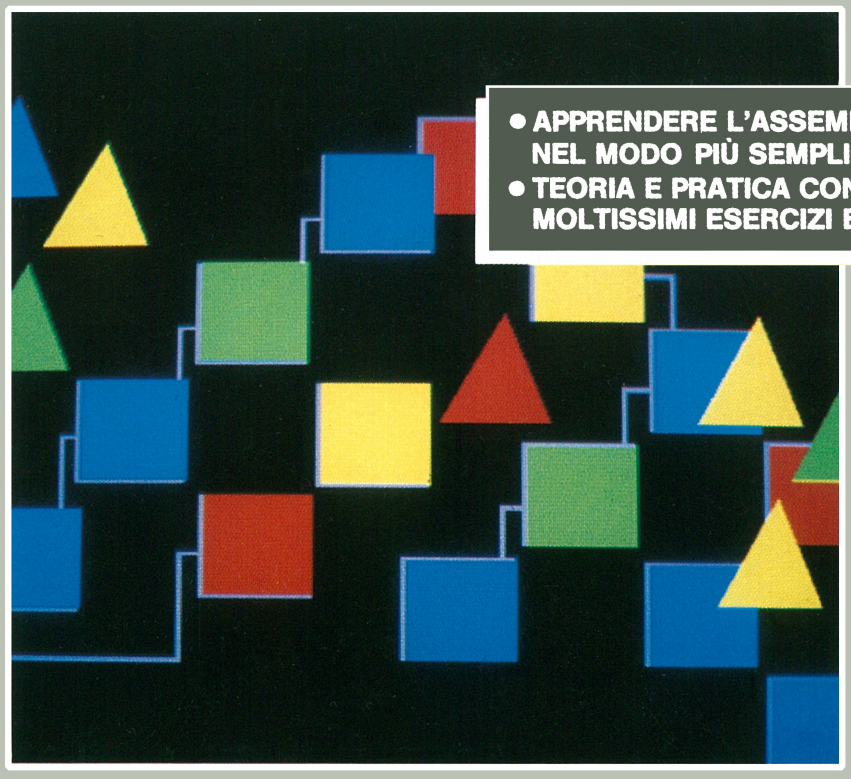
LINGUAGGI

PROGRAMMARE IN ASSEMBLER

ALAIN PINAUD



- APPRENDERE L'ASSEMBLER NEL MODO PIÙ SEMPLICE
- TEORIA E PRATICA CON MOLTISSIMI ESERCIZI ED ESEMPI

GRUPPO EDITORIALE
JACKSON

PROGRAMMARE IN ASSEMBLER

ALAIN PINAUD

GRUPPO
EDITORIALE
JACKSON
Via Rosellini, 12
20124 Milano

ALAIN PINAUD, 35 anni, lavora come informatico presso la casa costruttrice francese di elaboratori CII-HONEYWELL BULL, dove partecipa allo studio e alla realizzazione di sistemi speciali, sia dal punto di vista dell'hardware che del software.

SOMMARIO

PREFAZIONE

Questo libro si rivolge a quella "crescente coorte di infelici" che, pur conoscendo gli elementi di un linguaggio evoluto di un elaboratore (per esempio il BASIC) esitano ancora a intraprendere la strada maestra dell'assembler.....

Mi si dirà che hanno ragione di esitare, visto quello che si racconta:

— che l'assembler non è alla portata di un amatore, ma è cosa da professionista,
— che l'assembler è riservato ad un'élite,
— che l'assembler è terribilmente complesso e richiede studi molto lunghi,
— e blablabla, e blablabla....

Un vecchio proverbio cinese dice: "*Quello che può fare uno sciocco, lo può fare anche un altro*", e quando si vede il numero di sciocchi che sanno programmare in assembler, c'è da meravigliarsi che tanta gente intelligente non lo sappia ancora fare!

Tra questi sciocchi - gli stessi che si incontrano anche in certi ambienti di matematici - ce n'è di quelli che si raddrizzano dietro di loro il cespuglio di rovi che hanno superato, come per sviare gli inseguitori, e di tanto in tanto esclamano: "*Sì....potete seguirmi...ma la strada è lunga e difficile...mooolto difficile.....*".

Infine, in mezzo a questi sciocchi, esistono anche alcuni abbrutiti, particolarmente degenerati, che osano ancora pensare che l'assembler è alla portata di tutti, se solo gli si dedica il tempo necessario al suo studio, e lo si affronta con fare deciso e senza complessi.

Che questa opera, modesta, e ridicola, sia l'impulso, la "pacca sulle spalle" che vi spinga nella direzione della via maestra....la dritta e giusta via dell'assembler.

INTRODUZIONE

Un edificio, per quanto impressionante, è sempre formato da un insieme di pietre o di mattoni, e siccome è necessario partire da qualche cosa, si farà qui l'ipotesi che il **lettore abbia una buona conoscenza di un linguaggio evoluto molto semplice, come il BASIC,** che è attualmente il più diffuso, ed è previsto dalla maggioranza di Personal Computer in commercio.

Naturalmente, questa conoscenza di un linguaggio evoluto non è una condizione indispensabile per affrontare l'assembler, ma consentirà, quando se ne sentirà il bisogno, di sviluppare alcune analogie tra questi due tipi di linguaggi, adatte a chiarire o a consolidare i nostri studi.

Il lettore che non abbia ancora confidenza con nozioni come binario, bit, base, esadecimale, etc...., ha, a sua disposizione, alla fine del volume, alcune appendici destinate a facilitargliene la comprensione.

Poichè l'obiettivo principale di questa opera è di fornire i rudimenti che consentano di programmare in assembler nel senso generale del termine, era necessario, negli esempi concreti, utilizzare come supporto, le istruzioni di un assembler reale (ce ne è in abbondanza....).

Un'altra possibilità sarebbe stata quella di creare, per questo scopo, delle istruzioni inventate (per esempio in italiano). Ma questo sarebbe stato come fare un passo indietro per poi fare un salto più lungo.

È meglio abituarsi, il più presto possibile, alla terminologia esistente e, solo dopo averla assimilata, sarà possibile "sognare" un po'....

Se si vuole fare lo sforzo di comprendere un aborigeno d'Australia, non si può cominciare pretendendo da lui che capisca l'italiano, non vi pare? Il microprocessore è nato parlando in Americano, bisogna abituarsi a quest'idea.

Quale tra gli assembler esistenti era dunque opportuno scegliere? Quello del **microprocessore Z80 della Zilog** è stato adottato per due ragioni fondamentali; è uno dei più diffusi, e il suo set di istruzioni è il più ampio nella sua categoria, e include quello del cugino 8080 della Intel, ugualmente diffuso, e del più recente

8085, a meno di due istruzioni (ma non è il caso di entrare troppo nei dettagli tecnici, per il momento...!).

Cercando bene, si troverebbe anche una terza ragione, ma non ne parleremo....! (*)

Infine bisogna dire una cosa: una volta che si conoscono i principi di base del linguaggio assembler, essi restano validi su qualsiasi elaboratore, che sia a 8 bit o a 64 bit.

Le sole differenze saranno nella potenza, nella flessibilità e nell'estensione del set di istruzioni, oltre che, naturalmente, nella rapidità di esecuzione delle istruzioni.

Ma se la moltiplicazione o la divisione non compaiono tra le istruzioni di un assembler a 8 bit, certe istruzioni caratteristiche degli "8 bit" (scambi, registri, manipolazioni dei bit, stack....) non sono sempre presenti nei "grossi" microprocessori. E qualche volta questo è molto fastidioso.

Dunque non bisogna avere dei complessi fuori luogo!

È ora venuto il momento di passare alle cose serie:

io assemblo,
tu assembli,
egli assembla....

* Nota dell'editore: "No! L'autore non ha azioni della Zilog. Possiede un TRS-80!".

CAPITOLO 1

DEFINIZIONI E RICHIAMI DI NOZIONI DI BASE

È ancora un po' presto per dare una definizione di assembler. Dunque, per ora, esamineremo quei casi in cui è giustificato il suo impiego. Tentiamo comunque un primo approccio.

C'ERA UNA VOLTA.....

Il signor Yves Venelse aveva un sacro terrore dei rubinetti che perdono. Niente di più angosciante di quel POKE POKE. Roba da diventare pazzo furioso! Il suo urlo di rabbia fece entrare LET (in realtà si chiamava Colette), che domandò subito:

—"*Il signore mi ha chiamato?*"
—"*REM...sì*"
rispose Venelse schiarendosi la voce. Poi riprese:
—"*Voglio che questo rubinetto sia riparato entro stasera. Capito, LET?*"
Essa borbottò un "*Va bene, signore*" e scomparve.

La sera, tornando a casa, Yves Venelse constatò distrattamente che il rubinetto non perdeva più, e non ci fece più caso, ignorando quanto traffico questa riparazione avesse richiesto: si era dovuto cercare un idraulico libero, disposto a fare tanta strada per un semplice rubinetto, davanti al quale avrebbe dichiarato: "*che era un vecchio modello*", "*che non aveva di che ripararlo*", "*che avrebbe dovuto tornare in città*", e ancora "*che sarebbe venuto a costare molto caro*"..., senza contare le inondazioni provocate da quel maldestro...Ah! Il signore ha la vita facile!

J.P.Hachel, anche lui, aveva orrore dei rubinetti che perdono. Niente di più angosciante di quel POP....POP...

Amante del "fai da te" e meticoloso di natura, decise di fare il lavoro lui stesso. Dopo aver tolto l'acqua, prese la sua cassetta degli attrezzi, scelse la chiave adatta, e svitò la testa del rubinetto. Tolse la guarnizione responsabile e la rimpazzò con una nuova, che aveva religiosamente conservato in una scatoletta. Rimontò la testa del rubinetto, la fissò con l'aiuto della chiave, e riaprì l'acqua. Il guasto era riparato.

Questa è una storiella, ma non ne avrete sempre di questo genere....! Per il momento, anche se non sapete ancora che cos'è l'assembler, almeno sapete riparare un rubinetto che perde!

Su queste due storie possiamo fare le seguenti osservazioni:
— *Yves Venelse* parla un linguaggio evoluto. L'ordine "*voglio che questo rubinetto....*" è molto chiaro, e sufficiente perchè l'azione che ne consegue sia correttemente interpretata; mentre non ha niente a che fare con i dettagli riguardanti l'idraulico, e ancor meno la guarnizione del rubinetto.

D'altra parte ha forse bisogno, di sapere che cos'è una guarnizione? Dal momento in cui l'ordine viene dato fino a quando il rubinetto è riparato, passa un certo tempo, e si eseguono un certo numero di azioni che il signor Venelse ignora — e vuole ingnorare. L'unica cosa che conta per lui è il risultato.

— *J.P. Hachel* invece attribuisce pari importanza sia a quello che fa, che al risultato finale. Non parla nemmeno, o se parla, lo fa tra di sè. Probabilmente si dirà cose del genere:

— togliere l'acqua,
— prendere la chiave da 15,
— no, non è quella buona...,
— proviamo con quella da 16....OK.,
— svitare la testa del rubinetto...girare...ancora...,
— togliere la guarnizione,
— etc...

Il suo linguaggio non è molto evoluto, ma ad ogni "frase" corrisponde una piccola azione precisa, senza equivoci, che chiameremo "microazione". Anche se "togliere l'acqua" è in realtà un insieme di "micro-azioni" che si possono descrivere nel modo seguente:
— girare il rubinetto verso destra e continuare fino a quando si blocca...
Più tardi chiameremo tutto ciò **un "sottoprogramma"**.
Vediamo ora che informazioni possiamo estrarre da queste storie:

— Se Y. Venelse parla BASIC o qualcosa di simile, J.P. Hachel parla Assembler.

— Le azioni richieste dall'ordine di Y. Venelse saranno in fin dei conti, realizzate in "Assembler".
— Il lavoro di Y. Venelse si limita a ben poco. Basta che egli dica "*Io voglio che.....*". La realizzazione d'altra parte è più laboriosa. Ma cosa importa.
— Il lavoro di J.P. Hachel è molto più oneroso, ma in compenso la realizzazione è molto più rapida.
— Ci si ritrova un pò la stessa differenza che può esistere tra i due linguaggi: uno chiaro e netto, facile da formulare, l'altro laborioso, meticoloso, una piccola bomba che si prepara con cura.....

In quali casi dunque si giustifica l'impiego dell'"Assembler"?

— Tutte le volte che il fattore **tempo** gioca un ruolo fondamentale. Tra il tempo di esecuzione di un programma che funziona sotto l'interprete BASIC, e quello di un programma scritto direttamente in assembler, può esserci un rapporto enorme (da una a più decine).

— Tutte le volte che il fattore **spazio** gioca un ruolo fondamentale. La dimensione di un programma BASIC (sorgente) aggiunta a quella del suo interprete, sarà sempre superiore a quella dello stesso programma (che esegue la stessa funzione), realizzato in Assembler.

— Tutte le volte che è necessario realizzare delle **funzioni sconosciute** dell'interprete, o impossibili da realizzare con un linguaggio evoluto.

Naturalmente è sempre possibile (e lo si fa spesso) conciliare le due forme di linguaggio (evoluto e assembler), ad esempio scrivendo l'ossatura del programma in BASIC, e ricorrendo quando è necessario a dei piccoli programmi realizzati in assembler. Esamineremo questa possibilità in un capitolo successivo.

UNA PICCOLA COMPRESSA CON UN PO' D'ACQUA ZUCCHERATA

Ci sembra ora necessario fare un po' di anatomia. Fondamentalmente l'elaboratore è composto di un **microprocessore**, che è in un certo senso il suo cervello, e da una **memoria** di dimensioni variabili, in cui sono conservati programmi ed informazioni.

Per poter comunicare con il mondo esterno (ad esempio l'uomo) esso deve essere fornito di dispositivi che si chiamano **periferiche**, che possono limitarsi ad una semplice tastiera (ingresso delle informazioni), e un video (uscita delle informazioni).

Eventualmente, la memoria può essere ampliata con delle unità a cassette magnetiche o a piccoli dischi, che hanno anche loro il compito di conservare - ma in maniera più duratura - programmi e informazioni.

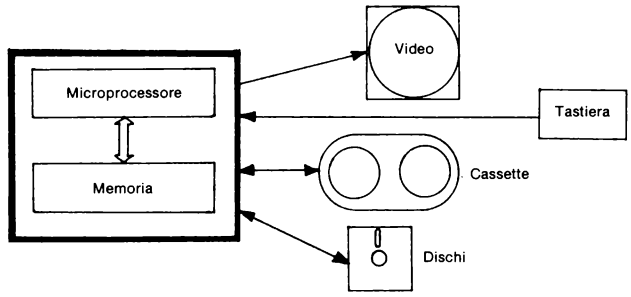


Figura 1

Da quando l'elaboratore viene acceso, il microprocessore cerca di eseguire (è un bisogno fisico!) il **programma** che si trova all'inizio della memoria (indirizzo 0).

Questo programma, per poter essere elaborato ed eseguito dal microprocessore, deve necessariamente essere composto di **istruzioni** conosciute dal microprocessore, per la cui esecuzione esso è stato progettato e che gli sono "congeniali". (ciascun tipo di microprocessore ha le sue).

Proprio come il DNA determina la nostra personalità, queste istruzioni determineranno la "personalità" del microprocessore, le sue prestazioni, la sua potenza, la sua "intelligenza".

Ogni istruzione produce, all'interno dell'elaboratore, una "micro-azione" ben precisa, e l'insieme di queste micro-azioni produrrà una azione percepibile dall'uomo, per esempio la stampa di un messaggio sul video, o la ricezione di un messaggio dalla tastiera, o ancora la lettura o la scrittura di una informazione su un'unità a cassetta.

Tuttavia, l'azione risultante può non essere percepita direttamente dall'uomo. È il caso, per esempio, di un calcolo complesso o di una scelta di numeri in memoria.

Quando un uomo cerca un nome nella sua memoria, o lo scrive su di un foglio di carta, vengono eseguite, a livello del suo cervello, migliaia di micro-azioni.

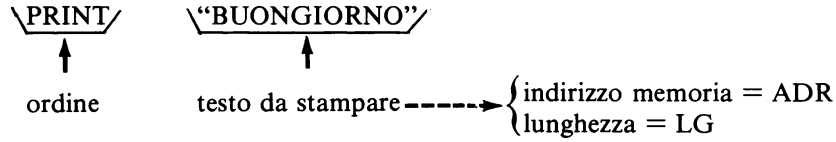
In questo ultimo esempio, si può pensare che il cervello selezioni un "programma di scrittura" che, in collegamento permanente con la memoria, lancia una quantità di ordini o di istruzioni che provocano le micro-azioni necessarie per il comando di alcuni muscoli specializzati, per coordinare i movimenti del polso, della mano e delle dita, mentre l'occhio, registrando i più piccoli spostamenti, li trasmette al cervello per correggere un evenutale "errore di direzione", o per regolare la pressione della penna sulla carta.

La prossima volta che voi batterete: PRINT "BUONGIORNO" sulla tastiera del vostro elaboratore, forse non vedrete più le cose nello stesso modo.....

In effetti, per l'elaboratore, questo semplice ordine in BASIC provoce tutto un tramestio interno, un po' simile a quello che deve esistere inconsciamente nel vostro cervello quando qualcuno vi dice: "*SCRIVI....questo o quello....*".

Cerchiamo un pò di capire che cosa succede realmente al livello della macchina.

La frase: PRINT "BUONGIORNO" battuta sulla tastiera è trasferita nella memoria dell'elaboratore dal programma Interprete del BASIC. Dopo aver analizzato l'ordine (PRINT), il controllo passa ad un piccolo programma specializzato (sottoprogramma), il cui compito è di stampare dei caratteri sullo schermo del video (noi lo chiameremo "STAMPA"). Per far ciò, è necessario comunicare a questo sottoprogramma l'indirizzo della memoria (perchè in questo caso noi lavoriamo in collegamento diretto con la memoria) che contiene il testo da stampare, nonchè la sua lunghezza.



Il sottoprogramma esegue allora le seguenti operazioni:

1. Prendere il carattere che si trova all'indirizzo di memoria ADR,
2. Chiamare un altro sottoprogramma per stampare questo carattere sul video (chiamiamolo "SCHERMO"),
3. Aggiungere 1 al valore dell'indirizzo ADR,
4. sottrarre 1 alla lunghezza LG,
5. Se LG è diverso da 0, ritornare al punto 1, altrimenti ritornare al programma che ha fatto la chiamata.

Queste operazioni possono essere rappresentate graficamente per mezzo di un "**diagramma a blocchi**". È più facile da capire, basta seguire le frecce....

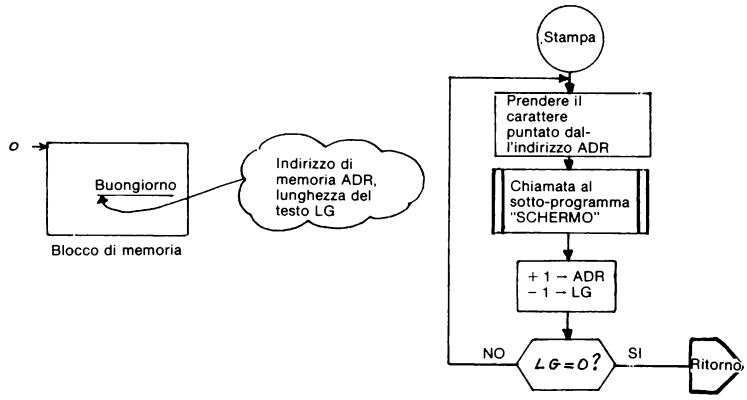


Figura 2

A titolo di documentazione, ecco il programma BASIC che permette di avvicinarsi al massimo a questo diagramma a blocchi:

```
10  A$ = 'BUONGIORNO'      'DEFINIZIONE DEL TESTO
20  ADR = 1                'PUNTATORE ALL'INDIRIZZO
30  LG = 10                'LUNGHEZZA (O LG=LEN(A$))
40  GOSUB 100              'CHIAMATA SUB STAMPA
50  .......
      .
      .
      .
100 'SOTTOPROGRAMMA STAMPA
102 'DATI DI INGRESSO :ADR = PUNTATORE INDIRIZZI
103 '                  LG  = LUNGHEZZA TESTO
104 '
110 C$ = MID$(A$,ADR,1)    'PRENDI CAR PUNTATO DA ADR
120 PRINT C$;              'STAMPA SUL VIDEO
130 ADR = ADR + 1          'PUNTATORE INDIRIZZI+1
140 LG = LG-1              'LUNGHEZZA -1
150 IF LG <> 0 THEN 110    'CONTROLLA SE FINE TESTO
160 RETURN                 'RITORNO AL PROG PRINCIPALE
```

Questo è evidentemente un modo di utilizzare il BASIC al minimo delle sue possibilità, poichè basterebbe scrivere PRINT "BUONGIORNO" per avere lo

stesso risultato...ma questo programma ci ha permesso di "scendere di un gradino" nella scala dell'evoluzione dei linguaggi dell'informatica. E non è finito! Fermiamoci un po' per respirare.

Dicevamo precedentemente che bisognava comunicare al sottoprogramma STAMPA i dati relativi all'indirizzo e alla lunghezza del dato da stampare.

Nel programma BASIC precedentemente, noi abbiamo chiamato questi dati ADR e LG. Ma se questo è sufficientemente chiaro in BASIC, non lo è affatto al livello della macchina.

L'elaboratore non conosce che gli indirizzi di memoria e i **registri.**
Bisognerà quindi specificare in partenza:

— l'indirizzo del testo da stampare si trova nella locazione di memoria 12693, per esempio, e la lunghezza di questo testo nella locazione 15312.

È più facile dire:

— l'indirizzo del testo da stampare si trova nel registro X e la lunghezza nel registro A.

Questo concetto di "cassetta delle lettere" deve evidentemente essere conosciuto dal programma e dai sottoprogrammi richiamati. Che cosa sono dunque i registri?
Sono un tipo di memoria rapida utilizzata dal microprocessore per conservare temporaneamente le informazioni, e che esso utilizza un po' come block-notes. Ma invece che accedervi tramite un indirizzo come per la memoria, vi accede tramite i loro nomi: A, B, H, L, ad esempio, rappresentano quattro registri del microprocessore Z-80.

Il numero di questi registri, le loro dimensioni e i loro nomi variano da un microprocessore ad un altro. Diversamente dalla memoria che è separata dal microprocessore, i registri sono al suo interno. Il set di istruzioni naturalmente utilizza questi registri.

Ma ritorniamo al nostro diagramma a blocchi della Figura 2. Come ci si deve esprimere per "dire" al microprocessore: "*prendere il carattere puntato dall'indirizzo ADR*"?

Possiamo farlo per mezzo di un'istruzione creata apposta per questo, e conosciuta dal microprocessore:

LD A, (HL)

che significa: caricare (LOAD in inglese LD) il registro A con il contenuto dell'indirizzo di memoria che si trova nel registro doppio HL.
Ecco stiamo bruscamente "scendendo la scala", ma parliamo assembler! Quello dello Z-80.

In questo caso particolare, il registro HL gioca il ruolo di "cassetta per le lettere" citata prima, e contiene l'indirizzo ADR che punta al testo "BUONGIORNO".

Naturalmente è compito del programma principale caricare questo registro prima di richiamare il sottoprogramma. Notiamo per inciso che questa forma di linguaggio, mentre diventa comprensibile per l'elaboratore, lo è sempre di meno per l'uomo....

Dopo l'esecuzione di questa istruzione, il registro A conterrà il primo carattere del testo, cioè la lettera "B".

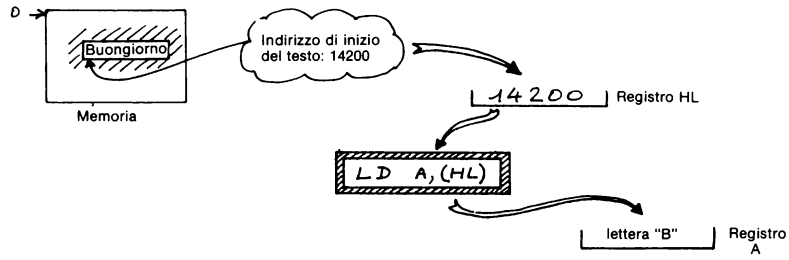


Figura 3

"*Sì, ma...*", direte voi, giustamente, "*io credevo che il microprocessore comprendesse realmente solo i simboli 0 e 1 del sistema binario. In questo caso, come fa a capire qualcosa del tipo: LD A, (HL).....*"?

In effetti esso non lo può capire, ed è proprio lì il dramma.

In realtà il codice dell'istruzione LD A, (HL) equivale per lo Z-80 alla codifica binaria:

0 1 1 1 1 1 1 0

cioè 7E nel sistema esadecimale.

Quando il microprocessore trova questa configurazione binaria in memoria, esegue quello che noi, esseri umani, chiamiamo l'istruzione LD A, (HL).

Poichè sarebbe fastidioso programmare un elaboratore in binario, i progettisti dei microprocessori, hanno previsto e scritto un programma che realizza questa conversione: noi diciamo a questo programma "LD A, (HL)" e lui traduce "01111110".

Questo programma è l'assemblatore

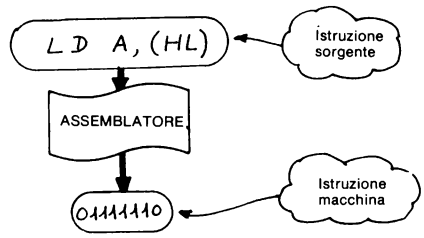


Figura 4

Ogni costruttore di microprocessori deve anche fornire il programma assemblatore corrispondente. In effetti, senza quest'ultimo, il microprocessore sarebbe difficilmente utilizzabile.

Ma questo programma assemblatore, in quale linguaggio è scritto? (costui comincia a innervosirmi con le sue domande). Forse in assembler? Esatto. Ma avrebbe potuto benissimo essere scritto in BASIC o in FORTRAN....Sì, ma l'interprete BASIC, in quale linguaggio è scritto? Hum...in assembler.

Di fatto, il **primo** assemblatore del *primo* microprocessore ha dovuto essere scritto su di un altro elaboratore (che non aveva niente del microprocessore, certamente) ...e il *primo* assemblatore del *primo* elaboratore ha dovuto essere scritto direttamente in binario....Non c'era altra soluzione!
Le istruzioni fornite all'assemblatore si chiamano "**istruzioni sorgente**", e quelle prodotte dall'assemblatore "**istruzioni macchina**".

L'insieme delle istruzioni "sorgente" formano il "**codice sorgente**", e l'insieme delle istruzioni "macchina" formano il "**codice macchina**" o "**codice oggetto**". Si trovano anche i termini "codice esterno" per il codice sorgente, e "codice interno" per il codice oggetto.

Possiamo ora dare una definizione dell'assemblatore?

Sì. Diremo che l'assemblatore è un **programma** che traduce le istruzioni scritte in linguaggio sorgente nel loro equivalente in linguaggio binario (stabilendo una corrispondenza biunivoca tra le istruzioni sorgente e le istruzioni macchina). Questo significa che per ogni istruzione sorgente (ingresso) esso produrrà un'istruzione macchina (uscita) direttamente interpretabile dall'elaboratore per il quale è stata concepita.

L'AMBIENTE DEL PROGRAMMA

Diversamente che per i linguaggi evoluti, l'utilizzo dell'assembler richiede delle nozioni, che chiameremo semi-tecniche, che è necessario chiarire da subito, e che sono strettamente legate alla macchina.

La memoria è, probabilmente lo sapete, il supporto che permette di conservare e trattenere nel tempo le informazioni. Nel corso del nostro studio, noi utilizzeremo soprattutto la memoria detta "centrale", che è direttamente collegata al microprocessore, e che contiene programmi e dati.
Si accede a questa memoria per mezzo di un indirizzo che assume i valori che vanno da 0 (inizio memoria) a FFFF (fine memoria) per un totale di 65536 "celle", che è in generale il valore limite per la maggioranza dei microprocessori attuali.

Poichè la programmazione in assembler si allontana dall'uomo per avvicinarsi alla macchina, è più comodo utilizzare il sistema di numerazione esadecimale per esprimere un indirizzo o un dato. Cominciamo dunque ad abituarci! (Vedi Appendice 1).
La memoria centrale può essere paragonata ad una cassettiera, in cui ogni cassetto rappresenta una cella di memoria che si chiama "**byte**".
Ogni byte a sua volta contiene otto informazioni binarie che si chiamano "**bit**".

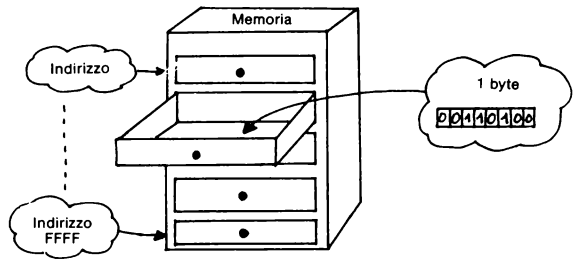


Figura 5

Ogni bit, che si può considerare come l'atomo dell'informazione", è indivisibile, e può assumere uno dei due valori binari 0 e 1.

In un esempio precedente, il testo "BUONGIORNO" era collocato a partire dall'indirizzo di memoria 14200 (anzi 3778H). Questo significa che il carattere "B" era all'indirizzo 3778H, il carattere "0" all'indirizzo 3779H, il carattere "N" all'indirizzo 377AH...etc. Vedremo più avanti che si può memorizzare un solo simbolo dell'alfabeto in un byte.

Il testo "BUONGIORNO" rappresenta un'informazione del tipo "**dato**" diversamente da un'informazione del tipo "**istruzione**"; questi sono i due tipi di informazione che possono trovarsi in memoria. Ma attenzione! Il microprocessore può eseguire soltanto istruzioni!

Se per errore gli si fanno eseguire dei dati (cosa che purtroppo può capitare) ciò che avviene non è poi così drammatico; l'elaboratore "si impianta" come dicono i programmatori.

È come quando si parte per pescare, tranquillamente. Si getta la lenza. Cinque minuti: niente. Dieci minuti: niente. Un'ora: niente. Allora si ritira la lenza: non ci son più nè vermi, nè amo, nè galleggiante.....D'altra parte non c'è più neanche il fiume e neanche la lenza....Il pescatore? Quale pescatore? Non c'è nessun pescatore. Solo gli uccellini...cip...cip...cip..

Ricordiamoci dunque che i dati sono destinati ad essere utilizzati dalle istruzioni, e non devono sostituirsi a queste, a meno che non sia fatto volontariamente. Mi è capitato di vedere dei programmatori viziosi.....

I **registri** possono essere considerati come delle celle di memoria contenenti uno o due bytes. Essi sono utilizzati dal microprocessore e dalle istruzioni del codice macchina come memorie temporanee.

Analogamente, quando si fa una moltiplicazione su un foglio di carta, è necessario scrivere i risultati parziali per poi sommarli. Ma una volta ottenuto il risultato, non sono più necessari e possono essere cancellati e dimenticati.

In generale si accede ai registri non tramite un indirizzo come avviene per la memoria, ma con un nome simbolico: A, X, HL, SP.....

Il microprocessore Z-80 possiede un secondo set di registri, A' B' ...L', che possono essere sostituiti con una semplice istruzione macchina agli 8 registri da A a L. Dopo l'esecuzione di questa istruzione, i registri A....L diventano A'...L', e viceversa.

Come mostra la Figura 6, certi registri sono semplici (8 bits) e altri doppi (16 bits).

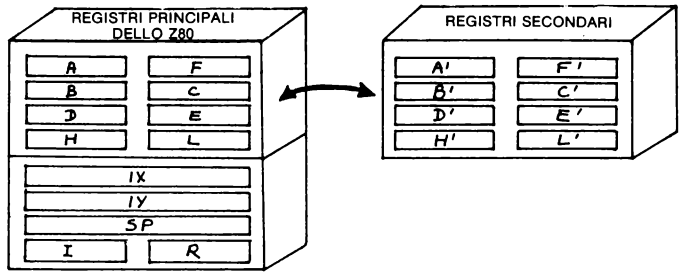


Figura 6

Inoltre certe istruzioni considerano i registri A e F, B e C, D e E, H e L, come dei registri doppi: AF, BC, DE, HL.

Per esempio, l'istruzione:

```
LD  B, 4
```

mette il valore 4 nel registro B, mentre l'istruzione

```
LD  BC, 1234H
```

mette il valore 1234H nel doppio registro BC, il che equivale a fare:

```
LD  B, 12H
LD  C, 34H
```

con istruzioni che utilizzano registri semplici.

Lasciamo ora i registri per parlare della **pila** (o stack, in inglese) che non si deve confondere con quella da 1 volt.....

Una delle difficoltà della programmazione in assembler è la gestione dello spazio di memoria. Per sistemare un'informazione, bisogna sapere in quale indirizzo metterla per poterla ritrovare più tardi.

Con lo "stack" è più semplice. Come dice il suo nome (pila) si "impilano" le informazioni da sistemare, una dopo l'altra, come se fossero una pila di piatti, senza doversi preoccupare dell'indirizzo di memoria in cui si trovano realmente, e per riprenderle si tolgono dalla pila.

Esempio

Mettere in memoria il contenuto dei registri doppi BC e DE, eseguire una certa elaborazione, poi ricaricare i registri BC e DE con i loro valori iniziali.

Basta fare:

```
PUSH  BC      ; inserimento nella pila di BC
PUSH  DE      ; inserimento nella pila di DE
   .
   .
   .          ; elaborazione
   .
POP   DE      ; recupero di DE
POP   BC      ; recupero di BC
```

Avete capito: l'istruzione *PUSH* inserisce nello "stack", e l'istruzione *POP* preleva dallo stack.

Ma attenzione all'ordine delle operazioni: se si inserisce prima il piatto "BC", e poi, sopra di lui il piatto "DE", bisognerà togliere DE per poter riprendere BC....È logico!
Nel semplice esempio precedente si suppone che l'elaborazione, che si trova tra l'inserimento e il recupero, non modifichi lo stack, ovviamente.

Dove si trova questo stack? In memoria. Un'istruzione particolare consente di fissarne l'indirizzo di partenza. Uno dei registri del microprocessore, denominato "stack pointer" (SP), cioè "puntatore della pila" deve essere opportunamente caricato prima di utilizzare istruzioni che operano sullo stack.

```
LD    SP,4000H     ; lo stack inizia all'indirizzo 4000H
LD    BC,1234H     ; si memorizza 1234H nel registro BC
LD    DE,5678H     ; si memorizza 5678H nel registro DE
PUSH  BC           ; si inserisce BC nello stack
PUSH  DE           ; si inserisce DE nello stack
```

Dopo l'esecuzione di queste cinque istruzioni, la memoria contiene le seguenti informazioni:

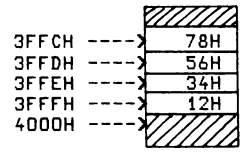


Figura 7

Osserviamo che le informazioni sono sistemate nello stack dal basso verso l'alto (come in una pila di piatti...). Ad ogni inserimento nello stack, il microprocessore sottrae 1 al contenuto del registro SP,per tenere aggiornato il puntatore. Quindi, dopo l'esecuzione delle precedenti istruzioni, il registro SP conterrà il valore 3FFCH. Sarà sufficiente eseguire due istruzioni POP per riportare SP al suo valore iniziale (4000H).

Notiamo che, in questo caso, le informazioni che si trovano agli indirizzi compresi tra 3FFCH e 3FFFH, non saranno modificate da queste due istruzioni, ma verranno cancellate da successivi inserimenti nello stack.

Poichè non è facile determinare a priori la dimensione, lo stack è generalmente sistemato "in fondo" alla memoria, cioè agli indirizzi alti, per non disturbare nessuno.

