ARM v6-M Instruction Set

MOV

```
MOV <Rd>,<Rm>                        T1
15                                    0
0 1 0 0 0 1 | 1 0 | D | Rm | Rd
                               Hilow
RD = D:Rd → RD = Rm
```

```
MOVS <Rd>,<Rm>                       T2
15                                    0
0 0 0 | 0 0 | 0 0 0 0 0 | Rm | Rd
Rd = Rm                           low
```

```
MOVS <Rd>,#<imm8>
15                                    0
0 0 1 0 0 | Rd | imm8
Rd = <imm8>                       low
```

ADD

```
ADD <Rdn>,<Rm>
15                                    0
0 1 0 0 0 1 | 0 0 | D | Rm | Rdn
Rdn = Rdn + Rm                  Hilow
```

```
ADDS <Rd>,<Rn>,<Rm>
15                                    0
0 0 | 0 1 1 0 0 | Rm | Rn | Rd
Rd = Rn + Rm                      low
```

```
ADDS <Rd>,<Rn>,#<imm3>
15                                    0
0 0 0 | 1 | 1 | 1 0 | imm3 | Rn | Rd
Rd = Rn + <imm3>                  low
```

```
ADDS <Rdn>,#<imm8>
               0-255d
15                                    0
0 0 1 | 1 0 | Rdn | imm8
Rdn = Rdn + <imm8>                low
```

```
ADCS <Rdn>,<Rm>
15                                    0
0 1 0 0 0 0 | 0 1 0 1 | Rm | Rdn
Rdn = Rdn + Rm + C  ← Carry
```

Subtract

```
SUBS <Rd>,<Rn>,<Rm>
15                                    0
0 0 0 | 1 1 | 0 | 1 | Rm | Rn | Rd
Rd = Rn - Rm
   = Rn + NOT(Rm) + 1
```

```
SUBS <Rd>,<Rn>,#<imm3>
15                                    0
0 0 0 | 1 | 1 | 1 | 1 | imm3 | Rn | Rd
Rd = Rn - <imm3>
   = Rn + NOT<imm3> + 1
```

```
SUBS <Rdn>,#<imm8>
15                                    0
0 0 1 | 1 1 | Rdn | imm8
Rdn = Rdn - <imm8>
    = Rdn + NOT<imm8> + 1
```

```
RSBS <Rd>,<Rn>,#0
15                                    0
0 1 0 0 0 0 | 1 0 0 1 | Rn | Rd
Rd = 0 - Rn
```

```
SBCS <Rdn>,<Rm>
15                                    0
0 1 0 0 0 0 | 0 1 1 0 | Rm | Rdn
Rdn = Rdn - Rm - NOT(C)
    = Rdn + NOT(Rm) + C 1)
```

Multiply

```
MULS <Rdm>,<Rn>,<Rdm>   (N, Z)
15                                    0
0 1 0 0 0 0 | 1 1 0 1 | Rn | Rdm
Rdm = Rn * Rdm                    low
```

4 bit * 4 bit → 8 bit
32 bit * 32 bit → 32 bit
Upper 32 bit are lost!

Compare

```
CMP <Rn>,<Rm>                        T1
15                                    0
0 1 0 0 0 0 | 1 0 1 0 | Rm | Rn
Rn - Rm → N,Z,C,V
```

```
CMP <Rn>,<Rm>                        T2
15                                    0
0 1 0 0 0 1 | 0 1 | N | Rm | Rn
Rn - Rm → N,Z,C,V
```

```
CMP <Rn>,#<imm8>
15                                    0
0 0 1 | 0 1 | Rn | imm8
Rn - <imm8> → N,Z,C,V
```

```
CMN <Rn>,<Rm>
15                                    0
0 1 0 0 0 0 | 1 0 1 1 | Rm | Rn
Rn + Rm → N,Z,C,V
```

Logical (update N and Z flags)

```
ANDS <Rdn>,<Rdn>,<Rm>
15                                    0
0 1 0 0 0 0 | 0 0 0 0 | Rm | Rdn
Rdn = Rdn & Rm
```

Bitclear

```
BICS <Rdn>,<Rdn>,<Rm>
15                                    0
0 1 0 0 0 0 | 1 1 1 0 | Rm | Rdn
Rdn = Rdn & !Rm
```

Exclusive OR

```
EORS <Rdn>,<Rdn>,<Rm>
15                                    0
0 1 0 0 0 0 | 0 0 0 1 | Rm | Rdn
Rdn = Rdn $ Rm
```

```
MVNS <Rd>,<Rm>
15                                    0
0 1 0 0 0 0 | 1 1 1 1 | Rm | Rd
Rd = !Rm     Not
```

```
ORRS <Rdn>,<Rdn>,<Rm>
15                                    0
0 1 0 0 0 0 | 1 1 0 0 | Rm | Rdn
Rdn = Rdn # Rm     OR
```

```
TST <Rn>,<Rm>
15                                    0
0 1 0 0 0 0 | 1 0 0 0 | Rm | Rn
Rn & Rm → N,Z
```

Shift/Rotate

```
ASRS <Rdn>,<Rdn>,<Rm>
15                                    0
0 1 0 0 0 0 | 0 1 0 0 | Rm | Rdn
Rdn = shift Rdn right
      by Rm<7:0> bits,
      fill with MSB 2)
```

```
ASRS <Rd>,<Rm>,#<imm5>
15                                    0
0 0 0 | 1 0 | imm5 | Rm | Rd
Rd = shift Rm right
     by <imm5> bits
     fill with MSB
```

```
LSLS <Rdn>,<Rdn>,<Rm>
15                                    0
0 1 0 0 0 0 | 0 0 1 0 | Rm | Rdn
Rdn = shift Rdn left
      by Rm<7:0> bits,
      fill with zeros 2)
```

```
LSLS <Rd>,<Rm>,#<imm5>
15                                    0
0 0 0 | 0 0 | imm5 | Rm | Rd
Rd = shift Rm left
     by <imm5> bits
     fill with zeros
```

```
LSRS <Rdn>,<Rdn>,<Rm>
15                                    0
0 1 0 0 0 0 | 0 0 1 1 | Rm | Rdn
Rdn = shift Rdn right
      by Rm<7:0> bits,
      fill with zeros 2)
```

```
LSRS <Rd>,<Rm>,#<imm5>
15                                    0
0 0 0 | 0 1 | imm5 | Rm | Rd
Rd = shift Rm right
     by <imm5> bits
     fill with zeros
```

```
RORS <Rdn>,<Rdn>,<Rm>
15                                    0
0 1 0 0 0 0 | 0 1 1 1 | Rm | Rdn
Rdn = cyclic rotate right
```

