Appunti di

PROGRAMMAZIONE DEL MICROPROCESSORE

A cura di Domenico Valle

Sommario

CAPITOLO 1

SISTEMI DI NUMERAZIONE

1.1 Sistemi di numerazione posizionali

Un **sistema di numerazione posizionale** è un criterio basato sull'uso di un numero limitato di segni grafici (cifre) per la rappresentazione dell'insieme infinito dei numeri.

Un sistema di numerazione posizionale è costituito da :

a) una **base b;**

b) un insieme di **cifre** distinte l'una dalle altre;

c) un **codice**, cioè un insieme di regole che permettono di interpretare il numero rappresentato da un gruppo di cifre;

d) un insieme di **algoritmi** per le quattro operazioni fondamentali.

Data una sequenza di cifre, il codice di interpretazione specifica che :

a) un peso è associato ad ogni posizione nella sequenza; i pesi sono crescenti a partire dalla cifra più a destra che ha peso 0 per un numero intero e -m per un numero frazionario di m cifre. La cifra più a destra è la cifra meno significativa, quella più a sinistra la più significativa;

b) ogni cifra indica quante volte deve essere considerato il peso corrispondente alla posizione nella quale si trova la cifra stessa.

Un **sistema di numerazione in base b** è un sistema di numerazione posizionale la cui base è b.

Un **numero in base b** è un numero rappresentato utilizzando il sistema di numerazione in base b.

Le cifre utilizzabili in un sistema di numerazione in base b vanno da 0 a (b-1), pertanto non si possono avere i sistemi di numerazione con base 0 e base 1.

Sia C_i la cifra di posizione i di un numero in base b di n cifre intere ed m cifre frazionarie, esso può essere rappresentato nella seguente forma detta **forma polinomia** :

[1.1] $C_{n-1}*b^{n-1} + C_{n-2}*b^{n-2} +.....+ C_1*b + C_0 + C_{-1}*b^{-1} + C_{-2}*b^{-2} +.....+ C_{-m}*b^{-m}$

Praticamente tutti i popoli dell'antichità usavano dei sistemi di numerazione di tipo additivo : es. i numeri romani sono composti da cifre ciascuna delle quali corrisponde sempre allo stesso valore, qualunque sia la sua posizione all'interno del numero; la quantità rappresentata è data dalla somma dei valori associati alle cifre usate nel numero corrispondente.

Nella vita di tutti i giorni oltre al sistema di numerazione decimale (base 10), molto utilizzato è il sistema di numerazione sessagesimale (base 60) : misura delle ore, misura in gradi degli angoli, misura della latitudine e della longitudine.

In informatica sono largamente usati i sistemi di numerazione binario ed esadecimale.

Sistema di numerazione Binario:
- base 2
- cifre 0,1 dette anche cifre binarie (bit : binary digit)

Sistema di numerazione Esadecimale:
- base 16
- cifre 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F (A,B,C,D,E,F hanno rispettivamente il valore 10,11,12,13,14,15)

1.2 Conversione di un numero in base b (con b<>10) a decimale

La rappresentazione del numero nella forma polinomia fornisce un metodo diretto per la determinazione del suo corrispondente nel sistema decimale.

1.3 Conversione di un numero da decimale a una base b (con b<>10)

1.3.1 Conversione di numeri interi

Viene utilizzato il metodo delle **divisioni successive** :
a) si effettua una successione di divisioni per b, utilizzando come dividendo, per la prima divisione, il numero decimale da convertire e, per le divisioni successive, il quoziente della divisione precedente;
b) il resto di ogni divisione determina una cifra del numero cercato a partire dalla cifra meno significativa;
c) il processo termina quando il quoziente è nullo.

1.3.2 Conversione di numeri frazionari

Viene utilizzato il metodo delle **moltiplicazioni successive** :
a) si effettua una successione di moltiplicazioni; nella prima moltiplicazione si moltiplica per b il numero decimale da convertire, nelle moltiplicazioni successive, si moltiplica per b la parte frazionaria del risultato della moltiplicazione precedente;
b) la parte intera di ogni moltiplicazione determina una cifra del numero cercato a partire da quella più significativa;
c) il processo termina
  c1) quando la parte frazionaria del risultato di una moltiplicazione è nulla;
  c2) fissando a priori il numero massimo di moltiplicazioni da effettuare.

1.3.3 Conversione di numeri misti (numero composto da una parte intera e da una parte frazionaria)

Si applicano le regole appena viste, separatamente, alla parte intera e alla parte frazionaria del numero.

1.4 Relazione tra numeri binari, decimali , esadecimali

1.4.1 Relazione tra numeri binari e numeri decimali

Con n bit il numero massimo decimale rappresentabile è $2^n - 1$:
si possono rappresentare tutti in numeri compresi tra 0 e $2^n - 1$.

1.4.2 Relazione tra numeri binari e numeri esadecimali

È immediato il passaggio da binario ad esadecimale e viceversa tenendo presente che per rappresentare una cifra esadecimale in binario necessitano 4 bit.

1.5 La forma complemento

1.5.1 Definizioni

Sia N un numero intero in base b di k cifre, si definisce :
- **complemento a b (complemento alla base) di N** il numero $C_b(N)$ tale che valga la seguente relazione :

[1.2] $N + C_b(N) = b^k$

- **complemento a b-1 (complemento alla base-1) di N** il numero $C_{b-1}(N)$ tale che valga la seguente relazione :

[1.3] $N + C_{b-1}(N) = b^k - 1$

Si tenga presente che, per qualsiasi base b :

[1.4] b^k = 1 seguito da k cifre uguali a 0
[1.5] $b^k - 1$ = k cifre tutte uguali a b-1.

1.5.2 Procedimenti per il calcolo del complemento a 1 e del complemento a 2 di un binario numero:

- **complemento a 1** : si trasformano le cifre 0 in 1 e viceversa;
- **complemento a 2** : a) si somma 1 al suo complemento a 1
  oppure, direttamente,
  b) a partire dalla cifra meno significativa si lasciano invariati tutti i bit fino al primo 1,si invertono tutti i bit successivi uguali a 1 in 0 e quelli uguali a 0 in 1.

1.5.3 Sottrazione con la forma complemento

Qualunque sia la base b è vera la relazione :

[1.6] $A - B = A + C_b(B)$

Procedimento per **calcolare la differenza usando la forma complemento** :
a) se necessario, si rappresentano il minuendo e il sottraendo con lo stesso numero di cifre (aggiungendo degli 0 alla sinistra della cifra più significativa del

sottraendo);
b) si calcola il complemento a b del sottraendo;
c) si somma il minuendo al complemento del sottraendo;
d) si trascura la cifra più significativa della somma ottenuta.

CAPITOLO 2

RAPPRESENTAZIONE DELLE INFORMAZIONI

2.1 Introduzione

Tutte le informazioni immesse in un elaboratore o fornite da un elaboratore sono rappresentate come sequenze di caratteri che fanno parte di un insieme di caratteri che potremmo chiamare **alfabeto esterno.**

Nel mondo occidentale tale alfabeto è costituito da
- le 26 lettere maiuscole e minuscole dell'alfabeto inglese
- le 10 cifre decimali
- i caratteri speciali : segni di interpunzione, simboli matematici, spazio, caratteri di controllo.

L'elaboratore è però in grado di trattare dati solo se rappresentati come sequenze di caratteri di un **alfabeto interno** che comprende i soli **caratteri 0 1** , cioè le cifre binarie o bit. I dati sono pertanto rappresentati all'interno di un elaboratore come numeri binari.

Per comunicare con un elaboratore è necessario, perciò, definire un **codice,** cioè una legge che **associa ad ogni carattere dell'alfabeto esterno uno e un solo numero binario**.

Essendo il bit una unità di informazione troppo piccola per poter essere elaborata in modo efficiente, normalmente vengono trattati gruppi di bit :
Byte : 8 bit
Word : gruppi di byte di lunghezza variabile; normalmente 2,3,4 byte (16,24,32 bit).

2.2 Codifica di dati alfanumerici

Dato alfanumerico : sequenza di caratteri dell'alfabeto esterno; con i dati alfanumerici non è possibile effettuare elaborazioni di tipo numerico quali addizioni, sottrazioni, etc...

I codici più utilizzati sono :
- **EBCDIC** (Extended Binary Code Decimal Interchange Code)
- **ASCII** (American Standard Code for Information Interchange)

L'EBCDIC utilizza, per rappresentare un carattere dell'alfabeto esterno, 1 byte, visto come due semibyte, quello di sinistra (più significativo) chiamato ZONE quello di destra (meno significativo) chiamato DIGIT.

L'ASCII è in realtà un codice a 7 bit, forzato a 8 ponendo a 0 l'ottavo bit (più significativo). Un carattere ASCII può essere visto come due semibyte (ZONE e DIGIT) come nell' EBCDIC.

2.3 Codifica dei numeri

2.3.1 BCD (Binary Coded Decimal)

Ogni cifra di un numero intero è rappresentata con il suo corrispondente numero binario su 4 bit.

2.3.2 Codifica in virgola fissa (rappresentazione fixed point)

Consiste nel rappresentare un numero assumendo una posizione prefissata del punto di separazione tra parte intera e parte frazionaria. In particolare può essere usata per rappresentare i numeri interi. Vi possono essere più codifiche per la rappresentazione binaria in virgola fissa ; esse differiscono per il criterio usato nella rappresentazione dei numeri negativi :

- **valore assoluto e segno** : si rappresentano il segno e il valore assoluto (modulo) separatamente. Questo tipo di codifica presenta diversi inconvenienti :

a) nelle operazioni aritmetiche è necessario trattare il bit del segno in modo diverso da quello usato per trattare gli altri bit;

b) vi è una doppia codifica dello 0.

Il rango dei numeri rappresentabili con n bit è

[2.1] $-2^{n-1} + 1 \ldots\ldots\ldots 2^{n-1} - 1$

- **forma complemento a 2** : è la più utilizzata; consiste nel rappresentare ogni numero negativo per mezzo del complemento a 2 del corrispondente numero positivo; in questa forma al bit più significativo viene associato come peso l'equivalente potenza di 2 ma con segno negativo; questo bit nel caso di numeri positivi sarà sempre 0 e nel caso di numeri negativi sarà 1 : pertanto può essere considerato il bit di segno.

Il rango dei numeri rappresentabili con n bit è

[2.2] $-2^{n-1} \ldots\ldots\ldots\ldots 2^{n-1} - 1$

- **forma complemento a 1**: un numero negativo è rappresentato dal complemento a 1 del corrispondente numero positivo; al bit più significativo viene associato come peso l'equivalente potenza di 2 ma diminuita di 1 e con segno negativo; anche in questo caso esistono 2 codifiche per lo 0.

Il rango dei numeri rappresentabili è

[2.3] $-2^{n-1} + 1 \ldots\ldots\ldots 2^{n-1} - 1$

2.3.3 Codifica in virgola mobile (floating point)

Il rango dei numeri rappresentabili in virgola fissa è limitato; si può ampliare questo rango utilizzando la codifica in virgola mobile (rappresentazione scientifica).

Un numero in base b viene considerato come il prodotto di due parti :

- **fattore di scala** : è una potenza della base (l'esponente viene chiamato anche caratteristica)
- **parte frazionaria** (mantissa) : è un numero tale che moltiplicato per il fattore di scala dà il numero che si vuole rappresentare.

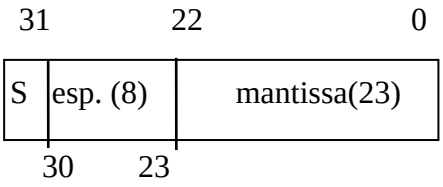
La rappresentazione scientifica si dice **normalizzata** se la parte frazionaria è tale che :

- in decimale
  a) è minore di 1;
  b) la cifra più significativa è diversa da 0.
- in binario (ANSI/IEEE)
  a) il primo bit diverso da 0 è immediatamente a sinistra della virgola.

Nel caso di numeri binari, da rappresentare su un certo numero di bit, la mantissa può essere codificata nella forma valore assoluto con segno o nella forma complemento a 2; la caratteristica molto spesso e ' codificata nella forma in eccesso a t : si somma alla caratteristica stessa una costante t uguale a 2**(k-1) – 1 con k numero dei bit utilizzati per rappresentare la caratteristica; in questo modo si evita l'uso del segno per la rappresentazione dell'esponente.

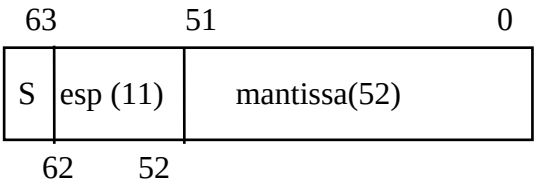
La **rappresentazione floating point** varia da calcolatore a calcolatore; si riporta la notazione **ANSI/IEEE 754 standard** con la quale si è cercato di mettere ordine in questo campo :

SR (Short Real)



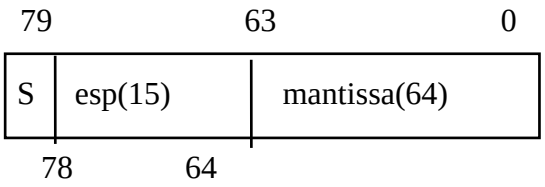
valori rappresentabili $1,18*(10^{-38})<=|x|<=3,4*(10^{38})$

LR (Long Real)



valori rappresentabili $2,23*(10^{-308})<=|x|<=1,80*(10^{308})$

XR (eXtended Real)



valori rappresentabili $3,4*(10^{-4932})<=|x|<=1,2*(10^{4932})$

Eccesso $t = 2^{k-1}-1$, con k = n. bit utilizzati per rappresentare la caratteristica.
La mantissa è normalizzata; l'1 che precede il "." e il "." non vengono rappresentati.