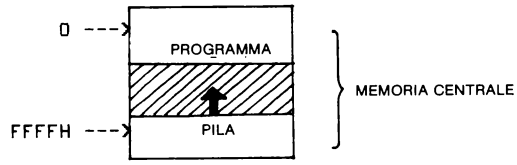
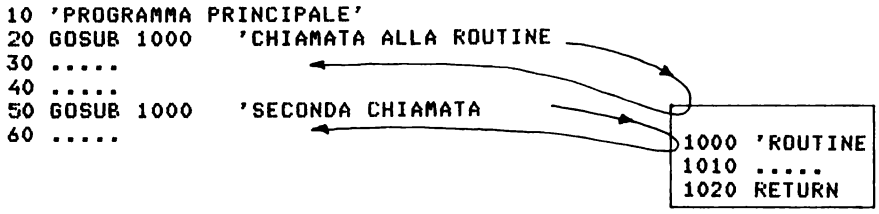


Figura 8

Vedremo ora che lo stack è, utilizzato - peraltro con discrezione - dai **sottoprogrammi**.

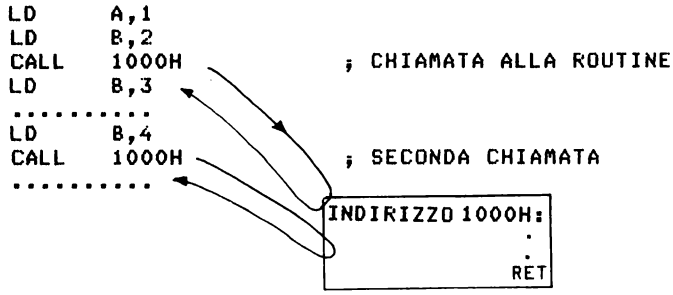
Voi conoscete senz'altro l'istruzione BASIC "GOSUB", che consente di interrompere momentaneamente l'esecuzione di un programma, per eseguire una "routine" o un sottoprogramma - una specie di programma di utilità, di cui si ha bisogno molto spesso - e poi tornare al punto in cui è stato interrotto il programma principale.



Questo artificio evita di fatto la scrittura di sequenze di istruzioni che si ripetano, e migliora la comprensione e la leggibilità dei programmi, che così risultano meglio strutturati.

Esiste una funzione equivalente in assembler:

— l'istruzione **CALL** *consente di richiamare una routine,*
— l'istruzione **RET** *provoca il ritorno al punto in cui è stata fatta la chiamata.*



Quando il microprocessore incontra l'istruzione CALL, colloca automaticamente nello stack (con un PUSH in un certo senso invisibile) l'indirizzo di memoria che punta l'istruzione immediatamente successiva alla CALL (LD B, 3 nel nostro esempio), e va a eseguire la routine che si trova all'indirizzo 10000H (per esempio).

Alla fine di questa routine, l'istruzione RET provoca un recupero dallo stack (con un POP invisibile), e il programma prosegue la sua esecuzione a partire dalla istruzione immediatamente successiva alla CALL.

Come nel caso delle istruzioni PUSH e POP, è necessario che il registro SP sia inizializzato prima di qualsiasi chiamata ad un sottoprogramma.

Nell'esecuzione della routine, bisognerà evidentemente sorvegliare i movimento dello stack, per evitare il rischio di non tornare all'indirizzo voluto! Non è proibito eseguire delle PUSH, ma a condizione di fare un uguale numero di POP prima di raggiungere l'istruzione RET.

Torniamo ora un po' indietro, al momento in cui avevamo memorizzato il testo "BUONGIORNO".

Avevamo visto che le lettere che compongono questo testo erano memorizzate in memoria una dietro l'altra. Ma in quale forma?

È tutto un problema di convenzioni in informatica: il codice dell'istruzione Z80 "LD A, (HL)" è 7EH, e quello della lettera "A" sarà 41H, ma questa volta si tratta di un codice internazionale, di importanza fondamentale in tutto quello che riguarda gli elaboratori e la trasmissione dei dati: **il codice ASCII.**

A.S.C.I.I. è formato dalle iniziali delle parole *American Standard Code for Information Interchange*, che significa: Codice Standard Americano per lo Scambio di Informazioni (i segnali di fumo erano per caso in ASCII?). La cosa interessante è che questo codice è conosciuto e interpretato da quasi tutte le periferiche degli elaboratori.

Se si schiaccia il tasto "A" di una tastiera collegata ad un elaboratore, ne uscirà...indovinate cosa...il codice 41H. E così, se si invia questo codice ad un terminale video (o ad una telescrivente) esso provocherà la stampa della lettera "A". Però! È bello essere capiti in questo mondo di elettroni!

Il codice "ASCII", è formato da 128 differenti combinazioni di cifre binarie, che permettono di rappresentare (vedi Appendice III):

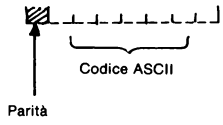
— le dieci cifre arabe (da 0 a 9),
— alcuni simboli speciali (=, $, %, &,.....),
— le lettere dell'alfabeto,
— alcuni codici di "funzione" (a capo, salto pagina, etc....)

Ahimè, cento volte ahimè!....Questo "Esperanto" del trattamento dell'informazione talvolta è violato dai costruttori di periferiche, che trovano più logico stampare un "[" al posto di un "↑", o qualche altra fantasia di questo tipo.

Per fortuna il microprocessore non capisce l'ASCII, così non corre il rischio di aver dei problemi, ed è compito del programma interpretare il codice dei caratteri.

Esistono altri codici ma, poichè l'ASCII è il più diffuso nel mondo dei microprocessori, noi non ce ne occuperemo.

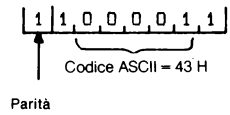
In ASCII, ogni simbolo o funzione è codificato in byte di cui utilizza gli ultimi 7 bits. Per questa ragione viene anche chiamato "codice a 7 elementi". Questi 7 bits sono sufficienti per rappresentare 128 configurazioni, e consentono di utilizzare l'ottavo bit per il "controllo di parità".



Quando si devono trasmettere delle informazioni codificate in ASCII ad una periferica lontana (per esempio all'altro capo di una linea di trasmissione telefonica) è necessario poter controllare la "bontà" delle informazioni trasmesse, che possono essere alterate da disturbi della linea.

L'ottavo bit (il bit di parità) indicherà dunque con il suo stato (0 o 1) se il numero di 1 del codice ASCII trasmesso è pari o dispari, e questo permette al dispositivo ricevente di verificare la validità di ogni byte trasmesso (o quasi, perchè un errore può nasconderne un altro.....Ma ci sono dei metodi che consentono di superare anche questo problema....).

Per esempio, per trasmettere il carattere "C", che in ASCII è 43H, si trasmetterà:



Se durante la trasmissione si ha la perdita o l'aggiunta di un "1", con buona probabilità ciò verrà segnalato dal bit di parità. Il dispositivo ricevente lo segnalerà o stampando un carattere speciale o richiedendo al dispositivo trasmittente di ripetere la trasmissione di quel carattere.

Ma anche qui regna l'anarchia più completa!

Alcune periferiche si aspettano una parità pari (cioè il bit di parità a "0" se il numero di "1" è pari), altre una parità dispari (il bit di parità a "1" se il numero di "1" è pari) - d'altra parte, in questo caso, il bit di parità viene considerato nel computo degli "1" o no? - altre lo ignorano completamente (cosa che, tutto sommato, può non essere così stupida come sembra), e così, molto spesso, è un dialogo tra sordi! E ancora una volta, è il programma incaricato della trasmissione che dovrà aggiustare le cose....

Quasi tutti i calcolatori hanno un'istruzione "magnifica", che spesso ha un ruolo molto importante: si chiama "**HALT**".

Come indica il suo nome, non fa assolutamente niente, e non è sempre facile non far niente!

Allora, stendetevi su di un tappeto, (così gli elettroni circolano meglio), rilassate tutti i vostri muscoli, e cercate di mettervi in "HALT" per qualche istante.

CAPITOLO 2

INTRODUZIONE ALL'ASSEMBLER

L'Assembler, a seconda del suo grado di evoluzione - eh sì, anche lui può essere più o meno evoluto - mette a disposizione del programmatore diversi tipi di funzione:

— le funzioni realizzate dalle istruzioni del microprocessore (queste sono obbligatorie in tutti gli assembler degni di questo nome). Sono le "**istruzioni** assembler" propriamente dette;

— alcune funzioni supplementari, in parte presenti solo negli assembler più evoluti, che servono o a deteminare l'inizio e la fine di un programma (queste sono indispensabili), o a facilitare la scrittura del programma, una sua eventuale modifica, e a migliorare la sua leggibilità e la sua comprensione. Le chiameremo "**pseudo-istruzioni**".

SIMBOLOGIA DELL'ASSEMBLER

Ci sono alcune prerogative che ogni buon assembler deve possedere. Una di queste è, ad esempio, l'indirizzamento "**simbolico**", che facilita notevolmente la scrittura dei programmi e consiste nell'utilizzare simboli alfanumerici per rappresentare degli indirizzi di memoria o dei dati.

Questi simboli sono chiamati "**etichette**" o "**label**". Le label sono quindi un mezzo mnemonico che permette al programmatore di associare un indirizzo di memoria al suo contenuto.

Così ad esempio, l'indirizzo di memoria destinato a contenere un prezzo, non sarà chiamato "124AH" ....ma "PREZZO".

Sarà poi compito dell'assemblatore sostituire queste tabelle con degli effettivi indirizzi di memoria, al momento della generazione del codice macchina.

Per quanto evoluto, il BASIC non ha questo metodo di indirizzamento. Sarebbe comodo poter scrivere:

```
GOTO INIZIO
```

o

```
GOSUB CONVERR
```

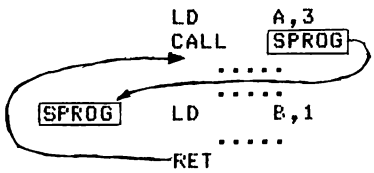
piuttosto che il solito:

```
GOTO 10
```

o l'altrettanto poco indicativo:

```
GOSUB 1500
```

che sono di una grande povertà semantica.
In assembler possiamo scrivere:



Anche il valore di una costante può essere rappresentato per mezzo di un simbolo:

```
TOTO    EQU     4           ; TOTO assume il valore 4
        ...
        ...
        LD      A,TOTO      ; scrive 4 nel registro A
```

Per far ciò si utilizza una **pseudo-istruzione** dell'assembler che specifica, nell'esempio, che il simbolo "TOTO" è equivalente (EQU) al valore "4".
In seguito l'istruzione "LD A, TOTO" caricherà questo valore 4 nel registro A, come avrebbe fatto l'istruzione "LD A, 4", ma in modo più "discorsivo".

FUNZIONI DELL'ASSEMBLATORE

Quando il codice sorgente del programma da assemblare è stato introdotto in memoria, in genere per mezzo di un altro programma speciale che si chiama "**editor**", o anche per mezzo dello stesso assemblatore, che in questo caso si

chiama "**editor-assemblatore**", il programma assemblatore esamina questo codice sorgente e comincia l'assemblaggio propriamente detto, determinando prima di tutto la lunghezza di ogni istruzione macchina corrispondente a ciascuna istruzione sorgente.

A seconda del tipo di istruzione questa lunghezza può variare da 1 a 4 bytes, almeno nel caso dello Z 80. Questo consente di determinare l'indirizzo di memoria dell'inizio di ogni istruzione macchina. Contemporaneamente, l'assemblatore costruisce una "tavola" (o tabella) dei simboli incontrati. Infine effettua un primo controllo sulla sintassi del codice sorgente. Tutto questo prende il nome di "I PASSATA" dell'assemblatore.

Durante la "II PASSATA" l'assemblatore esamina di nuovo il codice sorgente, associa un effettivo indirizzo di memoria ad ogni simbolo della tabella, e comincia a generare il codice oggetto insieme con il listing dell'assemblaggio, e procede ad una seconda ricerca degli errori che non hanno potuto essere rivelati durante la prima passata.

Il listing comprende, oltre il codice sorgente, gli indirizzi di memoria e il codice oggetto generato, insieme ovviamente agli evenutali errori riconosciuti dall'assemblatore. Prima di vedere un esempio di listing di assemblaggio dobbiamo studiare la forma e la sintassi del codice sorgente.

FORMATO DEL CODICE SORGENTE

Come al solito ci baseremo sulla sintassi dell'assembler dello Z80 della Zilog, sapendo che la comprensione dei suoi principi fondamentali faciliterà notevolmente l'eventuale; passaggio ad un altro tipo di microprocessore.

Il codice sorgente è formato da linee scritte una sotto l'altra. Ogni linea sorgente può avere due tipi di formato:

— il formato di commento, che inizia generalmente con un carattere speciale (';' nello Z80 e nel 8080), serve per documentare il programma, e non è elaborato dall'assemblatore, che si limita a riportarlo nel listing, senza modifiche.

```
; QUESTO È UN COMMENTO
```

— il formato funzionale, che è quello propriamente elaborato dall'assemblatore, che è composto da quattro campi: il campo "LABEL", il campo "OPERAZIONE", il campo "OPERANDO", il campo "COMMENTO".

Esempio:

```
LABEL   OPERATION       OPERANDE            COMMENTO
-----   ---------       --------            --------
INIZIO  LD              A,1                 ; CARICA 1 IN A
```

Secondo il tipo di funzione alcuni campi possono essere omessi, e in alcuni assembler il formato commento può essere considerato come un formato funzionale nel quale i campi label, operazione e operando sono omessi.

Esaminiamo ora le regole di sintassi generali che si applicano a questi diversi campi.

IL CAMPO LABEL

Il campo label deve iniziare nella prima posizione della linea sorgente, con un carattere non numerico. I caratteri speciali (%, $, =, ",....) non sono ammessi. Non può essere il nome di un'istruzione, di una pseudoistruzione o di un registro.

È formato da 1 a 6 (talvolta 8) caratteri senza spazi. In certi assembler (ad esempio quello dell'8080 dell'INTEL) la label deve terminare con il carattere speciale ":".

Questo campo è naturalmente **opzionale**.

Esempi di label corrette:

```
INIZIO
A2
A.VAL     (non sempre accettata)
VALORE
```

Esempi di label errate:

```
2A
LD        (se questo e' il nome di un'istruzione)
A VA1
TROPPOLUNGA
```

IL CAMPO OPERAZIONE

Il campo operazione comincia dopo il campo label (se questo è presente), dal quale deve essere separato almeno da uno spazio.

È **sempre obbligatorio**, salvo nel caso di un commento, ed è formato dal nome di un'istruzione o di una pseudo-istruzione assembler.

Esempi:

```
LD
PUSH  }          istruzioni Z80
CALL
EQU   }          pseudo-istruzioni
```

IL CAMPO OPERANDO

Il campo operando è il più complicato dei quattro...., tranne nel caso in cui è assente, cosa che può capitare (HALT per esempio).

Questo campo dipende strettamente dal tipo di operazioni, che si dividono in due importanti gruppi:

— ad un solo argomento (monadico)
— a due argomenti (diadico), separati da un carattere speciale (generalmente la virgola).

In quest'ultimo caso, il primo argomento rappresenta sempre una destinazione, e il secondo una sorgente.

Per esempio:

```
LD      A,B
```

trasferisce il contenuto del registro B (sorgente) nel registro A (destinazione).

```
PUSH    HL      ; operando ad un solo argomento
LD      A,41H   ; operando a due argomenti
```

Ciascuno dei due argomenti può avere a sua volta uno dei seguenti significati:

1) un nome di registro,
2) un registro puntatore,
3) un dato,
4) un indirizzo,
5) un indirizzo puntatore,
6) una condizione espressa da una parola chiave

Naturalmente, come abbiamo detto prima, il nome e il tipo degli argomenti dipendono strettamente dal tipo di operazione in cui sono coinvolti.
Prima di studiare dettagliatamente i differenti significati degli operandi, dobbiamo parlare delle espressioni.

Le espressioni

Uno degli esempi precedenti ci ha mostrato che si può scrivere:

```
LD      A,TOTO  ; perche' no?
```

se la label "TOTO" è stata definita da:

```
TOTO    EQU     4
```

Analogamente possiamo scrivere:

```
CALL    SPROG
```

dove "SPROG" rappresenta simbolicamente un indirizzo di memoria definito in un'altra parte del programma.

Ebbene, ora possiamo complicare un po' le cose scrivendo:

```
LD      A,TOTO+2
```

e

```
CALL    SPROG+12
```

Evidentemente gli effetti non saranno più gli stessi di prima, ma, nel primo caso, caricheremo il valore 6 nel registro 1, e nel secondo l'inizio del sottoprogramma non sarà più SPROG ma sarà più lontano di 12 bytes.

Possiamo anche fare:

```
CALL    SPROG + TOTO
```

non è vietato, e anche:

```
CALL    SPROG + TOTO - 2
```

basta sapere esattamente che cosa si fa.

Un'espressione - perchè è di questo che si tratta - è dunque un insieme di simboli e valori numerici, separati tra loro da operatori algebrici e/o logici.

Gli operatori algebrici li conoscete già dal BASIC. Sono:

+ per l'addizione
− per la sottrazione
* per la moltiplicazione
/ per la divisione

Bene. E gli operatori logici? Come gli operatori algebrici eseguono operazioni algebriche, così gli operatori logici.....Avete indovinato! Se vi sentite vagamente a disagio, non esitate ad "abusare" delle Appendici di questo volume.....

Gli operatori logici più comuni nell'assembler sono:

— AND (operatori logico "e")
— OR (operatore logico "o")
— NOT (operatore logico "non")
— XOR ("o" esclusivo)
— MOD (dà il resto della divisione tra due numeri interi)
— SHR (eseguono uno scorrimento a destra, "shift right"
— SHL e a sinistra, "shift left", rispettivamente, secondo il numero di bit specificato)
— HIGH (eseguono l'estrazione del byte alto "high" o basso
— LOW "low" da un numero di 16 bit).

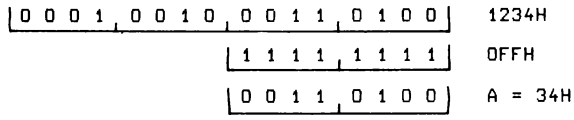
Notiamo che questi due ultimi operatori possono essere sostituiti da una combinazione di SHR e di AND.
Se VAL è un numero a 16 bit.

```
VAL   SHR 8      ---> HIGH
VAL   AND 0FFH   ---> LOW
```

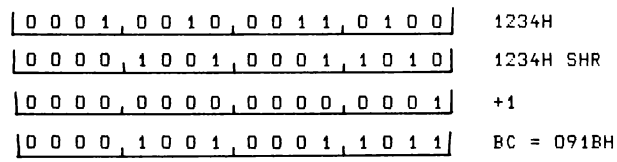
Esempi di impiego degli operatori logici

```
VAL1    EQU     1234H
MASK    EQU     0FBH
        LD      A,VAL1 AND 0FFH
        LD      BC,VAL1 SHR 1+1
        LD      DE,VAL1 OR MASK
```

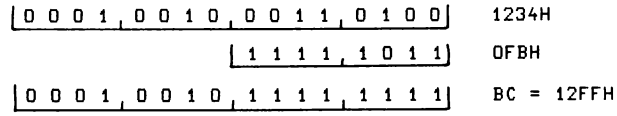
Dopo l'esecuzione delle precedenti istruzioni il registro A conterrà:



Il registro BC conterrà:



Il registro DE conterrà:



I SIGNIFICATI DEGLI ARGOMENTI DELL'OPERANDO

Vediamo ora i diversi significati che l'argomento dell'operando può assumere.

1. Un nome di registro

Nessun problema, in questo caso. Sarà uno dei registri (semplici o doppi) riconosciuti dal microprocessore:

A, B, C, D, E, H, L, I, R — per quelli semplici
AF, BC, DE, HL, SP, IX, IY, — per quelli doppi.

Notiamo che il registro F non può essere utilizzato come registro semplice, ma deve essere trattato com byte basso del registro doppio AF.

Esempi:

```
PUSH    HL
LD      A,3
LD      B,A
```

2. Un registro puntatore

È formato dal nome di un registro doppio, messo tra parentesi:

(BC), (DE), (HL), (SP), (IX), (IY).

AF non compare nell'elenco precedente per il seguente motivo: questi registri doppi devono contenere, in questo modo di indirizzamento, un indirizzo di memoria di 16 bit, e il registro F non può essere utilizzato per questo scopo. Il perchè lo vedremo più avanti.
Quello che ci interessa in un registro puntatore non è propriamente il contenuto del doppio registro, ma il dato che si trova in memoria nell'indirizzo puntato da questo doppio registro. Così, dopo l'esecuzione di:

```
LD      HL,1234H    ; carica l'indirizzo in HL
LD      A,(HL)      ; lettura tramite il registro puntatore
```

il registro A conterrà il byte di memoria che si trova all'indirizzo 1234H.
Per quanto riguarda IX e IY le cose sono un po' più complesse, ma niente paura! Ne verremo a capo ugualmente!

Questi registri sono utilizzati per il modo di indirizzamento che si chiama "**indicizzato**" (indexed), e per loro sono ammesse espressioni del genere:

(IX + 2) o (IX − VAL)

Ne riparleremo quando affronteremo i diversi modi di indirizzamento.

3. Un dato

A seconda del tipo di istruzione si tratterà di un dato a 8 o a 16 bit, nella sua forma numerica o simbolica, o anche sotto forma di un'espressione.

Esempi:

```
LD      A,2             ; forma numerica a 8 bit
LD      A,VAL           ; forma simbolica
LD      A,VAL-2         ; espressione
LD      HL,ADR          ; forma simbolica a 16 bit
```

4. Un indirizzo

Poichè si tratta di un indirizzo di memoria, sarà sempre di 16 bit e quindi dovrà essere contenuto in un registro doppio.

Valgono le stesse considerazioni già fatte per i dati a 16 bit:

```
LD      BC,ADRESS
LD      SP,PILA+VAL-2
LD      HL,1234H+2
```

5. Un indirizzo puntatore

Come con il registro puntatore si accede ad un byte di memoria tramite un registro doppio, vi si può più direttamente accedere mediante il suo stesso indirizzo di memoria, in forma numerica, simbolica, o con un'espressione:

```
LD      A,(1234H)
```

carica nel registro A il byte che si trova all'indirizzo di memoria 1234H.

```
LD      BC,(SPROG+2)
```

carica nel registro doppio BC i due byte che si trovano rispettivamente agli indirizzi di memoria SPROG + 2, SPROG + 3.

Attenzione però! Non tutto è permesso! L'istruzione:

```
LD      B,(...)
```

non esiste.....!

6. Una condizione espressa da una parola chiave

Niente di complicato, tranquillizzatevi!

Sapete che in BASIC è possibile eseguire dei salti "condizionati" dal verificarsi di certe "condizioni":

```
IF A < 0 THEN GOTO ...
IF B >= 4 THEN GOSUB ...
```

Le istruzioni macchina del microprocessore possono analogamente "testare" l'assenza o la presenza di certe condizioni che si manifestano tramite degli **indicatori** (**flags** = bandiere in inglese).
Nello Z80 questi indicatori sono i bit del registro F (F come Flag) e possono essere esaminati in qualsiasi istante.
Ad ogni condizione corrisponde una parola chiave riconosciuta dall'assemblatore, che compare sempre come primo argomento dell'operando nelle istruzioni di salto condizionato.

Queste parole chiave sono:

NZ	non zero
Z	zero
NC	non carry
C	carry
PO	parità dispari (odd)
PE	parità pari (even)
P	positivo (più)
N	negativo (meno)

Esempi:

```
JP      Z,SPROG
LD      A,1
....    ...
```

significa: se il flag Z (zero) è "settato" (= 1) il programma deve proseguire (istruzione JP = Jump, salto) a partire dall'indirizzo SPROG, altrimenti (Z = O) si esegue l'istruzione successiva (LD A, 1).
Vedremo in seguito il significato di questi diversi Flag, ve lo prometto!

DEFINIZIONE DEI DATI

Torniamo ai dati, e vediamo come possono essere definiti in assembler.

Un dato, (costante o indirizzo) può essere specificato:

1. *In notazione decimale con segno*

```
LD      HL,64200
LD      A,255
LD      B,-3
```

2. *In notazione esadecimale*

```
LD      A,0FFH
LD      HL,BA00H
```

Notiamo la presenza dello 0 davanti alla lettera esadecimale, per indicare all'assemblatore che non si tratta di una label di nome FFH (perchè no?), ma di un valore numerico. In effetti le label non iniziano mai con una cifra.

3. *In notazione ottale* (base 8, meno utilizzata):

```
LD      A,64O
```

o, per maggior chiarezza (la lettera O si confonde spesso con lo zero):

```
LD      A,64Q
```

Qualche anno fa il sistema di numerazione ottale (base 8) era di uso corrente sugli elaboratori dell'epoca. Ma tutto cambia, e adesso il sistema esadecimale (base 16) ne ha preso il posto. Ciononostante ci sono ancora alcuni elaboratori (e alcuni programmatori) che insistono a utilizzare il vecchio sistema.
I suffissi O e Q sono per loro.

4. *In notazione binaria*

```
LD      A,11000101B
```

5. *Sotto forma di carattere ASCII*

```
LD      A,'A'
LD      B,'*'
```

che è più eloquente di:

```
LD      A,41H
LD      B,2AH
```

6. *Sotto forma di un simbolo o di una label:*

```
LD      A,VAL
CALL    SPROG
```

che, ricordiamocelo, devono necessariamente essere definiti nel programma.

7. *Sotto forma di un'espressione:*

```
LD      A,2*(VAL-2)+TOTO AND 4
```

Le espressioni che fanno riferimento a indirizzi possono anche utilizzare il simbolo speciale "$", che prende il valore dell'indirizzo di inizio dell'istruzione stessa. Per esempio:

```
Indirizzo           Istruzione
---------           ----------
1000H               CALL    $+6
  .                   .
  .                   .
  .                   .
1006H               LD      HL,$-2
```

La CALL provocherà un salto all'indirizzo che si ottiene sommando 6 all'indirizzo di inizio dell'istruzione corrente (1000H) cioè a 10006H. HL conterrà il valore 10006H-2, cioè 10004H.

ESEMPIO DI PROGRAMMA

Tutto questo può forse sembrare un pò confuso, ma le idee si dovrebbero chiarire dopo l'esame del listing seguente, che commenteremo linea per linea. Alcune istruzioni vi risulteranno nuove, ma non è il caso di prendere questo pretesto per abbandonare una così buona strada.....!

```
                00100 ;-------------------------------------;
                00110 ;ESEMPIO DI PROGRAMMA CHE REALIZZA UNO;
                00120 ;SCAMBIO TRA DUE ZONE DI MEMORIA      ;
                00130 ;-------------------------------------;
                00140 ;
8000            00150          ORG     8000H          ; ORIGINE
                00160 ;
                00170 ; DICHIARAZIONE DI INDIRIZZI ESTERNI
                00180 ;
9000            00190 BUFF1    EQU     9000H          ; BUFF1
90C8            00200 BUFF2    EQU     BUFF1+200      ; BUFF2
                00210 ;
                00220 ; INIZIO DEL PROGRAMMA
                00230 ;
8000 31FFFF     00240 INIZ     LD      SP,0FFFFH      ; STACK
8003 210090     00250          LD      HL,BUFF1       ;
8006 11C890     00260          LD      DE,BUFF2       ;
8009 06C8       00270          LD      B,200          ;
800B CD1C80     00280          CALL    SCAMB          ; CHIAMATA1
800E 21C890     00290          LD      HL,BUFF2       ;
8011 110090     00300          LD      DE,BUFF1       ;
8014 06C8       00310          LD      B,200          ;
8016 CD1C80     00320          CALL    SCAMB          ; CHIAMATA2
8019 C31980     00330 STOP1    JP      STOP1          ; LOOP
                00340 ;
                00350 ; ROUTINE DI SCAMBIO
                00360 ; IN INGRESSO:HL PUNTA SUL PRIMO BUFFER
                00370 ;             DE PUNTA SUL SECONDO
                00380 ;             B CONTIENE LA LUNGHEZZA
                00390 ; IN USCITA  :HL=IND BUFF1+LUNGHEZZA
                00400 ;             DE=IND BUFF2+LUNGHEZZA
                00410 ;             B = 0
                00420 ;             C MODIFICATO DA ROUTINE
                00430 ;
801C 4E         00440 SCAMB    LD      C,(HL)         ; LETTURA
801D 1A         00450          LD      A,(DE)         ; LETTURA
801E 77         00460          LD      (HL),A         ; SCRITT.
801F 79         00470          LD      A,C            ; C-->A
```

```
8020 12       00480          LD       (DE),A        ; SCRITT.
8021 23       00490          INC      HL            ; IND+1
8022 13       00500          INC      DE            ; IND+1
8023 05       00510          DEC      B             ; LUNG-1
8024 C21C80   00520          JP       NZ,SCAMB      ; SE#0
8027 C9       00530          RET                    ; SE=0
              00540 ;-------------------------------;
8000          00550          END      INIZ
00000 TOTAL ERRORS

STOP1   8019 00330    00330
BUFF1   9000 00190    00200 00250 00300
BUFF2   90C8 00200    00260 00290
INIZ    8000 00240    00550
SCAMB   801C 00440    00280 00320 00520
```

SCOPO DEL PROGRAMMA

Si tratta di scambiare tra di loro i contenuti di due zone di memoria, chiamando un'apposita routine, poi di scambiarli nuovamente, per riprodurre la configurazione iniziale, chiamando una seconda volta la stessa routine.
Questo programma può essere realizzato in modo più semplice, ma utilizzando delle istruzioni più complesse.
Cercate per ora di capirlo così com'è. Potrete migliorarlo più tardi.....
Nota: questo programma è stato realizzato con l'Editor-Assembler del TRS-80.

SPIEGAZIONE DEL PROGRAMMA, linea per linea (o quasi!)

Possiamo subito individuare con il primo colpo d'occhio due zone principali nel listing:

— Una *zona che contiene il codice sorgente*, così come è stato introdotto per mezzo dell'editor. In genere, quest'ultimo numera automaticamente ogni linea sorgente, al momento della sua introduzione (100, 110,.....)
— Una *zona generata dall'assemblatore* (a sinistra) che comprende due colonne: una per gli indirizzi, l'altra per il codice macchina generato

LINEE da 100 a 140
Sono linee di commento, che cominciano con il carattere speciale ";".

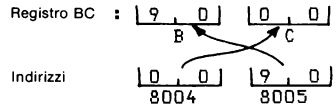
LINEA 150
Questa pseudo-istruzione dell'assembler è utilizzata per fissare l'indirizzo di origine del programma in memoria (là dove poi sarà memorizzato). Nell'esem-

pio, l'indirizzo è 8000H, e vediamo infatti che la prima istruzione del programma (LD SP,.....) è proprio a questo indirizzo.

LINEE 190 e 200
Poichè le due zone di memoria utilizzate (**buffers**) sono esterne al programma, è necessario definire i loro indirizzi per mezzo della pseudo-istruzione già vista: EQU. Il secondo buffer può essere definito relativamente al primo, come si vede alla linea 200, specificando che il suo indirizzo di inizio sarà uguale allo indirizzo di inizio del primo buffer, aumentato della sua lunghezza. A sinistra della linea possiamo constatare, non senza una certa emozione....che l'assemblatore ha capito, e che ha generato l'indirizzo 90C8H, che è proprio uguale a 9000H + 200. Se, in seguito, si vuole cambiare l'indirizzo dei due buffer, basterà modificare la linea 190. Notare infine che queste pseudo-istruzioni non generano codice macchina.

LINEA 240
È la prima istruzione del programma che si presenta in tutto il suo splendore, con un campo label (DEBUT), un campo istruzione (LD), un campo operando (SP, OFFFFH) e un campo commento (;stack).
Essa afferma che il registro puntatore dello stack "punterà" alla fine della memoria (ultimo indirizzo = FFFF), cosa necessaria poichè noi in seguito utilizzeremo lo stack con le istruzioni CALL. A sinistra della linea l'assemblatore ha generato l'indirizzo di memoria (8000H) che è sempre in esadecimale, e il codice macchina corrispondente all'istruzione (31FFFF). Questa istruzione occupa tre bytes, quindi l'indirizzo di memoria dell'istruzione seguente sarà 8003H.
Possiamo "indovinare" che il codice macchina contiene la codifica dell'istruzione (31) e un operando (FFFF). Vi rimandiamo alle Appendici per i codici delle diverse istruzioni.

LINEA 250
Il registro doppio HL è caricato con l'indirizzo di inizio del primo buffer. Il codice macchina generato è formato dal codice istruzione (21) e dall'operando (9000) che corrisponde all'indirizzo definito più sopra da una EQU. Stranamente osserviamo che questo valore è stato "invertito" nel codice macchina generato dall'assemblatore. È normale: il registro "basso" C è caricato con il contenuto dell'indirizzo "alto" 8005H, che contiene proprio 90H.....



il ";" del campo commento c'è solo per bellezza!

LINEA 260
Stessa operazione, ma con l'indirizzo di inizio del secondo buffer, che sarà caricato nel registro DE. Notiamo che queste ultime due istruzioni avrebbero potuto essere scritte:

```
LD      HL,9000H
LD      DE,90C8H
```

ma per le stesse ragioni dette prima, se si decide di spostare i buffer, bisognerebbe modificare tutti gli operandi relativi a questi indirizzi, mentre nel nostro caso basta modificare una sola linea, la 190. E poi, questo ci evita anche di calcolare la soma 90000H + 200!

LINEA 270
La lunghezza dei buffer (200 bytes) è caricata nel registro B. Avremmo anche potuto scrivere:

```
LD      B,LONG
```

con

```
LONG    EQU     200       ; lunghezza = 200 byte
```

oppure:

```
BUFF2   EQU     BUFF1 + LONG
```

Questo tipo di istruzione occupa due bytes, come si vede nella zona del codice macchina generato dall'assemblatore.

LINEA 280
È l'istruzione CALL, che ormai conosciamo bene. Possiamo osservare che l'assemblatore ha giustamente generato, nel codice macchina, l'indirizzo effettivo corrispondente alla label PERMUT, cioè 801CH (invertito....). Non è stupido, no?

LINEE da 290 a 320
Stessa sequenza di prima, ma il contenuto dei due buffer è nuovamente scambiato. Avremmo potuto scrivere:

```
LD      HL,BUFF1
LD      HL,BUFF2
```

che non avrebbe cambiato niente.

LINEA 330
Qui entriamo in un circolo vizioso! Quando il microprocessore esegue questa istruzione, salta (JP = JUMP) all'indirizzo indicato dall'operando, che, in questo caso, è l'indirizzo di questa stessa istruzione. Avremmo anche potuto scrivere:

```
JP      $
```

ricordate?

Affrontiamo ora la routine SCAMB, che viene chiamata due volte dal programma principale. Senza le istruzioni CALL/RET, avremmo dovuto scrivere due volte questa sequenza di istruzioni.....!

LINEA 440
È il punto di "ingresso" della routine. L'istruzione LD C,(HL) ha un operando del tipo "registro puntatore" e corrisponde ad un solo byte del codice macchina. Il registro C (primo argomento dell'operando = destinazione) è caricato con il byte che si trova all'indirizzo di memoria memorizzato nel doppio registro HL (secondo argomento = sorgente). Questo byte appartiene al buffer 1.

LINEA 450
Il byte del buffer 2 è, a sua volta, caricato nel registro A.....

LINEA 460
.....ed è poi scritto nella locazione del buffer 1 puntata da HL.
Notate il significato degli argomenti: (HL) = primo argomento = destinazione, A = secondo argomento = sorgente.

LINEA 470
Poichè l'istruzione

```
LD      (DE),C
```

non esiste, bisogna prenderne atto, e trasferire il contenuto del registro C nel registro A.....

LINEA 480
....e poi scriverlo nella locazione del buffer 2 puntato da DE.

LINEE 490 e 500
Queste due nuove istruzioni sommano 1 al contenuto del registro doppio specificato nell'operando (incremento). Quindi HL punterà il byte successivo del buffer 2.

LINEA 510
Qui si effettua l'operazione inversa (decremento) sul registro B, che inizialmente conteneva 200, e dopo l'esecuzione di questa istruzione conterrà 199.