Load

```
LDR <Rt>,[<Rn>,<Rm>]
15                                    0
0 1 0 1 | 1 0 0 | Rm | Rn | Rt
Rt = Mem[@(Rn+Rm)]
```

```
LDR <Rt>,[<Rn>,#<imm>]
15                                    0
0 1 1 0 1 | imm5 | Rn | Rt
<imm> = imm5:00
Rt = Mem[@(Rn + <imm>)]
```

```
LDR <Rt>,[PC,#<imm>]
15                                    0
0 1 0 0 1 | Rt | imm8
<imm> = imm8:00
Rt = Mem[@(PC + <imm>)]
```

```
LDRB <Rt>,[<Rn>,<Rm>]
15                                    0
0 1 0 1 | 1 1 0 | Rm | Rn | Rt
Rt = Byte[@(Rn+Rm)]
```

```
LDRB <Rt>,[<Rn>,#<imm>]
15                                    0
0 1 1 | 1 | 1 | imm5 | Rn | Rt
<imm> = imm5
Rt = Byte[@(Rn+<imm>)]
```

```
LDRH <Rt>,[<Rn>,<Rm>]
15                                    0
0 1 0 1 | 1 0 1 | Rm | Rn | Rt
Rt = Hw[@(Rn+Rm)]
```

```
LDRH <Rt>,[<Rn>,#<imm>]
15                                    0
1 0 0 0 | 1 | imm5 | Rn | Rt
<imm> = imm5:0
Rt = Hw[@(Rn+imm5)]
```

```
LDRSB <Rt>,[<Rn>,<Rm>]
15                                    0
0 1 0 1 | 0 1 1 | Rm | Rn | Rt
Rt = sign_extend(Byte[@(Rn+Rm)])
```

```
LDRSH <Rt>,[<Rn>,<Rm>]
15                                    0
0 1 0 1 | 1 1 1 | Rm | Rn | Rt
Rt = sign_extend(HWord[@(Rn+Rm)])
```

Store

```
STR <Rt>,[<Rn>,<Rm>]
15                      0
|0101|000|Rm|Rn|Rt|
Mem[@(Rn+Rm)] = Rt
```

```
STR <Rt>,[<Rn>,#<imm>]
15                      0
|01100|#imm5|Rn|Rt|
<imm> = imm5:00
Mem[@(Rn + <imm>)] = Rt
```

```
STRB <Rt>,[<Rn>,<Rm>]
15                      0
|0101|010|Rm|Rn|Rt|
Byte[@(Rn+Rm)] = Rt(7:0)
```

```
STRB <Rt>,[<Rn>,#<imm>]
15                      0
|011|1|0|imm5|Rn|Rt|
<imm> = imm5
Byte[@(Rn+<imm>)]= Rt(7:0)
```

```
STRH <Rt>,[<Rn>,<Rm>]
15                      0
|0101|001|Rm|Rn|Rt|
Hw[@(Rn+Rm)] = Rt(15:0)
```

```
STRH <Rt>,[<Rn>,#<imm>]
15                      0
|1000|0|imm5|Rn|Rt|
<imm> = imm5:0
Hw[@(Rn+<imm>] = Rt(15:0)
```

Load/Store Multiple

```
LDM <Rn>!,<registers>
LDM <Rn>,<registers>
15                      0
|1100|1|Rn|reg_list|
Registers in reg_list
are loaded from memory
starting at address in Rn
```

```
STM <Rn>!,<registers>
15                      0
|1100|0|Rn|reg_list|
Registers in reg_list
are stored to memory
starting at address in Rn
```

Push/Pop

```
PUSH {registers}
15          7           0
|1011|010|M|reg_list|
reg_list:R7…R0
M: LR
```

```
POP {registers}
15          7           0
|1011|110|P|reg_list|
reg_list: R7…R0
P: PC
```

Branch

```
B <label>
15                      0
|11100|imm11|
PC = PC + imm11:0
```

```
BLX <Rm>
15                      0
|010001|1|1|1|Rm|000|
LR = PC - 2 (LSB set to '1')
PC = Rm
```

```
BX <Rm>
15                      0
|010001|110|Rm|000|
PC = Rm
```

```
BL <label>
15                   0 15                      0
|11110|S|imm10|        |1|1|J1|1|J2|imm11|
I1 = NOT(J1 EOR S); I2 = NOT (J2 EOR S)
<imm> = S:I1:I2:imm10:imm11:0
LR = PC (LSB set to '1')
PC = PC + <imm>
```

```
B<c> <label>
15                      0
|1101|cond|imm8|
if (cond) then
     PC = PC + imm8:0
```

cond	short	Flag	cond	short	Flag	cond	short	Flag
0000	EQ	Z == 1	0110	VS	V == 1	1100	GT	Z == 0 and N == V
0001	NE	Z == 0	0111	VC	V == 0	1101	LE	Z == 1 or N != V
0010	CS/HS	C == 1	1000	HI	C == 1 and Z == 0	1110	AL	always
0011	CC/LO	C == 0	1001	LS	C == 0 or Z == 1	1111	--	--
0100	MI	N == 1	1010	GE	N == V			
0101	PL	N == 0	1011	LT	N !== V			

Stack Operations

```
ADD <Rd>,SP,#<imm>
15                      0
|1010|1|Rd|imm8|
<imm> = imm8:00
Rd = SP + <imm>
```

```
ADD SP,SP,#<imm>
15                      0
|1011|0000|0|imm7|
<imm> = imm7:00
SP = SP + <imm>
```

```
SUB SP,SP,#<imm>
15                      0
|1011|0000|1|imm7|
<imm> = imm7:00
SP = SP - <imm>
```

```
LDR <Rt>,[SP,#<imm>]
15                      0
|1001|1|Rt|imm8|
<imm> = imm8:00
Rt = Mem[SP + <imm>]
```

```
STR <Rt>,[SP,#<imm>]
15                      0
|1001|0|Rt|imm8|
<imm> = imm8:00
Mem[SP + <imm>] = Rt
```

Extend

```
SXTB <Rd>,<Rm>
15                      0
|1011|0010|01|Rm|Rd|
Rd[31:0]:=SignExt(Rm[7:0])
```

```
SXTH <Rd>,<Rm>
15                      0
|1011|0010|00|Rm|Rd|
Rd[31:0]:=SignExt(Rm[15:0])
```

```
UXTB <Rd>,<Rm>
15                      0
|1011|0010|11|Rm|Rd|
Rd[31:0]:=ZeroExt(Rm[7:0])
```

```
UXTH <Rd>,<Rm>
15                      0
|1011|0010|10|Rm|Rd|
Rd[31:0]:=ZeroExt(Rm[15:0])
```

Pseudo Instructions

```
LDR <Rt>, <label>    =>  LDR <Rt>, [PC, #<imm>]
LDR <Rt>, =<value>   =>  LDR <Rt>, [PC, #<imm>]
                         …
                         Literalpool
                         DCD  value
```

Weitere Befehle

```
REV    REV16  REVSH  SVC    CPSID  CPSIE  SETEND  BKPT   NOP    SEV
WFE    WFI    YIELD
```

Thumb® 16-bit Instruction Set
Quick Reference Card

This card lists all Thumb instructions available on Thumb-capable processors earlier than ARM®v6T2. In addition, it lists all Thumb-2 16-bit instructions.
The instructions shown on this card are all 16-bit in Thumb-2, except where noted otherwise.
All registers are Lo (R0-R7) except where specified. Hi registers are R8-R15.

Key to Tables

§	See Table **ARM architecture versions**.		`<loreglist+LR>`	A comma-separated list of Lo registers. plus the LR, enclosed in braces, { and }.
`<loreglist>`	A comma-separated list of Lo registers, enclosed in braces, { and }.		`<loreglist+PC>`	A comma-separated list of Lo registers. plus the PC, enclosed in braces, { and }.

