ASSEMBLY CON TASM/MASM 2

Assemblare, linkare, compilare 3
Struttura dei programmi 4
Ciclo di vita di un programma 5
Ambiente TASM 7
Strutture di controllo 9
Area dati e area codice 13
Programmi di tipo COM 16
Modelli di memoria 18
Programma COM con sintassi TASM 19

Assembly avanzato 22
Macro 22
Istruzioni aritmetiche 26
Stack 27

Procedure 31
Meccanismo di chiamata 33
Passaggio di parametri 35
Variabili locali 39

Direttive per programmi e librerie 43
Direttive JUMPS e LOCALS 43
Librerie: direttive INCLUDE, PUBLIC ed EXTRN 45
Makefile 48

Questo file costituisce un’estensione online del corso di Paolo Ollari – *Corso di sistemi e reti*

Assembly con TASM/MASM

Per affrontare adeguatamente i temi contenuti in questa sezione bisogna avere una sufficiente competenza su alcuni argomenti prerequisiti.
In particolare sono dati per conosciuti gli argomenti trattati nel testo Sistemi e reti, Volume 1, capitoli A1, A2 e A3.
Per verificare se la conoscenza necessaria degli argomenti è sufficiente, svolgere questa breve attività di autovalutazione. Se non si è in grado di rispondere a una domanda o si è solo parzialmente certi della risposta data, è meglio considerare la risposta come errata ai fini dell'autovalutazione.

1. I registri di uso generale dell'Intel 8086 sono e sono ampi **a)** AX, BX, CX, DX tutti a 8 bit; **b)** AH, BH, CH, DH tutti a 8 bit; **c)** AX, BX, CX, DX tutti a 16 bit; **d)** CS, DS, ES, SS tutti a 16 bit
2. L'indirizzamento nell'Intel 8086 è di tipo **a)** lineare; **b)** flat; **c)** segmentato; **d)** indiretto
3. Il program counter e il PSW nell'Intel 8086 sono realizzati dai registri **a)** IP, SS; **b)** CS:IP, DI; **c)** CS:IP, Flags; **d)** CS, ES
4. Quale istruzione sposta 3 in AL? **a)** mov bl,3; **b)** mov ax,3; **c)** mov 3,al; **d)** mov ah,3
5. Quale frase è corretta per un sistema operativo multiprogrammato? **a)** un processo genera un programma; **b)** un programma genera un solo processo; **c)** un programma genera più processi; **d)** un processo genera un solo programma
6. Per stampare un carattere sullo schermo: **a)** mov ah,2; mov dl,30; int 21; **b)** mov ah,2; mov dl,30h; int 21h; **c)** mov ah,1; int 21; **d)** int 20
7. Per terminare un programma COM: **a)** int 21; **b)** int 10; **c)** int 19; **d)** int 20h
8. Per richiedere l'input di un carattere da tastiera: **a)** mov ah,2; int 21; **b)** mov ah,0; int 21h **c)** mov ah,1; int 21h **d)** int 21h
9. Per allocare un byte in memoria in area dati: **a)** dw 0; **b)** dc 0; **c)** db 0; **d)** dw 1
10. Per leggere un byte dall'area dati: **a)** mov dl,[ax]; **b)** mov dl,[bx]; **c)** mov [bx],3; **d)** mov [ax],dl;
11. Indicare quale istruzione assembly è errata: **a)** mov ax,bx; **b)** mov ax, bl; **c)** mov al, bl; **d)** mov si,dx
12. Il PSP si trova **a)** in memoria, prima del processo; **b)** su disco, nel programma; **c)** in memoria, prima del programma; **d)** nel file eseguibile

Conoscenza necessaria: 10 risposte corrette.
Risposte corrette: cccbcbdccbba

Assemblare linkare, compilare

Naturalmente la programmazione dell'assembler x-86 con **Debug.exe** ha senso solo per ragioni didattiche.
Nella realtà, aldilà di brevi frammenti di codice di test, Debug non viene utilizzato per scrivere programmi, e i programmatori si affidano a veri e propri ambienti di sviluppo su linea di comando, i più noti dei quali sono **TASM**(NB) (*Borland*) e **MASM** (*Microsoft*). Un ambiente freeware molto apprezzato è anche **NASM**; in ogni caso tutti prodotti ormai facilmente reperibili senza alcun costo né necessità di licenza.

(NB)
La scelta di TASM deriva dal fatto che la sintassi MASM è correttamente interpretata da TASM, mentre non è vero il viceversa.

Come si è verificato utilizzando *Debug*, la principale difficoltà nello scrivere programmi in assembly è la gestione degli indirizzi e il loro ricalcolo ad ogni modifica del programma: questo non succede con gli ambienti ad assemblatore. Inoltre essi offrono la possibilità di utilizzare decine di pseudoistruzioni e direttive che rendono la programmazione assembly molto efficace.
La gestione semplificata degli indirizzi viene ottenuta da questi ambienti tramite l'uso delle **etichette** (*label*) al posto degli indirizzi numerici, e del concetto di **doppia passata** dell'**assemblatore**, che è il programma che analizza il file sorgente contenente il codice assembly e le pseudoistruzioni.
Con la doppia passata l'assemblatore traduce correttamente tutte le etichette nei corrispondendi indirizzi numerici senza che il programmatore debba più preoccuparsene, cosicchè nei files sorgenti scritti per questi ambienti, l'uso degli indirizzi numerici è praticamente abolito.
L'etichetta, inoltre, rende il codice molto più comprensibile, dato che gli identificatopri di etichetta sono scelti dal programmatore e possono descrivere, tramite il loro nome, la funzione svolta dall'indirizzo simbolico che rappresentano.

Win64

Con i sistemi operativi Microsoft a 64bit (**win64**, come XP64, Vista64 e Seven64), non è più possibile usare Command.com, e la shell di MSDOS non consente più di lanciare programmi x-86 a 16bit come, ad esempio, tasme.exe o masm.exe.

Per ovviare a questa situazione si può installare, su questi sistemi, un programma emulatore gratuito di MSDOS x-86 denominato **DOSBox**, oppure il programma emulatore gratuito di processore x-86 denominato **EMU8086** (*vedi* *Appendice*).

Struttura dei programmi

Come si è visto nella programmazione con *Debug*, i programmi contengono, generalmente, almeno due aree distinte di istruzioni: l'**area del codice** e l'**area dati**. Questo vale per qualsiasi programma scritto in qualsiasi linguaggio. Oltre a queste due aree, i programmi ne contengono altre, altrettanto importanti, tra cui: l'area di **startup**, l'area dello **stack** e l'area dello **heap**.

Area di startup

E' la zona iniziale di ogni programma (normalmente posta proprio nei primi byte del file eseguibile) ed è generata da un programma speciale denominato **linker** (*correlatore*).
I primi byte di un programma costituiscono la zona del formato (o **header**), area che contiene informazioni tipiche per il Sistema Operativo (che ospita il programma) in modo che il Sistema Operativo possa caricare correttamente il programma in memoria e farlo diventare processo. Una volta in memoria, il Sistema Operativo avvia le prime istruzioni dell'area di startup che in pratica consentono di eseguire la prima istruzione scritta dal programmatore (*entry point*) in modo corretto, eventualmente fornendo i dati di avvio ereditati dal Sistema Operativo (come ad esempio i parametri su riga di comando). Solo i files eseguibili .COM non possiedono un'area di Startup.

Area di Codice

E' la zona del programma che contiene le istruzioni da eseguire durante l'esecuzione, cioè durante il **runtime** del programma.
L'area di codice è scritta espressamente dal programmatore tramite le regole della sintassi.di un linguaggio di programmazione, nel nostro caso le regole della programmazione assembly x-86. Essa si trova immediatamente dopo l'area di startup, dalla quale eredita il controllo all'avvio dell'esecuzione del programma.

Area Dati

E' la zona in cui il programmatore alloca i dati tramite istruzioni presenti nell'area codice. Qui trovano posto le **variabili globali statiche** dei programmi, cioè quelle locazioni di memoria disponibili durante tutto il runtime. Quest'area è sia di lettura che di scrittura, a differenza dell'area di codice e di startup che sono esclusivamente aree di lettura.

Area dello heap

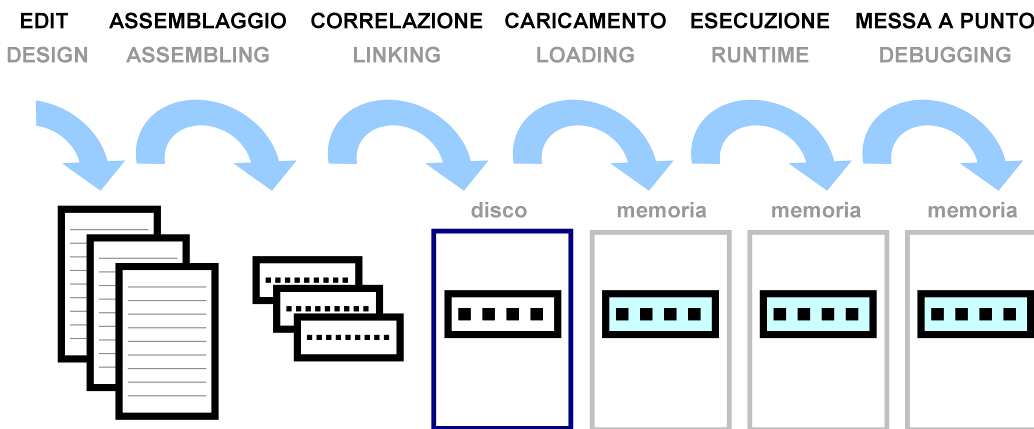
E' una zona opzionale, in cui durante il runtime, il programmatore, tramite istruzioni ben precise, alloca temporaneamente un po' di memoria per far posto a variabili la cui dimensione è accertabile solo durante l'esecuzione (p. es. la dimensione di una stringa in input). E' anche detta **memoria dinamica**, dato che la sua dimensione non è prefissata e può essere anche allocata e deallocata più volte durante il runtime. La gestione dell'area di heap viene ottenuta tramite istruzioni in area codice, che richiedono i servizi di allocazione/deallocazione al Sistema Operativo.

Area dello stack

E’ una zona di memoria gestita automaticamente dai compilatori su tipiche tecniche di programmazione scritte in area di codice dal programmatore, come la dichiarazione di variabili locali e il passaggio di parametri alle procedure: a fronte di questi costrutti, i compilatori gestiscono l’area di stack in modo trasparente al programmatore, allocando e deallocando le variabili locali e i parametri passati alle procedure. Solo nella programmazione in Assembly è possibile gestire direttamente l’area di stack con opportune istruzioni.

Ciclo di vita di un programma

La creazione, lo sviluppo, l’esecuzione e la messa a punto di un programma, segue un certo numero di fasi caratterizzate da attività specifiche, tempi specifici, applicazioni di supporto specifiche, errori specifici e file specifici.
E’ possibile sintetizzare il ciclo di vita di un programma tramite un diagramma e una tabella che riportano fasi, tempi, applicazioni, files ed errori relativi ad ogni fase.



Design Time	Compile Time		Load Time	RunTime	Debug Time
Editor	Assemblatore	Linker	Sistema Operativo		Debugger
Sorgenti	*Oggetto*	*Eseguibile*	*Processo*		

MAIN.ASM	MAIN.OBJ				
LIB1.ASM	LIB1.OBJ	MAIN.EXE			
LIB2.ASM	LIB2.OBJ				

Fase di edit

Il programmatore scrive il **testo** del programma (moduli o files *sorgenti*) con la sintassi di un linguaggio di programmazione. Spesso i programmi sono costituiti da più sorgenti, ma solo uno contiene l’*entry point* del programma. I rimanenti testi sono denominati **librerie di codice**.
Il programma utilizzato per scrivere il testo di un programma è un **editor**, spesso integrato in un **IDE** (*Integrated Development Equipment*). L’attività del programmatore in questa fase è detta **design time**. Gli errori più

frequenti a *design time* riguardano il formato dei files (es. i files devono essere rigorosamente files di testo), la loro irreperibilità o la loro corruzione.

Fase di compilazione

Una volta completato un modulo sorgente, esso deve essere **assemblato**, ovvero le istruzioni e le pseudoistruzioni presenti nei sorgenti in linguaggio simbolico ad alto livello, devono essere trasformate in assembly, a basso livello. Ogni file sorgente quindi viene ridotto, da un programma di supporto denominato **assemblatore**, a un file binario corrispondente (detto anche *file oggetto*).

Dopo l'assemblaggio, è necessaria la **correlazione** (*linking*), ad opera di un secondo programma a supporto, denominato **linker**.

Il linker collega tutti i files oggetto in uno solo, e genera il file eseguibile (*target* della compilazione), aggiungendovi, nella sua parte iniziale, la porzione di startup e l'eventuale **header** (cfr. Formato degli eseguibili e Rilocazione). Le due fasi di assemblaggio e linking sono spesso riunite in un unico passo, denominato *compile time*.

Durante il compile time si possono verificare i tipici errori di sintassi (del linguggio scelto) che sono sempre segnalati dai programmi compilatori, sottoforma di errori e/o warning.

Fase di caricamento

Una volta su disco, il **programma** eseguibile deve essere caricato in memoria dal Sistema Operativo per essere poi trasformato in **processo**.

Il Sistema Operativo legge lo header del file eseguibile e carica in memoria il programma, dopodichè cede il controllo al codice di startup dell'eseguibile. Spesso i Sistemi Operativi creano una zona di memoria di collegamento con il programma (cfr. PSP in Formato degli eseguibili e Rilocazione) prima di cedere il controllo.

Tipici errori della fase di load time sono errati formati degli header, o l'impossibilità del caricamento per scarsità di memoria.