LINEE 520 e 530
Continuando a decrementare il registro B, questo finirà per contenere un valore nullo. In questo caso, la "bandiera" costituita dall'indicatore Z sarà "issata" dal microprocessore. Lo scopo dell'istruzione JP NZ è proprio quello di verificare la presenza di questo indicatore: se il registro B è = da zero (NZ), il programma salta all'indirizzo indicato nel secondo argomento dell'operando (nel nostro caso è l'inzio della routine SCAMB). In caso contrario (tutto il buffer è stato trasferito) l'esecuzione prosegue con l'istruzione successiva, che nel nostro caso è una RET. Si ritorna dunque al programma principale.

LINEA 550
Fine del programma, segnalata dalla pseudo-istruzione END (che non genera alcun codice macchina) che precisa inoltre che il programma dovrà essere lanciato dall'indirizzo INIZ.

A questo punto è necessario chiarirci un particolare. C'è una piccola differenza tra il codice macchina e il codice oggetto.

Il **codice macchina** è quello che compare nella seconda colonna a sinistra del listing, e corrisponde alle istruzioni e ai dati trattati dal microprocessore. È questo il codice che sarà integralmente trasferito in memoria al momento del caricamente del programma.

Il **codice oggetto** è il codice generato dall'assemblatore. Questo codice generalmente viene trasmesso ad una periferica (registratore, disco) che ne assicura la conservazione (memorizzazione). Naturalmente il codice oggetto contiene il codice machina, ma anche altre informazioni relative alla sistemazione in memoria del programma, alla sua lunghezza, e al suo indirizzo di lancio, definito proprio dalla pseudo-istruzione END.
Queste informazioni non sono caricate in memoria con il codice macchina, ma sono destinate a informare il programma di caricamento detto **caricatore** (loader).

In effetti, l'esame del codice macchina da solo non permette di dire, a priori, quali indirizzi di memoria il programma deve occupare, e quale deve essere la

prima istruzione da eseguire (che non è necessariamente nella prima posizione del codice macchina).
Vedremo questo cose un più avanti.....

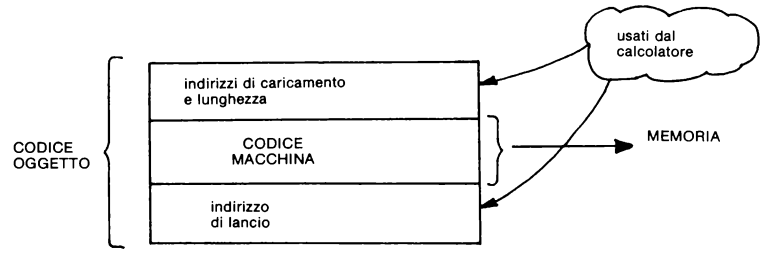


Figura 9

Torniamo al nostro listing.

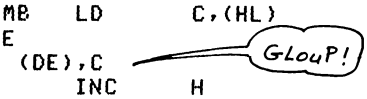
Dopo averci segnalato che il programma presenta in totale "0 ERRORI", l'assemblatore fornisce una lista delle label trovate, con gli indirizzi corrispondenti, i numeri di linea corrispondenti, e i numeri delle linee in cui vengono chiamate.

Vediamo, per esempio, che la label BUFF2 corrisponde all'indirizzo 90C8H, è definita alla linea 200 ed è utilizzata alle linee 260 e 290. Cosa possiamo chiedere di più?

Ecco un altro listing dello stesso programma (o quasi...), nel quale si sono infilati alcuni errori. A voi scoprirne le cause.

```
                00100 ;-----------------------------------------;
                00110 ; ECCO UN ESEMPIO DI PROGRAMMA DA NON  ;
                00120 ; IMITARE...E NON PER PROBLEMI DI      ;
                00130 ; COPYRIGHT...                         ;
                00140 ;-----------------------------------------;
UNDEFINED SYMBOL
0000            00160           ORG     FFFFH
BAD LABEL                                          Aïe!
00170 DICHIARAZIONI DI INDIRIZZI ESTERNI
2328            00180 BUFF1     EQU     9000
UNDEFINED SYMBOL
00C8            00190 BUFF2     EQU     BUFF+200
                00200 ;
                00210 ; INIZIO DEL PROGRAMMA
ILLEGAL OPCODE                                     Houpss...!
00220 INIZ      LOAD    SP,OFFFFH
ILLEGAL ADDRESSING MODE
00230           LD      HD,BUFF1
0000 11C800     00240           LD      DE,BUFF2
```

```
FIELD OVERFLOW
0003 06D0     00250          LD       B,2000
ILLEGAL OPCODE
00260          COLL      SCAMB
0005 21C800   00270          LD       HL,BUFF2
0008 112823   00280          LD       DE,BUFF1
ILLEGAL ADDRESSING MODE
00290          LD        B,200
000B CD1100   00300          CALL     SCAMB
000E C30E00   00310          JP       $
              00320 ;
              00330 ; ROUTINE
BAD LABEL
00340 ,
0011 4E       00350 SCAMB    LD       C,(HL)
ILLEGAL ADDRESSING MODE
00380          LD          (DE),C
0012 24       00390          INC      H
0013 C9       00400          RET
NO END STATEMENT
00011 TOTAL ERRORS
```




```
BUFF    0000 UNDEF***00190
BUFF1   2328 00180    00230 00280
BUFF2   00C8 00190    00240 00270
FFFFH   0000 UNDEF***00160 00220
SCAMB   0011 00350    00260 00300
```

LE ISTRUZIONI NECESSARIE ALL'ASSEMBLATORE

Prima di lanciarci nella scrittura di un programma in assembler, ci sembra utile conoscere le funzioni che possono essere realizzate dalle istruzioni del microprocessore. È necessario un minimo, anche se lo Z80 non è tanto "giusto" da questo punto di vista!

Per far questo non ci sono segreti: bisogna consultare la tabella dele istruzioni fornita dalla casa costruttrice del microprocessore, e cercare di memorizzarle. Con un pò di pratica questo viene da sè.
Cerchiamo, comunque, di analizzare le diverse funzioni.

Prima di tutto è necessario poter comunicare con la memoria, questo è il minimo! Questa comunicazione si stabilirà tra una parola di memoria e un registro, per mezzo dell'istruzione LD (LOAD = caricare), che permette la comunicazione in entrambi i sensi:

registro ----> memoria (scrittura in memoria)
memoria ----> registro (lettura in memoria)

La comunicazione è spesso necessaria anche tra gli stessi registri. L'istruzione LD consente anche questo.
È necessario anche avere a disposizione un certo numero di funzioni aritmetiche. Per tutti i microprocessori a 8 bit attualmente in circolazione esistono come minimo: l'addizione e la sottrazione. Questo è tutto.

"*Scusi*?....direte voi..."*ho capito male*?"

NO. È dura, ma bisogna arrendersi all'evidenza. Se si vogliono eseguire moltiplicazioni o divisioni in linguaggio macchina....bisogna "programmarsele". Tranquillizzatevi, però: esistono eccellenti routine previste per questo....
Le istruzioni di confronto possono anche essere classificate in questo gruppo. In genere esse confrontano sempre qualche cosa con il registro A. Vedremo come si fa.
Devono esserci anche istruzioni che realizzano le operazioni logiche AND, OR, OR ESCLUSIVO, NOT, molto utilizzati nei programmi assembler. Ve ne renderete conto!

Le istruzioni di comando e di controllo interne, che agiscono direttamente su alcune funzioni caratteristiche del microprocessore, possono avere anche loro una certa utilità.

Le istruzioni di "interruzione di sequenza", o istruzioni di salto, (condizionale o no),sono assolutamente indispensabili. E in queste includiamo le istruzioni di chiamata e ritorno da sottoprogrammi.

Le istruzioni di ingresso/uscita (INPUT/OUTPUT) permettono la comunicazione con le periferiche. Senza di esse, non sarebbe possibile alcuna comunicazione con il mondo esterno. Sarebbe impossibile programmare un elaboratore, e voi non vi trovereste a leggere questo libro.....

Le istruzioni che riguardano lo stack non sono indispensabili ma contribuiscono largamente all'efficienza di un programma.

Le istruzioni speciali che operano sui bit o sulle sequenze di caratteri, pur non essendo indispensabili, sono gradite quando ci sono.

Ma non contano soltanto le istruzioni? Anche il modo di servirsene ha la sua importanza.

I MODI DI INDIRIZZAMENTO

Che cosa esattamente si intende con "modo di indirizzamento?" Possiamo dire che si tratta semplicemente dei diversi modi di accedere ad un dato.

I modi di indirizzamento giocano un ruolo molto importante per quanto riguarda la potenza, la flessibilità, la maneggevolezza, l'efficienza del set di istruzioni di un elaboratore. Essi facilitano e semplificano la scrittura del programma, migliorandone la sua rapidità di esecuzione.

Lo Z80 ha 7 modi di indirizzamento, di cui ci occuperemo adesso.

Indirizzamento immediato

In questo modo, il valore da caricare si trova nella zona dello operando dell'istruzione, ed è quindi *immediatamente* accessibile.

```
LD      A,1
```

Indirizzamento tramite registri

Nell'ipotesi che il valore da caricare si trovi già, per esempio, nel registro B, utilizzeremo questo modo di indirizzamento eseguendo l'istruzione:

```
LD      A,B
```

Indirizzamento indiretto tramite registri

Si tratta del registro puntatore già citato. Se il valore 1 si trova all'indirizzo di memoria 1234H, e il registro doppio HL contiene questo indirizzo (LD HL,1234H), l'istruzione:

```
LD      A,(HL)
```

caricherà il valore 1 nel registro A, passando indirettamente attraverso il registro HL.

Indirizzamento diretto

Riprendendo l'esempio precedente, facendo:

```
LD      A,(1234H)
```

caricheremo direttamente il valore 1 nel registro A. Da cui il nome di indirizzamento diretto.

Indirizzamento relativo

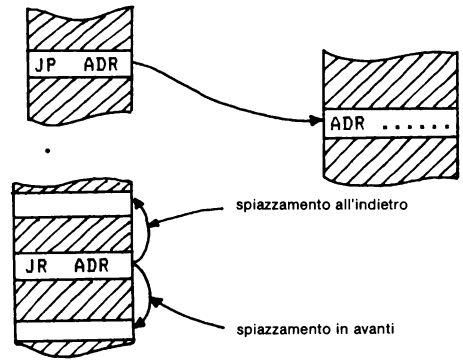
Ecco un modo di indirizzamento di cui non abbiamo ancora parlato. È utilizzato solamente dalle istruzioni di salto.

Il principio di funzionamento è molto semplice. Abbiamo visto nel listing del programma sullo scambio di zone di memoria, che l'istruzione JP NZ occupava tre bytes del codice macchina: 1 byte per il codice istruzione e due byte per l'argomento dell'operando che contiene un indirizzo a 16 bit.

Avremmo potuto sostituire questa istruzione di salto con:

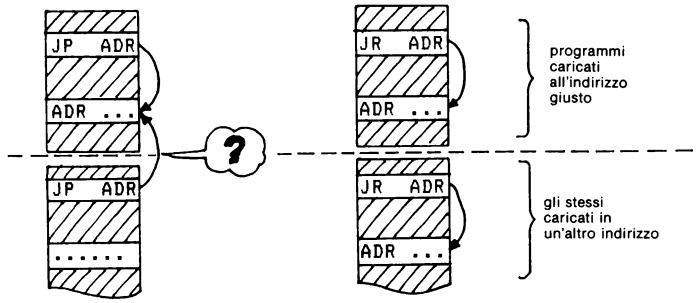
```
JR      NZ,SCAMB
```

e, in questo caso, il codice generato avrebbe occupato solo due byte, uno per il codice operazione (che non sarebbe stato lo stesso naturalmente) e l'altro che avrebbe rappresentato non un indirizzo, ma uno "spiazzamento" relativo. In effetti esistono due modi per "saltare":



In quest'ultimo caso, l'operando generato nel codice macchina dall'assemblatore è uguale alla differenza tra l'indirizzo da raggiungere e l'indirizzo attuale. C'è un limite: poichè lo spiazzamento occupa un byte, non potrà essere maggiore di +127 e minore di −128.

Ci sono quindi dei vantaggi a utilizzare i salti relativi quando questo è possibile, tanto più che il programma è in questo modo reso rilocabile, cioè può essere eseguito senza problemi in diverse zone di memoria, a condizione però di usare soltanto salti relativi all'interno del programma, cosa che esclude d'ufficio l'istruzione CALL, che non ha purtroppo l'equivalente con indirizzamento relativo.



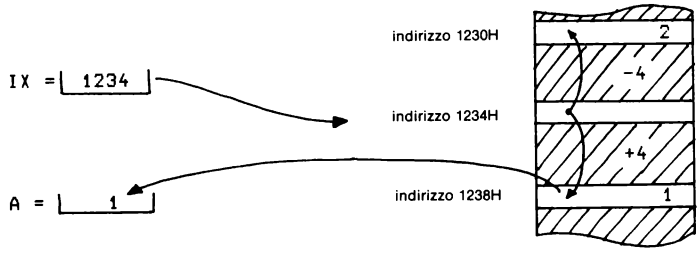
Indirizzamento indicizzato

In realtà si tratta più precisamente di un indirizzamento pseudo-indicizzato. Vedremo più in là che cosa conviene chiamare indirizzamento indicizzato.

Qui si tratta di accedere a un dato tramite uno spiazzamento - positivo o negativo - relativo ad un indirizzo contenuto in uno dei registri IX e IY riservati per questo indirizzamento.

```
LD      A,(IX+4)
```

In questo esempio, il registro A sarà caricato con il byte di memoria che si trova all'indirizzo memorizzato nel registro IX, aumentato di 4:



e se avessimo fatto:

```
LD      A,(IX-4)
```

il valore 2 sarebbe stato caricato in A.

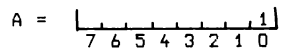
Come per i salti relativi, lo spiazzamento è limitato a 127 byte in avanti e 128 byte indietro.

Indirizzamento al bit

Questo interessante modo permette di leggere, scrivere, testare non più un byte, ma un bit ben preciso di questo byte, che potrà essere sia in un registro che in memoria.

```
SET     0,A
```

Posiziona al valore "1" il bit numero 0 del registro A



mentre gli altri bit naturalmente non sono modificati.

ALTRI MODI DI INDIRIZZAMENTO

Sì, esistono altri modi di indirizzamento, che si trovano però solo sui "mini" e sui "grossi" elaboratori. Tra i più comuni citeremo rapidamente:

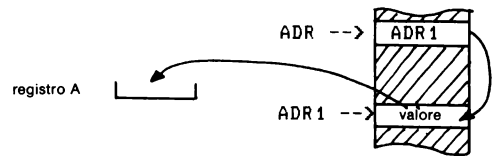
Indirizzamento indiretto in memoria

Conosciamo già l'indirizzamento indiretto tramite registri, ma in questo caso l'indirizzo, invece di essere memorizzato in un registro, si trova in memoria.

Per esempio, il segno "@" segnale l'operazione di indirizzamento indiretto:

```
LDA     @ADR
```

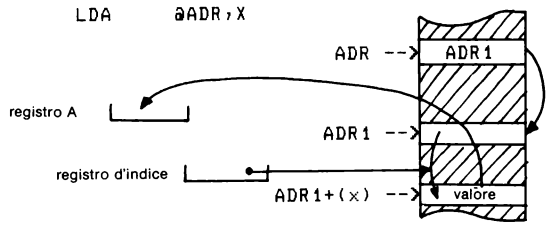
caricherà nel registro A non il dato che si trova all'indirizzo ADR, ma quello che si trova all'indirizzo contenuto in ADR. Cogliete la finezza? Si tratta di un indirizzamento doppio con un albero a camme in testa.



Indirizzamento indiretto indicizzato

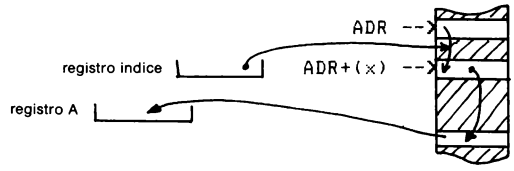
Il principio è lo stesso di quello di prima, ma all'indirizzo ottenuto si aggiunge il contenuto del registro indice (X, per esempio).

```
LDA     @ADR,X
```



In questo modo bisogna distinguere l'indirizzamento post-indicizzato da quello pre-indicizzato.
Quello descritto prima è l'indirizzamento post-indicizzato, poichè si aggiunge il contenuto del registro indice come ultima operazione.

Nell'indirizzamento pre-indicizzato questa operazione si fa prima.



Indirizzamento con base

Questo modo di indirizzamento consente la totale rilocabilità dei programmi, come l'abbiamo già vista nell'indirizzamento relativo (ma senza i limiti di quel caso....).

Per poter utilizzare questo modo, l'elaboratore (il microprocessore) deve essere munito di un registro di base, il cui contenuto è sistematicamente sommato agli indirizzi di memoria citati nel programma.
Naturalmente il set di istruzioni deve conoscere questo registro.

```
LDA     B.ADR
JMP     B.INIZIO
```

Con B = 0 è il modo normale che già conosciamo. Ma ci sono altre 65635 possibilità di "base" ....(per un registro di base a 16 bit).

RUOLO DEI REGISTRI

Passeremo ora in rivista i diversi registri dello Z80, parlando un pò delle funzioni che li coinvolgono, perchè alcuni di loro, e lo vedremo tra breve, sono specializzati, e non si può certo utilizzarli in tutte le salse.

Il registro A

Più spesso denominato con il dolce nome di "accumulatore", il registro A è il più importante del microprocessore o dell'elaboratore.

È lui che "accumula" la maggior parte dei compiti nel funzionamento intimo del microprocessore. In compenso è l'unico che accetta tutti i modi di indirizzamento (tranne naturalmente quello relativo, che si applica solo alle istruzioni di salto).
Tutte le operazioni aritmetiche, logiche e di confronto lo usano come intermediario. La sua capacità è di 8 litri.....scusate: di 8 bit!

Il registro F

Abbiamo già visto che il registro F è il "porta bandiera" del microprocessore. Lo esamineremo in dettaglio più avanti.

Il registro B

Questo registro sarebbe relativamente banale se non fosse utilizzato in modo particolare in un'istruzione di loop DJNZ. Può essere associato con il registro C per formare un registro doppio.

I registri C, D, E, H, L.

Sono semplici registri a 8 bit che possono essere uniti a coppie: BC, DE, HL.HL ha peraltro una caratteristica particolare: accetta, come A, pur essendo di 16 bit, tutti i modi di indirizzamento, tranne il relativo e l'indicizzato, e ha spesso il ruolo di puntatore della memoria, a tal punto che la cosa costruttrice INTEL lo chiama M nel suo assembler dello 8080 o dello 8085:

```
MOV     A,M
```

corrisponde a

```
LD      A,(HL)
```

e significa: sposta (MOVE = spostare) il byte di memoria puntato da M (registro doppio HL) nel registro A.

Il registro SP

Abbiamo già visto che si tratta di un registro a 16 bit utilizzato come puntatore dello stack.

I registri IX e IY

Sono anche essi registri a 16 bit di uso generale, ma il cui utilizzo è limitato al modo di indirizzamento indicizzato.

Il registro I

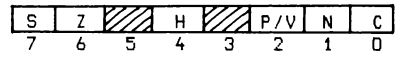
È un registro a 8 bit usato in un modo di funzionamento particolare dello Z80 legato alle interruzioni. Esso contiene il byte alto di un indirizzo al quale deve saltare il programma, nel caso di un'interruzione inviata da una periferica. Sarà compito di quest'ultima fornire il byte basso di questo indirizzo.

Il registro R

Come il precedente ha un utilizzo molto particolare di tipo tecnico. Non è mai utilizzato nella programmazione, e contiene un indirizzo in continua evoluzione destinato ad assicurare il "rinfresco" dinamico dei blocchi di memoria.
Non ci dilunghiamo sui registri del secondo blocco (A', F', B', C', D', E', H', L') che , se scambiati con il blocco principale, si comportano esattamente come i registri di quest'ultimo.
C'è un registro di cui non abbiamo parlato finora, in quanto non è direttamente accessibile dal programma. Ciononostante esiste e ha un ruolo fondamentale nell'esecuzione del programma.
È il registro a 16 bit chiamato PC, contatore di programma (Program Counter). Esso contiene l'indirizzo dell'istruzione che sta per essere eseguita.
Nel caso di un'istruzione JP, per esempio, il microprocessore scrive nel registro PC l'indirizzo indicato con l'operando di questa istruzione, facendo si che l'esecuzione del programma prosegua a partire da questa nuova locazione di memoria.

RUOLO DEGLI INDICATORI

Gli indicatori o "flag" sono raggruppati come abbiamo già visto, nel registro F, che ha la seguente struttura:



IL BIT 0

Contiene il flag di CARRY (riporto). È posizionato dalle operazioni aritmetiche, di shift e di confronto, nonchè da due istruzioni speciali che permettono di

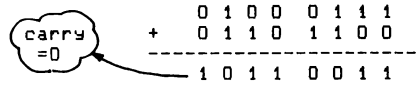
forzare il suo valore a 1 e di complementarlo.
Le istruzioni logiche (AND, OR, XOR) lo posizionano sempre a 0.

Esaminiamo ora quattro casi di utilizzo del bit C: l'addizione, la sottrazione, il confronto e lo shift.

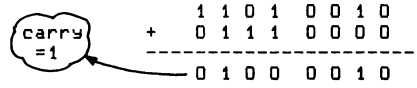
L'addizione

In questa operazione il bit del CARRY assume il valore del riporto generato dalla somma dei due bit più significativi.

Esempio di addizione a 8 bit:

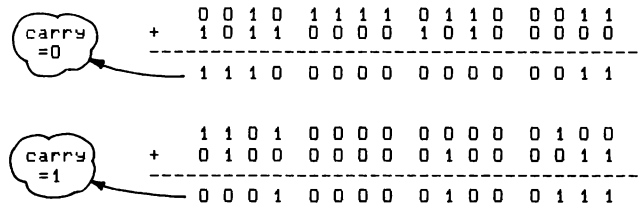


Vediamo ora un altro caso:



nel carry compare un riporto, poichè il risultato corretto sarebbe 142H.

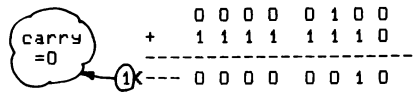
Esempio di addizione a 16 bit:



La sottrazione

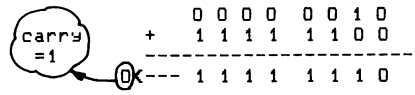
Il significato del bit C è esattamente lo stesso che per l'addizione, poichè la sottrazione non è altro che una *addizione in complemento a due* (vedi Appendice), e in questo caso il senso del riporto deve essere invertito.

Supponiamo di dover eseguire la sottrazione 4 - 2 in 8 bit. Il complemento a due di -2 e 0FEH, quindi questo equivale a fare 4 + 0FEH:



Troviamo come risultato 2, che è corretto.

Eseguiamo ora 2 - 4. Il complemento a due di -4 è 0FCH:



Vediamo subito che il risultato è un complemento a due che equivale a -2, ed è proprio quello che ci aspettavamo.

Il confronto

Quando il micro-processore confronta due valori, esegue di fatto una sottrazione invisibile tra di loro, che porta alla seguente interpretazione:

primo valore > o = al secondo valore: CARRY = 0
primo valore < del secondo valore : CARRY = 1

Nelle istruzioni di confronto il primo valore è sempre immagazzinato nell'accumulatore, e il secondo valore è definito nell'operando dell'istruzione, a 8 bit.

NOTA: *I valori confrontati sono sempre considerati valori assoluti, cioè senza segno* (F0H ad esempio deve essere considerato maggiore di 2)

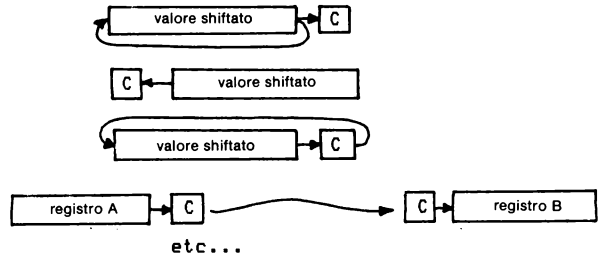
(A)	(OPERANDE)	(CARRY)
5	4	0
5	7	1
5	5	0
5	-2 (=FE)	1
-1 (=FF)	5	0

Quando i valori negativi sono sostituiti dal loro complemento a due, l'interruzione del CARRY diventa in effetti più chiara.

Lo shift

In questo caso, il Carry può essere considerato come un bit di estensione del valore da shiftare, in un certo senso un nono bit.

Secondo il tipo di istruzione (shift sinistro, destro, aperto, chiuso), esso partecipa o meno allo shift ed è spesso utilizzato per spostare una informazione da un posto ad un altro.



Le istruzioni di shift saranno esaminate in dettaglio nel capitolo seguente.

IL BIT 1

È l'indicatore **N** (negativo). È posizionato a 1 quando l'operazione precedente è stata una sottrazione o un decremento. Si utilizza abbastanza raramente.

IL BIT 2

È l'indicatore di Parità/Overflow **P/V** (overflow = trabocco). Questo bit ha un ruolo doppio, a seconda del tipo di istruzione che ha causato il suo posizionamento: è un indicatore di parità per le istruzioni logiche (numero di bit a 1 pari o dispari) e segnala un overflow (o superamento della capacità di parola) nell'esecuzione delle istruzioni aritmetiche.

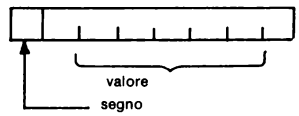
Dopo un'operazione logica:

Se A = 0 1 1 0 1 0 1 1 , P = 0 (dispari)

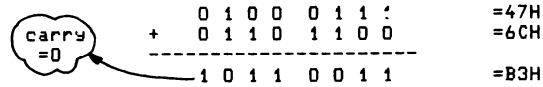
se A = 0 1 1 0 1 0 1 0 , P = 1 (pari)

Dopo un'operazione aritmetica, l'indicatore V è posizionato a 1 se il risultato dell'operazione è scorretto.

È necessario segnalare qui che l'aritmetica degli elaboratori considera i numeri a 8 bit come numeri con segno, cioè compresi tra -128 e + 127, e il bit più significativo fa le veci del segno (1 = valore negativo)

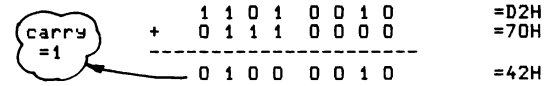


Riprendiamo uno degli esempi precedenti:



Da quando in qua l'addizione di due numeri positivi dà un risultato negativo? D'altra parte è proprio quello che capita in questo esempio, poichè B3H in effetti è -4DH in complemento a due...
Questa anomalia è segnalata dall'indicatore V che assume il valore 1.

L'esempio successivo ci dava:



e avevamo detto che il risultato corretto sarebbe stato 142H. Ma riflettiamo un pò. Se consideriamo che D2H è in effetti equivalente a -2EH in complemento a due, allora:

- 2EH + 70H = 42H

e il risultato in realtà è corretto. L'indicatore V assumerà quindi il valore 0.

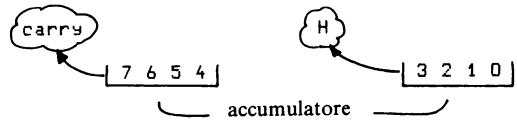
Si può riassumere la situazione dicendo che l'indicatore V assume il valore 1 quando si verifica una situazione anomala nel segno del risultato.

IL BIT 4

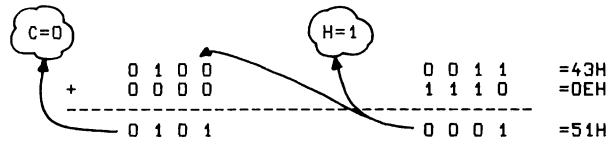
È l'indicatore H (half-carry). Tranquillizzatevi immediatamente, non dovrete utilizzare molto spesso questo indicatore!

È riservato all'esecuzione di un'istruzione molto particolare (DAA), che ha la funzione di trasformare un valore binario di 8 bit nel suo equivalente codificato in BCD (vedi Appendice).

È un "mezzo-carry", da cui il suo nome... In effetti dà le stesse indicazioni del bit di carry, ma su un valore di solo quattro bit, e rappresenta il riporto che si verifica non al bit 7, ma al bit 3 dell'accumulatore, considerando questo formato da due "quartine" (nibble) di quattro bit:

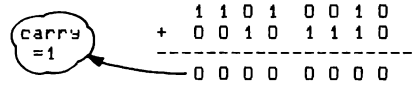


Esempio:



IL BIT 6

È l'indicatore **Z** (zero). È posizionato a 1 da alcune istruzioni quando danno un risultato nullo. Per esempio l'addizione:



posizionerà il bit Z a 1, indicando così che il risultato dell'operazione è 0.

IL BIT 7

È l'indicatore S del segno. È messo a 1 da alcune istruzioni e indica il segno del valore sotto test, o generato, come abbiamo visto prima:

S = 1, valore negativo
S = 0, valore positivo (da 0 a 127)

e corrisponde al bit più significativo di questo valore.

Adesso dovreste saperne abbastanza per affrontare un set di istruzioni "tipo", come quello dello Z80, che è fino a questo momento, il più completo. All'arrembaggio...!
(Non dimenticate di issare tutte le bandiere!)

ESERCITAZIONI PRATICHE DEL CAPITOLO 2

Eccoci arrivati alla fine del Capitolo II, e voi dovreste adesso saperne abbastanza da poter risolvere i semplici esercizi che vi proponiamo qui.

Non cercate troppe complicazioni, sono molto semplici, e anche se non riuscite per ora a venirne a capo, non è grave. La cosa principale è di tentare e fare in pratica! È il segreto per riuscire.

In Appendice vengono date le soluzioni di questi esercizi, ma cercate di non consultarle, se non come estrema risorsa!

ESERCIZIO 2.1: *Sommare due numeri compresi tra 1 e 127, disponibili nei registri B e C. Il risultato dovrà essere fornito nel registro B. Naturalmente non dovete usare l'istruzione di addizione!*

ESERCIZIO 2.2: *Sia dato il seguente programma:*

```
        LD      BC,0ABCH
        PUSH    BC
        CALL    ROUT
        JR      $               ; arrêt
ROUT    POP     HL
        INC     HL
        INC     HL
        PUSH    HL
        RET
```

Che cosa fa secondo voi? Sapete spiegare come funziona?

ESERCIZIO 2.3: *E cosa pensate di quest'altro?*

```
        LD      A,3DH
        LD      (ADR),A
ADR     INC     A
        INC     A
```

Che cosa conterrà il registro A alla fine del programma?

ESERCIZIO 2.4: *Scrivere un programma, il più breve possibile, che scriva il valore 0 nel più gran numero possibile di parole di memoria.*

ESERCIZIO 2.5: *Una zona di memoria puntata dall'etichetta BUFFER contiene una sequenza di dieci informazioni, per esempio i numeri da 0 a 9 in formato ASCII, in ordine crescente. Scrivere un programma che cambia l'ordine di queste informazioni in ordine decrescente.*

CAPITOLO 3

ISTRUZIONI DI UN ASSEMBLER TIPO Z80

Il set di istruzioni dello Z80 produce, salvo errore, 696 codici di operazione diversi... (!), che, dopo un'opportuna classificazione, diventano fortunatamente meno barbosi!

Nelle pagine seguenti cercheremo di suddividere questo set di istruzioni in nove "famiglie", segnalando per ognuna i diversi modi di indirizzamento consentiti, allo scopo di facilitarne lo studio.

Le informazioni del tipo: numero di cicli di clock, tempi di esecuzione, codice oggetto prodotto, sono state volontariamente ignorate per non appesantire troppo le tabelle. D'altra parte queste informazioni sono raramente utilizzate da chi programma in assembler. In ogni caso, i codici oggetto compaiono in Appendice.

Come ogni classificazione, anche questa ha delle imperfezioni, e talvolta abbiamo avuto dei problemi a inserire alcune istruzioni in una famiglia piuttosto che in un'altra. In questi casi sono stati previsti dei richiami.

Le nove famiglie sono le stesse che abbiamo già individuato nel capitolo precedente, e che ricordiamo qui:

1. lettura/scrittura in memoria,
2. caricamento e scambi di registri,
3. funzioni aritmetiche,
4. funzioni logiche e funzioni di confronto,
5. uso generale e controllo interno,
6. interruzioni di sequenze (salti),
7. ingresso/uscita,
8. operazioni speciali su bit e stringhe,
9. manipolazione dello stack.

Forse si sarebbe potuto inserire le istruzioni dello stack nella prima famiglia (lo stack è in memoria anche lui!). Noi abbiamo preferito utilizzare una famiglia separata per descrivere una funzione che, ricordiamocelo, non esiste su tutti gli elaboratori.
Prima di iniziare questo studio, definiamo preliminarmente i simboli che saranno utilizzati:

- **ADR** designerà sempre il valore di un indirizzo a 16 bit.
- **VAL8** rappresenterà un valore a 8 bit.
- **VAL16** sarà un valore a 16 bit o un indirizzo.
- **DEPL** sarà utilizzato in particolare nelle istruzioni che usano il modo di indirizzamento relativo, e rappresenterà la label di un indirizzo. Abbiamo preferito utilizzare un simbolo diverso da ADR, per sottolineare che il codice prodotto dall'assemblatore è in realtà uno "spiazzamento" relativo, che occupa un byte.
- **d** rappresenterà un valore con segno (+ 127, -128) nelle istruzioni che usano l'indirizzamento indicizzato.

FAMIGLIA 1: LETTURA/SCRITTURA IN MEMORIA

Queste funzioni sono realizzate dall'**istruzione LD.**
Lo scambio di informazioni avviene dalla memoria ai registri o viceversa.
Il puntatore alla memoria può essere un indirizzo diretto o un registro doppio.

Indirizzamento	Istruzioni	Flag
diretto a 8 bit	LD A,(ADR) LD (ADR), A	inalterati

La prima istruzione carica nel registro A il byte di memoria che si trova all'indirizzo definito da ADR, la seconda memorizza il contenuto del registro A a questo indirizzo. Che dire d'altro su queste istruzioni ormai ben conosciute?

Indirizzamento	Istruzioni		Flag

indiretto
tramite HL
a 8 bit
	LD A,(HL)
	LD B, (HL)
	LD C, (HL)
	LD D, (HL)
	LD E, (HL)
	LD H, (HL)
	LD L, (HL)
		LD (HL), A
		LD (HL), B
		LD (HL), C
		LD (HL), D
		LD (HL), E
		LD (HL), H
		LD (HL), L
			inalterati

LD A, (HL) carica nel registro A il byte di memoria il cui indirizzo è contenuto nel registro doppio HL.
LD (HL), A memorizza il contenuto del registro A a questo indirizzo. Le altre istruzioni agiscono nello stesso modo, ma con i registri da B a L.

Indirizzamento	Istruzioni		Flag
indiretto tramite BC DE a 8 bit	LD A,(BC) LD A, (DE)	LD (BC), A LD (DE), A	inalterati

Solo il registro A ha il privilegio di accedere alla memoria usando come registri puntatori i registri doppi BC o DE.

Indirizzamento	Istruzioni		Flag
indiretto indicizzato a 8 bit	LD (IX+d), A	LD A, (IX+d)	inalterati
	LD (IY+d),A	LD A, (IY+d)	
	LD (IX+d),B	LD B, (IX+d)	
	LD (IY+d),B	LD B, (IY+d)	
	LD (IX+d),C	LD C, (IX+d)	
	LD (IY+d),C	LD C, (IY+d)	
	LD (IX+d),D	LD D, (IX+d)	
	LD (IY+d),D	LD D, (IY+d)	
	LD (IX+d),E	LD E, (IX+d)	
	LD (IY+d),E	LD E, (IY+d)	
	LD (IX+d),H	LD H, (IX+d)	
	LD (IY+d),H	LD H, (IY+d)	
	LD (IX+d),L	LD L, (IX+d)	
	LD (IY+d),L	LD L, (IY+d)	

Con queste istruzioni il registro considerato da (A a L) viene memorizzato all'indirizzo contenuto dal registro indice IX o IY aumentato del valore dello spiazzamento "d", o viceversa.