Operation		**§**	**Assembler**	**Updates**	**Action**	**Notes**
Move	Immediate		`MOVS Rd, #<imm>`	N Z	Rd := imm	imm range 0-255.
	Lo to Lo		`MOVS Rd, Rm`	N Z	Rd := Rm	Synonym of `LSLS Rd, Rm, #0`
	Hi to Lo, Lo to Hi, Hi to Hi		`MOV Rd, Rm`		Rd := Rm	Not Lo to Lo.
	Any to Any	6	`MOV Rd, Rm`		Rd := Rm	Any register to any register.
Add	Immediate 3		`ADDS Rd, Rn, #<imm>`	N Z C V	Rd := Rn + imm	imm range 0-7.
	All registers Lo		`ADDS Rd, Rn, Rm`	N Z C V	Rd := Rn + Rm	
	Hi to Lo, Lo to Hi, Hi to Hi		`ADD Rd, Rd, Rm`		Rd := Rd + Rm	Not Lo to Lo.
	Any to Any	T2	`ADD Rd, Rd, Rm`		Rd := Rd + Rm	Any register to any register.
	Immediate 8		`ADDS Rd, Rd, #<imm>`	N Z C V	Rd := Rd + imm	imm range 0-255.
	With carry		`ADCS Rd, Rd, Rm`	N Z C V	Rd := Rd + Rm + C-bit	
	Value to SP		`ADD SP, SP, #<imm>`		SP := SP + imm	imm range 0-508 (word-aligned).
	Form address from SP		`ADD Rd, SP, #<imm>`		Rd := SP + imm	imm range 0-1020 (word-aligned).
	Form address from PC		`ADR Rd, <label>`		Rd := label	label range PC to PC+1020 (word-aligned).
Subtract	Lo and Lo		`SUBS Rd, Rn, Rm`	N Z C V	Rd := Rn – Rm	
	Immediate 3		`SUBS Rd, Rn, #<imm>`	N Z C V	Rd := Rn – imm	imm range 0-7.
	Immediate 8		`SUBS Rd, Rd, #<imm>`	N Z C V	Rd := Rd – imm	imm range 0-255.
	With carry		`SBCS Rd, Rd, Rm`	N Z C V	Rd := Rd – Rm – NOT C-bit	
	Value from SP		`SUB SP, SP, #<imm>`		SP := SP – imm	imm range 0-508 (word-aligned).
	Negate		`RSBS Rd, Rn, #0`	N Z C V	Rd := – Rn	Synonym: `NEGS Rd, Rn`
Multiply	Multiply		`MULS Rd, Rm, Rd`	N Z * *	Rd := Rm * Rd	* C and V flags unpredictable in §4T, unchanged in §5T and above
Compare			`CMP Rn, Rm`	N Z C V	update APSR flags on Rn – Rm	Can be Lo to Lo, Lo to Hi, Hi to Lo, or Hi to Hi.
	Negative		`CMN Rn, Rm`	N Z C V	update APSR flags on Rn + Rm	
	Immediate		`CMP Rn, #<imm>`	N Z C V	update APSR flags on Rn – imm	imm range 0-255.
Logical	AND		`ANDS Rd, Rd, Rm`	N Z	Rd := Rd AND Rm	
	Exclusive OR		`EORS Rd, Rd, Rm`	N Z	Rd := Rd EOR Rm	
	OR		`ORRS Rd, Rd, Rm`	N Z	Rd := Rd OR Rm	
	Bit clear		`BICS Rd, Rd, Rm`	N Z	Rd := Rd AND NOT Rm	
	Move NOT		`MVNS Rd,` [redacted] `Rm`	N Z	Rd := NOT Rm	
	Test bits		`TST Rn, Rm`	N Z	update APSR flags on Rn AND Rm	
Shift/rotate	Logical shift left		`LSLS Rd, Rm, #<shift>`	N Z C*	Rd := Rm << shift	Allowed shifts 0-31. * C flag unaffected if shift is 0.
			`LSLS Rd, Rd, Rs`	N Z C*	Rd := Rd << Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.
	Logical shift right		`LSRS Rd, Rm, #<shift>`	N Z C	Rd := Rm >> shift	Allowed shifts 1-32.
			`LSRS Rd, Rd, Rs`	N Z C*	Rd := Rd >> Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.
	Arithmetic shift right		`ASRS Rd, Rm, #<shift>`	N Z C	Rd := Rm ASR shift	Allowed shifts 1-32.
			`ASRS Rd, Rd, Rs`	N Z C*	Rd := Rd ASR Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.
	Rotate right		`RORS Rd, Rd, Rs`	N Z C*	Rd := Rd ROR Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.

Thumb 16-bit Instruction Set Quick Reference Card

Operation		§	Assembler	Action	Notes
Load	with immediate offset, word		`LDR Rd, [Rn, #<imm>]`	Rd := [Rn + imm]	imm range 0-124, multiple of 4.
	halfword		`LDRH Rd, [Rn, #<imm>]`	Rd := ZeroExtend([Rn + imm][15:0])	Clears bits 31:16. imm range 0-62, even.
	byte		`LDRB Rd, [Rn, #<imm>]`	Rd := ZeroExtend([Rn + imm][7:0])	Clears bits 31:8. imm range 0-31.
	with register offset, word		`LDR Rd, [Rn, Rm]`	Rd := [Rn + Rm]	
	halfword		`LDRH Rd, [Rn, Rm]`	Rd := ZeroExtend([Rn + Rm][15:0])	Clears bits 31:16
	signed halfword		`LDRSH Rd, [Rn, Rm]`	Rd := SignExtend([Rn + Rm][15:0])	Sets bits 31:16 to bit 15
	byte		`LDRB Rd, [Rn, Rm]`	Rd := ZeroExtend([Rn + Rm][7:0])	Clears bits 31:8
	signed byte		`LDRSB Rd, [Rn, Rm]`	Rd := SignExtend([Rn + Rm][7:0])	Sets bits 31:8 to bit 7
	PC-relative		`LDR Rd, <label>`	Rd := [label]	label range PC to PC+1020 (word-aligned).
	SP-relative		`LDR Rd, [SP, #<imm>]`	Rd := [SP + imm]	imm range 0-1020, multiple of 4.
	Multiple, not including base		`LDM Rn!, <loreglist>`	Loads list of registers (not including Rn)	Always updates base register, Increment After.
	Multiple, including base		`LDM Rn, <loreglist>`	Loads list of registers (including Rn)	Never updates base register, Increment After.
Store	with immediate offset, word		`STR Rd, [Rn, #<imm>]`	[Rn + imm] := Rd	imm range 0-124, multiple of 4.
	halfword		`STRH Rd, [Rn, #<imm>]`	[Rn + imm][15:0] := Rd[15:0]	Ignores Rd[31:16]. imm range 0-62, even.
	byte		`STRB Rd, [Rn, #<imm>]`	[Rn + imm][7:0] := Rd[7:0]	Ignores Rd[31:8]. imm range 0-31.
	with register offset, word		`STR Rd, [Rn, Rm]`	[Rn + Rm] := Rd	
	halfword		`STRH Rd, [Rn, Rm]`	[Rn + Rm][15:0] := Rd[15:0]	Ignores Rd[31:16]
	byte		`STRB Rd, [Rn, Rm]`	[Rn + Rm][7:0] := Rd[7:0]	Ignores Rd[31:8]
	SP-relative, word		`STR Rd, [SP, #<imm>]`	[SP + imm] := Rd	imm range 0-1020, multiple of 4.
	Multiple		`STM Rn!, <loreglist>`	Stores list of registers	Always updates base register, Increment After.
Push	Push		`PUSH <loreglist>`	Push registers onto full descending stack	
	Push with link		`PUSH <loreglist+LR>`	Push LR and registers onto full descending stack	
Pop	Pop		`POP <loreglist>`	Pop registers from full descending stack	
	Pop and return	4T	`POP <loreglist+PC>`	Pop registers, branch to address loaded to PC	
	Pop and return with exchange	5T	`POP <loreglist+PC>`	Pop, branch, and change to ARM state if address[0] = 0	
If-Then	If-Then	T2	`IT{pattern} {cond}`	Makes up to four following instructions conditional, according to pattern. pattern is a string of up to three letters. Each letter can be T (Then) or E (Else).	The first instruction after IT has condition cond. The following instructions have condition cond if the corresponding letter is T, or the inverse of cond if the corresponding letter is E. See Table **Condition Field**.
Branch	Conditional branch		`B{cond} <label>`	If {cond} then PC := label	label must be within – 252 to + 258 bytes of current instruction. See Table **Condition Field**.
	Compare, branch if (non) zero	T2	`CB{N}Z Rn,<label>`	If Rn {== | !=} 0 then PC := label	label must be within +4 to +130 bytes of current instruction.
	Unconditional branch		`B <label>`	PC := label	label must be within ±2KB of current instruction.
	Long branch with link		`BL <label>`	LR := address of next instruction, PC := label	This is a 32-bit instruction. label must be within ±4MB of current instruction (T2: ±16MB).
	Branch and exchange		`BX Rm`	PC := Rm AND 0xFFFFFFFE	Change to ARM state if Rm[0] = 0.
	Branch with link and exchange	5T	`BLX <label>`	LR := address of next instruction, PC := label Change to ARM	This is a 32-bit instruction. label must be within ±4MB of current instruction (T2: ±16MB).
	Branch with link and exchange	5T	`BLX Rm`	LR := address of next instruction, PC := Rm AND 0xFFFFFFFE	Change to ARM state if Rm[0] = 0.
Extend	Signed, halfword to word	6	`SXTH Rd, Rm`	Rd[31:0] := SignExtend(Rm[15:0])	
	Signed, byte to word	6	`SXTB Rd, Rm`	Rd[31:0] := SignExtend(Rm[7:0])	
	Unsigned, halfword to word	6	`UXTH Rd, Rm`	Rd[31:0] := ZeroExtend(Rm[15:0])	
	Unsigned, byte to word	6	`UXTB Rd, Rm`	Rd[31:0] := ZeroExtend(Rm[7:0])	
Reverse	Bytes in word	6	`REV Rd, Rm`	Rd[31:24] := Rm[7:0], Rd[23:16] := Rm[15:8], Rd[15:8] := Rm[23:16], Rd[7:0] := Rm[31:24]	
	Bytes in both halfwords	6	`REV16 Rd, Rm`	Rd[15:8] := Rm[7:0], Rd[7:0] := Rm[15:8], Rd[31:24] := Rm[23:16], Rd[23:16] := Rm[31:24]	
	Bytes in low halfword, sign extend	6	`REVSH Rd, Rm`	Rd[15:8] := Rm[7:0], Rd[7:0] := Rm[15:8], Rd[31:16] := Rm[7] * &FFFF	