Fase di esecuzione

Il **runtime** è il tempo durante il quale il processo opera in memoria e in CPU, dalla prima istruzione di codice all'ultima prevista dal programmatore. Tipici errori di runtime sono le divisioni per zero, i loop infiniti, le terminazioni anomale per mancanza o incongruenza delle risorse richieste dal programma, ecc…

Fase di messa a punto

Il runtime può anche essere avviato tramite un programma speciale, denominato **debugger** (attività di debugging).

In questo caso il debugger carica ed esegue il programma nelle modalità impostate dal programmatore, ad esempio **passo passo** (per verificare il flusso dell'esecuzione) o tramite **breakpoint**, ovvero sospensioni dell'esecuzione su istruzioni critiche, per eplorare lo stato di registri, variabili e memoria (**watch**). Tutto ciò al fine di individuare le cause di eventuali malfunzionamenti riscontrati al runtime.

Ambiente TASM

Così come descritto, l'ambiente di sviluppo TASM prevede le fasi tipiche del ciclo di vita di un programma, fornendo i relativi programmi di supporto.
Bisogna tener presente che, malgrado si sia scelto di sviluppare direttamente in Assembly, e quindi in apparenza potrebbe sembrare inutile una fase di compilazione, l'uso delle fondamentali *due passate* per risolvere il problema degli indirizzi numerici, rende necessaria la presenza di un programma *assemblatore* e un programma *linker*.
I sorgenti per TASM possono essere scritti con un qualsiasi editor di testo (p. es. *Notepad* di Windows), avendo cura di usare sempre l'estensione **.asm** per ogni modulo sorgente; il programma assemblatore si chiama **tasm.exe** e il programma linker si chiama **tlink.exe**. Il programma debugger, infine, si chiama *Turbo Debugger* (**td.exe**).
In particolare, per usare TASM, sono sufficienti i seguenti files, reperibili anche all'interno delle cartelle dei compilatori Borland C 3.1 e/o Turbo Pascal 7.0: **tasm.exe**, **tlink.exe**, **td.exe**, **dpmiload.exe**, **dpmimem.dll**

I sorgenti TASM/MASM sono ricchi di *parole chiave* (**pseudoistruzioni** o **direttive**) che, oltre a semplificare l'attività di scrittura del codice Assembly aggiungono funzionalità supplemetari ai programmi.

Programma. Il secondo programma più corto del mondo

Si prenda in considerazione questo breve sorgente assembly per TASM/MASM che si limita a terminarsi correttamente (in realtà il programma non fa nulla, ma è corretto).
NB. *In **verde** le parole chiave previste dal linguaggio assembly TASM/MASM;*
*in **blu** le etichette o i simboli decisi dal programmatore; in nero il codice sorgente.*

Ricordare che TASM/MASM usa la notazione **decimale** di default, pertanto quando è necessario usare numeri espress in in base **esadecimale** bisogna utilizzare il suffisso **h** (o **H**).

corto.asm

```
00  SEG_UNICO segment      ; definizione dell'unico segmento del programma
01
02  assume CS:SEG_UNICO    ; assegnazione di tutti i registri di segmento all'unico segmento
03  assume DS:SEG_UNICO
04  assume ES:SEG_UNICO
05  assume SS:SEG_UNICO
06
07  START:                 ; AREA CODICE
08    mov ax, 4c00h        ; unica istruzione del programma: terminazione programma EXE
09    int 21h
10  ends
11  end START
```

L'ambiente TASM/MASM esige sapere l'esatta definizione dei registri di segmento. Nell'esempio i quattro registri di segmento CS, DS, ES e SS hanno lo stesso valore e condividono lo stesso segmento: il programma consiste solo di due istruzioni, non gestisce né dati, né stack.
Assegnato un nome di fantasia all'unico segmento (SEG_UNICO, riga 0) con la parola chiave **segment**, si informa l'ambiente che i registri di

segmento condividono lo stesso valore (righe 2-5) con la parola chiave **assume**.
L'etichetta START: (riga 7) indica l'inizio dell'area Codice (e deve essere 'chiusa' con la parola chiave **end** START a riga 11) per indicare la fine del file sorgente, mentre la parola chiave **ends** (riga 10) indica che la sezione dei segmenti è terminata.

Esempio. Ciclo di vita di un programma Assembly

Aldilà degli elementi specifici del sorgente, come le parole chiave e le varie etichette, il ciclo di creazione dell'eseguibile **corto.exe** a partire dal sorgente **corto.asm** si sviluppa in questo modo:

```
C:\tasm>tasm corto.asm                                           assemblaggio
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland International

Assembling file:   corto.asm
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  424k
C:\tasm>
C:\tasm>tlink corto.obj                                          linking (correlazione)
Turbo Link  Version 5.1 Copyright (c) 1992 Borland International
Warning: No stack
C:\tasm>
C:\tasm>corto.exe                                                esecuzione
C:\tasm>
```

Il tutto può essere ottenuto anche senza usare estensioni per i nomi dei files:

```
C:\tasm>tasm corto
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland International

Assembling file:   corto.asm
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  424k
C:\tasm>
C:\tasm>tlink corto
Turbo Link  Version 5.1 Copyright (c) 1992 Borland International
Warning: No stack
C:\tasm>
C:\tasm>corto
C:\tasm>
```

Ecco invece una tipica videata del debugger **td.exe**, applicato al file **corto.com**:

```
C:\tasm>td corto.com                       <──────────────────── debugging

  ≡  File  Edit  View  Run  Breakpoints  Data  Options  Window  Help        READY
╔═[■]═CPU 80486══════════════════════════════════════════════════1═════[↕]═╗
║  cs:0000►B8004C        mov     ax,4C00                   ▲  ax 0000  │c=0║
║  cs:0003 CD21          int     21                        ■  bx 0000  │z=0║
║  cs:0005 0000          add     [bx+si],al                ▒  cx 0000  │s=0║
║  cs:0007 0000          add     [bx+si],al                ▒  dx 0000  │o=0║
║  cs:0009 0000          add     [bx+si],al                ▒  si 0000  │p=0║
║  cs:000B 0000          add     [bx+si],al                ▒  di 0000  │a=0║
║  cs:000D 0000          add     [bx+si],al                ▒  bp 0000  │i=1║
║  cs:000F 0000          add     [bx+si],al                ▒  sp 0000  │d=0║
║  cs:0011 0000          add     [bx+si],al                ▒  ds 5BC0  │   ║
║  cs:0013 0000          add     [bx+si],al                ▒  es 5BC0  │   ║
║  cs:0015 0000          add     [bx+si],al                ▒  ss 5BD0  │   ║
║  cs:0017 0000          add     [bx+si],al                ▒  cs 5BD0  │   ║
║  cs:0019 0000          add     [bx+si],al                ▒  ip 0000  │   ║
║  cs:001B 0000          add     [bx+si],al                ▒           │   ║
║  cs:001D 0000          add     [bx+si],al                ▼           │   ║
╟◄▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒►┼────────────────╢
║  ds:0000 CD 20 FF 9F 00 9A F0 FE =  ƒ Ü≡■          │  ss:0002 CD4C     ║
║  ds:0008 1D F0 E0 01 72 22 AA 01 ↔≡α☺r"¬☺          │  ss:0000►00B8     ║
║  ds:0010 72 22 89 02 CD 1C 3F 0E r"ë☻═∟?♫          │  ss:FFFE 0000     ║
║  ds:0018 01 01 01 00 02 FF FF FF ☺☺☺ ☻             │  ss:FFFC 5BD0     ║
║  ds:0020 FF FF FF FF FF FF FF FF                   │  ss:FFFA 0005     ║
╚═══════════════════════════════════════════════════════════════════════════╝
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
C:\tasm>
```

Se il file sorgente contiene errori di sintassi, l'assemblatore mostra un messaggio esplicito che indica il tipo di errore e la riga su cui è stato rilevato. E' ovvio che in presenza di errori, il file oggetto non viene creato, interrompendo la fase di creazione del file eseguibile.

Allo stesso modo, ma più raramente, il linker mostra un messaggio d'errore nel caso di incongruenza nella correlazione.

```
C:\tasm>tasm corto
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland International

Assembling file:   corto.asm
**Error** corto.asm(10) Constant too large      <────────── Errore assemblatore
Error messages:    1
Warning messages:  None
Passes:            1
Remaining memory:  424k
C:\tasm>
C:\tasm>tlink corto
Turbo Link  Version 5.1 Copyright (c) 1992 Borland International
Fatal: Unable to open file 'corto.obj'    <──────────────────────── Errore linker
C:\tasm>
```

In questo caso l'assemblatore TASM ha trovato un errore alla riga 10 del file sorgente **corto.asm** (**int 21ah**) e il linker segnala errore perché manca il file **corto.obj** da cui ricavare l'eseguibile.

Strutture di controllo

Le principali strutture di controllo utilizzate in Assembly x-86, oltre alla sequenza, sono la condizione (se-allora) e l'iterazione (ripeti n volte), rispettivamente implementate dalle istruzioni di salto condizionato

(istruzioni tipo J* [indirizzo]) e dall'istruzione loop (tipo LOOP* [indirizzo]).
Le istruzioni di salto condizionato, che spostano l'esecuzione all'indirizzo in esse specificato, sono numerose e vanno utilizzate, di norma, subito dopo l'istruzione di confronto CMP; tale istruzione, che equivale ad una sottrazione, imposta i flag del registro omonimo in base al risultato della sottrazione, consentendo alle istruzioni di salto condizionato di operare.
In base allo stato dei singoli flag del registro dei Flags, infatti, le varie istruzioni di salto condizionato effettuano il salto all'indirizzo specificato o meno.

Istruzione CMP

Sintassi: **`CMP operando1, operando2`**

Scopo: Viene eseguita la sottrazione **`operando1 - operando2`** e impostati i flags opportuni in base all'esito della sottrazione.

Esempi:
```
CMP AL,2
CMP AX, BX
MOV [BX], AL
```

Nota: *L'istruzione viene utilizzata in base alle regole generali della sintassi x-86 (cfr. Sintassi e indirizzamenti). Naturalmente* ***`operando1`*** *e* ***`operando2`*** *rimangono invariati dopo la CMP.*

Istruzione J*

Sintassi: **`J* indirizzo`**

Scopo: La notazione J* significa un intero gruppo di istruzioni analoghe (es. JE, JG, JLE, ecc.), tutte con la medesima sintassi. Il flusso dell'esecuzione si sposta su **`indirizzo`** se le condizioni sui flags previste dalla J* sono verificate, altrimenti il flusso dell'esecuzione prosegue regolarmente in sequenza. Questo gruppo di istruzioni sono dette **salti condizionati**, e si usano spesso dopo l'istruzione CMP per sfruttarne le modifiche di stato dei flag.

Esempi:
```
JE 109h
JG 110h
JB 112h
```

Nota: *Se l'esito del confronto CMP precedente ha impostato il flag di Zero, allora i due operandi di CMP sono uguali e la JE salta all'indirizzo 109h. Analogamente per gli altri due casi, se gli operandi sono, rispettivamente, il primo maggiore del secondo (numeri con segno), il primo minore del secondo (numeri senza segno).*
Ricordare che la distanza tra l'indirizzo dell'istruzione di salto e l'indirizzo presso il quale si vuole saltare non deve superare 128.

Infine vediamo l'istruzione di **salto incondizionato**, fondamentale per dirigere il flusso del codice in modo da saltare alcune parti del programma che per qualche motivo non devono essere eseguite. Spesso il salto incondizionato è usato per realizzare strutture di controllo diverse da quelle standard (es. cicli do-while):

Istruzione JMP

Sintassi: **JMP indirizzo**

Scopo: Sposta il flusso dell'esecuzione a **indirizzo**.

Esempi: **JMP 120h**
JMP CS:120h

Nota: *L'istruzione viene usata per spostare il flusso dell'esecuzione ad un indirizzo specifico in modo incondizionato. L'istruzione viene detta salto incondizionato e non ha i limiti di estensione del salto condizionato. Può infatti essere specificato un indirizzo completo seg:ofs.*

Una tabella di riferimento per consultare velocemente il comportamento delle istruzioni di salto condizionato, è la seguente:

Istruzioni di salto	Flags Z	C	S	O	P	Operatore equivalente	Descrizione
JE, JZ	1					=	Salta se uguali
JNE, JNZ	0					≠	Salta se diversi
JA, JNBE	0	0				>	Salta se maggiore, senza segno
JAE, JNB, JNC		0				>=	Salta se maggiore o uguale, senza segno
JB, JC, JNAE		1				<	Salta se minore, senza segno
JBE, JNA	1	1				<=	Salta se minore o uguale, senza segno
JG, JNLE	0		=	=		>	Salta se maggiore, con segno
JGE, JNL			=	=		>=	Salta se maggiore o uguale, con segno
JL, JNGE			≠	≠		<	Salta se minore, con segno
JLE, JNG	1		≠	≠		<=	Salta se minore o uguale, con segno
JNO				0			Salta se non c'è overflow
JNP, JPO					0		Salta se c'e' non c'e' parità (ovvero c'è parità dispari)
JNS			0				Salta se non c'e' segno
JO				1			Salta se c'è overflow
JP, JPE					1		Salta se c'e' parità (pari)
JS			1				Salta se c'e' segno

Programma. Input e output di un carattere: etichette e salti

Ecco come si presenta un codice TASM/MASM che attende un tasto, se il tasto è lo zero (0), viene stampato a schermo una zeta maiuscola, altrimenti una n minuscola:

cmpj.asm

```
00  SEG_UNICO segment      ; definizione dell'unico segmento del programma
01
02  assume CS:SEG_UNICO    ; assegnazione di tutti i registri di segmento all'unico segmento
03  assume DS:SEG_UNICO
04  assume ES:SEG_UNICO
05  assume SS:SEG_UNICO
06
07  START:                 ; AREA CODICE
08    mov ah,00            ; Sottofunzione 00h di INT 16h, Input di un carattere da Tastiera
09    int 16h              ; Lancio interruzione sw Bios 16h
10    cmp al,30h           ; Confronto carattere in input (AL) con carattere zero (30h = '0')
11    je  OKZERO           ; Se uguali, salta all'indirizzo dell'etichetta OKZERO ove si stamperà 'Z'
12    mov al,6eh           ; Altrimenti si stampa il carattere 'n' (6eh = 'n')
13    mov ah,0eh           ; Sottofunzione 0Eh di INT 10h, Stampa carattere sullo Schermo
14    int 10h              ; Lancio interruzione sw Bios 10h
15    jmp FINE             ; Salta all'etichetta FINE ove si trova il termine del programma
16  OKZERO:
17    mov al,5ah           ; Si stampa il carattere 'Z' (5ah = 'Z')
18    mov ah,0eh           ; Sottofunzione 0Eh di INT 10h, Stampa carattere sullo Schermo
19    int 10h              ; Lancio interruzione sw Bios 10h
20  FINE:
```

```
21    mov ax, 4c00h       ; Terminazione di file .EXE
22    int 21h
23
24  ends
25  end START
```

Molto interessante l'uso delle etichette **OKZERO** e **FINE**: esse consentono di evitare l'uso di indirizzi numerici, lasciando il compito di calcolarli adeguatamente all'assemblatore con la doppia passata.
Le etichette che segnalano indirizzi all'interno dell'area codice sono dette **label**, e devono terminare con i due punti (:). Quando le etichette sono usate nel codice, invece, si parla di **riferimento** all'etichetta.

