

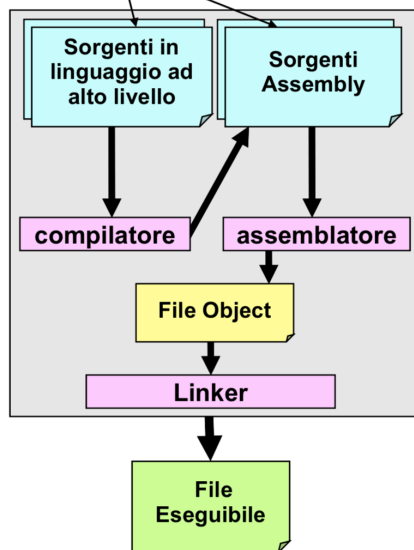
ASSEMBLY x86

Linguaggio testuale equivalente a linguaggio macchina, ogni programma deve essere tradotto in assembly per essere interpretato dalla macchina.

Le istruzioni eseguibili da una CPU sono definite dal **ISA** (instruction set architecture), differente per ogni tipo di architettura.

in questo caso si fa riferimento ad architettura x86 versione AMD64 (AT&T).

File creati dall'utente



a = b + c

```
movw b, %ax
movw c, %bx
addw %ax, %bx
movw %bx, a
```

```
000101..0101001
1011101..010100
01011..11101010
010..1110101010
```

OSS per disassemblare un programma ed ottenere il codice assembly equivalente si può usare il comando da terminale:

OBJDUMP *nome_file* -d (oppure -S)

Scheletro programma assembly

```
1 .org [INDIRIZZO CARICAMENTO]
2
3 .data
4
5 # Dichiarazione costanti e variabili globali
6
7 .text
8
9     # Corpo del programma
10
11 hlt # Per arrestare l'esecuzione
```

Solitamente si iniziano i programmi con **.globl**

HLT istruzione che serve per mandare il processore in IDLE alla fine del programma

Direttive assembly

.org *address, fill* => imposta location counter (indirizzo assoluto), imposta byte a fill

.equ *symbol, expression* => definisce una costante (equivalente a *symbol = expression*)

.byte *expression* => riserva memoria di (*expression**byte)

.word *expression* => riserva memoria di (*expression**word)

.long *expression* => riserva memoria di (*expression**long)

.quad *expression* => riserva memoria di (*expression**quad)

.ascii "*string*" => riserva memoria per vettore di caratteri impostato a "*string*"

.fill *repeat, size, value* => riserva memoria di (*repeat* celle * dimensione *size*) e le imposta a *value*. default *size* = 1, *value* = 0

.comm *symbol, length* => dichiara area di memoria con nome *symbol* e dimensione *length* nella sezione bss del programma

.driver *idn* / **.handler** *idn* => identifica inizio routine di servizio associata a *idn*

.data => delimita la sezione contenente la sezione data del programma

.text => delimita la sezione contenente la sezione testo del programma

OSS le assegnazioni in assembly si effettuano tramite ":"

Tipi di dato

C	SUFFISSO	BYTE
CHAR	B	1
SHORT	W	2
INT	L	4
DOUBLE	Q	8

Indicato con x nelle istruzioni

l'accesso a memoria avviene in due passi:

1. Si salva l'indirizzo della locazione di memoria in un registro
2. Si accede al contenuto dell'indirizzo tramite l'operando (%registro) o tramite l'operando (%base, %indice, %scala)

OSS sono quindi equivalenti ai puntatori alle locazioni di memoria in C

Operandi

TIPO	SUFFISSO
registro	%
immediato	\$
Memoria	(%registro)
Memoria (aritmetica dei puntatori)	Immediato (%base, %indice, %scala)
Esadecimale (indirizzo)	0x

Scala può valere solo: 1, 2, 4, 8

immediato(Base, indice, scala) => **(base + (indice * scala)) + immediato**

Con MOV => base[indice + imm] (imm deve essere multiplo di scala che deve corrispondere con il tipo della base)

I registri general purpose (64bit)

64 bit	32 bit	16bit	8bit	Nome
%RSP	%ESP	%SP		Stack pointer
%RBP	%EBP	%BP		Base pointer
%RAX	%EAX	%AX	%AL	Accumulatore
%RBX	%EBX	%BX	%BL	Base
%RCX	%ECX	%CX	%CL	Contatore
%RDX	%EDX	%DX	%DL	Dati
%RSI	%ESI	%SI		sorgente
%RDI	%EDI	%DI		Destinazione
%R8	%R8D	%R8W	%R8B	
%R9	%R9D	%R9W	%R9B	
%R10	%R10D	%R10W	%R10B	
%R11	%R11D	%R11W	%R11B	
%R12	%R12D	%R12W	%R12B	
%R13	%R13D	%R13W	%R13B	
%R14	%R14D	%R14W	%R14B	
%R15	%R15D	%R15W	%R15B	