Thumb 16-bit Instruction Set Quick Reference Card

Operation		§	Assembler	Action	Notes
Processor state change	Supervisor Call		`SVC <immed_8>`	Supervisor Call processor exception	8-bit immediate value encoded in instruction. Formerly SWI.
	Change processor state	6	`CPSID <iflags>`	Disable specified interrupts	
		6	`CPSIE <iflags>`	Enable specified interrupts	
	Set endianness	6	`SETEND <endianness>`	Sets endianness for loads and saves.	`<endianness>` can be `BE` (Big Endian) or `LE` (Little Endian).
	Breakpoint	5T	`BKPT <immed_8>`	Prefetch abort *or* enter debug state	8-bit immediate value encoded in instruction.
No Op	No operation		`NOP`	None, might not even consume any time.	Real NOP available in ARM v6K and above.
Hint	Set event	T2	`SEV`	Signal event in multiprocessor system.	Executes as NOP in Thumb-2. Functionally available in ARM v7.
	Wait for event	T2	`WFE`	Wait for event, IRQ, FIQ, Imprecise abort, or Debug entry request.	Executes as NOP in Thumb-2. Functionally available in ARM v7.
	Wait for interrupt	T2	`WFI`	Wait for IRQ, FIQ, Imprecise abort, or Debug entry request.	Executes as NOP in Thumb-2. Functionally available in ARM v7.
	Yield	T2	`YIELD`	Yield control to alternative thread.	Executes as NOP in Thumb-2. Functionally available in ARM v7.

Condition Field

Mnemonic	Description
`EQ`	Equal
`NE`	Not equal
`CS / HS`	Carry Set / Unsigned higher or same
`CC / LO`	Carry Clear / Unsigned lower
`MI`	Negative
`PL`	Positive or zero
`VS`	Overflow
`VC`	No overflow
`HI`	Unsigned higher
`LS`	Unsigned lower or same
`GE`	Signed greater than or equal
`LT`	Signed less than
`GT`	Signed greater than
`LE`	Signed less than or equal
`AL`	Always. Do not use in `B{cond}`

In Thumb code for processors earlier than ARMv6T2, cond must not appear anywhere except in Conditional Branch (`B{cond}`) instructions.

In Thumb-2 code, cond can appear in any of these instructions (except `CBZ`, `CBNZ`, `CPSID`, `CPSIE`, `IT`, and `SETEND`).
The condition is encoded in a preceding `IT` instruction (except in the case of `B{cond}` instructions).
If `IT` instructions are explicitly provided in the Assembly language source file, the conditions in the instructions must match the corresponding `IT` instructions.

ARM architecture versions

4T	All Thumb versions of ARM v4 and above.
5T	All Thumb versions of ARM v5 and above.
6	All Thumb versions of ARM v6 and above.
T2	All Thumb-2 versions of ARM v6 and above.

Document Number

ARM QRC 0006E

Change Log

Issue	Date	Change
A	Nov 2004	First Release
B	May 2005	RVCT 2.2 SP1
C	March 2006	RVCT 3.0
D	March 2007	RVCT 3.1
E	Sept 2008	RVCT 4.0

www.arm.com

ARM® Thumb® Cortex-M0/M1 Instruction Set ordered by machine code

This card lists all Thumb instructions ordered by machine code to ease manually disassemble Thumb code.
See the respective ***Thumb® 16-bit Instruction Set Quick Reference Card*** for details on the individual instructions.
Version 1.3, 2019-08-20, Andreas Gieriet

Zürcher Hochschule für Angewandte Wissenschaften
zh aw School of Engineering
InES Institute of Embedded Systems

0000 – 0x0xxx Instructions
```
0000 0000 00mm mddd    MOVS  Rddd, Rmmm         ; Rddd  = Rmmm                  --> alias for LSLS Rddd,Rmmm,#0
0000 0iii iimm mddd    LSLS  Rddd, Rmmm, #0biiiii; Rddd  = Rmmm LSL #0b0iiiii
0000 1iii iimm mddd    LSRS  Rddd, Rmmm, #0biiiii; Rddd  = Rmmm LSR #0b0iiiii
```

0001 – 0x1xxx Instructions
```
0001 0iii iimm mddd    ASRS  Rddd, Rmmm, #0biiiii; Rddd  = Rmmm ASR #0b0iiiii
0001 100m mmnn nddd    ADDS  Rddd, Rnnn, Rmmm   ; Rddd  = Rnnn  +  Rmmm
0001 101m mmnn nddd    SUBS  Rddd, Rnnn, Rmmm   ; Rddd  = Rnnn  -  Rmmm
0001 110i iinn nddd    ADDS  Rddd, Rnnn, #0biii ; Rddd  = Rnnn  +  #0b0iii
0001 111i iinn nddd    SUBS  Rddd, Rnnn, #0biii ; Rddd  = Rnnn  -  #0b0iii
```