ATTENZIONE:
A. Il nome di fantasia di una label può apparire una sola volta nel codice (mentre i riferimenti sono liberi), altrimenti l'assemblatore non saprebbe a quale indirizzo associarne il nome.
B. La distanza tra l'etichetta e ogni suo riferimento tramite un salto condizionato rimane vincolata a un massimo di 128 byte. Ciò deriva dal limite progettuale delle istruzioni di salto condizionato.
*Entrambi questi limiti possono essere risolti dall'assemblatore attraverso speciali direttive LOCALS e JUMPS (cfr.*oltre*)*

Le etichette sono identificativi ideati dal programmatore, ovvero nomi di fantasia. Esse non possono né cominciare con un numero, né riportare spazi o caratteri speciali. Va da sé che il programmatore usi nomi significativi per esse, magari in base a indicazioni precise che vengono dette **regole di naming**.

Istruzione INC

Sintassi: **INC destinazione**

Scopo: L'istruzione INC incrementa di una unità **destinazione**, che può essere un registro o una locazione di memoria.

Esempi: **INC AX**
INC DL
INC byte ptr [102]
INC word ptr [BX]

Nota: *Non modifica il registro dei Flags, pertanto se servisse valutarli, si può usare l'equivalente ADD destinazione,1 (cfr. istruzione ADD più oltre). La sua duale è DEC, che ha la stessa sintassi e decrementa di una unità* ***destinazione****.*
Notare che quando ***destinazione*** *è una cella di memoria, va specificata l'ampiezza della cella da incrementare con le parole chiave* ***byte ptr*** *(incrementa solo quella locazione) o* ***word ptr*** *(incrementa il valore in memoria a partire da quella locazione e ampio due byte).*

Istruzione LOOP

Sintassi: **LOOP indirizzo**

Scopo: All'esecuzione di LOOP la CPU decrementa di una unità il registro contatore CX; se CX è diverso da zero, il flusso passa

all'istruzione posta ad indirizzo, altrimenti il flusso dell'esecuzione prosegue regolarmente in sequenza.

Esempi: **LOOP 110h** **LOOP ANCORA**

Nota: *Naturalmente l'iterazione automatica di LOOP funziona solo se, prima del blocco da ripetere chiuso da LOOP, si imposta il registro CX con il numero delle iterazioni desiderate. LOOP salta quasi sempre all'indietro, ovvero indirizzo è quasi sempre una locazione di memoria precedente a LOOP, ma seguente all'impostazione di CX.*

Programma. Cicli

Stampare le 26 lettere minuscole dell'alfabeto inglese.

alfabe.asm

```
00  SEG_UNICO segment     ; definizione dell'unico segmento del programma
01
02  assume CS:SEG_UNICO   ; assegnazione di tutti i registri di segmento all'unico segmento
03  assume DS:SEG_UNICO
04  assume ES:SEG_UNICO
05  assume SS:SEG_UNICO
06
07  START:                ; AREA CODICE
08    mov cx,1ah          ; Numero di iterazioni: 26 (1ah = 26)
09    mov dl,61h          ; Codice Ascii della a minuscola (61h = 'a')
10  ANCORA:
11    mov ah,02           ; Sottofunzione 02h di INT 21h, Stampa carattere sullo Schermo
12    int 21h             ; Lancio interruzione sw MsDos 21h
13    inc dl              ; Incrementa di una unità il codice Ascii
14    loop ANCORA         ; Ripeti CX volte dall'indirizzo 105h, quindi prosegui
15    mov ax, 4c00h       ; Terminazione di file .EXE
16    int 21h
17
18  ends
19  end START
```

Area dati e area codice

Naturalmente le API di MsDos forniscono gli strumenti per memorizzare dati singoli (variabili), array di caratteri (stringhe), array di numeri e l'I/O di stringhe sullo schermo e dalla tastiera.

L'allocazione di dati in memoria avviene tramite una **pseudoistruzione**, ovvero una indicazione contenuta nel codice affinchè il dato da allocare sia semplicemente posto in memoria – per distinguerlo dall' Op. code di una istruzione.

Le pseudoistruzioni non generano alcun codice macchina, ma servono solo per indicare come deve comportarsi un traduttore (nel nostro caso Debug).

Le pseudoistruzioni per allocare dati in memoria sono:

Pseudoistruzione D*

Sintassi: **D* dato**

Scopo: Alloca **dato** all'indirizzo corrente.

Esempi:
```
DB 0
DB 41h
DB 'A'
DB ?
DB 10 DUP (0)
```

Nota: *Nel primo caso viene allocato un byte in memoria (0), nel secondo un byte che rappresenta il codice Ascii della A maiuscola, nel terzo caso un modo equivalente al secondo e nel terzo si allocano due byte contigui.*
Nel caso DB ? si alloca un byte senza inizializzarlo (il valore in quella cella sarà casuale), mentre con DB 10 DUP (0) si indica l'allocazione di 10 byte tutti inizializzati a 0.
E' disponibile anche DW (alloca due byte) DD (4 byte) DQ (8 byte) e DT (10 byte), queste ultime spesso usate per memorizzare numeri in virgola fissa per i calcoli con il coprocessore matematico.

Ovviamente l'area di memoria destinata a contenere i dati (variabili e array) non deve essere eseguita come codice. Essa è detta **area Dati**, e deve essere separata dall'**area Codice**, che contiene le istruzioni da eseguire.

Come si è visto, l'assembler x-86 prevede che i programmi siano sviluppati, attraverso le aree tipiche di un programma quali area di Codice, area Dati e area di Stack, all'interno di zone di memoria contenute in *segmenti*, cioè blocchi di memoria ampi 64 kBytes. Per gli eseguibili di tipo EXE le tre aree tipiche devono essere assegnate ai rispettivi segmenti in memoria (eventualmente sempre lo stesso), ma tuttavia con la necessità di essere esplicitati espressamente nel codice così da poter permettere il caricamento rilocante dinamico. Quindi, per raggiungere gli indirizzi desiderati all'interno dell'area Dati, si utilizza un'istruzione dedicata (**LEA**) che restituisce l'indirizzo numerico di una etichetta.

Istruzione LEA

Sintassi: **`LEAdestinazione, etichetta`**

Scopo: Recupera l'indirizzo effettivo (numerico) di **`etichetta`** e lo pone in **`destinazione`** (LEA: *Load Effective Address*).
`destinazione` deve essere un registro a 16 bit.

Esempi:
```
LEA BX, TITOLO
LEA DX, TABELLA[SI]
```

Nota: *Naturalmente* ***`etichetta`*** *deve essere una etichetta definita nell'area Dati (negli esempi TITOLO e TABELLA.*
Una forma equivalente per recuperare l'indirizzo numerico di una etichetta è anche:
`MOV destinazione, OFFSET etichetta`

Accedere alla memoria

Per leggere e scrivere valori in area dati ma in generale per accedere alla memoria di un programma x-86 bisogna usare la sintassi con le **parentesi quadre**.
Es. Per leggere il primo byte del PSP per un file .COM si usa l'istruzione:
MOV DL,[0].
All'interno delle parentesi quadre si specifica un indirizzo di memoria e le parentesi quadre significano "la cella che si trova a quell'indirizzo".
Questa operazione è detta **deferenziazione**, del tutto analoga a quella ottenuta in linguaggio C con l'operatore * agente su un puntatore..

Analogamente, per scrivere in una locazione di memoria (p.es. modificare il primo byte del PSP): MOV [0],65.

Naturalmente dentro le parentesi quadre si possono usare anche dei registri, per effettuare accessi a locazioni decise a runtime, cioè utilizzando valori di indirizzi variabli.
Es., per leggere tutto il PSP:
```
    MOV AH,2
    MOV BX, 0
    MOV CX,100h


  MOV DL,[BX]
    INT 21h
    INC BX


    LOOP
```

Ricordare che gli unici registri abilitati a indirizzare (cioè a contenere indirizzi e quindi ad essere usanti dento le parentesi quadre) sono **BX, SI, DI, BP**.

Programma. Area Dati e area Codice

Scrivere un programma **m3e.asm** che, presi in input tre caratteri da tastiera, indichi quale è il maggiore (come codice Ascii).

m3e.asm

```
00  SEG_DATI segment           ; definizione del segmento per l'AREA DATI
01    MSG DB "Il max vale:"
02    VAR DB 0
03  ends
04
05  SEG_STACK segment stack    ; definizione del segmento per l'AREA DI STACK
06  ends
07
08  SEG_CODICE segment         ; definizione del segmento per l'AREA CODICE
09
10  assume cs:SEG_CODICE       ; associazione dei nomi dei segmenti ai registri di segmento
11  assume ss:SEG_STACK
12  assume ds:SEG_DATI
13  assume es:SEG_DATI
14
15  START:
16    mov ax, SEG_DATI         ; impostazioni a runtime del registro di segmento DATI
17    mov ds,ax
18    mov es, ax
19    mov ax, SEG_STACK        ; impostazioni a runtime del registro di segmento STACK
20    mov ss, ax
21
22    mov ah,02                ; inizio del codice operativo
23    mov dl,3fh               ; stampa 1mo prompt di input (?)
24    int 21h
25    mov ah,01                ; input 1mo carattere
26    int 21h
27    lea bx,VAR               ; Indirizzo effettivo ottenuto tramite etichetta (LEA)
28    mov [bx],al              ; salvataggio 1mo carattere in memoria (all'indirizzo VAR)
29    mov ah,02                ; stampa 2do prompt di input (?)
30    mov dl,3fh
31    int 21h
32    mov ah,01                ; input 2do carattere
33    int 21h
34    cmp al,[bx]              ; confronto...
35    jl  SALTA1               ; se il carattere in input è minore, non faccio nulla...
36    mov [bx],al              ; ...altrimenti salvo questo input come maggiore.
37  SALTA1:
38    mov ah,02                ; stampa 3zo prompt di input (?)
39    mov dl,3fh
40    int 21h
41    mov ah,01                ; input 3zo carattere
42    int 21h
43    cmp al,[bx]              ; confronto...
44    jl  SALTA2               ; se il carattere in input è minore, non faccio nulla...
45    mov [bx],al              ; ...altrimenti salvo questo input come maggiore.
46  SALTA2:
47    mov cx,0dh               ; stampo il messaggio finale di 12 (=0ch) caratteri,
48    lea bx,MSG               ; ... più uno, il risultato
49  CICLO:
50    mov ah,02
51    mov dl,[bx]
52    int 21h
53    inc bx
54    loop CICLO
55    mov ax, 4c00h            : terminazione speciale per file EXE
56    int 21h
57
58  Ends
59  end START
```

Le tre etichette principali **SEG_DATI**, **SEG_STACK** e **SEG_CODICE** danno il nome ai tre segmenti di memoria di questo programma, segmenti

inizializzati dalla pseudoistruzione **segment/ends**. In effetti questo programma non usa una area dello Stack, quindi in linea di principio non sarebbe necessario specificarla.
Nell'area Codice, infine, la pseudoistruzione **assume cs:,ds:,ss:,es:** consente di associare le tre aree ai registri di segmento appropriati. Si noti come i registro Extra (ES) sia impostato sullo stesso valore del registro dati DS.
Infine, come istruzioni effettive, bisogna impostare i registri di segmento con i valori indicati tramite le pseudoistruzioni.
Si noti come non sia possibile impostare un registro di segmento direttamente, ma solo tramite un registro d'appoggio (in questo caso AX).

La compilazione di un sorgente destinato a diventare un file eseguibile di tipo EXE:

```
C:\tasm>
C:\tasm>tasm m3e    ; l'assemblatore tasm.exe genera il file oggetto m3e.obj dal file sorgente m3e.asm
C:\tasm>tlink m3e   ; il linker tlink.exe genera il file eseguibile m3e.exe dal file oggetto m3e.obj
C:\tasm>
```

Programmi di tipo COM

Nel modello di file eseguibile COM, area Dati e area Codice (e anhe l'area di Stack) risiedono nello stesso segmento da 64kByte, cioè in locazioni di memoria contigue. Se, come spesso si opta, l'area Dati viene posta all'inizio della zona della memoria del programma, è necessario porre una istruzione iniziale di salto incondizionato per saltare l'area Dati e avviare correttamente l'area Codice. In altri casi si può optare allocando l'area Dati immediatamente dopo l'area di Codice.
Inoltre i files COM prevedono il PSP proprio all'inizio del segmento, per una lunghezza di 256 byte (100h): la prima istruzione del codice dovrà trovarsi a quest'indirizzo per non sovrascrivere il PSP.

