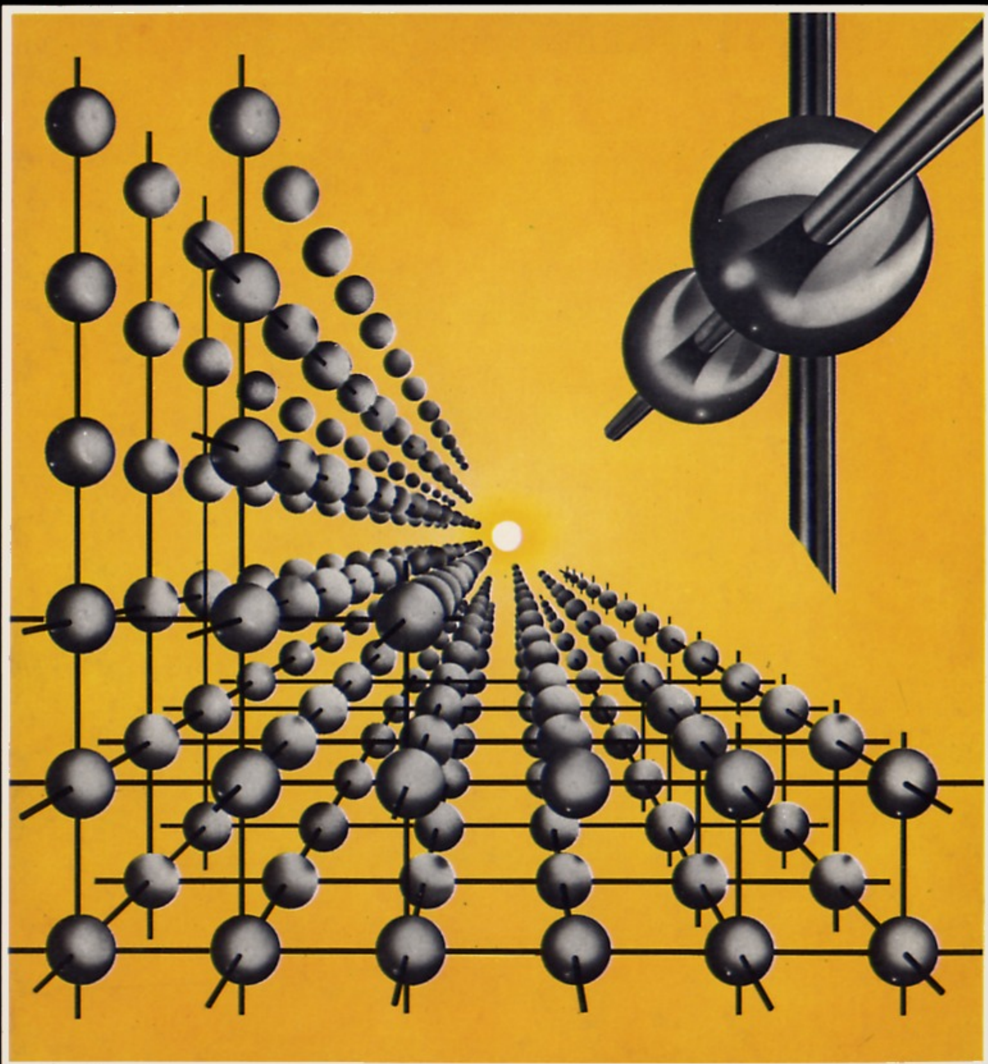
MSX: LINGUAGGIO MACCHINA E ASSEMBLY

Parliamo la lingua del nostro computer

di IAN SINCLAIR



edizioni JCE

MSX: LINGUAGGIO MACCHINA E ASSEMBLY

Parliamo la lingua del nostro computer

di IAN SINCLAIR

Traduzione di MARIAGRAZIA LARI

JACOPO CASTELFRANCHI EDITORE
Via dei Lavoratori, 124
CINISELLO BALSAMO (MI)

Pubblicato per la prima volta in Gran Bretagna da:

Collins Professional and Technical Books 1985
William Collins Sons & Co. Ltd

Titolo originale:
Introducing MSX Assembly Language and Machine Code

Copyright © Ian Sinclair 1985
Copyright © per l'edizione italiana: Edizioni JCE 1985

Prima edizione: Ottobre 1985

I programmi inseriti in questo libro hanno scopo esemplificativo e educativo. Sono stati verificati attentamente ma non sono garantiti per alcuno scopo particolare. Nonostante sia stata presa ogni precauzione, l'editore non può essere ritenuto responsabile per errori che potrebbero avvenire nell'esecuzione.

TUTTI I DIRITTI RISERVATI

Non può essere fatto alcun utilizzo di questo libro, programmi e/o testi, eccetto per studio personale dell'acquirente, senza il preventivo permesso scritto dell'editore.

Le riproduzioni in ogni forma e per qualsiasi scopo sono proibite.

Fotocomposto elettronicamente da: JCE S.A.S.

Stampato in Italia da:
Gemm Grafica S.r.l.
Via Magretti
Paderno Dugnano (MI)

INDICE

Prefazione .... pag. V

Capitolo 1: ROM, RAM, bytes e bit .... pag. 1
Capitolo 2: Frugando dentro l'MSX .... pag. 21
Capitolo 3: Il miracoloso microprocessore .... pag. 35
Capitolo 4: I dettagli dello Z80 .... pag. 51
Capitolo 5: Le azioni dei registri .... pag. 69
Capitolo 6: Allarghiamo la visuale .... pag. 89
Capitolo 7: Cassette e parametri .... pag. 111
Capitolo 8: Correzione, messa a punto, controllo e ZEN pag. 135
Capitolo 9: L'ultima battuta .... pag. 155
Appendice A: Elenco delle parole chiave, degli indirizzi e dei bytes di spostamento .... pag. 181
Appendice B: Codifica dei numeri in BCD .... pag. 183
Appendice C: Memorizzazione delle variabili .... pag. 185
Appendice D: Conversione tra decimale ed esadecimale .... pag. 187
Appendice E: I codici operativi dello Z80, con mnemoniche, codici esadecimali, codici decimali e codici binari .... pag. 189
Appendice F: Metodi di indirizzamento dello Z80 .... pag. 201
Appendice G: I tempi di esecuzione delle istruzioni .... pag. 203
Appendice H: Il codice macchina espresso con una riga DATA .... pag. 205

PREFAZIONE

In genere, chi usa un computer MSX e sa già scrivere ed utilizzare programmi BASIC, è spesso riluttante ad intraprendere il passo successivo, cioè imparare a servirsi del linguaggio usato dal microprocessore Z80, il cuore del microcomputer MSX.
Questa riluttanza è comprensibile. Molti libri che trattano dello Z80 presumono che il lettore conosca già non solo i termini usati, ma anche gran parte del funzionamento del chip del microprocessore. Altri libri, invece, appaiono agli occhi del principiante, come se fossero scritti in una lingua sconosciuta senza sottotitoli. Altri ancora hanno un inizio promettente, ma poi perdono per strada il lettore inesperto, perchè affrontano improvvisamente temi assai più difficili o argomenti, come le routines aritmetiche, di ben scarsa utilità per la maggior parte dei lettori.
Questo libro è diretto al vero principiante del linguaggio macchina: chi possiede un computer MSX e sa programmare in BASIC, ma non ha la più pallida idea di quanto avvenga all'interno del suo computer. Questo libro non ha lo scopo di farvi diventare degli esperti nella programmazione in linguaggio macchina dello Z80, perché potrete arrivarci solo con molta esperienza, molte letture ed un vero desiderio di risolvere i problemi.
Questo libro non pretende neppure di spiegarvi tutto ciò che lo Z80 è in grado di fare. Ciò che spero di poter ottenere, invece, è di presentarvi l'inizio di un grosso argomento e di farvi capire alcuni dei perché e dei come della programmazione dello Z80.
Un nuovo, immenso mondo vi si aprirà davanti agli occhi, capirete molto meglio come funziona il vostro computer e potrete finalmente comprendere libri più complessi sulla programmazione in linguaggio macchina.
Devo, però, chiarire bene un punto: questo tipo di programmazione non è mai facile. Può diventare familiare, può diventare un normale modo di operare, ma non sarà mai facile. Anche l'imparare è un compito che richiede impegno, un certo sforzo per capire che cosa avviene ed un po' di tempo per fare esercizio sul vostro computer.
Per aiutarvi, ho incluso la descrizione di un utilissimo programma, chiamato ZEN Assembler; rimane però il fatto che l'impegno principale deve venire da voi lettori.
Penso, però, che ne valga proprio la pena.
Buon lavoro, dunque.

CAPITOLO 1

ROM, RAM, bytes e bit

Quando accendete il televisore, usate l'elettricità e riceverete un segnale TV, però non vedete il macchinario che genera la corrente, non vedete la telecamera che produce il segnale TV e non vedete il trasmettitore che emette il segnale. Potéte tranquillamente godervi i programmi TV senza dovervi mai preoccupare di come il segnale arrivi fino al vostro televisore o di che cosa accada dentro di esso. Potete anche divertirvi ad usare il vostro computer programmando in BASIC o utilizzando programmi scritti da altri, senza nemmeno chiedervi quali istruzioni ci siano nel programma. Tuttavia, c'è una differenza: potete utilizzare il vostro computer in modo diverso, molto più efficacemente, oppure potete scrivere programmi che "girano" molto più rapidamente o che eseguono operazioni non contemplate dal BASIC, se conoscete qualcosa di quello che accade dentro il vostro computer.
Questo equivale a conoscere i segreti del linguaggio in cui è programmato il computer che avete acquistato — chiamato "codice macchina".
Una delle cose che scoraggiano gli utilizzatori di computer dal fare tentativi per andare oltre il BASIC è il numero di parole nuove che saltano fuori. Non potete fare a meno di queste nuove parole, perchè sono necessarie per descrivere cose nuove per voi. Gli autori di molti libri sulla programmazione in codice macchina sembrano partire dal presupposto che il lettore abbia una base di elettronica e quindi conosca già tutti i termini usati.
Io partirò dal presupposto che non abbiate questa base. Darò per scontato solo il fatto che voi possediate un computer MSX da 64K di memoria, e che abbiate qualche esperienza di programmazione in BASIC sul vostro computer. Una certa esperienza di programmazione BASIC è essenziale, altrimenti sarà assai più difficile, per voi, comprendere il codice macchina. Tutto questo significa che cominceremo dal punto giusto, cioé dall'inizio.
In questo libro, non voglio dover interrompere delle importanti spiegazioni con dettagli tecnici o matematici: questi si troveranno nelle Appendici. In questo modo, potrete leggere la spiegazione completa di alcuni punti se ne avrete il desiderio, o saltarla in caso contrario. Per cominciare, dobbiamo occuparci della memoria.L'unità di memoria di un computer è, per quanto ci riguarda, un circuito elettrico che funziona come un interruttore. Quando entrate in una stanza, accendete la luce e non state certo a ragionare sul fatto che la luce rimane accesa finchè non premete di nuovo l'interruttore. Non andate a dire ai vostri amici che il circuito della

luce ha una memoria, eppure ogni unità di memoria di un computer è proprio una specie di interruttore in miniatura che può essere acceso o spento. Ciò che lo fa diventare una memoria è che rimarrà nella condizione in cui si trova, acceso o spento, finché non verrà apportato un cambiamento.
Un'unità di memoria del computer di questo tipo è chiamata "BIT", abbreviazione di Binary digit, cioé un'unità che può essere azionata come interruttore solo in due possibili modi.
La memoria di un computer consiste di un enorme numero di interruttori, incredibilmente piccoli, ognuno dei quali può essere acceso o spento. Non c'é nessun'altra possibilità, nessuna posizione intermedia.
Continueremo a servirci dell'idea di interruttore, perché è molto utile per spiegare come usiamo la memoria.

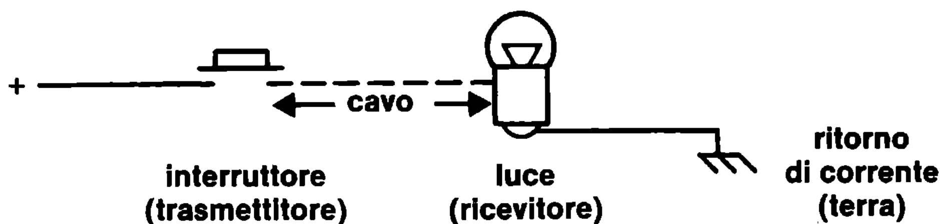
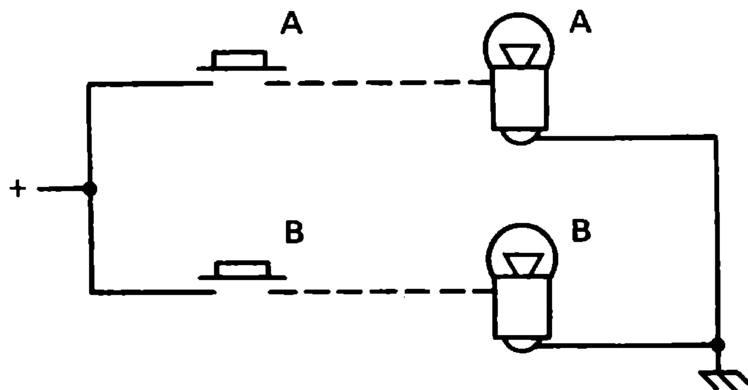


Fig. 1.1. Un interruttore a linea singola ed un sistema di segnalazione luminoso.



A	B
spenta	spenta
spenta	accesa
accesa	spenta
accesa	accesa

Fig. 1.2. Segnalazione a due linee. Si possono inviare quattro diversi segnali.

Supponiamo di voler fare dei segnali, avendo a disposizione dei circuiti elettrici e degli interruttori. Potremo usare un circuito come quello della Fig. 1.1. Quando l'interruttore è abbassato, la luce è accesa e potremmo dargli il significato di SI. Quando l'interruttore è alzato, la luce si spegne e potremmo dargli il significato di NO. Potreste dare altri due significati, a vostro piacimento, a queste due condizioni (chiamate "stati") della luce, ma ciò vale finché ce ne sono solo due. Le cose cambiano, se usate due interruttori e due luci, come nella Fig. 1.2.

Ora sono possibili quattro diverse combinazioni delle luci: (a) ambedue spente; (b) A accesa, B spenta; (c) A spenta, B accesa; (d) ambedue accese. Questo gruppo di quattro possibilità vuol dire che noi potremmo dare ai nostri segnali quattro diversi significati. L'uso di una linea permette due possibili codici, l'uso di due linee ne permette quattro. Se avete voglia di verificare le possibili combinazioni, vedrete che l'uso di tre linee permetterà otto diversi codici. Riflettendo un momento, possiamo fare questa considerazione: se 4 è 2x2, e 8 è 2x2x2, allora quattro linee possono permettere 2x2x2x2, cioé 16, codici. Infatti, questo è vero, e poiché scriviamo di solito 2x2x2x2 come 2^4 (2 elevato a 4), possiamo trovare quanti codici possono essere trasmessi da un certo numero di linee. Per esempio, otto linee possono definire 2^8 codici, cioè 256 codici. Un insieme di otto interruttori, quindi, potrebbe essere impostato in modo da definire 256 diversi significati. Sta a noi decidere come vogliamo usare questi segnali. L'insieme di otto è particolarmente importante, perché la memoria del vostro computer è basata appunto su gruppi di otto.

Un modo assai utile di usare questi segnali acceso/spento, è chiamato codice binario. Il codice binario è una maniera per scrivere i numeri usando solo due cifre, 0 e 1. Possiamo immaginare che 0 significhi "spento" e 1 "acceso", così 256 diversi numeri potrebbero essere segnalati con otto interruttori, dando a 0 il significato di spento e ad 1 il significato di acceso. Questo gruppo di otto viene chiamato "byte" ed è l'unità di misura che usiamo per specificare la dimensione della memoria del nostro computer. Ecco perché i numeri 8 e 256 si incontrano così spesso nella programmazione in codice macchina.

Per indicare un numero, i singoli bit di un byte sono disposti in modo analogo a quello da noi usato normalmente per indicare i numeri.

Quando scrivete un numero decimale, per esempio 253, il 3 significa 3 unità, il 5 viene scritto all'immediata sinistra del 3 e significa 5 decine, il 2 viene scritto due posti più a sinistra del 3 e significa 2 centinaia. Queste posizioni indicano l'importanza (o la significatività) di una cifra, come mostrato nella Fig. 1.3. Il 3 di 253 viene chiamato "la cifra (digit) meno significativa", mentre il 2 è la "cifra (digit) più significativa". Cambiate il 3 in 4 oppure in 2 e il cambiamento sarà solo di una parte su 253. Cambiate il 2 in 1 oppure in 3 ed il cambiamento sarà di 100 parti su 253, molto più rilevante.