CAPITOLO 3

ELEMENTI DI PROGRAMMAZIONE

3.1 Introduzione

Per risolvere un problema utilizzando un microprocessore ed in generale un computer occorre seguire una ben precisa sequenza di operazioni che possiamo raggruppare sotto le due fasi:

- **Analisi** : definizione del problema e stesura dell'algoritmo;
- **Programmazione** : scrittura del programma;

Scopo dell'**analisi** è quello di definire un **algoritmo** : successione di istruzioni che definiscono le operazioni da eseguire su dei dati per ottenere dei risultati (la parola algoritmo deriva dal nome di un matematico arabo, al-Khuwarizmi, vissuto nel nono secolo D.C.). Possiamo dire anche che un algoritmo è un elenco di istruzioni che permette di risolvere una classe di problemi; non è detto però che ogni elenco di istruzioni sia un algoritmo: esistono dei precisi requisiti che devono essere soddisfatti affinché un elenco di istruzioni possa essere considerato un algoritmo; essi sono :

a) **Finitezza** : deve essere finito;
b)**Generalità** : deve valere per tutti i problemi di una determinata classe;
c)**Non ambiguità** : qualunque sia l'esecutore ogni istruzione deve essere interpretata sempre allo stesso modo.

Scopo della **programmazione** è quello di definire un **programma** : descrizione dell'algoritmo in una forma comprensibile e quindi eseguibile da parte dell'elaboratore.

3.2 Analisi

3.2.1 Definizione del problema

In questa fase è necessario definire, in modo il più completo possibile, alcune specifiche relative al problema che si vuole risolvere :

- Definizione delle uscite (output);
- Definizione degli input necessari al programma per ottenere gli output desiderati;
- Definizione di come gli output devono essere generati dagli input (funzione di trasferimento se si vuol vedere il programma come un sistema, algoritmo in termini prettamente informatici).

3.2.2 Stesura dell'algoritmo : diagramma a blocchi

Vi sono diversi modi per schematizzare l'algoritmo formulato in fase di definizione del problema. Uno dei più utilizzati è il diagramma a blocchi.

Il diagramma a blocchi o flow-chart è una rappresentazione grafica dell'algoritmo; esso indica il flusso (cioè la sequenza) delle operazioni da eseguire per realizzare la trasformazione, descritta nell'algoritmo, dei dati iniziali per ottenere i risultati finali.

Un particolare simbolo grafico, detto **blocco elementare** o più semplicemente **blocco,** è associato ad ogni tipo di istruzione. I blocchi sono collegati tra loro da linee munite di frecce che indicano la sequenza delle azioni.

Un diagramma a blocchi è un insieme di blocchi elementari costituito da:
- un blocco iniziale
- un blocco finale
- un numero finito n (n>=1) di blocchi di azione e/o blocchi di lettura/scrittura
- un numero finito m (m>=0) di blocchi di controllo.

Un **algoritmo** (e relativo diagramma a blocchi) è **strutturato** se può essere visto come una composizione delle sole strutture fondamentali :
- **sequenza**
- **selezione**
- **iterazione**

(Teorema di Iacopini-Bohm : qualsiasi diagramma a blocchi può essere trasformato in un altro che esegue la stessa funzione e che utilizza soltanto le strutture fondamentali).

3.3 Programmazione

Definito l'algoritmo, esso deve essere messo in una forma che l'elaboratore (che è l'esecutore delle nostre istruzioni) possa capire per poterlo eseguire : ciò viene fatto descrivendo le azioni individuate nell'algoritmo tramite un linguaggio di programmazione , che pertanto può essere definito come un formalismo per descrivere all'elaboratore un algoritmo.

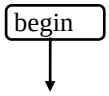
Un certo tempo, percentualmente non indifferente, della fase di programmazione deve essere dedicato al cosiddetto **"debugging**", cioè alla prova del programma.

3.4 Costanti e variabili

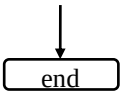
Esistono due tipi di insiemi di dati che possono essere usati in un programma : le costanti e le variabili.
La differenza è costituita dal fatto che mentre una costante mantiene lo stesso valore per l’intera durata del programma, una variabile può cambiare il suo contenuto durante l’elaborazione, ad esempio per effetto di un calcolo.

ESERCIZI

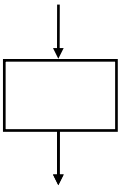
1) Fare l'esempio di un diagramma a blocchi strutturato e di uno non strutturato. Convertire il diagramma a blocchi non strutturato in uno strutturato.
2) Data una stringa convertire tutti i caratteri alfabetici minuscoli in caratteri alfabetici maiuscoli.
3) Data una stringa convertire tutti i caratteri non alfabetici in spazio.
4) Data una stringa convertire tutti i caratteri numerici in lettere.
5) Data una stringa convertire tutte le doppie alfabetiche in triple.
6) Data una stringa invertire il contenuto.
7) Scrivere un programma per eseguire la somma dei primi 100 numeri naturali



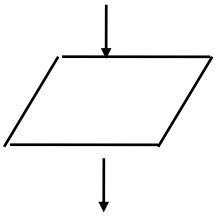
blocco iniziale



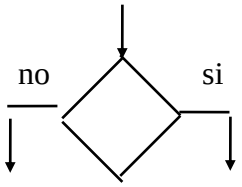
blocco finale



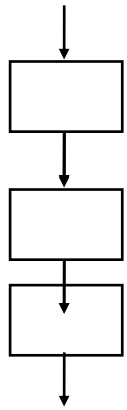
blocco azione



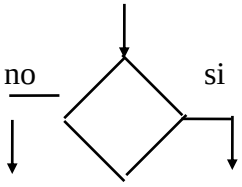
blocco di I/O



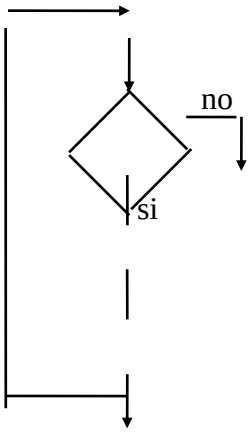
blocco di controllo



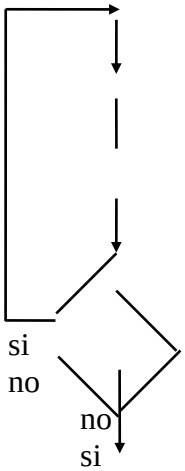
sequenza



selezione



iterazione while ...do



iterazione do ....while
repeat....until

CAPITOLO 4

LA PROGRAMMAZIONE IN LINGUAGGIO ASSEMBLY

4.1 Il linguaggio assembly

I linguaggi di programmazione possono essere suddivisi in linguaggi ad alto livello e linguaggi a basso livello, a seconda se sono più vicini al linguaggio umano o al linguaggio della macchina.

Il linguaggio assembly è un linguaggio a basso livello; esso viene anche detto linguaggio simbolico : un elaboratore è in grado di comprendere solo istruzioni in termini di sequenza di 0 ed 1, per cui il programmatore dovrebbe codificare i vari passi del programma esclusivamente utilizzando i 2 simboli binari, ciò sarebbe ovviamente quasi impossibile da realizzare : il linguaggio assembly permette di specificare le varie operazioni e gli operandi (dati) con notazioni simboliche (sequenza di caratteri alfanumerici, in particolare i codici operativi (istruzioni) sono dei codici mnemonici) invece che con notazioni binarie .

Nel linguaggio assembly le istruzioni mantengono una corrispondenza uno a uno con quelle previste dal linguaggio macchina ed è quindi strettamente condizionato dalla struttura hardware del processore; non è pertanto definibile uno standard come per i linguaggi ad alto livello (C, Pascal, Fortran, Cobol, Basic, etc.).

Detto set di istruzioni (instruction set) la lista delle istruzioni che il mP può eseguire, il linguaggio assembly può essere definito come la rappresentazione in una forma facile da comprendere dall'uomo del set di istruzioni di un determinato mP.

4.2 L'Assembler

Un programma scritto in linguaggio assembly non è comunque capito dal mP: l'Assembler è appunto un programma che traduce il programma scritto in linguaggio assembly (programma sorgente) in un programma scritto in linguaggio macchina (programma oggetto).

Molto spesso le parole Assembler e linguaggio macchina vengono utilizzate al posto di linguaggio assembly.

4.2.1 Funzionamento dell'Assembler

L'assembler ha il compito di scandire il programma sorgente al fine di creare un programma oggetto in codice binario che potrà poi essere caricato in memoria centrale per l'esecuzione.

Elementi tipici di un assemblatore sono

- **il location counter (LC : contatore di locazioni) :** indica di volta in volta il numero di locazioni di memoria occupate dalle istruzioni esaminate sino a quel momento;

- **la symbol table (tavola dei simboli)** : contiene le coppie "nome simbolico-indirizzo fisico";
- **la tabella dei codici operativi (predefinita)** : contiene le coppie "codice operativo simbolico - codice operativo macchina".

La fase di **traduzione** richiede per lo meno **due scansioni (passate)** del programma sorgente:
- **I passo** : viene esaminato frase per frase tutto il programma e tramite l'LC viene simulata l'assegnazione della memoria centrale al programma oggetto; in questa fase viene costruita la tavola dei simboli;
- **II passo** : viene nuovamente esaminato il programma frase per frase. Ogni istruzione viene tradotta utilizzando le informazioni contenute nella tabella dei codici operativi e nella tavola dei simboli precedentemente costruita.

Gli assemblatori possono essere suddivisi in due grandi categorie :
- **assoluti :** attribuiscono indirizzi definitivi alle istruzioni ed ai dati, cioè indirizzi che corrispondono esattamente alle locazioni di memoria in cui verranno poste le informazioni. In questo caso l'assembler effettua il calcolo degli indirizzi a partire da quello specificato dall'utente;
- **rilocabili :** danno agli indirizzi solo valori relativi, con riferimento all'indirizzo specificato per la prima istruzione. Quando il programma verrà caricato in memoria, tutti gli indirizzi dovranno essere modificati in funzione dell'indirizzo effettivo della prima istruzione; questa operazione viene effettuata da un opportuno programma detto loader (caricatore).

Macroassemblatori, infine, sono detti quegli assemblatori che permettono la definizione di macroistruzioni: blocco di istruzioni individuato da un nome e caratterizzato da dei parametri di scambio, definito una volta per tutte in testa al programma e richiamato all'occorrenza all'interno del programma. Il macroassemblatore per ogni macroistruzione che incontra produce un'espansione locale della medesima nel senso che sostituisce alla chiamata della macroistruzione la parte di codice ad essa associata, introducendo al posto dei parametri formali i corrispondenti valori dei parametri attuali.

Le macroistruzioni vengono anche chiamate **sottoprogrammi aperti** a differenza dei sottoprogrammi propriamente detti (**sottoprogrammi chiusi**), in quanto nel programma eseguibile le istruzioni della macro compaiono tante volte quante sono richiamate mentre le istruzioni del sottoprogramma compaiono una volta sola.

Il prodotto dell'assembler non è il programma eseguibile. È un programma collegatore **(linker)** che, dopo aver costruito un unico programma organico mettendo insieme gli eventuali moduli distinti che possono comporre il programma e collegato il medesimo con le routine di utilità messe a disposizione dall'assemblatore (libreria di sistema), genera il modulo eseguibile che, a sua volta, dovrà essere caricato in memoria **(dal loader)** per poter essere eseguito.

4.3 La struttura di un programma assembly

Il linguaggio assembly permette al programmatore di rappresentare il codice macchina di un'istruzione mediante un'abbreviazione simbolica (codice mnemonico), esso inoltre consente di esprimere con nomi simbolici i valori costanti, gli indirizzi dei dati, gli indirizzi delle istruzioni.

Un programma assembly si presenta come una serie di frasi (statement) che possono essere :

- **istruzioni eseguibili** : costituiscono il programma vero e proprio e verranno tradotte in linguaggio macchina;
- **direttive di assemblaggio** (pseudoistruzioni) : non verranno tradotte in linguaggio macchina ma servono a dare delle indicazioni per effettuare la traduzione.

Ogni **statement è suddiviso in 4 campi** (non sempre necessariamente tutti presenti)

etichetta codice operativo operandi commento

- etichetta : nome simbolico dello statement;
- codice operativo : codice simbolico dell'istruzione;
- operandi : dati coinvolti nell'istruzione;
- commento : contiene delle informazioni esplicative della funzione dello statement; non viene preso in considerazione in fase di traduzione in linguaggio macchina.

4.4 Uso del linguaggio assembly

Tenendo presente che il linguaggio assembly permette di avere un diretto controllo del processore, esso viene utilizzato :

- quando è impossibile o difficoltoso fare qualcosa con un linguaggio ad alto livello;
- per avere programmi più veloci (un programma spende il 90% del tempo per eseguire il 10% delle istruzioni : queste potrebbero essere scritte in assembly);
- per avere una piena conoscenza del processore;
- per gioco.

4.5 Il programmatore in linguaggio assembly

Un programmatore è una persona capace di definire un piano di azione nel tempo: dato un problema lo risolve facendo certe azioni in un determinato ordine.

Al programmatore in linguaggio assembly si richiede in particolare :

- di porre molta attenzione ai dettagli;
- di essere capace di leggere e comprenderei manuali tecnici;
- di possedere buone capacità di calcolo;
- pazienza ;
- costanza.

Sono inoltre indispensabili i seguenti requisiti :

- Conoscenza del DOS (per programmazione assembly in ambiente DOS);
- Conoscenza di un Editor;
- Conoscenza dell'aritmetica esadecimale;
- Esperienza di programmazione.