Direttiva ORG

Sintassi: **ORG valore**

Scopo: La fondamentale direttiva ORG (*Origine*) indica all'assemblatore che l'origine degli indirizzi del segmento deve valere **valore** e non zero.

Esempi: **ORG 100h**

Nota: *Si impone che il Location Counter dell'assemblatore inizi il conteggio degli indirizzi a partire dal valore 100h e non dal valore 0. Tipico caso dell'assemblaggio dei files .COM. Con la direttiva ORG 100h si impone che gli indirizzi dell'eseguibile di tipo COM comincino dal 256mo byte (=100h) del segmento di codice, per saltare l'area del PSP.*

Programma. Area Dati e area Codice per file COM

Allocare i tre codici Ascii della parola HAL e stamparli sullo schermo.

hal.asm

```
00  SEG_UNICO segment      ; definizione dell'unico segmento del programma
01
02  assume CS:SEG_UNICO    ; assegnazione di tutti i registri di segmento all'unico segmento
03  assume DS:SEG_UNICO
04  assume ES:SEG_UNICO
05
06  ORG 100h               ; la prima istruzione di codice deve partire all'indirizzo 256 (100h)
07
08  START:
09  jmp MAIN               ; prima istruzione di codice per saltare l'AREA DATI
10
11  MSG DB 'H','A','L'     ; AREA DATI
12
13  MAIN:                  ; AREA CODICE
14      lea bx,MSG         ; In BX l'indirizzo del primo byte dell'AREA DATI a partire dall'etichetta MSG
15      mov cx,3           ; Contatore del ciclo a 3 (3 caratteri da stampare)
16  ANCORA:
17      mov dl,[bx]        ; Indirizzamento indiretto. In DL il codice Ascii che si trova
                           ; in area Dati all'indirizzo specificato in BX
18      mov ah,2           ; Sottofunzione 02h di MsDos, stampa carattere
19      int 21h            ; Interruzione sw MsDos
20      inc bx             ; Incremento dell'indirizzo in AREA DATI: la prossima cella contiene
                           ; il prossimo codice Ascii da stampare
21      loop ANCORA        ; Iterazione a partire dall'istruzione mov dl,[bx]
22      int 20h            ; Terminazione files COM
23
24  ends
25  end START
```

La compilazione di un sorgente destinato a diventare un file eseguibile di tipo COM:

```
C:\tasm>
C:\tasm>tasm hal     ; l'assemblatore genera il file oggetto hal.obj dal file sorgente
hal.asm
C:\tasm>tlink hal /t ; il linker tlink.exe genera il file eseguibile hal.com dal file
oggetto hal.obj
C:\tasm>
```

Le tre locazioni di memoria così allocate possono ospitare a tutti gli effetti delle variabili. Infatti in quelle locazioni i dati possono essere cambiati a runtime per memorizzare altri valori. Nell'esempio, le tre locazioni vengono manipolate, aggiungendo una unità ad ogni cella, ottenendo i tre codici Ascii della stringa 'IBM', che poi verrà stampata a schermo.

Esempio. Variabili

Allocare in un buffer la parola HAL, quindi aggiungere una unità ad ogni codice Ascii e stampare il buffer risultante.

ibm.asm

```
00  SEG_UNICO segment      ; definizione dell'unico segmento del programma
01
02  assume CS:SEG_UNICO    ; assegnazione di tutti i registri di segmento all'unico segmento
03  assume DS:SEG_UNICO
04  assume ES:SEG_UNICO
05
06  ORG 100h               ; la prima istruzione di codice deve partire all'indirizzo 256 (100h)
07
08  START:
09  jmp MAIN               ; prima istruzione di codice per saltare l'AREA DATI
```

```
10
11  MSG DB "HAL"            ; AREA DATI
12
13  MAIN:                   ; AREA CODICE
14      lea bx,MSG          ; In BX l'indirizzo del primo byte dell'AREA DATI contenuto nell'etichetta MSG
15      mov cx,3            ; Contatore del ciclo a 3 (3 caratteri da modificare)
16  ANCORA:
17      mov al,[bx]         ; lettura della variabile (locazione di memoria puntata da BX)
18      inc al              ; incremento di una unità
19      mov [bx],al         ; salvataggio della variabile (nella locazione di memoria puntata da BX)
20      inc bx              ; prossima variabile in memoria
21      loop ANCORA         ; iterazione
22      lea bx,MSG          ; Ora si stampa il buffer modificato
23      mov cx,3
24  STAMPA:
25      mov dl,[bx]
26      mov ah,2
27      int 21h
28      inc bx
29      loop STAMPA
30      int 20h
31
32  ends
33  end START
```

Modelli di memoria

Per evitare la gestione esplicita delle direttive di segmento (parole chiave *segment* e *assume*), l'assemblatore TASM mette a disposizione una pseudoistruzione molto efficace che semplifica l'uso delle direttive di segmento: con la pesudoistruzione **MODEL**, infatti, il programmatore può decidere il modello di memoria desiderato per il proprio programma, senza più preoccuparsi di gestire i segmenti.

Direttiva MODEL

Sintassi: **`.MODEL tipo`**

Scopo: L'assemblatore regola la generazione dei segmenti di Codice, Dati e Stack in base al tipo indicato

TINY, un solo segmento comune per area Codice, Dati e Stack. Dedicato ai programmi eseguibili di tipo COM

SMALL, un segmento per area Codice, un segmento per area Dati e Stack, solo per eseguibili di tipo EXE

MEDIUM, più segmenti per l'area Codice, un solo segmento per area Dati e area Stack.

COMPACT, più segmenti per l'area Dati, un solo segmento per area Codice e area Stack.

LARGE, più segmenti per l'area Codice, più segmenti per l'area di Dati, più segmenti per l'area di Stack.

HUGE, come LARGE, con la possibilità che un dato contiguo possa essere maggiore di un segmento (es. un array > 64KB).

Esempi: **`MODEL TINY   .MODEL SMALL   .MODEL LARGE`**

Nota: *In realtà esiste un sesto modello di memoria, denominato FLAT, che non prevede segmentazione e usato solo nelle piattaforme a 32bit.*

Ecco come diventa lo stesso sorgente **`m3e.asm`** di un esempio precedente in sintassi semplificata:

Programma. File eseguibile di tipo EXE sintassi TASM

Scrivere un programma **m3et.asm** che, presi in input tre caratteri da tastiera, indichi quale è il maggiore (come codice Ascii).

m3et.asm

```
00  .MODEL SMALL              ; direttiva di segmento per il modello di memoria desiderato
01  .STACK                    ; direttiva per la definizione del segmento dell'area di Stack
02  .DATA                     ; direttiva per la definizione del segmento dell'area Dati
03
04    MSG DB "Il max vale:"
05    VAR DB 0
06
07  .CODE                     ; direttiva per la definizione del segmento dell'area Codice
08
09         mov ax,@DATA       ; impostazioni a runtime del valore di segmento per l'area Dati
10         mov ds,ax
11
12         mov ah,02
13         mov dl,3fh
14         int 21h
15         mov ah,01
16         int 21h
17         lea bx,VAR
18         mov [bx],al
19         mov ah,02
20         mov dl,3fh
21         int 21h
22         mov ah,01
23         int 21h
24         cmp al,[bx]
25         jl  SALTA1
26         mov [bx],al
27  SALTA1:
28         mov ah,02
29         mov dl,3fh
30         int 21h
31         mov ah,01
32         int 21h
33         cmp al,[bx]
34         jl  SALTA2
35         mov [bx],al
36  SALTA2:
37         mov cx,0dh
38         lea bx,MSG
39  CICLO:
40         mov ah,02
41         mov dl,[bx]
42         int 21h
43         inc bx
44         loop CICLO
45         mov ax, 4c00h
46         int 21h
47  end
```

Le direttive **.MODEL**, **.DATA**, **.STACK** e **.CODE** sono dette **direttive di segmento semplificate**, e rendono i sorgenti assembly molto più semplici da gestire, evitando al programmatore lo sforzo di definire i vari segmenti del programma in modo esplicito.
L'unica accortezza da ricordare è il caricamento esplicito del segmento Dati tramite l'etichetta di sistema **@DATA**

Programma COM con sintassi TASM

Per imparare la programmazione assembly x-86 è più che sufficiente sviluppare programmi eseguibili di formato COM, che sono anche più semplici nella struttura, evitando di usare pseudoistruzioni e istruzioni per la definizione e il caricamento dei registri di segmento.
Spesso, infatti, anche i programmi eseguibili di formato EXE vengono ridotti a COM, per sfruttarne la semplicità e la velocità di caricamento, con un apposito applicativo del Sistema Operativo MSDOS, denominato **EXE2BIN**. Ecco i codici sorgenti del solito programma che calcola il massimo di tre caratteri in input, nella versione con direttive di segmento e direttive di segmento semplificate:

sorgente per file COM
con direttive di segmento standard (**MASM**)
m3cm.asm

```
SEG_UNICO segment           ; unico segmento per Area Dati e Codice
assume CS:SEG_UNICO
assume DS:SEG_UNICO

ORG 100H                    ; il codice inizia a 100h, e non a zero

START: jmp MAIN             ; si salta l'area Dati

MSG DB "Il max vale:"       ; area Dati
VAR DB 0

MAIN:                       ; area Codice
 mov ah,02
 mov dl,3fh
 int 21h
 mov ah,01
 int 21h
 lea bx, VAR
 mov [bx],al
 mov ah,02
 mov dl,3fh
 int 21h
 mov ah,01
 int 21h
 cmp al,[bx]
 jl  SALTA1
 mov [bx],al
SALTA1:
 mov ah,02
 mov dl,3fh
 int 21h
 mov ah,01
 int 21h
 cmp al,[bx]
 jl  SALTA2
 mov [bx],al
SALTA2:
 mov cx,0dh
 lea bx,msg
CICLO:
 mov ah,02
 mov dl,[bx]
 int 21h
 inc bx
 loop CICLO
 int 20h
ends                        ; fine segmento unico
end START                   ; fine programma
```

sorgente per file COM
con direttive di segmento semplificate (**TASM**)
m3ct.asm

```
.MODEL TINY                 ; modello di memoria per files COM
.CODE                       ; definizione area Codice e Dati

ORG 100h                    ; il codice inizia a 100h e non a zero

START:  jmp MAIN            ; si salta l'area Dati

MSG DB "Il max vale:"       ; area Dati
VAR DB 0

MAIN:                       ; area Codice
 mov ah,02
 mov dl,3fh
 int 21h
 mov ah,01
 int 21h
 lea bx, VAR
 mov [bx],al
 mov ah,02
 mov dl,3fh
 int 21h
 mov ah,01
 int 21h
 cmp al,[bx]
 jl  SALTA1
 mov [bx],al
SALTA1:
 mov ah,02
 mov dl,3fh
 int 21h
 mov ah,01
 int 21h
 cmp al,[bx]
 jl  SALTA2
 mov [bx],al
SALTA2:
 mov cx,0dh
 lea bx,msg
CICLO:
 mov ah,02
 mov dl,[bx]
 int 21h
 inc bx
 loop CICLO
 int 20h
end START                   ; fine programma
```

Si noti che, dovendo risiedere dati e codice nello stesso segmento, come sempre per i files eseguibili COM, la prima istruzione di codice deve saltare l'area Dati con un salto incondizionato.

La compilazione di un sorgente destinato a diventare un eseguibile di tipo COM deve ricordare al linker di non immettere l'area di startup nell'eseguibile tramite l'**opzione /t**, pertanto i sorgenti di questo tipo devono essere compilati nel seguente modo:

```
C:\tasm>
C:\tasm>tasm m3ct      ; l'assemblatore genera il file oggetto m3ct.obj dal file sorgente
m3ct.asm
C:\tasm>tlink m3ct /t  ; il linker genera il file eseguibile m3ct.com dal file oggetto
m3ct.obj
C:\tasm>
```

Assembly avanzato

Macro

Come sempre accade nella programmazione, speciali valori sono così importanti da meritarsi un nome proprio, così da poterli velocemente individuare all'interno del codice sorgente. Assegnare il nome ad un valore è altresì fondamentale per questioni di manutenzione del codice. Infatti, se il valore dovesse essere modificato, l'uso di un nome simbolico consente di modificare il valore solo una volta, avendo usando solo il nome del valore all'interno del codice. Assegnare un nome a un valore significa definire una **macro** costante.

Macro costanti

Si veda questo breve codice che stampa a schermo la cifra 0 e, a capo, la cifra 1:

```
.MODEL TINY
.CODE
ORG 100h
START:
  mov ah,02
  mov dl,'0'  ; il codice Ascii dello zero (30h) può essere scritto con questa sintassi derivata dal C: 30h = '0'
  int 21h     ; stampa a video il carattere zero
  mov ah,02
  mov dl,0dh  ; il codice Ascii 0dh (=13d) è il Carriage Return (CR). Sposta il cursore all'inizio della riga corrente
  int 21h
  mov dl,0ah  ; il codice Ascii 0ah (=10d) è il Line Feed (LF). Sposta il cursore nella riga sottostante
  int 21h
  mov ah,02
  mov dl,'1'
  int 21h     ; stampa a video il carattere uno
  int 20h
end START
```

OUTPUT

```
C:\>acapo
0
1
C:\>
```

La stampa a schermo dei caratteri Ascii speciali 0dh (=13d) e 0ah (=10d), detti rispettivamente **CR** (*Carriage Return*) e **LF** (*Line Feed*), provoca l'effetto dell' "andare a capo".
Siccome si tratta di valori speciali, usati per un compito dedicato, è buona norma nominarli e usare, nel codice sorgente, il loro nome.
Nominare un valore, significa creare una **costante macro**, cioè un nome simbolico associato ad un valore: quando l'assemblatore incontra quel nome simbolico nel sorgente, sostituisce il simbolo con il valore corrispondente (*espansione della macro*).

