX*cd*	*reg*	Âăcopy *reg* to PC, and exchange instruction sets (T flag := *reg*[0])
	SWI*cd*	imm_{24}	Âăsoftware interrupt

[†] S = set condition flags [‡] B = byte, can be replaced by H for half word(2 bytes)

cd: condition code

AL or omitted	always	(ignored)
EQ	equal	Z = 1
NE	not equal	Z = 0
CS	carry set (same as HS)	C = 1
CC	carry clear (same as LO)	C = 0
MI	minus	N = 1
PL	positive or zero	N = 0
VS	overflow	V = 1
VC	no overflow	V = 0
HS	unsigned higher or same	C = 1
LO	unsigned lower	C = 0
HI	unsigned higher	C = 1 ∧ Z = 0
LS	unsigned lower or same	C = 0 ∨ Z = 1
GE	signed greater than or equal	N = V
LT	signed less than	N ≠ V
GT	signed greater than	Z = 0 ∧ N = V
LE	signed less than or equal	Z = 1 ∨ N ≠ V

um: update mode

FA / IB	ascending, starting from *reg*
EA / IA	ascending, starting from *reg* + 4
FD / DB	descending, starting from *reg*
ED / DA	descending, starting from *reg* − 4

reg: register

R0 to R15	register according to number
SP	register 13
LR	register 14
PC	register 15

arg: right-hand argument

#imm_8	immediate on 8 bits, possibly rotated right
reg	register
reg, *shift*	register shifted by distance

shift: shift register value

LSL #imm_5	shift left 0 to 31
LSR #imm_5	logical shift right 1 to 32
ASR #imm_5	arithmetic shift right 1 to 32
ROR #imm_5	rotate right 1 to 31
RRX	rotate carry bit into top bit
LSL *reg*	shift left by register
LSR *reg*	logical shift right by register
ASR *reg*	arithmetic shift right by register
ROR *reg*	rotate right by register

mem: memory address

[*reg*, #±imm_{12}]	*reg* offset by constant
[*reg*, ±*reg*]	*reg* offset by variable bytes
[reg_a, ±reg_b, *shift*]	reg_a offset by shifted variable reg_b [†]
[*reg*, #±imm_{12}]!	update *reg* by constant, then access memory
[*reg*, ±*reg*]!	update *reg* by variable bytes, then access memory
[*reg*, ±*reg*, *shift*]!	update *reg* by shifted variable, then access memory [†]
[*reg*], #±imm_{12}	access address *reg*, then update *reg* by offset
[*reg*], ±*reg*	access address *reg*, then update *reg* by variable
[*reg*], ±*reg*, *shift*	access address *reg*, then update *reg* by shifted variable [†]

[†] shift distance must be by constant