4.6 Strumenti utilizzati nella programmazione in linguaggio assembly

Per poter programmare in linguaggio assembly bisogna disporre dei seguenti programmi:
- **EDITOR** (Es. Edit del DOS) : permette di scrivere il programma sorgente (source code);
- **ASSEMBLER** (Es. MASM della Microsoft, TASM della Borland o incluso nel "Development System" di alcuni linguaggi di programmazione ad alto livello quali C o Pascal) : programma che partendo dal "source code" genera "l'object code" cioè il programma in linguaggio macchina;
- **LINKER** (Es. LINK del DOS, TLINK della Borland): genera il programma eseguibile (l'EXE);
- **DEBUGGER** (Es. DEBUG del DOS, TD della Borland):aiuta nella fase di debugging (prova) del programma.

Sono di grande utilità, inoltre :
- **CROSS-REFERENCE PROGRAM** : permette di avere la lista alfabetica di tutti i simboli usati e il rispettivo indirizzo di memoria;
- **LIBRARY MANAGER** : collezione di moduli oggetto;
- **MAKE :** documenta le relazioni esistenti tra un insieme di file;
- **PROFILER** : analizza e misura le performance del programma.

4.7 Sistemi di sviluppo integrati Hardware/Software

Il linguaggio assembly ha sicuramente facilitato di molto la scrittura di programmi in linguaggio macchina, permettendo a chi lo usa di ottenere il massimo dalla macchina a disposizione in termini di velocità, di compattezza e di efficienza.

Tuttavia nell'ambito dell'informatica industriale lascia ancora insoluti molti problemi.

Supponiamo infatti di dover programmare una piccola scheda a mP per controllare la temperatura di un forno : in ingresso ci sono i segnali provenienti dalle sonde termiche e in uscita i comandi per pilotare le resistenze di riscaldamento. Tale scheda prende il nome di calcolatore dedicato perché finalizzata ad un impiego specifico e non modificabile. È ovvio che tale scheda sarà priva di un software ad alto livello e dell'Hardware tipico di un calcolatore quali tastiera, video, memorie di massa, etc., pertanto il software deve essere sviluppato in un altro ambiente e quindi il programma, in linguaggio macchina, riversato sulla scheda.

La diversità di ambiente potrà riservare delle sorprese e ciò che funziona sul calcolatore utilizzato per lo sviluppo del software potrà non funzionare sulla scheda.

Per ovviare a questi inconvenienti sono stati creati degli ambienti software e dei calcolatori particolari chiamati **sistemi di sviluppo**.

Si devono distinguere 2 casi:

- la CPU del sistema di sviluppo è la stessa del sistema dedicato;
- la CPU del sistema di sviluppo è diversa dalla CPU del sistema dedicato : in questo caso vengono utilizzati degli assembler particolari chiamati Cross Assembler che generano un codice diverso dal codice della CPU che li ospita.

Il calcolatore di sviluppo viene chiamato **Host**, quello dedicato **Target**.

L'Host deve avere delle caratteristiche particolari:

- una area di memoria RAM e una memoria di massa di debite dimensioni;
- una tastiera e un video;
-una CPU di emulazione (fisica o virtuale) con connettore esterno ICE (In Circuit Emulator) uguale a quella utilizzata dal target, che dovrà essere in grado di sostituire fisicamente quest'ultima mediante il connettore.

Una volta messo a punto il programma esso dovrà essere trasferito sul target. Vi sono due possibilità a seconda delle dimensioni e prestazioni del target stesso:

- tramite porta seriale;
- il programma viene memorizzato su un supporto fisico (ROM,EPROM,PROM), che viene quindi sistemato sul target.

4.8 Microprocessori e microprocessore 8086

Microprocessore : introdotto nel 1971 è sostanzialmente una CPU realizzata in un unico circuito integrato

Potenza di un mP (parametri indicativi):

- parallelismo dei dati : numero di bit dei dati che possono essere trattati dal mP : ad es. un mP ad 8 bit può eseguire in modo diretto solo operazioni su dati numerici di 8 bit
- dimensione della memoria che il mP può indirizzare
- velocità con cui vengono eseguite le istruzioni proporzionale alla massima frequenza del segnale di clock

Struttura di un mP

- **ALU** : esegue le operazioni aritmetico – logiche

- **REGISTRI**
  - PC (program counter) : contiene l'indirizzo della prima istruzione che deve essere eseguita
  - IR (instruction register) : contiene l'istruzione da eseguire
  - MAR (memory address register) indirizzo dell'istruzione/dato che deve essere prelevato dalla memoria
  - MDR (memory data register) :: per lo scambio di dati tara mP e memoria o dispositivi di I/O
  - SR (status register) o registro dei flag; flag comuni a tutti i mP :
    - S (segno) segno dell'ultima operazione eseguita
    - Z (sero) risultato dell'ultima operazione eseguita e 0

- C : riporto
- H (half carry) : riporto generato dal 4 bit (utile quando si opear in BCD)
- P (parity) : parità (numero degli 1) del risultato
- O (overflow) : superamento della capacità di un registro a rappresentare correttamente numeri relativi in complemento a 2 : operandi con lo stesso segno producono un risultato di segno diverso
- SP (stack pointer) indica la prima locazione libera dello STACK (lo STACK è una speciale area di memoria in cui possono essere salvati e recupearti dati in ordine LIFO.

- **UC** (unit control)
  Contiene il decodificatore delle istruzioni ed i circuiti di temporizzazione che generano segnali di comando e sincronizzazione sia per il mP che per la memoria e gli altri dispositivi.
  Tutte le operazioni dell'unità di controllo sono regolate e sincronizzate da un clock interno o esterno.
  La UC
  - preleva l'istruzione (fetch)
  - decodifica (decode) : ad ogni istruzione che il mP è in grado di eseguire corrisponde una serie di microistruzioni cioè una sequenza di configurazioni binarie che costituiscono i segnali di attivazione per i vari elementi del mP e e dei dispositivi esterni
  - esegue(execute)

Temporizzazione
I mP sono macchine sequenziali sincrone : tutte le azioni che vengono intraprese sono sincronizzate da un segnale di cadenza : i vari segnali vengono attivati in corrispondenza di determinate transizioni e livelli di clock.
Il ciclo di clock (detto anche stato macchina) rappresenta l'intervallo di tempo durante il quale viene eseguito il più piccolo passo di elaborazione ossia una microoperazione.

esecuzione di un'istruzione : ciclo istruzione

ciclo macchina (CM) : fetch dell'istruzione
lettura o scrittura di dati in memoria
lettura o scrittura di dati su periferica
operazione interna alla CPU

ciclo istruzione = + cicli macchina = + cicli di clock

Il microprocessore 8086
La famiglia 80x86 nasce nel 1981 con il mP 8086 successore del mP 8080 che aveva un parallelismo a 8 bit e indirizzamento della memoria a 16 bit (memoria indirizzabile 64 K).

Segmentazione
L' 8086 ha un parallelismo a 16 bit e indirizzamento a 20 bit (memoria indirizzabile 1 M).
La memoria viene suddivisa in segmenti; l'indirizzo effettivo si ottiene aggiungendo 4 bit a destra dell'indirizzo di segment e sommando l'indirizzo di offset (cfr. paragrafo 5.2)

Stack
Nell' 8086 lo stack utilizza la memoria centrale : l'indirizzo di inizio è individuato dal valore caricato nel registro di segmento SS con offset SP. Ogni volta che si effettua una scrittura (PUSH) si decrementa di 2 SP e poi si scrive all'indirizzo SS:SP;
si legge (POP) all'indirizzo SS:SP e poi si incrementa SP di 2

Registri
I registri dell'8086 sono tutti a 16 bit; si possono classificare come :

- Registri generali : 4 registri general purpose : AX (accumulatore), BX, CX, DX in questi registri è utilizzabile separatamente la parte bassa (AL, BL, CL, DL) e la parte alta (AH, BH, CH, DH)

- Registri puntatori : SI, DI, SP, BP : utilizzabili come general purpose ma più specificatamente contengono la parte offset di operandi . in particolare SP è lo stack pointer

- Registri di segmento : CS, DS, ES, SS : utilizzati (possono essere inizializzati direttamente dal programmatore) per contenere l'indirizzo dei vari segmenti di memoria

- Registro di stato (PSW : program Status Word)

- Program counter (IP) : l'indirizzo della prossima istruzione da eseguire è contenuto nella coppia CS:IP

CAPITOLO 5

INTRODUZIONE AL LINGUAGGIO ASSEMBLY 8086

5.1 Convenzioni

- 1 statement per riga;
- uno statement può iniziare ovunque nella riga;
- si possono usare indifferentemente lettere maiuscole e lettere minuscole;
- i commenti devono essere preceduti dal carattere ";" ;
- i nomi possono essere composti utilizzando lettere, cifre, e i simboli ?, @, _, $;il primo carattere non può essere un numero, ed è sconsigliato l'uso del simbolo @; sono riconosciuti solo i primi 31 caratteri;
- numeri : si possono usare numeri decimali, esadecimali, binari;
  i numeri esadecimali devono essere seguiti da un "h" o "H" e se iniziano con una lettera (A-F) devono essere preceduti da 0;
  i numeri binari devono essere seguiti da "b" o "B".

5.2 Indirizzamento della memoria da parte dell'8086

L'8086 usa un bus indirizzi a 20 bit che gli consente di indirizzare 2^{20} (1 MB = 1.048.576 byte) locazioni di memoria.

Dato che i registri interni sono a 16 bit viene utilizzato il seguente meccanismo per costruire gli indirizzi a 20 bit a partire da valori a 16 bit:

la memoria viene suddivsa in segmenti di 64 KB (2^{16} = 65.536 byte), ciascuno dei quali è individuato da un indirizzo di partenza detto indirizzo di segmento (segment address) le cui ultime 4 cifre binarie sono pari a 0.

Per individuare una locazione di memoria all'interno di un segmento è sufficiente usare un indirizzo di spiazzamento (offset address) che rappresenta la posizione del byte all'interno del segmento.

Pertanto disponendo dei 16 bit più significativi (gli ultimi 4 sono certamente a 0) dell'indirizzo di segmento e conoscendo lo spiazzamento è possibile costruire l'indirizzo reale.

Per rappresentare un indirizzo completo si usa la notazione

indirizzo di segmento : indirizzo di offset

L'indirizzo effettivo verrà calcolato come

indirizzo di segmento * 10h + indirizzo di offset.

per esempio se CS =1AB3h e IP = 051Fh l'indirizzo effettivo EA (Effective Address) viene così calcolato

(1AB3h * 10h) + 0051Fh = 1B04Fh

In ogni momento sono attivi in memoria 4 segmenti cui sono associati i 4 registri di segmento CS, SS, DS, ES.

5.3 Metodi di indirizzamento

Si richiama il **formato di un'istruzione assembly**

etichetta codice operativo operandi commento

gli operandi possono essere distinti in sorgente e destinazione pertanto il formato può essere anche visto come

etichetta codice operativo destinazione, sorgente commento

Sotto la voce metodi di indirizzamento si raggruppano tutte le tecniche utilizzate per definire la sorgente e la destinazione (operandi) coinvolte in una istruzione.

- Indirizzamento immediato
l'operando sorgente è una costante

```
MOV BL, 0AH          ;la costante 0A hex viene copiata nel registro BL
```

- Indirizzamento diretto o assoluto
viene fornito l'indirizzo del dato o attraverso un valore costante racchiuso tra [ ] (le [ ] sono necessarie perché si possa distinguere da un indirizzamento immediato) o attraverso il nome simbolico di una variabile (come nei linguaggi di programmazione ad alto livello : dalla tavola dei simboli costruita dall'assembler si risale all'indirizzo della variabile).

```
MOV   [1FFEH],AX      ;carica il valore di AL nella locazione 1FFEH e
                      ;quello di AH nella locazione 1FFFH

MOV  AL,[152H]        ;carica in AL il contenuto della locazione di
                      ;memoria di indirizzo 152H
```

- Indirizzamento da registro
l'operando o gli operandi sono dei registri del mP.

```
MOV BX,CX             ;copia nel registro BX il contenuto di CX.
```

- Indirizzamento Indiretto tramite registro
l' indirizzo del dato è contenuto in uno dei registri del mP; il nome del registro deve essere racchiuso tra [ ] per evitare di confonderlo con l'indirizzamento da registro.
SP,BP,BX,SI,DI : possono essere usati nell'indirizzamento indiretto;
non è necessario specificare il segmento perché il mP ne assume uno di default, in particolare :

- per SP,BP lo stack segment
-per BX,SI,DI il data segment (nella manipolazione di stringhe per DI viene assunto come segmento di default l'extra segment).

```
MOV [BX],AL          ;il contenuto di AL viene caricato nella locazione
                     ;di memoria il cui indirizzo è contenuto in BX.
```

- Indirizzamento indiretto tramite registro con spiazzamento
al registro puntatore viene sommato uno spiazzamento costante a 8/16 bit con segno.

```
MOV  [BX+03H],AL     ;il valore contenuto in AL viene caricato alla
                     ;locazione di memoria di indirizzo [BX+03H].
```

- Indirizzamento indicizzato
l'indirizzo dell'operando è dato dalla somma del valore contenuto in un registro base (BX o BP) con quello contenuto in un registro indice (DI o SI).

```
MOV [BX+DI],AL       ;carica il valore contenuto in AL nella locazione
                     ;di memoria di indirizzo [BX+DI].
```

Alla somma dei due registri puntatori è possibile aggiungere uno spiazzamento costante con segno a 8/16 bit.

```
MOV [BX+DI+03H],AL
```

