Sintassi Assembly AT&T

Istruzioni

La sintassi classica di un'istruzione è la seguente:

```
istruzione operando1, operando2
```

Il numero di operandi dipende dal tipo di istruzione.

Struttura di un programma Assembly

I programmi Assembly sono solitamente composti da almeno tre sezioni: *text*, *data* e *bss*. Ognuna di queste sezioni può essere eventualmente vuota. Altre sezioni possono essere create mediante la direttiva `.section`.
La sezione `.data` viene utilizzata per dichiarare variabili globali inizializzate e costanti.
La sezione `.text` contiene il codice Assembly vero e proprio. Questa sezione deve iniziare con la dichiarazione `.global _start` che fornisce al sistema operativo la locazione di memoria in cui si trova la prima istruzione del programma (è simile alla funzione main di Java o di C).
La sezione `.bss` consente di riservare spazio in memoria quando il programma verrà caricato in memoria per essere eseguito; può servire per contenere variabili non inizializzate.

Esempio: Hello, World!

```
# Nome file
# ----------
# hello.s
#
# Istruzioni per la compilazione
# ------------------------------
# as -o hello.o hello.s
# ld -o hello hello.o
#
# Funzionalita'
# -------------
# Stampa a video la scritta "Hello, World!" e va a capo
#
# Commenti
# --------
# Il carattere # indica l'inizio di un commento.
#
```

```
.section .data                    #sezione definizione variabili

hello:                            #etichetta
     .ascii "Hello, World!\n"     #stringa costante

hello_len:
     .long . - hello              #lunghezza della stringa in byte


.section .text                    #sezione istruzioni
     .global _start               #punto di inizio del programma

_start:
     movl  $4, %eax               #Carica il codice della system
                                  #call WRITE.

     movl  $1, %ebx               #Mette a 1 il contenuto di EBX.
                                  #Il codice 1 corrisponde al file
                                  #descriptor dello standard
                                  #output (terminale).

     Leal hello, %ecx             #Carica in ECX l'indirizzo di
                                  #memoria associato all'etichetta
                                  #hello.

     movl hello_len, %edx         #Carica in EDX la lunghezza
                                  #della stringa contenuta in
                                  #hello.

     int $0x80                    #Esegue la system call in EAX
                                  #tramite l'interrupt 0x80.

     Movl $1, %eax                #Carica il codice della system
                                  #call EXIT.

     xorl  %ebx, %ebx             #Azzera EBX. Contiene il codice
                                  #di ritorno della system call.

     int $0x80                    #Esegue la system call in EAX
                                  #tramite l'interrupt 0x80.
```

NOTA: è necessario che tutte le variabili siano inizializzate durante la dichiarazione nella sezione `.data`.

Assemblare, verificare ed eseguire un programma Assembly

Il processo di creazione di un programma Assembly passa attraverso le seguenti fasi:

1. Scrittura di uno o più file ASCII (estensione *.s*) contenenti il programma *sorgente*, tramite un normale *editor di testo*.

2. Assemblaggio dei file sorgenti, e generazione dei file *oggetto* (estensione *.o*), tramite un *assemblatore*.
3. Creazione, del file *eseguibile*, tramite un *linker*.
4. Verifica del funzionamento e correzione degli eventuali errori, tramite un *debugger*.

L'assemblatore

L'Assemblatore trasforma i file contenenti il programma sorgente in altrettanti file oggetto contenenti il codice in linguaggio macchina. Durante il corso verrà utilizzato l'assemblatore ***gas*** della GNU.
Per assemblare un file è necessario eseguire il seguente comando:

```
as –o miofile.o miofile.s
```

Si consulti la documentazione (`man as`) per l'elenco delle opzioni disponibili.

Il linker

Il linker combina i moduli oggetto e produce un unico file eseguibile. In particolare: unisce i moduli oggetto, risolvendo i riferimenti a simboli esterni; ricerca i file di libreria contenenti le procedure esterne utilizzate dai vari moduli e produce un file eseguibile. Notare che l'operazione di linking deve essere effettuata anche se il programma è composto da un solo modulo oggetto.
Durante il corso verrà utilizzato il linker ***ld*** della GNU.
Per creare l'eseguibile a partire da un file oggetto è necessario eseguire il seguente comando:

```
ld –o miofile.x miofile1.o miofile2.o miofile3.o
```

Si consulti la documentazione (`man ld`) per l'elenco delle opzioni disponibili.

