ARM Cortex-M0 Instruction Set

The assembler syntax is —here, items inside ( ) are optional. <label> must be placed at column 1. <MNEMONIC> must be placed **strictly** after column 1—:

```
(<label>)                                                    (<;><comment>)
        <MNEMONIC>      (<operand1>(, <operand2>(, <operand3>)))         (<;><comment>)
```

<Rx>, with x any letter, means a general purpose register (e.g.: R0, R1, …). Some instructions pose restrictions on the value of x. e.g.: 0≤x≤7

immedX means an immediate X-bit operand. All immediate operands are zero extended when operated with longer operands

<MNEMONIC>	<operand1>	<operand2>	<operand3>	flags	description
Data movement inside the processor					
MOV	<Rd>,	<Rm>			<Rd> ← <Rm>
MOVS	<Rd>,	<Rm>		NZ	<Rd> ← <Rm> 0≤{d,m}≤7
MOVS	<Rd>,	#immed8		NZ	<Rd> ← immed8 0≤d≤7
Memory access					
Load data from memory					
LDR	<Rt>,	[<Rn>]			<Rt> ← word_at[<Rn>] 0≤{t,n}≤7
LDR	<Rt>,	[<Rn>, #immed7]			<Rt> ← word_at[<Rn> + immed7]. immed7 = 4·k 0≤{t,n}≤7
LDR	<Rt>,	[<Rn>, <Rm>]			<Rt> ← word_at[<Rn> + <Rm>]
LDRH	<Rt>,	[<Rn>]			<Rt> ← extend_0(half_word_at[<Rn>]) 0≤{t,n}≤7
LDRH	<Rt>,	[<Rn>, #immed6]			<Rt> ← extend_0(half_word_at[<Rn> + immed6]) immed6 even 0≤{t,n}≤7
LDRH	<Rt>,	[<Rn>, <Rm>]			<Rt> ← extend_0(half_word_at[<Rn> + <Rm>])
LDRSH	<Rt>,	[<Rn>, <Rm>]			<Rt> ← extend_sign(half_word[<Rn> + <Rm>])
LDRB	<Rt>,	[<Rn>]			<Rt> ← extend_0(byte_at[<Rn>]) 0≤{t,n}≤7
LDRB	<Rt>,	[<Rn>, #immed5]			<Rt> ← extend_0(byte_at[<Rn> + immed5]) 0≤{t,n}≤7
LDRB	<Rt>,	[<Rn>, <Rm>]			<Rt> ← extend_0(byte_at[<Rn> + <Rm>])
LDRSB	<Rt>,	[<Rn>, <Rm>]			<Rt> ← extend_sign(byte_at[<Rn> + <Rm>])
LDR	<Rt>,	[SP, #immed10]			<Rt> ← word_at[SP + immed10]. immed10 = 4·k 0≤t≤7
LDR	<Rt>,	[PC, #immed10]			<Rt> ← word_at[PC + immed10]. immed10 = 4·k 0≤t≤7
ADR	<Rt>,	label			<Rt> ← label (ADdRess) 0≤t≤7
LDM	<Rn>!,	{<Ra>, <Rb>, …}			Load multiple registers. <Rn> **is not** in the register list and gets updated by address **increment**. Also LDMIA & LDMFD. Accepts ranges inside the list (e.g. {R0-R3, R4, R6-R7}) 0≤{n,a,b,…}≤7
LDM	<Rn>,	{<Ra>, <Rb>, …}			Load multiple registers. <Rn> **is** in the register list and gets updated by load. Also LDMIA and LDMFD. Accepts ranges inside the list (e.g. {R0-R3, R4, R6-R7}) 0≤{n,a,b,…}≤7
Load data from memory pseudo-instructions					
LDR	<Rt>,	=immed32			<Rt> ← immed32. Usually LDR <Rt>, [PC, #immed10] 0≤t≤7
LDR	<Rt>,	label			<Rt> ← word_at[label]. Assembled as previous 0≤t≤7
Write data to memory					
STR	<Rt>,	[<Rn>]			word_at[<Rn>] ← <Rt> 0≤{t,n}≤7
STR	<Rt>,	[<Rn>, #immed7]			word_at[<Rn> + immed7] ← <Rt>. immed7 = 4·k 0≤{t,n}≤7
STR	<Rt>,	[<Rn>, <Rm>]			word_at[<Rn> + <Rm>] ← <Rt>
STRH	<Rt>,	[<Rn>]			half_w_at[<Rn>] ← <Rt> 0≤{t,n}≤7
STRH	<Rt>,	[<Rn>, #immed6]			half_w_at[<Rn> + immed6] ← <Rt>. immed6 even 0≤{t,n}≤7
STRH	<Rt>,	[<Rn>, <Rm>]			half_w_at[<Rn> + <Rm>] ← <Rt>
STRB	<Rt>,	[<Rn>]			byte_at[<Rn>] ← <Rt> 0≤{t,n}≤7
STRB	<Rt>,	[<Rn>, #immed5]			byte_at[<Rn> + immed5] ← <Rt> 0≤{t,n}≤7
STRB	<Rt>,	[<Rn>, <Rm>]			byte_at[<Rn> + <Rm>] ← <Rt>
STR	<Rt>,	[SP, #immed10]			word_at[SP + immed10] ← <Rt>. immed10 = 4·k 0≤{t,n}≤7
STM	<Rn>!,	{<Ra>, <Rb>, …}			Store multiple registers to memory. <Rn> gets updated by address **increment**. Also STMIA and STMEA. Accepts ranges inside the list (e.g. {R0-R3, R4, R6-R7}) 0≤{n,a,b,…}≤7
Stack (full descending)					

