SC 4/3/2026
03/03/26 23:16

DEFINIZIONE DI FUNZIONE IN ASM

Per definire una funzione (Es. "main") utilizziamo una "etichetta" ("label")

```
nome_etichetta:
```

Per definire una funzione "somma"

```
SOMMA :
```

La visibilità di una etichetta è di norma limitata al file sorgente (.s) in cui è definita.

Per rendere la funzione visibile da altri file oggetto usiamo la direttiva "**.globl**"

```
.globl SOMMA

SOMMA :
```

Esempio:

```
int f() {
    int x;      dichiarazione
    x = 10;
    return x;
}
```

f.c

```
.globl f
f:
#  x → eax
```

f.s

FORMATO ISTRUZIONI ASM

Ogni istruzione è caratterizzata da:

- nome
- un certo numero di operandi
- semantica

Riguardo agli operandi:

Ⓐ Istruzioni senza operandi (Es.: ret, nop, ...)

Ⓑ Istruzioni unarie

$$\underset{\substack{\uparrow\\ \text{operando destinazione}}}{\mathrm{OP}\ \mathrm{D}} \quad | \quad \mathrm{D} \leftarrow \mathrm{OP}\ \mathrm{D}$$

Ⓒ Istruzioni binarie → NOTAZIONE AT&T in alternativa a notazione Intel

OPt S, D | D ← D OP S

operando sorgente → S

operando destinazione → D

Ⓓ Istruzioni Ternarie

TAGLIA DEI DATI

Per ogni istruzione dobbiamo definire la taglia dei dati su cui andrà ad operare seguendo la sua semantica

b	1 byte (8 bit)	es:	char
w	2 byte (16 bit)	es:	short
l	4 byte (32 bit)	es:	int, T*

ISTRUZIONI PER LO SPOSTAMENTO DI DATI

mov S, D | D ← S

operando sorgente S
- → immediato $ (es.: $10)
- → registro % (es.: %eax)
- → memoria

operando destinazione D
- → registro %
- → memoria

Regola: non si usano mai valori immediati per l'operando D

Regola: non è possibile usare l'indirizzamento a memoria per entrambi gli operandi

Esempio:

```
int f() {
    int x;      dichiarazione
    x = 10;     assegnazione
    return x;
}
```

f_c

```
.globl f
f:
# x → eax
movl $10, %eax
ret
```

f.s

OPERAZIONI ARITMETICO/LOGICHE

add	S, D	|	D ← D + S	
sub	S, D	|	D ← D - S	
mul	S, D	|	D ← D * S	senza segno
imul	S, D	|	D ← D * S	con segno
and	S, D	|	D ← D & S	bit a bit
or	S, D	|	D ← D | S	〃
xor	S, D	|	D ← D ^ S	〃
neg	D	|	D ← -D	
not	D	|	D ← ~D	(!D)
inc	D	|	D ← D+1	
dec	D	|	D ← D-1	

USO DI PARAMETRI NELLE FUNZIONI

f.c

```
int f(int x) {
        return x+1;
}
```

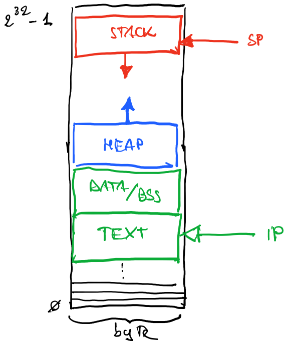
f_eq.c

```
int f(int x) {
        int a;
        a = x;
        a++;
        return a;
}
```

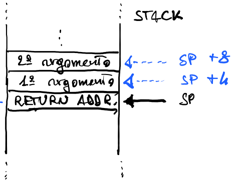
f.s

```
    .globl f
f:
    # a → eax
    movl ?, %eax
    incl %eax
    ret
```

STRUTTURA DEI DATI IN MEMORIA



Cosa accade quando viene chiamata la funzione f?



è il puntatore alla prossima istruzione della funzione chiamante da eseguire dopo la ret

Regola: Tutti i parametri attuali (argomenti) occupano 4 byte ciascuno nello stack indipendentemente dal loro Tipo

ACCESSO IN MEMORIA CON SPIAZZAMENTO (SEMPLIFICATA)

d (addr)

addr: puntatore a memoria (indirizzo)

d: spiazzamento

Es.: SP + 4 ⟹ 4 (%esp)

f.s

```
.globl f
f:

    # a → eax
    movl 4(%esp), %eax
    incl %eax
    ret
```

MOLTIPLICAZIONE

Moltiplicazione senza segno

mul S | D ← D * S

D: %edx : %eax

D (in D * S): %eax

S: registro 32bit / immediato

Es.: movl $2, %eax

mull $5

eax	edx
0 ... 010	?
0 ... 01010	0 ... 0

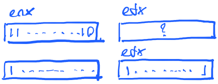
Moltiplicazione con segno

imul S | D ← D*S

%edx : %eax   %eax   r32/imm32

Es.: movl $-2 , %eax

imull $5



DIVISIONE

IDIV S | D ← D/S

EAX , EDX   EDX : EAX

idivl r32/imm32 | EAX ← EDX : EAX / S

EDX ← EDX : EAX % S

quoziente

modulo

GESTIONE DEL FLUSSO DI ESECUZIONE

SALTO INCONDIZIONATO

```
---
---
goto L;        =>        jmp L
---                      ---
---                      ---
L:                       L:
---                      ---
---                      ---
```

SALTO CONDIZIONATO

Le istruzioni di salto condizionato in ASM si sviluppano in due fasi:

① Esecuzione di istruzione aritmetico /logica che modifica EFLAGS

② Esecuzione di un salto condizionato al valore di EFLAGS

La proprietà valutata in ② per il salto è detta CONDITION CODE (cc)

Es.: risultato R

CONDITION CODE (cc)	PROPRIETÀ DI R	
e , z	R == 0	
ne , nz	R != 0	
g (>)	R > 0	con segno
l (<)	R < 0	
ge (>=)	R >= 0	
le (<=)	R <= 0	
a	R > 0	senza segno
b	R < 0	
ae	R >= 0	
be	R <= 0	

Il salto condizionato è implementato da

J cc L

CONDITION CODE

```
Es.:    subl %eax , %ecx        # R = ECX - EAX
        je L                    # EAX == ECX => R == 0
Es.:    subl %eax , %ecx
        jge L                   # ECX ≥ EAX => R ≥ 0
```