Assembly 32bit su macchine 64bit

La gran parte del codice ASM32 è compatibile con macchine a 64bit, tuttavia alcune estensioni in particolari istruzioni non sono riconosciute dai compilatori. È possibile utilizzare codice ASM32 su architetture a 64bit utilizzando dei flag di compilazione che permettono di simulare il comportamento di una architettura a 32bit.

```
as --32 –o miofile.o miofile.s
ld -m elf_i386 –o miofile.x miofile.o
```

Stampa di numeri

I numeri sono memorizzati nei registri e nelle variabili come interi in complemento a 2 su 32 bit. Affinché essi possano essere stampati a video occorre trasformarli in stringhe di caratteri cioè vettori di byte dove ciascun byte rappresenta un carattere secondo la cosiddetta codifica ASCII.
Per trasformare un numero intero in una stringa occorre scomporlo nelle sue cifre mediante divisioni successive per 10. Per la particolare conformazione della tabella ASCII il codice del carattere corrispondente alla cifra n si ottiene come n+48.

Tabella dei caratteri ASCII

La tabella seguente è relativa al codice US ASCII, ANSI X3.4-1986 (ISO *646 International Reference Version*). I codici decimali da 0 a 31 e il 127 sono caratteri non stampabili (*codici di controllo*). Il 32 corrisponde al carattere di "spazio". I codici dal 32 al 126 sono caratteri stampabili.

Char	Dec	Nome	Descrizione
	0	NUL (Ctrl-@)	NULL
	1	SOH (Ctrl-A)	START OF HEADING
	2	STX (Ctrl-B)	START OF TEXT
	3	ETX (Ctrl-C)	END OF TEXT
	4	EOT (Ctrl-D)	END OF TRANSMISSION
	5	ENQ (Ctrl-E)	ENQUIRY
	6	ACK (Ctrl-F)	ACKNOWLEDGE
	7	BEL (Ctrl-G)	BELL (Beep)
	8	BS (Ctrl-H)	BACKSPACE
	9	HT (Ctrl-I)	HORIZONTAL TAB
	10	LF (Ctrl-J)	LINE FEED
	11	VT (Ctrl-K)	VERTICAL TAB
	12	FF (Ctrl-L)	FORM FEED
	13	CR (Ctrl-M)	CARRIAGE RETURN
	14	SO (Ctrl-N)	SHIFT OUT
	15	SI (Ctrl-O)	SHIFT IN
	16	DLE (Ctrl-P)	DATA LINK ESCAPE
	17	DC1 (Ctrl-Q)	DEVICE CONTROL 1 (XON)
	18	DC2 (Ctrl-R)	DEVICE CONTROL 2
	19	DC3 (Ctrl-S)	DEVICE CONTROL 3 (XOFF)
	20	DC4 (Ctrl-T)	DEVICE CONTROL 4
	21	NAK (Ctrl-U)	NEGATIVE ACKNOWLEDGE
	22	SYN (Ctrl-V)	SYNCHRONOUS IDLE
	23	ETB (Ctrl-W)	END OF TRANSMISSION BLOCK
	24	CAN (Ctrl-X)	CANCEL

	25	EM (Ctrl-Y)	END OF MEDIUM
	26	SUB (Ctrl-Z)	SUBSTITUTE
	27	ESC (Ctrl-[)	ESCAPE
	28	FS (Ctrl-\)	FILE SEPARATOR
	29	GS (Ctrl-])	GROUP SEPARATOR
	30	RS (Ctrl-^)	RECORD SEPARATOR
	31	US (Ctrl-_)	UNIT SEPARATOR
	32		SPACE
!	33		EXCLAMATION MARK
"	34		QUOTATION MARK
#	35		NUMBER SIGN
$	36		DOLLAR SIGN
%	37		PERCENT SIGN
&	38		AMPERSAND
'	39		APOSTROPHE
(	40		LEFT PARENTHESIS
)	41		RIGHT PARENTHESIS
*	42		ASTERISK
+	43		PLUS SIGN
,	44		COMMA
-	45		HYPHEN, MINUS SIGN
.	46		PERIOD, FULL STOP
/	47		SOLIDUS, SLASH
0	48		DIGIT ZERO
1	49		DIGIT ONE
2	50		DIGIT TWO
3	51		DIGIT THREE
4	52		DIGIT FOUR
5	53		DIGIT FIVE
6	54		DIGIT SIX
7	55		DIGIT SEVEN
8	56		DIGIT EIGHT
9	57		DIGIT NINE
:	58		COLON
;	59		SEMICOLON
<	60		LESS-THAN SIGN, LEFT ANGLE BRACKET
=	61		EQUALS SIGN
>	62		GREATER-THAN SIGN, RIGHT ANGLE BRACKET
?	63		QUESTION MARK
@	64		COMMERCIAL AT SIGN
A	65		CAPITAL LETTER A