Indirizzamento	Istruzioni	Flag
immediato a 8 bit	LD (HL), VAL8	inalterati
indiretto e indiretto indicizzato	LD (IX+d), VAL8	
	LD (IY+d), VAL8	

Il valore immediato VAL8 è caricato nell'indirizzo di memoria contenuto in HL o in un registro indice. In quest'ultimo caso bisogna aggiungere all'indirizzo lo spiazzamento relativo "d", e il modo di indirizzamento è allora, immediato, indiretto e indicizzato. Abbiate pazienza! Ne vedremo degli esempi fra poco! Naturalmente, in questo modo di indirizzamento si possono effettuare solo scritture in memoria.

Indirizzamento	Istruzioni		Flag

diretto
a 16 bit
	LD (ADR), BC
	LD (ADR), DE
	LD (ADR), HL
	LD (ADR), SP
	LD (ADR), IX
	LD (ADR), IY
		LD BC,(ADR)
		LD DE, (ADR)
		LD HL, (ADR)
		LD SP, (ADR)
		LD IX, (ADR)
		LD IY, (ADR)
			inalterati

Il contenuto del doppio registro (a 16 bit) specificato è direttamente memorizzato all'indirizzo ADR e ADR + 1, e viceversa. In che ordine? Rammentiamolo: il registro basso (C) all'indirizzo basso ADR, e il registro alto (B) all'indirizzo alto (ADR+1).

Indirizzamento	Istruzioni	Flag
indiretto da memoria a memoria	LDI LDD	. S,Z,C inalterati .N,H=0 .P/V = 0 se BC =0 altrimenti = 1

Ecco due istruzioni molto potenti in un loop controllato. Il byte puntato da HL è caricato all'indirizzo puntato da DE. Inoltre BC è decrementato di 1, e mette a 0 il flag P/V quando raggiunge un valore nullo. Ma non è tutto: i due registri puntatori HL e DE vengono incrementati nel caso dell'istruzione LDI (I sta per Incremento), e decrementati nel caso dell'istruzione LDD (D sta per Decremento)

Indirizzamento	Istruzioni	Flag
	vedi le istruzioni speciali su blocchi di dati	

FAMIGLIA 2: CARICAMENTO E SCAMBIO DI REGISTRI

Questa nuova famiglia è molto simile alla prima, a tal punto che utilizza lo stesso codice mnemonico di operazione "LD".

CARICAMENTO

Indirizzamento	Istruzioni	Flag

immediato
a 8 bit
	LD A, VAL8
	LD B, VAL8
	LD C, VAL8
	LD D, VAL8
	LD E, VAL8
	LD H, VAL8
	LD L, VAL8
		inalterati

Il valore a 8 bit rappresentato da VAL8 è caricato in modo immediato nel registro considerato

Esempio:
```
LD      H,0FCH
```
carica il valore esadecimale 0FG nel registro H.

Indirizzamento	Istruzioni	Flag

immediato
a 16 bit
	LD BC, VAL16
	LD DE, VAL16
	LD HL, VAL16
	LD SP, VAL16
	LD IX, VAL16
	LD IY, VAL16
		inalterati

Con queste istruzioni la coppia di registri specificati nel primo argomento dell'operando è caricata con il valore a 16 bit VAL16.

Esempio:
```
LD      IX,4000H
LD      IY,65535
```

I valori 4000H e FFFFH (65535) saranno caricati rispettivamente nei registri indice IX e IY.

Indirizzamento	Istruzioni	Flag
registri a 8 bit	LD R1, R2 dove R1 e R2 sono uno dei registri a 8 bit: A,B,C,D,E,H,L,	inalterati

Qui non ci sono restrizioni di sorta, sono permessi tutti i colpi!

Esempio:
```
LD      A,A     ; e' ridicolo
LD      H,C     ; e' gia' meglio
```

Indirizzamento	Istruzioni	Flag
registri speciali a 8 bit	LD A,I LD A,R	H,N = 0 C inalterato S,Z modificati a seconda del risultato P/V = IFF
	LD I,A LD R,A	inalterati

Questi due tipi di istruzioni utilizzano i registri speciali dello Z80: I (interruzioni) e R (rinfresco della memoria).
Le due istruzioni di caricamento del registro A posizionano i flag del segno (S) e di zero (Z) a seconda del valore caricato in A. Inoltre, il flag (P/V contiene, dopo l'esecuzione, lo stato di un flip-flop chiamato IFF, che indica lo stato delle interruzioni (abilitate o disabilitate).

Indirizzamento	Istruzioni	Flag

registri a 16 bit	LD SP,HL
	LD SP,IX
	LD SP,IY
		inalterati

Il registro puntatore dello stack (stack pointer) ha il privilegio di poter essere caricato da uno dei registri doppi HL, IX o IY.

SCAMBI

È venuto il momento di cambiare un po' il codice mnemonico...
LD cominciava a diventare monotono!

Indirizzamento	Istruzioni	Flag
registri a 16 bit	EXX	inalterati
	EX AF,AF'	assumono lo stato del registro F'

L'istruzione EXX realizza lo scambio dei registri BC, DE, HL con i loro omologhi BC', DE', e HL' del secondo blocco di registri.
La seconda istruzione fa la stessa cosa con i registri doppi AF e AF'. L'esecuzione di queste istruzioni implica l'utilizzo di un registro che il microprocessore tiene segreto, e che permette il trasferimento:

AF ----> registro intermedio
AF' ----> AF
registro intermedio ----> AF'
nel caso dell'istruzione EX AF, AF'.

Indirizzamento	Istruzioni	Flag
registri a 16 bit	EX DE,HL	inalterati

Questa istruzione scambia il contenuto dei registri doppi DE e HL del blocco principale.

Indirizzamento	Istruzioni	Flag
indiretto a 16 bit tramite SP	EX (SP),HL EX (SP),IX EX (SP),IY	inalterati

Qualcuno dirà che queste istruzioni sono mal poste in questa famiglia... In questo caso, effettivamente, lo scambio avviene tra uno dei registri doppi HL, IX, e IY da una parte, e la memoria puntata dal registro SP, dall'altra (lo stack).

È giunto il momento di rilassarsi un po'... con qualche esempio!

```
LD      IX,4000H
LD      BC,4243H      ; caratteri B e C
LD      (IX),B
LD      (IX+1),C
LD      (IX-1),41H
```

Dopo l'esecuzione di questo breve programma, le locazioni di memoria agli indirizzi 3FFFH,4000H, e 4001H conterranno rispettivamente i valori 41H, 42H e 43H, cioè i caratteri "A", "B", e "C" in ASCII.

```
LD      HL,4000H
LD      A,0AAH
LD      (HL),A
EX      DE,HL
LD      (4000H+1),DE
```

Ora le locazioni di memoria da 4000H a 4002H contengono i valori esadecimali 0AAH, OOH e 40H. OK?

FAMIGLIA 3: FUNZIONI ARITMETICHE

Passiamo ora ad una famiglia molto importante, ma molto semplice. Bisognerà ricordarsi che queste istruzioni operano sempre su valori di 7 o 15 bit più il segno, e che i flag li interpretano come tali.

ADDIZIONI con e senza "CARRY"

Poichè queste istruzioni sono molto simili, le abbiamo raggruppate in una stessa tabella.
Le addizioni con Carry (flag C) **ADC** funzionano come le addizioni semplici **ADD**, ma aggiungono al risultato il valore del bit C, nello stato in cui si trovava prima dell'esecuzione dell'istruzione.
Il flag N è messo sempre a 0. Gli altri flag sono posizionati a seconda del risultato.

Indirizzamento	Istruzioni	Flag
immediato a 8 bit	ADD A,VAL8 ADC A,VAL8	modificati

Il valore VAL8 a 8 bit è aggiunto a quello contenuto nell'accumulatore A, che contiene poi il risultato. Nel caso dell'istruzione ADC è aggiunto anche il flag C.

Indirizzamento	Istruzioni		Flag
registri a 8 bit	ADD A,A	ADC A,A	modificati
	ADD A,B	ADC A,B	
	ADD A,C	ADC A,C	
	ADD A,D	ADC A,D	
	ADD A,E	ADC A,E	
	ADD A,H	ADC A,H	
	ADD A,L	ADC A,L	

Nel caso di queste istruzioni, il contenuto del registro considerato è aggiunto a quello dell'accumulatore. L'istruzione ADC addiziona in più il valore del bit del Carry al risultato.

Indirizzamento	Istruzioni		Flag
indiretto e indiretto indicizzato a 8 bit	ADD A,(HL) ADD A,(IX + d) ADD A,(IY + d)	ADC A,(HL) ADC A,(IX + d) ADC A,(IY + d)	modificati

Qui non è più il registro ma il byte di memoria puntato dal secondo argomento dell'operando che è aggiunto al contenuto dell'accumulatore.

Indirizzamento	Istruzioni		Flag

registri
a 16 bit
	ADD HL,BC
	ADD HL,DE
	ADD HL,HL
	ADD HL,SP
		ADC HL,BC
		ADC HL,DE
		ADC HL,HL
		ADC HL,SP
			modificati

Queste istruzioni operano su valori di 16 bit (15 + il segno). HL funziona da un doppio accumulatore, e riceve il risultato della addizione del suo contenuto con uno dei registri doppi BC, DE, HL o SP.

Indirizzamento	Istruzioni	Flag

registri
a 16 bit
	ADD IX,BC
	ADD IX,DE
	ADD IX,IX
	ADD IX,SP
	ADD IY,BC
	ADD IY,DE
	ADD IY,IY
	ADD IY,SP
		modificati

In questo caso manca l'istruzione ADC. Ma si prenderà la sua rivincita in seguito.

SOTTRAZIONI con o senza "carry"

Il principio è lo stesso dell'addizione, i simboli mnemonici che troveremo sono **SUB** e **SBC**. Con quest'ultimo il valore del bit C è sottratto al risultato. Il flag N è sempre forzato a 1 in una sottrazione.

Per complicare un po' le cose, la sintassi della sottrazione obbedisce a delle regole un po' originali. Mentre per l'addizione, per esempio, si scriveva ADD A,B, la sottrazione è privata del suo primo argomento e si scrive SUB B, che ha lo stesso significato di SUB A,B, ma manca di omogeneità con le altre istruzioni.

Il peggio è che SBC continua a scriversi SBCA,B...., il chè è veramente assurdo. Vedremo più avanti che purtroppo non è l'unico caso. Ma questo non vi deve scoraggiare, visto che ora siete preavvertiti.

Indirizzamento	Istruzioni		Flag
immediato a 8 bit	SUB VAL8	SBC A,VAL8	modificati

Il valore a 8 bit rappresentato da VAL8 è sottratto dall'accumulatore. L'istruzione SBC, inoltre, sottrae al risultato il valore che il bit C ha prima dell'esecuzione dell'istruzione.

Indirizzamento	Istruzioni		Flag
registri a 8 bit	SUB A	SBC A,A	modificati
	SUB B	SBC A,B	
	SUB C	SBC A,C	
	SUB D	SBC A,D	
	SUB E	SBC A,E	
	SUB H	SBC A,H	
	SUB L	SBC A,L	

Notate che l'istruzione: SUB A dà sempre un risultato nullo. Invece SBC A,A può dare come risultato 0 o .1 (cioè 0FFH) a seconda del valore del carry.....

Indirizzamento	Istruzioni		Flag
indiretto e indiretto indicizzato a 8 bit	SUB (HL) SUB (IX + d) SUB (IY + d)	SBC A,(HL) SBC A,(IX + d) SBC A,(IY + d)	modificati

Il byte di memoria puntato indirettamente dall'argomento è sottratto dal contenuto dell'accumulatore (con il carry, nel caso di SBC).

Indirizzamento	Istruzioni	Flag
registri a 16 bit	SBC HL,BC SBC HL,DE SBC HL,HL SBC HL,SP	modificati

Ecco la rivincita di cui parlavamo prima! Qui non ci sono istruzioni SUB equivalenti, in quanto bisogna sorvegliare il valore di C prima di eseguire una sottrazione a 16 bit. È meglio che niente, visto che il cugino 8080 non ha neanche questa possibilità!

INCREMENTI E DECREMENTI

Gli incrementi e i decrementi non sono altro che addizioni e sottrazioni con l'unità.

Decrementare il registro A, per esempio, significa togliergli un 1. Queste istruzioni sono molto utilizzate nei programmi.

Indirizzamento	Istruzioni		Flag
registri a 8 bit	INC A INC B INC C INC D INC E INC H INC L	DEC A DEC B DEC C DEC D DEC E DEC H DEC L	C inalterato, gli altri modificati

Vediamo un esempio:

```
LD      A,40H
DEC     A
```

il registro A dopo l'esecuzione conterrà il valore 3FH.

Indirizzamento	Istruzioni		Flag
indiretto e indiretto indicizzato	INC (HL) INC (IX + d) INC (IY + d)	DEC (HL) DEC (IX + d) DEC (IY + d)	C inalterato gli altri modificati

In questo caso il valore contenuto nella locazione di memoria puntata dall'operando sarà incrementato o decrementato di 1.

Indirizzamento	Istruzioni		Flag
registri a 16 bit	INC BC INC DE INC HL INC SP INC IX INC IY	DEC BC DEC DE DEC HL DEC SP DEC IX DEC IY	inalterati

Queste istruzioni hanno una incresciosa particolarità: non alterano i flag. Questo talvolta è fastidioso, ma meno quando lo si sa.

```
LD      HL,0FFFFH
INC     HL
```

Che cosa succede, secondo voi?

Indirizzamento	Istruzioni	Flag
implicito	DAA	N inalterato gli altri modificati

Indirizzamento implicito? Che cosa significa? Vuole semplicemente dire che questa istruzione, come altre dello stesso stile, sa implicitamente che deve "lavorare" sul registro A. Ma che cosa fa esattamente? Esegue quello che si usa chiamare: un aggiustamento decimale. Adesso capirete tutto.

Eseguiamo l'addizione di due numeri: 8 e 6:

```
LD      A,8
LD      A,6
ADD     A,B
```

Il registro A contiene ora il valore binario 0000 1110 cioè 0EH in esadecimale. Ma se noi vogliamo ottenere il risultato in decimale, c'è una soluzione: l'istruzione DAA. Eseguita dopo l'addizione produrrà il risultato 0001 0100 nell'accumulatore, cioè 14...in esadecimale. Ora, 8 e 6 fanno proprio 14! È come fare un'addizione in esadecimale. Con numeri grandi occorre utilizzare il carry, e passarlo al byte seguente.

L'istruzione DAA lavora sui "nibbles" (1 nibbles = 4 bit = 1/2 byte). Quando il valore del nibble destro di A supera 9, o il bit H è a 1, l'accumulatore è incrementato di 6. Se ora il valore del nibble sinistro supera 9, o il bit C è a 1, questo nibble è incrementato di 6. È un modo per trasformare il binario in BCD (decimale codificato in binario).

FAMIGLIA 4: LE FUNZIONI LOGICHE

Le istruzioni logiche comprendono l'intersezione (*AND*), l'unione (*OR*), la disgiunzione (*XOR*), la negazione, il confronto (poichè si tratta di confronti logici e non aritmetici), lo shift e la rotazione.

INTERSEZIONE

Queste istruzioni, come la maggior parte di quelle che seguono, agiscono in modo implicito sul registro A, che quindi non figura nell'operando.

Indirizzamento	Istruzioni	Flag
immediato a 8 bit	AND VAL8	C e N = 0 P = parità H = 1 S e Z modificati

Si opera un'intersezione logica tra il dato a 8 bit VAL8 e l'accumulatore. Il risultato si trova in A.

Ci si può domandare: perchè il flag H è messo a 1?

Notare che il flag P indica la parità del risultato. S e Z indicano rispettivamente se il valore è negativo o nullo, quando assumono il valore 1.

Indirizzamento	Istruzioni	Flag
indiretto e indiretto indicizzato a 8 bit	AND (HL) AND (IX + d) AND (IY + d)	come AND immediato

Viene eseguita l'intersezione tra il valore puntato dall'operando e il contenuto dell'accumulatore. Il risultato viene posto in A.

```
LD      A,0AAH
LD      HL,4000H
LD      (HL),A
INC     A               ; A = 0ABH ora
AND     (HL)
```

L'accumulatore conterrà il valore 0AAH.

Indirizzamento	Istruzioni	Flag
registro a 8 bit	AND A AND B AND C AND D AND E AND H AND L	come AND immediato

Viene eseguita l'intersezione tra il contenuto del registro indicato nell'operando e il contenuto dell'accumulatore.

A cosa dunque può servire AND A? A priori a niente! Ciononostante è da inserire nella lista dei "trucchi" del programmatore. Vediamo un esempio: vi ricordate che l'istruzione LD non modifica i flag. Allora come facciamo a sapere se il valore contenuto in A è nullo, per esempio? Basterà fare

```
LD      A,(ADR)
AND     A
```

e i due flag S e Z saranno posizionati, informandoci così sulla natura del valore caricato.

UNIONE

Indirizzamento	Istruzioni	Flag
immediato a 8 bit	OR VAL8	C e N = 0 P = parità H = 1 S e Z modificati

Si esegue l'unione logica del dato immediato VAL8 con il contenuto del registro A

```
LD      A,8
OR      4
```

Il registro A contiene ora il valore 12 (OCH).

Indirizzamento	Istruzioni	Flag
indiretto e indiretto indicizzato a 8 bit	OR (HL) OR (IX + d) OR (IY + d)	come OR immediato

Non c'è proprio niente da dire; vediamo allora un piccolo esempio:

```
LD      IX,TRUC
LD      BC,1234H
LD      (IX),B
LD      (IX+1),C
AND     0               ; si' : AND z.e.r.o.!
OR      (IX)
OR      (IX+1)
```

Il registro A conterrà il valore 36H.

Indirizzamento	Istruzioni	Flag
registro a 8 bit	OR A OR B OR C OR D OR E OR H OR L	come OR immediato

L'istruzione OR A ha il potere, come AND A, di far posizionare i flag Z e S a seconda del valore caricato nell'accumulatore. Un altro esempio? Eccolo:

```
LD      BC,0AFEH - 2
LD      A,0
OR      B
OR      C
```

Il registro A conterrà il valore OFEH.

DISGIUNZIONE

La disgiunzione, chiamata anche OR eclusivo, è realizzata dalle seguenti istruzioni:

Indirizzamento	Istruzioni	Flag
immediato a 8 bit	XOR VAL8	C e N = 0 P = parità H = 1 S e Z modificati

```
LD      A,0AFH
XOR     5
```

e il registro A conterrà il valore 0AAH. Semplice, no?

Indirizzamento	Istruzioni	Flag
indiretto e indiretto indicizzato	XOR (HL) XOR (IX + d) XOR (IY + d)	come XOR immediato

Ecco un esempio: si vuole ottenere alternativamente il valore 0,1, 0,1....in un byte di memoria, ogni volta che si passa da un indirizzo preciso di programma. Il programma avrà la seguente struttura:

```
        LD      HL,OSCIL        ;
        LD      (HL),0          ; inizializzazione del byte
                                ; "oscillante"
ETIC    ...
        ...
        LD      A,(HL)
        XOR     1               ; cambiamento di stato
        LD      (HL),A          ; memorizzazione nel byte
                                ; oscillante
        ...
        JR      ETIC            ; continuaz. del trattamento
```

Indirizzamento	Istruzioni	Flag
registri a 8 bit	XOR A XOR B XOR C XOR D XOR E XOR H XOR L	come XOR immediato

Ancora un'altra astuzia del programmatore: l'istruzione XOR A, che sembra completamente inutile, è di fatto geniale, perchè consente, per esempio di mettere l'accumulatore a zero con un solo byte (LD A,0 ne occupa due) indipendentemente dal valore contenuto precedentemente

```
XOR     A
LD      B,A
LD      C,A
```

I registri A,B,C, saranno azzerati.

COMPLEMENTAZIONE E NEGAZIONE

Le due istruzioni che seguono realizzano rispettivamente il complemento a 1(NOT) o complementazione, e il complemento a 2 o negazione dell'accumulatore.

Indirizzamento	Istruzioni	Flag
implicito	CPL	N e H = 1 gli altri invariati
	NEG	N = 1 gli altri modificati

Esempio:

```
LD      A,14H
CPL
NEG
```

dopo l'istruzione CPL l'accumlatore conterrà OEBH, e dopo la NEG conterrà 15H.

```
0 0 0 1   0 1 0 0 ----------> il contenuto iniziale di A
1 1 1 0   1 0 1 1 ----------> complemento a 1 di A
0 0 0 1   0 1 0 0 ----------> altro complemento a 1
                1 ----------> + 1 da'
0 0 0 1   0 1 0 1 ----------> il complemento a 2
```

CONFRONTO

Questo tipo di istruzione è molto utilizzata nei programmi. Per interpretare correttamente il bit carry, i due valori confrontati, di cui uno è sempre nell'accumulatore, devono essere considerati come senza segno, se non ci si vuole ritrovare in breve tempo all'ospedale psichiatrico più vicino (lo psichiatra non tarderebbe certo a fare lui dei confronti!)

Mi segua bene, Dottore: il confronto tra due valori in effetti non è che una sottrazione, che mette a 1 il bit del carry se non c'è riporto dal bit 7 dell'operando, segnalando il questo modo che quest'ultimo è maggiore del contenuto dell'accumulatore (è molto chiaro), a meno che le due quantità abbiano segni diversi, nel qual caso il senso del carry deve essere invertito.....

Perchè sforzare le meningi, quando basta semplicemente consultare la tabella seguente:

Carry	A e operando in valore assoluto
0	A > o = all'operando
1	A < dell'operando

D'altra parte Z indica sempre l'uguaglianza, se assume il valore 1. Gli altri flag conservano il loro significato

Indirizzamento	Istruzioni	Flag
immediato a 8 bit	CP VAL8	N = 1 gli altri modificati

Il valore immediato VAL8 viene confrontato con l'accumulatore.

Esempio:

```
LD      A,5
CP      4           ; C=0,A>4
CP      7           ; C=1,A<7
CP      5           ; C=0,A=5
CP      0FEH        ; C=1,A<0FEH
LD      A,0FFH
CP      5           ; C=0,A>5
```

Indirizzamento	Istruzioni	Flag
registri a 8 bit	CP A CP B CP C CP D CP E CP H CP L	N = 1 gli altri modificati

Il registro considerato è confrontato con l'accumulatore. L'istruzione CP A appare tuttavia enigmatica, bisogna ammetterlo! Essa può essere utilizzata per mettere C a 0, senza modificare gli altri flag (V e H, per esempio).

Indirizzamento	Istruzioni	Flag
indiretto e indiretto indicizzato a 8 bit	CP (HL) CP (IX + d) CP (IY + d)	N= 1 gli altri modificati

Fin qui nessuna difficoltà.

Indirizzamento	Istruzioni	Flag
indiretto con incremento decremento	CPI CPD	N = 1 C invariato P/V = 1 salvo che per BC = 0 Z = 1 se A = (HL)

Ritroviamo qui due istruzioni equivalenti a quelle incontrate nella prima famiglia: LDI e LDD.

Il byte puntato da HL viene confrontato con il contenuto dell'accumuatore; se i due sono uguali il flag Z è messo a 1. Nello stesso tempo il registro doppio BC viene decrementato, e P/V passa a zero quando BC raggiunge il valore zero.

Infine, a seconda dell'istruzione, HL è incrementato per l'istruzione CPI e decrementato per la CPD.

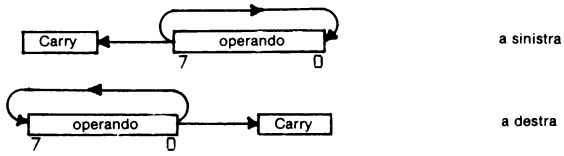
Indirizzamento	Istruzioni	Flag
	vedere istruzioni speciali per blocchi di dati	

SHIFT

In tutto ci sono cinque grandi categorie di shift:
— circolare a sinistra o a destra,
— circolare tramite carry, a sinistra o a destra,
— aperto aritmetico, a sinistra o a destra,
— aperto logico, solo a destra,
— circolare BCD, a sinistra e a destra.

Questi cinque gruppi verranno ora esaminati con i loro diversi modi di indirizzamento.

Circolare a sinistra e a destra



I bit dell'operando sono spostati verso sinistra o verso destra. Il bit che esce da una parte entra dall'altra, e viene anche copiato nel carry.

Indirizzamento	Istruzioni		Flag
registri a 8 bit	RLC A RLC B RLC C RLC D RLC E RLC H RLC L	RRC A RRC B RRC C RRC D RRC E RRC H RRC L	H = N = 0 S,Z,C e P modificati
	RLCA	RRCA	H = N = 0 S,Z,P inalterati C modificato

RLC = Rotate Left Circular RRC = Rotate Right Circular

Il registro operando è shiftato circolarmente di una posizione binaria verso sinistra (left) o verso destra (right). Il bit uscente è ricopiato nel carry.
Le istruzioni RLCA/RRCA provengono dal microprocessore 8080, ed eseguono le stesse operazioni di RLC A e RRC A, tranne che i flag S,Z e P non sono

modificati da queste ultime. In compenso RLCA/RRCA occupano solo un byte di codice, contro i due occupati da RLC A/RRC A.

```
Esempio: LD      B,0A4H
         RLC     B
```

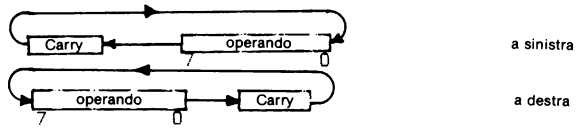
Il registro B conterrà il valore 49H, e il flag C sarà posizionato a 1.

Indirizzamento	Istruzioni		Flag
indiretto e indiretto indicizzato a 8 bit	RLC (HL) RLC (IX + d) RLC (IY + d)	RRC (HL) RRC (IX + d) RRC (IY + d)	come RCL A

Il byte di memoria puntato dall'operando è shiftato circolarmente di una posizione binaria verso sinistra o verso destra. Il bit uscente è copiato nel carry.

Circolare tramite carry, sinistra/destra

Immaginate che l'operando non abbia più 8 bit, ma 9,...., e che sia shiftato circolarmente: avrete un'idea esatta di che cosa può essere questo tipo di shift.



Gli 8 bit dell'operando sono spostati verso sinistra o verso destra di una posizione binaria. Il bit che esce da un lato entra nel carry, il vecchio valore del carry entra dall'altro lato dell'operando. È come uno shift circolare su 9 bit.

Indirizzamento	Istruzioni		Flag
registri a 8 bit	RL A RL B RL C RL D RL E RL H RL L	RR A RR B RR C RR D RR E RR H RR L	H = N = 0 S,Z,C e P modificati
	RLA	RRA	H = N = 0 C modificato S,Z,P invariati

RL = Rotate Left, RR = Rotate Right

A proposito delle istruzioni RLA e RRA c'è da osservare, come già fatto in precedenza, che sono prese dal microprocessore 8080.

Indirizzamento	Istruzioni		Flag
diretto e indiretto indicizzato a 8 bit	RL (HL) RL (IX + d) RL (IY + d)	RR (HL) RR (IX + d) RR (IY + d)	come per RL A

Il byte di memoria puntato dall'operando è shiftato circolarmente di una posizione binaria, a sinistra o a destra.

Esempio:

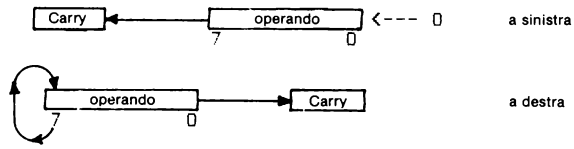
```
LD      IX,4000H
LD      0A4H
OR      A           ; carry = 0
LD      (IX + 3),A
RL      (IX + 3)
```

Il byte di memoria posto all'indirizzo 4003H conterrà il valore 49H e il carry sarà posizionato a 1.

Aperto aritmetico a sinistra o a destra

Questi shift si chiamano **aperti** perchè il bit che esce dall'operando a sinistra o a destra è perso, mentre il bit che entra è zero, e **aritmetici** perchè lo shift a destra

conserva il segno (il bit 7 dell'operando).



Indirizzamento	Istruzioni		Flag
registri a 8 bit	SLA A SLA B SLA C SLA D SLA E SLA H SLA L	SRA A SRA B SRA C SRA D SRA E SRA H SRA L	H = N = 0 S,Z,C,P modificati

SLA = Shift Left Arithmetic, SRA = Shift Right Arithmetic

Poichè tutto questo è chiaro, vediamo un esempio:

```
LD      B,0A5H
SRA     B
```

Il registro B conterrà il valore 4AH e il flag del carry sarà messo a 1.

Indirizzamento	Istruzioni		Flag
indiretto e indiretto indicizzato a 8 bit	SLA (HL) SLA (IX + d) SLA (IY + d)	SRA (HL) SRA (IX + d) SRA (IY + d)	come SLA A

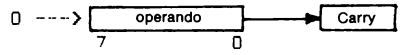
Esempio:

```
LD      IY,4000H
LD      A,0A5H
LD      (IY+3),A
SRA     (IY+3)
```

Il byte di memoria posto all'indirizzo 4003H conterrà il valore OD2H, e il flag del carry sarà messo a 1.

Aperto logico a destra

Per compensare SRA che ricopia il bit del segno dopo lo shift, ecco un nuovo tipo di istruzione che non presenta questo vantaggio (o svantaggio, dipende.....).



Indirizzamento	Istruzioni	Flag
registri a 8 bit	SRL A SRL B SRL C SRL D SRL E SRL H SRL L	H = N = 0 S,Z,C,P modificati

SRL = Shift Right Logical

Esempio:
```
LD      B,83H
SRL     B
```

Il registro B conterrà il valore 41H, e il flag del carry sarà messo a 1.

Indirizzamento	Istruzioni	Flag
indiretto e indiretto indicizzato a 8 bit	SRL (HL) SRL (IX + d) SRL (IY + d)	come SRL A

Il byte di memoria puntato dall'operando sarà shiftato logicamente di una posizione verso destra. Con l'istruzione SRL si possono realizzare shift aperti su 16 bit.

Esempio: Si debba shiftare verso destra il contenuto del registro BC.

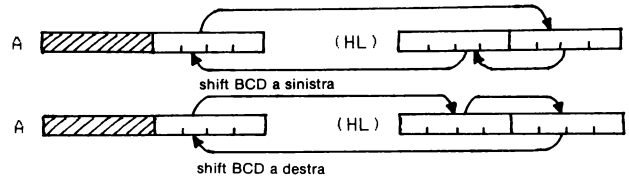
```
LD      BC,2345H
SRL     B           ; B-->carry
RR      C           ; carry-->C
```

Al momento della SRL, il registro B è shiftato verso destra, e il bit che esce entra nel carry. La RR consente poi di trasportare questo bit nel registro C, che viene anche lui shiftato di una posizione verso destra.

Circolare BCD a sinistra e a destra

Quest'ultimo gruppo di shift contiene due istruzioni particolarmente potenti nell'aritmetica decimale.
Tutti gli shift studiati finora coinvolgevano un bit alla volta, che è la più piccola unità del sistema di numerazione binario.
Nell'aritmetica decimale, la più piccola unità è costituita da quattro bit (nibble), che rappresentano i valori da zero a nove.
Bisogna dunque aspettarsi che gli shift di un numero in BCD coninvolgano un nibble.

Nel caso di queste istruzioni, l'accumulatore sarà utilizzato come registro di "manovra", e l'operando sarà il registro puntatore HL.

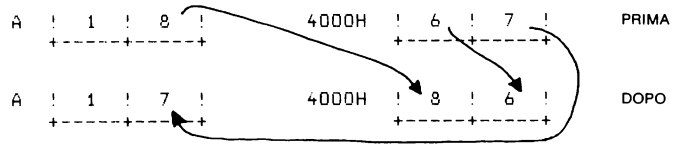


Indirizzamento	Istruzioni	Flag
indiretto a 8 bit	RLD RRD	N = H = 0 C invariato S,Z,P modificati

RLD = Rotate Left Decimal, RRD = Rotate Right Decimal.

Esempio:
```
LD      B,67H
LD      HL,4000H
LD      (HL),B
LD      A,18H
RRD
```

Dopo l'esecuzione, il byte di memoria 4000H contiene 86H, e il registro A contiene 17H. Notare che il byte alto di A non è modificato.



FAMIGLIA 5: USO GENERALE E CONTROLLI INTERNI

È la famiglia dei diseredati....Non si sapeva dove metterli, e allora li si è messi qua.

Operazioni dirette sul carry

Indirizzamento	Istruzioni	Flag
implicito	SCF	C = 1 H = N = 0
	CCF	C modificato N = 0

La prima istruzione forza il carry a prendere il valore 1 (Set Carry Flag), mentre la seconda inverte il suo stato (complement Carry Flag).

Le istruzioni.....di riposo

Nell'equipe che ha progettato lo 8080 e lo Z80, doveva esserci un ricercatore pigro, molto pigro....Bisogna però congratularsi caldamente con lui per aver avuto l'idea di aggiungere le due seguenti istruzioni:

Indirizzamento	Istruzioni	Flag
implicito	NOP HALT	inalterati

La prima istruzione non esegue....alcuna operazione (No Operation) e viene utilizzata soprattutto in due occasioni. Quando in un programma si devono eliminare alcune istruzioni, le si può sostituire al livello di codice oggetto con delle istruzioni NOP, utilizzando un particolare programma (Patch). In questo modo si evita di assemblare un'altra volta il programma, soprattutto se è molto lungo. Il secondo tipo di utilizzo è legato dalla nozione di tempo. Quando il programma deve eseguire soprattutto operazioni di ingresso/uscita, con dei tempi precisi, può essere necessario correggere questa o quella parte aggiungendo delle istruzioni NOP, in modo da far passare qualche microsecondo e ottenere così il sincronismo desiderato. È quello che si dice "passare il tempo senza fare niente"!

L'istruzione HALT arresta semplicemente l'esecuzione del programma. Questo non potrà poi essere rimesso in esecuzione che tramite un'intervento esterno (Reset o interruzione). Ma se l'esecuzione del programma viene arrestata, il microprocessore continua a lavorare eseguendo delle NOP interne per assicurare il rinfresco delle memorie.

Le istruzioni legate alle interruzioni

Il microprocessore Z80 possiede tre modi differenti per trattare le interruzioni. Esamineremo molto rapidamente le istruzioni che permettono di passare da un modo all'altro.
Ma forse voi vi domandate che cos'è esattamente un'interruzione. È un evento esterno che provoca l'interruzione del programma in corso, il trattamento specifico dell'interruzione ricevuta (con un piccolo programma chiamato programma di servizio dell'interruzione) e la ripresa del programma interrotto.
È una specie di "grido di allarme" che viene da una periferica, per dire: "Ho finito", oppure "Attenzione, invio un messaggio", o qualche cosa del genere.

Indirizzamento	Istruzioni	Flag
implicito	EI DI IM 0 IM 1 IM 2	inalterati

Le prime due istruzioni sono usate rispettivamente per attivare (Enable Interrupt) o disattivare (Disable Interrupt) il sistema di accettazione dell'interruzione del microprocessore.
Le tre istruzioni seguenti permettono il passaggio nei differenti modi di interruzione citati prima (l'8080 possiede solo il modo 0).

FAMIGLIA 6: LE INTERRUZIONI DI SEQUENZA

Queste istruzioni, che si chiamano anche "salti", danno l'aspetto dinamico dei programmi, che senza di loro avrebbero semplicemente un andamento di "ordine" e di "stile".

Fai questo,
fai quello,
e poi ancora questo,
e solo dopo, quello.....

Ora noi possiamo scrivere:

Se...questo, allora fai quello, altrimenti....

Possiamo distinguere tre gruppi di salti:
— i salti propriamente detti,
— le chiamate ai sottoprogrammi,
— i ritorni dai sottoprogrammi.

