Test if number is even or odd (look at last bit)

```
tst r5, #1 ;if even the Z flag is set to 0
beq even
```

Compare signs of two signed numbers

```
;r0=number1, r2=number2
;do a XOR operation to check signs!
;and then , since XOR returns 1 with different signs, invert the result with
mvn
eor r3, r0, r2
;invert it, so: same sign -> 1, different sign -> 0
mvn r3, r3 ;mvn inverts ALL the bits
;take the MSB of the eor between the bits of the respective numbers
lsr r3, r3, #31
;r3 stores the bit of the flag (either 0x1 or 0x0)
```

Two's Complement of a 32 bit number

```
;r2 stores the number
mvn r2, r2
add r2, r2, #1
```

Two's Complement of a 64 bit number (in two registers)

```
;r0 UPPER 32 BITS
;r1 LOWER 32 BITS

;two's complement of both upper and lower bits
mvn r0, r0
mvn r1, r1
;add 1 to the lower 32 bits
;if the lower 32 bits are all 1 -> overflow -> this means we're gonna add 1 to
the ;upper 32 bits instead
adds r1, r1, #1
;check if overflow of lower 32 bits
bvc no_overflow ;no overflow
```

```
;overflow: propagate the sum of 1 to the upper 32 bits
add r0, r0, #1
```

Remainder of Unsigned Division

```
LDR R0, =10     ; Dividendo (numeratore)
LDR R1, =3      ; Divisore (denominatore)
UDIV R2, R0, R1 ; R2 = R0 / R1 (Quoziente)
MLS R3, R2, R1, R0 ; R3 = R0 - (R2 * R1) (Calcolo del Resto)
```

Remainder of Signed Division

```
LDR R0, =-10    ; Dividendo (numeratore)
LDR R1, =3      ; Divisore (denominatore)
SDIV R2, R0, R1 ; R2 = R0 / R1 (Quoziente con segno)
MLS R3, R2, R1, R0 ; R3 = R0 - (R2 * R1) (Calcolo del Resto)
```

Count bits set to 1 (Brian Kernighan)

```
brianKernighan  PROC
                                ;r0: number
                                stmfd sp!, {r4-r5, lr}

                                mov r4, #0      ;counter

ciclo                   ;check if the number is not zero
                                cmp r0, #0
                                beq endAlgo

                                ;do n = n AND (n-1)
                                sub r5, r0, #1
                                and r0, r0, r5
                                ;increment counter
                                add r4, r4, #1
                                b ciclo

endAlgo                 mov r0, r4
                                ldmfd sp!, {r4-r5, pc}
                                ENDP
```

Check if a number is Prime, linear algorithm

```
isPrime                 PROC
                                stmfd sp!, {r4-r8, r10-r11, lr}

                                ;r0: number to test wether it's prime or not

                                cmp r0,#0
                                beq not_prime
                                cmp r0, #3
                                ble prime

                                mov r1, r0              ;original number
                                sub r2, r1, #1  ;test number
                                ;while test number > 1: perform
original_number % test_number, it it's 0 -> prime
                                ;if test_number reaches 1 -> not prime
                                ;linear complexity

while                   ;check test_number > 1
                                cmp r2, #1
                                ble prime

                                ;perform r1 % r2
                                bl mod
                                ;result in r0
                                ;if remainder == 0 -> not prima
                                cmp r0, #0
                                beq not_prime

                                ;test_number --
                                sub r2, r2, #1
                                ;loop back
                                b while

not_prime               mov r0, #0
                                ldmfd sp!, {r4-r8, r10-r11, pc}

prime                   mov r0, #1
                                ldmfd sp!, {r4-r8, r10-r11, pc}
                                ENDP

mod                             PROC
                                ;calculate r1 % r2
                                udiv r3, r1, r2 ;r3 = r1/r2
                                mls r0, r3, r2, r1
                                ;result in r0
```

```
                                bx lr

                                ENDP
```