0010 – 0x2xxx Instructions
```
0010 0ddd iiii iiii    MOVS  Rddd, #0biiiiiiii  ; Rddd  =          #0b0iiiiiiii
0010 1nnn iiii iiii    CMP   Rnnn, #0biiiiiiii  ; flags = Rnnn  -  #0b0iiiiiiii
```

0011 – 0x3xxx Instructions
```
0011 0ddd iiii iiii    ADDS  Rddd, #0biiiiiiii  ; Rddd  = Rddd  +  #0b0iiiiiiii
0011 1ddd iiii iiii    SUBS  Rddd, #0biiiiiiii  ; Rddd  = Rddd  -  #0b0iiiiiiii
```

0100 – 0x4xxx Instructions
```
0100 0000 00mm mddd    ANDS  Rddd,  Rmmm        ; Rddd  = Rddd  &  Rmmm
0100 0000 01mm mddd    EORS  Rddd,  Rmmm        ; Rddd  = Rddd  ^  Rmmm
0100 0000 10mm mddd    LSLS  Rddd,  Rmmm        ; Rddd  = Rddd LSL Rmmm
0100 0000 11mm mddd    LSRS  Rddd,  Rmmm        ; Rddd  = Rddd LSR Rmmm
0100 0001 00mm mddd    ASRS  Rddd,  Rmmm        ; Rddd  = Rddd ASR Rmmm
0100 0001 01mm mddd    ADCS  Rddd,  Rmmm        ; Rddd  = Rddd  +  Rmmm +  carry
0100 0001 10mm mddd    SBCS  Rddd,  Rmmm        ; Rddd  = Rddd  -  Rmmm - ~carry
0100 0001 11mm mddd    RORS  Rddd,  Rmmm        ; Rddd  = Rddd ROR Rmmm
0100 0010 00mm mddd    TST   Rddd,  Rmmm        ; flags : Rddd  &  Rmmm
0100 0010 01mm mddd    RSBS  Rddd,  Rmmm, #0    ; Rddd  = 0     -  Rmmm   --> alias for NEGS Rddd, Rmmm
0100 0010 10mm mnnn    CMP   Rnnn,  Rmmm        ; flags : Rnnn  -  Rmmm
0100 0010 11mm mnnn    CMN   Rnnn,  Rmmm        ; flags : Rnnn  +  Rmmm
0100 0011 00mm mddd    ORRS  Rddd,  Rmmm        ; Rddd  = Rddd  |  Rmmm
0100 0011 01mm mddd    MULS  Rddd,  Rmmm, Rddd  ; Rddd  = Rddd  *  Rmmm
0100 0011 10mm mddd    BICS  Rddd,  Rmmm        ; Rddd  = Rddd  & ~Rmmm   --> bit clear
0100 0011 11mm mddd    MVNS  Rddd,  Rmmm        ; Rddd  =         ~Rmmm
0100 0100 dmmm mddd    ADD   Rdddd, Rmmmm       ; Rdddd = Rdddd + Rmmmm
0100 0101 nmmm mnnn    CMP   Rnnnn, Rmmmm       ; flags : Rnnnn - Rmmmm
0100 0110 dmmm mddd    MOV   Rdddd, Rmmmm       ; Rdddd =         Rmmmm
0100 0111 0mmm m...    BX    Rmmmm              ;             PC=Rmmmm (mmmm==0b1111: unpredictable)
0100 0111 1mmm m...    BLX   Rmmmm              ; LR = IPC+2,PC=Rmmmm (mmmm==0b1111: unpredictable)
0100 1ttt iiii iiii    LDR   Rttt, [PC, #off]   ; Rttt = [((IPC+4)&~0b011)+0b0iiiiiiii00] --> +1020 max
                       LDR   Rttt, label        ; --> the assembler calculates the above from the label
                       LDR   Rttt, =lab         ; --> pseudo instruction: the assembler stores the lab/lit
                       LDR   Rttt, =lit         ;     in litpool, access PC relative with LDR Rttt,litpool
```

0101 – 0x5xxx Instructions
```
0101 000m mmnn nttt    STR   Rttt, [Rnnn, Rmmm]  ; [Rnnn + Rmmm] = Rttt
0101 001m mmnn nttt    STRH  Rttt, [Rnnn, Rmmm]  ; [Rnnn + Rmmm] = Rttt               --> low half
0101 010m mmnn nttt    STRB  Rttt, [Rnnn, Rmmm]  ; [Rnnn + Rmmm] = Rttt               --> low byte
0101 011m mmnn nttt    LDRSB Rttt, [Rnnn, Rmmm]  ; Rttt<sss1> = [Rnnn + Rmmm]<1>      --> low byte
0101 100m mmnn nttt    LDR   Rttt, [Rnnn, Rmmm]  ; Rttt = [Rnnn + Rmmm]
0101 101m mmnn nttt    LDRH  Rttt, [Rnnn, Rmmm]  ; Rttt<0021> = [Rnnn + Rmmm]<21>     --> low half
0101 110m mmnn nttt    LDRB  Rttt, [Rnnn, Rmmm]  ; Rttt<0001> = [Rnnn + Rmmm]<1>      --> low byte
0101 111m mmnn nttt    LDRSH Rttt, [Rnnn, Rmmm]  ; Rttt<ss21> = [Rnnn + Rmmm]<21>     --> low half
```

0110 – 0x6xxx Instructions
```
0110 0iii iinn nttt    STR   Rttt, [Rnnn, #off]  ; [Rnnn + 0b0iiiii00] = Rttt         --> +124 max
0110 1iii iinn nttt    LDR   Rttt, [Rnnn, #off]  ; Rttt = [Rnnn + 0x0iiiii00]         --> +124 max
```

0111 – 0x7xxx Instructions
```
0111 0iii iinn nttt    STRB  Rttt, [Rnnn, #off]  ; [Rnnn + 0b0iiiii] = Rttt           --> +31 max, low byte
0111 1iii iinn nttt    LDRB  Rttt, [Rnnn, #off]  ; Rttt<0001> = [Rnnn + 0x0iiiii]<1>  --> +31 max, low byte
```

1000 – 0x8xxx Instructions
```
1000 0iii iinn nttt    STRH  Rttt, [Rnnn, #off]  ; [Rnnn + 0x0iiiii0] = Rttt          --> +62 max, low half
1000 1iii iinn nttt    LDRH  Rttt, [Rnnn, #off]  ; Rttt<0021> = [Rnnn + 0x0iiiii0]<21> --> +62 max, low half
```

1001 – 0x9xxx Instructions
```
1001 0ttt iiii iiii    STR   Rttt, [SP, #off]   ; [SP + 0b0iiiiiiii00] = Rttt          --> +1020 max
1001 1ttt iiii iiii    LDR   Rttt, [SP, #off]   ; Rttt = [SP + 0b0iiiiiiii00]          --> +1020 max
```

1010 – 0xAxxx Instructions
```
1010 0ddd iiii iiii    ADR   Rddd, label       ; Rddd = ((IPC+4)&~0b011)+0b0iiiiiiii00   --> +1020 max
1010 1ddd iiii iiii    ADD   Rddd, SP, #off    ; Rddd = SP + 0b0iiiiiiii00               --> +1020 max
```