Ma anche tre modi di salti:
— assoluti,
— relativi,
— in pagina zero.

I SALTI

Vi ricordate delle istruzioni JP? Eccole:

Indirizzamento	Istruzioni		Flag
assoluto (diretto)	JP ADR		inalterati
	JP C,ADR	JP NC,ADR	
	JP Z,ADR	JP NZ,ADR	
	JP M,ADR	JP P,ADR	
	JP PE,ADR	JP PO,ADR	

La prima istruzione (**JP ADR**) si chiama salto incondizionato: provoca il salto sistematico all'indirizzo ADR. Naturalmente a questo indirizzo ci deve essere un programma.....!
Le istruzioni seguenti si chiamano salti condizionati, cioè il salto viene effettuato solo se la condizione richiesta è soddisfatta.

Naturalmente le istruzioni di salto condizionato si possono presentare in un programma solo dopo istruzioni che posizionano in qualche modo i flag.

```
....
CP      8
JP      Z,UGUAL         ; salta se A = 8
JP      C,INF           ; salta se A < (
....                    ; A > 8
```

In questo esempio, l'istruzione CP posiziona i flag Z e C. Il salto condizionato JP Z controlla la presenza del flag Z e provoca il salto all'indirizzo UGUAL se Z è a 1, segnalando in questo modo che il contenuto dell'accumulatore è uguale a 8. In caso contrario, il programma prosegue in sequenza e "cade" sull'istru-

zione JP C che controlla lo stato del flag C. Se questo è a 1, si effettua un salto all'indirizzo INF. Altrimenti possiamo essere sicuri che il contenuto di A è maggiore di 8, poichè non soddisfa i test precedenti, e il programma prosegue in sequenza.

Indirizzamento	Istruzioni	Flag
indiretto	JP (HL) JP (IX) JP (IY)	inalterati

Finora abbiamo utilizzato come operando dell'istruzione JP un indirizzo di salto. D'ora in poi potremo effettuare un salto non più solo ad un indirizzo dato, ma anche ad un indirizzo calcolato, **contenuto** in uno dei registri HL, IX o IY.

Esempio:

```
LD      HL,(ADR)
INC     HL
LD      A,L
AND     0FEH
LD      L,A
JP      (HL)
```

In questo esempio, si effettuerà un salto ad un indirizzo a priori sconosciuto, di cui si sa solo che sarà sempre pari. Anche qui bisognerà agire con molta saggezza e diffidare degli imprevisti!

Ma ecco venirci incontro saltellando...un gruppo di istruzioni che vi risveglierà qualche ricordo: i salti relativi.

Indirizzamento	Istruzioni		Flag
relativo	JR DISP		inalterati
	JR C,DISP	JR NC,DISP	
	JR Z,DISP	JR NZ,DISP	
	DJNZ DISP		

DISP rappresenta una label che indica l'indirizzo dove si desidera saltare. Il programma assemblatore, che è intelligente, calcolerà la differenza tra l'indirizzo corrispondente a questa label e l'indirizzo dell'istruzione di salto, e la tradurrà nel codice oggetto sotto la forma di un byte contenente un numero con segno, che permette uno "spiazzamento" di 127 byte in avanti e 128 indietro.

Le istruzioni JR della tabella sono in tutto identiche alle istruzioni JP studiate prima, salvo che il codice generato è più corto e che il programma che le contiene può essere tradotto facendo a meno di certe precauzioni.

```
        LD    B,0
LOOP    INC   B
        JR    NZ,LOOP
```

Il piccolo programma precedente rappresenta quello che si chiama un LOOP (ciclo). Fin quando il registro B è diverso da zero, si salta all'indirizzo LOOP. Ma inevitabilmente B raggiugerà il valore OFFH, e dopo essere incrementato di 1, ritornerà a zero. Allora Z sarà posto a 1, il salto, questa volt non sarà effettuato, e il programma uscirà da questo "loop" infernale.
L'ultima istruzione: **DJNZ** è un po' particolare (Decrement and Jump if Not Zero), e utilizza il registro B per raggiungere il suo scopo, in modo che il programma precedente si potrà adesso scrivere:

```
        LD    B,0
LOOP    DJNZ  LOOP
        ......
```

Qui si decrementa B invece che incrementarlo, ma il risultato è lo stesso. È più semplice, no?

LE CHIAMATE A SOTTOPROGRAMMI

La CALL dell'assembler è la GOSUB del BASIC. Queste istruzione che è stata citata nei primi capitoli, effettua un salto al sottoprogramma il cui indirizzo è specificato nell'operando, non senza aver prima effettuato un salvataggio nell'area di stack dell'indirizzo di ritorno.
Quest'ultimo punto differenzia la CALL da una semplice JP, con la quale per altro ci sono analogie sorprendenti.
Così c'è una CALL incondizionata e delle CALL condizionate. Invece non ci sono CALL che utilizzano il modo di indirizzamento relativo. E questo è fastidioso. Anzi, esiste un solo modo di indirizzamento: quello diretto.

Indirizzamento	Istruzioni		Flag
assoluto (diretto)	CALL ADR		inalterati
	CALL C,ADR	CALL NC,ADR	
	CALL Z,ADR	CALL NZ,ADR	
	CALL P,ADR	CALL M,ADR	
	CALL PO,ADR	CALL PE,ADR	

Non è poi così male!

Esempio:
```
LD      BC,VAL
CALL    MOLT
LD      (RISULT),BC
```

Possiamo immaginare che questo piccolo programma, dopo aver caricato due numeri nei registri B e C, chiami una routine che ne esegue la moltiplicazione e ne carica il risultato nel registro doppio BC.
Al momento dell'esecuzione della CALL, l'indirizzo dell'istruzione seguente (LD (RISULT),BC) è inserito nell'area di stack, e viene effettuato un salto all'indirizzo MOLT. Alla fine dell'esecuzione della routine un'istruzione di RETurn provoca una estrazione dalla area di stack, e quindi un salto sull'istruzione LD (RISULT),BC. Geniale, vero?
Si può anche desiderare di chiamare la routine MOLT solo se il registro A è diverso da zero, per esempio. In questo caso scriveremo:

```
CP      0
CALL    NZ,MOLT
```

e il gioco è fatto!

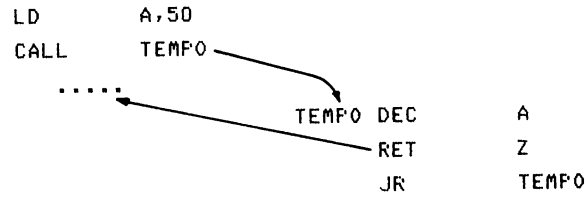
I RITORNI DAI SOTTOPROGRAMMI

È molto bello poter cambiare un sottoprogramma, ma bisogna anche poter tornare indietro. Le istruzioni seguenti sono fatte per questo.

Indirizzamento	Istruzioni		Flag
indiretto tramite stack	RET		inalterati
	RETI		
	RETN		
	RET C	RET NC	
	RET Z	RET NZ	
	RET P	RET M	
	RET PO	RET PE	

Quando viene eseguita un'istruzione di ritorno, il microprocessore estrae un indirizzo dall'area di stack e lo carica nel registro PC che è (ve ne ricordate) il puntatore delle istruzioni. Il programma prosegue quindi a partire da questo indirizzo.
Le prime tre istruzioni della tabella sono ritorni incondizionati da sottoprogrammi. RETI e RETN sono usati nei sottoprogrammi di servizio delle interruzioni, e non diremo altro.

Le istruzioni seguenti sono anch'esse dei ritorni, ma solo a certe condizioni. Nel caso che le condizioni non siano verificate, non producono alcun effetto (come le istruzioni di salto condizionato in generale).



Il programma precedente chiama una routine che determina un tempo proporzionale al valore che si trova nell'accumulatore. Se noi avessimo scritto: LD A,100, il tempo passato nella routine sarebbe stato praticamente il doppio di questo.

Vediamo che nella routine TEMPO il registro A è decrementato fino a quando raggiunge un valore nullo. Solo in questo caso sarà effettuto il ritorno al programma chiamante. Avremmo potuto anche scrivere:

```
TEMPO   DEC   A
        JR    NZ,TEMPO
        RET
```

non è niente di strano: probabilmente ci avevate già pensato!

LE CHIAMATE AI SOTTOPROGRAMMI IN PAGINA ZERO

Sono delle CALL, che presentano due sostanziali differenze:

— il codice operativo di queste istruzioni occupa solo un byte, contro i tre occupati delle CALL normali;
— si riferiscono a sottoprogrammi che iniziano a indirizzi fissi situati all'inizio della memoria (pagina zero).

Indirizzamento	Istruzioni	Flag
diretto	RST 0 RST 8 RST 10H RST 18H RST 20H RST 28H RST 30H RST 38H	inalterati

Come si può immaginare, i sottoprogrammi chiamati saranno memorizzati agli indirizzi 0,8,10H,.....e 38H. Nel modo 0 dello Z80, che è poi quello dell'8080, l'arrivo di un'interruzione comporta una chiamata a uno di questi indirizzi che, in questo caso, contengono delle routine di servizio delle interruzioni. Queste istruzioni prendono anche il nome di ReSTart.

Attenzione, però: questi sottoprogrammi sono, per la maggior parte del tempo, riservati al Sistema Operativo, e memorizzati su ROM.

FAMIGLIA 7: INGRESSI/USCITE

Esistono solo due istruzioni, nei microprocesori, che permettono il dialogo con il mondo esterno (periferiche):

— l'istruzione IN per l'ingresso,
— l'istruzione OUT per l'uscita.

(I termini ingresso e uscita sono sempre relativi al microprocessore e sono, naturalmente, invertiti se si riferiscono alle periferiche).

Il dialogo avviene nel modo seguente: il microprocessore invia il contenuto di un registro verso un dispositivo periferico che corrisponde all'indirizzo dato nell'operando dell'istruzione OUT, o riceve in questo registro l'informazione inviata dal dispositivo periferico il cui indirizzo figura nell'operando dell'istruzione IN.

Il registro in questione, è di solito, l'accumulatore, ma lo Z80 permette l'utilizzo di tutti i registri semplici da A a L.

In generale, l'indirizzo della periferica è specificato direttamente nell'operando dell'istruzione, su 8 bit. Ma ancora, lo Z80 permette di utilizzare il registro C per contenere un indirizzo calcolato.

Indirizzamento	Istruzioni	Flag
diretto a 8 bit	OUT (VAL8),A IN A,(VAL8)	inalterati

Il contenuto del registro A è inviato alla periferica il cui indirizzo è VAL8, nel caso dell'istruzione OUT; nel caso dell'istruzione IN il registro A viene caricato con l'informazione emessa dalla periferica di indirizzo VAL8.

Indirizzamento	Istruzioni		Flag
indiretto tramite registro 8 bit	OUT (C),A OUT (C),B OUT (C),C OUT (C),D OUT (C),E OUT (C),H OUT (C),L	IN A,(C) IN B,(C) IN C,(C) IN D,(C) IN E,(C) IN H,(C) IN L,(C)	non modificati da OUT per IN S,Z,H,P modificati C inalterati N = 0

Nel caso dell'istruzione OUT il contenuto del registro (da A a L) è inviato verso la periferica il cui indirizzo si trova nel registro C. In questo caso l'indirizzamento della periferica si può considerare indiretto (tramite registro puntatore). Il contrario succede con l'istruzione IN.

Indirizzamento	Istruzioni		Flag
indiretto tramite registro con incremento/ decremento	INI OUTI	IND OUTD	N = 1 C inalterato Z = 1 se B = 0 gli altri indeterminati

Questo nuovo tipo di istruzioni non ci sono completamente sconosciute. Le abbiamo già incontrate nel caso delle istruzioni LD e CP.
L'indirizzo della periferica è sempre contenuto nel registro C, ma l'informazione emessa (OUT) o ricevuta (IN), non si trova più in un registro, ma nella locazione di memoria puntata da HL. L'indirizzamento è dunque doppiamente indiretto, sia per l'indirizzo che per il dato. Ad ogni esecuzione di una di queste istruzioni, il registro B è decrementato di 1,HL è incrementato dalle istruzioni OUTI e INI e decrementato dalle istruzioni OUTD e IND.

Indirizzamento	Istruzioni	Flag
	vedere le istruzioni speciali su blocchi di dati	

FAMIGLIA 8: ISTRUZIONI SPECIALI SUI BIT E SU BLOCCHI DI DATI

Questa famiglia comprende, in effetti, alcune istruzioni speciali che non si incontrano normalmente con gli altri microprocessori, e a volte neanche con elaboratori più potenti.

ISTRUZIONI SUI BIT

Di norma il microprocessore opera su valori di 8 bit, sia che si trovino in un registro che in memoria; in casi eccezionali opera su valori di 4 bit.
Le istruzioni sui bit consentano di aumentare il grado di precisione, realizzando le tre funzioni seguenti:

— mettere a 1 un bit (istruzione SET),
— mettere a zero un bit (istruzione RES),
— testare il valore di un bit (istruzione BIT).

Il bit in questione può trovarsi indifferentemente in un registro o in memoria.

Indirizzamento		Istruzioni	
	test	bit a 1	bit a 0
registri a 8 bit	BIT b,A BIT b,B BIT b,C BIT b,D BIT b,E BIT b,H BIT b,L	SET b,A SET b,B SET b,C SET b,D SET b,E SET b,H SET b,L	RES b,A RES b,B RES b,C RES b,D RES b,E RES b,H RES b,L
flag	Z modificato N = 0, H = 1 C inalterato	inalterati	inalterati

Il simbolo **b** rappresenta il numero del bit, e assume i valori da 0 a 7

```
! 7 ! 6 ! 5 ! 4 ! 3 ! 2 ! 1 ! 0 !
+---+---+---+---+---+---+---+---+
```

L'istruzione BIT carica nel flag Z l'inverso del bit testato: se b = 1, Z = 0 e viceversa.

Esempio:

```
LD      A,8
SET     1,A
BIT     1,A
JR      NZ,UGUALE
```

L'istruzione SET forzerà a 1 il bit 1 dell'accumulatore, che quindi conterrà il valore 0AH. L'istruzione BIT effettuerà poi il test del bit 1, e il flag Z sarà posizionato a 0 (poichè questo bit è a 1....). Verrà poi effettuato un salto all'indirizzo UGUALE.

Indirizzamento	Istruzioni		
indiretto e indiretto indicizzato	test	bit a 1	bit a 0
	BIT b,(HL) BIT b,(IX + d) BIT b,(IY + d)	SET b,(HL) SET b,(IX + d) SET b,(IY + d)	RES b,(HL) RES b,(IX + d) RES b,(IY + d)
flag	Z modificato N = 0, H = 1 C inalterato	inalterati	inalterati

Sono le stesse operazioni della tabella precedente, ma il bit in questione questa volta si trova in memoria, nella locazione puntata da uno dei registri specificati nell'operando.

ISTRUZIONI SUI BLOCCHI DI DATI

Queste istruzioni, al contrario di quelle sui bit che operano su quantità inferiori al byte, operano su blocchi, cioè insiemi di byte consecutivi, ovvero su zone di memoria.

Basta molto poco, per esempio, perchè l'istruzione LDI, che già effettuava un certo numero di operazioni, si trasformi in un'operazione su blocchi. Questo passo è stato fatto!

Distingueremo tre gruppi di istruzioni sui blocchi:

— le istruzioni di trasferimento,
— le istruzioni di confronto (ricerca),
— le istruzioni di Ingresso/Uscita.

I codici mnemonici di queste istruzioni terminano con il suffisso "**R**", indicando così una Ripetizione del processo.

Trasferimento

Indirizzamento	Istruzioni		Flag
indiretto su blocchi	LDIR	LDDR	N, P, H = 0 gli altri invariati

Queste istruzioni permettono di trasferire una zona di memoria da un indirizzo ad un altro, partendo dall'inizio (LDIR) o dalla fine (LDDR). Per fare questo sono utilizzati i seguenti registri:

```
HL -----> indirizzo della zona sorgente
DE -----> indirizzo della zona destinazione
BC -----> lunghezza della zona in byte
```

Alla fine dell'operazione, BC è uguale a 0, e HL/DE sono stati incrementati o decrementati (LDIR/LDDR) del valore contenuto inizialmente in BC, più 1. È ormai un gioco da bambini scrivere:

```
LD      HL,ZONA1
LD      DE,ZONA2
LD      BC,LUNGH
LDIR
```

per trasferire la zona 1 nella zona 2, per una lunghezza LUNGH.

Ricerca

Queste due istruzioni consentono di ricercare tra i byte di un blocco puntato da HL un valore uguale a quello contenuto nell'accumulatore. Il processo termina quando si trova il valore cercato, oppure quando si raggiunge la fine del blocco. Come per le istruzioni precedenti, il blocco può essere esaminato dall'alto al basso, (secondo gli indirizzi crescenti, con la CPIR) o dal basso all'alto (con la CPDR).

Indirizzamento	Istruzioni		Flag
indiretto su blocchi	CPIR	CPDR	C inalterato N = 1 S e H modificati V = 0 se BC = 0 Z = 1 se si trova il valore cercato

Nel caso dell'istruzione CPIR, il contenuto del registro A è confrontato con quello del byte di memoria puntato da HL. BC viene decrementato e HL incrementato. Se i due valori sono uguali, l'istruzione ha termine, e Z viene messo a 1, altrimenti la ricerca continua fino a trovare il valore cercato, oppure la fine del blocco. In questo caso BC contiene il valore zero, e V è messo a zero.

Esempio: Supponiamo che il blocco di byte 010203040506H sia memorizzato a partire dall'indirizzo 4000H.

```
LD      HL,4000H
LD      BC,6
LD      A,3
CPIR
```

Si tratta dunque di ricercare il valore 3 nel blocco di byte che si trovano a partire dall'indirizzo 4000H. Quando l'esecuzione del programma è terminata, HL contiene 4003H, BC contiene 3 e Z è a 1, segnalando che il byte è stato trovato. P/V non è messo a zero perchè non è stata raggiunta la fine del blocco.

Ingresso/Uscita

Queste istruzioni consentono di dialogare con una periferica, non più solo con un byte, ma con una zona di memoria da 1 a 256 byte.

Indirizzamento	Istruzioni		Flag
indiretto su blocchi	OTIR	OTDR	C inalterato N = Z = 1
	INIR	INDR	gli altri invariati

L'indirizzo della periferica è contenuto nel registro C, e l'indirizzo del blocco da emettere o da ricevere, nel registro HL. Come per le altre istruzioni di blocco, la penultima lettera del codice mnemonico, indicherà se l'indirizzo del blocco si deve Incrementare o Decrementare.

Esempio: Si debba inviare il blocco di dati ASCII "BUONGIORNO", situato in memoria a partire dall'indirizzo 4000H, alla periferica di indirizzo 7.

```
LD      HL,4000H
LD      C,7
LD      B,10
OTIR
```

FAMIGLIA 9: ISTRUZIONI DI GESTIONE DELL'AREA DI STACK

Eccoci arrivati all'ultima famiglia, che per altro non vi è affatto sconosciuta, in quanto abbiamo già avuto l'occasione di utilizzare le istruzioni PUSH e POP negli esempi dei primi capitoli.

Indirizzamento	Istruzioni		Flag
indiretto a 16 bit	PUSH AF PUSH BC PUSH DE PUSH HL PUSH IX PUSH IY	POP AF POP BC POP DE POP HL POP IX POP IY	inalterati salvo che da POP AF

Si può dire che l'indirizzamento è indiretto, poichè l'accesso all'area di stack si effettua tramite il registro a 16 bit SP. Non ci tratterremo oltre su queste istruzioni, salvo che per segnalare che permettono anche lo scambio tra i registri doppi:

```
PUSH    IX
PUSH    BC
POP     IX
POP     BC
```

Dopo l'inserimento nell'area di stack di IX e BC, si effettua un prelevamento nello stesso ordine, e questo consente uno scambio tra i due registri IX e BC.

Dopo aver "accatastato" tutte queste istruzioni in questo capitolo, è necessario ora riconsiderarle una per una.....Gli esercizi seguenti servono a questo, non trascurateli!

ESERCITAZIONI PRATICHE DEL CAPITOLO 3

Siete ora a conoscenza del set completo di istruzioni che vi permetteranno di scrivere qualsiasi programma in assembler.

Questa nuova serie di esercizi vi aiuterà a fare il punto sulle nuove conoscenze acquisite.

Come potrete constatare, non esiste un'unica soluzione per un problema, e può darsi che voi troviate soluzioni più astute di quelle proposte.

A voi il compito di trovare varianti interessanti! E se non trovate alcuna soluzione, chiudete il libro e riprendetelo fra uno o due giorni; dovrebbe andare meglio. Ma soprattutto non fatevi prendere dal panico: è una cosa che si cura!

ESERCIZIO 3.1; *Scrivere un programma sotto forma di subroutine, che realizzi la conversione di un simbolo esadecimale in forma ASCII nel suo equivalente in binario. Esempio: 41H (lettera A in ASCII) deve diventare OAH.*
Il registro A sarà utilizzato in ingresso e in uscita della subroutine per contenere i dati.

ESERCIZIO 3.2: *Utilizzando la subroutine precedente, convertire questa volta un numero esadecimale da 00H a 0FFH, nel registro BC ed espresso in ASCII, nel suo equivalente binario. Il risultato va nel registro A. Esempio:*

BC = 3941H ——> A = 9AH

ESERCIZIO 3.3: *L'istruzione CP consente di confrontare numeri lunghi un byte. Scrivere una subroutine che confronti due numeri memorizzati nei registri HL e DE (confronto su 16 bit).*

ESERCIZIO 3.4: *Scrivere una subroutine che confronti due zone di memoria puntate da HL e DE, la cui lunghezza è nel registro C.*

ESERCIZIO 3.5: *Scrivere un programma che consente di moltiplicare il contenuto del registro HL per 10. Il risultato va in HL.*

CAPITOLO 4

PSEUDO - ISTRUZIONI E MACRO - ISTRUZIONI

Questi nuovi comandi obbediscono alle stesse regole di sintassi delle istruzioni: il loro formato sorgente possiede anch'esso i campi label, operazione, operando e commento.

Il termine "pseudo" indica che avremo a che fare con delle "specie" di istruzioni (ma non proprio), pur essendo molto simili alle istruzioni ... e il termine "macro" indica che sono semplicemente delle specie di istruzioni "giganti" ...

Le *pseudo-istruzioni*, chiamate anche *direttive*, sono dei comandi diretti non al microprocessore, come è per le istruzioni, ma al programma assemblatore stesso.
Quęsti comandi consentono di imporre certe regole circa la stampa del listing, la memorizzazione del programma, la definizione di costanti o di dati in memoria, offrendo contemporaneamente alcune facilitazioni nella scrittura del codice sorgente.

Le *macro-istruzioni*, invece, elevano il grado di evoluzione del linguaggio assembler, abbreviando la scrittura del codice sorgente, con l'uso di parole nuove, realizzando funzioni predefinite dal programmatore.

Quando una sequenza di istruzioni si ripete in un programma (com'è stancante ...!) basta dichiararla come macro-istruzione, assegnandole un nome.

Ogni volta che questo nome sarà chiamato in seguito, la sequenza di istruzioni corrispondente sarà prodotta al posto di questo nome. L'assembler, in questo caso, prende il nome di *macro-assembler.*

Ma ricominciamo da capo.

LE DIRETTIVE (O PSEUDO-ISTRUZIONI)

Non le descriveremo tutte (ce n'è ogni giorno una nuova, come se facessero dei figli ...) ma ci limiteremo alle più usate.

Prima di tutto quelle che sono d'obbligo in ogni assembler che si rispetti:

DEFINIZIONE DELL'ORIGINE: ORG

Questa direttiva permette di fissare l'indirizzo d'origine del programma-o di una porzione del programma-in memoria. Il suo operando indica l'indirizzo di memoria al quale deve essere generato il codice oggetto che segue.

Esempio:

```
ORG     1000H
LD      A,2
```

L'istruzione LD A,2 sarà memorizzata a partire dall'indirizzo 1000H della memoria. Se è necessario, possono essere utilizzati più comandi ORG:

```
ORG     0
JP      PROG1
ORG     8
JP      PROG2
.........
```

In assenza di direttive ORG, il codice oggetto è generato a partire dall'indirizzo 0. Il campo operando può contenere un'espressione, a patto che sia completamente definita nella prima passata dell'assemblatore.

FINE DEL PROGRAMMA SORGENTE: END

È l'ultima linea del programma sorgente. Ha la funzione di segnalare all'assemblatore che è stata raggiunta la fine del programma.

Il suo operando precisa l'indirizzo dal quale il programma dovrà essere lanciato una volta caricato in memoria, cioè l'indirizzo della prima istruzione che deve essere eseguita (che non è necessariamente la prima del programma).

Esempio:

```
ROUT    LD      A,4
        RLCA
        AND     B
        RET
INIZ    LD      SP,4000H
        .......
        END     INIZ
```

Dopo il caricamento di questo programma in memoria, sarà eseguita l'istruzione situata all'indirizzo INIZ.

L'operando non è necessariamente una label, ma può avere la forma di un indirizzo esadecimale o decimale, anche se questo si trova al di fuori del programma.

```
END   100H
```

DEFINIZIONE DI EQUIVALENZA: EQU e DEFL

Quando si preferisce manipolare dei simboli-sempre più espliciti- piuttosto che dei numeri, in un programma, si usa la direttiva EQU, che assegna un valore definito e immutabile ad un simbolo.

Esempio:

```
ABUF    EQU   4000H
FINBUF  EQU   4FFFH
LONG    EQU   FINBUF-ABUF+1
N       EQU   4
        SET   N,A
        ........
```

Ma quando è stata definita un'equivalenza, non può più essere modificata durante l'esecuzione di un programma. Tuttavia c'è un modo per far questo: se si usa la direttiva DEFL al posto della EQU l'assemblatore non produce errore di doppia assegnazione:

```
N       DEFL  4
        SET   N,A
N       DEFL  7
        BIT   N,A
        .......
```

DEFINIZIONE DI DATI: DEFB, DEFW e DEFM

In certi assembler si chiamano anche: BYTE, WORD e ASCII. Contrariamente alle precedenti queste pseudo-istruzioni producono un codice oggetto.

In un programma non ci sono necessariamente solo istruzioni: possono anche esserci dei dati. Ricordatevi il testo "BUONGIORNO" degli esempi del Capitolo I. Oh! com'è lontano!.

Possiamo ora definirlo nel modo seguente:

```
DEFM        "BUONGIORNO"
```

e l'assemblatore produrrà una stringa ASCII corrispondente a questo testo (DEFM:**DEF**ine Message)

Nello stesso modo possiamo definire un byte o una parola (word) di due byte:

```
        ORG   4000H
BYTE    DEFB  41H              ; DEFinizione di un Byte
WORD    DEFW  0AFFH             ; DEFinizione di una parola
                                ; (Word)
INIZ    LD    A,(BYTE)
        LD    BC,(WORD)
```

Il registro A conterrà il valore 41H e il registro BC conterrà il valore 0AFFH.

Attenzione: non confondere con LD BC,WORD che significa che BC conterrà l'indirizzo di memoria corrispondente alla label WORD, cioè 4001H.

Quando il valore da definire è in ASCII, e corrisponde ad un carattere stampabile, possiamo scrivere:

BYTE DEFB "A"

Naturalmente le label possono anche loro partecipare alle danze:

LG DEFB FBUF - BUF

Attenzione: in questo caso il valore risultante non deve superare OFFH.

```
APROG   DEFW  PROG
INIZ    LD    HL,(APROG)
        PUSH  HL
        RET
PROG    RLCA
  .........
```

È un RET "bidone"!

Ma prendiamo un esempio concreto: supponiamo di dover emettere verso una periferica un testo in ASCII. La chiamata alla routine VISU provocherà l'invio di un carattere del messaggio verso il video, per esempio. Scriveremo:

```
INIZ    LD      HL,ATEXT     ;PUNTA INIZIO TESTO
CONTIN  LD      A,(HL)
        OR      A            ;PER CONTROLLARE Z
        JR      Z,FIN        ;SE Z=1 --->STOP
        CALL    VISU         ;EMISSIONE DI UN CARATTERE
        INC     HL           ;+1 ALL'INDIRIZZO TESTO
        JR      CONTIN       ;CONTINUA
ATEXT   DEFM    'OH!QUANTI MARINAI'
        DEFB    0DH
        DEFM    'QUANTI CAPITANI...'
        DEFB    0
```

Il testo definito nel programma precedente è composto da due linee di caratteri

ASCII, separate dal codice esadecimale 0DH che corrisponde al CR (ritorno carrello, o "a capo"; vedi il codice ASCII nell'Appendice). Poichè questo codice non è stampabile, bisogna definirlo nella sua forma esadecimale (o decimale). Nel nostro esempio il testo termina con il carattere 0, che ne indica la fine (avrebbe potuto essere qualsiasi altro codice).

Questo programma emette dunque un messaggio, carattere per carattere, tramite la routine VISU che è specifica per quella periferica, (Una routine del genere si chiama DRIVER) fino a quando incontra lo zero, che segnala la fine del testo. Allora si effettua un salto all'indirizzo FINE, che permette di uscire dal LOOP (l'indirizzo FINE non è rappresentato nell'esempio).

Noterete che il testo deve essere delimitato da due apostrofi.
Se questo carattere deve comparire all'interno del testo, è necessario ripeterlo nel codice sorgente.

Esempio: DEFM "L'ASSEMBLER È FACILE'

darà errore nell'assemblaggio, e dovrà invece essere scritto:

DEFM "L'"ASSEMBLER È "FACILE"

Notiamo infine che certi assembler ammettono più argomenti nelle DEFB e DEFW:

```
DEFB    0,1,0ACH,7,ADR, '*'
```

RISERVA DI SPAZIO: DEFS

Questa direttiva permette di riservare il posto per un certo numero di byte in memoria:

```
DEFS    257
```

riserverà una zona di 257 byte nel programma. In pratica il puntatore degli indirizzi (program counter) viene incrementato di questo valore dall'assemblatore.

Attenzione alle sorprese: lo spazio riservato non è necessariamente messo a zero ...

```
BYTE1   DEFB   4
        DEFS   10
BYTE2   DEFB   2
```

Se BYTE1 è, per esempio, all'indirizzo di memoria 1F4H, BYTE2 sarà all'indirizzo 1FFH.

Le direttive che studieremo ora sono presenti solo negli assembler "ben dotati" e nei micro-assembler. I loro nomi variano da un assembler ad un altro, ma la cosa importante è sapere che esistono e che cosa fanno.

DIRETTIVE CHE AGISCONO SULLA STAMPA DEL LISTING

PAGE (o SEJECT) forza il listing dell'assemblaggio a proseguire nella pagina seguente (salto di pagina). Talvolta è possibile definire, contemporaneamente, il numero di linee per pagina:

```
        PAGE    66
ADR     LD      SP,40FFH
        .......
```

TITLE (o STITLE) permette di stampare sistematicamente all'inizio di ogni pagina del listing, un testo dato:

```
TITLE "PROGRAMMA DI GESTIONE . GIUGNO 1982"
```

SUBTTL (o STITLE) è identica alla precedente, ma stampa un sottotitolo, dopo il testo del titolo.

NOLIST e **LIST** consentono di interrompere o di riprendere la stampa del listing:

```
        LD      SP,4FFFH
        NOLIST
; questa parte del programma
; non sara' stampata
        LIST
        PUSH    AF
        ......
```

DIRETTIVE DI ASSEMBLAGGIO CONDIZIONATO

Capita spesso che un programma contenga delle sequenze opzionali che devono o non devono essere assemblate, a seconda della configurazione desiderata.

Si può, per esempio, considerare un programma la cui configurazione "mini" consente di stampare dei risultati su di un video, mentre la configurazione "maxi" li stampa sul video e sulla stampante.

Per evitare di scrivere due programmi quasi identici, basterà inserire all'inizio del codice sorgente:

```
MINI    EQU     0
MAXI    EQU     1
```

precedute dalla scelta della configurazione:

oppure:

```
CONFIG  EQU     MAXI
CONFIG  EQU     MINI
```

per poter poi usare le direttive che permettono di assemblare o no alcune sequenze del programma. Queste direttive vi ricorderanno forse qualcosa: IF e ELSE?

Il test sul valore di un simbolo dirà se le linee che seguono devono essere assemblate o no:

```
IF      il simbolo e' vero...
si assembla questo
ELSE   (altrimenti)
si assembla quello
ENDIF
```

(la ENDIF indica la fine della sequenza sorgente che è oggetto dell'assemblaggio condizionato).

Nell'esempio precedente, noi scriveremo:

```
IF   CONFIG = MAX1
.....                    ; stampa sulla stampante
.....
ENDIF
.....                    ; stampa sul video
```

La ELSE in questo caso non è di nessuna utilità.

Altro esempio:

```
INCDEC  EQU     1
        ....
        ....
        IF      INCDEC=1
        INC     A
        ELSE
        DEC     A
        ENDIF
```

Modificando la prima linea del programma l'assemblatore produrrà l'istruzione INC A o DEC A a seconda del risultato del test.

DIRETTIVE DI RIPETIZIONE

Le linee sorgente comprese tra le direttive **REPEAT** (o RPT) e **ENDR** sono ripetute un numero di volte precisato nell'operando:

```
        REPEAT 3
        DEFB   2
        ENDR
```

produrrà:

```
        DEFB   2
        DEFB   2
        DEFB   2
```

Analogamente, scrivendo:

```
N       DEFL   0
        REPEAT 3
        DEFB   N
N       DEFL   N+1
        ENDR
```

l'assemblatore produrrà:

```
        DEFB   0
        DEFB   1
        DEFB   2
```

DIRETTIVE DI CARICAMENTO DI UN MODULO SORGENTE

La direttiva **INCLUDE** (o LOAD), quando viene incontrata, provoca la chiamata e l'inserimento nel codice sorgente di un altro programma sorgente inviato da una periferica (in particolare un disco), il cui nome compare nell'operando della direttiva:

```
LD      SP,0FFFFH
INCLUDE PROG1
.....
```

Il modulo sorgente PROG1 verrà inserito nel codice sorgente all'indirizzo della chiamata.

DICHIARAZIONE E DEFINIZIONE DI SIMBOLI ESTERNI

Sono le direttive:

EXTERNAL (o EXT o EXTRN)
GLOBAL (o ENTRY o PUBLIC)

Dovete sapere che se si fa riferimento, in un programma, ad un simbolo assente da questo programma, l'assemblatore segnala sistematicamente un errore. D'altra parte, può essere interessante assemblare separatamente diversi pezzi di un programma, per riunirli in seguito in un unico insieme.
Vedremo che questo può essere possibile grazie ad un programma di utilità che si chiama "**editor di collegamento**" (LINK).

Ma in questo caso, come si può scrivere, per esempio:

```
CALL    ROUT1
```

se ROUT1 si trova in un'altra porzione del programma sorgente, senza provocare errore di assemblaggio?

Basterà dichiarare, nel nostro caso, che ROUT1 è esterno al programma attuale:

```
        EXTERNAL ROUT1
        ........

        CALL     ROUT1
```

Nel modulo in cui si trova questa routine, sarà inoltre necessario segnalare che il simbolo ROUT1 può essere chiamato da un modulo esterno:

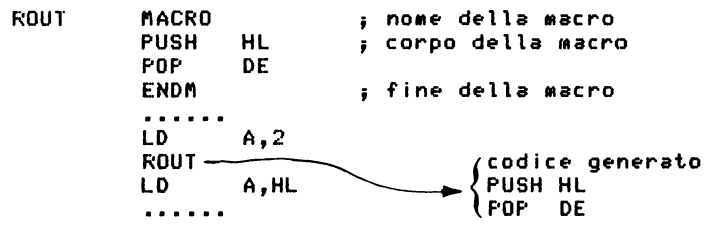
```
        GLOBAL  ROUT1
ROUT1   LD      A,1
        ......
```

Quando queste due condizioni sono entrambe presenti, l'assemblatore non segnala errore.