I registri in **blu** sono “**callee-save**”, gli altri caller-save

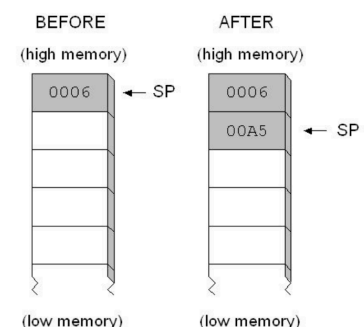
SYSTEM V ABI

Definisce le convenzioni per l'utilizzo dei registri

- il valore di ritorno della funzione si salva in **%RAX**
- **%RBP** è il puntatore alla finestra di stack, all'inizio di ogni programma si inizializza allo stesso valore di RSP in modo da usarlo come puntatore all'indirizzo iniziale dello stack corrispondente, dove sono salvati in parametri in eccesso rispetto ai registri (PUSH)
- I primi 6 parametri della funzione devono essere salvati in RDI, RSI, RDX, RCX, R8, R9 nel caso di system call si usa R10 al posto di RCX
- Registri **callee-save** => prima di poter utilizzare uno di questi registri bisogna salvarne il contenuto sulla stack (PUSH) per poi ripristinarlo (POP) alla fine del programma.
Serve per evitare la perdita di dati nella funzione chiamante.

PUSHx S	$\%rsp = \%rsp - (tipo\ di\ x)$ $(\%rsp) = S$
POPx D	$D = \%rsp$ $(\%rsp) = \%rsp + (tipo\ di\ x)$

```
push 0006h
push 00A5h
```



OSS PUSH e POP possono essere usate in generale per salvare sullo stack ogni tipo di variabile in eccesso al numero di registri

ATTENZIONE le istruzioni modificano anche %rsp, cambiano tutti gli offset collegati

Istruzioni logico-aritmetiche

ADDx S, D	$D = D + S$	
SUBx S, D	$D = D - S$	
IMULx S, D	$D = D * S$	(operazione molto costosa)
XORx S, D	$D = D \wedge S$	
ORx S, D	$D = D \vee S$	
ANDx S, D	$D = D \& S$	
NOTx D	$D = \sim D$	
NEGx D	$D = -D$	complemento a 2
INCx D	$D = D + 1$	
DECx D	$D = D - 1$	
LEAx S, D	$D = \&S$	load effective address

ATTENZIONE non si possono avere 2 operandi a memoria contemporaneamente

Shift

Sposta tutti i bit di un registro destinazione del numero di posizioni date dalla sorgente permettono moltiplicazioni e divisioni ottimizzate con potenze di 2

shift	logico	aritmetico	Shift logico aggiunge 0 in posizioni vuote Shift aritmetico conserva il segno
SX (left)	SHLx S, D	SALx S, D	
DX (right)	SHRx S, D	SARx S, D	

Ottimizzazioni

1. Per settare a 0 un registro invece di MOV si può usare XOR

x = 0	<code>movl \$0, %eax</code>	<code>xorl %eax, %eax</code>
--------------	-----------------------------	------------------------------

2. Invece di usare IMUL si possono usare addizioni consecutive, LEA o shift

x = x*2	<code>imul \$2, %eax</code>	<code>addl %eax, %eax</code> (loop per + di 2)
x = x*4	<code>imul \$4, %eax</code>	<code>xorl %eax, %eax</code> <code>leal (&ecx, %eax, 4)</code>
x = x*16	<code>imul \$16, %eax</code>	<code>sal \$4, %eax</code> (solo con potenze di 2) <code>#4 = log₂(16)</code>

3. Per equazioni è possibile usare la LEA invece di più istruzioni

x = z + 2y - 7	<code>leal -7(%ecx, %edx, 2), %eax.</code>	(lea non effettua side effect su flags)
-----------------------	--------------------------------------------	-----------------------------------------

4. Moltiplicazione con shift-and-add (codice su appendice A)

- Shift a destra del moltiplicando
- IF (carry = 1) risultato += moltiplicatore
- Shift a sinistra del moltiplicatore
- Goto 1 (finché moltiplicando = 0)

If - else (salti)

Le istruzioni if - else vengono implementate in assembly tramite confronti e salti, prima si effettua il test con CMP, dopodiché si usa Jcc per verificare condizione e nel caso saltare.