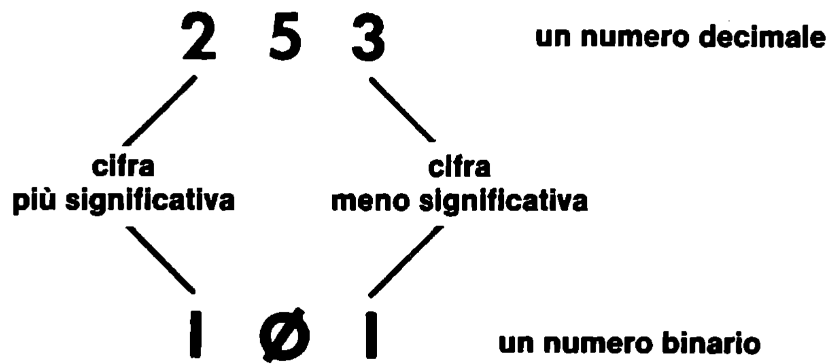


Fig. 1.3. La significatività delle cifre. Il nostro sistema numerico usa la posizione di una cifra in un numero per indicare la sua significatività o importanza.

Dopo aver dato un'occhiata ai bit ed ai bytes, ritorniamo alla nostra immagine della memoria come insieme di interruttori.
Nel nostro computer, abbiamo bisogno di due tipi di memoria. Un tipo deve essere permanente, come gli interruttori meccanici o i collegamenti fissi, poiché deve essere usata per trattenere le istruzioni in codice numerico che rendono operativo il computer. Questo tipo di memoria viene chiamato ROM (Read Only Memory = Memoria di Sola Lettura). Da ciò ne consegue che potete individuare e copiare quello che è contenuto nella memoria, ma non potete né cancellarlo né modificarlo. La ROM è la parte più importante del vostro computer, poiché contiene tutte le istruzioni che mettono il computer in grado di eseguire le azioni del BASIC. Quando scrivete un programma in BASIC, il computer lo memorizza, sotto forma di un altro insieme di istruzioni in codice numerico, in una parte della memoria che può essere usata, cancellata, di nuovo usata e così via. Questo è un tipo diverso di memoria, che può essere sia "letta" che "scritta" e, se fossimo coerenti, dovremmo chiamarla RWM (Read—Write Memory = Memoria di Lettura e Scrittura). Sfortunatamente, non siamo molto coerenti e la chiamiamo RAM (Random Access Memory = Memoria ad Accesso Casuale). Questo nome venne usato ai primordi della programmazione per distinguere questo tipo di memoria da un altro che aveva un diverso funzionamento. Ma ormai siamo legati al nome RAM e forse lo saremo per sempre! La differenza principale fra la ROM e la RAM è che ogni bit della RAM si comporta come un interruttore solo se è alimentato dall'elettricità. Quando togliete l'alimentazione elettrica, l'azione dell'interruttore si arresta. Se tornate a fornire l'alimentazione, l'azione dell'interruttore della RAM ricomincerà — ma non nel modo in cui si trovava prima. Ogni bit della RAM può essere"acceso" o "spento", ma questo avviene in modo casuale. Quando spegnete il computer, allora, perdete tutto ciò che era memorizzato nella RAM, poiché quando lo riaccendete, avrete solo un insieme di segnali casuali. È un po' come lanciare in aria un "puzzle": non potete certo aspettarvi che ricada ancora in ordine.

LE STRANEZZE DEL CODICE NUMERICO

Ora possiamo tornare ai bytes. Abbiamo visto prima che un byte è un gruppo di 8 bit che possono essere disposti in 256 diversi modi, a seconda di quali bit sono 1 e quali 0. Comunque, il modo più utile per disporre questi bit è quello che noi chiamiamo codice binario. Ogni diversa disposizione dei bit viene usata per rappresentare un numero decimale da 0 a 255 (non da 1 a 256, poiché abbiamo bisogno di un codice per rappresentare lo zero). Ogni byte dei 65536 bytes della RAM di quasi tutti i computer MSX può immagazzinare un numero compreso tra 0 e 255.
I numeri, di per sé, non sono di grande utilità, e non troveremmo certo molto utile il nostro computer se potesse gestire solo numeri da 0 a 255, perciò usiamo questi numeri come codici. Ogni codice numerico, infatti, può essere usato per indicare diverse cose. Se avete utilizzato i codici ASCII in BASIC, saprete che ogni lettera dell'alfabeto, ogni segno d'interpunzione ed ognuna delle cifre da 0 a 9 è codificata in ASCII sotto forma di un numero compreso fra 32 (spazio) e 127 (freccia sinistra). Quella selezione vi lascia a disposizione parecchi numeri di codice ASCII che possono essere usati per altri scopi, quali i caratteri grafici.
Il codice ASCII non è comunque l'unico. I computer MSX usano i loro particolari significati in codice per altri numeri compresi tra 0 e 255. Per esempio, quando battete la parola PRINT in una riga di programma, ciò che viene posto nella memoria del computer (quando premete RETURN) non è la sequenza dei codici ASCII per la parola PRINT, rappresentata da 80,82,73,78,84 — un byte per ogni lettera. Ciò che viene effettivamente immesso in memoria, è un solo byte: la forma binaria del numero 145. Questo singolo byte è detto "simbolo" (token) e può essere usato in due modi dal computer. Il primo modo è di assegnare i codici ASCII per i caratteri che compongono la parola PRINT. Essi sono memorizzati nella ROM, così, quando farete il listato di un programma, vedrete apparire la parola PRINT e non un carattere il cui codice è 145. Il secondo uso, ancora più importante, del "simbolo", è di definire un insieme di istruzioni, anch'esse contenute nella ROM sotto forma di codici numerici. Queste istruzioni faranno sì che i caratteri vengano stampati sullo schermo e i numeri che compongono questi codici sono, appunto, quello che noi chiamiamo *codice macchina.* Essi controllano *direttamente* ciò che fa la "macchina". Il controllo diretto è la ragione principale per usare il codice macchina. Con il BASIC, gli unici comandi che possiamo usare sono quelli per cui è previsto un "simbolo". Se non c'é un "simbolo" per una certa azione, non potremo eseguirla. Normalmente, mettiamo insieme una serie di istruzioni BASIC per portare a compimento l'azione desiderata: questo, in sostanza, è un programma BASIC. Usando il codice macchina, invece, possiamo formare i nostri comandi e quindi fare ciò che desideriamo. Fra parentesi, il fatto che PRINT generi un "simbolo", è la

ragione per cui è possibile usare un punto interrogativo, ?, al posto di PRINT. Il computer MSX è stato progettato in modo che anche ? non posto tra virgolette immetta in memoria 145.

UNA BREVE ESERCITAZIONE

Per aiutarvi a "digerire" tutte queste informazioni provate un breve programma. Questo, nella Fig. 1.4, è progettato per visualizzare alcuni dei simboli che si trovano già memorizzati nella ROM e fa uso dell'istruzione BASIC PEEK. PEEK deve essere seguita da un numero o da una variabile numerica entro parentesi e significa: "Trova quale byte è immagazzinato a questo numero di indirizzo". Tutti i bytes della memoria del vostro computer sono numerati da 0 in avanti, un numero per ogni byte. Dal momento che questo procedimento è simile alla numerazione delle case di una strada, chiamiamo "indirizzi" questi numeri. L'azione di PEEK è trovare quale numero, compreso tra 0 e 255, sia memorizzato a ciascun indirizzo. Il computer MSX converte automaticamente questi numeri dalla forma binaria, in cui si trovano memorizzati, nel corrispondente numero decimale. Se usiamo CHR$ nel nostro programma, possiamo stampare il carattere il cui codice ASCII è il numero indicato dalla funzione PEEK.

```
10 FOR N=32472 TO 32549
20 PRINT CHR$(PEEK(N));:NEXT
```

Fig. 1.4. Un programma che visualizza la dicitura del copyright.

Il programma usa la variabile N come numero di indirizzo e stampa i caratteri che vi si trovano immagazzinati. Come potete constatare facendo "girare" il programma, viene stampata la dicitura del copyright che appare per qualche istante sullo schermo quando accendete il computer MSX. Questa dicitura non potrebbe apparire se i codici ASCII (o degli altri codici) non fossero memorizzati nella ROM. Come vedete, essi sono memorizzati fra gli indirizzi 32472 e 32549. Il programma, attraverso l'istruzione PEEK, trova ogni byte memorizzato e poi lo stampa sotto forma di carattere ASCII con l'istruzione CHR$. La funzione PEEK può essere usata per trovare ciò che è contenuto in una qualsiasi parte della memoria, se conosciamo i numeri di indirizzo da usare.
Ora proviamo un programma, quello della Fig. 1.5, che usa la funzione PEEK per un'altra parte della memoria.

```
10 FOR N=5033 TO 5100
20 PRINTCHR$(PEEK (N));:NEXT
30 PRINT
40 FOR N=5100 TO 5125
```

```
50 PRINTCHR$(PEEK(N));:NEXT
60 PRINT
70 FOR N=5126 TO 5160
80 PRINTCHR$(PEEK(N));:NEXT
90 PRINT:PRINT
```

Fig.1.5 Dove sono memorizzate le parole dei tasti funzione programmabili.

Questa volta stiamo considerando dei bytes immagazzinati a numeri di indirizzo più bassi, da 5033 a 5160. Quando il programma viene eseguito, vedrete le parole che sono assegnate ai tasti funzione programmabili. Ora, tutto questo potrebbe sembrarvi un po' strano, se ci pensate bene. Infatti, voi sapete che è possibile riprogrammare questi tasti funzione usando comandi come KEY1, "PRINTTAB". Questa riprogrammazione non sarebbe possibile se le parole chiave non fossero contenute nella memoria RAM, perché solo la RAM può essere modificata. Ma noi, invece, abbiamo trovato queste parole memorizzate nella memoria ROM. La risposta è che le parole devono essere contenute sia nella ROM che nella RAM. Devono essere contenute nella RAM mentre il computer è acceso, in modo che possiate modificarle come volete. Devono essere contenute anche nella ROM perché altrimenti le parole non sarebbero disponibili nel momento in cui accendete il computer. Una delle prime azioni del vostro computer, quando viene acceso, sarà di copiare i bytes che si trovano a questi indirizzi della ROM agli indirizzi della RAM dove verranno immagazzinati. Questi indirizzi si trovano nella parte alta della memoria, a circa 64500.
Ora consideriamo un po' più da vicino un'altra parte della ROM.
La fig. 1.6 é un programma che vi permette di vedere come sono memorizzate le parole chiave del BASIC. Quando farete girare questo programma, vedrete che l'immagazzinamento non è esattamente lineare. Ogni parola è stata immagazzinata senza la prima lettera. Questo è possibile, poiché le parole sono memorizzate in gruppi. Non sono in perfetto ordine alfabetico, ma ogni parola di un gruppo comincia con la stessa lettera e i gruppi sono in ordine. In questo modo, quando fate girare il programma, usando il tasto STOP per controllare il listato, vedrete che il primo gruppo di parole è BS,TN e TTR$.

```
10 N%=14968
20 PRINT N%;" ";PEEK (N%);" ";
30 N%=N%+1:K%=PEEK(N%)
40 IF K%<128 THEN PRINT CHR$ (K%);:GOTO 30
50 PRINT CHR$(K%—128)
60 N%=N%+1
70 IF N%<15647 THEN 20
```

Fig. 1.6. Un programma che visualizza le parole chiave del BASIC MSX.

Queste sono le parole chiave ABS, ATN e ATTR$, memorizzate senza la "A" per risparmiare spazio. Sulla sinistra dello schermo vedrete il numero di indirizzo di ogni parola e poi un altro numero. Il secondo numero, che varia da 0 a 255, è il numero di "displacement" (dislocamento).
Ogni volta che usate una parola chiave del BASIC il computer toglierà la prima lettera e cercherà la parola in questa lista. Quando viene trovata la parola, verrà letto il numero. Per esempio, se avete usato ATN, il computer troverà TN e poi leggerà il numero 6. Poi aggiungerà un numero d'indirizzo (sempre lo stesso) al precedente numero 6 per trovare l'indirizzo a cui si trova immagazzinato il numero del simbolo (token). Il numero del simbolo è ciò che apparirà in un programma. Lo stesso numero 6 che va con ATN può essere usato anche per individuare dove si trova memorizzata la subroutine ATN.
Il programma inizia assegnando ad N% il numero di indirizzo 14968, cioé l'indirizzo da cui comincia la tabella delle parole chiave. La riga 20 stampa il numero di indirizzo, uno spazio e poi il byte che si trova memorizzato a questo indirizzo. Questo sarà il byte di "displacement" (dislocamento). Il punto e virgola posto alla fine della riga 20 fa sì che la stampa continui sulla stessa riga. La riga 30 incrementa N% ed assegna PEEK(N%) a K%. Nella riga 40, il carattere che corrisponde a PEEK(K%) viene stampato sulla stessa riga ed il punto e virgola, insieme all'istruzione GOTO 30, fa in modo che il computer continui a stampare caratteri finché PEEK(K%) è minore di 128. Un numero maggiore di 128 non è un ordinario codice ASCII.
Ora dobbiamo vedere come mai l'ultimo carattere di ogni parola sia codificato in un modo diverso. Infatti, il numero che troviamo per l'ultimo carattere ha 128 addizionato al codice ASCII. Per esempio, il contenuto del primo indirizzo, evidenziato dal programma con la funzione è costituito dai numeri 246, 66 (visualizzato come B) e 211 (visualizzato come S). 246 è il numero di "displacement" per il comando, 66 è il codice ASCII per la lettera B ed infine 211—128=83, che è il codice ASCII per la lettera S. Ed infatti, la parola (A) BS, senza la A, è immagazzinata all'indirizzo 14968. L'ultima lettera viene codificata in modo diverso per risparmiare memoria! Se venisse lasciato uno spazio fra le parole, questo spazio occuperebbe un byte. In questo modo, invece, non c'é alcun spreco, perché l'ultima lettera di una parola ha sempre un numero di codice maggiore di 128, così il computer la può riconoscere facilmente.
Abbiamo seguito lo stesso schema del programma BASIC della Fig. 1.5, usando la riga 50 per stampare la lettera corretta, senza il punto e virgola. In questo modo, l'indirizzo, il codice e la parola successivi verranno stampati sulla riga seguente.C'é un altro insieme di numeri immagazzinato in un'altra parte della memoria, costituito da altri indirizzi.
Questi sono gli indirizzi delle subroutines che eseguono le azioni del BASIC e sono memorizzate nello stesso ordine di queste parole.

LA STRUTTURA DEL COMPUTER MSX

Diamo ora uno sguardo al diagramma del computer MSX nella Fig. 1.7.

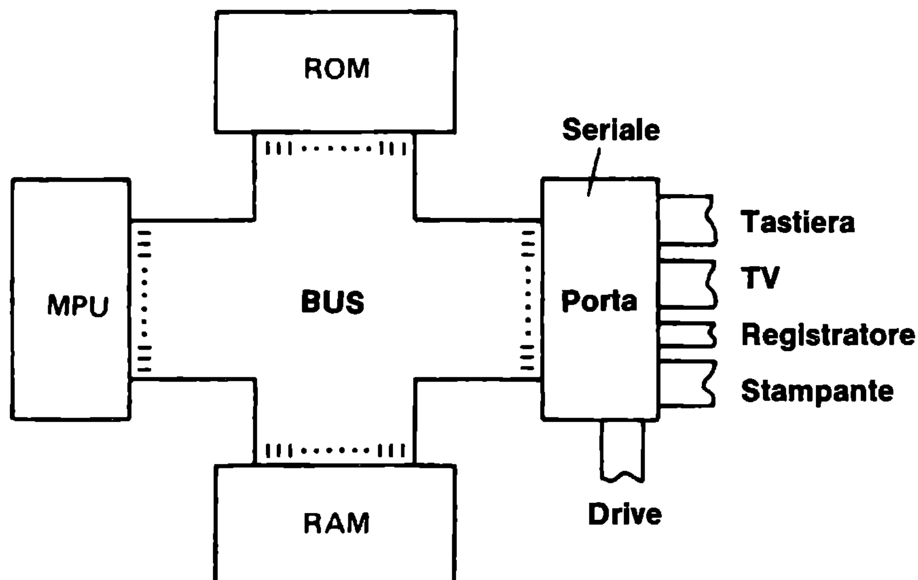


Fig. 1.7. Un diagramma a blocchi di un computer MSX. Le connessioni indicate da "BUS" sono formate da molti collegamenti che uniscono tra loro le unità del sistema.