LE MACRO-ISTRUZIONI

Per definire una macro, si utilizzano due direttive: **MACRO** e **ENDM**, che delimitano la lista delle linee del codice sorgente che formano il corpo della macro.

Esempio:



In questo esempio, la macro di nome ROUT è formata dalle due istruzioni PUSH HL e POP DE. La semplice comparsa di questo nome nella zona operazione del codice sorgente provoca la generazione, da parte dell'assemblatore, della lista delle istruzioni definite precedentemente, che vengono a inserirsi nel programma.

Attenzione: non bisogna fare confusione tra la macro-istruzioni e i sotto-programmi. Le prime hanno l'unico scopo di semplificare e abbreviare la scrittura del codice sorgente, ma non hanno nessuna incidenza sulla lunghezza del codice oggetto, come invece hanno i sotto-programmi.

L'utilizzo delle macro sarebbe relativamente ridotto se non fosse possibile renderle parametriche ...

Un esempio ci chiarirà questo concetto:

```
ROUT    MACRO  P
        LD     &P,2
        AND    (HL)
        ENDM
```

Nella macro definita sopra, P rappresenta il nome (qualsiasi) di un parametro che bisognerà sostituire, al posto del segno & con il simbolo fornito nella chiamata della macro.

Se, per esempio, scriviamo:

```
ROUT    A
```

l'assemblatore produrrà:

```
LD      A,2
AND     (HL)
```

mentre scrivendo:

```
ROUT    B
```

avrebbe prodotto:

```
LD      B,2
AND     (HL)
```

Si possono naturalmente "passare" più parametri:

```
ROUT    MACRO  P,X,Y
LABEL&X LD     &P,&Y
        AND    (HL)
        INC    HL
        JR     NZ,LABEL&X
        ENDM
```

e la chiamata:

```
        ROUT   A,1,0AH
```

produrrà la sequenza:

```
LABEL1  LD      A,0AH
        AND     (HL)
        INC     HL
        JR      NZ,LABEL1
```

Super-macro-geniale!
Ma immaginate un po' la seguente definizione:

```
MPROG   MACRO
LAB1    LD      A,(HL)
        AND     A
        JR      NZ,LAB1
        ENDM
```

Che cosa succederà se questa macro viene chiamata più di una volta nel programma-cosa che certamente capiterà? La label LAB1 sarà generata più di una volta, e l'assemblatore, intransigente di fronte a tale ignominia, segnalerà errore di doppia definizione di etichetta!

Bisognerà allora usare una nuova direttiva, per segnalare che l'utilizzo di questa label è interno (o locale) alla macro, e si scriverà:

```
MPROG   MACRO
        LOCAL   LAB1
LAB1    LD      A,(HL)
        .....
```

e il problema sarà risolto!

Ultimo punto a proposito delle macro:

Può capitare di dover uscire prematuramente da una macro, per esempio in seguito a un test di una condizione IF.

La direttiva **EXITM** sarà utilizzata per questo scopo:

```
PROG    MACRO   N1
        LD      A,1
        IF      &N1=0
        EXITM
        AND     (HL)
        ENDIF
        .....
```

Non è una buona occasione per fare un EXIT dal Capitolo 4?

CAPITOLO 5

TECNICHE E PRATICA DELL'ASSEMBLER

In questo capitolo, essenzialmente pratico, descriveremo con degli esempi, alcune delle principali tecniche di programmazione adoperate implicitamente in tutti i programmi scritti in assembler.

Così ci troveremo a parlare di loop, tabelle, ricerche, sottoprogrammi, calcoli, conversioni, gestione degli input/output ... e ci accorgeremo di utilizzare talvolta queste tecniche con grande facilità.

La programmazione non è ingombra di nomi altisonanti contrariamente alla matematica, dove per risolvere questo o quel problema, dovete appoggiarvi su Talete, passando da Kuriosevic, e che grazie al teorema di Stroupf e Williams, siete giunti infine a dimostrare che il lemma di Plucker modificato da Schwartz poteva, in certi casi ...

No! La programmazione è una cosa personale. Quando avete scritto un programma, sentite che siete proprio voi che l'avete scritto.

In effetti, non ci sono metodi rigidi di programmazione, ma un insieme di metodi "aperti", messi a disposizione del programmatore, che li adatterà più o meno al tipo di problema che deve risolvere.

Così, due programmi scritti separatamente da due persone, possono usare tecniche diverse, e ciononostante risolvere gli stessi problemi. Può darsi che uno sia più veloce dell'altro, o che sia più ingombrante in memoria; è qui che interviene l'arte della programmazione, che si apprende solo con l'esperienza.

Poichè queste diverse tecniche sono più o meno dipendenti le une dalle altre, ci limiteremo in questo capitolo a descrivere e a commentare piccoli programmi

di uso generale. Quando avrete capito il meccanismo di ognuno di questi, sarete sulla buona strada!

```
                00100 ; ROUTINE DI CONVERSIONE
                00110 ; HEXA  ---> 1 BYTE ASCII
                00120 ; INGRESSO:  (A)=CIFRA HEXA
                00130 ; USCITA  :  (A)=BYTE ASCII
                00140 ;
8000            00145          ORG     8000H
8000 C690       00150 HEXASC   ADD     A,90H
8002 27         00160          DAA
8003 CE40       00170          ADC     A,40H
8005 27         00180          DAA
8006 C9         00190          RET
8000            00200          END     HEXASC
00000 TOTAL ERRORS
```

Lo scopo di questo programma è di convertire i numeri esadecimali da 0 a 9 nei corrispondenti codici ASCII, da 30 a 39, e i numeri esadecimali da 0A a 0F nei codici ASCII da 41 a 46 (vedi tabella ASCII in Appendice).

Due esempi ci aiuteranno a capire meglio:

valore esadecimale:	0A	09
	90	90
addizione:	+ 0A	+ 09
	9A	99
aggiustamento decimale DAA:	+ 66	+ 00
	(1)00	99
addizione col carry:	+ 40	+ 40
	+ 1	+ 0
	41	0 D9
azione DAA:	+ 00	+ 60
	41	1 39

E credete che se funziona per questi due casi, funzionerà anche per gli altri!

```
                010   ;CONVERSIONE BINARIO-DECIMALE
                020   ;IN ASCII ,FINO A 5 CIFRE
                030   ;INGRESSO:(HL):IND RISULTATO
                040   ;          (DE):VALORE BINARIO
                050   ;
8000            060            ORG     8000H
8000 01F0D8     070   BINDC5   LD      BC,-10000     ;5 CIFRE
8003 CD1B80     080            CALL    CBD           ;
```

```
8006 0118FC  090   BINDC4  LD      BC,-1000      ;4 CIFRE
8009 CD1B80  100           CALL    CBD           ;
800C 019CFF  110   BINDC3  LD      BC,-100       ;3 CIFRE
800F CD1B80  120           CALL    CBD           ;
8012 01F6FF  130   BINDC2  LD      BC,-10        ;2 CIFRE
8015 CD1B80  140           CALL    CBD           ;
8018 01FFFF  150   BINDC1  LD      BC,-1         ;1 CIFRA
             160   ;
             170   ;ROUTINE DI CONVERSIONE
             180   ;
801B 3E2F    190   CBD     LD      A,'0'-1       ;0 ASCII - 1
801D E5      200           PUSH    HL            ;SALVA IND
801E D5      210           PUSH    DE            ;VALORE BIN
801F E1      220           POP     HL            ;DE--->HL
8020 3C      230   CBD1    INC     A             ;
8021 09      240           ADD     HL,BC         ;
8022 38FC    250           JR      C,CBD1        ;
8024 ED42    260           SBC     HL,BC         ;
8026 EB      270           EX      DE,HL         ;
8027 E1      280           POP     HL            ;MEM RISULT
8028 77      290           LD      (HL),A        ;IN IND
8029 23      300           INC     HL            ;CONTINUA
802A C9      310           RET                   ;RITORNO
8000         320           END     BINDC5
00000 TOTAL ERRORS
```

Questa routine esegue la conversione di un numero di 16 bit contenuto nel registro DE, in una sequenza di caratteri ASCII espressi in forma decimale, e il cui indirizzo è passato tramite il registro HL.

Per utilizzare la routine, basta scrivere:

```
LD      HL,4000H        ; indirizzo della sequenza
LD      DE,0FB3AH       ; numero da convertire
CALL    BINDC5          ; chiamata per la conversione
```

Dopo l'esecuzione, il blocco di byte situati a partire dall'indirizzo 4000H conterrà:

```
4000H:    36    (6 ASCII)
4001H:    34    (4)
4002H:    33    (3)
4003H:    31    (1)
4004H:    34    (4)
```

cioè il numero 64314.

Altro esempio.

```
LD      DE,00ABH        ; numero da convertire
CALL    BINDC5          ; chiamata alla conversione
```

La sequenza risultante sarà: 30 30 31 37 31, cioè 00171. In questo caso era possibile fare:

```
CALL    BINDC5
```

e la sequenza sarebbe stata: 31 37 31, cioè il numero 171.

Esaminando questo programma, vediamo che utilizza una subroutine CBD che viene chiamata tante volte quant'è il numero di cifre desiderato.

Notato che l'ultima chiamata non è fatta con una CALL.

Sarebbe stato più costoso fare:

```
BINDC1  LD      BC,-1
        CALL    CBD
        RET
```

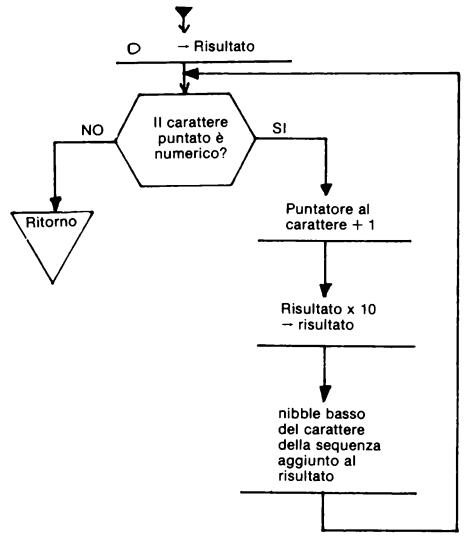
Inoltre la cosa non presentava nessun interesse.

Il metodo di calcolo è semplice: si calcola con una sottrazione quanto manca a 10000 dal numero, e successivamente a 1000, 100, 10, 1; e per ogni "fetta", questo numero è aggiunto al valore 2FH per dare la cifra delle decine di migliaia, delle migliaia, ..., di valore compreso tra 30 e 39 (da 0 a 9 in ASCII).

```
                00130 ; CONVERSIONE DECIMALE-BINARIO
                00140 ; INGRESSO (HL):IND SEQUENZA ASCII
                00150 ;           (DE):RISULTATO BINARIO
                00160 ; FINE DELLA CONVERSIONE SU UN
                00170 ; CARATTERE ASCII NON NUMERICO
                00180 ;
8000            00190           ORG     8000H
8000 110000     00200 DECBIN    LD      DE,0            ;0-->RISULT
8003 7E         00210 DCB1      LD      A,(HL)          ;CARATTERE
                00211                                   ;SEQUENZA
8004 FE30       00220           CP      '0'             ;NUMERICO?
8006 D8         00230           RET     C               ;SE NO,FINE
8007 FE3A       00240           CP      '9'+1           ;E >9?
8009 D0         00250           RET     NC              ;SE SI FINE
800A 23         00260           INC     HL              ;INDIRIZZO
                00261                                   ;CAR SUCC
800B E5         00270           PUSH    HL              ;SALVATO
800C 62         00280           LD      H,D             ;
800D 6B         00290           LD      L,E             ;DE-->HL
800E 19         00300           ADD     HL,DE           ;
800F 29         00310           ADD     HL,HL           ;
8010 19         00320           ADD     HL,DE           ;
8011 29         00330           ADD     HL,HL           ;10*DE-->HL
8012 D630       00340           SUB     '0'             ;NIB BASSO
8014 5F         00350           LD      E,A             ;
```

```
8015 1600     00360          LD       D,0          ;
8017 19       00370          ADD      HL,DE        ;RIS TOT
8018 EB       00380          EX       DE,HL        ;-->DE
8019 E1       00390          POP      HL           ;PUNTATORE
              00391                               ;CARATTERE
801A 18E7     00400          JR       DCB1         ;CONTINUA
8000          00410          END      DECBIN
00000 TOTAL ERRORS
```

Questo programma esegue la conversione inversa del precedente. La sequenza di byte decimali ASCII, memorizzata all'indirizzo puntato da HL, e chiusa da un carattere non numerico, è convertita in un valore binario, fornito in uscita nel registro DE.



Se la zona di memoria che inizia all'indirizzo 4000H contiene: 36 34 33 31 34 00 (numero 64314), la chiamata:

```
LD      HL,4000H
CALL    DECBIN
```

restituirà il valore OFB3AH nel registro DE.

Il programma prima di tutto controlla se ogni cifra della sequenza è compresa tra 30 e 39, cioè 0 e 9 in ASCII, o se si tratta di un carattere di fine sequenza (00).

Per ogni "fetta" (potenza di 10), il risultato corrente è moltiplicato per 10, e il valore binario di ogni cifra della sequenza (nibble basso) gli viene aggiunto.

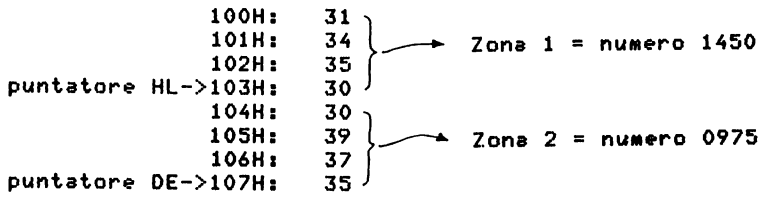
Nel nostro caso, la scomposizione sarà:

```
DE = 0
DE * 10 = 0 ----------------> + 6 --------> DE = 6
DE * 10 = 60 ---------------> + 4 --------> DE = 64
DE * 10 = 640 --------------> + 3 --------> DE = 643
DE * 10 = 6430 -------------> + 1 --------> DE = 6431
DE * 10 = 64310 ------------> + 4 --------> DE = 64314
```

Naturalmente, poichè tutte le operazioni si fanno in binario, il risultato 64314 è anch'esso espresso in binario.

```
                00100 ; ADDIZIONE DECIMALE INTERA
                00110 ; DI DUE ZONE ASCII
                00120 ; INGRESSO(HL):PUNTA LA FINE ZONA 1
                00130 ;          (DE):PUNTA LA FINE ZONA 2
                00140 ;           (B):LUNGHEZZA
                00150 ; RISULTATO NELLA ZONA 1
                00160 ;
0025            00170           ORG     25H
0025 4E         00180 ADEC      LD      C,(HL)          ;CAR ZONA 1
0026 1A         00190           LD      A,(DE)          ;CAR ZONA 2
0027 81         00200           ADD     A,C             ;CAR1+CAR2
0028 D630       00210           SUB     '0'             ;
002A FE3A       00220           CP      3AH             ;RIPORTO?
002C 3805       00230           JR      C,NRET          ;NO
002E D60A       00240           SUB     0AH             ;
0030 2B         00250           DEC     HL              ;RIPORTO
0031 34         00260           INC     (HL)            ;SUL CAR
0032 23         00270           INC     HL              ;SEGUENTE
0033 77         00280 NRET      LD      (HL),A          ;RISULTATO
0034 2B         00290           DEC     HL              ;PASSA AL
0035 1B         00300           DEC     DE              ;CAR SEG
0036 10ED       00310           DJNZ    ADEC            ;CONTINUA
0038 C9         00320           RET                     ;SE NO RET
0025            00330           END     ADEC
00000 TOTAL ERRORS
```

Questo programma esegue l'addizione decimale di due sequenze ASCII della stessa lunghezza. Supponiamo per esempio che queste due sequenze siano memorizzate a partire dall'indirizzo 100H:



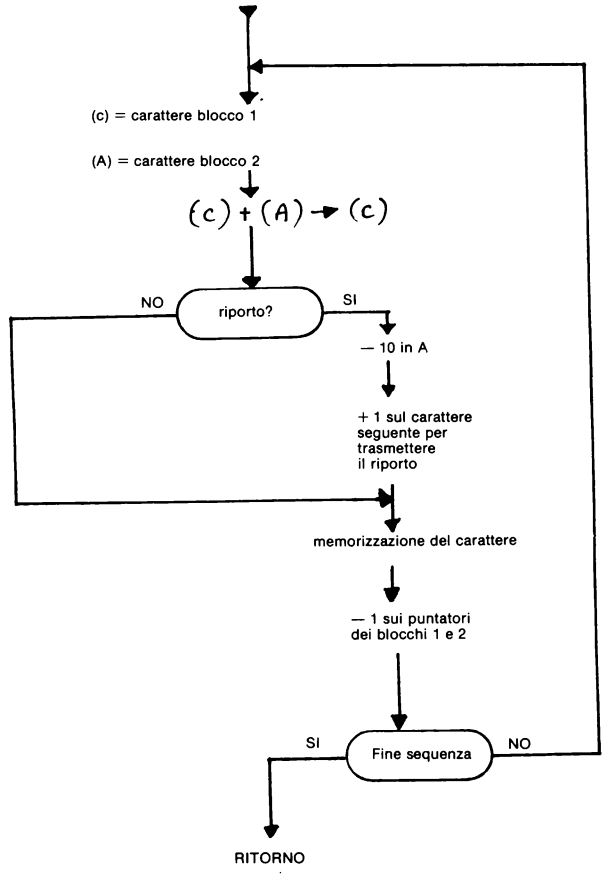
La chiamata:

```
LD      HL,103H
LD      DE,107H
LD      B,4
CALL    ADEC
```

restituirà i risultati:

```
100H:     32
101H:     34
102H:     32
103H:     35
```

corrispondenti al numero 2425 in ASCII.



Per seguire da vicino lo svolgimento di questo programma, vi proponiamo qui di seguito il listing che descrive passo passo l'esecuzione di ogni istruzione e il contenuto dinamico dei registri.

LOC	INST	MNEM	OPER	SP	RF	RA	RB	RC	RD	RE	RH	RL
0025	4E	LD	C,(HL)	0204	AB	32	04	30	01	07	01	03
0026	1A	LD	A,(DE)	0204	AB	35	04	30	01	07	01	03
0027	81	ADD	A,C	0204	20	65	04	30	01	07	01	03
0028	D630	SUB	A,30	0204	22	35	04	30	01	07	01	03
002A	FE3A	CP	3A	0204	BB	35	04	30	01	07	01	03
002C	3805	JR	C,07	0204	BB	35	04	30	01	07	01	03
0033	77	LD	(HL),A	0204	BB	35	04	30	01	07	01	03
0034	2B	DEC	HL	0204	BB	35	04	30	01	07	01	02
0035	1B	DEC	DE	0204	BB	35	04	30	01	06	01	02
0036	10ED	DJNZ	EF	0204	BB	35	03	30	01	06	01	02
0025	4E	LD	C,(HL)	0204	BB	35	03	35	01	06	01	02
0026	1A	LD	A,(DE)	0204	BB	37	03	35	01	06	01	02
0027	81	ADD	A,C	0204	28	6C	03	35	01	06	01	02
0028	D630	SUB	A,30	0204	2A	3C	03	35	01	06	01	02
002A	FE3A	CP	3A	0204	2A	3C	03	35	01	06	01	02
002C	3805	JR	C,07	0204	2A	3C	03	35	01	06	01	02
002E	D60A	SUB	A,0A	0204	22	32	03	35	01	06	01	02
0030	2B	DEC	HL	0204	22	32	03	35	01	06	01	01
0031	34	INC	(HL)	0204	20	32	03	35	01	06	01	01
0032	23	INC	HL	0204	20	32	03	35	01	06	01	02
0033	77	LD	(HL),A	0204	20	32	03	35	01	06	01	02
0034	2B	DEC	HL	0204	20	32	03	35	01	06	01	01

LOC	INST	MNEM	OPER	SP	RF	RA	RB	RC	RD	RE	RH	RL
0035	1B	DEC	DE	0204	20	32	03	35	01	05	01	01
0036	10ED	DJNZ	EF	0204	20	32	02	35	01	05	01	01
0025	4E	LD	C,(HL)	0204	20	32	02	35	01	05	01	01
0026	1A	LD	A,(DE)	0204	20	39	02	35	01	05	01	01
0027	81	ADD	A,C	0204	28	6E	02	35	01	05	01	01
0028	D630	SUB	A,30	0204	2A	3E	02	35	01	05	01	01
002A	FE3A	CP	3A	0204	2A	3E	02	35	01	05	01	01
002C	3805	JR	C,07	0204	2A	3E	02	35	01	05	01	01
002E	D60A	SUB	A,0A	0204	22	34	02	35	01	05	01	01
0030	2B	DEC	HL	0204	22	34	02	35	01	05	01	00
0031	34	INC	(HL)	0204	20	34	02	35	01	05	01	00
0032	23	INC	HL	0204	20	34	02	35	01	05	01	01
0033	77	LD	(HL),A	0204	20	34	02	35	01	05	01	01
0034	2B	DEC	HL	0204	20	34	02	35	01	05	01	00
0035	1B	DEC	DE	0204	20	34	02	35	01	04	01	00
0036	10ED	DJNZ	EF	0204	20	34	01	35	01	04	01	00
0025	4E	LD	C,(HL)	0204	20	34	01	32	01	04	01	00
0026	1A	LD	A,(DE)	0204	20	30	01	32	01	04	01	00
0027	81	ADD	A,C	0204	20	62	01	32	01	04	01	00
0028	D630	SUB	A,30	0204	22	32	01	32	01	04	01	00
002A	FE3A	CP	3A	0204	BB	32	01	32	01	04	01	00
002C	3805	JR	C,07	0204	BB	32	01	32	01	04	01	00

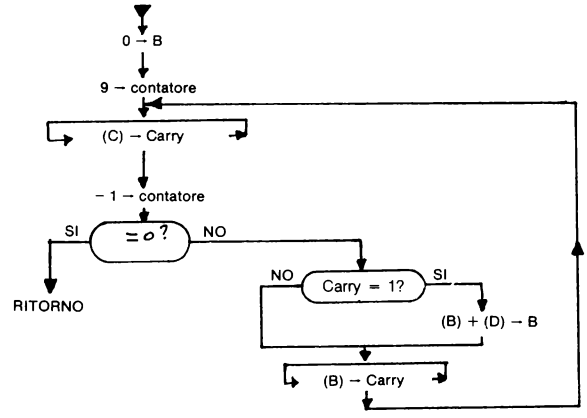
LOC	INST	MNEM	OPER	SP	RF	RA	RB	RC	RD	RE	RH	RL
0033	77	LD	(HL),A	0204	BB	32	01	32	01	04	01	00
0034	2B	DEC	HL	0204	BB	32	01	32	01	04	00	FF
0035	1B	DEC	DE	0204	BB	32	01	32	01	03	00	FF
0036	10ED	DJNZ	EF	0204	BB	32	00	32	01	03	00	FF
0038	C9	RET		0206	BB	32	00	32	01	03	00	FF

```
                00100 ;MOLTIPLICAZIONE BINARIA
                00110 ; (D) *(C)--->(BC)
                00120 ;
0000 0600       00130 MULT      LD      B,0             ;
0002 1E09       00140           LD      E,9             ;CONT BIT
0004 79         00150 MU0       LD      A,C             ;
0005 1F         00160           RRA                     ;
0006 4F         00170           LD      C,A             ;
0007 1D         00180           DEC     E               ;N BIT-1
0008 C8         00190           RET     Z               ;RET SE=0
0009 78         00200           LD      A,B             ;
000A 3001       00210           JR      NC,MU1          ;
000C 82         00220           ADD     A,D             ;
000D 1F         00230 MU1       RRA                     ;SHIFT
000E 47         00240           LD      B,A             ;
000F 18F3       00250           JR      MU0             ;
0000            00260           END     MULT
00000 TOTAL ERRORS
```

Come già sappiamo lo Z80 non possiede istruzioni di moltiplicazione. Questa routine consente di superare lo scoglio!

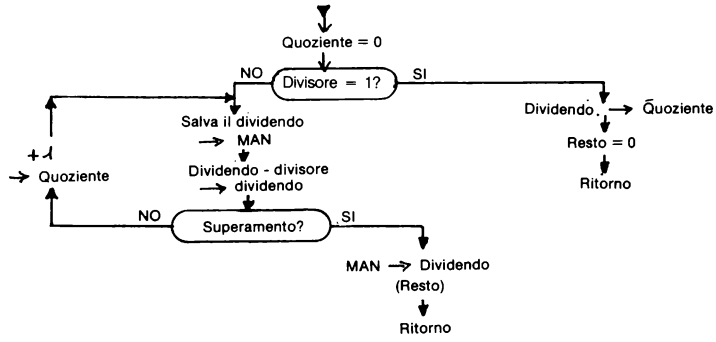
```
LD      D,2BH
LD      C,0A0H
CALL    MULT
```

Il registro BC conterrà il valore 1AE0H.
L'interesse di questo metodo è che la routine ha una durata praticamente costante, comunque siano i numeri da moltiplicare.
Il procedimento di calcolo è molto semplice, ed è rappresentato dal diagramma a blocchi seguente:



Ma passiamo ora alla divisione, che pure non compare nel set di istruzioni del microprocessore.

```
                00100 ;DIVISIONE BINARIA
                00110 ; (HL)/(DE)
                00120 ; IN USCITA :(BC)=QUOZIENTE
                00130 ;            (HL)=RESTO
                00140 ;
0000 010000     00150 DIV     LD      BC,0          ;RIS=0
0003 7A         00160         LD      A,D           ;BYTE ALTO
0004 FE00       00170         CP      0             ;QUOZ=0?
0006 200B       00180         JR      NZ,DIV1       ;NO
0008 7B         00190         LD      A,E           ;SI.BYTE
0009 FE01       00200         CP      1             ;BASSO=1?
000B 2006       00210         JR      NZ,DIV1       ;NO
000D 44         00220         LD      B,H           ;SI.HL->BC
000E 4D         00230         LD      C,L           ;
000F 210000     00240         LD      HL,0          ;RESTO=0
0012 C9         00250         RET                   ;RITORNO
0013 222500     00260 DIV1    LD      (MAN),HL      ;SALVA IL
0016 7D         00270         LD      A,L           ;RESTO
0017 93         00280         SUB     E             ;
0018 6F         00290         LD      L,A           ;
0019 7C         00300         LD      A,H           ;
001A 9A         00310         SBC     A,D           ;
001B 67         00320         LD      H,A           ;
001C 3803       00330         JR      C,DIV2        ;
001E 03         00340         INC     BC            ;QUOZ+1
001F 18F2       00350         JR      DIV1          ;
0021 2A2500     00360 DIV2    LD      (HL),MAN      ;RESTO->HL
0024 C9         00370         RET                   ;RITORNO
0025            00380 MAN     DEFS    2             ;MANOVRA
0000            00390         END     DIV
00000 TOTAL ERRORS
```

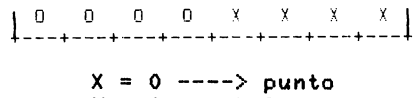


Il metodo usato qui è il più semplice, e procede per sottrazioni successive, come è facile vedere dal diagramma a blocchi seguente.
Il programma potrebbe essere semplificato usando l'istruzione SBC HL,DE, al posto delle linee 270 ÷ 320, ma bisognerebbe farla precedere da una OR A per non sottrarre il carry ...

Il programma seguente usa le tecniche di ricerca tabellare, di loop, e di chiamate di ingresso/uscita. Il principio è il seguente: si invia un carattere numerico dalla tastiera, e il programma fornisce nel registro A la codifica in codice Morse di questo carattere. L'ingresso di un carattere dalla tastiera viene effettuato tramite una chiamata alla routine TAST.

Nel caso che un carattere non compaia nella tabella, viene inviato un messaggio di errore al terminale video, tramite la routine VIDEO.
Poichè queste due routine sono specifiche delle periferiche, non verranno descritte in questa sede.

La codifica in Morse avrà la seguente struttura in un byte:



X = 0 ----> punto
X = 1 ----> linea

```
; TRANSCODIFICA DI UN CODICE NUMERICO
; IN CODICE MORSE
;
INIZ    CALL    TAST        ;ACQUISIZIONE DEL CARATTERE
                            ;DALLA TASTIERA.RISULTATO
                            ;--->A
        LD      B,A         ;SALVA CARATTERE IN B
        LD      HL,TABL     ;PUNTA INIZIO TABELLA
SCRUT   LD      A,(HL)      ;ESTRAI CAR NUMERICO
        OR      A           ;POSIZIONA FLAG
        JR      Z,FTAB      ;FINE TABELLA(ZERO)
        CP      B           ;E'UGUALE A B?
        INC     HL          ;SI PREPARA IL CARATTERE
        INC     HL          ;SEGUENTE
        JR      NZ,SCRUT    ;NO.SI CONTINUA
        DEC     HL          ;TROVATO,ESTRARRE IL
        LD      A,(HL)      ;CODICE MORSE
        RET                 ;RITORNO
;
FTAB    LD      HL,MESER    ;PUNTA MESSAGGIO ERRORE
        CALL    VIDEO       ;EMISSIONE VERSO VIDEO
        JR      INIZ        ;CONTINUA
;
TABL    DEFW    0F31H       ; 1 .----
        DEFW    0732H       ; 2 ..---
        DEFW    0333H       ; 3 ...--
        DEFW    0134H       ; 4 ....-
        DEFW    0035H       ; 5 .....
        DEFW    1036H       ; 6 -....
        DEFW    1837H       ; 7 --...
        DEFW    1C38H       ; 8 ---..
        DEFW    1E39H       ; 9 ----.
```

```
        DEFW    1F30H           ; O -----
        DEFB    0               ; FINE TABELLA
;
MESER   DEFM    'ERRORE.QUESTO CARATTERE NON E'' NELLA'
        DEFM    'TABELLA.RICOMINCIARE'
        DEFB    0               ; FINE DEL MESSAGGIO
```

Non bisognerà dimenticare che, in memoria, gli elementi della tabella saranno immagazzinati in quest'ordine:

```
TABL    --------> 1 ASCII, codice Morse di 1
TABL+2  --------> 2 ASCII, codice Morse di 2
TABL+4  --------> 3 ASCII, codice Morse di 3
   etc....
```

Con una simile organizzazione della tabella, si può effettuare qualsiasi tipo di transcodifica. Perchè non fare una conversione da una tastiera anglosassone (QWERTY) a una europea (AZERTY) ...

```
        CALL    TAST            ;INGRESSO TASTIERA QWERTY
        LD      B,A             ;SALVA IL CARATTERE
        LD      HL,QWERTY       ;INIZIO TABELLA QWERTY
        LD      DE,AZERTY       ;INIZIO TABELLA AZERTY
SCRUT   LD      A,(HL)          ;ESTRAI CARATTERE
        OR      A               ;POSIZIONA I FLAG
        JR      Z,FTAB          ;SE=0 FINE TAB QWERTY
        CP      B               ;TROVATO IL CARATTERE?
        INC     HL              ;INCREMENTA I PUNTATORI
        INC     DE              ;DELLE DUE TABELLE
        JR      NZ,SCRUT        ;SE NON TROVATO,CONTINUA
        DEC     DE              ;STOP.E'STATO TROVATO
        LD      A,(DE)          ;PRELEVA IL CARATTERE
                                ;NELLA TABELLA AZERTY
        RET                     ;RITORNO
;
QWERTY  DEFM    'QWERTY...'     ;TABELLA QWERTY
        DEFB    0               ;FINE TABELLA
AZERTY  DEFM    'AZERTY'        ;TABELLA AZERTY
```

Qui il principio è un po' diverso: si utilizzano due tabelle. Il puntatore della prima tabella servirà da puntatore della seconda.

E ora parleremo un po' della gestione ingresso/uscita, e più particolarmente di quello che si chiama un "*driver*".

Il driver è un programma indipendente, che ha il compito di pilotare una periferica, scaricando il programma utente di un certo numero di compiti di routine.

Se noi vogliamo, per esempio, inviare un messaggio ad una stampante, forniamo l'indirizzo di questo messaggio al driver corrispondente.
Di quello che succede poi non vogliamo sapere niente. L'importante è che il messaggio sia stampato ...!

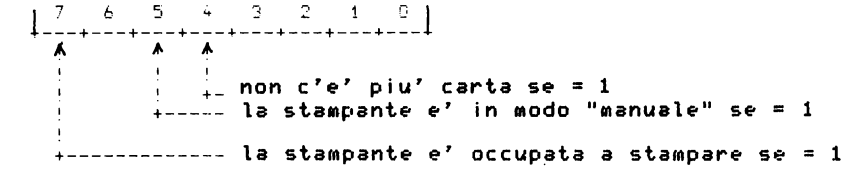
Ma chi si occuperà di accertare se la stampante è pronta a ricevere ogni carattere del messaggio? Se essa arriva alla fine di una riga, chi dovrà inviare un comando di posizionamento all'inizio della linea seguente, e alla fine di una pagina, un comando di posizionamento all'inizio della pagina seguente? È il driver della stampante.

Il programma utente, d'altra parte, definisce, in una zona speciale chiamata **blocco di comando**, i parametri necessari al driver, come per esempio il numero di linee per pagina, il numero di caratteri per linea, etc. ...

Per stampare un messaggio, basterà dunque fare:

```
LD      HL,MESSAG                ; indirizzo del messaggio
        CALL   PRINT             ; chiamata al driver
MESSAG  DEFM   ' GIU'  LE ZAMPE  .....
```

Cerchiamo di immaginare che cosa potrà essere questo driver.
La comunicazione con la stampante avverrà tramite un'istruzione OUT all'indirizzo 7, per esempio, per l'invio di un carattere, e l'istruzione IN riceverà lo stato della periferica sotto la forma:



Definiremo un blocco di comandi (CB) del driver della stampante con la struttura seguente:

```
CB + 0 ----->: numero di linee per pagina
CB + 1 ----->: numero di caratteri per linea
CB + 2 ----->: lunghezza del margine basso
CB + 3 ----->: contatore linee/pagina (gestito dal driver)
CB + 4 ----->: contatore caratteri/linea (gestito dal driver)
```

Prima di chiamare il driver, il programma utente dovrà evidentemente, riempire la zona da CB+0 a CB+2, per assegnare i parametri per la stampa.