PUSH	{<Ra>, <Rb>, …}
	{<Ra>, <Rb>, …, LR}
					Store 1 or more registers into stack updating SP 0≤{a,b,…}≤7

POP	{<Ra>, <Rb>, …}
	{<Ra>, <Rb>, …, PC}
					Restore 1 or more registers from stack updating SP 0≤{a,b,…}≤7

ADD	SP,	<Rm>			SP ← SP + <Rm>. Discards 2 LSBs of <Rm> 0≤m≤7
ADD ADD	SP, SP,	SP, #immed9	#immed9		SP ← SP + immed9. immed9 = 4·k
SUB SUB	SP, SP,	SP, #immed9	#immed9		SP ← SP - immed9. immed9 = 4·k
ADD	<Rd>,	SP,	<Rd>		<Rd> ← SP + <Rd>. Discards 2 LSBs of <Rd> 0≤d≤7
ADD	<Rd>,	SP,	#immed10		<Rd> ← SP + immed10. immed10 = 4·k 0≤d≤7
Arithmetic					
ADD ADD	<Rd>, <Rd>,	<Rd>, <Rm>	<Rm>		<Rd> ← <Rd> + <Rm>. <Rd> or <Rm> may be PC
ADD	<Rd>,	PC,	#immed10		<Rd> ← PC + immed10. immed10 = 4·k 0≤d≤7
ADDS ADDS	<Rd>, <Rd>,	<Rd>, #immed8	#immed8	NZCV NZCV	<Rd> ← <Rd> + immed8 0≤d≤7
ADDS	<Rd>,	<Rn>,	#immed3	NZCV	<Rd> ← <Rn> + immed3 0≤{d,n}≤7
ADDS	<Rd>,	<Rn>,	<Rm>	NZCV	<Rd> ← <Rn> + <Rm> 0≤{d,n,m}≤7
ADCS ADCS	<Rd>, <Rd>,	<Rd>, <Rm>	<Rm>	NZCV NZCV	<Rd> ← <Rd> + <Rm> + C 0≤{d,m}≤7
RSBS	<Rd>,	<Rn>,	#0	NZCV	<Rd> ← -<Rn> (Reverse SubStract) 0≤{d,n}≤7
SUBS SUBS	<Rd>, <Rd>,	<Rd>, #immed8	#immed8	NZCV NZCV	<Rd> ← <Rd> - immed8 0≤d≤7