È un diagramma molto semplice perché ho omesso tutti i dettagli, ma è sufficiente per darvi un'idea di quanto avviene all'interno del vostro computer. Questo tipo di diagramma è chiamato "diagramma a blocchi", perché ogni unità viene rappresentata come un blocco, senza alcun dettaglio sulla sua struttura interna. I diagrammi a blocchi si possono paragonare a carte geografiche su vasta scala che riportano le strade provinciali fra le città, ma non le strade secondarie o quelle comunali. Un diagramma a blocchi è tuttavia sufficiente per indicare i percorsi principali dei segnali elettrici all'interno del computer.

I nomi di due di questi blocchi dovrebbero esservi già familiari, ROM e RAM, ma non gli altri due. Il blocco indicato con MPU è particolarmente importante. MPU significa Unità del Microprocessore (Microprocessor Unit) — alcuni diagrammi a blocchi usano le lettere CPU (Central Processing Unit = Unità Centrale di Elaborazione). La MPU è la principale unità operativa del sistema ed è, in effetti, una sola unità. La MPU è un singolo integrato — uno di quei "chip" al silicio di cui avrete certo sentito parlare, incassato in una piastra di plastica nera e fornito di 40 "piedini" (pin) di connessione, sistemati in due file di 20 ciascuna (Fig. 1.8).

Ci sono diversi tipi di MPU in commercio: quella del vostro computer MSX è chiamata Z80 (o Z80A). È assai diversa da altre MPU, per cui libri su MPU quali il chip 6502 o il chip 68008 non vi aiuteranno a capire lo Z80.

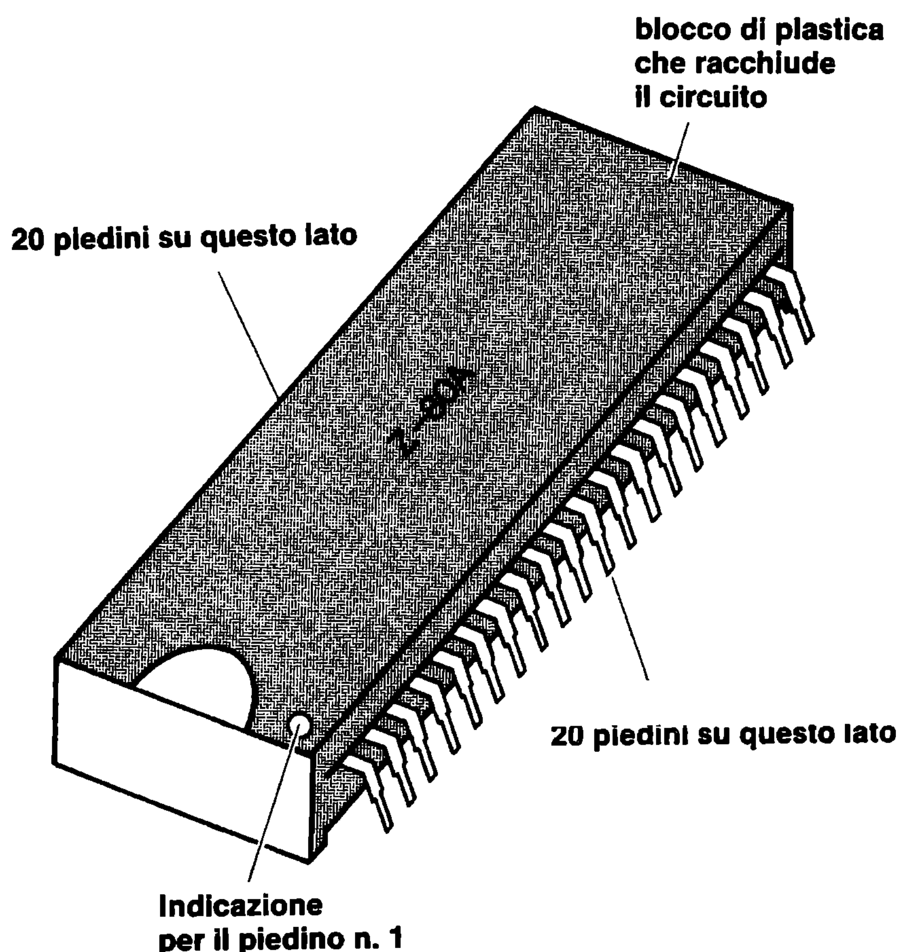


Fig. 1.8. L'MPU dello Z80. La parte effettivamente funzionante è più piccola di un'unghia, e l'involucro di plastica (52 mm per 14 mm di larghezza) rende più agevole lavorarci.

Che cosa fa la MPU? La risposta è: praticamente tutto, nonostante le azioni che la MPU può eseguire siano poche e molto semplici. La MPU può caricare un byte, cioé un byte che si trova nella memoria può essere copiato ed immagazzinato nella MPU. La MPU può anche immagazzinare un byte, cioé una copia di un byte immagazzinato nella MPU può essere posta ad un qualsiasi indirizzo della memoria. Queste due azioni (si veda la Fig. 1.9) sono quelle che la MPU esegue per la maggior parte del tempo. Combinando queste azioni, possiamo copiare un byte da un indirizzo della memoria ad un qualsiasi altro indirizzo. Non pensate che questo sia utile? Questa azione di copiatura è quella che avviene quando

premete la lettera H sulla tastiera e vedete comparire H sullo schermo. La MPU considera la tastiera come una parte di memoria e lo schermo come un'altra, per cui copia i bytes dall'una all'altro mentre voi premete i tasti. Ho semplificato molto, tuttavia basta questo per farvi capire quanto

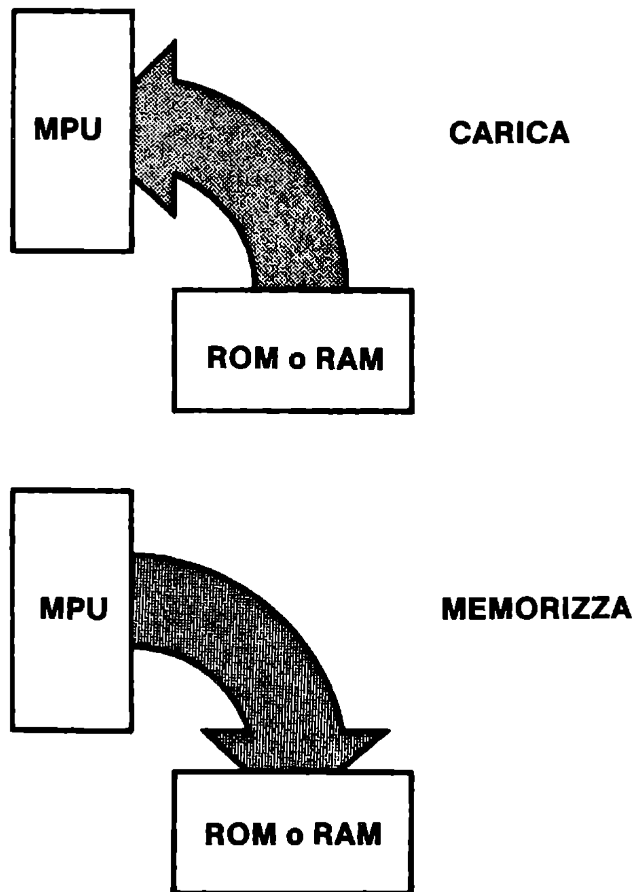


Fig. 1.9. Caricamento e memorizzazione. Caricare significa far segnali alla MPU dalla memoria. La memorizzazione è il procedimento opposto.

importante sia quest'azione. Se ci pensate bene, quando voi scrivete un programma BASIC, eseguite un'azione di copiatura. Battete una lettera sulla tastiera ed essa compare sullo schermo in virtù di quest'azione di copiatura. Viene poi immagazzinata nella memoria del computer sempre per un'azione di copiatura. Dopo che avete scritto il programma, potete registrarlo su nastro, facendo uso di un altro tipo di azione di copiatura. Tutte queste azioni utilizzano il caricamento e l'immagazzinamento e sono tutte eseguite dalla MPU. Perfino quando fate girare un programma BASIC, molte di queste azioni sono ancora delle azioni di copiatura.
Il caricamento e l'immagazzinamento sono due importantissime azioni

della MPU, ma ce ne sono delle altre ugualmente importanti. Un insieme di azioni è *l'insieme aritmetico.* Per quasi tutti i tipi di MPU, le operazioni aritmetiche sono costituite solo dall'addizione e dalla sottrazione. Le operazioni aritmetiche, nella maggior parte, possono usare solo numeri di un byte. Poiché un numero di un byte significa un numero compreso tra 0 e 255, come può fare il computer ad eseguire operazioni come moltiplicazioni di numeri più grandi, divisioni,innalzamenti a potenza, logaritmi, seni, etc.? Grazie ai programmi in codice macchina che si trovano nella ROM. Se questi programmi non si trovassero nella ROM, dovreste scriverli per conto vostro. Non ci sono molti utilizzatori di computer disposti a svolgere questo compito, eppure, per qualche strana ragione, molti libri sulla programmazione in codice macchina, si concentrano proprio su queste azioni.
C'é poi anche l'insieme logico. Azione logica significa prendere delle decisioni sulla base delle informazioni ricevute. Dal punto di vista del computer, questo significa confrontare due numeri al fine di generarne un terzo. La logica della MPU è, come tutte le azioni della MPU, semplice e soggetta a regole rigorose. Le azioni logiche confrontano i singoli bit di due bytes e danno una "risposta" che dipende dal valore dei bit confrontati e dalla regola logica che è stata seguita. Le tre regole logiche sono chiamate AND, OR e XOR e la Fig. 1.10 mostra come sono applicate.

AND

Il risultato di AND fra due bit sarà 1 se ambedue i bit sono 1, altrimenti:

1 AND 1=1 $\left\{ \begin{array}{l} 1 \text{ AND } 0=0 \\ 0 \text{ AND } 1=0 \end{array} \right\}$ 0 AND 0=0

AND applicato ai bit corrispondenti ai due bytes

```
     10110111
AND  00001111
     00000111
```

solo
questi bit
sono presenti in ambedue
i bytes.

OR

Il risultato di OR fra due bit sarà 1 se *uno* o *ambedue* i bit sono 1, 0 in caso contrario:

$$1 \text{ OR } 1=1 \left\{ \begin{array}{l} 1 \text{ OR } 1=0 \\ 0 \text{ OR } 1=1 \end{array} \right\} 0 \text{ OR } 0=0$$

OR applicato ai bit corrispondenti di due bytes.

```
     10110111
OR   00001111
     10111111
      ↑
   l'unico
   bit che è
   0 in
   ambedue i bytes.
```

XOR (OR-esclusivo)

Come OR, ma il risultato è zero se i bit sono identici

$$1 \text{ XOR } 1=0 \left\{ \begin{array}{l} 1 \text{ XOR } 0=1 \\ 0 \text{ XOR } 1=1 \end{array} \right\} 0 \text{ XOR } 0=0$$

```
     10110111
XOR  00001111
     10111000
```

se due bit
sono identici
il risultato
è zero

Fig. 1.10. Le regole delle tre importanti azioni logiche, AND, OR e XOR.

Quando si usa la regola logica AND, il "risultato" dell'operazione logica AND fra due bit sarà 1 solo se *ambedue* i bit sono 1. Se un solo bit è 0, il risultato sarà 0, e se ambedue i bit sono 0 il risultato sarà sempre 0. Come viene usata AND? Senz'altro in BASIC avrete programmato qualcosa tipo:

IF A$="END" AND B%=100 THEN 5000

per decidere quando qualcosa deve aver luogo in un programma. Il

BASIC usa i codici 1 e 0 per eseguire questi test, 11111111 significa vero e 00000000 falso. Ora, se A$="END", verrà immagazzinato il numero binario 11111111, e se B%=100, verrà immagazzinato un altro 11111111. L'operazione di AND su questi numeri darà un altro 11111111, che significa vero, e, come risultato, si avrà il salto alla riga 5000 del programma. Occorre sottolineare che il BASIC non esegue azioni logiche come AND, OR o XOR su termini tipo A$="END", ma solo su numeri rappresentanti "vero", 11111111, oppure "falso", 00000000
L'azione OR dà uno zero solo se tutti e due i bit confrontati sono 0. Solo 0 OR 0 dà 0, perché 1 OR 0 dà 1 e 0 OR 1 dà 1. Anche 1 OR 1 dà 1.
OR viene usata con i codici numerici di VERO o FALSO nello stesso modo di AND. Potreste avere un'istruzione BASIC di questo tipo:

IF A$ = "X" OR A$ = "x"

per cui, se una di queste due ipotesi è vera (11111111) e l'altra è falsa (00000000), allora il confronto darà 11111111, che significa vero. L'azione XOR (OR esclusivo) è simile a quella di OR, però quando ambedue i bit sono 1, il risultato è 0, non 1. I due punti principali da ricordarsi nell'applicazione di XOR sono i seguenti: (a) se viene eseguita l'operazione di XOR confrontando un numero binario con se stesso, il risultato sarà zero: (b) se l'operazione XOR viene eseguita confrontando un numero binario con un altro numero binario per due volte consecutive, si otterrà nuovamente il primo numero (Fig. 1.11). Questo può essere utile per scopi di codifica e decodifica. Se un numero è codificato, eseguendo l'operazione XOR con un numero "chiave", il risultato può essere usato come codice. Quando viene di nuovo eseguita l'operazione XOR fra questo codice ed il numero "chiave", si otterrà nuovamente il primo numero.

Numero Binario:	01101011	
XOR con:	11011000	Chiave
	10110011	Risultato
	↓	
	10110011	
XOR con:	11011000	Chiave
	01101011	Numero originale

Fig. 1.11 Se viene eseguita l'operazione XOR fra un numero ed un altro (la "chiave"), il risultato sarà un numero. Se viene effettuata l'operazione XOR fra questo numero e la chiave ancora una volta, si otterra nuovamente il numero originale.

Ricordate, quando parliamo di numeri, che questi sono la "materia prima" del computer.

Usando i codici ASCII, possiamo codificare qualsiasi messaggio sotto forma di un numero o di un insieme di numeri, per cui questi metodi si applicano sia alle lettere dell'alfabeto che ai numeri stessi.
Un altro insieme di azioni è chiamato *l'insieme dei salti.* Un salto significa un cambiamento d'indirizzo, è assai simile all'azione di GOTO in BASIC.
Una combinazione di un test e di un salto è il modo in cui la MPU attua le sue decisioni. In BASIC potete scrivere:

IF A=36 THEN GOTO 1050

Nella stessa maniera si può far eseguire alla MPU un'istruzione che si trova non all'indirizzo successivo, ma ad uno totalmente diverso. La MPU è uno strumento programmato, cioé esegue ogni azione come conseguenza dell'aver ricevuto un byte di istruzioni che era stato precedentemente immagazzinato nella memoria. Normalmente, quando alla MPU viene trasmessa un'istruzione proveniente da un certo indirizzo (di solito della ROM), la MPU esegue l'istruzione e poi legge il byte di istruzioni che si trova all'indirizzo successivo.
Anche il BASIC esegue le istruzioni contenute in una riga e poi si sposta alla riga che viene subito dopo. Un'istruzione di salto impedirà che questo avvenga, e farà sì che la MPU legga invece da un altro indirizzo, quello specificato nell'istruzione di salto. Questa azione di salto può essere condizionata dall'esito di un test.
Il test viene di solito effettuato sul risultato dell'azione precedente: ad esempio, se il risultato era positivo, negativo o zero.
Questo elenco non è né lungo né entusiasmante, ma le azioni di cui non ho parlato non sono importanti a questo stadio, oppure non sono particolarmente diverse da quelle elencate. Ciò che voglio sottolineare è che il magico microprocessore non è poi uno strumento così intelligente. Ciò che lo rende vitale per il computer è che può essere programmato e che può eseguire le varie azioni molto rapidamente. Ugualmente importante è il fatto che il microprocessore può essere programmato attraverso l'invio di segnali elettrici.
Questi segnali sono inviati ad otto "piedini" (pin), chiamati piedini dei dati, della MPU. Non ci vuole molto ad indovinare che questi otto piedini corrispondono agli otto bit di un byte. Ogni byte della memoria può quindi modificare la MPU dividendo con essa i suoi segnali elettrici.
Chiamiamo questo procedimento "lettura". La lettura implica che un byte della memoria sia collegato alla MPU lungo otto linee, in modo che ogni bit 1 dia un segnale 1 ad uno dei piedini, ed ogni bit 0 dia un segnale 0.
Proprio come la lettura di un giornale o l'ascolto di un disco non distrugge ciò che è stato scritto o registrato, la lettura di una memoria non modifica assolutamente la memoria stessa e non viene cancellato niente. Il procedimento opposto della scrittura, invece, cambia la memoria, nello stesso modo in cui, registrando un nastro, si cancella ciò che vi era prima

inciso. Quando la MPU scrive un byte ad un indirizzo della memoria, quello che si trovava memorizzato a quell'indirizzo non esiste più, ma viene rimpiazzato dal nuovo byte. Ecco perché è così facile scrivere nuove righe in BASIC che sostituiscano quelle vecchie, mantenendo lo stesso numero di riga.