Pseudoistruzione EQU

Sintassi: **`nome EQU espressione`**

Scopo: Crea il **`nome`** che sarà sostituito con **`espressione`** durante l'assemblaggio.

Esempi:
```
CR EQU 0dh
RIGA EQU 80
COLONNA EQU 25
SCHERMO EQU RIGA*COLONNA
```

Nota: *Normalmente i simboli delle costanti macro sono scritti in maiuscolo.*
Si noti che l'assemblatore, durante la prima passata, può ricalcolare valori costanti tramite operatori aritmetici (+,-,,/) e sostituire, al simbolo, il valore costante ricalcolato. La pseudoistruzione EQU è del tutto equivalente alla direttiva #define del linguaggio C.*

Ovviamente la pseudoistruzione EQU non genera alcuna riga di codice macchina, essendo una direttiva. L'assemblatore si limita a sostituire ai simboli individuati nel sorgente, i rispettivi valori costanti durante il compile time. Per questo motivo le costanti EQU vanno citate prima dell'inizio del codice.
Il programma sottostante è equivalente al precedente; usa EQU per indicare costanti speciali. L'output rimane invariato

```
CR EQU 13       ; Direttive EQU per CR e LF
LF EQU 10

.MODEL TINY
.CODE
ORG 100h
START:
  mov ah,02
  mov dl,'0'
  int 21h
  mov ah,02
  mov dl,CR     ; il simbolo CR sarà sostituito con il valore 13 (=0dh)
  int 21h
  mov dl,LF     ; il simbolo LF sarà sostituito con il valore 10 (=0ah)
  int 21h
  mov ah,02
  mov dl,'1'
  int 21h
  int 20h
end START
```

Macro di codice

Dovendo stampare diverse righe di zeri e uni, nel codice dovremmo usare varie volte le sei righe di codice che stampano a schermo un 'acapo'.
Il codice ripetuto appesantisce il sorgente e lo rende meno leggibile, cosicchè è possibile riunire un blocco di codice sorgente e assegnargli un nome simbolico, citando il solo nome nel codice. In questo caso si parla di **macro di codice**.
Come prima, durante la prima passata, l'assemblatore, quando incontra il nome simbolico di una macro di codice, sostituisce ad essa l'intero blocco di codice corrispondente (*espansione della macro*), operazione di nuovo eseguita a compile time.

Pseudoistruzione MACRO/ENDM

Sintassi:
```
nome MACRO
   (codice)
ENDM
```

Scopo: Crea il blocco di **(codice)** che sarà sostituito al simbolo **nome** durante l'assemblaggio..

Esempi:
```
BEEP MACRO        ACAPO MACRO
 mov ah,2          mov ah,2
 mov dl, 7         mov dl, 13
 int 21h           int 21h
ENDM               mov dl, 10
                   int 21h
                  ENDM
```

Nota: *Negli esempi, una macro BEEP che emette un suono (infatti il codice Ascii speciale 7 non emette simboli sullo schermo, ma un breve beep). Quindi una macro ACAPO che emette un acapo sullo schermo.*

Si veda il seguente codice, che stampa una sequenza di zeri e uni su quattro righe:

```
ACAPO MACRO          ; Definizione della macro, con nome simbolico ACAPO
  mov ah,2
  mov dl, 13
  int 21h
  mov dl, 10
  int 21h
ENDM                 ; terminazione del blocco macro

.MODEL TINY
.CODE
ORG 100h

START:
  mov ah,02
  mov dl,'0'
  int 21h
  ACAPO              ; Uso della macro. In questo punto la macro ACAPO verrà espansa nelle 5 istruzioni che la compongono
  mov ah,02
  mov dl,'1'
  int 21h
```

```
  ACAPO           ; Uso della macro. Altre 5 istruzioni espanse
  mov ah,02
  mov dl,'0'
  int 21h
  ACAPO           ; Uso della macro. Altre 5 istruzioni espanse
  mov ah,02
  mov dl,'1'
  int 21h
  int 20h
end START
```

OUTPUT

```
C:\>macapo
0
1
0
1
C:\>
```

Macro con parametri e etichette

Le macro di codice diventano veramente interessanti se utilizzate con *parametri*, ovvero se dotate di argomenti che possono essere variabili nel momento dell'uso.
La seguente è una macro che stampa un carattere sullo schermo, indicato al momento dell'uso:

```
STAMPACAR MACRO carattere
  mov ah, 2
  mov dl, carattere
  int 21h
ENDM
```

Il suo uso e' intuibile:

```
STAMPACAR 'P'
```

Se invece una macro dovesse contenere una o più etichette, si presenterebbe il problema dell'**uso ripetuto dell'etichetta**, dato che la macro viene espansa nel codice del programma e il nome dell'etichetta verrà ripetuto tante volte quante volte la macro è usata.
Per ovviare a questo problema si usa una direttiva dedicata **LOCAL** che consente di dichiarare le etichette utilizzate nella macro, lasciando il compito all'assemblatore di gestirne correttamente la ripetizione.

Direttiva LOCAL

Sintassi: **LOCAL nome**

Scopo: Impone all'assemblatore di trasformare il simbolo nome usato in una macro in un simbolo univoco per ogni espansione della macro.

Nota: *La direttiva va posta all'inizio del blocco di codice della macro.*

Si osservi questo codice che acquisisce un carattere in input dopo aver mostrato un rudimentale prompt ('?'), e presenta una macro che stampa un carattere, ma solo se numerico:

```
STAMPACARNUM MACRO regchar  ; regchar è il parametro della macro, un registro a 8bit
LOCAL NONOK                 ; l'etichetta NONOK deve essere dichiarata con la direttiva LOCAL
  cmp regchar, '0'
  jl NONOK
  cmp regchar, '9'
  jg NONOK
  mov ah, 2
  mov dl, regchar
  int 21h
NONOK:                      ; la dichiarazione dell'etichetta impedisce l'errore di duplicazione, nel caso di più usi della macro
ENDM

.MODEL TINY
.CODE
ORG 100h

START:
   mov ah,2                 ; stampa a video del carattere ? come prompt per l'input
   mov dl, '?'
   int 21h
   mov ah,0                 ; input di un carattere da tastiera, senza echo. Il carattere digitato ritorna in AL
   int 16h
   STAMPACARNUM al          ; AL è il valore del parametro della macro
   int 20h
end START
```

OUTPUT

```
C:\>mparam
?1
C:\>
```

(si è digitato il carattere 1, che viene regolarmente stampato a schermo)

Istruzioni aritmetiche

Una stringa numerica (decimale) è una stringa i cui codici Ascii sono compresi tra 48 (30h) e 57 (39h), pertanto per verificare se una stringa è numerica è sufficiente controllarne ogni codice Ascii e verificare che sia compreso entro questi due limiti.
D'altra parte se si deve stampare un numero decimale ad una singola cifra sullo schermo, è sufficiente trovarne il codice Ascii aggiungendo 48 (30h).
Vediamo quindi le quattro istruzioni per il calcolo aritmetico.

Istruzione ADD

Sintassi: **ADD sorgente, destinazione**

Scopo: Effettua la somma (anche con segno) tra **sorgente** e **destinazione**. Il risultato viene collocato in **sorgente**. **destinazione** non può essere un immediato, ma può essere una zona di memoria

Esempi: **ADD BX, 256**
ADD BX, CX
ADD [102], CX
ADD byte ptr [BX], 1

Nota: *Come per INC/DEC, se agisce su una zona di memoria, va precisata la dimensione del* **sorgente** *con le parole chiave* ***byte ptr*** *o* ***word ptr****.*

Istruzione SUB

Sintassi: **SUB minuendo, sottraendo**

Scopo: Sottrae da **minuendo** il **sottraendo** (anche con segno). Il risultato viene collocato in **minuendo**. **minuendo** non può essere un immediato, ma può essere una zona di memoria.

Esempi:
```
SUB BX, 256
SUB BX, CX
SUB [102], CL
SUB word ptr [BX], 1
```

Nota: *Vedi ADD.*

Istruzione MUL

Sintassi: **MUL moltiplicatore**

Scopo: Effettua la moltiplicazione senza segno tra: AL e **moltiplicatore**, se **moltiplicatore** è a 8 bit, oppure tra AX e **moltiplicatore**, se **moltiplicatore** è a 16 bit. Nel primo caso colloca in AX il risultato, nel secondo caso in DX:AX. **moltiplicatore** non può essere un immediato, ma può essere una cella di memoria.

Esempi:
```
MUL CH
MUL [102]
MUL AX
```

Nota: *Se il risultato è maggiore del contenitore, saranno impostati i flag di Overflow o di Carry, altrimenti azzerati.*

Istruzione DIV

Sintassi: **DIV divisore**

Scopo: Effettua la divisione senza segno tra: AX e **divisore**, se **divisore** è a 8 bit, oppure tra DX:AX e **divisore**, se **divisore** è a 16 bit. Nel primo caso colloca in AL il quoziente e in AH il resto, nel secondo caso in AX il quoziente e in DX il resto. **divisore** non può essere un immediato, ma può essere una cella di memoria.

Esempi:
```
DIV BL
DIV [102]
DIV AX
```

Nota: *Se il quoziente non sta nel contenitore, avviene un errore di overflow o di divisione per zero. Es., MOV AX,0A100; MOV BL,2; DIV BL; genera un errore perché A100h / 2 = 5080h, che non sta in un byte.*

Stack

Solo con la programmazione assembly il programmatore può utilizzare espressamente la zona di memoria dello **stack**.
Ricordiamo che tutti i linguaggi ad alto livello usano lo stack, ma in modo trasparente al programmatore, per allocare/deallocare le variabili locali, far transitare i parametri alle procedure e gestire gli indirizzi di andata e ritorno delle subroutines.
Per velocizzare tutti questi processi, lo stack assume la forma di una struttura dati a **pila** (o **LIFO**, *Last In, First Out*).

L'immissione di un valore nello stack si appoggia sull'ultimo valore presente nello stack, in modo tale che l'ultimo valore immesso, sempre in cima alla pila, sia immediatamente accessibile. Per raggiungere i valori sotterrati nella pila è necessario scaricare quelli che lo ricoprono, come quando si vuole prendere un piatto in mezzo ad una pila di piatti.

Per gestire velocemente le operazioni di scrittura (inserimento) e lettura (prelevamento) dallo stack, l'ISA x-86 prevede istruzioni specifiche (rispettivamente **PUSH** e **POP**) e automatismi specifici su alcuni registri: il registro **SP** (*Stack Pointer*) contiene sempre e automaticamente l'indirizzo dell'ultimo elemento sulla cima dello stack.

Lo stack x-86 è organizzato a *word* (due byte), ovvero ogni elemento in pila è sempre ampio due byte.
Lo stack x-86 inizia (ha la base) sempre alla fine di un segmento di memoria, ovvero l'indirizzo del primo elemento di uno stack ha sempre valore di offset pari a FFFEh.
In altre parole, all'avvio di un qualsiasi programma eseguibile (EXE o COM) il registro SP contiene sempre il valore **FFFE**h.
Ciò significa che la pila dello stack x-86 **cresce diminuendo** gli indirizzi (dello Stack Pointer) di due unità alla volta per ogni elemento.
Questa scelta è opportuna, dato che lo stack si amplia a runtime senza controllo: se si perde il controllo dello stack e lo si riempie indefinitamente (**Stack Overflow**), vengono sovrascritte locazioni di memoria del programma, ma non del Sistema Operativo.
Le istruzioni per la gestione esplicita dello stack sono:

Istruzione PUSH

Sintassi: **`PUSH sorgente`**
Scopo: Decrementa SP di due unità e pone **`sorgente`** sullo stack all'indirizzo contenuto in SP.
Esempi: **`PUSH AX`**
`PUSH [BX]`
Nota: ***`sorgente`*** *non può essere un valore immediato, almeno nell'8086/88, ma può essere una locazione di memoria, purchè ampia due byte.*

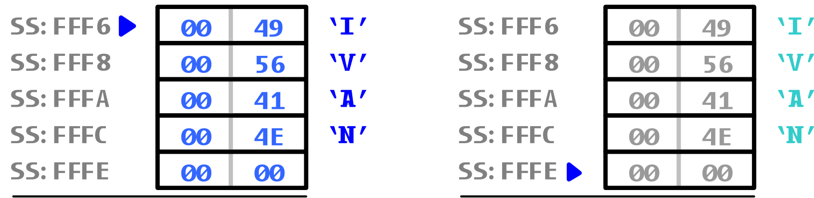
Istruzione POP

Sintassi: **`POP destinazione`**
Scopo: Preleva una word dallo stack, dall'indirizzo contenuto in SP, e la deposita in **`destinazione`**, quindi incrementa SP di due unità.
Esempi: **`POP CX`**
`POP [SI]`
Nota: *Nel primo caso, il valore a due byte in cima allo stack viene posto in CX. Nel secondo caso il valore in cima allo stack viene posto direttamente in memoria, occupando due celle contigue a partire dall'indirizzo specificato (SI).*

Ad esempio, per salvare sullo stack la parola NAVI, bisogna inserire sullo stack i quattro codici Ascii 4eh ('N'), 41h ('A'), 56h ('V'), 49h ('I') con

quattro istruzioni PUSH. I codici Ascii andranno memorizzati in uno o più registri (non è importante), ma a 16 bit:

```
MOV AX, 4eh
PUSH AX
MOV AX, 41h
PUSH AX
MOV AX, 56h
PUSH AX
MOV AX, 49h
PUSH AX
```



Quindi si potrebbero riprendere e stampare a video:

```
MOV AH,2
POP DX
INT 21h
POP DX
INT 21h
POP DX
INT 21h
POP DX
INT 21h
```

ottenendo la parola bifronte IVAN.