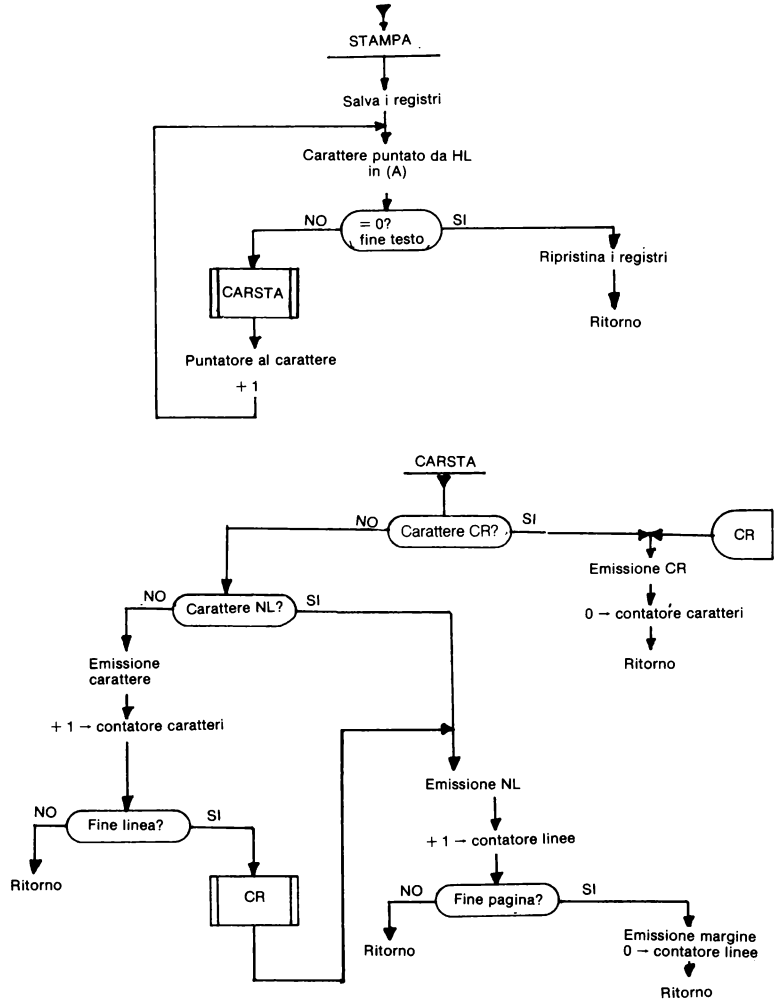
```
;ESEMPIO DI DRIVER PER STAMPANTE
;CHIAMATA: (HL)=INDIRIZZO DEL TESTO DA STAMPARE
;          CALL STAMPA
;
STAMPA  PUSH    AF          ;SALVA I REGISTRI
        PUSH    BC          ;UTILIZZATI DAL
        PUSH    HL          ;DRIVER
        PUSH    IX          ;
        LD      IX,CB       ;INDIRIZZO BLOCCO COMANDO
CONTIN  LD      A,(HL)      ;
        OR      A           ;FINE DEL TESTO?
        JR      NZ,STA      ;NO
        POP     IX          ;
        POP     HL          ;RIPRISTINA I REGISTRI
        POP     BC          ;
        POP     AF          ;
        RET                 ;RITORNO
STA     CALL    CARSTA      ;STAMPA DI UN CARATTERE
        INC     HL          ;CARATTERE SEGUENTE
        JR      CONTIN      ;
;
; STAMPA DI UN CARATTERE
;
CARSTA  CP      0DH         ;=RITORNO CARRELLO?
        JR      NZ,NOCR     ;NO
CR      LD      A,0DH       ;SI
        CALL    EMIS        ;EMISSIONE DI UN CR
        LD      (IX+4),0    ;CONTATORE CARATTERI=0
        RET                 ;
NOCR    CP      0AH         ;=NUOVA LINEA?
        JR      Z,LF        ;SI
        CALL    EMIS        ;SE NO,EMISSIONE DI UN CAR
        INC     (IX+4)      ;CONT CAR +1
        CP      (IX+1)      ;RAGGIUNTO IL MASSIMO?
        RET     NZ          ;NO
        CALL    CR          ;PASSARE A NUOVA LINEA
        LD      A,0AH       ;CODICE"NUOVA LINEA"
LF      CALL    EMIS        ;EMISSIONE LF
        INC     (IX+3)      ;CONT LINEE +1
        CP      (IX+0)      ;PAGINA PIENA?
        RET     NZ          ;NO
        LD      B,(IX+2)    ;VALORE DEL MARGINE
        LD      A,0AH       ;EMISSIONE DEL
PAGE    CALL    EMIS        ;CODICE NUOVA LINEA
        DJNZ    PAGE        ;FINO COMPLETAMENTO MARGINE
        LD      (IX+3),0    ;CONTATORE LINEE=0
        RET                 ;

EMIS    LD      C,A         ;SALVA IL CARATTERE
ATT     IN      A,(7)       ;LETTURA STATO STAMPANTE
        AND     0B0H        ;MASCHERA

        JR      NZ,ATT      ;NON DISPONIBILE
        LD      A,C         ;CARATTERE
        OUT     (7),A       ;EMISSIONE SU STAMPANTE
        RET                 ;RITORNO
```

```
;BLOCCO DI COMANDO
;
CB      DEFB    60          ;LINEE PER PAGINA
        DEFB    50          ;CARATTERI PER LINEA
        DEFB    6           ;MARGINE BASSO
        END     CARSTA
```

Questo programma usa abbondantemente la tecnica dei sottoprogrammi: è così facile! La lettura dello stato della stampante è abbastanza difficile ... e nel caso che non ci sia più carta, bisognerebbe forse inviare un messaggio ... sul video, naturalmente! A voi di migliorare il programma! Per finire, ecco un diagramma a blocchi del programma:



CAPITOLO 6

IL SOFTWARE DI SUPPORTO ALL'ASSEMBLER

Sarebbe ora, visto che ci avviciniamo alla fine di quest'opera, di riassumere le diverse operazioni che possono essere effettuate su di un programma scritto in assembler - e nell'ordine cronologico, perchè qualche volta abbiamo bruciato le tappe!

Prima di assemblare, dunque, bisogna creare il codice sorgente: questo è lo scopo del programma chiamato "**editor del testo**".
Dopodichè è possibile **assemblarlo**.

Nel caso in cui, il codice sorgente è suddiviso in più parti distinte, dobbiamo utilizzare l'"**editor di collegamento**" (LINK), che partendo dai codici oggetto usciti dall'assembler, produrrà un codice oggetto unico ed eseguibile.

Il programma può allora essere caricato in memoria per l'esecuzione. È il compito del "**caricatore**" (LOADER).

Per i pessimisti, presi da un angoscioso dubbio sul lavoro eseguito dai programmi citati, esiste un programma di utilità che si chiama "**disassemblatore**". Questo programma può anche essere utilizzato da quei curiosi che, possedendo soltanto il codice oggetto di un programma, vorrebbero sapere "che cos'ha nella pancia"!

Il disassemblatore, in effetti, esegue la funzione inversa dell'assemblatore. Se quest'ultimo fa la conversione: LD A,B in 78H, il disassemblatore, partendo da 78H lo converte in LD A,B.

Evidentemente questa conversione è parziale: mancano le label e i commenti! Ma cominciamo dall'inizio!

L'EDITOR DEL TESTO

Questo programma di utilità può talvolta essere compreso nell'assembler (**Editor/Assembler**), ma più spesso lo si trova sotto una forma autonoma, in modo da servire anche ad altri scopi (creazione di codici sorgente FORTRAN, BASIC e altri linguaggi compilativi).

Esso consente di ricevere un testo da un dispositivo di ingresso (in genere una tastiera) e di conservarlo per formare il codice sorgente. Le linee di ingresso sono, in genere, numerate automaticamente per facilitarne il reperimento.

Oltre alle funzioni di lettura/scrittura su di un supporto magnetico (o altro), offre un certo numero di funzioni di stampa, destinate a facilitare la correzione del programma sorgente.

Esistono due grandi tipi di editor: **l'editor di linea** e **l'editor di pagina** (o video). Il primo è usato soprattutto con periferiche di stampa su carta (per esempio telescriventi), che non permettono di cancellare o di correggere il testo introdotto.

Il secondo si usa con le periferiche a video, e permette la cancellazione o l'inserimento di un testo qualsiasi, in un indirizzo qualsiasi, di una pagina video, agendo semplicemente sul puntatore di posizione-cosa naturalmente impossibile con l'editor del primo tipo.

Poichè esiste una gran varietà di editor di testo, dai più semplici ai più complicati, ci accontenteremo di elencare le funzioni principali che possono realizzare:

— ingresso di un testo dalla periferica di ingresso
— lettura di un testo da un supporto magnetico
— scrittura di un testo su di un supporto magnetico
— inserimento di linee o caratteri
— sostituzione di linee o caratteri
— ricerca di caratteri
— sostituzione di testi
— posizionamento su un numero di linea
— posizionamento su un carattere
— cancellazione di caratteri o linee
— stampa di linee
— stampa di linee su stampante
— rinumerazione di linee
— macro-funzioni di stampa.

Ma se alcuni editor sono flessibili, logici e pratici, ce ne sono altri che esigono molto "allenamento".

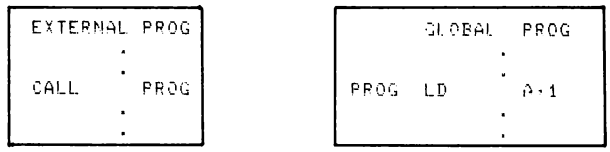
Come regola generale, è meglio evitare di fare errori (!) durante l'introduzione del codice sorgente, per evitare che i tentativi di correzione comportino, a loro volta, altri errori.

L'EDITOR DI COLLEGAMENTO

Abbiamo visto in precedenza che un programma sorgente, o per la sua lunghezza o per ragioni di praticità, poteva essere scomposto in più moduli, interconnessi tramite le direttive EXTERNAL e GLOBAL.

Questi moduli sono allora assemblati separatamente, ma non possono essere eseguiti, poichè certi riferimenti esterni non sono soddisfatti a livello di codice oggetto.

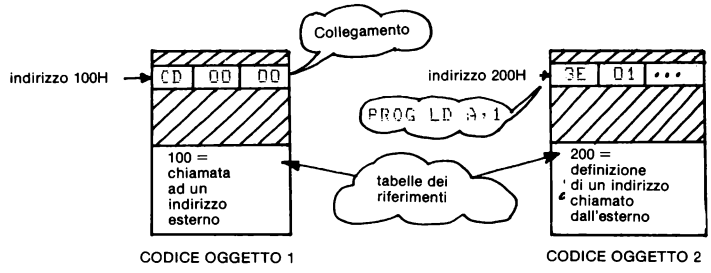
Per esempio:



MODULO 1 MODULO 2

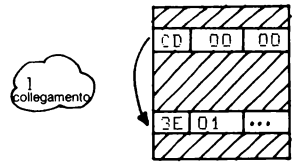
In questo esempio, il modulo 1 non potrà essere eseguito, poichè l'assemblatore non può conoscere il valore reale della label PROG, che è definita solo nel modulo 2. D'altra parte nessun errore è segnalato in fase di assemblaggio, poichè questa label è stata definita come esterna.

I codici oggetto prodotti avranno allora la struttura:



È in questo momento che interviene il grande Zorro ... scusate, l'editor di collegamento, che aggiusterà i riferimenti non definiti nei programmi oggetto.

L'unico modulo oggetto prodotto avrà la seguente struttura:



e questa volta sarà eseguibile.

Ma l'editor di collegamento ci offre un'altra interessante prestazione. Quando si assembla un programma, è possibile definire se il codice oggetto prodotto deve essere **assoluto** e **rilocabile**.

Nel primo caso il programma potrà essere eseguito solo agli indirizzi definiti tramite la direttiva ORG, mentre nel secondo caso, il codice oggetto prodotto non sarà in alcun caso eseguibile direttamente, ma dovrà essere sottoposto all'editor di collegamento, che gli assegnerà un indirizzo di origine, definito in quel momento dall'utente, e che può-in linea di massima-essere qualunque.

Naturalmente, il codice esadecimale che figura sul listing di assemblaggio, come anche gli indirizzi corrispondenti, non si dovranno più prendere in considerazione.

Nel caso in cui più moduli oggetto devono essere fusi insieme, è sufficiente o che uno di essi sia assoluto, o fissare un indirizzo di origine al momento del passaggio all'editor di collegamento.

Le direttive assembler che permettono di determinare se un programma deve essere assoluto o rilocabile sono:

ASEG (o equivalenti) per il codice assoluto
CSEG (o equivalenti) per il codice rilocabile

e se non ne abbiamo parlato finora, è perchè il loro utilizzo è specifico dell'editor di collegamento.

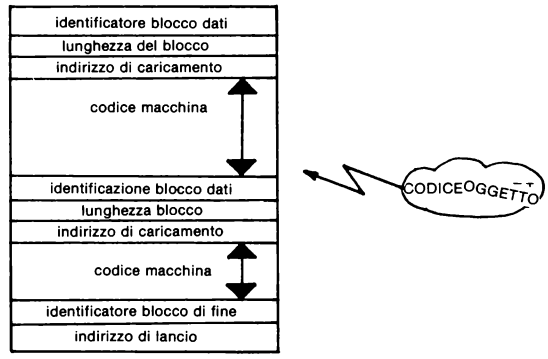
IL CARICATORE

Quando un programma è stato assemblato -e ha eventualmente subito un'elaborazione dall'editor di collegamento- è pronto per essere caricato in memoria per la sua esecuzione.

Questo caricamento è realizzato da un programma che si chiama **caricatore**, il quale sfrutta le informazioni contenute nel codice oggetto prodotto dall'assemblatore o dall'editor di collegamento, che gli permetteranno di collocare correttamente in memoria il codice eseguibile del programma.

In generale, il caricatore fa parte integrante di un sistema operativo che gestisce l'elaboratore, ed è attivato da comandi di tipo LOAD (caricamento) o RUN (caricamento + esecuzione).

Esamineremo la struttura generale del codice oggetto di un programma, per chiarire alcuni punti.



Questa struttura può variare da un sistema all'altro, ma il principio resta lo stesso.

Vediamo dunque che il codice oggetto è formato di blocchi consecutivi, che possono essere (almeno) di due tipi: blocco di dati e blocco di fine, che si differenziano per un byte di identificazione, per esempio 01 per il primo, 02 per il secondo.

Il blocco di dati contiene due informazioni, circa la lunghezza del blocco e il suo indirizzo di inserimento in memoria.

Il blocco di fine contiene l'indirizzo dal quale il programma dovrà essere lanciato al momento dell'esecuzione.

In generale, ogni volta che si incontra una direttiva ORG nel codice sorgente, l'assemblatore crea un nuovo blocco. Nel caso contrario, i blocchi hanno lunghezza standard di 128 o 256 byte.

IL DISASSEMBLATORE

È lo strumento fondamentale dei curiosi! È l'assemblatore che va a ritroso!

Come abbiamo già visto, questo programma di utilità consente di convertire il codice binario nel suo codice sorgente originario ...

Per prova, cerchiamo di disassemblare il binario del programma seguente:

```
        ORG     1000H
DEB     LD      A,1             ; registro A = 1
        RET                     ; ritorno
BYTE    DEFB    2               ; valore = 2
        DEFM    'TEXTE'         ; e' un testo
        .....
        LD      A,(BYTE)        ; (A) = byte
        .....
```

Otterremo:

```
1000H:  3E01        LD      A,1
1001H:  C9          RET
1003H:  02          LD      (BC),A
1004H:  54          LD      D,H
1005H:  45          LD      B,L
1006H:  58          LD      E,B
1007H:  54          LD      D,H
1008H:  45          LD      B,L
......              ..........
XXXXH:  3A0301      LD      A,(1003H)
......              ..........
```

C'è un fondo di verità, naturalmente, ma riconosciamolo, è piuttosto sconcertante!
C'è un trucco molto semplice da conoscere: quando il codice sembra un po' originale, e c'è una quantità di istruzioni LD ... provate con l'ASCII. Probabilmente è una zona di dati.

A parte questo, si trovano degli ottimi disassemblatori, che operano a partire da un codice binario caricato in memoria, o da un codice oggetto memorizzato su disco.

Alcuni di loro danno anche dei nomi alle label, del tipo: LAB1, LAB2, LAB3, ... e questo è sempre apprezzabile. Altri (magari gli stessi) costruiscono una tabella dei riferimenti incrociata (l'indirizzo in cui è definita una label e gli indirizzi in cui le si fa riferimento).

Alcuni producono un codice sorgente che può essere direttamente riassemblato ...! Il progresso è senza fine!

Il disassemblatore è il "passe-partout", la chiave che "apre" qualsiasi programma; nessuno gli resiste, che si tratti di programmi ritenuti inviolabili o induplicabili, o che si tratti di programmi che presentano anomalie, cosa che, ahimè, capita troppo spesso.

Quante volte abbiamo dovuto ricorrere a questo strumento per operare su dei programmi che sembravano delle "piccole meraviglie" e presentavano invece ignobili difetti.

Volete imparare a programmare in assembler? Imparate anche a disassemblare!

CAPITOLO 7

RELAZIONI CON I LINGUAGGI EVOLUTI

I programmi scritti in assembler sono, in generale, messi in opera in due modi diversi:

- il programma è autonomo (esempio: programma di utilità, compilatore, monitor etc ...)
- il programma dipende da un altro programma scritto in linguaggio evoluto (BASIC, FORTRAN, PASCAL ...) ed è spesso visto da quest'ultimo come una semplice routine, che realizza una funzione precisa.

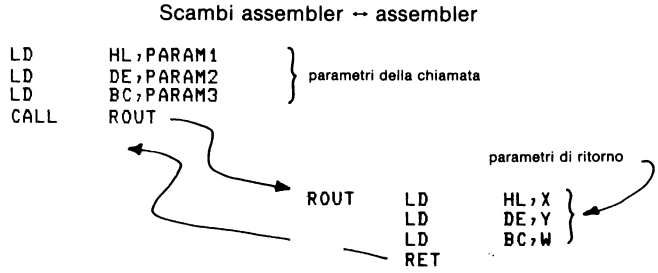
Un linguaggio, per evoluto che sia, non può realizzare tutte le funzioni, e in particolare quelle uscite fresche fresche dalla fantasia spesso delirante dei programmatori ...!

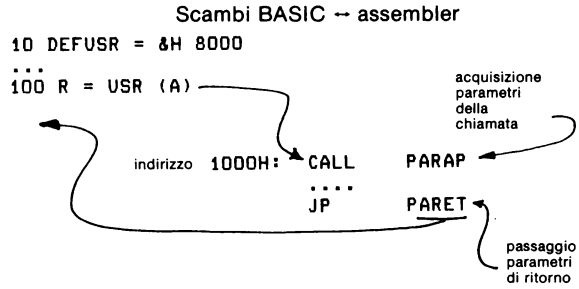
Talvolta, anche, una funzione è condizionata dal tempo, e il linguaggio evoluto deve adattarsi a fare appello al linguaggio macchina se vuole condurre a buon fine un certo compito.

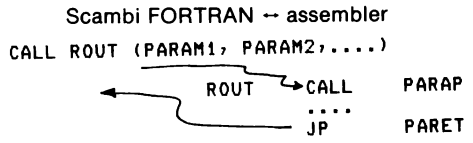
Ci sono anche dei metodi più terra-terra. Ad esempio, il driver della stampante del vostro BASIC non è perfettamente adatto al modello che voi possedete? Basta sostituire all'indirizzo del driver standard quello del vostro ... e i comandi LPRINT verranno trattati secondo le vostre necessità! Questa operazione si effettua con la funzione BASIC: POKE.

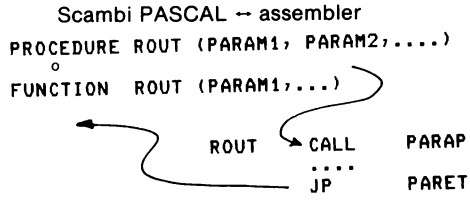
Vi ricordate dell'istruzione assembler CALL? Esiste una funzione simile in quasi tutti i linguaggi evoluti, che permette di richiamare una routine scritta in assembler, passandole uno o più parametri.
Dopo l'esecuzione di questa routine, si effettua un ritorno al linguaggio evoluto, accompagnato eventualmente da un passaggio di parametri.

Vediamo qualche esempio:









Le label PARAP e PARET rappresentano punti di ingresso di routine i cui indirizzi sono dati da chi ha costruito il compilatore o l'interprete, e che permettono il trasferimento dei parametri del linguaggio evoluto verso i registri del microprocessore e viceversa.

Talvolta questa operazione non è necessaria, e i parametri sono messi direttamente nei registri HL,DE e BC. Un'altra variante è data dal registro HL, che punta una zona di memoria contenente una lista di parametri. Queste modalità variano da un sistema all'altro, e sono presentate qui solo come esempio.

Poichè il FORTRAN e il PASCAL sono dei linguaggi compilativi (i compilatori sono delle specie di assemblatori che traducono il codice sorgente evoluto in codice macchina) sarà necessario usare le direttive per i riferimenti esterni tipiche di questi linguaggi, per creare dei collegamenti con la routine assembler. Dopo la fase di compilazione sarà quindi necessario passare attraverso l'editor di collegamento.

In BASIC, d'altra parte, (a meno di avere a che fare con un compilatore) il problema sarà diverso. Il codice oggetto della routine dovrà essere caricato separatamente in memoria, e unito al codice sorgente del programma BASIC interpretato.

Il comando BASIC DEFUSR permetterà di **DEF**inire l'indirizzo della routine dell'utente (**USeR**) (svolgendo qui la funzione dell'editor di collegamento), e la funzione USR provocherà la chiamata a questa routine. Il passaggio dei parametri si farà come è mostrato di seguito:

R = USR (A)

dove A rappresenta il parametro da passare alla routine, e R il parametro di ritorno.
Quando un parametro non è sufficiente, è possibile passare l'indirizzo di una tabella:

R = USR (VARPTR (T (0))

con T(0) = parametro 1; T(1) = parametro 2 ... e la routine avrà il compito di andare a cercare i parametri nella tabella.

Ma prendiamo subito un esempio molto semplice, che consiste nel moltiplicare un numero per 2, per mezzo di una routine assembler (programma realizzato sul TRS - 80)

```
10 DEFUSR = &H 8000
20 FOR AD = &H 8000 TO &H 8006
30 READ V : POKE AD,V:NEXT AD
40 INPUT "NUMERO" ;N
50 PRINT N ; "* 2 =" ; USR(N)
60 GOTO 40
70 DATA 205,127,10,41,195,154,10
```

Routine ricostituita:

```
8000H: CD 7F 0A     CALL  0A7FH    ;PARÁM--->HL
8003H: 29           ADD   HL,HL    ;HL*2
8004H: C3 9A 0A     JP    0A9AH    ;HL--->VARIABILE
```

In questo esempio, il codice macchina è, caricato dinamicamente in memoria dal programma BASIC, per mezzo del loop delle linee 20 e 30, che riempie la zona che va da 8000H a 8006H con i codici delle istruzioni della linea 70 espresse sotto forma decimale.

Le chiamate 0A7FH e 0A9AH definite da chi ha concepito il BASIC TRS-80 permettono di scambiare i parametri tra i due programmi BASIC e binario.

Naturalmente, nel caso che non ci siano parametri da passare, queste chiamate non sono necessarie, e basta che la routine termini con l'istruzione RET.

Partendo da un programma assemblato, il procedimento sarà:

— caricamento del codice oggetto della routine (LOAD)
— caricamento del programma BASIC con la struttura:

```
10  DEFUSR = XXXX        (indirizzo di lancio della routine)
    .....
100 A = USR (B)          (chiamata della routine)
```

Ecco! L'opera è terminata. Ha raggiunto il suo scopo: provocare lo scatto, la scintilla (a proposito, non lasciatelo mai in un ambiente umido e alla portata dei bambini)?

Come tutti i libri con una "vocazione" pedagogica non è un manuale dello Z80, poichè gli mancano un mucchio di cose.

Quando avrete letto questo libro, se la vostra curiosità si è aguzzata, e vi porta, con tutta naturalezza, a considerare qualche istruzione e a rifletterci sopra, è un buon risultato!

Continuate! Siete sulla buona strada! Procuratevi dei listing di programmi, disassemblate i codici oggetto e cercate di capire il loro funzionamento e di migliorarli. Questo vale un buon corso di perfezionamento. Allora il destino di quest'opera, come quello della scintilla, si sarà compiuto. Può darsi che le Appendici vi insegnino ancora qualche cosa.

APPENDICE 1

LA MATEMATICA DELL'INFORMATICA

L'uomo ha dieci dita, e fin dai tempi più antichi egli di solito conta utilizzando la **base 10**.

Quando l'uomo di Neanderthal, dall'alto della sua roccia, voleva dire a un suo collega, che si trovava non lontano da lì, che poteva contare 58 dinosauri nella pianura, doveva prima alzare 8 dita (unità), poi, dopo un istante, 5 dita (decine, numero di mani complete) per finire con un gesto significativo, che voleva dire "gnam ... gnam!".

Bisogna dire che avevano una vista acuta in quei tempi, ma ciononostante c'erano dei limiti, e quando la distanza era troppo grande bisognava usare un altro metodo: ad esempio, le due braccia.

Ma se, con le due mani, si poteva contare fino a 10 (base 10), con le due braccia si può contare solo fino a due (base 2). Inoltre, non bisogna dimenticare che il numero corrispondente alla base non viene mai utilizzato: 10 dinosauri sarebbero stati codificati, con le mani:

tempo 1: 0 dita alzate (unità)
tempo 2: 1 dito alzato (decine)
tempo 3: gram ... gram!

Analogamente 58 dinosauri sarebbero stati codificati come segue:

tempo 1 : 8 unità ----> 8
tempo 2 : 5 volte la base = 5 * 10 ---->50
tempo 3 : gnam ... gnam!

E se ce ne fossero 1253:

tempo 1 : 3 unità		=	3
tempo 2 : 5 volte la base	= 5 * 10	=	50
tempo 3 : 2 volte la base * la base	= 2 * 10 * 10	=	200
tempo 4 : 1 volta	= 1 * 10 * 10 * 10	=	1000
tempo 5 : gnam ... gram!			1253

Con le braccia (base 2), 58 dinosauri dovrebbero essere codificati:

Cioè:

tempo 1 : 0 unità		0
tempo 2 : 1 volta la base	= 1 * 2	2
tempo 3 : 0 volte la base per la base	= 0 * 2 * 2	0
tempo 4 : 1 volta ...	= 1 * 2 * 2 * 2	8
tempo 5 : 1 volta ...	= 1 * 2 * 2 * 2 * 2	16
tempo 6 : 1 volta ...	= 1 * 2 * 2 * 2 * 2 * 2	32
tempo 7 : gnam ... gnam!		58

Il vantaggio del metodo è evidente: lasciava all'uomo un braccio libero, cosa che gli permetteva di scambiarli quando c'erano molti dinosauri, perchè in questo caso la trasmissione era lunga e fastidiosa ...

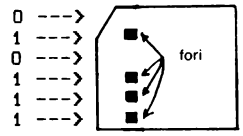
Così, come vedete, 58 in base 10 e 111010 in base 2 sono due rappresentazioni dello stesso numero (c'è un numero infinito di basi). Una è la rappresentazione decimale, l'altra la rappresentazione binaria.

Come il braccio dell'uomo di Neanderthal non poteva assumere che due stati (braccio alzato e braccio abbassato), le informazioni trattate dall'elaboratore saranno codificate con lo stesso principio:

— **Stato 0**: assenza di corrente, assenza di fori su una scheda o su un nastro perforato, assenza di magnetizzazione su una cassetta o su un disco.

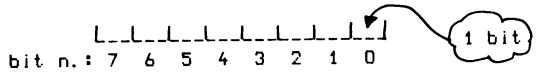
— **Stato 1**: circolazione di corrente, foro su una scheda su un nastro perforato, magnetizzazione su un supporto magnetico.

Il numero 58 potrà allora essere rappresentato in binario su una scheda perforata nel modo seguente:

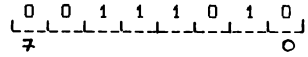


La memoria centrale è formata da un mosaico di cellule chiamate **bit**, ognuno dei quali può assumere uno dei due stati 0 o 1.

Di solito, questi bit sono raggruppati in gruppi i 8 per formare un **byte**, e la maggior parte dei microprocessori utilizzano l'indirizzamento a byte, cioè si accede simultaneamente a 8 bit della memoria, in lettura o in scrittura.
Il byte può essere rappresentato così:



e il numero 58 può essere codificato nel modo seguente:



con il bit 0 che rappresenta il bit meno significativo (o di peso minore) del numero, e il bit 7 che rappresenta il bit più significativo (o di peso maggiore) del numero, dove il concetto di peso dipende sostanzialmente dal valore della potenza del 2 associata ad ogni bit:

bit 0 ---> 2^0 =1
bit 1 ---> 2^1 =2
bit 2 ---> 2^2 =4
bit 3 ---> 2^3 =8
bit 4 ---> 2^4 =16
bit 5 ---> 2^5 =32
bit 6 ---> 2^6 =64
bit 7 ---> 2^7 =128

Più è elevata la potenza, più forte è il peso. Il nostro numero 58, nella sua rappresentazione binaria ha i bit 1, 3, 4, e 5 posizionati a 1. È dunque facile ritrovare il suo valore decimale:

bit 1 ---> 2
bit 3 ---> 8
bit 4 ---> 16
bit 5 ---> 32

58

Al contrario, un numero decimale dovrà essere diviso per le successive potenze del due (la base) per trovare la sua configurazione binaria.

Esempio: $133 = 2^7 + 2^2 + 2^0 = 128 + 4 + 1$

corripondente a:

```
      1  0  0  0  0  1  0  1
      └──┴──┴──┴──┴──┴──┴──┘
bits: 7              2     0
```

Ma questa rappresentazione è pesante e ingombrante. Si è quindi deciso di suddividere il byte in due **nibble** (4 bit):

```
      1  0  0  0  0  1  0  1
      └──┴──┴──┘  └──┴──┴──┘
```

e di far corrispondere ad ogni nibble 16 simboli diversi e unici:

0000	corrisponderà a	0
0001		1
0010		2
0011		3
0100		4
0101		5
0110		6
0111		7
1000		8
1001		9
1010	corrisponderà a:	.....?

I simboli numerici sono esauriti? Non importa. Usiamo gli alfabetici:

1010	corrisponderà a:	A
1011		B
1100		C
1101		D
1110		E
1111		F

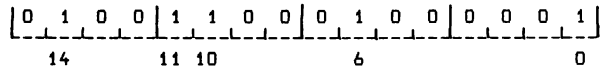
Ecco! Sono state scritte tutte le combinazioni possibili dei nibble, e bisogna fermarsi.

Con questa operazione, senza saperlo (ma è poi vero?), passiamo dalla base 2 (binario) alla base 16 (esadecimale), con il vantaggio di una scrittura leggermente più compatta (15 diventa F). Tutti questi sistemi di numerazione sono, ricordiamocelo, diversi modi di rappresentazione degli stessi numeri. Tutto sta nel conoscere la base utilizzata.

Un piccolo consiglio: imparate a memoria la tabella di corrispondenza binario-/esadecimale. Andiamo ... non è poi così difficile!

Il numero 58 (base 10) potrà essere rappresentato da 3A in base 16, che è senz'altro più gradevole di 00111010 ... comunque ciascuno ha i suoi gusti!

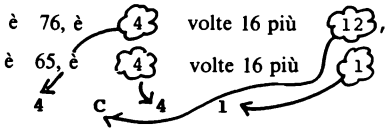
Naturalmente, il procedimento si ripete su più byte:



corrisponderà a 4C41 in base 16 (questo è facile), e a:

$2^{14} + 2^{11} + 2^{10} + 2^6 + 1$ in base 10, cioè 19521.

Ma 19521.....è 76 per 256 più 65,

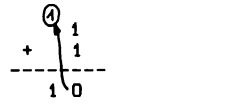


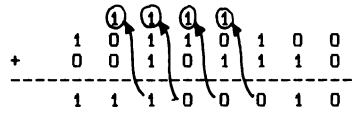
Siamo ripassati in base 16! Poi è molto più facile passare in base 2, se si conoscono i suoi fondamenti:

```
 4    C    4    1
 ↓    ↓    ↓    ↓
0100 1100 0100 0001
```

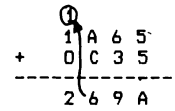
Poichè siamo in binario, cerchiamo di addizionare due numeri:
1 + 1 = 2 ... ?
No. In base 2 il 2 non esiste. Bisogna quindi aggiungere un riporto al risultato (come per 5 + 5 in base 10):

Proviamo ancora:



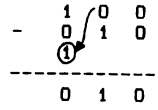


Il microprocessore procede nello stesso modo per effettuare una addizione. In esadecimale è un po' più complicato, a meno che si sappia contare sulle proprie dita ...

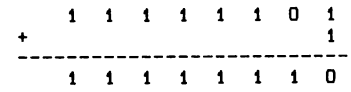


Si conta come in base 10, ma oltre il 9 si continua con A. C'è un riporto solo oltre la F.

E la sottrazione binaria?



Certo! Ma è più facile (almeno per il microprocessore) usare quello che si chiama il **complemento a 2**. Il complemento a 2 è il complemento a 1 più 1! Il complemento a 1 consiste nell'invertire il valore di ogni bit del numero, su una lunghezza uguale al formato in cui è rappresentato (di solito 8 o 16). Così, 010 deve essere rappresentato 00000010 se si lavora su numeri a 8 bit. Il suo complemento a 1 sarà: 11111101, e il suo complemento a 2:



Effettuiamo ora un'addizione di questo complemento con il valore 100 dell'esempio precedente:

e questo ci dà lo stesso risultato della sottrazione 0100-0010. Ecco come una sottrazione si trasforma in addizione.

Bisogna notare che l'ultimo riporto "cade nel vuoto", ma questo non è un problema, perchè il risultato è comunque su un formato di 8 bit.

Questo ci porta ora ad affrontare la nozione di **numero con segno**.

In informatica, ci sono due modi di considerare un numero: in valore assoluto (tutti i bit sono significativi) e in valore con segno. In quest'ultimo caso, il bit più significativo del numero indicherà se è negativo (= 1 in complemento a 2) o positivo (= 0).

Per esempio, il numero binario:

```
1 1 1 1 1 1 1 0
```

(FE in base 16), sarà uguale a 254 in valore decimale assoluto, e a —2 in valore decimale con segno, poichè rappresenta, in quest'ultimo caso, il complemento a 2 del numero 2 (notate che l'addizione di un numero e il suo complemento a 2 dà sempre 0). È facile: basta spiegarsi subito sul valore della base e sul tipo di rappresentazione ...

Un altro tipo di rappresentazione è talvolta utilizzata in informatica: la rappresentazione **BCD**, che significa **Binary Coded Decimal**.

Viene usata in genere nei programmi che eseguono operazioni direttamente in decimale, senza passare dal binario.

Le cifre decimali assumono i valori da 0 a 9, e si fa corrispondere ad ogni cifra del numero un nibble codificato nel modo seguente:

```
0000 ---> 0
0001 ---> 1
0010 ---> 2
  ......
1001 ---> 9
```

La codifica è la stessa che per il codice esadecimale, ma questa volta si ferma a 9. Così, il numero 1794 sarà rappresentato:

```
  1       7       9       4
  ↓       ↓       ↓       ↓
0001    0111    1001    0100
```

Per finire, diremo qualche parola a proposito degli **operatori logici**.
Esistono quattro funzioni logiche utilizzate spesso sui numeri binari: il **NOT**, **l'AND**, **l'OR**, e **l'OR ESCLUSIVO.**

Il NOT l'abbiamo già visto: è il complemento a 1. **L'AND** (o intersezione logica) di due numeri binari si effettua secondo le regole seguenti:

```
0 AND 0 ---> 0
0 AND 1 ---> 0
1 AND 0 ---> 0
1 AND 1 ---> 1
```

applicate ad ogni copia di bit dei due numeri.

Esempio:

```
        1  0  1  0  1  1  0  1
AND     0  1  1  1  1  0  1  1
      ------------------------
        0  0  1  0  1  0  0  1
```

In forma esadecimale scriveremo: AD AND 7B = 29. Cosa c'è di più facile?

Il bit risultante sarà a 1 se il bit del primo numero e il bit del secondo numero sono entrambi a 1.

L'**OR** (o unione logica) obbedisce alle seguenti regole:

```
0 OR 0 ---> 0
0 OR 1 ---> 1
1 OR 0 ---> 1
1 OR 1 ---> 1
```

Esempio:

```
        1  0  1  0  1  1  0  1
OR      1  1  0  0  0  1  0  1
      ------------------------
        1  1  1  0  1  1  0  1
```

che si può anche scrivere: AD OR C5 = ED in esadecimale. Il bit risultante sarà a 1 se il bit del primo numero o il bit del secondo numero saranno a 1.

L'OR ESCLUSIVO (o disgiunzione logica) obbedisce alle stesse regole del precedente, tranne che per l'ultima condizione:

```
0 EX OR 0 ---> 0
0 EX OR 1 ---> 1
1 EX OR 0 ---> 1
1 EX OR 1 ---> 0
```

Perchè il bit risultante sia a 1, bisogna dunque che il bit del primo numero o il bit del secondo numero siano a 1, ma non tutti e due contemporaneamente.