LA SEQUENZA OPERATIVA

Scrivete veramente dei programmi in BASIC?
Potrebbe sembrare una domanda sciocca, ma vi assicuro che è una domanda seria. L'effettivo lavoro del programma è svolto attraverso istruzioni codificate dirette alla MPU, e se scrivete solo in BASIC, non scrivete queste istruzioni. Tutto ciò che fate è scegliere da un menu quelle che chiamiamo parole chiave del BASIC, e sistemarle in modo tale da generare i risultati corretti. La vostra scelta è limitata alle parole chiave che si trovano nella ROM. Non possiamo cambiare la ROM, e se vogliamo eseguire un'azione non prevista da una parola chiave, dobbiamo combinare un certo numero di parole chiave (un programma BASIC) oppure operare direttamente sulla MPU con codici numerici (codice macchina).
Quando dovete eseguire delle azioni attraverso la combinazione di una serie di comandi BASIC, il risultato è lento, specialmente se ogni comando è un insieme di altri comandi. L'azione diretta è veloce, ma può essere difficile. L'azione diretta di cui parlo é il codice macchina e gran parte di questo libro sarà dedicata a capire questo "linguaggio", che è difficile proprio perché è semplice! Prendiamo una situazione che illustrerà questo paradosso. Supponiamo che vogliate far costruire un muro. Potete rivolgervi ad un costruttore, basterà dirgli che volete un muro attraverso il giardino sul retro della casa e poi non dovrete far altro che stare ad aspettare. E' quanto avviene quando voi usate il BASIC: voi date solo il comando "costruisci il muro". C'é molto lavoro da fare, ma voi non dovete preoccuparvi dei dettagli.
Ora pensate ad un'altra possibilità: supponete di avere un robot che possa eseguire le vostre istruzioni, senza fare alcun ragionamento, ma in modo estremamente rapido. Non potete dirgli "costruisci un muro", perché questa istruzione va oltre la sua capacità di comprensione. Dovete dirglielo con tutti i dettagli, tipo: "traccia una linea da un punto distante 10 metri dal muro della cucina della casa, misurato lungo la siepe a sud, ad un punto distante 12 metri dal muro del salotto della casa, misurato lungo quella siepe a sud. Scava una buca profonda 35 centimetri e larga 30 centimetri lungo il percorso della tua linea. Mischia tre sacchi di sabbia e due di cemento con quattro carriole di sassi per tre minuti. Unisci dell'acqua e mescola finché un secchio riempito con questo impasto non impiegherà 10 secondi a vuotarsi se tenuto capovolto. Riempi la buca con

questo impasto..." Le istruzioni sono molto dettagliate — devono essere dirette ad un robot senza cervello — ma saranno eseguite perfettamente e velocemente. Se avete dimenticato una cosa, non importa quanto sia ovvia, non sarà eseguita. Dimenticate di specificare dove si trovino il cemento, la sabbia e l'acqua, quanta calcina, quale impasto fare e dove deve essere posto, e i vostri mattoni saranno messi su senza calcina. Dimenticate di specificare l'altezza del muro, e il robot continuerà a mettere un mattone sopra l'altro come l'Apprendista Stregone, finché qualcuno non starnutirà, facendo crollare il muro.

Il parallelo con la programmazione è assai ovvio. Una parola chiave del BASIC è come l'istruzione "costruisci il muro" data al costruttore. Darà origine a parecchio lavoro, basandosi su istruzioni che non sono vostre —ma non potrà essere eseguita così velocemente come vorreste. Potrebbe anche essere eseguita in un modo diverso da quello che desideravate.
Se vi prendete la briga di specificare i dettagli, il codice macchina è molto più veloce, perché darete le vostre istruzioni direttamente ad una macchina, incredibilmente veloce ma priva d'intelligenza: il microprocessore.

Possiamo estendere questo paragone. Se voi dite al vostro costruttore "ripara l'automobile", potrebbe non volerlo fare o non esserne capace. Se invece date al robot un insieme corretto di istruzioni dettagliate, il robot eseguirà questo lavoro. Il codice macchina può essere usato per far eseguire al vostro computer istruzioni non contemplate dal BASIC. Bisogna dire, ad onor del vero, che quasi tutti i moderni computer prevedono una gamma di istruzioni molto più ampia di quella prevista dai primi modelli, per cui, sotto tale aspetto, il codice macchina non è oggi così importante come in passato.
Occorre dare un'ultima occhiata al diagramma a blocchi (Fig. 1.7) prima di cominciare ad esaminare il funzionamento interno del computer MSX. Il blocco contrassegnato da "PORTA" include più di un "chip". Una porta, nella terminologia del computer, indica qualcosa che viene usato per passare le informazioni, un byte per volta, da o al resto del sistema —la MPU, la ROM o la RAM.

C'é una parte separata per gestire questi "input" ed "output", poiché sono azioni importanti, ma lente. Se usiamo una porta, possiamo far sì che sia il microprocessore a decidere quando vuole leggere un input o scrivere un output. Inoltre, possiamo isolare gli input e gli output dalla normale azione dalla MPU. Ecco perché non appare niente sullo schermo in un programma BASIC se non vi è il comando PRINT nel programma stesso.

Ecco perché se premete il tasto PLAY del registratore non vi sarà alcun effetto finché non battete CLOAD e premete RETURN. La porta vi nasconde l'azione del computer finché non volete effettivamente un input o un output.

L'AZIONE POKE

I programmi che abbiamo finora esaminato hanno letto la memoria del computer, usando la funzione PEEK. C'è un'altra istruzione BASIC, POKE, che esegue l'azione opposta. POKE scriverà un byte nella memoria, voi dovete specificare il byte e l'indirizzo di memoria. In BASIC, la funzione POKE usa numeri decimali, a meno che non sia altrimenti specificato, per cui l'indirizzo dovrà essere compreso tra 32768 e 65535, ed il numero del byte tra 0 e 255. Una tipica istruzione POKE potrebbe avere questo formato: POKE 32770,5, dove il numero d'indirizzo viene scritto subito dopo POKE, poi viene immessa una virgola ed infine il numero del byte. Il numero 5 verrà così immagazzinato all'indirizzo 32770. Potreste usare l'istruzione POKE con indirizzi inferiori a 32768, ma dal momento che questi sono (normalmente) indirizzi della ROM, l'azione di POKE non avrà effetto. Non potete modificare il contenuto di un indirizzo della ROM!

POKE è un'istruzione che può causarvi qualche guaio se non sapete che cosa state facendo. Potete utilizzare la funzione PEEK quanto volete, perché PEEK copia semplicemente quanto è contenuto nella memoria del computer, senza modificare niente. POKE, invece, può sostituire un byte con un altro. Se l'indirizzo che avete indicato con la funzione POKE contiene qualcosa di essenziale per il funzionamento del computer, il risultato dell'istruzione POKE sarà di far "impazzire" la macchina!

Per esempio, potreste veder comparire disegni strani sullo schermo, oppure la tastiera potrebbe essere bloccata, per cui la pressione dei tasti rimarrebbe senza effetto. Quando accade un cosa del genere. premendo CTRL—STOP *è probabile* che si ritorni alla normalità, ma molto spesso si dovrà spegnere e poi riaccendere il computer. Ma in questa maniera perderete ogni programma che era in memoria. Quando lavorate con il codice macchina, che pone *sempre* i bytes direttamente in memoria, o quando usate l'istruzione POKE dal BASIC, *dovete* registrare ogni programma prima di provarlo. Altrimenti, in caso di fallimento del programma, potreste perderlo e dovreste riscriverlo di nuovo.

Ora, siete avvertiti!

C'é un tipo di funzione POKE che potete usare con molto minor rischio. Essa usa la parola VPOKE ed è la V che la rende particolare. Effetto della "V" è di selezionare una parte speciale della memoria quella parte riservata allo schermo. I numeri di indirizzo usati per la funzione VPOKE variano da 0 a 16383, a seconda del MODO dello schermo. Nel modo di scrittura (SCREEN 0) potete usare la gamma ordinaria dei numeri di codice ASCII. Per esempio, provate il programma della Fig. 1.12.

Con la funzione VPOKE il numero 65 (il codice ASCII per la lettera "A") viene posto agli indirizzi tra 0 e 1024 compresi. Questi sono i numeri di indirizzo usati dallo schermo in modo di scrittura (SCREEN 0), e l'effetto

```
10 FOR N%=0 TO 1024
20 VPOKEN%,65
30 NEXT
```

Fig. 1.12 L'effetto di VPOKE è di porre i caratteri direttamente sullo schermo.

del programma è di porre una "A" ad ogni punto dello schermo.
Vi prego di notare che questa è una parte separata di memoria.
Se usate l'istruzione POKE con gli stessi numeri, ricadreste invece negli indirizzi della ROM, non nella RAM video. Lo schermo del computer MSX usa una parte separata di memoria e per questo dovete usare un diverso tipo d'istruzione POKE per accedere ad essa. Ora apportiamo alcune modifiche al programma. Cambiate così la prima riga:

```
10 FOR N%=6144 TO 6911
```

e date il comando RUN. Questa volta non apparirà niente. Infatti, questa gamma di indirizzi è riservata allo schermo in modo di scrittura e grafico, SCREEN1. Provate a battere SCREEN1 e *poi* fate girare il programma. Vedrete di nuovo apparire l'insieme delle "A". La memoria del video è divisa in diverse parti, corrispondenti ai diversi schermi. Provate il programma della Fig. 1.13 per vedere gli effetti della funzione VPOKE su uno schermo ad alta risoluzione.

```
 5 SCREEN2
10 FOR N%=8192 TO 16383
20 VPOKEN%,10
30 NEXT
40 GOTO 40
```

Fig. 1.13 VPOKE può essere usata anche per controllare lo schermo ad alta risoluzione, come è mostrato in questa figura.

In questo caso la funzione è assai diversa. Gli indirizzi indicati da VPOKE contengono i numeri dei colori, ed ogni indirizzo rappresenta un punto sullo schermo. L'effetto di VPOKEN%,10 è di immettere il numero del colore 10 (giallo) in ogni parte della memoria video per questo tipo di schermo (SCREEN2). Il risultato, allora, sarà di colorare di giallo tutto lo schermo. Se voi indicate dei passi dopo i numeri di indirizzo,saranno visualizzate delle strisce gialle. Provate ad aggiungere STEP8 o STEP4 alla riga 10, per esempio, come modo rapido di tracciare delle linee sullo schermo! Potete esercitarvi quanto volete con VPOKE, perché non interferisce con la normale attività del computer. Usando CTRL L (per il modo di scrittura) o CTRL STOP (per il modo ad alta risoluzione) potete sempre ritornare alle normali azioni del programma. Ed è molto più sicuro della ordinaria funzione POKE!
Abbiamo così visto tutte le parti importanti del vostro computer MSX.

Ho usato alcuni termini liberamente — i puristi faranno delle obiezioni al modo in cui ho usato la parola “porta”, ad esempio — ma nessuno troverà da ridire sulle azioni del computer. Ciò che ora dobbiamo fare è vedere come sia organizzato il computer nell'uso della MPU, ROM, RAM e delle porte al fine di essere programmato in BASIC e di poter eseguire un programma BASIC. Sembra un buon punto per iniziare un nuovo capitolo!

CAPITOLO 2

Frugando dentro L'MSX

Non lo intendo in senso letterale, non dovete aprire il vostro computer! Ciò che intendo dire è che ora esamineremo come sia stato progettato il computer MSX per caricare ed eseguire programmi BASIC. Una volta che sapete come avviene questo, dovreste essere in grado di vedere come viene usato il codice macchina, e questo sarà di notevole aiuto in seguito, quando cominceremo ad esaminare l'uso delle routines ROM. Inizieremo con una versione semplificata dell'azione dell'intero sistema, omettendo i dettagli, per il momento.

Nella ROM del vostro computer MSX, che è compresa fra gli indirizzi da 0 a 32767, si trovano un gran numero di brevi programmi — subroutines — scritti in linguaggio macchina, e di insiemi (tabelle) di valori come la tabella delle parole chiave che avete già visto. Ci sarà almeno una subroutine in codice macchina per ogni parola chiave del BASIC, ed alcune di queste parole chiave possono richiedere l'uso di molte subroutines in sequenza. Quando accendete il vostro computer MSX, il primo programma in codice macchina che viene eseguito è chiamato "routine di inizializzazione".

È un programma abbastanza lungo, ma poiché il codice macchina è veloce (può eseguire istruzioni alla velocità di molte migliaia di istruzioni al secondo), praticamente non vi accorgerete di tutta questa attività: noterete soltanto un breve ritardo fra il momento in cui accendete il computer ed il momento in cui compare la dicitura del copyright MICROSOFT, sostituita poi dal messaggio BASIC e da "Ok".

Tuttavia in questo breve arco di tempo, è stata controllata la memoria RAM, una sezione della RAM è stata "scritta" con i bytes che verranno usati in seguito (come le parole dei tasti funzione), mentre la maggior parte della RAM è stata inizializzata per l'uso. Questo non significa che nella RAM non si trovi contenuto niente. Quando spegnete il computer, la RAM perde ogni traccia dei segnali immagazzinati, ma quando accendete nuovamente il computer le celle della memoria non rimangono tutte poste a 0.

In ogni byte, alcuni dei bit saranno posti ad 1 ed altri a 0, quando viene di nuovo fornita la corrente. Questo però avviene a caso, per cui,se poteste esaminare ciò che è immagazzinato in ogni byte subito dopo aver acceso il computer, trovereste un insieme di numeri senza alcun significato, sempre compresi tra 0 e 255, la gamma normale di numeri per un byte di memoria. Questi numeri sono "spazzatura", non sono stati immessi in memoria volutamente, e non rappresentano neppure dei dati utili o delle istruzioni.

Il primo lavoro del computer, quindi, è di far pulizia.
Al posto dei numeri casuali, il computer sostituisce uno schema molto più ordinato di 64 bytes contenenti il numero 255, seguiti da 64 bytes contenenti il numero 0. Provate ad accendere il computer e a scrivere, senza numero di riga:

```
FOR N=32768 TO 33000:?PEEK (N) ; : NEXT
```

e premete RETURN. Gli indirizzi che abbiamo usato sono la gamma di indirizzi dell'"inizio del BASIC", cioè dove vengono normalmente memorizzati i primi bytes di un programma BASIC. Se avete appena acceso il computer e non avete dato un numero di riga all'istruzione, non ci sarà niente immagazzinato a quegli indirizzi, soltanto lo schema lasciato dopo l'inizializzazione. Come vedrete sullo schermo, consiste di dieci bytes, che sono il risultato dell'istruzione, seguiti dalla catena di 255 e di 0.
Il programma di inizializzazione ha ancora molto da fare. La parte alta della RAM, dagli indirizzi 62337 a 65535, è "riservata al sistema". Questo perché le subroutines in codice macchina che eseguono le azioni del BASIC hanno bisogno di immagazzinare dati in memoria mentre lavorano. Vedremo fra un momento come sono usati alcuni di questi numeri di indirizzo. Una sezione separata e molto più ampia della memoria viene usata per immagazzinare numeri che fanno apparire sullo schermo le lettere e i grafici. Inoltre, una parte della RAM deve essere usata anche per contenere i dati creati quando un programma viene eseguito. È proprio questo che ora esamineremo.