Si consulti ora questo codice, che stampa in binario il valore memorizzato nella variabile VAR allocata in memoria. Si usa l'istruzione DIV (divisione) per memorizzare i resti delle divisioni per due, memorizzarli sullo stack, quindi riprenderli per stampare le cifre binarie.

```
.MODEL TINY
.CODE
ORG 100h

START:
        jmp MAIN

VAR DW 00A1h        ; area Dati; il valore VAR (A1h) sarà convertito in binario

MAIN:
        mov ax,VAR  ; il dividendo in AX
        mov bl,2    ; il divisore in BL
        mov cx,0    ; conterà il numero di divisioni, cioè il numero di cifre binarie
ANCORA:
        div bl      ; divisione per 2, in AH il resto, in AL il risultato
        push ax     ; salvataggio del resto e del risultato sullo stack
        inc cx      ; conteggio del numero delle cifre binarie
        mov ah,0    ; annullamento del resto, rimarrà solo il risultato per la prossima divisione
        cmp al,0    ; il risultato è zero?
```

```
        jne  ANCORA   ; se no, si continua la divisione per due
STAMPA:
        pop  dx       ; si preleva dallo stack il valore, tra cui il resto della divisione per due
        mov  dl,dh    ; si mette il resto (0 o 1) in DL per la stampa a schermo
        add  dl,'0'   ; si aggiunge il codice Ascii dello zero per ottenere il codice Ascii del numero ('0' o '1') corrispondente
        mov  ah,2
        int  21h      ; stampa a schermo della cifra binaria
        loop STAMPA   ; ancora cifre da stampare?
        int  20h
end START
```

OUTPUT

```
C:\>bin
10100001
C:\>
```

Lo stack di questo programma si riempie nel seguente modo, a seguito di otto chiamate PUSH AX.
Nella colonna in grigio, i resti, a fianco i risultati

Stack	SP	valori	Descrizione
ss:FFEE	►	0100	8va divisione: 01h / 2 = 00h, resto 1
ss:FFF0		0001	7ma divisione: 02h / 2 = 01h, resto 0
ss:FFF2		0102	6ta divisione: 05h / 2 = 02h, resto 1
ss:FFF4		0005	5ta divisione: 0Ah / 2 = 05h, resto 0
ss:FFF6		000A	4ta divisione: 14h / 2 = 0Ah, resto 0
ss:FFF8		0014	3za divisione: 28h / 2 = 14h, resto 0
ss:FFFA		0028	2da divisione: 50h / 2 = 28h, resto 0
ss:FFFC		0150	1ma divisione: A1h / 2 = 50h, resto 1
ss:FFFE		0000	Valore iniziale dello Stack Pointer

Bisogna ricordare che le operazioni sullo stack devono sempre essere **bilanciate**, ovvero lo Stack Pointer (SP) deve sempre tornare al valore di partenza alla fine del programma. Le istruzioni PUSH e POP, pertanto, devono essere eseguite lo stesso numero di volte.

Procedure

L'uso delle macro di codice semplifica notevolmente la scrittura dei programmi assembly ed è vivamente consigliata. Un effetto collaterale dell'uso delle macro di codice è l'espansione del codice sorgente e del codice eseguibile, cioè il suo incremento in quantità. Ciò significa anche maggior memoria principale utilizzata. Le macro di codice, inoltre, rallentano il tempo di compilazione e, soprattutto, non possono adeguarsi circa situazioni che avvengono solo a *runtime*.
Una soluzione a questi problemi è l'uso di **procedure** (*subroutines*) che solo apparentemente svolgono un compito analogo alla macro.
Una procedura è ancora un blocco di codice con un nome simbolico, ma stavolta il *nome* della procedura non è un simbolo ma l'**indirizzo della sua prima istruzione** in memoria.
Le procedure, infatti, sono allocate in memoria, in uno spazio privato, e devono essere chiamate a *runtime* dal codice del programma (o da altre procedure).
Ciò significa che il blocco di codice di una procedura non viene ripetuto nel sorgente ad ogni occorrenza del suo nome, ma solo usato dal chiamante: il blocco di codice di una procedura è unico e allocato in memoria, cioè le procedure operano a *runtime*.

Le procedure, pertanto, non incrementano il codice sorgente ed eseguibile del programma, e quindi risparmiano anche nell'uso della memoria principale, rispetto alle macro di codice. Inoltre, operando a runtime, possono crearsi veri e propri ambienti autonomi di elaborazione, es. mediante la creazione, sempre a runtime, di zone di memoria private, dette **variabili locali**.
L'unico effetto collaterale di una procedura, rispetto alle macro di codice, è una maggior lentezza nell'esecuzione, dato che il codice chiamante deve preparare la memoria (di solito lo stack) per avviare la procedura, e la procedura, a sua volta, deve ripristinare la memoria al suo termine e prima di ritornare al chiamante. Queste operazioni sono dette **meccanismo di chiamata** e ritorno della procedura.

Le procedure vanno definite con una sintassi molto simile a quella delle macro di codice, anche se la collocazione delle procedure deve essere posta necessariamente nell'area codice, prima del programma principale, o dopo.

PseudoIstruzioni PROC / ENDP

Sintassi: **nome PROC**
(codice)
ENDP

Scopo: Definisce il blocco di **(codice)** che sarà chiamato tramite l'istruzione **CALL nome**. Al termine bisogna ridare il controllo al chiamante con l'istruzione **RET**

Esempi:

```
BEEP PROC          ACAPO PROC
 mov ah,2            mov ah,2
 mov dl, 7           mov dl, 13
 int 21h             int 21h
 ret                 mov dl, 10
ENDP                 int 21h
                     ret
                    ENDP
```

Nota: *I due esempi sono simili a quelli riportati nella sintassi delle direttive MACRO/ENDM, ma il blocco di codice termina con l'istruzione* **ret** *per completare il meccanismo di chiamata.*
Di fianco a PROC si può aggiungere il modificatore ***FAR*** *se la chiamata avviene da un segmento di codice differente da quello che contiene la procedura (per i modelli di memoria MEDIUM, LARGE e HUGE)*

Una procedura deve essere chiamata dal codice del programma e quindi deve ritornare al chiamante per consentirgli il regolare flusso di esecuzione. L'Isa x-86 preve due istruzioni caratteristiche per gestire il meccanismo di chiamata:

Istruzione CALL

Sintassi: **CALL target**

Scopo: L'istruzione CALL esegue le seguenti operazioni:
1) salva nello stack l'indirizzo di ritorno;
2) trasferisce il controllo all'operando target tramite un salto incondizionato.
L'indirizzo di ritorno è l'indirizzo dell'istruzione successiva a quella di CALL.Preleva una word dallo stack, dall'indirizzo contenuto in SP, e la deposita in **destinazione**, quindi incrementa SP di due unità.

Esempi:

```
CALL ACAPO
CALL word ptr [BX]OP CX
```

Nota: *Nel primo caso la CALL ACAPO può essere vista come l'unione delle due istruzioni*
PUSH IP
JMP ACAPO, ricordando che il nome di una procedura è il suo indirizzo in memoria.
Nel secondo caso un esempio di chiamata dinamica, ovvero una chiamata che assume valore solo a runtime (in base al valore attuale di BX). **Word ptr** *serve per indicare all'assemblatore che la locazione puntata da BX riguarda due byte contigui a partire dall'indirizzo contenuto in BX.*

Istruzione RET

Sintassi: **RET**

Scopo: L'istruzione RET assume che l'indirizzo di ritorno si trovi attualmente in cima allo stack. Essa esegue le seguenti operazioni:
1) preleva dallo stack dell'indirizzo di ritorno
2) salto all'indirizzo di ritorno.

Esempi: **RET**

Nota: *La RET, che va sempre posta come ultima istruzione di un blocco di procedura, esegue, praticamente, le seguenti istruzioni:*
POP indirizzoritorno
JMP indirizzoritorno,
oppure, con una sola istruzione logica: POP IP

In entrambi i casi, se la procedura è di tipo **FAR** – cioè si trova in un segmento di codice differente da quello del chiamante, sia CALL che RET, invece di salvare/rileggere solo la parte *offset* dell'indirizzo del program counter (indirizzo di ritorno su due byte), salvano e rileggono sia la parte *seg* che la parte *offset* dell'indirizzo (indirizzo di ritorno su quattro byte) in modo del tutto trasparente al programmatore.

Meccanismo di chiamata

Si veda il seguente esempio che illustra il meccanismo di chiamata e ritorno di una procedura tramite lo stack.
La colonna in grigio mostra gli indirizzi effettivi delle righe di codice.

```
          .MODEL TINY
          .CODE
          ORG 100h

          START:
cs:0100     jmp MAIN      ; All'avvio si deve saltare il codice delle procedure

          ACAPO PROC      ; La procedura sta nell'area di codice, ma deve essere saltata all'avvio, così come si salta l'area Dati
cs:0103     mov ah,2
cs:0105     mov dl, 13
cs:0107     int 21h
cs:0109     mov dl, 10
cs:010B     int 21h
cs:010D     ret           ; L'istruzione RET è necessaria per far funzionare il meccanismo di ritorno
          ENDP

          MAIN:
cs:010E     mov ah,02
cs:0110     mov dl,'0'
cs:0112     int 21h
cs:0114     call ACAPO    ; La procedura ACAPO viene chiamata esplicitamente con l'istruzione CALL, per garantire il ritorno
cs:0117     mov ah,02
cs:0119     mov dl,'1'
cs:011B     int 21h
cs:011D     int 20h
          end START
```

OUTPUT

```
C:\>pacapo
0
1
C:\>
```

Lo stack del programma subisce il seguente movimento (tre passi, compreso lo stato iniziale):

```
 Stack SP valori  Descrizione

ss:FFFA     ....
ss:FFFC     ....
ss:FFFE ►  0000      Valore iniziale dello Stack Pointer

ss:FFFA     ....
ss:FFFC ►  0117      Indirizzo di ritorno, sulla CALL ACAPO
ss:FFFE    0000      Valore iniziale dello Stack Pointer

ss:FFFA     ....
ss:FFFC     ....
ss:FFFE ►  0000      Dopo la RET nella procedura ACAPO
```

Preservare i registri

L'utilizzo delle procedure comporta un effetto collaterale abbastanza grave, detto **interferenza**: i registri usati dalla procedura sovrascrivono il contenuto precedentemente salvato in quei registri dal chiamante, con l'effetto che al ritorno della procedura il chiamante non ritrova più i valori precedentemente salvati nei registri.
Per evitare l'interferenza, la procedura deve preservare i registri in ingresso, ovvero salvare il contenuto dei registri che essa stessa userà al suo interno, salvandoli ordinatamente sullo stack, per poi ripristinarli ordinatamente appena prima di ritornare il controllo al chiamante (appena prima dell'istruzione RET).
La preservazione dei registri può essere effettuata puntualmente, salvando sullo stack solo i registri usati dalla procedura, o in modo completo sfruttando due apposite istruzioni x-86, PUSHA e POPA che, rispettivamente, salvano sullo stack e riprendono dallo stack tutti i registri (ma solo per l'x-86 a partire dall'80186, con l'esclusione, quindi, dell'8086/88).

Il codice precedente, dotato di preservazione dei registri, appare come segue (l'output non cambia):

```
.MODEL TINY
.CODE
ORG 100h

START:
  jmp MAIN

ACAPO PROC
  push ax        ; si preservano i soli registri AX e DX, gli unici usati dalla procedura, inviandoli sullo stack
```

```
  push dx
  mov ah,2
  mov dl, 13
  int 21h
  mov dl, 10
  int 21h
  pop dx         ; si ricaricano i registri preservati, in ordine inverso, per restituirli invariati al chiamante
  pop ax
  ret
ENDP

MAIN:
  mov ah,02
  mov dl,'0'
  int 21h
  call ACAPO
  mov ah,02
  mov dl,'1'
  int 21h
  int 20h
end START
```

OUTPUT

```
C:\>pacapo
0
1
C:\>
```

Passaggio di parametri

Le procedure diventano realmente fondamentali quando permettono il **passaggio dei parametri**, ovvero possono svolgere il proprio compito sulla base di valori che il chiamante decide a *runtime*.
In realtà si è già usato un sistema di passaggio di parametri, ad esempio durante l'uso delle interruzioni sw: valorizzare un registro prima della chiamata all'istruzione INT significa passare – **tramite registro** – un parametro alla routine dell'interruzione sw.
Il passaggio dei parametri tramite registri è molto veloce e semplice, ma ha molte limitazioni, prima di tutto la quantità dei registri disponibili.
Le procedure, per linguaggi ad alto e a basso livello come l'assembly, usano in realtà lo stack per passare i parametri e, quando serve, per ritornarli al chiamante.
L'idea è semplice: il chiamante, prima di chiamare la procedura con la consueta istruzione CALL, deposita sullo stack i valori che intende passare alla procedura. La procedura, prima di iniziare il suo compito, preleva dallo stack i parametri e li usa al suo interno.
Per ritornare valori dalla procedura al chiamante, si usa lo stesso meccanismo.
In questo caso il passaggio di parametri si dice **tramite lo stack**.