- Override (indirizzamento con cambio di segmento)
Nei metodi di indirizzamento esposti si è supposto che il segmento di riferimento fosse il data segment ed in alcuni casi lo stack segment. quando è necessario fare riferimento ad un dato, posto in un segmento diverso, bisogna far precedere l'indirizzo del dato dal nome del registro di segmento in cui il dato si trova (Segment Override Prefix)

```
MOV AL,ES:[BX+DI+03H]  ;in questo caso si fa riferimento all'extra
                       ;segment anziché al data segment.
```

5.4 Creazione ed esecuzione di un programma (programma e listati)

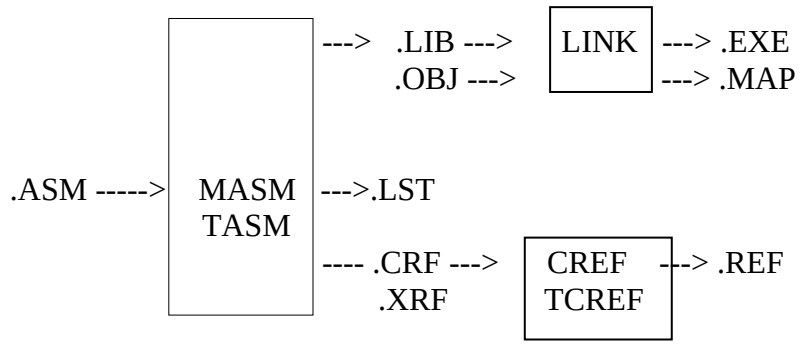
I passi da seguire per scrivere e far eseguire un programma in linguaggio assembly sono :
- scrittura, tramite un editor, del programma sorgente (estensione .ASM)
- generazione, tramite assembler, del programma oggetto (estensione .OBJ)
- generazione, tramite linker, del programma eseguibile (estensione .EXE)
- richiamo, da DOS, del programma eseguibile

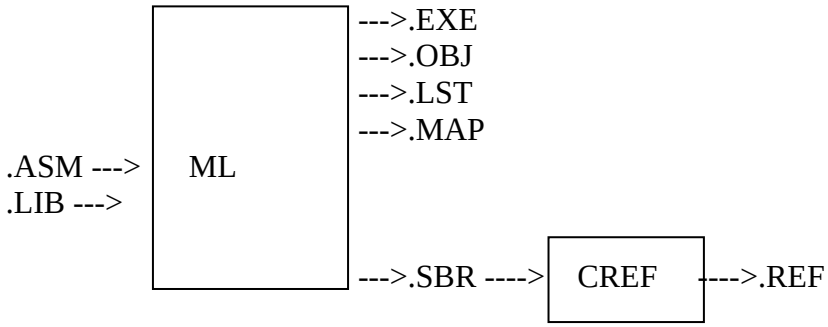
Vengono di seguito elencate le estensioni dei file utilizzati nel processo di creazione di un programma, e gli input ed output che si hanno nei vari passaggi. Si fa uso delle seguenti sigle :

- MASM per assembler Microsoft
- TASM per assembler Borland
- ML per assembler Microsoft versione 6.0 e successive

e, per i file, sono utilizzate le seguenti estensioni :

- ASM programma sorgente
- CRF informazioni per il Cross-Reference (MASM)
- EXE programma eseguibile
- LIB programma di libreria
- LST listato
- OBJ modulo oggetto
- MAP mappa
- REF report del Cross-Reference
- SBR informazioni per il Cross-reference (ML)
- XRF informazioni per il Cross-reference (TASM)





5.5 Struttura di un programma in linguaggio assembly 8086

Un programma è generalmente formato dalle seguenti sezioni
- intestazione : normalmente si tratta di una serie di commenti in cui viene indicato il nome del programma, l'autore, la data di creazione, lo scopo, l'input, l'output, etc...

segmento dati : dichiarazione ed inizializzazione delle variabili

```
DSEG    SEGMENT
        .........
        .........
DSEG    ENDS
```

- **segmento di stack :** viene definita ed eventualmente inizializzata l'area di stack

```
SSEG    SEGMENT  STACK
        .........
        .........
SSEG    ENDS
```

- **segmento extra** : eventuale area supplementare per i dati

```
ESEG    SEGMENT
        .........
        .........
ESEG    ENDS
```

- **segmento di codice**

```
CSEG    SEGMENT
        ..........
        ..........
CSEG    ENDS
```

- **chiusura del programma** : direttiva END in cui viene specificato da quale istruzione deve partire l'esecuzione del programma

```
        END etichetta di partenza
```

5.6 Definizione dei dati

5.6.1 Le direttive DB, DW, DD, DQ, DT

In un programma occorre specificare i dati utilizzati; per essi bisogna riservare un'opportuna area di memoria.

Si possono dati
- costanti : mantengono lo stesso valore per tutta la durata del programma
- variabili : il valore può variare nel corso del programma

La distinzione tra costante e variabile per l'assembler è fittizia in quanto anche una costante puòessere variata; normalmente si chiamano costanti le variabili inizializzate al momento della dichiarazione.

Si possono avere, per i dati, i seguenti raggruppamenti in byte :

BYTE	1 byte	8 bit	byte
WORD	2 byte	16 bit	word
DWORD	4 byte	32 bit	doubleword
QWORD	8 byte	64 bit	quadword
TBYTE	10 byte	80 bit	tenbyte

cui corrispondono le seguenti direttive per la definizione dei dati :

DB	Define Byte
DW	Define Word
DD	Define Dword
DQ	Define Qword
DT	Define Tbyte

Nella definizione dei dati il nome del dato, scelto dal programmatore, rispettando le convezioni del linguaggio, va messo nel campo etichetta, nel campo codice operativo l'opportuna direttiva, nel campo operandi l'eventuale valore di inizializzazione.

Esempi

```
TOTALE    DB      ?       ;variabile di nome TOTALE non inizializzata (?)
                          ;di un BYTE

ZERO      DW      0       ;variabile di nome ZERO di 1 WORD inizializzata
                          ;con il valore 0

LISTA     DD    ?,?,?,?,?
                          ;variabile di 5 DWORD non inizializzati

TABELLA   DB    15,100,3,4,7,8,99,32,87,98
                ;variabile di 10 BYTE inizializzati ai
                          ;rispettivi valori elencati

STAR      DB      "*"     ;oppure
STAR      DB      '*'     ;variabile di 1 BYTE inizializzata con il
                          ;carattere "*"

TRATTINO  DB  35  DUP("-")
                          ;variabile di 35 BYTE inizializzati  tutti con il
                          ;carattere "-"

WLISTA    DW    200 DUP(?)
                          ;variabile di 200 WORD non inizializzata
```

Si possono avere anche dichiarazioni del tipo

```
WLISTA      DW    1,2,3,10 DUP(0),99,100
                                ;lista di WORD inizializzati con i valori  1,2,3
                                ;10 volte 0, 99, 100

WTABELLA  DW    10 DUP(5 DUP(0),6,7,8,9,10)
                ;tabella di 100 numeri di tipo WORD : sequenza
                                ;0,0,0,0,0,6,7,8,9,10 ripetuta per 10 volte.
```

5.6.2 Uso delle direttive Define... per la dichiarazione di variabili numeriche

- rango dei numeri rappresentabili

direttiva	numeri con segno	numeri senza segno o floating point
DB	-128 +127	0 255
DW	-32.768 +32.767	0 65.535
DD	-2.147.483.648 +2.147.483.647	10^{-38} a 10^{+38} positivi e negativi floating point
DQ	-922337203685477808 +922337203685477807	10^{-308} a 10^{+308} positivi e negativi floating point
DT		10^{-4932} a 10^{+4932} positivi e negativi floating point

oppure numeri nella forma decimal-packed: 2 cifre per ogni byte e il byte più significativo per il segno.

Il processore può fare operazioni aritmetiche solo con byte e word, pertanto se non si ha il coprocessore matematico non si possono utilizzare le direttive DD,DQ,DT per memorizzare valori numerici.

5.6.3 Uso delle direttive Define per la dichiarazione di variabili di tipo carattere

a) DB definisce dati di 1 byte: in questo byte può essere memorizzato sia un carattere che un numero.
Esempi

```
UNO     DB    ?                   ;UNO variabile di un byte
QUAT    DB    ?,?,?,?             ;QUAT variabile di 4 byte
MOLT    DB    200 DUP(?)          ;MOLT variabile di 200 byte
STAR    DB    "*"                 ;STAR costante "*"
NOME    DB    "R","o","b","i","n"    ;oppure
```

```
NOME   DB    "Robin"              ;NOME costante "Robin"
```

b) DW definisce dati di 2 byte: può memorizzare 2 caratteri; nel caso venga assegnato un solo carattere esso viene memorizzato nel byte meno significativo e il byte più significativo viene messo a 00H.

c) DD,DQ,DT : non sono normalmente usati per memorizzare caratteri; comunque in essi possono essere memorizzati al massimo 2 caratteri nei primi 2 byte (più significativi); i byte restanti vengono messi a 00H.

5.6.4 Uso delle direttive Define.. per memorizzare indirizzi

a) DB : non può essere utilizzata in quanto permette di dichiarare variabili di un solo byte

b) DW : una variabile dichiarata DW può contenere un offset (16 bit)

Es.

```
LISTA      DB   100 DUP(?)        ;LISTA variabile di 100 BYTE
LOFFSET  DW  LISTA                ;LOFFSET costante che contiene
                                  ;l'offset del I byte di LISTA

Se DISPLAY è una procedura
OFFSET    DW  DISPLAY             ;la costante OFFSET contiene
                                  ;l'offset di DISPLAY
```

c) DD : una variabile dichiarata DD può contenere un indirizzo completo (indirizzo di segmento + offset);

per la variabile LISTA e la procedura DISPLAY dell'esempio precedente, l'indirizzo completo si può memorizzare

```
LADDR    DD  LISTA
DADDR    DD  DISPLAY
```

L'offset viene memorizzato nella metà di sinistra (più significativa) mentre l'indirizzo di segmento viene memorizzato nella metà di destra (meno significativa).

d)DQ e DT non vengono utilizzate per dichiarare variabili che devono contenere indirizzi.

5.7 Attributi

Un attributo descrive una caratteristica particolare di un dato.

Un dato ha 5 attributi:

1)TYPE (tipo) 1 per i byte
2 per i word
4 per i doubleword
8 per i quadword

e cosi via

2)LENGTH (lunghezza) : numero totale di gruppi di byte riservati per il dato
3)SIZE (grandezza) :numero totale di byte riservati per il dato (= lunghezza * tipo)
4)SEG : indirizzo di segmento del dato
5)OFFSET : indirizzo di offset del I byte del dato

I termini TYPE,LENGTH,SIZE,SEG,OFFSET sono degli operatori
Esempi :

```
LIST    DW    100 DUP(?)
MOV AX,TYPE   LIST        equivale a    MOV AX,2
MOV AX,LENGTH LIST        equivale a    MOV AX,100
MOV AX,SIZE   LIST        equivale a    MOV AX,200
MOV AX,SEG    LIST        equivale a    MOV AX, indirizzo di
                                              segmento di LIST
MOV AX,OFFSET LIST        equivale a    MOV AX, indirizzo di
                                              offset di LIST
```

5.8 L'operatore PTR (puntatore)

L'operatore PTR permette "l'override" del tipo di una variabile.
Esempi:

1) TOTAL DW ?

L'istruzione

```
MOV AX,TOTAL
```

copia i 2 byte di TOTAL in AX.
Se volessimo prendere separatamente ciascun byte di TOTAL, utilizzando l'operatore PTR, potremmo scrivere :

```
MOV BH,BYTE PTR TOTAL
MOV CH,BYTE PTR TOTAL+1
```

2) DUE DB ?,? ;DUE variabile di 2 BYTE

ma

```
MOV AX,DUE
```

è errata perché AX è di 16 bit mentre DUE non è una sola variabile di 16 bit bensì un insieme di 2 variabili ciascuna di 1 byte; utilizzando PTR si può effettuare il trasferimento nel seguente modo :

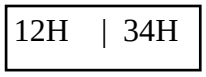
```
MOV AX, WORD PTR DUE
```

ATTENZIONE

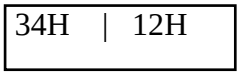
```
X   DB   "AB"    ;in X troviamo    4142H
Y   DW   "AB"    ;in Y troviamo    4241H
```

Cioè in una word i byte sono memorizzati in ordine inverso; il dato meno significativo viene memorizzato all'indirizzo più basso.

Esempio:
supponiamo di voler memorizzare il valore 1234H a partire dalla locazione 500H; il risultato non è



bensì



Normalmente non ci si deve preoccupare di questo; è importante però tenerlo presente quando si esamina la memoria con un debugger o quando viene forzata a WORD, tramite PTR, una coppia di byte: i dati contenuti in DUE, dell'esempio precedente, sono trasferiti in AX in ordine inverso; per averli nello stesso ordine si devono usare le 2 istruzioni :

```
MOVE AH,DUE
MOVE AL,DUE+1
```

5.9 Direttiva LABEL

La direttiva LABEL permette di definire un nome con attributi specifici.
Il formato è il seguente :

```
nome    LABEL   tipo
```

Il nome non occupa spazio nel programma in linguaggio macchina, ma consente di riferirsi ad una particolare locazione del programma.

```
BNAME    LABEL  BYTE
```

```
WNAME    DW      100 DUP (?)
```

La prima istruzione definisce un'etichetta BNAME che punta ad un insieme di dati costituiti da una sequenza di byte. La seconda istruzione definisce un insieme di dati WNAME, costituito da una sequenza di word.
In questo modo è possibile usare BNAME per accedere ai dati come bytes e WNAME per accedere ai dati come word; l'insieme di dati viene così ad avere due nomi diversi.