VARIABILI SULLA TABELLA

I programmi BASIC fanno un grande uso delle variabili, cioé di lettere che rappresentano numeri e parole. Ogni volta che voi "definite una variabile" usando una riga di programma tipo:

```
N =20 oppure A$="SMITH"
```

il computer deve riservarsi uno spazio di memoria con il nome (N o A$ o qualsiasi altro nome) ed il valore (20 o SMITH) che voi avete assegnato alla variabile. La parte di memoria usata per poter rintracciare le variabili è chiamata "tabella della lista di variabili" (VLT). Non occupa un posto fisso nella memoria, ma viene immagazzinata nello spazio libero proprio sopra il vostro programma. Se aggiungete una o più righe al vostro programma, l'indirizzo della VLT dovrà essere spostato verso numeri di indirizzo più alti. Se cancellate una riga dal vostro programma, la VLT sarà spostata verso il basso, in modo che segua l'ultima riga del BASIC.
Ora, dal momento che l'indirizzo della VLT può e deve spostarsi quando il programma viene modificato, il computer, tutte le volte, deve tener nota

dell'inizio della tabella. Questo è fatto usando una delle parti della memoria riservata al sistema, gli indirizzi 63170 e 63171. Potreste chiedervi perché siano stati usati due indirizzi.
La ragione sta nel fatto che ogni byte può contenere un numero non maggiore di 255. Se usiamo due bytes, possiamo mettere il numero corrispondente al *multiplo di 256* in un byte e il resto nell'altro byte. Un numero come 257, per esempio, è un 256 ed un resto.
Potremmo codificarlo come 1,1. Questo significa che un 1 è contenuto nel byte riservato ai 256, ed 1 nel byte riservato alle unità. *L'ordine di immagazzinamento dei numeri è byte-basso poi byte-alto.* Per trovare il numero che è immagazzinato, moltiplichiamo il numero del byte-alto per 256 e poi aggiungiamo il numero del byte-basso. Per esempio, se trovaste 23,128 immagazzinato in due indirizzi consecutivi, che siano stati usati in questa maniera, il numero sarebbe:

$$128 \times 256 + 23 = 32791$$

Il numero più grande che possiamo immagazzinare usando due bytes è 255,255, cioè 255×256+255=65535. Ecco perché non potete usare numeri molto grandi, tipo 70000, come numeri di riga con il computer MSX, il sistema operativo usa solo due bytes per immagazzinare i suoi numeri di riga. In effetti, per altre ragioni, il massimo numero che voi potete usare è 65529.
Tutto questo vuol dire che possiamo trovare l'indirizzo immagazzinato agli indirizzi 63170 e 63171 usando la formula:

```
?PEEK(63170)+256*PEEK(63171)
```

Se usate questa istruzione subito dopo aver acceso il vostro computer MSX, il risultato per il computer MSX da 64K sarà il numero d'indirizzo 31771. Infatti, è proprio al di sopra dell'indirizzo a cui viene immagazzinato il primo byte di un programma BASIC. Per vederne la dimostrazione pratica, battete la riga:

```
10 N=20
```

e provate di nuovo ?PEEK(63170)+256*PEEK(63171).
Se avete battuto questa riga come nell'esempio, cioè con uno spazio fra il numero di riga e la "N", allora avrete l'indirizzo 32780. La tabella della lista di variabili si è spostata verso l'alto nella memoria, esattamente di 9 bytes. È più del numero di bytes che avete battuto, come potete notare, ma ne vedremo il perché in seguito. Il punto principale, comunque, è che questo numero di indirizzo è cambiato per permettere l'inserimento di una riga di programma.
Una gran parte degli indirizzi importanti usati dal computer sono "allocati in modo dinamico", come quelli della VLT. "Allocati in modo dinamico" significa che il computer cambierà automaticamente l'indirizzo a cui

devono essere memorizzati i gruppi di bytes. Poi annoterà dove sono stati memorizzati modificando un indirizzo contenuto in una coppia di bytes, come nell'esempio precedente. È importante sapere tutto ciò, per un corretto uso del vostro computer. Ad esempio, se voi fate "slittare" la VLT, immettendo dei nuovi numeri con la funzione POKE agli indirizzi 63170 e 63171, il computer non può trovare i valori della variabile. Provate a far questo: dopo aver trovato l'indirizzo della VLT, ma senza far girare il nostro programma di una sola riga, 10 N=20, battete ?N. La risposta sarà zero. Perché? Perché il programma non è stato eseguito. L'indirizzo 32780 è l'indirizzo di inizio della *VLT, ma non si crea alcuna VLT finché il programma non viene eseguito.* Questo vi permette di aggiungere o cancellare facilmente delle righe a questo stadio.
Tutto ciò che dovrà essere cambiato è la coppia di numeri contenuti negli indirizzi 63170 e 63171. I valori della VLT sono messi a posto soltanto quando il programma viene eseguito e la tabella stessa non viene effettivamente spostata ma solo creata di nuovo. Tutto quello che cambia quando richiamate una riga di programma, è l'indirizzo di inizio contenuto in 63170 e 63171. Ecco perché non potete ritornare all'esecuzione del programma dopo aver richiamato e modificato la riga, dovete di nuovo dare il comando RUN per creare una nuova VLT ad un nuovo indirizzo. Se fate girare il nostro programma di una sola riga ora, e poi battete ?N, avrete la risposta corretta 20. Ora battete (senza numero di riga) POKE 63171,129 e premete RETURN. Provate ?N e guardate cosa ottenete. Il mio computer MSX si è bloccato quando ho provato a far questo, ed è probabile che succeda lo stesso con il vostro. Non è servito a niente premere CTRL/STOP e questa è una dimostrazione efficace del perché bisogna registrare ogni programma su cui state lavorando *prima* di usare la funzione POKE. Perché il computer si è bloccato? Perché l'istruzione ?N richiedeva alla macchina di trovare il valore di N e questo valore doveva essere cercato nella VLT.
Ora, dal momento che l'effetto di POKE era stato di far slittare la VLT, il valore non poteva essere ottenuto. Il computer, in ogni caso, doveva continuare a cercare questo valore ed ecco perché non poteva eseguire i vostri comandi, immessi dalla tastiera.

UNO SGUARDO ALLA TABELLA

È arrivato il momento di far qualcosa di più costruttivo, e di dare un'occhiata a quello che si trova memorizzato nella VLT. Quando facciamo queste ricerche, è importante assicurarsi che il computer non contenga i dati di un lavoro precedente, per cui è consigliabile spegnerlo e poi riaccenderlo, prima di intraprendere ogni nuovo tentativo. Se premete semplicemente i tasti CTRL/SHIFT, non verranno modificati i valori che voi avete eventualmente immesso in memoria con la funzione POKE. È noio-

so lo so, ma questo è il codice di macchina!
Al lavoro, dunque! Dopo aver spento e riacceso il computer, battete di nuovo la riga:

```
10 N=20
```

e trovate l'indirizzo della VLT usando

```
?PEEK(63170)+256*PEEK(63171).
```

Si dovrebbe ottenere ancora l'indirizzo 32780. Ora date il comando RUN, in modo che i valori possano essere immessi nella VLT e guardate che cosa vi si trova memorizzato, con l'istruzione:

```
FORX=32780 TO 32790:?X;"    ";PEEK(X) :NEXT
```

seguita da RETURN. Questa istruzione dà il listato della Fig. 2.1

```
32780  8
32781  78
32782  0
32783  66
32784  32
32785  0
32786  0
32787  0
32788  0
32789  0
32790  0
```

Fig. 2.1. Valori assunti dalla tabella della lista di variabili per una variabile numerica.

Riconoscete qualcosa in quel listato? Dovreste riconoscere il secondo byte di 78, perché è il codice ASCII di N! Il byte successivo è 0, perché abbiamo chiamato N la nostra variabile, non N1 o NG o qualsiasi altro nome di due lettere. Se usassimo un nome di due lettere, allora sarebbero stati occupati sia l'indirizzo 32781 che l'indirizzo 32782. Il successivo insieme di bytes, allora, deve essere il modo in cui è stato codificato il numero 20. A questo punto, non preoccupatevi di come siano usati questi numeri per rappresentare il 20, accettateli così come sono! Come faccio a sapere che sono gli otto bytes successivi a rappresentare il numero 20? Facile, perché il primo byte di questa sequenza è 8, ed indica il numero di bytes che saranno usati per codificare il numero! La codifica usata per i dati immessi nella VLT è molto logica ed anche assai semplice da capire, una volta che ne conoscete il meccanismo. Se continuate a leggere la tabella, troverete che il byte dell'indirizzo 32791 è un altro "8", ed è seguito dal codice ASCII per X, cioé 88. Perché mai appare il codice ASCII per X? Semplicemente perché abbiamo usato X come variabile per

stampare i valori della VLT! Il computer MSX usa sempre otto bytes per ogni valore di una variabile numerica a doppia precisione, non importa se questa variabile è un numero piccolo come 20, od uno molto più grande come 1463170068315, o una frazione o un numero negativo. Questo procedimento rende semplice l'immagazzinamento delle variabili numeriche e facilita anche al computer l'individuazione delle variabili. Ogni lettera usata per una variabile numerica, ricordatevelo, sarà usata per numeri a doppia precisione, a meno che non sia contrassegnata dal simbolo ! o %. Se, ad esempio, il computer deve cercare il valore di una variabile "Y", quando troverà "N" (codificata in ASCII come 78) non dovrà perdere tempo con i nove bytes successivi (uno per la seconda lettera, otto per il valore), ma si sposterà più avanti, al primo indirizzo in cui troverà immagazzinato un altro nome di variabile.
Se siete curiosi ed avete predisposizione per la matematica, l'Appendice B mostra il metodo di codifica usato per convertire i numeri in otto bytes per le variabili a doppia precisione, ed in quattro bytes per le variabili a precisione singola. Per gli scopi di questo libro, comunque, non è necessario sapere come viene effettuata la codifica, è sufficiente sapere come viene memorizzato il codice e quanti bytes sono richiesti.

COLLEGAMENTO DI UNA STRINGA

Ora dobbiamo vedere come viene immagazzinata una variabile di stringa. Anche questa volta, spegnete e riaccendete il computer, e poi scrivete questa riga:

```
10 AB$="QUESTA È UNA STRINGA"
```

Date il comando RUN e poi trovate l'indirizzo della VLT usando gli indirizzi 63170 e 63171, come nell'esempio precedente. Dovreste ottenere 32803 come numero d'indirizzo. Ora usate questa istruzione:

```
FOR X=32803 TO 32813:?X;"   ";PEEK(X) :NEXT
```

seguita da RETURN per trovare ciò che è contenuto nella VLT. Questo listato compare nella Fig. 2.2.

Il primo valore di questa tabella è 3, che ci dice che vi sono solo tre bytes disponibili per il valore. Dice anche al computer che questa è una variabile di stringa, non un numero, perché le variabili numeriche usano come bytes di "riconoscimento" solamente 2, 4 o 8.
Il byte successivo è 65, cioè il codice ASCII per B. Se aveste usato A$, allora il secondo numero (all'indirizzo 32805) sarebbe stato 0. Ciò che dobbiamo prendere in considerazione, ora, sono i tre bytes che seguono il nome, perché questi dovrebbero essere i tre bytes usati per codificare la stringa.
È abbastanza difficile, però, che questi tre bytes siano il codice ASCII per

32803	3
32804	65
32805	66
32806	21
32807	10
32808	128
32809	8
32810	88
32811	0
32812	69
32813	50

Fig. 2.2. I valori della VLT per una variabile di stringa.

le lettere. Se ragioniamo un attimo su questi numeri, troveremo la soluzione. Il numero che segue il codice B è 21. 21 è il numero dei caratteri della nostra stringa. Contate il numero delle lettere e degli spazi e avrete appunto questo risultato. Il byte successivo è 10, seguito da 128. Ora è molto probabile che due bytes insieme costituiscano un indirizzo, e se li combiniamo nel modo consueto, usando 10+256*128, troviamo 32778. Il prossimo passo da compiere è ?PEEK(32778). Il valore trovato è 81, il codice ASCII per la Q. 32778 è l'indirizzo del primo byte della nostra stringa.

Rifacciamo da capo tutto il ragionamento. Il computer MSX prevede un inserimento di sei bytes nella sua VLT per ogni stringa. Di questi sei bytes, il primo è il "3", che specifica quanti bytes occorrono per la codifica della lunghezza della stringa e del suo indirizzo. Gli altri due bytes sono riservati al nome della stringa ed usano direttamente i codici ASCII. Quando viene usato un nome di una sola lettera, il secondo byte è zero. I tre bytes che seguono contengono la lunghezza della stringa e l'indirizzo di memoria del suo primo byte. Basta solo un byte per la lunghezza perché nessuna stringa può oltrepassare i 255 caratteri. Due bytes occorrono per l'indirizzo, che è immagazzinato sempre nell'ordine basso-poi-alto.

In questo esempio, la stringa è memorizzata ad un indirizzo inferiore a quello della VLT, nella parte di memoria della "scrittura BASIC". Questa è la parte di memoria che contiene il programma, e poiché i codici ASCII per la stringa sono posti qui quando scrivete il programma, va benissimo anche questa locazione di memoria per la stringa.

I numeri devono essere trasferiti alla VLT perché non sono memorizzati come codici ASCII. Ecco perché potete eseguire delle operazioni di stringa su una variabile numerica, o delle operazioni numeriche su una variabile di stringa. Ma che cosa accade quando viene creata una stringa che non esiste nel programma? Se, ad esempio, battete:

```
10 A$="AB":B$="CD":C$=A$+B$
```

e date il comando RUN, troverete che la VLT è più lunga, come era logico supporre. Questa volta dovete esaminare gli indirizzi di memoria da 32800 a 32817. Troverete che i dati per A$ e B$ danno degli indirizzi posti entro la regione di memoria del programma,come vi aspettavate.
Il listato è mostrato nella Fig. 2.3.

```
32800  3
32801  65
32802  0
32803  2
32804  9
32805  128
32806  3
32807  66
32808  0
32809  2
32810  17
32811  128
32812  3
32813  67
32814  0
32815  4
32816  101
32817  241
```

Fig. 2.3. I dati della VLT per una stringa *non* memorizzata nella parte di memoria riservata al programma.

La variabile C$, però, dà i bytes 101 e 241 per il suo indirizzo. Esso corrisponde all'indirizzo 61797 (dato da 101+256*241) per questa stringa. Vediamo che cosa contengono gli indirizzi compresi tra 61797 e 61800, con l'istruzione:

```
FORX=61797 TO 61800:?X;"";PEEK(X);""CHR$(PEEK(X)):NEXT
```

I codici ASCII per le lettere ABCD si trovano memorizzati qui e l'uso di CHR$ nel programma visualizza la lettera corrispondente ad ogni numero di codice. Come certamente saprete, se avete un programma BASIC che usa questa parte della memoria e richiede più di 200 bytes, il programma, ad un certo punto, si arresterà, e comparirà il messaggio di errore "Out of string space". Potete riservare un'area di stringa più ampia usando il comando CLEAR.
Per esempio, CLEAR 1000 riserverà mille bytes per le stringhe in questa parte della memoria. Qui verranno immagazzinate solo le stringhe che non sono state definite nel programma. Tuttavia, a meno che non vogliate

formare delle stringhe attraverso la concatenazione, o comandi come LEFT$, MID$ e RIGHT$, o conversioni S$=STR$(V), quest'area di immagazzinamento non verrà utilizzata. Per i programmi che non usano queste operazioni, potete creare una maggior area di memoria eliminando questo spazio con il comando CLEAR 0.