SUBS	<Rd>,	<Rn>,	#immed3	NZCV	<Rd> ← <Rn> - immed3 0≤{d,n}≤7
SUBS	<Rd>,	<Rn>,	<Rm>	NZCV	<Rd> ← <Rn> - <Rm> 0≤{d,n,m}≤7
SBCS SBCS	<Rd>, <Rd>,	<Rd>, <Rm>	<Rm>	NZCV NZCV	<Rd> ← <Rd> - <Rm> - ~C 0≤{d,m}≤7
CMP	<Rn>,	#immed8		NZCV	<Rn> - immed8 (CoMPare) 0≤n≤7
CMP	<Rn>,	<Rm>		NZCV	<Rn> - <Rm> 0≤{n,m}≤14
CMN	<Rn>,	<Rm>		NZCV	<Rn> + <Rm> (CoMpare Negative) 0≤{n,m}≤7
MULS	<Rd>,	<Rm>,	<Rd>	NZ	<Rd> ← <Rm> · <Rd> 0≤{d,m}≤7
Bitwise logic					
MVNS	<Rd>,	<Rm>		NZ	<Rd> ← ~<Rm> (MoVe Not) 0≤{d,m}≤7
ANDS ANDS	<Rd>, <Rd>,	<Rd>, <Rm>	<Rm>	NZ NZ	<Rd> ← <Rd> & <Rm> 0≤{d,m}≤7
ORRS ORRS	<Rd>, <Rd>,	<Rd>, <Rm>	<Rm>	NZ NZ	<Rd> ← <Rd> | <Rm> 0≤{d,m}≤7
EORS EORS	<Rd>, <Rd>,	<Rd>, <Rm>	<Rm>	NZ NZ	<Rd> ← <Rd> ^ <Rm> 0≤{d,m}≤7
BICS BICS	<Rd>, <Rd>,	<Rd>, <Rm>	<Rm>	NZ NZ	<Rd> ← <Rd> & ~<Rm> (BIt Clear) 0≤{d,m}≤7
TST	<Rn>,	<Rm>		NZ	<Rn> & <Rm> (TeST) 0≤{d,m}≤7
Shift					
ASRS	<Rd>,	<Rm>,	#immed5	NZC	<Rd> ← <Rm> >> immed5 (Arithmetic Shift Right) 0≤{d,m}≤7
ASRS	<Rd>,	#immed5		NZC	<Rd> ← <Rd> >> immed5 (Arithmetic Shift Right) 0≤d≤7
ASRS ASRS	<Rd>, <Rd>,	<Rd>, <Rs>	<Rs>	NZC NZC	<Rd> ← <Rd> >> <Rs> (Arithmetic Shift Right) 0≤{d,s}≤7
LSRS	<Rd>,	<Rm>,	#immed5	NZC	<Rd> ← <Rm> >> immed5 (Logical Shift Right) 0≤{d,m}≤7
LSRS	<Rd>,	#immed5		NZC	<Rd> ← <Rd> >> immed5 (Logical Shift Right) 0≤d≤7
LSRS LSRS	<Rd>, <Rd>,	<Rd>, <Rs>	<Rs>	NZC NZC	<Rd> ← <Rd> >> <Rs> (Logical Shift Right) 0≤{d,s}≤7
LSLS	<Rd>,	<Rm>,	#immed5	NZC	<Rd> ← <Rm> << immed5 (Logical Shift Left) 0≤{d,m}≤7
LSLS	<Rd>,	#immed5		NZC	<Rd> ← <Rd> << immed5 (Logical Shift Left) 0≤d≤7
LSLS LSLS	<Rd>, <Rd>,	<Rd>, <Rs>	<Rs>	NZC NZC	<Rd> ← <Rd> << <Rs> (Logical Shift Left) 0≤{d,s}≤7
RORS RORS	<Rd>, <Rd>,	<Rd>, <Rs>	<Rs>	NZC NZC	<Rd> ← <Rd> ROR <Rs> (ROtate Right) 0≤{d,s}≤7
Reordering					
REV	<Rd>,	<Rn>			<Rd> ← chg_endian(<Rn>) (REVerse) 0≤{d,n}≤7
REV16	<Rd>,	<Rn>			<Rd> ← chg_endian_of_both_halves(<Rn>) 0≤{d,n}≤7
REVSH	<Rd>,	<Rn>			<Rd> ← extend_0(chg_endian(<Rn>[15:0])) 0≤{d,n}≤7
Extension					
SXTB	<Rd>,	<Rm>			<Rd> ← extend_sign(<Rm>[7:0]) (Sign eXTend) 0≤{d,m}≤7
SXTB	<Rd>				<Rd> ← extend_sign(<Rd>[7:0]) 0≤d≤7
SXTH	<Rd>,	<Rm>			<Rd> ← extend_sign(<Rm>[15:0]) 0≤{d,m}≤7
SXTH	<Rd>				<Rd> ← extend_sign(<Rd>[15:0]) 0≤d≤7
UXTB	<Rd>,	<Rm>			<Rd> ← extend_0(<Rm>[7:0]) (Unsigned eXTend) 0≤{d,m}≤7
UXTB	<Rd>				<Rd> ← extend_0(<Rd>[7:0]) 0≤d≤7
UXTH	<Rd>,	<Rm>			<Rd> ← extend_0(<Rm>[15:0]) 0≤{d,m}≤7
UXTH	<Rd>				<Rd> ← extend_0(<Rd>[15:0]) 0≤d≤7
Branch					
B	label				Branch to an address (range is ±2K addresses)
B<cond>	label				Conditional branch (range is -256/+254). See below for <cond>
BX	<Rm>				Branch to address in <Rm> (4·k). To return from a function or interrupt handler use BX LR or POP {PC} <Rm> != {PC,SP}
BL	label				LR ← PC and branch (range is ±16M addresses) (Branch with Link) This opcode is a 32-bit one
BLX	<Rm>				LR ← PC and branch to address in <Rm> (4·k) <Rm> != {PC,SP}
Miscellaneous					
NOP					No Operation
BKPT	#immed8				Software breakpoint
CPSIE	I				Enable maskable interrupts
CPSID	I				Disable maskable interrupts
DMB					Ensures that all memory accesses are completed before a new memory access is committed (Data Memory Barrier) (32 bit opcode)
DSB					Ensures that all memory accesses are completed before the next instruction is executed (Data Synchronization Barrier) (32 bit opc.)
ISB					Flush pipeline and ensure that all previous instructions are completed before executing the next one (Instruction Synchronization Barrier) (32 bit opcode)
MRS	<Rd>,	<SpecialReg>			Read special register (see below) (32 bit opcode)
MSR	<SpecialReg>,	<Rd>			Write special register (see below) (32 bit opcode)
SEV					Send event to all processors in multicore (including itself)
SVC	#immed8				SuperVisor Call
WFE					If there is no record of a previous event, enter sleep mode. Otherwise, clear event record and continue (Wait For Event)
WFI					Enter sleep mode (Wait For Interrupt)
YIELD					Same as NOP on the Cortex-M0 processor