B	66		CAPITAL LETTER B
C	67		CAPITAL LETTER C
D	68		CAPITAL LETTER D
E	69		CAPITAL LETTER E
F	70		CAPITAL LETTER F
G	71		CAPITAL LETTER G
H	72		CAPITAL LETTER H
I	73		CAPITAL LETTER I
J	74		CAPITAL LETTER J
K	75		CAPITAL LETTER K
L	76		CAPITAL LETTER L
M	77		CAPITAL LETTER M
N	78		CAPITAL LETTER N
O	79		CAPITAL LETTER O
P	80		CAPITAL LETTER P
Q	81		CAPITAL LETTER Q
R	82		CAPITAL LETTER R
S	83		CAPITAL LETTER S
T	84		CAPITAL LETTER T
U	85		CAPITAL LETTER U
V	86		CAPITAL LETTER V
W	87		CAPITAL LETTER W
X	88		CAPITAL LETTER X
Y	89		CAPITAL LETTER Y
Z	90		CAPITAL LETTER Z
[	91		LEFT SQUARE BRACKET
\	92		REVERSE SOLIDUS (BACKSLASH)
]	93		RIGHT SQUARE BRACKET
^	94		CIRCUMFLEX ACCENT
_	95		LOW LINE, UNDERLINE
`	96		GRAVE ACCENT
a	97		SMALL LETTER a
b	98		SMALL LETTER b
c	99		SMALL LETTER c
d	100		SMALL LETTER d
e	101		SMALL LETTER e
f	102		SMALL LETTER f
g	103		SMALL LETTER g
h	104		SMALL LETTER h
i	105		SMALL LETTER i
j	106		SMALL LETTER j

k	107		SMALL LETTER k
l	108		SMALL LETTER l
m	109		SMALL LETTER m
n	110		SMALL LETTER n
o	111		SMALL LETTER o
p	112		SMALL LETTER p
q	113		SMALL LETTER q
r	114		SMALL LETTER r
s	115		SMALL LETTER s
t	116		SMALL LETTER t
u	117		SMALL LETTER u
v	118		SMALL LETTER v
w	119		SMALL LETTER w
x	120		SMALL LETTER x
y	121		SMALL LETTER y
z	122		SMALL LETTER z
{	123		LEFT CURLY BRACKET, LEFT BRACE
|	124		VERTICAL LINE, VERTICAL BAR
}	125		RIGHT CURLY BRACKET, RIGHT BRACE
~	126		TILDE
	127		DELETE

Istruzioni ed etichette di salto

In Assembly non esiste il costrutto IF ... THEN ... ELSE e quindi le istruzioni di salto servono per far saltare l'esecuzione del programma ad una certa istruzione in funzione del valore di una condizione. Le uniche condizioni che si possono valutare sono <,=,> tra due valori numerici e la presenza di zero nel registro ECX. In particolare, la valutazione di una condizione di <,=,> consiste di due istruzioni: la prima sottrae tra loro i due valori numerici e imposta i bit SF e ZF del registro EFLAGS, la seconda effettua il salto in base al valore di tali flags.
Le etichette sono essenziali per le istruzioni di salto in quanto indicano a quale punto della sequenza di istruzioni bisogna saltare. Occorre inserire prima dell'istruzione a cui si vuole saltare un nome simbolico seguito dal carattere ":".