VERSO IL LINGUAGGIO MACCHINA

Il vostro computer MSX ha un comando BASIC che permette di ottenere gli indirizzi direttamente dalla VLT. Il comando è VARPTR, e deve essere seguito dal nome della variabile posto fra parentesi. Se battete ?VARPTR(A$) e poi premete RETURN, vedrete comparire -32733 sullo schermo. Questo NON è l'indirizzo della variabile, è l'indirizzo del suo dato nella VLT. VARPTR dà l'indirizzo a cui è immagazzinata la lunghezza della stringa, ignorando i bytes del nome. Il segno negativo può sembrare assai strano, ma se battete ?PEEK(-32733) avrete 2 come risposta, cioè la lunghezza di A$.
La ragione di questo segno negativo sta nel modo in cui questi numeri di indirizzo vengono memorizzati. Parleremo di questo più avanti, ma se vi sentite più a vostro agio con gli effettivi numeri di indirizzo, allora aggiungete 65536 al numero che avete ottenuto. Nel nostro esempio, —32733+65536 dà 32803, cioè il modo normale di scrivere il numero d'indirizzo. Per trovare la dimensione del byte di lunghezza, dovete usare ?PEEK(VARPTR(A$)). Se volete l'indirizzo, dovete prendere il secondo e il terzo byte che seguono il byte di lunghezza. Per esempio, potete usare:

```
?PEEK(VARPTR(A$)+2)*256+PEEK(VARPTR(A$)+3)
```

che darà 32777 come indirizzo della stringa. Questo ci permette di effettuare degli utili stratagemmi in un programma BASIC, come lo scambio dei nomi delle variabili di stringa, ma, dal momento che esiste l'apposita istruzione SWAP, l'istruzione VARPTR è usata soprattutto per trasferire dati a programmi in codice macchina, ma è meglio non usarla all'inizio della vostra carriera come programmatori in linguaggio macchina! Per il momento, accontentatevi di provare quanto segue. Battete questa istruzione sotto forma di comando unico, senza numero di riga, controllatela attentamente e poi premete RETURN.

```
Q=VARPTR(A$):P=VARPTR(B$):X=PEEK(Q):Y=PEEK
(Q+1):Z=PEEK(Q+2):X1=PEEK(P):Y1=PEEK(P+1)
:Z1=PEEK(P+2):POKEQ,X1:POKEQ+1,Y1:POKEQ+2,Z1:
POKEP,X:POKEP+1,Y:POKEP+2,Z
```

Dopo aver fatto girare il programma, battete ?A$,B$ e vedrete che i valori sono stati scambiati. A$ ora è "CD" e B$ è "AB". Questo è avvenuto perché abbiamo scambiato i valori nella VLT. I valori letti da VARPTR(A$) sono stati immessi nelle locazioni della VLT riservate a B$, e i valori letti da VARPTR(B$) sono stati immessi negli spazi riservati ad A$. Ecco che cosa c'è dietro il semplice comando BASIC SWAP!

COME È MESSO IN MEMORIA UN PROGRAMMA

Ora è arrivato il momento di vedere come un programma sia immagazzinato nella memoria del vostro computer MSX. Come abbiamo fatto prima, useremo la funzione PEEK per vedere che cosa contengono alcune parti della memoria e scoprire che cosa accade. La prima cosa che dobbiamo sapere è dove comincia di solito un programma nel computer MSX. Questo avviene all'indirizzo 32768. Il byte che si trova a questo indirizzo è *sempre* 0, e i bytes di programma che formano il programma cominciano all'indirizzo successivo, 32769. Questa parte della memoria è il primo gruppo di indirizzi sopra quelli della ROM.
Passiamo ora ad esaminare come un programma viene memorizzato dal vostro computer MSX. Scrivete il programma della Fig. 2.4, ma non fatelo girare.

```
10 A=10
20 PRINT A
30 C$="MSX"
```

Fig. 2.4 Un semplice programma BASIC.

L'indirizzo a cui inizia il programma sarà il normale indirizzo di inizio 32769. In questo esempio, esamineremo i bytes da 32768 a 32800.
Dopo aver usato la solita istruzione per stampare i valori individuati dal comando PEEK, a partire dall'indirizzo 32768 in poi, otterrete il listato della Fig. 2.5. A prima vista, vi apparirà come un insieme di numeri senza significato, ma se osservate più attentamente, potrete notare un certo schema logico. Come al solito, i codici ASCII svolgono l'utile funzione di indicatori. Al 32773, per esempio, potete vedere il numero 65, il codice ASCII per A. Dal momento che sappiamo che la riga è 'A=10', possiamo cercare la rimanente parte di questa riga. Il 10 è riconoscibile come 10 all'indirizzo 32776, per cui il numero 239, che segue immediatamente la 'A', deve rappresentare il segno '='. Ora, questo **non** è il codice ASCII per '=', ma uno di quei 'simboli' (token) di cui avevo parlato nel Capitolo 1. È un simbolo perché si richiede al computer di eseguire un'azione, non di immagazzinare semplicemente un codice ASCII. Il 15 che sta fra il 239 e il 10 è per ora un mistero, è un numero di codice per il computer, che dice alla macchina quanto è grande il numero che segue. Ulteriori dettagli su

questa codifica sono contenuti nell'Appendice C. Lo 0 all'indirizzo 32777 indica la fine di questa riga.
Ora dobbiamo vedercela con i primi quattro bytes della riga, a partire dal numero d'indirizzo 32769. I primi due bytes sono, come avrete immaginato dal loro aspetto, un indirizzo. 10 e 128 costituiscono l'indirizzo 32778. Che cos'è questo indirizzo? Ma naturalmente è l'indirizzo del primo byte della riga successiva, il byte che segue lo zero posto come indicatore della fine della prima riga!
Ecco come il sistema operativo del computer MSX può prelevare le righe e metterle nella sequenza corretta, non importa quale ordine abbiate usato per inserirle. Il mistero finale è facilmente risolto. Il terzo e il quarto byte di ognuna delle righe riportano la sequenza 10, 20, 30, i numeri di riga maggiori di 255.
Per i numeri di riga minori di 256, il secondo di questi bytes, il byte più significativo, è zero. Una riga numerata 300, per fare un esempio pratico, sarebbe codificata come 44 1, cioè 1*256+44.

```
32768  0
32769  10
32770  128
32771  10
32772  0
32773  65
32774  239
32775  15
32776  10
32777  0
32778  18
32779  128
32780  20
32781  0
32782  145
32783  32
32784  65
32785  0
32786  31
32787  128
32788  30
32789  0
32790  67
32791  36
32792  239
32793  34
```

```
32794  77
32795  83
32796  88
32797  34
32798  0
32799  0
32800  0
```

Fig. 2.5. I bytes che rappresentano il programma in memoria.

Ora vediamo come le altre righe sono immagazzinate in memoria. Abbiamo già incontrato il simbolo (token) PRINT, codificato come 145, e tutto il resto dovrebbe esservi ormai familiare. L'unica novità è la fine del programma. L'ultima riga termina con uno 0, come al solito, ma subito dopo, là dove dovrebbero trovarsi i bytes d'indirizzo per la riga successiva, agli indirizzi 32799 e 32800, vi è un'altra coppia di zeri. Questo è l'indicatore che il computer usa per END. Quindi il computer riconosce la fine di un programma BASIC, anche se non usate la parola END.
Possiamo apportare delle interessanti modifiche al nostro programma. Supponiamo, ad esempio, di usare la funzione POKE per gli indirizzi usati per contenere i numeri di riga. Se battete:

```
POKE32780,10:POKE32788,10
```

e premete RETURN, avrete messo il numero di riga 10 in ogni indirizzo riservato al numero di riga per le righe 20 e 30. Ora date il comando LIST e osservate il risultato! È un programma di righe 10! Contrariamente a quanto potreste aspettarvi verrà eseguito in modo del tutto normale. Tutto ciò che un programma richiede per 'girare' nel modo corretto, è che l'indirizzo della riga successiva sia esatto. I numeri di riga non hanno niente a che vedere con questo; sono semplicemente un modo pratico di indicare le righe per nostra utilità. Molti linguaggi di programmazione non usano affatto dei numeri di riga. Un programma così modificato, comunque, non è del tutto normale. L'istruzione LIST 10, per esempio, fa il listato di tutte e tre le righe! In questo semplice esempio, la modifica dei numeri di riga non comporta praticamente nessuna differenza e anche se vi fosse stata un'istruzione GOTO nel programma, le cose non sarebbero cambiate. Provate a scrivere di nuovo il programma in questo modo:

```
10 A=20
20 PRINT A
30 GOTO 10
```

Date il comando RUN per accertarvi che funzioni e poi cambiate la seconda coppia di numeri di riga ancora in 10, con le stesse istruzioni POKE. Vedrete che il programma funziona ancora, perché la riga conte-

nente GOTO 10 non ha bisogno di usare ancora numeri di riga *dopo* che il programma è stato eseguito per la prima volta. Se cambiate la numerazione *prima* dell'esecuzione, diventa una faccenda del tutto diversa. Naturalmente, potete richiamare le righe del programma nel modo consueto ed anche modificare la numerazione delle righe richiamate per riportarla al valore corretto. Potrete anche registrare un programma che sia stato così modificato e di nuovo immetterlo in memoria, mentre l'istruzione RENUM agirà normalmente per ripristinare i numeri di riga originali.

L'ESECUZIONE DI UN PROGRAMMA

Ora che abbiamo visto come un programma è codificato e memorizzato dal computer MSX, possiamo soffermarci un attimo a vedere come avviene la sua esecuzione. Questa azione è svolta dalla parte più complicata del sistema operativo ed occorre dare un indirizzo di inizio, che è il solito 32769 per tutti i computer MSX. Supponiamo di esaminare le azioni, omettendo i dettagli, del programma di tre righe della Fig. 2.4. Al primo indirizzo del programma BASIC, la subroutine RUN leggerà i primi due bytes e li memorizzerà temporaneamente. Questi bytes saranno usati al posto dell'indirizzo dell'"inizio del BASIC" quando verrà eseguita la riga successiva. Poi vengono letti e memorizzati i bytes dei numeri di riga. Come mai? Da quanto abbiamo appena visto, non sembra che sia necessario avere dei numeri di riga. La ragione è questa: se c'è un errore di sintassi nella riga, il computer potrà stampare il messaggio "Syntax error in 10" piuttosto che "Syntax error da qualche parte"! Il byte successivo è un codice ASCII, e il computer lo prenderà come nome di variabile. In passato, occorreva usare la parola LET per "definire una variabile". Ciò richiedeva un altro simbolo (token), mentre i progetti più moderni fanno a meno di LET, per il fatto che, tutto sommato, è abbastanza semplice sistemare le subroutines in modo diverso. Lo schema seguito è di considerare il codice ASCII per una lettera posta immediatamente dopo il numero di riga *sempre* come nome di variabile. Quindi, se mettete un numero al posto della lettera, sarà considerato come facente parte del numero di riga. Il simbolo speciale (token) per il segno "=" fa sì che entri in azione una subroutine. Essa crea un inserimento di dati nella VLT, al primo indirizzo disponibile, e vi immette i numeri indicanti il tipo di variabile (2,3,4 o 8). Il codice ASCII per "A" viene messo alla posizione successiva. L'indirizzo seguente della VLT viene lasciato libero, non c'è nessuna seconda lettera per questo nome di variabile. Sarà quindi letto il numero 10 e convertito nella forma speciale usata dal computer, che potete trovare nell'Appendice B.
Questo insieme di bytes viene posto anche nella VLT come inserimento di dati per A. Viene poi letto il byte successivo del programma, è 0, che

indica la fine della riga. Questo è un segnale affinché l'indirizzo della prima riga, che era stato letto come prima azione, sia immesso nel microprocessore. Il tipo di azione che abbiamo esaminato in dettaglio per la riga 10 si ripete ora per la riga 20. Questa volta, c'è dell'altro lavoro da fare quando viene letto il simbolo dell'azione: poiché questo è il simbolo di PRINT, deve essere richiamata la subroutine per PRINT, Essa definirà l'indirizzo del primo posto libero sullo schermo. Questo viene fatto annotando l'indirizzo in una coppia di bytes della RAM. Se leggete questi bytes, avrete l'indirizzo. Viene poi trovato il valore di A nella VLT e i bytes sono di nuovo convertiti sotto forma di codici ASCII. I codici vengono poi immessi, uno per uno, nella memoria video. Dopo che ogni codice ASCII è stato memorizzato, il numero della memoria video sarà incrementato. L'immagazzinamento dei codici ASCII nella memoria video fa in modo che i caratteri siano visibili sullo schermo per effetto di un'altra subroutine. Ancora una volta, lo zero alla fine della riga fa sì che venga usato il successivo numero di riga. Alla fine della terza riga, però, il numero della "riga successiva" è zero e il programma termina. Il computer ritorna nello stato di attesa, pronto per un nuovo comando.
Non è *proprio* tutto così semplice come sembra da questa descrizione, comunque, abbiamo esaminato i punti essenziali. E' importante rendersi conto di quante azioni ci siano da svolgere, ed ognuna deve essere eseguita un passo alla volta. Ciò che rende lento questo tipo di BASIC è che ogni simbolo (token) richiama una subroutine, che deve essere cercata. Per esempio, se avete un programma con un loop di questo tipo:

```
10 FOR N=1 TO 50
20 PRINT N
30 NEXT
```

l'azione di lettura del simbolo PRINT, 145, e di individuazione dell'indirizzo a cui si trova la corretta subroutine, sarà eseguita 50 volte. Non c'è un modo semplice per assicurarci che, una volta trovata la subroutine, questa sia semplicemente usata 50 volte.
Il tipo di BASIC del vostro computer MSX è un BASIC "interpretato", cioè ogni istruzione viene eseguita man mano che il computer arriva ad essa. Quindi, se l'istruzione dice di trovare l'indirizzo della subroutine 50 volte, così sarà fatto. L'alternativa è uno schema chiamato di "compilazione", in cui tutto il programma viene convertito in un efficiente codice macchina prima di essere eseguito. La compilazione viene effettuata da un altro programma, chiamato "compilatore". Al momento, non conosco alcun programma compilatore per il computer MSX, ma può darsi che ce ne sia uno disponibile quando leggerete questo libro!

CAPITOLO 3

Il miracoloso microprocessore

In questo capitolo cominceremo a prender confidenza col microprocessore Z80 del computer MSX. Il microprocessore, o MPU, è, come ricordate, la parte "operativa" del computer, distinta dalla parte riservata all'immagazzinamento dei dati (memoria) o dalla parte che gestisce l'input e l'output (porte), ed il compito del microprocessore è di controllare ciò che fa il resto del computer. Inoltre, la MPU decide anche quanta memoria possa essere utilizzata ogni volta.
La MPU in se stessa consiste di un insieme di parti di memoria riservate ai numeri, governate da una ben precisa e completa organizzazione.
Attraverso circuiti che, molto opportunamente, sono chiamati "gates"*, può essere controllato il modo in cui i segnali elettrici per i bytes sono suddivisi fra varie parti della memoria della MPU.

Sono queste azioni di ripartizione dei segnali che costituiscono l'addizione, la sottrazione, le operazioni logiche e le altre azioni della MPU.
Ognuna di queste azioni è programmata. Non accadrà niente finché un byte d'istruzione non è presente ad ognuno degli otto terminali dei dati, sotto forma di segnale 1 o 0, e questi bytes sono usati per controllare i "gates" all'interno della MPU. Ciò che rende particolarmente utile l'intero sistema è il fatto che le istruzioni del programma siano sotto forma di segnali elettrici su otto linee, per cui questi segnali possono essere cambiati molto rapidamente. La velocità è decisa da un altro circuito elettrico, chiamato "generatore di impulsi di clock" o semplicemente "clock" (orologio). La velocità scelta come standard per il computer MSX è decisamente molto alta. Sarete senz'altro abituati ad orologi che oscillano una volta al secondo, ma l'"orologio" del computer MSX oscilla circa tre *milioni e mezzo* di volte al secondo. Questo non significa che il computer possa eseguire tre milioni e mezzo di istruzioni al secondo, perché alcune istruzioni richiedono molte oscillazioni di clock per la loro esecuzione, ma, in ogni caso, ciò significa che le cose si svolgono velocemente. Dev'essere per forza così, poiché la MPU opera in sequenza. Può fare solo una cosa per volta, e quindi ogni operazione deve essere eseguita rapidamente.

* "Gate", dall'inglese "cancello", indica appunto il dispositivo che consente l'entrata o l'uscita dei segnali alla MPU.