1011 – 0xBxxx Instructions
```
1011 0000 0iii iiii    ADD   SP, SP, #off      ; SP = SP + 0b0iiiiiii00                  --> +508 max
1011 0000 1iii iiii    SUB   SP, SP, #off      ; SP = SP - 0b0iiiiiii00                  --> +508 max
~~1011 00i1 iiii innn    CBZ  Rnnn, label      ; if Rnnn==zero, PC = IPC+4 + 0x0iiiiii0  --> +126 max~~
1011 0010 00mm mddd    SXTH Rddd, Rmmm         ; Rddd<ss21> = Rmmm<4321> --> low half
1011 0010 01mm mddd    SXTB Rddd, Rmmm         ; Rddd<sss1> = Rmmm<4321> --> low byte
1011 0010 10mm mddd    UXTH Rddd, Rmmm         ; Rddd<0021> = Rmmm<4321> --> low half
1011 0010 11mm mddd    UXTB Rddd, Rmmm         ; Rddd<0001> = Rmmm<4321> --> low byte
1011 0100 rrrr rrrr    PUSH {reg0-7}           ; rrrrrrrr = Lo reg-mask --> pushes regs to SP (decrements SP)
1011 0101 rrrr rrrr    PUSH {LR,reg0-7]        ; rrrrrrrr = Lo reg-mask --> pushes regs to SP (decrements SP)
1011 0110 0100 xxxx    -                       ; unpredictable
1011 0110 0101 0...    SETEND LE               ; sets little-endian mode in CPSR
1011 0110 0101 1...    SETEND BE               ; sets big-endian mode in CPSR
1011 0110 0110 0aif    CPSIE aif               ; Enable Processor State  --> a=imprecise-abort, i=IRQ, f=FIQ
1011 0110 0111 0aif    CPSID aif               ; Disable Processor State --> a=imprecise-abort, i=IRQ, f=FIQ
1011 0110 011x 1xxx    -                       ; unpredictable
~~1011 10i1 iiii innn    CBNZ Rnnn, label      ; if Rnnn!=zero, PC = IPC+4 + 0x0iiiiii0      --> + 126 max~~
1011 1010 00mm mddd    REV   Rddd, Rmmm        ; Rddd<4321> = Rmmm<1234> --> reverse all
1011 1010 01mm mddd    REV16 Rddd, Rmmm        ; Rddd<4321> = Rmmm<3412> --> reverse low half, rev. upper half
1011 1010 10xx xxxx    -                       ; undefined
1011 1010 11mm mddd    REVSH Rddd, Rmmm        ; Rddd<4321> = Rmmm<ss12> --> reverse low half, sign extended
1011 1100 rrrr rrrr    POP  {reg0-7}           ; rrrrrrrr = Lo reg-mask  --> pops regs from SP (increments SP)
1011 1101 rrrr rrrr    POP  {PC,reg0-7]        ; rrrrrrrr = Lo reg-mask  --> pops regs from SP (increments SP)
1011 1110 iiii iiii    BKPT #0biiiiiiii        ; breakpoint, arg ignored by HW
1011 1111 0000 0000    NOP                     ; do nothing
1011 1111 0001 0000    YIELD                   ; do nothing, NOP-Hint: signal to HW to suspend/resume threads
1011 1111 0010 0000    WFE                     ; do nothing, NOP-Hint, wait for event
1011 1111 0011 0000    WFI                     ; do nothing, NOP-Hint: wait for interrupt
1011 1111 0100 0000    SEV                     ; do nothing, NOP-Hint: signal event to multi-processor system
~~1011 1111 cccc mmmm    ITsel cond            ; if-then: sel=mmmm: T=then/E=else, cond=cccc: as for Bcc<11:8>~~
```

1100 – 0xCxxx Instructions
```
1100 0nnn rrrr rrrr    STMIA Rnnn! {reg0-7}    ; rrrrrrrr = Lo reg-mask, inc Rnnn
1100 1nnn rrrr rrrr    LDMIA Rnnn! {reg0-7}    ; rrrrrrrr = Lo reg-mask, inc Rnnn if Rnnn not in mask
                       LDMIA Rnnn  {reg0-7}    ; rrrrrrrr = Lo reg-mask, load Rnnn if Rnnn in mask
```

1101 – 0xDxxx Instructions
```
1101 0000 iiii iiii    BEQ label               ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 0001 iiii iiii    BNE label               ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 0010 iiii iiii    BHS/BCS label           ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 0011 iiii iiii    BLO/BCC label           ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 0100 iiii iiii    BPL label               ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 0101 iiii iiii    BMI label               ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 0110 iiii iiii    BVS label               ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 0111 iiii iiii    BVC label               ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 1000 iiii iiii    BHI label               ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 1001 iiii iiii    BLS label               ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 1010 iiii iiii    BGE label               ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 1011 iiii iiii    BLT label               ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 1100 iiii iiii    BGT label               ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 1101 iiii iiii    BLE label               ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 1110 xxxx xxxx    -                       ; undefined --> can be used for instruction emulation
1101 1111 iiii iiii    SVC #0biiiiiiii         ; supervisor call (formerly called SWI), arg ignored by HW
```

1110 – 0xExxx Instructions
```
1110 0iii iiii iiii    B label                 ; PC = IPC+4 + 0biiiiiiiiiii0       --> -2048/+2046 max
~~1110 1xxx xxxx xxxx    -                     ; 32 bit instructions~~
```

1111 – 0xFxxx Instructions
```
1111 0xii iiii iiii 11y1 ziii iiii iiii BL  label  ; LR=IPC+4,PC=IPC+4+0bXYZii...ii0,X,Y,Z=f(x,y,z), +/-16M
1111 0011 1110 1111 1000 dddd ssss ssss MRS Rdddd,S; Rdddd = special register S (encoded as 0bssssssss)
1111 0011 1000 mmmm 1000 1000 ssss ssss MSR S,Rmmmm; special register S (encoded as 0bssssssss) = Rmmmm
1111 0011 1011 1111 1000 1111 0100 1111 DSB        ; data synchronization barrier
1111 0011 1011 1111 1000 1111 0101 1111 DMB        ; data memory barrier
1111 0011 1011 1111 1000 1111 0110 1111 ISB        ; instruction synchronization barrier
~~1111 xxxx xxxx xxxx xxxx xxxx xxxx xxxx -        ; other 32 bit instructions~~
```

1) IPC is the PC of the current instruction (IPC+4 is given by the pipeline, IPC+2/+4 is the return address in the LR)
2) a dot means don't care, but must be set to 0.
3) <4321>: word, <21>: low half word, <1>: low byte, <0001>: zero extend byte, <sss1>: sign extend byte, etc.
4) Undefined instructions can be used to emulate instructions (they trigger the undefined exception).
5) Unpredictable instructions do any unpredictable actions and are therefore illegal instructions.
6) Unallocated codes are undefined unless they are explicitly marked as unpredictable.
7) CBZ, CBNZ, IT are the only 16 bit instructions which are not part of Cortex-M0/M1 Thumb code.
8) BL, DMB, DSB, ISB, MRS, MSR are the only 32 bit instructions as part of the Cortex-M0/M1 instruction set.