CMPx S, D	calcola D - S
Jcc etichetta	salto condizionato su etichetta, dipende da cc
JMP etichetta	salto su etichetta, no condizioni

ATTENZIONE in questo modo si ha un sistema equivalente al goto, quindi nel caso di normali condizioni di if bisogna ribaltare l'espressione

Condizioni

TEST	CONDIZIONE (con segno)	CONDIZIONE (senza segno)
=	e	
!=	ne	
<	l	b
≤	le	be
>	g	a
≥	ge	ae

C	Assembly
If (x ≤ y) goto L	# x=%eax y=%ecx cmpl %ecx, %eax jle L

Confronti tramite flags

Il calcolatore esegue i confronti tra le variabili tramite un'operazione di sottrazione per poi controllare lo stato dei flags e conoscere il risultato del confronto

Condition	Aritmetica non segnata	Aritmetica segnata
dest > source	CF = 0 and ZF = 0	ZF = 0 and SF = 0F
dest ≥ source	CF = 0	SF = 0F
dest = source	ZF = 1	ZF = 1
dest ≤ source	CF = 1 or ZF = 1	ZF = 1 or SF ≠ 0F
dest < source	CF = 1	SF ≠ 0F
dest ≠ source	ZF = 0	ZF = 0

OSS oltre alle normali condizioni di test si può usare direttamente lo stato dei flag per effettuare operazioni salto, tramite:

Jnome_flag

JNnome_flag (la N permette di avere condizione inversa)

Cicli while-for

In entrambi i casi si imposta la condizione di ciclo con CMP e Jcc, se condizione fallisce si eseguono operazioni fino a raggiungere JMP che salta nuovamente al test CMP

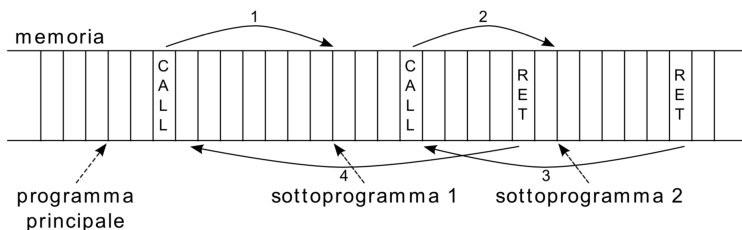
Nel caso di ciclo for la condizione si effettua su un contatore decrementato ad ogni loop

Le subroutine

Permettono di implementare le chiamate a funzione, la chiamata si effettua tramite l'istruzione **CALL** il ritorno tramite la funzione **RET**.

CALL <i>nome_routine</i>	$\%rsp = \%rsp - 8$ $(\%rsp) = \%rip$ $\%rip = \text{nome_routine}$
RET	$\%rip = \%rsp$ $\%rsp = \%rsp + 8$

Quindi ad ogni chiamata a funzione, la CALL prima di saltare salva l'indirizzo di ritorno sulla cima dello stack, per poi recuperarlo alla fine tramite RET. È quindi importante che alla fine del programma tutte le alterazioni a %RSP vengano eliminate

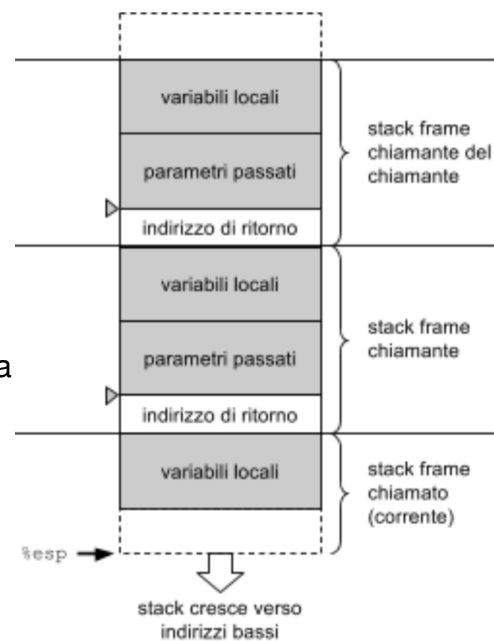


Passaggio parametri funzioni

Definito dalle “**calling conventions**”, si salvano sulla cima dello stack i parametri (in eccesso ai registri) da passare alla funzione prima di effettuare la chiamata.