Esempio:

```
        0  1  0  1  1  1  0  1
EX OR   0  1  1  0  1  0  0  0
      ------------------------
        0  0  1  1  0  1  0  1
```

APPENDICE 2

CORREZIONE DEGLI ESERCIZI

CAPITOLO 2

ESERCIZIO 2.1

```
SOM     INC     B               ;+1 IN B
        DEC     C               ;-1 IN C
        JR      NZ,SOM          ;FINO A QUANDO C=0
```

ESERCIZIO 2.2

Al momento della chiamata alla routine tramite la CALL ROUT, l'indirizzo dell'istruzione di stop JR $ è inserito nello stack, e si effettua un salto all'indirizzo ROUT. Qui viene estratto dallo stack l'indirizzo di ritorno (POP HL), incrementato di 2 e reinserito nello stack (PUSH). Al momento della RET (ritorno), il programma proseguirà quindi due byte più in là di quello che era previsto, cioè all'indirizzo ROUT. La JR $ non viene eseguita. La POP HL "recupera" 0ABCH che è incrementato di 2, e la RET (ritorno) provocherà una prosecuzione del programma all'indirizzo 0ABEH.

ESERCIZIO 2.3

Questo programma esegue semplicemente una modifica di se stesso. Il codice 3DH corrispondente all'istruzione DEC A (vedere listing in Appendice) viene scritto al posto dell'istruzione INC A situata all'indirizzo ADR. Il registro A sarà quindi decrementato e poi incrementato, e ritroverà il suo valore iniziale 3DH!

ESERCIZIO 2.4

Non mancano le soluzioni, e voi ne avrete certamente trovata una!
Ecco un programma che sarebbe affascinante....se funzionasse perfettamente: utilizza 2 istruzioni...cosa dire di meglio? Rimette a zero la totalità della memoria, tranne 4 byte, e da quel momento comincia ad auto-distruggersi! Dopo di chè c'è proprio da divertirsi.....
Per quanto semplice, il suo meccanismo è piuttosto complesso. Cercate comunque di capirlo fino in fondo, e in particolare a partire dal momento in cui la CALL ADR diventa una CALL 00....
Per funzionare (o quasi), questo programma deve essere caricato all'indirizzo 0FFFAH, in modo che l'indirizzo di ritorno dalla CALL sia zero!

```
FFFA  31FCFF              LD      SP,ADR-1
FFFD  CDFDFF      ADR     CALL    ADR
0000              ;FINE DELLA MEMORIA.SI RIPARTE DA 0
```

Divertente, no?

ESERCIZIO 2.5

```
          LD      C,5           ;CONTATORE
          LD      HL,BUFF       ;PUNTA INIZIO BUFFER
          LD      DE,BUFF+9     ;PUNTA FINE BUFFER
SCAMB     LD      A,(DE)        ;CARATTERI DA
          LD      B,(HL)        ;SCAMBIARE
          LD      (HL),A        ;(DE)--->(HL)
          LD      A,B           ;
          LD      (DE),A        ;(HL)--->(DE)
          INC     HL            ;INIZIO +1
          DEC     DE            ;FINE -1
          DEC     C             ;5 VOLTE
          JR      NZ,SCAMB      ;CONTINUA SE NON 0
                                ;FINE
```

CAPITOLO 3

ESERCIZIO 3.1

```
;CONVERSIONE HEXA ASCII---->BINARIO
;
HEXBIN    SUB     30H           ;
          CP      10            ;E' UNA CIFRA?
          RET     C             ;SI.RITORNO
          SUB     7             ;E' UNA LETTERA(DA A A F)
          RET                   ;
```

ESERCIZIO 3.2

```
LD      A,B             ;
CALL    HEXBIN          ;CONVERSIONE BYTE ALTO
RLCA                    ;SHIFTARE IL NIBBLE ALTO
RLCA                    ;DI A
RLCA                    ;
RLCA                    ;
LD      B,A             ;A--->B
LD      A,C             ;BYTE         BASSO
CALL    HEXBIN          ;CONVERSIONE
OR      B               ;RIUNIONE DEI DUE BYTE
                        ;FINE
```

ESERCIZIO 3.3

```
; IN USCITA:Z=1 SE DE=HL
;           C=1 SE DE>HL
;
        LD      A,H
        SUB     D
        RET     NZ
        LD      A,L
        SUB     E
        RET
```

ESERCIZIO 3.4

```
; IN USCITA ,Z=1 SE C'E' UGUAGLIANZA
;
COMPAR  LD      A,(DE)
        CP      (HL)
        RET     NZ
        DEC     C
        RET     Z
        INC     HL
        INC     DE
        JR      COMPAR
```

Ma si può anche utilizzare l'istruzione CPI:

```
COMPAR  LD      B,0
        LD      A,(DE)
        CPI
        RET     NZ
        RET     PO
        INC     DE
        JR      COMPAR
```

ESERCIZIO 3.5

```
MUL10   LD      D,H
        LD      E,L             ;HL--->DE
        ADD     HL,HL           ;HL*2
        ADD     HL,HL           ;HL*4
        ADD     HL,DE           ;HL*5
        ADD     HL,HL           ;HL*10
```

APPENDICE 3

IL CODICE ASCII

BYTE									
BASSI \ ALTI	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	0000	NUL	DLE	SP	0	@	P	`	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENG	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	-	=	M	]	m	}
E	1110	SO	RS	.	>	N	↑	n	~
F	1111	SI	VS	/	?	O	←	o	DEL

Esempio di utilizzo:

A = 41H
? = 3FH

Significato delle funzioni più usate:

BEL : campana, avvertimento sonoro
BS : backspace, arretramento del cursore
HT : tabulazione orizzontale
LF : line feed, ritorno linea
VT : tabulazione verticale
CR : carriage return, ritorno carrello (ritorno linea)
SP : spazio

APPENDICE 4

IL SET DI ISTRUZIONI DELLO Z80

Per la storia, ecco come è stata creata questa lista.

a. Un programma BASIC, partendo dalla radice di ogni codice mnemonico, produce la famiglia corrispondente.

Esempio: LD <SR>, <SR>

(SR rappresenta i registri semplici), creerà tutti i codici:
LD A,A LD A,B ....LD L,L.

Lo schedario così costruito è memorizzato su disco.

b. Scelta dallo schedario per ordine crescente.
c. Numerazione delle istruzioni (programma BASIC).
d. Assemblaggio dello schedario per generare il codice macchina (listing su disco).
e. Soppressione dei numeri di linea (divenuti inutili) da parte del programma BASIC.
f. Modifica del codice macchina (dd, nn, etc....) da un software di trattamente del testo.
g. Disposizione del testo su due colonne di 60 linee per pagina (programma BASIC).
h. Stampa della lista (vedere poi).

I simboli utilizzati hanno il seguente significato:

dd : valore di spiazzamento con segno su 8 bit,
N : valore di 8 bit dell'operando,
nn : valore di 8 bit nel codice macchina,
ADR : label di indirizzo nell'operando,
aaaa : valore di indirizzo su 16 bit nel codice macchina (byte alto + byte basso),
DEPL : spiazzamento nell'operando dell'istruzioni in modo relativo,
NN : valore di 16 bit o indirizzo nell'operando,
nnnn : valore di 16 bit nel codice macchina (byte alto + byte basso).

```
0000 8E         ADC A,(HL)      *  0066 DDCBdd4E   BIT 1,(IX+dd)
0001 DD8Edd     ADC A,(IX+dd)   *  006A FDCBdd4E   BIT 1,(IY+dd)
0004 FD8Edd     ADC A,(IY+dd)   *  006E CB4F       BIT 1,A
0007 8F         ADC A,A         *  0070 CB48       BIT 1,B
0008 88         ADC A,B         *  0072 CB49       BIT 1,C
0009 89         ADC A,C         *  0074 CB4A       BIT 1,D
000A 8A         ADC A,D         *  0076 CB4B       BIT 1,E
000B 8B         ADC A,E         *  0078 CB4C       BIT 1,H
000C 8C         ADC A,H         *  007A CB4D       BIT 1,L
000D 8D         ADC A,L         *  007C CB56       BIT 2,(HL)
000E CEnn       ADC A,N         *  007E DDCBdd56   BIT 2,(IX+dd)
0010 ED4A       ADC HL,BC       *  0082 FDCBdd56   BIT 2,(IY+dd)
0012 ED5A       ADC HL,DE       *  0086 CB57       BIT 2,A
0014 ED6A       ADC HL,HL       *  0088 CB50       BIT 2,B
0016 ED7A       ADC HL,SP       *  008A CB51       BIT 2,C
0018 86         ADD A,(HL)      *  008C CB52       BIT 2,D
0019 DD86dd     ADD A,(IX+dd)   *  008E CB53       BIT 2,E
001C FD86dd     ADD A,(IY+dd)   *  0090 CB54       BIT 2,H
001F 87         ADD A,A         *  0092 CB55       BIT 2,L
0020 80         ADD A,B         *  0094 CB5E       BIT 3,(HL)
0021 81         ADD A,C         *  0096 DDCBdd5E   BIT 3,(IX+dd)
0022 82         ADD A,D         *  009A FDCBdd5E   BIT 3,(IY+dd)
0023 83         ADD A,E         *  009E CB5F       BIT 3,A
0024 84         ADD A,H         *  00A0 CB58       BIT 3,B
0025 85         ADD A,L         *  00A2 CB59       BIT 3,C
0026 C6nn       ADD A,N         *  00A4 CB5A       BIT 3,D
0028 09         ADD HL,BC       *  00A6 CB5B       BIT 3,E
0029 19         ADD HL,DE       *  00A8 CB5C       BIT 3,H
002A 29         ADD HL,HL       *  00AA CB5D       BIT 3,L
002B 39         ADD HL,SP       *  00AC CB66       BIT 4,(HL)
002C DD09       ADD IX,BC       *  00AE DDCBdd66   BIT 4,(IX+dd)
002E DD19       ADD IX,DE       *  00B2 FDCBdd66   BIT 4,(IY+dd)
0030 DD29       ADD IX,IX       *  00B6 CB67       BIT 4,A
0032 DD39       ADD IX,SP       *  00B8 CB60       BIT 4,B
0034 FD09       ADD IY,BC       *  00BA CB61       BIT 4,C
0036 FD19       ADD IY,DE       *  00BC CB62       BIT 4,D
0038 FD29       ADD IY,IY       *  00BE CB63       BIT 4,E
003A FD39       ADD IY,SP       *  00C0 CB64       BIT 4,H
003C A6         AND (HL)        *  00C2 CB65       BIT 4,L
003D DDA6dd     AND (IX+dd)     *  00C4 CB6E       BIT 5,(HL)
0040 FDA6dd     AND (IY+dd)     *  00C6 DDCBdd6E   BIT 5,(IX+dd)
0043 A7         AND A           *  00CA FDCBdd6E   BIT 5,(IY+dd)
0044 A0         AND B           *  00CE CB6F       BIT 5,A
0045 A1         AND C           *  00D0 CB68       BIT 5,B
0046 A2         AND D           *  00D2 CB69       BIT 5,C
0047 A3         AND E           *  00D4 CB6A       BIT 5,D
0048 A4         AND H           *  00D6 CB6B       BIT 5,E
0049 A5         AND L           *  00D8 CB6C       BIT 5,H
004A E6nn       AND N           *  00DA CB6D       BIT 5,L
004C CB46       BIT 0,(HL)      *  00DC CB76       BIT 6,(HL)
004E DDCBdd46   BIT 0,(IX+dd)   *  00DE DDCBdd76   BIT 6,(IX+dd)
0052 FDCBdd46   BIT 0,(IY+dd)   *  00E2 FDCBdd76   BIT 6,(IY+dd)
0056 CB47       BIT 0,A         *  00E6 CB77       BIT 6,A
0058 CB40       BIT 0,B         *  00E8 CB70       BIT 6,B
005A CB41       BIT 0,C         *  00EA CB71       BIT 6,C
005C CB42       BIT 0,D         *  00EC CB72       BIT 6,D
005E CB43       BIT 0,E         *  00EE CB73       BIT 6,E
0060 CB44       BIT 0,H         *  00F0 CB74       BIT 6,H
0062 CB45       BIT 0,L         *  00F2 CB75       BIT 6,L
0064 CB4E       BIT 1,(HL)      *  00F4 CB7E       BIT 7,(HL)
00F6 DDCBdd7E   BIT 7,(IX+dd)   *  0163 D9         EXX
00FA FDCBdd7E   BIT 7,(IY+dd)   *  0164 76         HALT
00FE CB7F       BIT 7,A         *  0165 ED46       IM 0
0100 CB78       BIT 7,B         *  0167 ED56       IM 1
0102 CB79       BIT 7,C         *  0169 ED5E       IM 2
0104 CB7A       BIT 7,D         *  016B ED78       IN A,(C)
0106 CB7B       BIT 7,E         *  016D DBnn       IN A,(N)
```

```
0108 CB7C      BIT 7,H          *  016F ED40      IN B,(C)
010A CB7D      BIT 7,L          *  0171 ED48      IN C,(C)
010C DCaaaa    CALL C,ADR       *  0173 ED50      IN D,(C)
010F FCaaaa    CALL M,ADR       *  0175 ED58      IN E,(C)
0112 D4aaaa    CALL NC,ADR      *  0177 ED60      IN H,(C)
0115 CDaaaa    CALL ADR         *  0179 ED68      IN L,(C)
0118 C4aaaa    CALL NZ,ADR      *  017B 34        INC (HL)
011B F4aaaa    CALL P,ADR       *  017C DD34dd    INC (IX+dd)
011E ECaaaa    CALL PE,ADR      *  017F FD34dd    INC (IY+dd)
0121 E4aaaa    CALL PO,ADR      *  0182 3C        INC A
0124 CCaaaa    CALL Z,ADR       *  0183 04        INC B
0127 3F        CCF              *  0184 03        INC BC
0128 BE        CP (HL)          *  0185 0C        INC C
0129 DDBEdd    CP (IX+dd)       *  0186 14        INC D
012C FDBEdd    CP (IY+dd)       *  0187 13        INC DE
012F BF        CP A             *  0188 1C        INC E
0130 B8        CP B             *  0189 24        INC H
0131 B9        CP C             *  018A 23        INC HL
0132 BA        CP D             *  018B DD23      INC IX
0133 BB        CP E             *  018D FD23      INC IY
0134 BC        CP H             *  018F 2C        INC L
0135 BD        CP L             *  0190 33        INC SP
0136 FEnn      CP N             *  0191 EDAA      IND
0138 EDA9      CPD              *  0193 EDBA      INDR
013A EDB9      CPDR             *  0195 EDA2      INI
013C EDA1      CPI              *  0197 EDB2      INIR
013E EDB1      CPIR             *  0199 E9        JP (HL)
0140 2F        CPL              *  019A DDE9      JP (IX)
0141 27        DAA              *  019C FDE9      JP (IY)
0142 35        DEC (HL)         *  019E DAaaaa    JP C,ADR
0143 DD35dd    DEC (IX+dd)      *  01A1 FAaaaa    JP M,ADR
0146 FD35dd    DEC (IY+dd)      *  01A4 D2aaaa    JP NC,ADR
0149 3D        DEC A            *  01A7 C3aaaa    JP ADR
014A 05        DEC B            *  01AA C2aaaa    JP NZ,ADR
014B 0B        DEC BC           *  01AD F2aaaa    JP P,ADR
014C 0D        DEC C            *  01B0 EAaaaa    JP PE,ADR
014D 15        DEC D            *  01B3 E2aaaa    JP PO,ADR
014E 1B        DEC DE           *  01B6 CAaaaa    JP Z,ADR
014F 1D        DEC E            *  01B9 38dd      JR C,DEPL
0150 25        DEC H            *  01BB 18dd      JR DEPL
0151 2B        DEC HL           *  01BD 30dd      JR NC,DEPL
0152 DD2B      DEC IX           *  01BF 20dd      JR NZ,DEPL
0154 FD2B      DEC IY           *  01C1 28dd      JR Z,DEPL
0156 2D        DEC L            *  01C3 02        LD (BC),A
0157 3B        DEC SP           *  01C4 12        LD (DE),A
0158 F3        DI               *  01C5 77        LD (HL),A
0159 10dd      DJNZ DEPL        *  01C6 70        LD (HL),B
015B FB        EI               *  01C7 71        LD (HL),C
015C E3        EX (SP),HL       *  01C8 72        LD (HL),D
015D DDE3      EX (SP),IX       *  01C9 73        LD (HL),E
015F FDE3      EX (SP),IY       *  01CA 74        LD (HL),H
0161 08        EX AF,AF'        *  01CB 75        LD (HL),L
0162 EB        EX DE,HL         *  01CC 36nn      LD (HL),N
01CE DD77dd    LD (IX+dd),A     *  0256 4C        LD C,H
01D1 DD70dd    LD (IX+dd),B     *  0257 4D        LD C,L
01D4 DD71dd    LD (IX+dd),C     *  0258 0Enn      LD C,N
01D7 DD72dd    LD (IX+dd),D     *  025A 56        LD D,(HL)
01DA DD73dd    LD (IX+dd),E     *  025B DD56dd    LD D,(IX+dd)
01DD DD74dd    LD (IX+dd),H     *  025E FD56dd    LD D,(IY+dd)
01E0 DD75dd    LD (IX+dd),L     *  0261 57        LD D,A
01E3 DD36ddnn  LD (IX+dd),N     *  0262 50        LD D,B
01E7 FD77dd    LD (IY+dd),A     *  0263 51        LD D,C
01EA FD70dd    LD (IY+dd),B     *  0264 52        LD D,D
01ED FD71dd    LD (IY+dd),C     *  0265 53        LD D,E
01F0 FD72dd    LD (IY+dd),D     *  0266 54        LD D,H
01F3 FD73dd    LD (IY+dd),E     *  0267 55        LD D,L
```

```
01F6 FD74dd     LD (IY+dd),H   *  0268 16nn       LD D,N
01F9 FD75dd     LD (IY+dd),L   *  026A ED5Bnnnn   LD DE,(NN)
01FC FD36ddnn   LD (IY+dd),N   *  026E 11nnnn     LD DE,NN
0200 32nnnn     LD (NN),A      *  0271 5E         LD E,(HL)
0203 ED43nnnn   LD (NN),BC     *  0272 DD5Edd     LD E,(IX+dd)
0207 ED53nnnn   LD (NN),DE     *  0275 FD5Edd     LD E,(IY+dd)
020B 22nnnn     LD (NN),HL     *  0278 5F         LD E,A
020E DD22nnnn   LD (NN),IX     *  0279 58         LD E,B
0212 FD22nnnn   LD (NN),IY     *  027A 59         LD E,C
0216 ED73nnnn   LD (NN),SP     *  027B 5A         LD E,D
021A 0A         LD A,(BC)      *  027C 5B         LD E,E
021B 1A         LD A,(DE)      *  027D 5C         LD E,H
021C 7E         LD A,(HL)      *  027E 5D         LD E,L
021D DD7Edd     LD A,(IX+dd)   *  027F 1Enn       LD E,N
0220 FD7Edd     LD A,(IY+dd)   *  0281 66         LD H,(HL)
0223 3Annnn     LD A,(NN)      *  0282 DD66dd     LD H,(IX+dd)
0226 7F         LD A,A         *  0285 FD66dd     LD H,(IY+dd)
0227 78         LD A,B         *  0288 67         LD H,A
0228 79         LD A,C         *  0289 60         LD H,B
0229 7A         LD A,D         *  028A 61         LD H,C
022A 7B         LD A,E         *  028B 62         LD H,D
022B 7C         LD A,H         *  028C 63         LD H,E
022C ED57       LD A,I         *  028D 64         LD H,H
022E 7D         LD A,L         *  028E 65         LD H,L
022F 3Enn       LD A,N         *  028F 26nn       LD H,N
0231 ED5F       LD A,R         *  0291 2Annnn     LD HL,(NN)
0233 46         LD B,(HL)      *  0294 21nnnn     LD HL,NN
0234 DD46dd     LD B,(IX+dd)   *  0297 ED47       LD I,A
0237 FD46dd     LD B,(IY+dd)   *  0299 DD2Annnn   LD IX,(NN)
023A 47         LD B,A         *  029D DD21nnnn   LD IX,NN
023B 40         LD B,B         *  02A1 FD2Annnn   LD IY,(NN)
023C 41         LD B,C         *  02A5 FD21nnnn   LD IY,NN
023D 42         LD B,D         *  02A9 6E         LD L,(HL)
023E 43         LD B,E         *  02AA DD6Edd     LD L,(IX+dd)
023F 44         LD B,H         *  02AD FD6Edd     LD L,(IY+dd)
0240 45         LD B,L         *  02B0 6F         LD L,A
0241 06nn       LD B,N         *  02B1 68         LD L,B
0243 ED4Bnnnn   LD BC,(NN)     *  02B2 69         LD L,C
0247 01nnnn     LD BC,NN       *  02B3 6A         LD L,D
024A 4E         LD C,(HL)      *  02B4 6B         LD L,E
024B DD4Edd     LD C,(IX+dd)   *  02B5 6C         LD L,H
024E FD4Edd     LD C,(IY+dd)   *  02B6 6D         LD L,L
0251 4F         LD C,A         *  02B7 2Enn       LD L,N
0252 48         LD C,B         *  02B9 ED4F       LD R,A
0253 49         LD C,C         *  02BB ED7Bnnnn   LD SP,(NN)
0254 4A         LD C,D         *  02BF F9         LD SP,HL
0255 4B         LD C,E         *  02C0 DDF9       LD SP,IX
02C2 FDF9       LD SP,IY       *  0334 CB8B       RES 1,E
02C4 31nnnn     LD SP,NN       *  0336 CB8C       RES 1,H
02C7 EDA8       LDD            *  0338 CB8D       RES 1,L
02C9 EDB8       LDDR           *  033A CB96       RES 2,(HL)
02CB EDA0       LDI            *  033C DDCBdd96   RES 2,(IX+dd)
02CD EDB0       LDIR           *  0340 FDCBdd96   RES 2,(IY+dd)
02CF ED44       NEG            *  0344 CB97       RES 2,A
02D1 00         NOP            *  0346 CB90       RES 2,B
02D2 B6         OR (HL)        *  0348 CB91       RES 2,C
02D3 DDB6dd     OR (IX+dd)     *  034A CB92       RES 2,D
02D6 FDB6dd     OR (IY+dd)     *  034C CB93       RES 2,E
02D9 B7         OR A           *  034E CB94       RES 2,H
02DA B0         OR B           *  0350 CB95       RES 2,L
02DB B1         OR C           *  0352 CB9E       RES 3,(HL)
02DC B2         OR D           *  0354 DDCBdd9E   RES 3,(IX+dd)
02DD B3         OR E           *  0358 FDCBdd9E   RES 3,(IY+dd)
02DE B4         OR H           *  035C CB9F       RES 3,A
02DF B5         OR L           *  035E CB98       RES 3,B
02E0 F6nn       OR N           *  0360 CB99       RES 3,C
```

```
02E2 EDBB       OTDR          *  0362 CB9A       RES 3,D
02E4 EDB3       OTIR          *  0364 CB9B       RES 3,E
02E6 ED79       OUT (C),A     *  0366 CB9C       RES 3,H
02E8 ED41       OUT (C),B     *  0368 CB9D       RES 3,L
02EA ED49       OUT (C),C     *  036A CBA6       RES 4,(HL)
02EC ED51       OUT (C),D     *  036C DDCBddA6   RES 4,(IX+dd)
02EE ED59       OUT (C),E     *  036O FDCBddA6   RES 4,(IY+dd)
02FO ED61       OUT (C),H     *  0374 CBA7       RES 4,A
02F2 ED69       OUT (C),L     *  0376 CBAO       RES 4,B
D2F4 D3nn       OUT (N),A     *  0378 CBA1       RES 4,C
02F6 EDAB       OUTD          *  037A CBA2       RES 4,D
02F8 EDA3       OUTI          *  037C CBA3       RES 4,E
02FA F1         POP AF        *  037E CBA4       RES 4,H
02FB C1         POP BC        *  038O CBA5       RES 4,L
02FC D1         POP DE        *  0382 CBAE       RES 5,(HL)
02FD E1         POP HL        *  0384 DDCBddAE   RES 5,(IX+dd)
02FE DDE1       POP IX        *  0388 FDCBddAE   RES 5,(IY+dd)
0300 FDE1       POP IY        *  038C CBAF       RES 5,A
0302 F5         PUSH AF       *  038E CBA8       RES 5,B
0303 C5         PUSH BC       *  039O CBA9       RES 5,C
0304 D5         PUSH DE       *  0392 CBAA       RES 5,D
0305 E5         PUSH HL       *  0394 CBAB       RES 5,E
0306 DDE5       PUSH IX       *  0396 CBAC       RES 5,H
0308 FDE5       PUSH IY       *  0398 CBAD       RES 5,L
030A CB86       RES O,(HL)    *  039A CBB6       RES 6,(HL)
030C DDCBdd86   RES O,(IX+dd) *  039C DDCBddB6   RES 6,(IX+dd)
031O FDCBdd86   RES O,(IY+dd) *  03AO FDCBddB6   RES 6,(IY+dd)
0314 CB87       RES O,A       *  03A4 CBB7       RES 6,A
0316 CB8O       RES O,B       *  03A6 CBBO       RES 6,B
0318 CB81       RES O,C       *  03A8 CBB1       RES 6,C
031A CB82       RES O,D       *  03AA CBB2       RES 6,D
031C CB83       RES O,E       *  03AC CBB3       RES 6,E
031E CB84       RES O,H       *  03AE CBB4       RES 6,H
0320 CB85       RES O,L       *  03BO CBB5       RES 6,L
0322 CB8E       RES 1,(HL)    *  03B2 CBBE       RES 7,(HL)
0324 DDCBdd8E   RES 1,(IX+dd) *  03B4 DDCBddBE   RES 7,(IX+dd)
0328 FDCBdd8E   RES 1,(IY+dd) *  03B8 FDCBddBE   RES 7,(IY+dd)
032C CB8F       RES 1,A       *  03BC CBBF       RES 7,A
032E CB88       RES 1,B       *  03BE CBB8       RES 7,B
0330 CB89       RES 1,C       *  03CO CBB9       RES 7,C
0332 CB8A       RES 1,D       *  03C2 CBBA       RES 7,D
03C4 CBBB       RES 7,E       *  043F C7         RST 00
03C6 CBBC       RES 7,H       *  044O CF         RST 08
03C8 CBBD       RES 7,L       *  0441 D7         RST 10H
03CA C9         RET           *  0442 DF         RST 18H
03CB D8         RET C         *  0443 E7         RST 20H
03CC F8         RET M         *  0444 EF         RST 28H
03CD DO         RET NC        *  0445 F7         RST 30H
03CE CO         RET NZ        *  0446 FF         RST 38H
03CF FO         RET P         *  0447 9E         SBC A,(HL)
03DO E8         RET PE        *  0448 DD9Edd     SBC A,(IX+dd)
03D1 EO         RET PO        *  044B FD9Edd     SBC A,(IY+dd)
03D2 C8         RET Z         *  044E 9F         SBC A,A
03D3 ED4D       RETI          *  044F 98         SBC A,B
03D5 ED45       RETN          *  045O 99         SBC A,C
03D7 CB16       RL (HL)       *  0451 9A         SBC A,D
03D9 DDCBdd16   RL (IX+dd)    *  0452 9B         SBC A,E
03DD FDCBdd16   RL (IY+dd)    *  0453 9C         SBC A,H
03E1 CB17       RL A          *  0454 9D         SBC A,L
03E3 CB1O       RL B          *  0455 DEnn       SBC A,N
03E5 CB11       RL C          *  0457 ED42       SBC HL,BC
03E7 CB12       RL D          *  0459 ED52       SBC HL,DE
03E9 CB13       RL E          *  045B ED62       SBC HL,HL
03EB CB14       RL H          *  045D ED72       SBC HL,SP
03ED CB15       RL L          *  045F 37         SCF
03EF 17         RLA           *  0460 CBC6       SET O,(HL)
```

```
03F0 CB06       RLC (HL)        *  0462 DDCBddC6  SET 0,(IX+dd)
03F2 DDCBdd06   RLC (IX+dd)     *  0466 FDCBddC6  SET 0,(IY+dd)
03F6 FDCBdd06   RLC (IY+dd)     *  046A CBC7      SET 0,A
03FA CB07       RLC A           *  046C CBC0      SET 0,B
03FC CB00       RLC B           *  046E CBC1      SET 0,C
03FE CB01       RLC C           *  0470 CBC2      SET 0,D
0400 CB02       RLC D           *  0472 CBC3      SET 0,E
0402 CB03       RLC E           *  0474 CBC4      SET 0,H
0404 CB04       RLC H           *  0476 CBC5      SET 0,L
0406 CB05       RLC L           *  0478 CBCE      SET 1,(HL)
0408 07         RLCA            *  047A DDCBddCE  SET 1,(IX+dd)
0409 ED6F       RLD             *  047E FDCBddCE  SET 1,(IY+dd)
040B CB1E       RR (HL)         *  0482 CBCF      SET 1,A
040D DDCBdd1E   RR (IX+dd)      *  0484 CBC8      SET 1,B
0411 FDCBdd1E   RR (IY+dd)      *  0486 CBC9      SET 1,C
0415 CB1F       RR A            *  0488 CBCA      SET 1,D
0417 CB18       RR B            *  048A CBCB      SET 1,E
0419 CB19       RR C            *  048C CBCC      SET 1,H
041B CB1A       RR D            *  048E CBCD      SET 1,L
041D CB1B       RR E            *  0490 CBD6      SET 2,(HL)
041F CB1C       RR H            *  0492 DDCBddD6  SET 2,(IX+dd)
0421 CB1D       RR L            *  0496 FDCBddD6  SET 2,(IY+dd)
0423 1F         RRA             *  049A CBD7      SET 2,A
0424 CB0E       RRC (HL)        *  049C CBD0      SET 2,B
0426 DDCBdd0E   RRC (IX+dd)     *  049E CBD1      SET 2,C
042A FDCBdd0E   RRC (IY+dd)     *  04A0 CBD2      SET 2,D
042E CB0F       RRC A           *  04A2 CBD3      SET 2,E
0430 CB08       RRC B           *  04A4 CBD4      SET 2,H
0432 CB09       RRC C           *  04A6 CBD5      SET 2,L
0434 CB0A       RRC D           *  04A8 CBDE      SET 3,(HL)
0436 CB0B       RRC E           *  04AA DDCBddDE  SET 3,(IX+dd)
0438 CB0C       RRC H           *  04AE FDCBddDE  SET 3,(IY+dd)
043A CB0D       RRC L           *  04B2 CBDF      SET 3,A
043C 0F         RRCA            *  04B4 CBD8      SET 3,B
043D ED67       RRD             *  04B6 CBD9      SET 3,C
04B8 CBDA       SET 3,D         *  052C CB20      SLA B
04BA CBDB       SET 3,E         *  052E CB21      SLA C
04BC CBDC       SET 3,H         *  0530 CB22      SLA D
04BE CBDD       SET 3,L         *  0532 CB23      SLA E
04C0 CBE6       SET 4,(HL)      *  0534 CB24      SLA H
04C2 DDCBddE6   SET 4,(IX+dd)   *  0536 CB25      SLA L
04C6 FDCBddE6   SET 4,(IY+dd)   *  0538 CB2E      SRA (HL)
04CA CBE7       SET 4,A         *  053A DDCBdd2E  SRA (IX+dd)
04CC CBE0       SET 4,B         *  053E FDCBdd2E  SRA (IY+dd)
04CE CBE1       SET 4,C         *  0542 CB2F      SRA A
04D0 CBE2       SET 4,D         *  0544 CB28      SRA B
04D2 CBE3       SET 4,E         *  0546 CB29      SRA C
04D4 CBE4       SET 4,H         *  0548 CB2A      SRA D
04D6 CBE5       SET 4,L         *  054A CB2B      SRA E
04D8 CBEE       SET 5,(HL)      *  054C CB2C      SRA H
04DA DDCBddEE   SET 5,(IX+dd)   *  054E CB2D      SRA L
04DE FDCBddEE   SET 5,(IY+dd)   *  0550 CB3E      SRL (HL)
04E2 CBEF       SET 5,A         *  0552 DDCBdd3E  SRL (IX+dd)
04E4 CBE8       SET 5,B         *  0556 FDCBdd3E  SRL (IY+dd)
04E6 CBE9       SET 5,C         *  055A CB3F      SRL A
04E8 CBEA       SET 5,D         *  055C CB38      SRL B
04EA CBEB       SET 5,E         *  055E CB39      SRL C
04EC CBEC       SET 5,H         *  0560 CB3A      SRL D
04EE CBED       SET 5,L         *  0562 CB3B      SRL E
04F0 CBF6       SET 6,(HL)      *  0564 CB3C      SRL H
04F2 DDCBddF6   SET 6,(IX+dd)   *  0566 CB3D      SRL L
04F6 FDCBddF6   SET 6,(IY+dd)   *  0568 96        SUB (HL)
04FA CBF7       SET 6,A         *  0569 DD96dd    SUB (IX+dd)
04FC CBF0       SET 6,B         *  056C FD96dd    SUB (IY+dd)
04FE CBF1       SET 6,C         *  056F 97        SUB A
0500 CBF2       SET 6,D         *  0570 90        SUB B
```

```
0502 CBF3       SET 6,E         *  0571 91        SUB C
0504 CBF4       SET 6,H         *  0572 92        SUB D
0506 CBF5       SET 6,L         *  0573 93        SUB E
0508 CBFE       SET 7,(HL)      *  0574 94        SUB H
050A DDCBddFE   SET 7,(IX+dd)   *  0575 95        SUB L
050E FDCBddFE   SET 7,(IY+dd)   *  0576 D6nn      SUB N
0512 CBFF       SET 7,A         *  0578 AE        XOR (HL)
0514 CBF8       SET 7,B         *  0579 DDAEdd    XOR (IX+dd)
0516 CBF9       SET 7,C         *  057C FDAEdd    XOR (IY+dd)
0518 CBFA       SET 7,D         *  057F AF        XOR A
051A CBFB       SET 7,E         *  0580 A8        XOR B
051C CBFC       SET 7,H         *  0581 A9        XOR C
051E CBFD       SET 7,L         *  0582 AA        XOR D
0520 CB26       SLA (HL)        *  0583 AB        XOR E
0522 DDCBdd26   SLA (IX+dd)     *  0584 AC        XOR H
0526 FDCBdd26   SLA (IY+dd)     *  0585 AD        XOR L
052A CB27       SLA A           *  0586 EEnn      XOR N
```

Stampato in Italia da:
S.p.A. Alberto Matarelli - Milano - Stabilimento Grafico

Cod. 329A

PROGRAMMARE IN ASSEMBLER

ALAIN PINAUD

Una schiera sempre più vasta di utenti di personal computer vorrebbe avvicinarsi alla programmazione in assembler, ma esita per molteplici ragioni, non ultima perché ritengono l'assembler terribilmente complesso e necessitante di lunghi studi. Questo però non è vero se si riesce, come riesce all'autore, a semplificare i concetti, anche quelli più complessi. È possibile allora, in poco tempo e con semplicità, apprendere quei rudimenti che consentiranno, poi, di programmare autonomamente. Questo utilizzando numerosi esempi pratici.
Una volta conosciuti e spiegati, comunque, i principi base del linguaggio assembler rimangono validi per qualsiasi microprocessore, a 8, a 16, a 32, come pure a 64 bit. Poiché, però, bisogna far riferimento ad un assembler esistente, si è scelto quello dello Z80, che, anche se datato dal punto di vista tecnologico, è didatticamente tra i più validi in quanto dotato del set di istruzioni più ampio nella sua categoria.

- Nozioni di base
- Introduzione all'assembler
- Assembler Z80
- Pseudo-istruzioni e macro-istruzioni
- Pratica dell'assembler
- Software di supporto
- Relazioni con i linguaggi evoluti
- Matematica e informatica
- Correzione degli esercizi
- Il codice ASCII
- Il set di istruzioni dello Z80

GRUPPO EDITORIALE JACKSON

L. 17.000 Cod. 329A ISBN 88-7056-105-4