ARM Cond. Jumps

School of Engineering
InES Institute of Embedded Systems

Flag- Dependent

Symbol	Condition	Flag
EQ	Equal	Z == 1
NE	Not equal	Z == 0
CS	Carry set	C == 1
CC	Carry clear	C == 0
MI	Minus/negative	N == 1
PL	Plus/positive or zero	N == 0
VS	Overflow	V == 1
VC	No overflow	V == 0

Arithmetic - unsigned: higher and lower

Symbol	Condition	Flag
EQ	Equal	Z == 1
NE	Not equal	Z == 0
HS (=CS)	Unsigned higher or same	C == 1
LO (=CC)	Unsigned lower	C == 0
HI	Unsigned higher	C == 1 and Z == 0
LS	Unsigned lower or same	C == 0 or Z == 1

Arithmetic - signed: greater and less

Symbol	Condition	Flag
EQ	Equal	Z == 1
NE	Not equal	Z == 0
MI	Minus/negative	N == 1
PL	Plus/positive or zero	N == 0
VS	Overflow	V == 1
VC	No overflow	V == 0
GE	Signed greater than or equal	N == V
LT	Signed less than	N != V
GT	Signed greater than	Z == 0 and N == V
LE	Signed less than or equal	Z == 1 or N != V

C Reference Card (ANSI)

Program Structure/Functions

type fnc(*type*$_1$, ...);	function prototype
type name;	variable declaration
int main(void) {	main routine
declarations	local variable declarations
statements	
}	
type fnc(*arg*$_1$, ...) {	function definition
declarations	local variable declarations
statements	
return *value*;	
}	
/* */	comments
int main(int argc, char *argv[])	main with args
exit(*arg*);	terminate execution

C Preprocessor

include library file	#include <*filename*>
include user file	#include "*filename*"
replacement text	#define *name text*
replacement macro	#define *name*(*var*) *text*
Example. #define max(A,B) ((A)>(B) ? (A) : (B))	
undefine	#undef *name*
quoted string in replace	#
Example. #define msg(A) printf("%s = %d", #A, (A))	
concatenate args and rescan	##
conditional execution	#if, #else, #elif, #endif
is *name* defined, not defined?	#ifdef, #ifndef
name defined?	defined(*name*)
line continuation char	\

Data Types/Declarations

character (1 byte)	char
integer	int
real number (single, double precision)	float, double
short (16 bit integer)	short
long (32 bit integer)	long
double long (64 bit integer)	long long
positive or negative	signed
non-negative modulo 2^m	unsigned
pointer to int, float,...	int*, float*,...
enumeration constant	enum *tag* {*name*$_1$=*value*$_1$,...};
constant (read-only) value	*type* const *name*;
declare external variable	extern
internal to source file	static
local persistent between calls	static
no value	void
structure	struct *tag* {...};
create new name for data type	typedef *type name*;
size of an object (type is size_t)	sizeof *object*
size of a data type (type is size_t)	sizeof(*type*)

Initialization

initialize variable	*type name*=*value*;
initialize array	*type name*[]={*value*$_1$,...};
initialize char string	char *name*[]="*string*";

Constants

suffix: long, unsigned, float	65536L, -1U, 3.0F
exponential form	4.2e1
prefix: octal, hexadecimal	0, 0x or 0X
Example. 031 is 25, 0x31 is 49 decimal	
character constant (char, octal, hex)	'a', '\ooo', '\x*hh*'
newline, cr, tab, backspace	\n, \r, \t, \b
special characters	\\\\, \?, \\', \\"
string constant (ends with '\0')	"abc...de"

Pointers, Arrays & Structures

declare pointer to *type*	*type* **name*;
declare function returning pointer to *type*	*type* *f();
declare pointer to function returning *type*	*type* (*pf)();
generic pointer type	void *
null pointer constant	NULL
object pointed to by *pointer*	**pointer*
address of object *name*	&*name*
array	*name*[*dim*]
multi-dim array	*name*[*dim*$_1$][*dim*$_2$]...

Structures

```
struct tag {          structure template
  declarations        declaration of members
};
```

create structure	struct *tag name*
member of structure from template	*name*.*member*
member of pointed-to structure	*pointer* -> *member*
Example. (*p).x and p->x are the same	
single object, multiple possible types	union
bit field with *b* bits	unsigned *member*: *b*;

Operators (grouped by precedence)

struct member operator	*name*.*member*
struct member through pointer	*pointer*->*member*
increment, decrement	++, --
plus, minus, logical not, bitwise not	+, -, !, ~
indirection via pointer, address of object	**pointer*, &*name*
cast expression to type	(*type*) *expr*
size of an object	sizeof
multiply, divide, modulus (remainder)	*, /, %
add, subtract	+, -
left, right shift [bit ops]	<<, >>
relational comparisons	>, >=, <, <=
equality comparisons	==, !=
and [bit op]	&
exclusive or [bit op]	^
or (inclusive) [bit op]	|
logical and	&&
logical or	||
conditional expression	*expr*$_1$? *expr*$_2$: *expr*$_3$
assignment operators	+=, -=, *=, ...
expression evaluation separator	,

Unary operators, conditional expression and assignment operators group right to left; all others group left to right.

Flow of Control

statement terminator	;
block delimiters	{ }
exit from switch, while, do, for	break;
next iteration of while, do, for	continue;
go to	goto *label*;
label	*label*: statement
return value from function	return *expr*

Flow Constructions

```
if statement        if (expr1)  statement1
                    else if (expr2)  statement2
                    else  statement3
while statement     while (expr)
                      statement
for statement       for (expr1; expr2; expr3)
                      statement
do statement        do   statement
                    while(expr);
switch statement    switch (expr) {
                       case const1: statement1 break;
                       case const2: statement2 break;
                       default: statement
                    }
```

ANSI Standard Libraries

<assert.h>	<ctype.h>	<errno.h>	<float.h>	<limits.h>
<locale.h>	<math.h>	<setjmp.h>	<signal.h>	<stdarg.h>
<stddef.h>	<stdio.h>	<stdlib.h>	<string.h>	<time.h>

Character Class Tests <ctype.h>

alphanumeric?	isalnum(c)
alphabetic?	isalpha(c)
control character?	iscntrl(c)
decimal digit?	isdigit(c)
printing character (not incl space)?	isgraph(c)
lower case letter?	islower(c)
printing character (incl space)?	isprint(c)
printing char except space, letter, digit?	ispunct(c)
space, formfeed, newline, cr, tab, vtab?	isspace(c)
upper case letter?	isupper(c)
hexadecimal digit?	isxdigit(c)
convert to lower case	tolower(c)
convert to upper case	toupper(c)

String Operations <string.h>

s is a string; cs, ct are constant strings

length of s	strlen(s)
copy ct to s	strcpy(s,ct)
concatenate ct after s	strcat(s,ct)
compare cs to ct	strcmp(cs,ct)
only first n chars	strncmp(cs,ct,n)
pointer to first c in cs	strchr(cs,c)
pointer to last c in cs	strrchr(cs,c)
copy n chars from ct to s	memcpy(s,ct,n)
copy n chars from ct to s (may overlap)	memmove(s,ct,n)
compare n chars of cs with ct	memcmp(cs,ct,n)
pointer to first c in first n chars of cs	memchr(cs,c,n)
put c into first n chars of s	memset(s,c,n)

C Reference Card (ANSI)

Input/Output `<stdio.h>`

Standard I/O

standard input stream	`stdin`
standard output stream	`stdout`
standard error stream	`stderr`
end of file (type is `int`)	`EOF`
get a character	`getchar()`
print a character	`putchar(`*chr*`)`
print formatted data	`printf("`*format*`",`*arg*$_1$`,...)`
print to string `s`	`sprintf(s,"`*format*`",`*arg*$_1$`,...)`
read formatted data	`scanf("`*format*`",&`*name*$_1$`,...)`
read from string `s`	`sscanf(s,"`*format*`",&`*name*$_1$`,...)`
print string `s`	`puts(s)`