IL CODICE MACCHINA

Un programma per la MPU, come abbiamo visto, è formato da codici numerici ed ognuno di questi codici è un numero compreso fra 0 e 255 (un numero di un solo byte). Alcuni di questi numeri possono essere bytes di istruzione, che obbligano la MPU a fare qualcosa. Altri possono essere bytes di dati, cioé numeri da addizionare, o memorizzare, o spostare, oppure codici ASCII per le lettere. La MPU non può distinguerli l'uno dall'altro — opera semplicemente nel modo in cui è stata programmata. Sta al programmatore definire i numeri e metterli nell'ordine corretto in cui dovranno essere immagazzinati in memoria. L'ordine corretto, per quanto riguarda la MPU, è molto semplice. Il primo byte inviato alla MPU dopo aver acceso il computer o dopo aver completato un'altra istruzione, viene considerato come byte d'istruzione, cioé un byte che obbligherà la MPU a far qualcosa. Ora, molte istruzioni dello Z80 sono formate da un solo byte e non richiedono dati. Altre possono essere seguite da uno o due bytes di dati e alcune istruzioni richiedono due o più bytes, anziché uno solo, e possono essere seguite anch'esse da dati. Quando la MPU legge un byte d'istruzione, analizza l'istruzione per trovare se essa debba essere seguita o no da uno o più bytes. Se, ad esempio, il byte d'istruzione è uno di quelli che devono essere seguiti da due bytes di dati, allora, dopo che la MPU ha analizzato il primo byte, considererà i due bytes successivi che le vengono inviati come bytes di dati per quell'istruzione. L'azione della MPU è completamente automatica, e fa parte delle caratteristiche proprie del chip Z80. La difficoltà è che il programmatore in codice macchina deve attenersi esattamente alle stesse regole e fare in modo che il programma sia corretto. Il 100% di esattezza può essere abbastanza buono! Se inviate al microprocessore un byte d'istruzione quando si aspetta un byte di dati, o viceversa, avrete dei guai. Guai che quasi sempre consistono in un ciclo (loop) senza fine, che fa scomparire l'immagine dallo schermo e disattiva la tastiera. Anche i tasti CTRL STOP non riescono a far uscire il computer da questo loop, come avete visto, e l'unico rimedio è di spegnerlo. In genere, perderete tutto quello che si trovava in memoria, per cui è essenziale salvare ogni programma in codice macchina, o un programma BASIC che origina azioni dirette in codice macchina (come la funzione POKE) prima di provarlo.

A questo punto, desidero rimareare che la programmazione in codice macchina è noiosa. Non è necessariamente difficile — si tratta di mettere insieme delle semplici istruzioni per una macchina semplice — ma spesso è difficile ricordarsi tutti i dettagli che occorrono. Quando programmate in BASIC, i messaggi d'errore del computer vi mantengono sulla giusta strada e vi aiutano a scoprire gli errori. Quando usate il codice macchina, dovete agire per conto vostro, e dovete scoprire da soli gli errori che avete fatto. Sotto questo aspetto, un programma chiamato "assemblatore" è di

notevole aiuto. Ne riparleremo più avanti, al Capitolo 8. Nel frattempo, il miglior modo d'imparare il codice macchina è di scriverlo, usarlo e fare errori. Cominceremo a vedere come si fa tra poco, ma prima dobbiamo esaminare i modi di scrivere i numeri che costituiscono i bytes di un programma in codice macchina.

BINARIO, DECIMALE ED ESADECIMALE

Un programma in codice macchina è formato da un insieme di codici numerici. Poiché ogni codice numerico è un modo di rappresentare gli 1 e gli 0 di un byte, sarà formato da numeri compresi tra 0 e 255 quando lo scriviamo in forma decimale. Il programma è inutilizzabile finché non viene immesso nella memoria del computer MSX: infatti, poiché la MPU agisce molto velocemente, l'unica maniera di inviarle bytes alla stessa velocità con cui essa può utilizzarli, è d'immagazzinare i bytes nella memoria e poi lasciare che la MPU li prelevi in ordine. Non potreste mai digitare i numeri tanto velocemente da soddisfare la MPU ed anche metodi come il nastro o il disco non sono abbastanza veloci.
L'inserimento dei bytes in memoria è perciò una parte essenziale del lavoro di stesura di un programma in codice macchina. Vedremo in seguito alcuni metodi più dettagliatamente. Dei programmi semplici e *molto brevi* potrebbero essere inseriti in memoria col metodo più primitivo, cioé usando otto interruttori. Ogni interruttore potrebbe essere posizionato in modo da dare un segnale elettrico 1 o 0 e si potrebbe premere un bottone affinché la memoria immagazzini i numeri rappresentati dagli interruttori e poi venga scelto il successivo indirizzo di memoria. Ma programmare in questo modo è davvero troppo noioso e dopo un po' che lavorate con numeri binari, vi si incrocerà la vista! Dal momento che abbiamo i computer, sembra ragionevole utilizzare il computer per immettere numeri in memoria e un passo altrettanto ovvio sarà di usare la rappresentazione numerica più conveniente.
Quale sia la rappresentazione numerica più conveniente dipende da come immettete i numeri e da quanta programmazione in codice macchina fate. Il computer MSX contiene delle routines per convertire i numeri binari che si trovano nella sua memoria in numeri decimali, che verranno stampati sullo schermo, ed eseguirà anche l'azione opposta. Quando usate la funzione PEEK, l'indirizzo desiderato può essere scritto in forma decimale ed il risultato di PEEK sarà un altro numero decimale, compreso tra 0 e 255. Anche quando usate la funzione POKE potete indicare sia il numero d'indirizzo che il byte in forma decimale. Se volete usare il codice macchina solo per delle brevi routines, allora questa rappresentazione numerica è perfettamente adeguata.
I veri programmatori in codice macchina ritengono che l'uso dei numeri decimali sia tutt'altro che conveniente. Un numero decimale per un byte

può essere di una cifra (come 4) o di due (come 17) o di tre (come 143). Una codifica molto più conveniente è quella chiamata codifica esadecimale. Tutti i numeri di un solo byte possono essere rappresentati da due cifre esadecimali. Inoltre, i veri programmatori in codice macchina scrivono i loro programmi nel cosidetto **linguaggio assembly**. Esso usa, come comandi, le versioni abbreviate dei nomi dei comandi diretti alla MPU. Dei programmi chiamati "assemblatori" poi convertono questi comandi nel corretto codice binario. Praticamente tutti gli assemblatori fanno comparire sullo schermo questi codici in forma esadecimale, anziché decimale.
Inoltre, quando immettete i numeri dei dati in linguaggio assembly, dovrete probabilmente utilizzare la codifica esadecimale. Poiché il computer MSX contiene anche delle routines per lavorare con la codifica esadecimale, sembra abbastanza ragionevole imparare ad usare questa facilitazione. Infatti, non solo vi permetterà di fare facilmente dei progressi nell'apprendimento del codice macchina, ma vi permetterà anche di utilizzare dell'eccellente software per lo sviluppo del codice macchina, quale lo ZEN, di cui parleremo più avanti.
Esadecimale significa una scala di sedici, e la ragione per cui è così ampiamente usata, sta nel fatto che è naturalmente adatta a rappresentare dei bytes binari. Quattro bit, un semibyte, possono rappresentare numeri compresi fra lo 0 e il 15 della scala decimale. Questa è appunto la gamma fra cui può variare una cifra esadecimale (Fig. 3.1). Poiché non abbiamo simboli per rappresentare singole cifre maggiori di 9, dobbiamo usare le lettere A,B,C,D,E ed F, oltre le cifre da 0 a 9, nella scala esadecimale. Il vantaggio è che un byte può essere rappresentato da un numero di due cifre, ed un indirizzo completo da un numero di quattro cifre.

Esadec.	Decim.	Esadec.	Decim.
0	0	B	11
1	1	C	12
2	2	D	13
3	3	E	14
4	4	F	15
5	5	poi	
6	6	10	16
7	7	11	17
8	8	fino	
9	9	20	32
A	10	21	33
		etc.	

Fig. 3.1. Cifre decimali ed esadecimali.

I codici numerici usati come istruzioni sono stati progettati nella codifica esadecimale, per cui possiamo notare molto meglio le relazioni fra i comandi. Per esempio, possiamo trovare che un insieme di comandi correlati fra loro cominciano tutti con la stessa cifra quando sono scritti nella notazione esadecimale. Usando invece quella decimale, questa correlazione non apparirebbe. Inoltre, è molto più facile scrivere il numero binario effettivamente usato dal computer quando avete davanti la versione esadecimale. La conversione tra binario ed esadecimale è molto più semplice della conversione tra binario e decimale. L'uso di programmi assemblatori, come l'eccellente ZEN, richiedono una certa familiarità con la codifica esadecimale, ed i libri che riguardano la MPU Z80 presumono tutti che voi conosciate il sistema esadecimale. Sembra proprio che dobbiamo cominciare ad occuparcene!

LA SCALA ESADECIMALE

La scala esadecimale consiste di sedici cifre, partendo dallo 0 e arrivando fino al 9 come nella notazione decimale. La cifra successiva però non è 10, perché questo indicherebbe un sedici e zero unità e, dal momento che non abbiamo simboli per cifre oltre il 9, usiamo le lettere da A ad F. Il numero che scriviamo come 10 (dieci) decimale, viene scritto come 0A nella codifica esadecimale, 11 è 0B, 12 è 0C e così via fino a 15, che è 0F. Lo zero non *deve* essere scritto, ma i programmatori hanno l'abitudine di scrivere un byte di dati con due cifre ed un indirizzo con quattro, anche se sono richieste meno cifre. Il numero che segue 0F è 10, cioè 16 nella notazione decimale, e la scala poi si ripete fino a 1F, trentuno decimale, che è seguito da 20, trentadue decimale. La grandezza massima di un byte, 255 decimale, è FF esadecimale. Quando si scrivono dei numeri esadecimali, di solito vengono contraddistinti in qualche modo per non confonderli con i numeri decimali. Non potete certo confondere un numero come 3E con un numero decimale, ma un numero come 26 potrebbe essere decimale o esadecimale. La convenzione seguita da molti programmatori dello Z80 è di usare una H maiuscola per contrassegnare un numero esadecimale, posta *dopo* il numero. Per esempio, il numero 47H significa 47 esadecimale, ma 47 e basta significherebbe il numero decimale quarantasette.
Quando scrivete dei numeri esadecimali per un programma dello Z80, è consigliabile seguire questa convenzione. Questa è, ad esempio, anche la convenzione seguita dall'assemblatore ZEN per il computer MSX. Il computer MSX da parte sua, comunque, usa una diversa forma quando volete usare la funzione POKE con numeri esadecimali. Il metodo del computer MSX è di mettere &H *prima* del numero, per cui POKE &H800C, &HA immette in memoria il numero esadecimale 0AH all'indirizzo esadecimale 800CH. Se lavorate con le istruzioni PEEK e POKE, non c'è un gran vantaggio nell'uso della codifica esadecimale, soprattutto perché ciò

implica l'uso di altri due caratteri per ogni numero. Quando i bytes di dati devono essere letti in un loop, questo metodo può essere utile perché vi permette di mantenere i dati in forma esadecimale e ciò ne rende il controllo più semplice di quanto sarebbe nella corrispondente forma decimale.
Il merito principale della codifica esadecimale sta nella sua facile conversione nella codifica binaria. Se guardate la tabella binario-esadecimale della Fig. 3.2, potete vedere che 9H è 1001 in binario e FH è 1111. Il numero esadecimale 9FH perciò è 10011111 in binario, basta scrivere le cifre binarie che corrispondono alle cifre esadecimali. Per fare un altro esempio, il byte esadecimale B8 è 10111000, perché B è 1011 e 8 è 1000.

Esad.	Binario	Esad.	Binario
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Fig. 3.2. Cifre esadecimali e binarie.

La conversione in senso opposto è altrettanto facile, raccogliete in gruppi di quattro le cifre binarie, cominciando dal lato meno significativo del numero (quello destro) e poi convertite ogni gruppo nella sua corrispondente cifra esadecimale. La Fig. 3.3 riporta esempi della conversione in un senso e nell'altro, in modo che possiate constatarne la facilità.

CONVERSIONE: DA ESADECIMALE A BINARIO

Esempio:
2CH .... 2H è 0010 binario
CH è 1100 binario
Quindi 2CH è 00101100 binario (byte di dati)

Esempio:
4A7FH .... 4H è 0100 binario
AH è 1010 binario
7H è 0111 binario
FH è 1111 binario
Quindi 4A7FH è 0100101001111111 binario (un indirizzo)

CONVERSIONE: DA BINARIO AD ESADECIMALE

Esempio:

01101011 .... 0100 è 6H
1011 è BH

Quindi 01101011 è 6BH

Esempio: 10110110010010, notate che non è un numero completo di bytes.

Raggruppate in quattro, a partire dal meno significativo:

0010 è 2H
1001 è 9H
1101 è DH

il restante 10 è 2, quindi 2D92H.

Fig. 3.3. Conversione tra esadecimale e binario.

Il computer MSX è uno dei computer meglio progettati poiché può effettuare la conversione fra esadecimale e decimale attraverso comandi BASIC. Se volete trovare l'equivalente esadecimale di un numero decimale, basta usare l'istruzione HEX$(numero).
Per esempio, ?HEX$(46) visualizzerà 2E, cioé il valore esadecimale del numero decimale 46. Per trovare l'equivalente decimale di un numero esadecimale, basta usare &H. Per esempio, ?&H3F3A darà come risultato 16186, l'equivalente numero decimale. Questo va bene per numeri non maggiori di &H7FFF, ma dopo questo numero accadono delle cose un po' strane. Per esempio, se digitate ?&HC004, avrete come risultato — 16380. Quando trovate che una conversione vi dà un numero negativo, potete ottenere il risultato corretto aggiungendo 65536. In questo caso, ad esempio, la risposta giusta è 65536—16380, che è uguale a 49156. La ragione di questo strano comportamento è il modo in cui i segni negativi sono rappresentati nei numeri binari, ma lo vedremo più avanti. Se volete usare dei numeri negativi con l'istruzione POKE, però, il computer li userà nel modo corretto. Ad esempio, POKE 49156,10 produce lo stesso effetto di POKE—16380,10. Il fatto che sia stata inserita una routine di conversione nel computer MSX significa che non dovete preoccuparvi di imparare i metodi, abbastanza noiosi, in verità, per eseguire la conversione fra numeri decimali ed esadecimali. Tuttava, nell'ipotesi che vogliate comunque farlo per servirvene quando non avete accanto l'amico computer MSX, i metodi sono riportati nell'Appendice D.
Il computer MSX non ha però inserito un assemblatore. Un assemblatore vi permette di scrivere in codice macchina con la stessa facilità con cui scrivete in BASIC, usando comandi formati non da numeri, ma da parole.

Poiché un eccellente assemblatore, lo ZEN, è disponibile su nastro, questa non è tuttavia una perdita catastrofica.
Supponendo, il che è ragionevole, che non vogliate sostenere il costo di un assemblatore, come fate per creare programmi in codice macchina?
La risposta è: scrivete il vostro programma in linguaggio assembly, che è senza dubbio il modo più facile per scrivere dei programmi in codice macchina, e poi convertitelo nella codifica esadecimale. Effettuare questa conversione significa cercare in un insieme di tabelle, chiamato "elenco delle istruzioni", il numero esadecimale che rappresenta ogni istruzione. Gli elenchi delle istruzioni sono forniti dai costruttori di microprocessori e Zilog, che ha progettato lo Z80, ne fornisce uno per questo chip.
Tuttavia, per aiutarvi, un manuale di veloce consultazione è stato incluso nell'Appendice E. Non consultatelo, per il momento, vi metterebbe fuori strada! Sebbene soltanto alcuni comandi della MPU siano veramente diversi fra loro, ci sono molte sottili variazioni per ogni comando, ed ogni variazione ha un separato numero di codice. Finché non capirete perché occorrono così tante variazioni, guardare le tabelle ora servirebbe solo a confondervi le idee.

NUMERI NEGATIVI

I numeri negativi sono molto importanti nei programmi in codice macchina, soprattutto se lavorate senza un assemblatore. La ragione sta nel fatto che talvolta voi desiderate che la MPU esegua l'equivalente di un GOTO, magari saltando ad un certo punto che si trova 30 passi prima dell'indirizzo attuale. Questo viene di solito effettuato ponendo, nel programma, un numero per i dati che rappresenta il numero di passi da saltare. Se volete eseguire un salto all'indietro ad un passo precedente, dovrete usare un numero *negativo* per questo byte di dati. Questo accade spesso, perché è il modo in cui un "loop" viene programmato in codice macchina. Perciò dobbiamo saper scrivere un numero negativo nella codifica esadecimale. Inoltre, poiché la conversione decimale-esadecimale del computer MSX può dare numeri negativi (come abbiamo appena visto), è importante sapere quando questo avverrà e perché. Lo strano è che non vi è alcun segno negativo nell'aritmetica esadecimale, e nemmeno in quella binaria. La conversione di un numero nella sua forma negativa viene effettuata con il *completamento* del numero, e la Fig. 3.4 mostra questo procedimento. Al primo sguardo, e molto spesso anche al secondo, al terzo e al quarto, tutto ciò sembra completamente folle! Per esempio, per i numeri di un solo byte, la forma decimale del numero —1 è 255! Si usa un numero grande per rappresentarne uno negativo piccolo! Tutto ciò comincia ad acquistare un certo senso logico quando considerate i numeri nella loro forma binaria. I numeri che possono essere considerati

come negativi cominciano tutti con un 1 e i numeri positivi cominciano tutti con uno 0. La MPU può quindi individuarli eseguendo un test sul bit di sinistra, il bit più significativo.