Esempio:

```
salta_qui:
     addl ...
```

È importante che il nome dell'etichetta sia unico in tutto il programma. Anche in questo caso, come per i nomi delle variabili, l'assemblatore trasforma i nomi delle etichette in

numeri binari (che in questo caso indicano l'indirizzo dell'istruzione che segue) a meno che non si voglia conservarli per il debug (con l'opzione `--gstabs`).
In Assembly non esistono istruzioni ad alto livello per realizzare i cicli come FOR ..., WHILE ...; essi si devono costruire manualmente a partire dalle istruzioni di salto condizionato. Se si vuole eseguire un ciclo per un certo numero di volte occorre utilizzare ECX come contatore.

Sintassi	Descrizione
`cmp op1, op2`	**Compare** – Confronta `op2` con `op1`; le varie istruzioni di salto che compaiono dopo si dovranno leggere come `op2` maggiore/minore `op1`. Solo `op1` può essere un parametro immediato. Esegue la sottrazione `op2 – op1` senza memorizzare il risultato ed in base ad esso imposta i flag ZF e SF del registro EFLAGS. Le istruzioni di salto condizionato successive opereranno in base ai flag impostati da `cmp`.
`jg lbl`	**Jump if greater** – Salta all'istruzione associata all'etichetta `lbl` se l'istruzione `cmp op1, op2` ha verificato che `op2` è maggiore di `op1`.
`jl lbl`	**Jump if less** – Salta all'istruzione associata all'etichetta `lbl` se l'istruzione `cmp op1, op2` ha verificato che `op2` è minore di `op1`.
`jge lbl`	**Jump if greater or equal** – Salta all'istruzione associata all'etichetta `lbl` se l'istruzione `cmp op1, op2` ha verificato che `op2` è maggiore o uguale di `op1`.
`jle lbl`	**Jump if less or equal** – Salta all'istruzione associata all'etichetta `lbl` se l'istruzione `cmp op1, op2` ha verificato che `op2` è minore o uguale di `op1`.
`je lbl`	**Jump if equal** – Salta all'istruzione associata all'etichetta `lbl` se l'istruzione `cmp op1, op2` ha verificato che `op2` è uguale a `op1`.
`jne lbl`	**Jump if not equal** – Salta all'istruzione associata all'etichetta `lbl` se l'istruzione `cmp op1, op2` ha verificato che `op2` è diverso da `op1`.
`jcxz lbl`	**Jump if CX is zero** – Salta all'istruzione associata all'etichetta `lbl` se il registro CX contiene il valore 0.
`jmp lbl`	**Jump** – Salta incondizionatamente all'istruzione associata all'etichetta `lbl`.
`test op1, op2`	**Test** – Esegue l'AND bit a bit tra gli operandi, imposta lo Zero Flag ma, a differenza dell'istruzione AND, non memorizza il risultato in modo da poter fare più test in sequenza. L'istruzione `test %ax, %ax` imposta lo Zero Flag a 1 se e solo se AX è zero.

`loop lbl`	**Loop** – Quando il processore incontra questa istruzione, per prima cosa decrementa ECX di 1 e successivamente controlla se ECX è zero; se ECX non è zero allora salta all'istruzione indicata dall'etichetta `lbl`. Solo ECX si può utilizzare per questa e di conseguenza non è possibile innestare cicli uno dentro l'altro.

NOTA: `jg`, `jge`, `jl`, e `jle` funzionano con operandi con segno; per operandi senza segno usare `ja`, `jae`, `jb`, e `jbe` rispettivamente (`ja` significa "jump if above", `jb` significa "jump if below").

Esempio: statement IF … ELSE

Java	Assembly

`if (x > y)`
`  {`
`  // block 1`
`  }`
`else`
`  {`
`  // block 2`
`  }`
`// block 3`
	`movl x, %eax`
	`movl y, %ebx`
	`cmp %eax, %ebx`
	`jl isless  # ebx<eax`
	`  # block 2 (blocco else)`
	`jmp after`
	`isless:`
	`  # block 1 (blocco then)`
	`after:`
	`  # block 3 (fuori da if)`

Esercizi

Esercizio 1

Scrivere un programma Assembly che sommi i numeri 100, 33 e 68 e metta il risultato in una variabile chiamata "somma". Stampare a monitor il risultato.

Esercizio 2

Scrivere un programma Assembly che confronti 2 numeri caricati nei registri EAX ed EBX e stampi una stringa che indichi quale sia il maggiore ed il minore oppure indichi che i due numeri sono uguali.