Condition codes for the `B<cond>` instruction		
`<cond>`	meaning	test
`EQ`	EQual	`Z == 1`
`NE`	Not Equal	`Z == 0`
`CS`	Carry Set	`C == 1`
`CC`	Carry Clear	`C == 0`
`MI`	< 0 (MInus)	`N == 1`
`PL`	≥ 0 (PLus)	`N == 0`
`VS`	oVerflow Set	`V == 1`
`VC`	oVerflow Clear	`V == 0`
Intended to be used after a `SUBS`, `SBCS`, `CMP` or `CMN` instruction		
`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 or Equal)	`N == V`
`LT`	signed < (Less Than)	`N != V`
`GT`	signed > (Greater Than)	`Z == 0 && N == V`
`LE`	signed ≤ (Less or Equal)	`Z == 1 || N != V`

Special registers	
name	contents
`APSR`	`NZCV` flags
`IPSR`	exception number
`EPSR`	Thumb state
`IEPSR`	both `IPSR` and `EPSR`
`IAPSR`	both `IPSR` and `APSR`
`EAPSR`	both `EPSR` and `APSR`
`PSR`	all `PSR`s
`MSP`	main `SP`
`PSP`	process `SP`
`PRIMASK`	interrupt mask flag
`CONTROL`	`SP` (`MSP`/`PSP`) used in thread mode

Some ARM assembler directives (much simplified)

`DCB`

```
[<label>]
        DCB     <expr>[, <expr>]...
```

Where each `<expr>` is: a numeric expression that evaluates to an integer between -128 and +255; a single ASCII character enclosed in single quotes (e.g.: `'e'`); or an ASCII string enclosed in double quotes (e.g.: `"Hello, World!"`)

Allocates one or more bytes of memory, defining the initial runtime contents of the memory

`DCW`

```
[<label>]
        DCW     <expr>[, <expr>]...
```

Where each `<expr>` is a numeric expression that evaluates to an integer between -32 768 and +65 535

Allocates one or more halfwords of memory, aligned on two-byte boundaries, defining the initial runtime contents of the memory

`DCD`

```
[<label>]
        DCW     <expr>[, <expr>]...
```

Where each `<expr>` is a numeric expression

Allocates one or more words of memory, aligned on four-byte boundaries, defining the initial runtime contents of the memory

`END`

```
[<label>]
        END
```

Informs the assembler that it has reached the end of the source file. There must be no more directives or mnemonics after `END`

`ALIGN`

```
[<label>]
        ALIGN   <expr>
```

Where `<expr>` is a numeric expression that evaluates to any power of 2 between 0 and 2^{31}

Inserts as many zero bytes (or `NOP` instructions) as needed until an address multiple of `<expr>` is reached

`EXPORT`

```
[<label>]
        EXPORT  <symbol>
```

Declares `<symbol>` to be reachable from other source files

`ENTRY`

```
[<label>]
        ENTRY
```

Declares an entry point to a program where execution must start

`AREA`

```
[<label>]
        AREA    <section_name>[, <attribute>]...
```

Declares a new code or data section to be linked independently of other sections. The section named `RESET` will be linked at address `0x00000000`, so it must contain the initial `SP` and `PC` values at addresses `0x00000000` and `0x00000004` respectively

`<attribute>` may be one of: `CODE`, `DATA`, `READONLY`, `READWRITE`