2^i

```
; i in r0
mov r1, #1
lsl r1, r1, r0 -> 1*2^i
```

2^-i

```
; i in r0
mov r1, #1
lsr r1, r1, r0 -> 1*2^(-i) = 1 / (2^i)
```

Check Lowercase Letter

```
check_lower             PROC

                                cmp r0, #'a'
                                blt nope
                                cmp r0, #'z'
                                bgt nope

                                mov r0, #1
                                bx lr


nope                    mov r0, #0
                                bx lr
                                ENDP
```

Get max value in a vector of dimension N

```
EXPORT get_max
get_max FUNCTION
        ;R0=Vett
        ;R1=dim

        MOV   r12, sp
        STMFD sp!,{r4-r8,r10-r11,lr}
```

```
        LDR R6, [R0], #4         ; Carica il primo elemento dell'array in R6
(massimo iniziale)
        SUBS R1, R1, #1          ; Decrementa la dimensione (R1 = dim - 1)
        BLE exitMax              ; Se R1 <= 0, salta direttamente all'uscita

loopMax
        LDR R4, [R0], #4         ; Carica l'elemento corrente in R4 e avanza
il puntatore R0
        CMP R4, R6               ; Confronta l'elemento corrente (R4) con il
massimo attuale (R6)
        MOVGT R6, R4             ; Se R4 > R6, aggiorna il massimo in R6
        SUBS R1, R1, #1          ; Decrementa il contatore R1
        BGT loopMax              ; Ripeti finch R1 > 0

exitMax
        MOV R0, R6               ; Salva il massimo trovato in R0 (registro di
ritorno)

        LDMFD sp!,{r4-r8,r10-r11,pc}
        ENDFUNC
```

Get min value in a vector of dimension N

```
get_min FUNCTION
        ;R0=Vett (puntatore all'array)
        ;R1=dim  (dimensione dell'array)

        MOV   r12, sp
        STMFD sp!,{r4-r8,r10-r11,lr} ; Salva i registri callee-saved nello
stack

        LDR R6, [R0], #4         ; Carica il primo elemento dell'array in R6
(minimo iniziale)
        SUBS R1, R1, #1          ; Decrementa la dimensione (R1 = dim - 1)
        BLE exitMin              ; Se R1 <= 0, salta direttamente all'uscita

loopMin
        LDR R4, [R0], #4         ; Carica l'elemento corrente in R4 e avanza
il puntatore R0
        CMP R4, R6               ; Confronta l'elemento corrente (R4) con il
minimo attuale (R6)
        MOVLT R6, R4             ; Se R4 < R6, aggiorna il minimo in R6
        SUBS R1, R1, #1          ; Decrementa il contatore R1
        BGT loopMin              ; Ripeti finch R1 > 0
```

```
exitMin
        MOV R0, R6              ; Salva il minimo trovato in R0 (registro di
ritorno)

        LDMFD sp!,{r4-r8,r10-r11,pc} ; Ripristina i registri e ritorna
        ENDFUNC
```

Check if value is within a range

```
value_is_in_a_range FUNCTION

    ; R0 = VALUE
    ; R1 = MIN
    ; R2 = MAX
    ; R0 returns:
    ;   - 1 if MIN <= VALUE <= MAX
    ;   - 0 otherwise

    ; Save current SP for faster access to parameters in the stack
    MOV     r12, sp
    ; Save volatile registers
    STMFD   sp!, {r4-r8, r10-r11, lr}

    ; Compare VALUE with MIN
    CMP     R0, R1
    BLO     outOfRange      ; If VALUE < MIN, branch to outOfRange

    ; Compare VALUE with MAX
    CMP     R0, R2
    BHI     outOfRange      ; If VALUE > MAX, branch to outOfRange

    ; If VALUE is within the range
    MOV     R0, #1          ; Set R0 to 1 (true)
    B       exitFuncV         ; Branch to exit

outOfRange
    MOV     R0, #0          ; Set R0 to 0 (false)

exitFuncV
    ; Restore volatile registers
    LDMFD   sp!, {r4-r8, r10-r11, pc}
```

```
    ENDFUNC
```