1. Sottraggo a %RSP un numero di byte equivalente a quelli necessari per salvare tutti i parametri
2. Tramite MOV e %RSP spostato i parametri sulla cima dello stack (posso unire 1 e 2 usando push)
3. Eseguo istruzione CALL
4. Ripristino valore di %RSP

ATTENZIONE i parametri devono essere passati in modo che il primo sia quello con l'indirizzo più basso nella stack e gli altri crescano di conseguenza



C	Assembly (mov)	Assembly (push)	NOTE
F(10, 20)	$\text{subq } \$16, \%rsp$ $\text{movq } \$10, (\%rsp)$ $\text{movq } \$20, 8(\%rsp)$ call F $\text{addq } \$16, \%rsp$	$\text{pushq } 20$ $\text{pushq } 10$ call F $\text{addq } \$16, \%rsp$	bisogna comunque ripristinare valore originale di rsp in entrambi i casi

ATTENZIONE normalmente le variabili a 32 bit (o minori) dovrebbero essere promosse a grandezza massima prima del CALL (in questo caso non necessario), %RSP non cambia e le operazioni collegate ad esso devono essere effettuate comunque a 64 bit

Acquisizione parametri funzioni

L'acquisizione avviene in modo analogo andando a prelevare i parametri direttamente dalla stack tramite l'offset di %RSP

C	Assembly
Int F(long int x, long int y) return x+y;	$\text{f: movq } 8(\%rsp), \%rax$ $\text{addl } 16(\%rsp), \%rax$ ret

ATTENZIONE presenza indirizzo di ritorno e di eventuali altre modifiche al registro %RSP

Conversioni di tipo (cast)

1. Passaggio da tipo più grande a più piccolo
basta prendere solo la parte del registro necessaria

C	Assembly
<code>int a; char b; b = (char) a</code>	<code>movb %al, %bl</code> #basta usare prefisso e nome registro più piccoli

2. Passaggio da tipo più piccolo a più grande
in questo caso bisogna riempire lo spazio vuoto che può essere ancora falsato dal valore precedente, per far questo si usano le istruzioni:

MOVZ _{xx} S, D	D = ZeroExtend(S)	Copia S in D riempiendo con tutti 0 i bit in più, non considera il segno
MOVS _{xx} S, D	D = SignExtend(S)	Copia S in D e copia il bit più significativo in tutti i bit in più

xx sono i prefissi, prima quello di S poi quello di D

C	Assembly
<code>int a; char b; a = (int) b</code>	<code>Movbl %bl, %eax</code>

Coindizioni booleane

Una volta effettuato un test con CMP si può salvare un valore booleano su un registro con

SET _{cc} D	D = condizione	Se la condizione è verificata scrive 1 in D
----------------------------	----------------	---------------------------------------------

C	Assembly
<code>a = b < 0</code>	<code>cmpl \$0, %rbx setl %eax</code>

ATTENZIONE in questo caso non si ha salto, quindi non si inverte condizione

Maschere di bit

Si usano per operare su un solo bit, si implementano combinando l'istruzione corrispondente con un numero pari al bit o ai bit che si vogliono cambiare




1. Bit setting (bit impostato a 1) => istruzione OR
2. Bit clearing (bit impostato a 1) => istruzione AND
3. Inversione di un bit => istruzione XOR
4. Bit testing (estrae valore bit) => istruzione TEST

Appendice A (shift-and-add)

```
1      .org 0x800
2      .data
3          op1: .long 3
4          op2: .long 2
5
6      .text
7          movl op1, %ebx      # carico il moltiplicando
8          movl op2, %edx      # carico il moltiplicatore
9          xorl %eax, %eax     # %eax conterra' il risultato
10
11         cmpl $0, %edx
12         jz .fine            # Non devo moltiplicare per zero!
13
14     .moltiplica:
15         cmpl $0, %ebx
16         jz .fine            # Quando il moltiplicando e' zero ho finito
17         shrl $1, %ebx
18         jnc .nosomma        # Se C = 0 non sommo
19         addl %edx, %eax
20
21     .nosomma:
22         shll $1, %edx
23         jc .overflow        # Controllo l'overflow
24         jmp .moltiplica
25
26     .overflow:
27         movl $-1, %eax      # In caso di overflow, imposto il risultato a -1
28
29     .fine:
30         hlt
```

Il registro FLAGS



-  Riservato (da non modificare)
-  Control Flags
-  Status Flags

- **carry** (CF): vale 1 se l'ultima operazione ha prodotto un riporto
- **parity** (PF): vale 1 se nel risultato dell'ultima operazione c'è un numero pari di 1
- **zero** (ZF): vale 1 se l'ultima operazione ha come risultato 0
- **sign** (SF): vale 1 se l'ultima operazione ha prodotto un risultato negativo
- **overflow** (OF): vale 1 se il risultato dell'ultima operazione supera la capacità di rappresentazione
- **interrupt enable** (IF): indica se c'è la possibilità di interrompere l'esecuzione del programma in corso
- **direction** (DF): modifica il comportamento delle operazioni su stringhe