File I/O

declare file pointer	`FILE *`*fp*`;`
pointer to named file	`fopen("`*name*`","`*mode*`")`
modes: `r` (read), `w` (write), `a` (append), `b` (binary)	
get a character	`getc(`*fp*`)`
write a character	`putc(`*chr*`,`*fp*`)`
write to file	`fprintf(`*fp*`,"`*format*`",`*arg*$_1$`,...)`
read from file	`fscanf(`*fp*`,"`*format*`",`*arg*$_1$`,...)`
read and store `n` elts to `*ptr`	`fread(*ptr,eltsize,n,`*fp*`)`
write `n` elts from `*ptr` to file	`fwrite(*ptr,eltsize,n,`*fp*`)`
close file	`fclose(`*fp*`)`
non-zero if error	`ferror(`*fp*`)`
non-zero if already reached EOF	`feof(`*fp*`)`
read line to string `s` (< `max` chars)	`fgets(s,max,`*fp*`)`
write string `s`	`fputs(s,`*fp*`)`

Codes for Formatted I/O: `"%-+ 0`*w.pmc*`"`

`-`	left justify
`+`	print with sign
space	print space if no sign
`0`	pad with leading zeros
w	min field width
p	precision
m	conversion character: `h` short, `l` long, `L` long double
c	conversion character:

`d,i`	integer	`u`	unsigned
`c`	single char	`s`	char string
`f`	double (`printf`)	`e,E`	exponential
`f`	float (`scanf`)	`lf`	double (`scanf`)
`o`	octal	`x,X`	hexadecimal
`p`	pointer	`n`	number of chars written
`g,G`	same as `f` or `e,E` depending on exponent		

Variable Argument Lists `<stdarg.h>`

declaration of pointer to arguments	`va_list` *ap*`;`
initialization of argument pointer	`va_start(`*ap*`,`*lastarg*`);`
lastarg is last named parameter of the function	
access next unnamed arg, update pointer	`va_arg(`*ap*`,`*type*`)`
call before exiting function	`va_end(`*ap*`);`

Standard Utility Functions `<stdlib.h>`

absolute value of `int n`	`abs(n)`
absolute value of `long n`	`labs(n)`
quotient and remainder of `int`s `n,d`	`div(n,d)`
returns structure with `div_t.quot` and `div_t.rem`	
quotient and remainder of `long`s `n,d`	`ldiv(n,d)`
returns structure with `ldiv_t.quot` and `ldiv_t.rem`	
pseudo-random integer `[0,RAND_MAX]`	`rand()`
set random seed to `n`	`srand(n)`
terminate program execution	`exit(status)`
pass string `s` to system for execution	`system(s)`

Conversions

convert string `s` to double	`atof(s)`
convert string `s` to integer	`atoi(s)`
convert string `s` to long	`atol(s)`
convert prefix of `s` to `double`	`strtod(s,&endp)`
convert prefix of `s` (base `b`) to `long`	`strtol(s,&endp,b)`
same, but `unsigned long`	`strtoul(s,&endp,b)`

Storage Allocation

allocate storage	`malloc(size), calloc(nobj,size)`
change size of storage	`newptr = realloc(ptr,size);`
deallocate storage	`free(ptr);`

Array Functions

search `array` for `key`	`bsearch(key,array,n,size,cmpf)`
sort `array` ascending order	`qsort(array,n,size,cmpf)`

Time and Date Functions `<time.h>`

processor time used by program	`clock()`
Example. `clock()/CLOCKS_PER_SEC` is time in seconds	
current calendar time	`time()`
`time`$_2$`-time`$_1$ in seconds (`double`)	`difftime(time`$_2$`,time`$_1$`)`
arithmetic types representing times	`clock_t,time_t`
structure type for calendar time comps	`struct tm`

`tm_sec`	seconds after minute
`tm_min`	minutes after hour
`tm_hour`	hours since midnight
`tm_mday`	day of month
`tm_mon`	months since January
`tm_year`	years since 1900
`tm_wday`	days since Sunday
`tm_yday`	days since January 1
`tm_isdst`	Daylight Savings Time flag

convert local time to calendar time	`mktime(tp)`
convert time in `tp` to string	`asctime(tp)`
convert calendar time in `tp` to local time	`ctime(tp)`
convert calendar time to GMT	`gmtime(tp)`
convert calendar time to local time	`localtime(tp)`
format date and time info	`strftime(s,smax,"`*format*`",tp)`
`tp` is a pointer to a structure of type `tm`	

Mathematical Functions `<math.h>`

Arguments and returned values are `double`

trig functions	`sin(x), cos(x), tan(x)`
inverse trig functions	`asin(x), acos(x), atan(x)`
arctan(y/x)	`atan2(y,x)`
hyperbolic trig functions	`sinh(x), cosh(x), tanh(x)`
exponentials & logs	`exp(x), log(x), log10(x)`
exponentials & logs (2 power)	`ldexp(x,n), frexp(x,&e)`
division & remainder	`modf(x,ip), fmod(x,y)`
powers	`pow(x,y), sqrt(x)`
rounding	`ceil(x), floor(x), fabs(x)`

Integer Type Limits `<limits.h>`

The numbers given in parentheses are typical values for the constants on a 32-bit Unix system, followed by minimum required values (if significantly different).

`CHAR_BIT`	bits in `char`	(8)
`CHAR_MAX`	max value of `char`	(`SCHAR_MAX` or `UCHAR_MAX`)
`CHAR_MIN`	min value of `char`	(`SCHAR_MIN` or 0)
`SCHAR_MAX`	max `signed char`	(+127)
`SCHAR_MIN`	min `signed char`	(−128)
`SHRT_MAX`	max value of `short`	(+32,767)
`SHRT_MIN`	min value of `short`	(−32,768)
`INT_MAX`	max value of `int`	(+2,147,483,647) (+32,767)
`INT_MIN`	min value of `int`	(−2,147,483,648) (−32,767)
`LONG_MAX`	max value of `long`	(+2,147,483,647)
`LONG_MIN`	min value of `long`	(−2,147,483,648)
`UCHAR_MAX`	max `unsigned char`	(255)
`USHRT_MAX`	max `unsigned short`	(65,535)
`UINT_MAX`	max `unsigned int`	(4,294,967,295) (65,535)
`ULONG_MAX`	max `unsigned long`	(4,294,967,295)

Float Type Limits `<float.h>`

The numbers given in parentheses are typical values for the constants on a 32-bit Unix system.

`FLT_RADIX`	radix of exponent rep	(2)
`FLT_ROUNDS`	floating point rounding mode	
`FLT_DIG`	decimal digits of precision	(6)
`FLT_EPSILON`	smallest x so $1.0\text{f} + x \neq 1.0\text{f}$	(1.1E − 7)
`FLT_MANT_DIG`	number of digits in mantissa	
`FLT_MAX`	maximum `float` number	(3.4E38)
`FLT_MAX_EXP`	maximum exponent	
`FLT_MIN`	minimum `float` number	(1.2E − 38)
`FLT_MIN_EXP`	minimum exponent	
`DBL_DIG`	decimal digits of precision	(15)
`DBL_EPSILON`	smallest x so $1.0 + x \neq 1.0$	(2.2E − 16)
`DBL_MANT_DIG`	number of digits in mantissa	
`DBL_MAX`	max `double` number	(1.8E308)
`DBL_MAX_EXP`	maximum exponent	
`DBL_MIN`	min `double` number	(2.2E − 308)
`DBL_MIN_EXP`	minimum exponent	