5.10 La direttiva EQU (equate)

La direttiva EQU permette di associare a un nome un valore : essa non genera un codice macchina né definisce uno spazio in memoria, è soltanto un'abbreviazione : l'assembler sostituisce nel programma al nome il valore associato nella direttiva EQU.

Esempio :

```
K          EQU  1000
ITEM1    DB     K  DUP(?)
```

è come dire all'assembler

```
ITEM1    DB   1000 DUP(?)
```

Il formato della direttiva EQU, come si ricava dall'esempio, è:

```
nome  EQU  espressione
```

nome : scelto dal programmatore, rispettando le regole del linguaggio;
espressione : numero intero positivo compreso tra 0 e 65.535; si possono usare numeri decimali, esadecimali, binari. Alcuni assemblatori permettono anche valori non mumerici cioè sequenze di caratteri, quindi sono valide espressioni del tipo :

```
MESSAGGIO     EQU    "Ciao"
```

5.11 Il Location Counter (LC) : il simbolo $

Il simbolo $ permette di prelevare il valore attuale dell' LC :

```
CORRENTE   EQU   $
```

in CORRENTE troviamo il valore di LC al momento in cui CORRENTE viene esaminata dall'assembler.
$ può essere utile per ricavare il valore della lunghezza di una stringa :

```
MESS      DB   "Ciao"
L_MESS  EQU  $-MESS
```

la seconda dichiarazione sta ad indicare

L_MESS = Indirizzo attuale - Indirizzo di MESS

e quindi la lunghezza di MESS.

ESERCIZI

1) scambio di due dati: a) a 8 bit, b) a 16 bit
2) somma, differenza, prodotto di due numeri a 32 bit
3) quarta potenza di un numero naturale a 8 bit
4) acquisito da tastiera un numero intero n di una cifra si visualizzino sullo schermo n asterischi in orizzontale, in verticale, in diagonale.
5) convertire un valore binario nella corrispondente stringa esadecimale e visualizzare il risultato
6) acquisiti da tastiera due numeri interi di una cifra, sommarli e visualizzare il risultato
7) verificare se un numero dato è pari o dispari
8) acquisiti da tastiera due valori interi relativi, sommarli e visualizzare il risultato
9) convertire i caratteri alfabetici di una stringa da minuscolo in maiuscolo
10) acquisire due nomi, determinare quale dei due venga prima in ordine alfabetico
11) visualizzare la tavola pitagorica relativa ai primi n valori
12) acquisire da tastiera una stringa di caratteri, determinare la frequenza con cui appare un carattere dato
13) verificare la correttezza di un codice a 5 byte confrontandolo con quello di riferimento
14) ordinare un insieme di caratteri
15) ordinare un insieme di nomi
16) invertire l'ordine dei caratteri in una stringa
17) eliminare dallo stack il terzo elemento a partire dalla cima
18) programma che a scelta consenta di effettuare tra due numeri naturali una delle quattro operazioni elementari
19) procedure di libreria per la manipolazione di stringhe
  a) input/output
  b) confronto
  c) ricerca
  d) trasferimento (copia, estrai, concatena)
20) visualizzare i primi n elementi della successione di Fibonacci
21) sommare 2 matrici quadrate di numeri interi positivi
22) procedura Delay(millisecondi)
23) procedura ClrScr
24) aritmetica in BCD
25) data una matrice di caratteri determinare se una certa parola compare orizzontalmente o verticalmente.

CAPITOLO 6

ISTRUZIONI ASSEMBLY

Di seguito illustreremo brevemente le principali istruzioni del linguaggio assembly.

Gli esempi riportati suppongono che nel programma siano stati definiti nel Data Segment i seguenti dati:

```
Dati        SEGMENT

datob       DB    01
datow       DW    0102h
stringa1    DB    "CIAO          "
stringa2    DB    "ARRIVEDERCI"

Dati        ENDS
```

Per indicare lo stato dei flag sono stati utilizzati i seguenti simboli :

x : modificato secondo il risultato
u : indefinito
0 : resettato (messo a zero)
1 : settato (messo a uno)
spazio : non alterato

6.1 ISTRUZIONI DI TRASFERIMENTO.

6.1.1 Istruzioni MOV

Operano il trasferimento dei dati da una sorgente ad una destinazione.

Sintassi : MOV destinazione, sorgente
Funzione : destinazione ⟵ sorgente
Flag : non alterati

Destinazione	*Sorgente*
Registro	Dato immediato
Memoria	Dato immediato
Registro	Registro
Registro	Memoria
Memoria	Registro

Limitazioni:
- sorgente e destinazione devono avere la stessa dimensione (8 o 16 bit)
- il caricamento di un dato immediato non è ammesso per i registri di segmento; occorre caricare prima il dato immediato in AX e poi trasferirlo nel registro di segmento.
- i registri CS e IP non possono essere usati come destinazione
- non è ammesso il trasferimento diretto da memoria a memoria
- non è ammesso il trasferimento diretto da registro di segmento a registro di segmento

Esempi :

MOV BX,03B8h	Carica il valore 03B8h nel registro BX
MOV [0310h],03B8h	Carica il valore 03B8h nelle locazioni 0310h e 0311h
MOV BX,CX	Trasferisce il contenuto del registro CX in BX
MOV BX,[CX]	Trasferisce il contenuto delle locazioni [CX] e [CX+1] in BX
MOV [0310h],BX	Trasferisce il contenuto di BX nelle locazioni 0310h e 0311h

6.1.2 Istruzioni di PUSH e POP

Operano il trasferimento di dati nell'area di stack oppure dall'area di stack.
L'area di stack viene utilizzata dall'80x86 nel modo LIFO (Last In First OUT), a significare che i dati vengono accatastati uno sull'altro, cosicché essa cresce per indirizzi decrescenti di memoria.
Queste istruzioni effettuano sempre il trasferimento di 16 bit ed aggiornano automaticamente il registro SP, che punta sempre alla cima dello stack, decrementandolo di 2 nelle istruzioni di PUSH ed incrementandolo di 2 in quelle di POP.

Sintassi : PUSH sorgente
Funzione : stack ← sorgente
Flag : non alterati

L'istruzione di PUSH decrementa di 2 il contenuto del registro Stack Pointer (SP, punta sempre all'ultimo dato inserito nello stack) e poi deposita il valore della sorgente.
La sorgente può essere :
. registro
. memoria
. registro dei flag

Esempi :

PUSH BX	Trasferisce nello stack il contenuto del registro BX
PUSH [CX]	Trasferisce nello stack il contenuto delle locazioni [CX] e [CX+1]

PUSH F Trasferisce nello stack il contenuto del registro dei flag

Sintassi : POP destinazione
Funzione : stack ⟵ destinazione
Flag : non alterati

L'istruzione di POP preleva dallo stack una word e la deposita nella destinazione; quindi incrementa di 2 il contenuto del registro Stack Pointer (SP, punta sempre all'ultimo dato inserito nello stack).

La sorgente può essere :
. registro
. memoria
. registro dei flag

Esempi :

POP BX Ripristina dallo stack il contenuto del registro BX
POP [CX] Ripristina dallo stack il contenuto delle locazioni [CX] e [CX+1]
POP F Ripristina dallo stack il contenuto del registro dei flag

6..1.3. Istruzioni di scambio.

Sintassi : XCHG destinazione, sorgente
Funzione : destinazione ⟷ sorgente
Flag : non alterati

Lo scambio può avvenire tra :
. registri
. registro e memoria

Limitazioni:
. gli operandi devono avere entrambi la stessa dimensione (8 o 16 bit)
. non è ammesso lo scambio con registri di segmento
. non è ammesso lo scambio tra 2 locazioni di memoria

Esempi :

XCHG BX,CX Scambia il contenuto del registro BX con CX
XCHG BX,[CX] Scambia il contenuto del registro BX con quello della locazione [CX] e [CX+1]

L'istruzione XLAT trasferisce nell'accumulatore AL il contenuto della locazione di memoria il cui offset si ottiene sommando al contenuto del registro BX quello di AL.

Sintassi : XLAT

Funzione : AL ⟵ [BX+AL]
Flag : non alterati

In questo caso il registro BX viene utilizzato per indirizzare l'inizio di una tabella dati (massimo 256), ciascuno dei quali può essere trasferito in AL conoscendone la posizione nella tabella (specificata prima del trasferimento dal contenuto di AL che ha la funzione di indice).

L'istruzione LEA trasferisce in un registro, indicato nell'istruzione, l'indirizzo di offset della locazione di memoria indicata come secondo operando.

Sintassi : LEA registro, memoria
Funzione : registro ⟵ offset[memoria]
Flag : non alterati

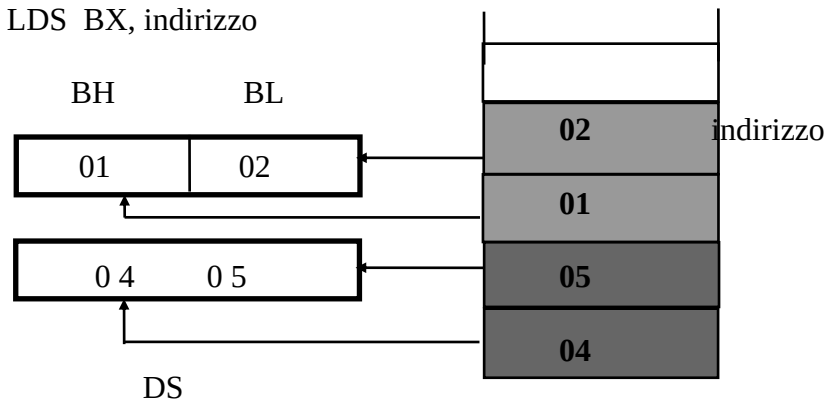
Esempio:

LEA BX, datob Carica in BX l'offset della locazione di memoria <datob>

Le istruzioni LDS (Load Data Segment) e LES (Load Extra Segment) operano il trasferimento di dati a 32 bit, denominati puntatori (segmento + offset). Il contenuto dei primi 2 byte di memoria a partire dall'indirizzo specificato nell'istruzione sono caricati nel registro indicato nell'istruzione, mentre il contenuto dei 2 byte successivi è memorizzato nel registro di segmento DS oppure ES.

Sintassi : LDS registro, memoria
Funzione : registro ⟵ [memoria]
DS ⟵ [memoria+2]
Flag : non alterati

Sintassi : LES registro, memoria
Funzione : registro ⟵ [memoria]
ES ⟵ [memoria+2]
Flag : non alterati


LDS BX, indirizzo
BH
BL
01
02
0 4
0 5
DS
02
indirizzo
01
05
04

6.2 ISTRUZIONI ARITMETICHE.

Le istruzioni di questo gruppo sono quelle che agiscono più frequentemente sui flag; la conoscenza dell'azione che queste esercitano su di essi è fondamentale per la programmazione.

Sintassi : INC destinazione
Funzione : destinazione ⟵ destinazione+1
Sintassi : DEC destinazione
Funzione : destinazione ⟵ destinazione-1
Flag :

O	D	I	T	S	Z	A	P	C
x				x	x	x	x	

Sintassi : NEG destinazione
Funzione : destinazione ⟵ complemento a 2 della destinazione
Flag :

O	D	I	T	S	Z	A	P	C
x				x	x	x	x	1

Tutte e tre le istruzioni possono agire su un operando costituito da:
. registro
. memoria

Esempi:

INC BX	Incrementa di uno il contenuto del registro BX
INC datobyte	Incrementa di uno il contenuto della locazione di memoria <datobyte>
NEG BX	Esegue il complemento a 2 del contenuto del registro BX

Sintassi : ADD destinazione, sorgente
Funzione : destinazione ⟵ destinazione + sorgente
Sintassi : ADC destinazione,sorgente
Funzione : destinazione ⟵ destinazione + sorgente + carry
Sintassi : SUB destinazione,sorgente
Funzione : destinazione ⟵ destinazione - sorgente
Sintassi : SBB destinazione,sorgente
Funzione : destinazione ⟵ destinazione – sorgente - carry
Sintassi : CMP destinazione,sorgente
Funzione : destinazione - sorgente, altera i flag a seconda del risultato, che tuttavia non viene posto nella destinazione (che rimane perciò inalterata) ma scartato.
Flag :

O	D	I	T	S	Z	A	P	C
x				x	x	x	x	x

Destinazione	*Sorgente*
Registro	Dato immediato
Memoria	Dato immediato
Registro	Registro
Registro	Memoria
Memoria	Registro

Esempi:

ADD BX,03B8h Somma il valore 03B8h al contenuto del registro BX
ADD datow,03B8h Somma il valore 03B8h al contenuto della word di memoria [datow]
ADD BX,CX Somma il contenuto del registro CX a quello di BX
ADD BX, datow Somma il contenuto della word di memoria [datow] a quello di BX
ADD datow, BX Somma il contenuto del registro BX alla word di memoria [datow]

L'istruzione di moltiplicazione ha due versioni, una per i numeri senza segno (MUL) e l'altra per quelli con segno (IMUL).
Nell'istruzione viene specificata la sorgente di uno dei due operandi, mentre l'altro è posto nell'accumulatore.