Il passaggio di parametri tramite lo stack deve tener presente che, sullo stack, come ultimo valore, verrà sempre posto l'indirizzo di ritorno della procedura– ad opera dell'istruzione CALL. Pertanto la procedura dovrà prelevare i parametri senza eliminare dalla cima dello stack l'indirizzo di

ritorno, che dovrà essere usato dall'istruzione RET per ritornare correttamente al chiamante.

Esistono varie tecniche per passare i parametri sullo stack. Le più diffuse prendono il nome di **cdecl** (usata dal linguaggio C e derivati) e **stdcall** (usata dal linguaggio Pascal e dalle API di alcuni Sistemi Operativi).
In questa sezione vedremo un passaggio di parametri alle procedure abbastanza simile allo stile del C o cdecl, che usa il registro **BP** (*Base Pointer*) per prelevare i dati sullo stack senza modificare il registro **SP** (*Stack Pointer*). Si ricorda che il registro BP ha la proprietà di indirizzare in memoria, cioè di contenere indirizzi di memoria.

1. Prima di tutto il chiamante deve porre nello stack i parametri richiesti dalla procedura. L'operazione si effettua con la consueta istruzione PUSH, ripetuta tante volte quanti sono i parametri da passare.
2. Quindi si effettua la chiamata normalmente, con l'istruzione CALL. Essa immetterà sulla cima dello stack, come di consueto, l'indirizzo di ritorno.
3. La procedura, a sua volta, deve immediatamente salvare sullo stack il registro BP, dato che verrà usato e sovrascritto per prelevare i parametri.
4. Quindi il registro BP deve essere impostato con il valore dello Stack pointer SP, mediante una istruzione MOV: in questo modo BP punta alla cima dello stack.
5. Ora i parametri possono essere prelevati uno a uno tramite BP, avendo cura di ricordare che il primo parametro è profondo 4 byte nello stack: infatti i primi due byte in cima alla pila riportano il valore di BP (appena memorizzato), e i successivi due byte riportano il valore dell'indirizzo di ritorno. Ogni parametro si scosta di due byte, pertanto a BP+4 corrisponde il valore del primo parametro, a BP+6 il valore del secondo parametro, a BP+8 il valore del terzo parametro, e così via.
6. Ora può essere scritto il codice della procedura, comprese le eventuali istruzioni per preservare i registri.
7. Infine, appena prima dell'istruzione RET, va ripristinato il registro BP, che se tutto è stato svolto correttamente, si trova attualmente in cima allo stack. Una volta prelevato il valore originale di BP, l'indirizzo di ritorno è disponibile in cima alla pila per l'istruzione RET.
8. Il chiamante, quando riprende il controllo, si ritrova i parametri ancora sullo stack, per cui deve ripristinare lo stato dello stack deallocandoli, cioè facendo tornare lo Stack Pointer SP al valore originario. Ciò è semplice, tramite una istruzione ADD: si aggiungono allo Stack Pointer tante 'doppiette' quanti sono i parametri (es., per 3 parametri: ADD SP,6). Una delle maggiori differenze tra la tecnica cdecl e stdcall consiste nel fatto che cdecl impone che sia il chiamante a deallocare i parametri dallo stack, mentre in stdcall è la procedura a farlo.

Bisogna ricordare che il salvataggio immediato di BP – e il suo successivo ripristino, è fondamentale benchè BP non sia di norma usato dai moduli che chiamano le procedure. Infatti una procedura può – e spesso lo fa, chiamarne un'altra al suo interno (**chiamata annidata**), alla quale passare parametri. Se BP non fosse preservato, le chiamate annidate non funzionerebbero.

Il seguente codice usa una procedura a cui viene passato sullo stack il codice Ascii da stampare a schermo. Siccome lo stack usa elementi a 16 bit, il codice Ascii (8bit) viene enucleato nella parte bassa del registro AX, che e' a 16 bit.

```
          .MODEL TINY
          .CODE
          ORG 100h

          START:
cs:0100   jmp MAIN

          STAMPACAR PROC
cs:0103     push bp         ; si preserva BP sullo stack, come prima istruzione della procedura
cs:0104     mov bp,sp       ; si memorizza lo stack Pointer in BP, in modo che BP possa servire pre reperire il parametro
cs:0106     mov dx,[bp+4]   ; ecco il parametro, profondo 4 byte dentro lo stack (il codice Ascii del ?)
cs:0109     mov ah,2
cs:010B     int 21h
cs:010D     pop bp          ; ripristino di BP. Ora sullo stack c'è l'indirizzo di ritorno, così che RET funzioni a dovere
cs:010E     ret
          ENDP

          MAIN:
cs:010F     mov al,'?'
cs:0112     push ax         ; passaggio del parametro sullo stack (il codice Ascii del ?, in AL all'interno di AX)
cs:0113     call STAMPACAR
cs:0116     add sp,2        ; deallocazione dello stack. Un parametro, una "doppietta"
cs:0119     int 20h
          end START
```

OUTPUT

```
C:\>pparam
?
C:\>
```

Seguendo il listato del programma, si può seguire l'andamento dello stack per ogni istruzione che lo modifica implicitamente (come CALL e RET) o esplicitamente come PUSH, POP e ADD SP,2.

```
 Stack SP valori  Descrizione

ss:FFF8     ....
ss:FFFA     ....
ss:FFFC     ....
ss:FFFE ►  0000     Valore iniziale dello Stack Pointer

ss:FFF8     ....
ss:FFFA     ....
ss:FFFC ►  003F     PUSH AX; 3Fh è il codice Ascii del carattere ?
```

```
ss:FFFE    0000      Valore iniziale dello Stack Pointer

ss:FFF8    ....
ss:FFFA ►  0116      CALL STAMPACAR; 116h è l'indirizzo di ritorno
ss:FFFC    003F      PUSH AX
ss:FFFE    0000      Valore iniziale dello Stack Pointer

ss:FFF8 ►  0000      PUSH BP; in BP c'era il valore 0
ss:FFFA    0116      CALL STAMPACAR
ss:FFFC    003F      PUSH AX
ss:FFFE    0000      Valore iniziale dello Stack Pointer
```

Nella procedura ora si pone in **BP** lo Stack Pointer, con **mov bp,sp**, cioè **BP = FFF8h.**
All'indirizzo **BP+4=FFFCh**, c'è l'indirizzo del parametro sullo stack, cosicchè **mov dx,[bp+4]** pone in **DX** il valore **003Fh**, cioè in **DL** il codice Ascii **(3Fh)** del punto interrogativo.

```
ss:FFF8    0000      POP BP; ripristinato BP (0000h)
ss:FFFA ►  0116
ss:FFFC    003F
ss:FFFE    0000

ss:FFF8    0000
ss:FFFA    0116      RET; caricato l'indirizzo di ritorno (0116h)
ss:FFFC ►  003F
ss:FFFE    0000

ss:FFF8    ....
ss:FFFA    ....
ss:FFFC    ....
ss:FFFE ►  0000      ADD SP, 2 e valore iniziale dello Stack Pointer
```

Variabili locali

Una delle proprietà fondamentali delle procedure è la possibilità di creare un ambiente di memoria privato con il quale interagire per completare compiti anche abbastanza articolati. L'area di memoria privata di una procedura è **allocata sullo stack** e deallocata appena prima del ritorno al chiamante.
Le variabili che prendono posto nell'area privata delle procedure sono dette **variabili locali** o **variabili automatiche**.

Come visto in precedenza, una volta preso il controllo, una procedura memorizza la cima dello stack in BP per poter prelevare eventuali parametri sotterrati nella pila.
Per creare memoria alle variabili locali, bisogna invece estendere lo stack al di sopra della cima, di tante "doppiette" quante sono le variabili locali da creare. Così, utilizzando sempre BP come base, si raggiungeranno le variabili locali con sottrazioni di "doppiette": in BP-2 ci sarà l'indirizzo della prima variabile locale, in BP-4 l'indirizzo della seconda, in BP-6 l'indirizzo della terza, e così via.
Al termine, l'area delle variabili locali deve essere deallocata dalla procedura, riportando lo stack Pointer al suo valore originale.

1. La procedura, dopo aver memorizzato in BP la cima dello stack, lo amplia opportunamente sottraendo allo Stack Pointer SP tante doppiette quante sono le variabili locali da usare, es. SUB SP,4, alloca due variabili locali da due byte l'una (o quattro variabili locali da un byte l'una).
2. Ora la procedura può scrivere nella variabile locale con la consueta MOV, indicando l'indirizzo della variabile tramite BP, es. MOV [BP-2], AX, mette nella prima variabile locale il valore del registro AX
3. Allo stesso modo la procedura può leggere le variabili locali, usando sempre BP per indirizzarle, es. MOV DL, byte ptr [BP-4] pone nel registro DL la variabile locale di ampiezza un byte dalla seconda area di memoria allocata sullo stack.
4. Al termine, la zona delle variabili locali viene deallocata riportando lo Stack pointer SP al valore originale che ora è contenuto in BP (es. MOV SP, BP).

Come esempio vediamo una versione di listato molto simile a quello usato per il passaggio di un parametro. In questo caso si passa alla procedura una cifra ed essa ne stamperà il simbolo Ascii sullo schermo, dopo aver usato una variabile locale per memorizzare la base dei codici Ascii numerici, cioè il codice Ascii di zero (30h):

```
             .MODEL TINY
             .CODE
             ORG 100h

             START:
cs:0100      jmp MAIN

             STAMPANUM PROC
cs:0103        push bp                      ; consueta predisposizione dello stack frame per il prelevamento del parametro
cs:0104        mov bp,sp
cs:0106        mov dx,[bp+4]
cs:0109        sub sp,2                     ; allocazione della variabile locale
cs:010C        mov byte ptr [bp-2],30h      ; scrittura della variabile locale, con il valore 30h
cs:0110        add dx,[bp-2]                ; lettura della variabile locale. Si somma l'Ascii di 0 per ottenere il simbolo
cs:0113        mov ah,2
cs:0115        int 21h
cs:0117        mov sp,bp                    ; deallocazione della variabile locale
cs:0119        pop bp
cs:011A        ret
cs:0113      ENDP

             MAIN:
cs:011B        mov al,9                     ; preparazione del parametro, il numero nove (non il codice Ascii)
cs:011E        push ax                      ; passaggio del parametro sullo stack
cs:011F        call STAMPANUM
cs:0122        add sp,2                     ; deallocazione dello stack
cs:0125        int 20h
             end START
```

OUTPUT

```
C:\>varloc
9
C:\>
```

Seguendo il listato del programma, si può seguire l'andamento dello stack all'atto dell'allocazione e deallocazione dell'area di memoria locale:

Stack	SP valori	Descrizione

```
ss:FFF6    ....
ss:FFF8    ....
ss:FFFA    ....
ss:FFFC    ....
ss:FFFE ►  0000    Valore iniziale dello Stack Pointer

ss:FFF6    ....
ss:FFF8    ....
ss:FFFA    ....
ss:FFFC ►  0009    PUSH AX; 9h è il codice Ascii del carattere zero
ss:FFFE    0000    Valore iniziale dello Stack Pointer

ss:FFF6    ....
ss:FFF8    ....
ss:FFFA ►  0122    CALL STAMPANUM; 122h è l'indirizzo di ritorno
ss:FFFC    0009    PUSH AX
ss:FFFE    0000    Valore iniziale dello Stack Pointer

ss:FFF6    ....
ss:FFF8 ►  0000    PUSH BP; in BP c'era il valore 0
ss:FFFA    0122    CALL STAMPANUM
ss:FFFC    0009    PUSH AX
ss:FFFE    0000    Valore iniziale dello Stack Pointer

ss:FFF6 ►  ....    SUB SP, 2; si alloca un elemento sullo stack
ss:FFF8    0000    PUSH BP; in BP c'era il valore 0
ss:FFFA    0122    CALL STAMPANUM
ss:FFFC    0009    PUSH AX
ss:FFFE    0000    Valore iniziale dello Stack Pointer

ss:FFF6 ►  ..30    MOV byte ptr [bp-2],30h; si scrive nella variabile locale
ss:FFF8    0000    PUSH BP; in BP c'era il valore 0
ss:FFFA    0122    CALL STAMPANUM
ss:FFFC    0009    PUSH AX
ss:FFFE    0000    Valore iniziale dello Stack Pointer

ss:FFF6 ►  ..30    ADD DX,[bp-2]; si legge nella variabile locale
ss:FFF8    0000    PUSH BP; in BP c'era il valore 0
ss:FFFA    0122    CALL STAMPANUM
ss:FFFC    0009    PUSH AX
ss:FFFE    0000    Valore iniziale dello Stack Pointer

ss:FFF6    ..30    questa locazione ora non è più valida
ss:FFF8 ►  0000    MOV SP, BP; dealloca la variabile e ripristina lo Stack Pointer
ss:FFFA    0122    CALL STAMPANUM
ss:FFFC    0009    PUSH AX
ss:FFFE    0000    Valore iniziale dello Stack Pointer

ss:FFF6    ..30
ss:FFF8    0000    POP BP; ripristinato BP (0000h)
ss:FFFA ►  0122
ss:FFFC    0009
ss:FFFE    0000

ss:FFF6    ..30
ss:FFF8    0000
ss:FFFA    0122    RET; caricato l'indirizzo di ritorno (0116h)
ss:FFFC ►  0009
ss:FFFE    0000

ss:FFF6    ....
ss:FFF8    ....
ss:FFFA    ....
ss:FFFC    ....
ss:FFFE ►  0000    ADD SP, 2 e valore iniziale dello Stack Pointer
```

Notazioni per il passaggio di parametri e le variabili locali

Per rendere il codice assembly più leggibile e semplice da utilizzare, è spesso conveniente utilizzare uno stile che fa uso di qualche macro costante per poter servirsi di nomi simbolici al posto delle notazioni che indirizzano brutalmente lo stack, sia per quanto riguarda la gestione dei parametri, che la gestione delle variabili locali.