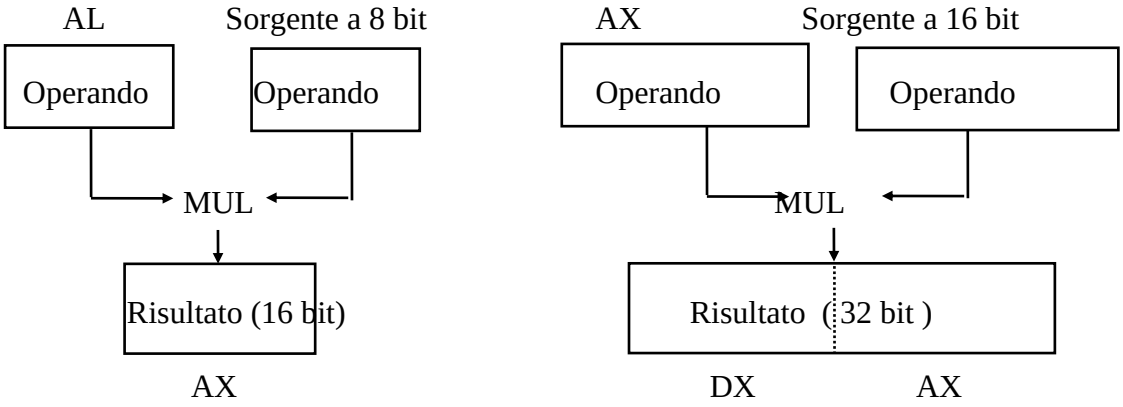
Sintassi : MUL sorgente
IMUL sorgente
Flag :

O	D	I	T	S	Z	A	P	C
x								x

La sorgente può essere rappresentata da:
. un registro
. un dato in memoria

L'istruzione di moltiplicazione può operare su numeri a 8 bit oppure su numeri a 16 bit.

La figura mostra il diverso modo di operare.



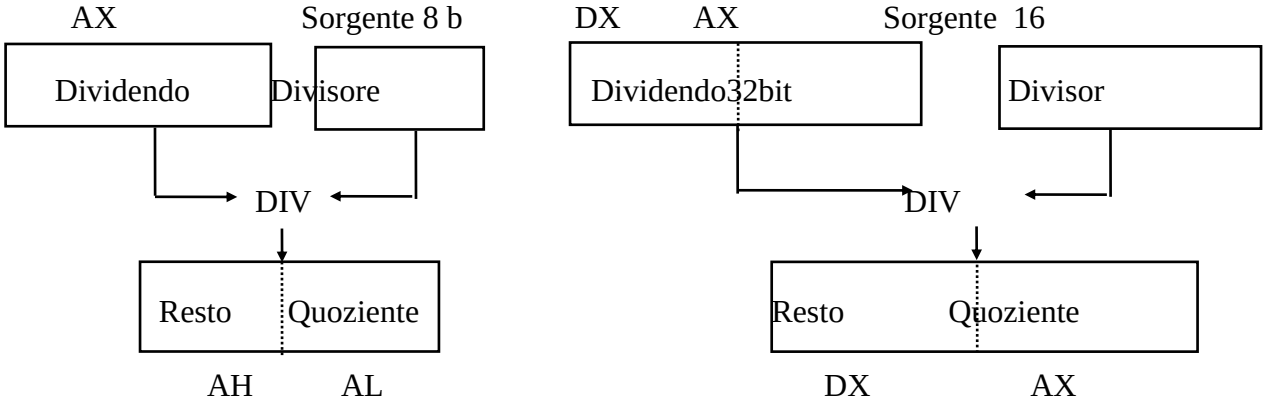
Moltiplicazione a 8 bit Moltiplicazione a 16 bit

L'istruzione di divisione ha anch'essa due versioni, una per i numeri senza segno (DIV) e l'altra per quelli con segno (IDIV) .
Nell'istruzione viene specificata la sorgente del divisore, mentre il dividendo è posto nell'accumulatore.

Sintassi : DIV sorgente
IDIV sorgente
Flag :

O	D	I	T	S	Z	A	P	C
u				u	u	u	u	u

La sorgente può essere rappresentata da:
. un registro
. un dato in memoria
L'istruzione di divisione può operare su numeri a 8 bit oppure su numeri a 16 bit.
La figura mostra il diverso modo di operare.



Divisione a 8 bit Divisione a 16 bit

Le istruzioni CBW e CWD convertono rispettivamente un byte posto in AL in una word che occupa tutto il registro AX e una word posta in AX in una double word che occupa DX e AX, entrambe con estensione del segno.

Estendere il segno significa mettere tutti 0 nella parte che viene estesa se il numero è positivo, tutti 1 se è negativo.
Queste istruzioni sono particolarmente utili quando il formato degli operandi (8, 16, 32 bit) nelle istruzioni di IMUL ed IDIV non è conforme con lo standard richiesto. Ad esempio se divisore e dividendo sono entrambi di un byte, occorre estendere il dividendo ad una word in tutto l'accumulatore AX, ovviamente estendendone il segno.

Sintassi : CBW
CDW
Flag : non alterati

Le istruzioni DAA e DAS operano l'aggiustamento decimale dell'accumulatore AL, rispettivamente dopo una somma o dopo una sottrazione.
Per ottenere un risultato decimale corretto si dovrà operare la somma o la sottrazione a partire da dati decimali codificati in binario ed aggiustare il risultato dopo ogni operazione.

Sintassi : DAA
DAS
Flag :

O	D	I	T	S	Z	A	P	C
u				x	x	x	x	x

Esempio:

```
MOV  AL,15d
MOV  CL,48d
ADD  AL,CL
DAA
```

5D
63

dopo l'istruzione ADD
dopo l'istruzione DAA

6.3 ISTRUZIONI LOGICHE.

Eseguono le operazioni logiche bit per bit tra destinazione e sorgente, ponendo il risultato nella destinazione.

Sintassi : AND destinazione, sorgente
Funzione : destinazione ⟵ destinazione AND sorgente
Sintassi : OR destinazione, sorgente
Funzione : destinazione ⟵ destinazione OR sorgente
Sintassi : XOR destinazione, sorgente
Funzione : destinazione ⟵ destinazione XOR sorgente
Sintassi : TEST destinazione, sorgente
Funzione : destinazione AND sorgente, altera i flag a seconda del risultato, che tuttavia non viene posto nella destinazione (che rimane perciò inalterata) ma scartato.
Flag :

O	D	I	T	S	Z	A	P	C
0				X	x	u	x	0

Destinazione	*Sorgente*
Registro	Dato immediato
Memoria	Dato immediato
Registro	Registro
Registro	Memoria
Memoria	Registro

Mediante l'uso di una opportuna maschera le istruzioni di OR, AND e TEST possono settare, resettare o testare i singoli bit di un registro o di una locazione di memoria. La maschera è un particolare byte o word che, con la sua configurazione di 1 e di 0, permette di operare sui bit che interessano e di trascurare gli altri.

Esempi:

Supponiamo che il registro BL contenga il seguente byte :

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
1	0	1	1	0	1	0	0

a) se vuole azzerare i bit 4 e 5 si dovrà utilizzare la seguente maschera

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
1	1	0	0	1	1	1	1

il cui valore esadecimale è CF ed eseguire l'istruzione:
AND BL, 0CFh

b) se vuole settare il bit 0 si dovrà utilizzare la seguente maschera

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	0	0	0	0	0	1

il cui valore esadecimale è 01 ed eseguire l'istruzione:
AND BL,01h

c) se vuole testare il bit 5 si dovrà utilizzare la seguente maschera

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	1	0	0	0	0	0

il cui valore esadecimale è 20 ed eseguire l'istruzione:
TEST BL,20h
Se dopo l'istruzione il flag di Z è settato (uguale a 1) allora significa che il bit 5 è uguale a zero, altrimenti è uguale a 1.

L'istruzione NOT esegue la negazione di ogni bit dell'operando.

Sintassi : NOT destinazione
Funzione : nega ogni bit della destinazione
Flag : non alterati

La destinazione può essere rappresentata da:
. registro
. memoria

6.4 GRUPPO DI PRIMITIVE DI STRINGA.

Nella programmazione dell'80x86 prendono il nome di stringhe i blocchi di dati (byte o word) posti in memoria sequenzialmente.
Le istruzioni MOVSB e MOVSW operano il trasferimento di blocchi di dati (rispettivamente a 8 o 16 bit) da una zona all'altra della memoria. L'istruzione trasferisce un byte o una word da una sorgente in memoria il cui indirizzo è dato da DS (registro segmento) e SI (offset) ad una destinazione il cui indirizzo è dato da ES (registro segmento) e DI (offset).
Con l'override (cioè specificando il registro di segmento nell'istruzione) si può fare in modo che SI punti ad una stringa anche nell'Extra Segment (ES:SI), mentre DI non può mai essere usato per puntare ad una stringa contenuta nel Data Segment(DS).
Dopo il trasferimento sia SI che DI vengono incrementati o decrementati (di 1 se il trasferimento è ad un byte o di 2 se è relativo ad una word) in base allo stato del flag di direzione D, che dovrà essere opportunamente caricato con le istruzioni CLD (Flag D=0, incremento) o STD (flag D=1, decremento).

Sintassi : MOVSB
Funzione : [ES:DI] ⟵ [DS:DI]
se flag D=0 (incremento) DI ⟵DI+1
SI ⟵SI+1
se flag D=1 (decremento) DI ⟵ DI-1
SI ⟵SI-1
Flag : non alterati

Sintassi : MOVSW
Funzione : [ES:DI] ⟵ [DS:DI]
se flag D=0 (incremento) DI ⟵DI+2
SI ⟵SI+2
se flag D=1 (decremento) DI ⟵DI-2
SI ⟵SI-2
Flag : non alterati

Per ripetere lo spostamento su una serie di dati si può far precedere l'istruzione MOVSB o MOVSW dall'istruzione REP, la quale esegue l'iterazione del trasferimento per il numero di volte indicato da CX. Questo dovrà quindi essere inizialmente caricato con il numero di byte o word da trasferire e sarà automaticamente decrementato di 1 ad ogni ripetizione

Sintassi : REP

Le istruzioni CMPSB e CMPSW effettuano il confronto tra 2 blocchi oppure analizzano uno stringa alla ricerca di un particolare byte o word. L'istruzione opera il confronto tra una stringa in memoria il cui indirizzo è dato da DS (registro segmento) e SI (offset) ed un'altra il cui indirizzo è dato da ES (registro segmento) e DI (offset); l'operazione consiste praticamente nell'esecuzione di una sottrazione, il

cui risultato viene scartato dopo che sulla base di questo sono stati opportunamente modificati i flag (in particolare il flag di zero).
Dopo il trasferimento sia SI che DI vengono aggiornati con le stesse modalità previste per le istruzioni MOVS.

Sintassi : CMPSB
CMPSW
Flag :

O	D	I	T	S	Z	A	P	C
x				x	x	x	x	x

Per ripetere il confronto su una serie di dati si può far precedere l'istruzione CMPSB o CMPSW da una delle seguenti istruzioni:

Sintassi : REP
Funzione : ripete se CX>0

Sintassi : REPZ
REPE
Funzione : ripete il confronto se i dati sono uguali e CX>0; termina se i dati sono diversi o se non ci sono più dati da confrontare (CX=0)

Sintassi : REPNZ
REPNE
Funzione : ripete il confronto se i dati sono diversi e CX>0; termina se i dati sono uguali o se non ci sono più dati da confrontare (CX=0)

Esempio:

```
CLD                 ; azzera flag di direzione (incremento registri indice)
MOV  SI, stringa1   ; indirizzo iniziale sorgente dati
MOV  DI, stringa2   ; indirizzo iniziale dati da confrontare
MOV  CX, 000Bh      ; numero di dati (11 byte)
REPE                ; confronta e ripete se sorgente e dato da confrontare
CMPSB               ; sono uguali e se CX>0
```

Le istruzioni LODSB e LODSW trasferiscono il dato puntato dal registro indice SI nell'accumulatore, rispettivamente in AL (per il modo byte) e AX (per il modo word), aggiornando il registro SI in base al flag di direzione D.

Le istruzioni STOSB e STOSW operano in senso inverso, trasferendo il contenuto dell'accumulatore AL o AX nella destinazione puntata dal registro indice DI , aggiornando il registro DI in base al flag di direzione D.
Queste ultime istruzioni non alterano lo stato dei flag.

Le istruzioni SCASB e SCASW confrontano il dato puntato dal registro indice DI con l'accumulatore, rispettivamente con AL (per il modo byte) e AX (per il modo word), aggiornando il registro DI in base al flag di direzione D.

6.5 ISTRUZIONI DI CONTROLLO DEL CONTATORE DI PROGRAMMA.

6.5.1 Istruzioni di salto incondizionato.

Le istruzioni di salto incondizionato trasferiscono il controllo del contatore di programma ad un istruzione nel segmento di codice corrente (salto intrasegmento) oppure di un diverso segmento (salto intersegmento)
L'istruzione a cui trasferire il controllo è identificata da un etichetta (label); questa può essere formata da una sequenza di un massimo di 31 caratteri alfanumerici, di cui il primo alfabetico.
Una label identifica, quindi, all'interno del segmento codice l'indirizzo di una istruzione. Una label NEAR identifica un indirizzo a cui si può accedere solo dallo stesso segmento codice; ad una label FAR si può, invece, accedere da un segmento codice diverso.
È una buona regola di programmazione evitare i salti tra segmenti diversi.
Come si potrà facilmente intuire la maggior parte delle label di un programma sono di tipo NEAR; l'assemblatore consente di definirle in forma abbreviata :

label:
istruzioni

Esempio :

```
ripeti:
        --- istruzioni  ---
```

Le label FAR possono essere definite esplicitamente tramite la dichiarazione LABEL :

label LABEL FAR

Esempio :

```
cicla   LABEL       FAR
```

Salti intersegmento (ad una istruzione di un segmento diverso).

L'indirizzo di salto può essere specificato secondo le seguenti modalità:

a) direttamente nell'istruzione tramite l'indirizzo del segmento ed offset
Sintassi : JMP label (dichiarata con LABEL FAR)

Esempio:

```
cicla   LABEL       FAR
```

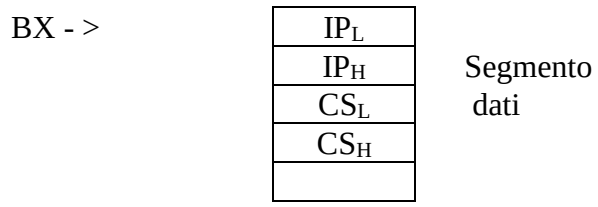
```
codice SEGMENT    CODE

        --- istruzioni ---

        JMP   cicla
```

b) è prelevato dalle locazioni di memoria indicate nell'istruzione.
In questo caso è necessario il suffisso FAR per indicare che si tratta di un salto intersegmento.
La memoria può essere indirizzata in uno qualsiasi dei modi conosciuti.

Sintassi : JMP FAR [memoria]

Esempio:
JMP FAR [BX]

In questo caso l'indirizzo di salto viene ricavato nel seguente modo:



Salto intrasegmento.
L'indirizzo di salto può essere specificato secondo le seguenti modalità:
a) indicando nell'istruzione una label che individui un'istruzione che appartenga allo stesso segmento

Sintassi : JMP label

Esempio:

```
repeat:
        --- istruzioni  ----
        JMP  repeat
```

b) È prelevato da un registro. Il valore contenuto nel registro viene caricato nel contatore di programma (IP).

Sintassi : JMP registro

Esempio:
JMP CX

c) È prelevato dalle locazioni di memoria indicate nell'istruzione.
In questo caso occorre precisare all'assemblatore che si tratta di un salto intrasegmento aggiungendo il suffisso NEAR.
I valori contenuti nella locazione di memoria indicata nell'istruzione ed in quella successiva sono caricati nel contatore di programma (IP).
La memoria può essere indirizzata in uno dei modi conosciuti.

Sintassi : JMP NEAR [memoria]

Esempio:
JMP NEAR [BX]

6.5.2. Istruzioni di salto condizionato.

Le istruzioni di salto condizionato eseguono un salto ad una istruzione del programma se è verificata la condizione in esse specificata. L'istruzione viene tradotta dall'assemblatore solo utilizzando l'indirizzamento diretto relativo con spiazzamento a 8 bit; la destinazione del salto non può quindi distare a livello di codice macchina più di 255 byte (127 byte avanti e 128 byte indietro rispetto al program counter dell'istruzione successiva; in caso contrario l'assemblatore genererà un errore).

Sintassi : Jcondizione label

Il set di istruzioni dell'8086 permette di esprimere non solo condizioni semplici legate allo stato di un solo flag, ma anche condizioni legate allo stato di più flag.

Condizioni sempli sui flag.

JE	salta se uguale	Z=1
JZ	salta se è zero	Z=1
JNE	salta se non è uguale	Z=0
JNZ	salta se non è zero	Z=0
JS	salta se il segno è negativo	S=1
JNS	salta se il segno è positivo	S=0
JP	salta se c'è parità	P=1
JPE	salta se la parità è pari	P=1
JNP	salta se non c'è parità	P=0
JPO	salta se la parità è dispari	P=0
JO	salta se c'è overflow	O=1
JNO	salta se non c'è overflow	O=0

Relazioni aritmetiche fra numeri senza segno.

JB	salta se inferiore	C=1

JNAE	salta se non è superiore o uguale	C=1
JC	salta se c'è riporto	C=1
JA	salta se è superiore	C=0 e Z=0
JNBE	salta se non è inferiore o uguale	C=0 e Z=0
JNB	salta se non è inferiore	C=0
JAE	salta se non è superiore o uguale	C=0
JNC	salta se non c'è riporto	C=0
JNA	salta se non è superiore	C=1 o Z=1
JBE	salta se è inferiore o uguale	C=1 o Z=1

Relazioni aritmetiche fra numeri con segno.

JL	salta se è minore	S=1
JNGE	salta se non è maggiore o uguale	S=1
JG	salta se maggiore	S=0 e Z=0
JNLE	salta se non è minore o uguale	S=0 e Z=0
JNL	salta se non è minore	S=0
JGE	salta se è maggiore o uguale	S=0
JNG	salta se non è maggiore	S=0 o Z=1
JLE	salta se è minore o uguale	S=0 o Z=1

Le istruzioni LOOP vengono utilizzate per iterare una sequenza di istruzioni, utilizzando come contatore delle iterazioni il registro CX, che pertanto deve essere opportunamente inizializzato.

Sintassi : LOOP label ; nota (*)
Funzione : salta se CX è diverso da zero

Sintassi : LOOPZ label
LOOPE label
Funzione : salta se CX è diverso da zero ed il flag Z=1 (dati confrontati uguali)

Sintassi : LOOPNZ label
LOOPNE label
Funzione : salta se CX è diverso da zero ed il flag Z=0 (dati confrontati diversi)

(*) 127 byte avanti e 128 byte indietro rispetto al program counter dell'istruzione successiva

L'istruzione JCXZ è una particolare istruzione di salto, condizionato dal valore del registro CX: il salto viene eseguito se CX=0. Può essere utilizzato all'inizio di un loop per evitare l'esecuzione qualora CX sia inizialmente nullo.

6.5.3. Istruzioni di CALL e RET.

Le istruzioni CALL sono utilizzate per la chiamata ad una subroutine ed utilizzano gli stessi metodi di indirizzamento visti per le istruzioni JMP, fatta eccezione per l'indirizzamento diretto relativo con spiazzamento ad 8 bit. Valgono le stesse regole viste per le istruzioni di salto per distinguere le chiamate intersegmento (FAR) da quelle intrasegmento (NEAR).

La struttura di una procedura è la seguente :

```
nomeproc    PROC        tipo

            --- istruzioni ---

nomeproc    ENDP
```

Esempio di procedura di tipo NEAR :

```
calcola     PROC        NEAR
            .....
            .....
calcola     ENDP
```

Esempio di procedura di tipo FAR :

```
calcola     PROC        FAR
            .....
            .....
calcola     ENDP
```

- chiamate intersegmento

Sintassi : CALL label
CALL FAR [memoria]

- chiamate intrasegmento

Sintassi : CALL label
CALL [memoria]

Il contenuto dell'Instruction Pointer (IP) ed eventualmente del Code Segment (CS) vengono modificati secondo i meccanismi visti per le istruzioni di salto.
Prima di modificare questi 2 registri il microprocessore provvede però a salvare nello stack l'indirizzo di ritorno dalla subroutine, costituito solo dall'offset se si

tratta di una chiamata intrasegmento (2 byte) o dall'indirizzo di segmento + offset se la chiamata è intersegmento (4byte).

L'istruzione di RET termina l'esecuzione di una subroutine e ritorna al programma chiamante, prelevando l'indirizzo di ritorno dallo stack.

Sintassi : RET

Se la procedura è di tipo FAR, il processore dovrà prelevare dallo stack 2 word (offset e l'indirizzo del segmento codice).
Se la procedura è di tipo NEAR, il processore dovrà prelevare dallo stack 1 sola word (offset).
Quando viene incontrata una istruzione di RET, l'assemblatore genera una traduzione in linguaggio macchina diversa a seconda del tipo di chiamata alla procedura; ciò avviene in automatico senza che il programmatore debba fornire ulteriori informazioni.

6.6 ISTRUZIONI DI INPUT/OUTPUT.

Le istruzioni di Input/Output consentono ad un sistema a microprocessore di collegarsi con il mondo esterno, attraverso dispositivi generalmente indicati come *porte di Input/Output*.
Le porte di Input/Output sono riconosciute dall'8086 attraverso indirizzi a 16 bit.
L'istruzione IN consente alla CPU di leggere un dato da una porta, l'istruzione OUT di scrivere un dato sulla porta.
I dati letti o scritti possono essere a 8 oppure a 16 bit (le porte a 16 bit occupano naturalmente 2 spazi di indirizzi).

Sintassi : IN accumulatore, indirizzo porta (8bit)
IN accumulatore, DX

Sintassi : OUT indirizzo porta (8 bit),accumulatore
OUT DX, accumulatore

Flag : non alterati

L'uso del registro DX consente di modificare l'indirizzo della porta di Input/Output durante l'esecuzione del programma.

6.7. ISTRUZIONI DI CONTROLLO.

6.7.1 Istruzioni di controllo.

Sintassi : HALT
Funzione : arresta la CPU fino a quando non viene attuata un'interruzione o viene attivata la linea di Reset.

Sintassi : NOP
Funzione : non produce alcuna azione; serve per occupare spazio nell'allocazione di un programma, se si prevede di dover aggiungere qualche cosa, oppure per introdurre piccoli ritardi.

Sintassi : CLC
Funzione : azzera il flag di carry (C)

Sintassi : STC
Funzione : setta il flag di carry (C)

Sintassi : CMC
Funzione : complementa il flag di carry (C)

6.7.2 Istruzioni per la gestione delle interruzioni.

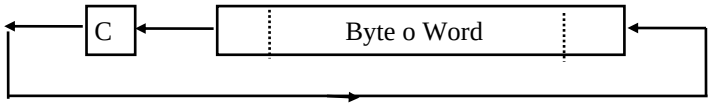
Le istruzioni per la gestione delle interruzioni saranno trattate in seguito in maniera approfondita in un altra parte del corso.

6.8 ISTRUZIONI DI ROTAZIONE E SHIFT.

Le istruzioni di rotazione e shift muovono, secondo diverse modalità, i bit all'interno di un byte o di una word, memorizzati in un registro oppure in memoria.
Nella rotazioni circolari il flag di carry (C) è interposto sul percorso di rotazione, mentre le rotazioni semplici e gli shift copiano in esso il bit uscente dal byte o dalla word.
Se il numero di movimenti da eseguire è 1 può essere precisato direttamente nell'istruzione, mentre in caso contrario (maggiore di 1) deve essere memorizzato nel registro CL.

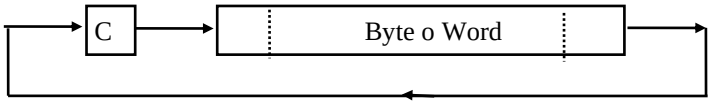
Sintassi : RCL registro, contatore
RCL [memoria],contatore
Funzione : rotazione circolare a sinistra
Flag :

O	D	I	T	S	Z	A	P	C
x								x



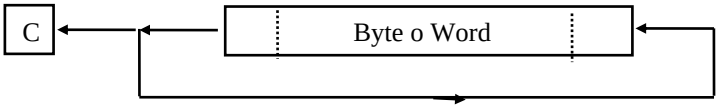
Sintassi : RCR registro, contatore
RCR [memoria],contatore
Funzione : rotazione circolare a destra
Flag :

O	D	I	T	S	Z	A	P	C
x				x	x	u	x	x



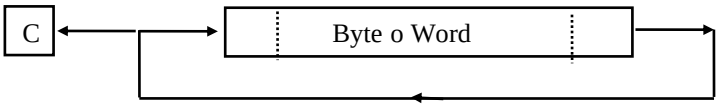
Sintassi : ROL registro, contatore
ROL [memoria],contatore
Funzione : rotazione a sinistra
Flag :

O	D	I	T	S	Z	A	P	C
x								x



Sintassi : ROR registro, contatore
ROR [memoria],contatore
Funzione : rotazione a destra
Flag :

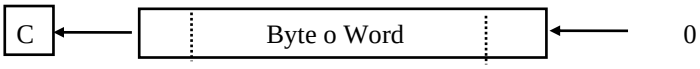
O	D	I	T	S	Z	A	P	C
x								x



Sintassi : SHL registro, contatore
SHL [memoria],contatore
Funzione : shift logico a sinistra
Sintassi : SAL registro, contatore
SAL [memoria],contatore
Funzione : shift aritmetico a sinistra

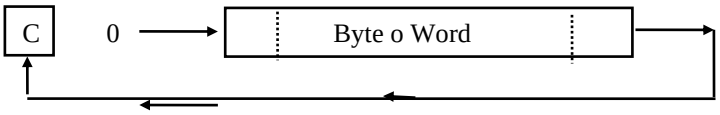
Flag :

O	D	I	T	S	Z	A	P	C
x				x	x	u	x	x



Sintassi : SHR registro, contatore
SHR [memoria],contatore
Funzione : shift logico a destra
Flag :

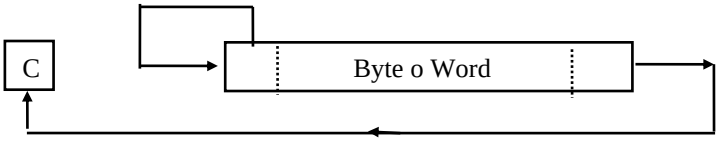
O	D	I	T	S	Z	A	P	C
x				x	x	u	x	x



Sintassi : SAR registro, contatore
SAR [memoria],contatore
Funzione : shift aritmetico a destra; notare che questa istruzione conserva il bit di segno (cioè il più 8significativo)

Flag :

O	D	I	T	S	Z	A	P	C
x				x	x	u	x	x



Esempi:

	C	AL	
SHL AL,1	X	1 0 1 1 0 1 0 0	prima
	1	0 1 1 0 1 0 0 0	dopo

	C	AL	
MOV CL,3 RCL BH,CL	X	1 0 0 1 1 0 1 0	prima
	0	1 1 0 1 0 X 1 0	dopo

APPENDICE A

1. L'Assembler Borland : generalità

L'assembler Borland (TASM) è un assembler veloce ad un passo : risolve tutti i riferimenti e genera tutto il codice macchina esaminando una sola volta il codice contenuto nel file sorgente.

I file più importanti sono
- TASM.EXE : l'assembler
- TLINK.EXE : il linker
- MAKE.EXE : programma di servizio per la compilazione
- TLIB.EXE : gestore di librerie
- TCPREF.EXE : programma per la generazione di tabelle di cross-reference.