In questo modo il listato precedente assume la seguente forma (il programma eseguibile è assolutamente identico):

```
.MODEL TINY
.CODE
ORG 100h

START:
jmp MAIN

STAMPANUM PROC
Parametro EQU word ptr [BP+4] ; Il nome Parametro equivale alla zona dello stack che contiene il primo parametro
Variabile EQU byte ptr [BP-2] ; Il nome Variabile equivale alla zona dello stack che contiene la prima variabile locale

  push bp
  mov bp,sp
  mov dx,Parametro            ; Uso del nome simbolico Parametro per recuperare il parametro
  sub sp,2
  mov Variabile,30h           ; Uso del nome simbolico Variabile per scrivere la variabile locale
  add dx,Variabile            ; Uso del nome simbolico Variabile per leggere la variabile locale
  mov ah,2
  int 21h
  mov sp,bp
  pop bp
  ret
ENDP

MAIN:
  mov al,9
  push ax
  call STAMPANUM
  add sp,2
  int 20h
end START
```

Direttive per programmi e librerie

Per una efficiente programmazione assembly, è necessario utilizzare alcune direttive all'assemblatore per superare alcuni limiti architetturali – come il problema della distanza tra etichetta e riferimento per i salti condizionati o per rendere più agevole la scrittura dei programmi – come ad esempio evitare di pianificare l'uso univoco dei nomi delle etichette.
Inoltre è fondamentale conoscere il modo in cui più moduli sorgenti concorrono per generare un file eseguibile, tecnica necessaria per i progetti sw che intendono avvalersi di moduli di libreria.

Direttive JUMPS e LOCALS

Per evitare di incorrere nel problema del salto lungo, cioè quando la distanza tra riferimento e etichetta supera i 128 bytes, è sufficiente citare una direttiva iniziale all'assemblatore, la direttiva **JUMPS**:

Direttiva JUMPS

Sintassi: `JUMPS`

Scopo: Impone all'assemblatore di trasformare il codice di eventuali salti a distanze superiori di 128 bytes, in un codice equivalente in grado di superare tale limite ed effettuare anche salti lunghi.

Nota: *La direttiva va posta all'inizio del modulo sorgente, subito dopo la direttiva che indica l'inizio dell'area Codice (**.CODE**). Spesso si usa anche quando non si è certi della presenza di salti lunghi nel codice. La direttiva vale solo per l'assemblatore TASM.*

La direttiva JUMPS si limita a trasformare il salto condizionato in una struttura di salto che utilizza un salto incondizionato JMP come supporto per raggiungere l'etichetta distante più di 128 byte dal suo riferimento. Infatti l'istruzione di salto incondizionato JMP non ha limiti di distanza tra riferimento e etichetta.

All'interno delle procedure spesso si vorrebbero usare etichette con nomi uguali in procedure diverse, soprattutto per indicare zone logiche del codice equivalenti (es. FINE, OK, ecc.). Questo genera un errore dell'assemblatore, che necessita di nomi univoci per le etichette in tutta l'area di Codice. Per evitare di pianificare uno schema di naming univoco per le etichette da usare nelle procedure, si può usare una direttiva speciale (**LOCALS**) e una notazione che rendono libero il programmatore nella scelta dei nomi:

Direttiva LOCALS

Sintassi: **LOCALS**

Scopo: Impone all'assemblatore di trasformare le etichette scritte con un prefisso speciale **@@nome** univoche aldilà della rimanente parte del **nome**

Nota: *La direttiva va posta all'inizio del modulo sorgente, subito dopo la direttiva che indica l'inizio dell'area Codice (**.CODE**). La direttiva vale solo per l'assemblatore TASM.*

In definitiva, un codice che usa tali direttive, e che stampa due stringhe con due procedure analoghe, è il seguente:

```
.MODEL TINY
.CODE
JUMPS             ; Direttiva per evitare il limite del salto lungo (in questo codice però non ce ne sono)
LOCALS            ; Direttiva per usare etichette con nomi uguali (tramite il prefisso @@)
ORG 100h

START:
  jmp MAIN

MSG_1 DB "Sistemi Abacus",0  ; Stringa ASCIIZ (termina con uno zero)
MSG_2 DB "Classe 3a$"        ; Stringa che termina con il carattere speciale $ (come nel servizio MsDos)

PROC STAMPAASCIIZ            ; procedura che stampa a schermo stringhe ASCIIZ (indirizzo passato sullo stack)
  push bp
  mov bp,sp
  mov bx,[bp+4]
@@ANCORA:                    ; ecco le etichette con il prefisso @@ che consentono nomi uguali in accordo con LOCALS
  mov dl,[bx]
  cmp dl,0
  je @@FATTO
  mov ah,2
  int 21h
  inc bx
  jmp @@ANCORA
@@FATTO:
  pop bp
  ret
ENDP
```

```
PROC STAMPADOLLARO              ; procedura che stampa a schermo stringhe terminanti con $ (indirizzo passato sullo stack)
  push bp
  mov bp,sp
  mov bx,[bp+4]
@@ANCORA:                       ; ecco le etichette con il prefisso @@ che consentono nomi uguali in accordo con LOCALS
  mov dl,[bx]
  cmp dl,'$'
  je @@FATTO
  mov ah,2
  int 21h
  inc bx
  jmp @@ANCORA
@@FATTO:
  pop bp
  ret
ENDP

MAIN:
  lea ax,msg_1
  push ax
  call STAMPAASCIIZ
  add sp,2

  mov ah,2
  mov dl, 10
  int 21h

  lea ax,msg_2
  push ax
  call STAMPADOLLARO
  add sp,2

  int 20h
end START
```

OUTPUT

```
C:\>jumpslcl
Sistemi Abacus
            Classe 3a
C:\>
```

Librerie: direttive INCLUDE, PUBLIC ed EXTRN

Come per i linguaggi ad alto livello, programmare in assembly diventa veramente proficuo quando si possono usare moduli di **libreria**, cioè files sorgenti o binari che contengono procedure o definizioni di utilità generale, utilizzabili nei programmi senza dover, ogni volta, riscrivere la soluzione di problemi già risolti.
Lo sviluppo dei programmi con lo stile del **progetto** e tramite moduli di libreria è una pratica oramai consolidata nel mondo della programmazione.
Un progetto è l'insieme di più moduli sorgenti (a volte anche moduli binari), di cui uno solo contiene il punto di ingresso del programma e, tutti gli altri, sono detti moduli di libreria. La compilazione di un progetto è la

compilazione di ogni modulo, e la loro unione tramite linker nel **target** del progetto, solitamente un file eseguibile.
Naturalmente un progetto deve affrontare il problema dei rapporti tra i moduli i quali possono essere, alternativamente, sia *client* che *server* di funzioni presenti in altri moduli: sono client se citano elementi presenti in altri moduli; sono server se contengono definizioni citate da altri moduli.
Il modulo principale, invece, è l'unico che è sempre un modulo client.

Il modo più semplice per realizzare il rapporto tra il modulo principale e altri moduli server è tramite la direttiva **INCLUDE**.
Con questa direttiva, usata dal modulo principale, si indica all'assemblatore di aprire da disco il file argomento della direttiva (modulo server) ed espanderlo nel modulo principale (modulo client) "così com'è" a partire dalla posizione in cui si trova la direttiva INCLUDE nel modulo principale. Il processo è del tutto paragonabile a quello di una macro di codice. In questo caso la libreria (modulo server) è detta **libreria di codice**.

Direttiva INCLUDE

Sintassi: **INCLUDE nomefile**

Scopo: Impone all'assemblatore di cercare il file nomefile, aprirlo ed espanderlo riga per riga nella posizione corrente.

Nota: *La direttiva può essere posta in qualsiasi zona del sorgente; il nome del file può essere indicato anche con il percorso. Solitamente i files assembly d'inclusione hanno estensione .INC*

Il progetto si compila come se fosse composto da un unico file, il file principale (**mainincl.asm**). Il file server **acapo.inc** deve essere raggiungibile (nell'esempio, è nella cartella corrente):

mainincl.asm

```
.MODEL TINY
.CODE
ORG 100h

START:
     jmp MAIN

INCLUDE acapo.inc        ; qui sarà espanso il file acapo.inc

MAIN:
     mov ah,02
     mov dl,'0'
     int 21h
     call ACAPO
     mov ah,02
     mov dl,'1'
     int 21h
     int 20h
end START
```

acapo.inc

```
CR EQU 13
LF EQU 10

ACAPO PROC
  mov ah,2
  mov dl, CR
  int 21h
  mov dl, LF
  int 21h
  ret
ENDP
```

OUTPUT

```
C:\>tasm mainincl
C:\>tlink mainincl /t
```

```
C:\>mainincl
0
1
C:\>
```

Più spesso il programmatore usa **librerie binarie**, ovvero moduli server che vengono assemblati autonomamente e collegati ai moduli client durante la fase di linking.
I moduli client devono dichiarare in testa al codice quali simboli tratti da moduli esterni verranno usati (direttiva **EXTRN**), in modo che l'assemblatore non cada in errore incontrando simboli mai definiti.
A sua volta il server deve dichiarare quali simboli possono essere utilizzati da altri moduli (direttiva **PUBLIC**) in modo che l'assemblatore e il linker sappia come effettuare il collegamento.

Direttiva EXTRN

Sintassi: **`EXTRN nome:tipo`**

Scopo: Indica all'assemblatore che un certo simbolo **`nome`** non è definito nel modulo sorgente attuale, bensì in uno esterno. **`tipo`** può essere NEAR o FAR se **`nome`** è il nome di una procedura; può essere BYTE o WORD se **`nome`** è l'etichetta in un'area dati.

Nota: *La direttiva può essere posta in testa al modulo client, per mettere in evidenza la lista di simboli esterni al sorgente, detti anche **dipendenze**.*
*Per quanto riguarda i nomi delle procedure, il **`tipo`** è sempre NEAR se il modello di memoria scelto è TINY, SMALL e COMPACT; FAR negli altri casi.*
Ogni direttiva EXTRN dovrebbe essere associata ad una duale direttiva PUBLIC contenuta in un modulo esterno.

Direttiva PUBLIC

Sintassi: **`PUBLIC nome`**

Scopo: Indica all'assemblatore che un certo simbolo **`nome`** può essere utilizzato da moduli esterni.

Nota: *La direttiva può essere posta in testa al modulo server, per mettere in evidenza la lista di simboli pubblici che il modulo offre ai moduli client.*
*Naturalmente ogni **`nome`** in ogni direttiva PUBLIC del modulo deve corrispondere ad una effettiva etichetta nel modulo (funzione o dato).*

Lo stesso progetto di poco fa, implementato con libreria binaria:

mainlib.asm

```
EXTRN ACAPO: NEAR

.MODEL TINY
.CODE
ORG 100h

START:
        mov ah,02
        mov dl,'0'
        int 21h
        call ACAPO
        mov ah,02
        mov dl,'1'
        int 21h
        int 20h
end START
```

libreria.asm

```
PUBLIC ACAPO
.MODEL TINY
.CODE

ACAPO PROC
  mov ah,2
  mov dl, 13
  int 21h
  mov dl, 10
  int 21h
  ret
ACAPO ENDP
end
```

In questo caso il processo di compilazione è radicalmente differente rispetto all'uso delle libreire sorgenti tramite la direttiva INCLUDE.
I due moduli sono assemblabili autonomamente, e danno luogo a due files oggetto .OBJ.

Sarà il linker a effettuare il collegamento tra i due moduli binari, come dalla seguente sintassi:

OUTPUT

```
C:\>tasm mainlib                 ; assemblaggio modulo client (principale). Genera mainlib.obj
C:\>tasm libreria                ; assemblaggio modulo server (libreria). Genera libreria.obj
C:\>tlink mainlib libreria /t    ; correlazione (linking) dei moduli. Genera mainlib.exe
C:\>mainlib
0
1
C:\>
```

Makefile

Nel caso della compilazione di progetti con librerie binarie, risulta molto utile utilizzare l'utility **MAKE** in dotazione con *Borland C* (file **MAKE.EXE**).
Il programma *Make* accetta in input un file di testo provvisto delle regole di compilazione di un progetto, ed esegue ordinatamente tutti i passi necessari per la sua compilazione, assemblando i vari moduli sorgenti (client e server) e linkandoli adeguatamente.
Un **makefile** quindi è un file di testo scritto con una determinata sintassi, spesso di nome makefile (senza estensione), che viene dato in input al programma *make.exe*.

Se il processo è esente da errori, l'output di make sarà il file *target* (solitamente un file eseguibile) e tutti i files intermedi del caso (solitamente files .obj).

Il makefile **mainlib.mak** per il progetto precedente, risulterebbe quindi come il seguente (le righe che iniziano con # sono commenti):

mainlib.mak

```
#uso: make -f mainlib.mak

.AUTODEPEND

mainlib.exe:
	TLINK mainlib.obj libreria.obj /t

libreria.obj: libreria.asm
	TASM libreria.ASM,libreria.OBJ

mainlib.obj: mainlib.asm
	TASM mainlib.ASM,mainlib.OBJ
```

Il processo di make, infine, si avvia nel seguente modo:

OUTPUT

```
C:\>make -f mainlib.mak
MAKE Version 3.6  Copyright (c) 1992 Borland International
Available memory 15728640 bytes

TLINK mainlib.obj libreria.obj /t
Turbo Link  Version 5.1 Copyright (c) 1992 Borland International
C:\>
```

Non è il caso di approfondire il discorso sui makefile, che non rientra negli obiettivi di questo testo. In ogni caso si tratta di un argomento di grande importanza per tutti i linguaggi di programmazione, anche ad alto livello.