Fasi per la creazione di un programma in linguaggio assembly:
1) Immettere il codice sorgente con un Editor di testi in formato ASCII; salvare in un file con estensione .asm.
2) Assemblare il codice sorgente con il programma TASM; viene prodotto un file con estensione .obj.
3) Eseguire il linking del file oggetto con il programma TLINK; viene prodotto un file con estensione .exe ed un file con estensione .map.

2. Opzioni dell'Assembler (TASM)

Per visualizzare l'elenco di tutte le opzioni di TASM digitare il comando

```
TASM
```

Tenendo presente che il comando TASM completo è

```
TASM[opzioni] sorgente[,oggetto][,listato][,cross-ref]
```

il comando, ad es.:

```
TASM /zi mioprog,,,
```

assembla il programma mioprog.asm, produce il file oggetto mioprog.obj, produce il file listato mioprog.lst ed il file cross-reference mioprog.xrf; inoltre il file mioprog.obj, grazie all'opzione /zi conterrà utili informazioni per il Debugger.

Il file di configurazione TASM.CFG consente di memorizzare, una volta per tutte alcune opzioni, che verranno automaticamente incluse nel comando TASM ogni volta che esso viene richiamato. Ad esempio si modifichi, con un Editor, il file TASM.CFG in modo che contenga le opzioni
```
/c /la /zi
```

e si assembli il file mioprog.asm con

TASM mioprog

in effetti è come se si fosse dato il comando

TASM /c /la /zi mioprog

Le opzioni indicate hanno il seguente significato :

/c : genera una tabella di riferimento (cross-reference) nel file .lst;
/la : generazione del listato esteso;
/zi : informazioni di Debug.

3. Opzioni del Linker (TLINK)

Il comando

TLINK

produce l'elenco di tutte le opzioni del Linker. Le più significative tra le varie opzioni sono indicate nel seguente esempio

TLINK /m /s /l /v mioprog,,,

/m : produrrà un file mappa contenente tutti i simboli pubblici;
/s : e con una mappa dettagliata dei segmenti del programma;
/l : includerà nel file sorgente i numeri di riga;
/v : darà informazioni di debug sui simboli;

4. Il file mappa (.MAP)

Viene normalmente creato dal Linker (TLINK). La mappa standard elenca
- i segmenti del programma;
- l'indirizzo iniziale;
- i messaggi di errore generati dal Linker.

Ulteriori informazioni si possono avere richiamando TLINK con le opzioni
/m (elenco ordinato dei simboli pubblici, se presenti nel programma) e
/s (mappa dettagliata dei segmenti)

5. Il file listato (.LST)

Viene generato dal TASM su richiesta dell'utente:
- tramite l'opzione /la (listato esteso) o /l (listato normale) del TASM;
- dal comando TASM[opzioni] sorgente[,oggetto][,listato][,cross-ref].

Il file listato (.lst) contiene:
- il codice sorgente ed il codice macchina equivalente in notazione esadecimale;
- informazioni sulla tabella dei simboli (symbol table); per ogni simbolo

viene indicato il tipo (text, number, byte, far, etc..) ed il valore
(numeri, nomi di variabile, etc...);
- una tabella di riferimenti (cross-reference : numero di riga dove appare un determinato simbolo) se specificata l'opzione /c nel comando TASM.

6. Il programma di servizio MAKE

Viene normalmente utilizzato in programmi di grandi dimensioni che richiedono l'assemblaggio o la compilazione di più file per produrre un unico file eseguibile. In questi casi la modifica del codice in un file sorgente richiederebbe la ricompilazione od il riassemblaggio di tutti i file; il programma MAKE evita questo inconveniente; esso opera basandosi sulla data ed ora di creazione/modifica di ogni file; riassembla o ricompila solo i file la cui data/ora non è congruente con gli altri file componenti il programma eseguibile.

Per utilizzare MAKE si deve creare, con un Editor, un file senza estensione cui si potreppe dare il nome del file .exe finale; ad es. partendo da un file "mioprog.asm" si vuol generare un file "mioprog.exe", allora si potrebbe creare il seguente file per MAKE di nome "mioprog"

```
mioprog.exe: mioprog.obj
    tlink /m /l mioprog,,,
mioprog.obj: mioprog.asm
    tasm /zi mioprog,,,
```

ed eseguire il programma

```
MAKE -fmioprog
```

Questo comando, partendo dal contenuto del file "mioprog", fa l'assemblaggio ed il linkaggio di "mioprog.asm" producendo "mioprog.exe", solo quando ciò è necessario (data ed ora di "mioprog.asm" successiva alla data ed ora di "mioprog.exe")

7. Il programma DEBUGGER

Gli errori di prgrammazione possono essere suddivisi in 2 categorie
- errori di sintassi;
- errori logici;

Mentre è molto semplice scoprire gli errori di sintassi, in quanto vengono segnalati dal compilatore o assemblatore, è molto più difficile scoprire gli errori logici in quanto la scoperta di questi errori è a carico esclusivo del programmatore. Un potente strumento che può aiutare il programmatore
nella scoperta di questi errori è il Debugger.

Il Debugger Borland può essere richiamato tramite uno dei seguenti comandi:
- TD
- TD nomeprogramma
- TW (se richiamato da Windows)

Esso presenta diversi Menù ed alcuni dei Menù, a loro volta hanno dei sottomenù.

Menù Desktop Manager (-)
- Repaint desktop : cancella il contenuto dello schermo;
- Restore standard : ripristina la composizione dello schermo iniziale del Debugger.

Menù File : gestione di file e directory.
Menù Edit : oltre alle scelte Copy e Paste presenta:
- Copy to log : copia temporanea degli elementi evidenziati;
- Dump pane to log : copia temporanea dell'intera finestra; con l'opzione Log del Menù View si può osservare il contenuto della finestra e confrontarlo con lo stato corrente dell'esecuzione passo-passo di Debug.

Menù View
- Breakpoints : consente di impostare dei punti di arresto;
- Watches : consente di osservare il cambiamento di valore durante l'esecuzione del programma delle variabili specificate;
- CPU, Dump, Registers, Numeric : consentono di esaminare il contenuto della CPU, della memoria, dei registri, dello stack del coprocessore (se installato).

Menù Run
- Run : esecuzione normale del programma;
- Go to cursor : esecuzione del programma fino al punto indicato dal cursore;
- Trace into : esecuzione del programma una riga alla volta;
- Step over : esecuzione del programma una riga alla volta saltando tutte le chiamate a procedure e funzioni;
- Execute to : esecuzione del programma sino all'indirizzo specificato;
- Program reset : carica nuovamente il programma.

Menù Breakpoints : attivazione/cancellazione di breakpoints.

Menù Data : esame/modifica dei dati del programma.

Menù Options : impostazioni opzioni per il Debugger.

Menù Window : gestione aspetto della finestra del Debugger.

Menù Help : richiamo dell'help.

8. Il programma PROFILER

È un programma che permette di analizzare le prestazioni di un programma, in termini di
- buchi neri : punti in cui il programma spreca una grande quantità di tempo;
- colli di bottiglia : punti in cui l'esecuzione del programma rallenta a causa di una cattiva progettazione dell'algoritmo.

Il programma può essere utilizzato per qualsiasi linguaggio che produce un file .map.

Per richiamarlo utilizzare uno dei comandi

TPROF
TPROF nomeprogramma

I menù più importanti sono View, Run, Statistics.

9. Esempi

Di seguito vengono riportati il sorgente di un programma d'esempio (scambia.asm) ed i relativi file .lst e .map.

SCAMBIA.ASM

```
; Scambio di 2 byte in memoria
dati    SEGMENT
        primo   DB 01h
        secondo DB 02h
dati    ENDS

sistema  SEGMENT STACK
          DW 100 DUP(?)
          top LABEL WORD
sistema  ENDS

codice   SEGMENT                                        ; definizione dei
segmenti
         ASSUME CS:codice, SS:sistema, DS:dati, ES:dati

 inizio:
       MOV AX,sistema                                   ; inizializzazione
reg.segmento
       MOV SS,AX
       LEA AX,top
       MOV SP,AX
       MOV AX,dati
       MOV DS,AX
       MOV ES,AX

       MOV AL,primo                                     ; programma
       MOV AH,secondo
       MOV primo,AH
       MOV secondo,AL

       MOV AL,00h                                       ; ritorno al sistema
operativo
       MOV AH,4Ch
       INT 21h
codice  ends
END     inizio
```

SCAMBIA.LST

```
Turbo Assembler  Version 3.1       04/12/95 12:46:43       Page 1
scambia.ASM

    1                   ;Scambio di 2 byte in memoria
    2 0000                dati    SEGMENT
```

```
 3 0000  01                  primo   DB 01h
 4 0001  02                  secondo DB 02h
 5 0002                dati  ENDS
 6
 7 0000                sistema  SEGMENT STACK
 8 0000  64*(????)              DW 100 DUP(?)
 9 00C8                         top LABEL WORD
10 00C8                sistema  ENDS
11
12 0000                codice   SEGMENT          ; definizione dei segmenti
13                        ASSUME CS:codice, SS:sistema, DS:dati, ES:dati
14
15 0000                 inizio:
16 0000  B8 0000s          MOV AX,sistema      ; inizializzazione reg.segmento
17 0003  8E D0             MOV SS,AX
18 0005  B8 00C8r          LEA AX,top
19 0008  8B E0             MOV SP,AX
20 000A  B8 0000s          MOV AX,dati
21 000D  8E D8             MOV DS,AX
22 000F  8E C0             MOV ES,AX
23
24 0011  A0 0000r          MOV AL,primo        ; programma
25 0014  8A 26 0001r       MOV AH,secondo
26 0018  88 26 0000r       MOV primo,AH
27 001C  A2 0001r          MOV secondo,AL
28
29 001F  B0 00             MOV AL,00h          ; ritorno al sistema operativo
30 0021  B4 4C             MOV AH,4Ch
31 0023  CD 21             INT 21h
32 0025                codice   ends
33                   END      inizio
```

```
Turbo Assembler        Version 3.1        04/12/95 12:46:43       Page 2
Symbol Table

Symbol Name          Type   Value                 Cref (defined at #)
??DATE               Text   "04/12/95"
??FILENAME           Text   "scambia         "
??TIME               Text   "12:46:41"
??VERSION            Number 030A
@CPU                 Text   0101H
@CURSEG              Text   CODICE                  #2  #7  #12
@FILENAME            Text   SCAMBIA
@WORDSIZE            Text   2                    #2  #7  #12
INIZIO               Near   CODICE:0000             #15  33
PRIMO                Byte   DATI:0000             #3  24  26
SECONDO              Byte   DATI:0001             #4  25  27
```

```
TOP                 Word   SISTEMA:00C8          #9  18

Groups & Segments      Bit Size Align  Combine Class      Cref (defined at #)

CODICE                 16  0025 Para   none            #12  13
DATI                   16  0002 Para   none            #2  13  13  20
SISTEMA                16  00C8 Para   Stack           #7  13  16
```

SCAMBIA.MAP

```
Start      Stop     Length     Name           Class

 00000H 00001H   00002H   DATI
 00010H 000D7H  000C8H  SISTEMA
 000E0H 00104H  00025H   CODICE

  Address        Publics by Name

  Address        Publics by Value

Program entry point at 000E:0000
```

10. Annotazioni sul File .lst

"r" : indirizzo rilocabile
"s" : segment
"Far" : procedura chiamata da un altro segmento
"Near" : procedura chiamata dal proprio segmento
@CURSEG : nome segmento corrente
@WORDSIZE : 2 per segmento a 16 bit; 4 per per segmento a 32 bit

Groups & Segment

Direttiva GROUP

```
nome_gruppo   GROUP  nome_segmento1, nome_segmento2,.....
```

riunisce in un unico blocco, non più grande di 64 KB, più segmenti; il vantaggio di questa direttiva sta nella possibilità d2i
- utilizzare il nome del gruppo nella direttiva ASSUME

Es.:

```
Cgroup   GROUP Cseg, Dseg
      ASSUME CS:Cgroup, DS:Cgroup
```

- fare l'override di un segmento e quindi stabilire degli indirizzamenti con riferimento alla base del gruppo.

Direttiva SEGMENT

```
nome_seg  SEGMENT  [allineamento (align)], [combine], ['class']
```

definisce l'inizio di un segmento di programma; la fine dello stesso è indicata da ENDS.

BIT : 16 fino a 64 K; 32 fino a 4 GB

SIZE : ampiezza

ALIGN : BYTE il segmento può iniziare a qualsiasi indirizzo
WORD il segmento inizia ad un indirizzo pari
PARA il segmento inizia ad un indirizzo multiplo di 16 (10h)
PAGE il segmento inizia ad un indirizzo multiplo di 256 (100h)
Per default è PARA.

COMBINE : indica al Link come riunire i segmenti di una particolare classe

NONE o PRIVATE : i segmenti sono tenuti logicamente separati; ciascuno ha il suo indirizzo di base. (default è NONE).

PUBLIC : Segmenti con lo stesso nome e la stessa classe sono caricati uno di seguito all'altro; l'indirizzo di base è quello del primo segmento.

STACK : come PUBLIC.

COMMON : tutti i segmenti con lo stesso nome e della stessa classe sono caricati sovrapposti; tutti i segmenti con lo stesso nome hanno un unico indirizzo di base.

CLASS : viene indicata con un nome messo tra ' apicì; essa determina l'ordine con cui il LINK deve correlare i segmenti: tutte le parti assegnate al medesimo nome di classe sono caricate in memoria una dopo l'altra.

TYPE : per etichette NEAR indirizzo nel medesimo segmento;
FAR indirizzo in segmento diverso;
VALUE : offset rispetto all'inizio del segmento.