Numero binario	00110110	Dec. 54
Invertito	11001001	
più 1	11001010	Dec. —54
Numero decimale —5		
In binario è 101, e nel binario		
a otto bit è	00000101	
invertito	11111010	
più 1	11111011	cioè il byte per —5

Fig. 3.4. Il complemento a due, o forma negativa, di un numero binario.

È un metodo semplice, che la macchina usa in modo efficiente, ma ha degli svantaggi per l'uomo. Uno di questi svantaggi è il fatto che le cifre di un numero negativo non sono identiche a quelle di un numero positivo. Per esempio, —40 decimale ha le stesse cifre di +40, in esadecimale —40 diventa D8H e +40 diventa 28H. Il numero decimale —85 diventa ABH e +85 diventa 55H. Non è così ovvio che uno sia la forma negativa dell'altro. Il secondo svantaggio è che l'uomo non può distinguere fra un numero di un solo byte che si considera negativo ed uno che è maggiore di un byte di 127.
Per esempio, 9FH significa 159 o —97? La risposta è che il programmatore non deve preoccuparsi. Il microprocessore userà il numero in modo corretto in ogni caso. La difficoltà è che dobbiamo sapere quale sia questo uso corretto in ciascuno dei casi. In questo libro, ed in altri che trattano della programmazione in codice macchina, vedrete spesso le parole "con segno" e "senza segno". Un numero con segno è un numero che può essere negativo o positivo. Per un numero di un solo byte, i valori da 0 a 7FH sono positivi, e i valori da 80H a FFH sono negativi. Questo corrisponde ai numeri decimali da 0 a 127 per i numeri positivi e da 128 a 255 per quelli negativi. I numeri senza segno sono sempre considerati positivi. Se trovate il numero 9CH definito come numero con segno, sapete che sarà considerato come numero negativo (è maggiore di 80H). Se è definito come numero senza segno, allora è positivo ed il suo valore viene ottenuto con la semplice conversione. C'è però un piccolo problema: quando usiamo i metodi di conversione rappresentati dai comandi HEX$ e &H, non verranno considerati i segni dei bytes singoli. Se, ad esempio, digitate: ?HEX$(—5), avrete FFF8 invece di F8, perché HEX$ lavora con numeri esadecimali a quattro cifre. Così, se usate ?&HF8, avrete 248 invece che —5. Come facciamo a convertire in decimale un numero esadecimale

con segno, di un solo byte, quando il computer MSX non accetta numeri negativi di un solo byte per la conversione? Semplice, digitate ?&HXX-256 dove XX sta per il numero esadecimale. Per esempio, ?&HFF—256 vi darà —1, che è il valore corretto di FFH considerato come numero negativo. Per le conversioni con HEX$, basta prendere le due cifre sul lato destro.

UNA BREVE PAUSA

Per sospendere un attimo la trattazione di tutte queste nozioni matematiche, diamo un'altra occhiata al video del computer MSX. Sappiamo già, dal Cap. 1, che ogni parte dello schermo può essere controllata da una qualunque cosa immagazzinata nella parte di memoria chiamata "memoria video". Possiamo trovare gli indirizzi della memoria video usando il comando BASIC BASE, e possiamo usare le funzioni VPEEK e VPOKE per leggere o modificare il contenuto di questi indirizzi. Il modo di schermo più semplice da usare è quello chiamato "memoria dello schermo di scrittura", SCREEN0. Ora, se cercate sul manuale MSX il comando BASE, vedrete che menziona vari tipi di schermo e di tabelle. È il momento di considerare un po' più attentamente tutto questo, perché potrete avere molti effetti interessanti usando l'istruzione BASE. Per lo schermo di scrittura, si possono usare solo i numeri 1 e 2 con BASE. Se digitate ?BASE(0), avrete 0 come risultato. Questo è chiamato l'indirizzo di inizio della "tabella dei nomi". Il risultato di ?BASE(2) è 2048, e questo indirizzo è l'inizio della tabella generatrice di forme. Che cosa vogliono dire questi nomi?
Forse l'uso di VPEEK ci aiuterà. Guardate l'effetto del programma della Fig. 3.5.

```
10 FOR N=2048 TO 2208 STEP 8
20 FOR Z=0 TO 7
30 PRINTVPEEK(N+Z);:NEXT
40 PRINT:NEXT
```

Fig. 3.5. Un programma che mostra il contenuto della tabella delle forme.

Esso sceglie alcuni indirizzi che iniziano da 2048, e stampa un gruppo di otto bytes a partire da ogni indirizzo. La Fig. 3.6 mostra i bytes generati.

```
0 0 0 0 0 0 0 0
60 66 165 129 165 153 66 60
60 126 219 255 255 219 102 60
108 254 254 254 124 56 16 0
16 56 124 254 124 56 16 0
16 56 84 254 84 16 56 0
```

16 56 124 254 254 16 56 0
0 0 0 48 48 0 0 0
255 255 255 231 231 255 255 2
55
56 68 130 130 130 68 56 0
199 187 125 125 125 187 199 2
55
15 3 5 121 136 136 136 112
56 68 68 68 56 16 124 16
48 40 36 36 40 32 224 192
60 36 60 36 36 228 220 24
16 84 56 238 56 84 16 0
16 16 16 124 16 16 16 16
16 16 16 255 0 0 0 0
0 0 0 255 16 16 16 16
16 16 16 240 16 16 16 16
16 16 16 31 16 16 16 16

Fig. 3.6. I bytes letti dalla tabella delle forme.

La prima fila è formata da zeri, ma la serie successiva è un insieme di numeri. Ora, quando i progettisti di computer parlano di una tabella generatrice di forme, intendono una serie di numeri che creano una figura sullo schermo. Che cosa crea il secondo insieme di numeri? Per vederlo, dobbiamo sapere qual è la correlazione tra numeri e figure. La Fig. 3.7 mostra una griglia per disegnare una figura, come quella che usereste per disegnare gli sprite.

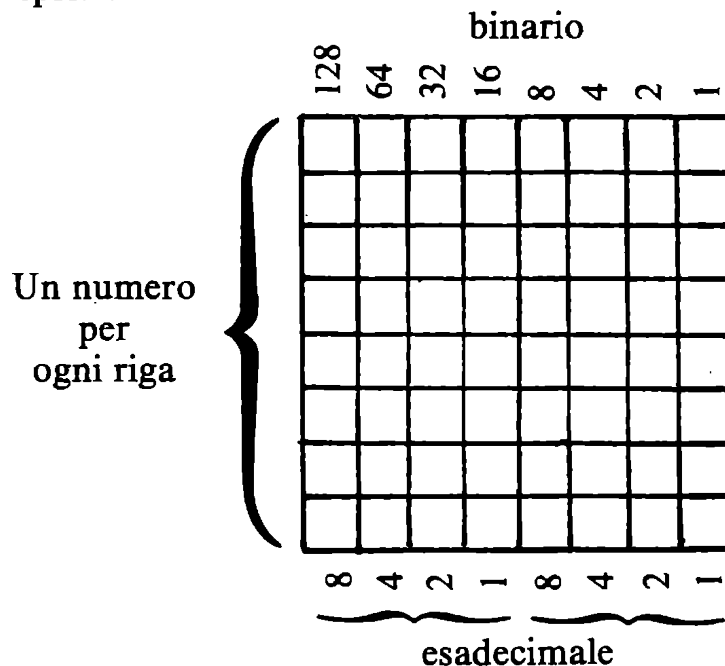
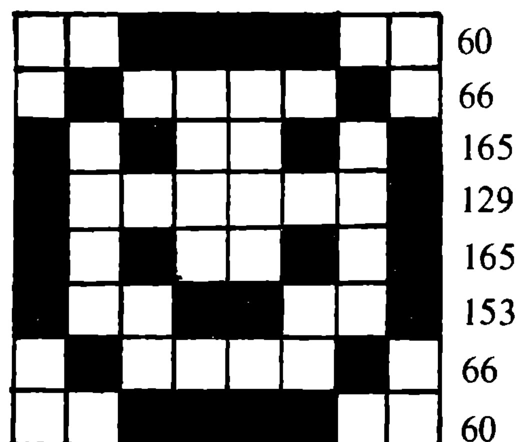


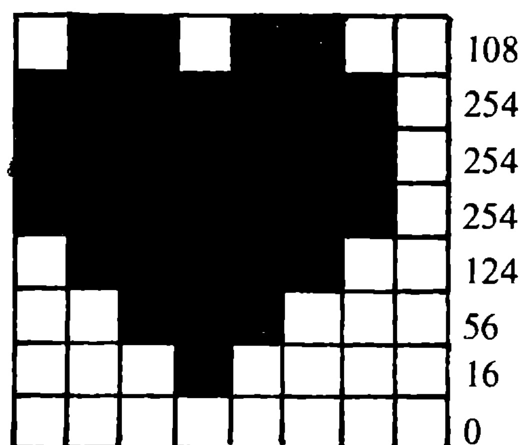
Fig. 3.7. La griglia per disegnare le forme.

Ogni colonna è contraddistinta da un numero. Questi numeri sono semplicemente i valori decimali del corrispondente numero binario. Alla base della griglia, i numeri sono divisi in due gruppi per l'uso con i codici esadecimali. Ogni carattere sullo schermo viene disegnato su una griglia come questa. Per vedere a quale forma corrisponde un insieme di numeri, dobbiamo analizzare i numeri, e la Fig. 3.8 mostra la prima fila di numeri (l'insieme degli 0 è stato saltato) così analizzata.



CHR$(1);CHR$(65)

Fig. 3.8. Analisi di un insieme di otto numeri di una forma.



CHR$(1);CHR$(67)

Fig. 3.9. La figura a forma di cuore analizzata.

Il numero 60, per esempio, può essere scritto come 32+16+8+4, per cui ho annerito i quadrati che corrispondono ai numeri 32, 16, 8 e 4 nella prima fila. Il numero 66 è 64+2, per cui ho annerito i quadrati sotto le colonne del 64 e del 2 nella seconda fila. Il resto dello schema è riempito in modo analogo, per generare la forma che vedete nella fig. 3.8. Questa è la

figura del "volto sorridente", che normalmente è programmata da ?CHR$(1) ;CHR$(65). Se saltiamo un insieme di numeri e passiamo al quarto insieme, possiamo vedere (Fig. 3.9) che viene rappresentata la figura a forma di cuore, normalmente programmata da ?CHR$(1); CHR$(67).
Possiamo dimostrare tutto questo, e che uso possiamo farne? La risposta a tutte e due le domande è questa: dal momento che questa è la memoria RAM, possiamo modificarla, e in questo modo possiamo cambiare le figure! Date un'occhiata alla Fig. 3.10, che mostra una figura a forma di H disegnata su una di queste griglie 8X8. Mostra anche i numeri usati come codici per questa figura.

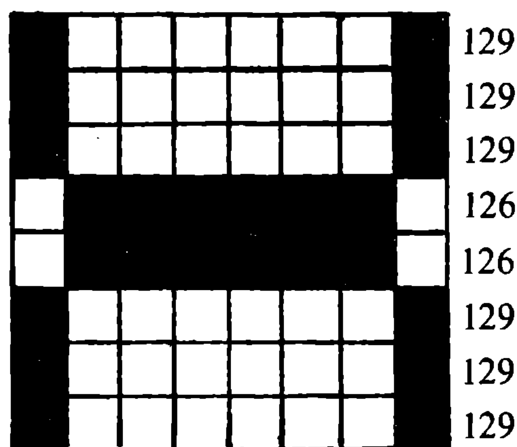


Fig. 3.10. La forma di una "H" disegnata sulla griglia.

```
10 FOR N%=2056 TO 2063
20 READ D%:VPOKE N%, D%:NEXT
30 DATA 129,129,129,126,126,129,129,129
```

Fig. 3.11. Un programma che immette i bytes delle forme nella tabella delle forme.

Il programma della Fig. 3.11 immette questi numeri, con la funzione VPOKE, nella seconda posizione della tabella delle forme. Quando fate questo, il risultato di CHR$(1) ;CHR$(65) è la parte sinistra della "H", non il volto sorridente.
Abbiamo creato la forma dei nostri caratteri e l'abbiamo immessa in memoria, dove rimarrà finché non immettiamo qualcosa di diverso a questi indirizzi con la funzione VPOKE, oppure finché non useremo il comando SCREEN, o finché non spegneremo il computer. Quando il computer viene riacceso, il sistema operativo immetterà di nuovo i numeri corrispondenti al volto sorridente in questa parte della RAM video, ed un comando SCREEN avrà lo stesso effetto.
La tabella delle forme non contiene però solo gli schemi per questi caratteri grafici. Poiché ci sono 32 caratteri, ognuno dei quali usa 8 bytes, ci

saranno 32×8=256 bytes usati in questo modo. Supponiamo di guardare che cosa contenga la tabella delle forme 256 bytes più avanti, dall'indirizzo 2304 in poi. La Fig. 3.12 mostra ancora il programma, con i nuovi indirizzi.

```
10 FOR N%=2304 TO 2464 STEP 8
20 FOR Z%=0 TO 7
30 PRINTVPEEK(N%+Z%) ;:NEXT
40 PRINT : NEXT
```

Fig. 3.12. Una parte successiva della tabella delle forme viene letta da questo programma.

Questa volta, le forme sono assai diverse (Fig. 3.13).

```
0 0 0 0 0 0 0 0
32 32 32 32 0 0 32 0
80 80 80 0 0 0 0 0
80 80 248 80 248 80 80 0
32 120 160 112 40 240 32 0
192 200 16 32 64 152 24 0
64 160 64 168 144 152 96 0
16 32 64 0 0 0 0 0
16 32 64 64 64 32 16 0
64 32 16 16 16 32 64 0
32 168 112 32 112 168 32 0
```

Fig. 3.13. I bytes letti dal programma della Fig. 3.12.

I primi otto bytes sono ancora 0, che sta ad indicare, come ora sappiamo, uno spazio vuoto. L'insieme successivo può essere analizzato come abbiamo fatto prima, ed avremo il punto esclamativo. Troverete che nessuna forma di questo insieme contiene qualcosa nella colonna "1" sul lato destro. Infatti, lo schermo in modo di scrittura non visualizza questa colonna, considera solo 7 colonne per ogni carattere. Ecco perché vediamo solo caratteri parziali quando stampiamo grafici sullo schermo in modo di scrittura. Potete cambiare ogni carattere di questo insieme altrettanto facilmente, come avete fatto con le figure. Per una facile prova, digitate VPOKE 2307,32. Questa istruzione immette un puntino come forma del primo carattere dell'insieme, mentre normalmente è uno spazio. Ogni spazio sullo schermo ora è un puntino! Anche questa volta, rimarrà in memoria finché non usate nuovamente VPOKE, o il comando SCREEN, o spegnete il computer.
Abbiamo già visto, nel Cap. 1, come sono usati gli indirizzi della "tabella dei nomi". Questi indirizzi, che partono da 0, corrispondono alle posizio-

i sullo schermo. L'indirizzo 0 indica la posizione nell'angolo in alto a inistra dello schermo. Questa, fra parentesi, è ancora più alla sinistra di uella che normalmente userete, a meno che non abbiate usato VIDTH40 per alterare la forma dello schermo. La posizione nell'angolo ı basso a destra dello schermo è l'indirizzo 959. Usando questi indirizzi, ossiamo far apparire lettere ed altre forme che vogliamo sullo schermo.)i solito, si usano istruzioni come VPOKE 24,65, che fanno apparire la ttera "A" sul centro-destra della prima riga. Potete ottenere tutte le orme grafiche, se usate VPOKE con numeri da 0 a 31. VPOKE 959,1, per sempio, visualizzerà il volto sorridente nell'angolo inferiore destro dello chermo.

Iel modo SCREEN1, le tabelle sono posizionate in maniera diversa, ed è uesta la ragione per cui lo schermo deve essere ripulito ogni volta che ambiate il modo. Nel modo SCREEN1, la "tabella dei nomi" delle posiioni di schermo inizia a 6144. Poiché lo schermo di SCREEN1 è di 32 aratteri per 24 righe, ha bisogno solo di 768 bytes, da 6144 a 6911. La tabella dei colori", usata per controllare il colore di ogni posizione del arattere, inizia a 8192. La tabella generatrice di forme ora comincia a 0, e 912 è l'inizio della "tabella degli attributi di sprite". Questa tabella conıene le informazioni su ogni sprite usato in una visualizzazione in CREEN1, ma NON le forme di sprite. Queste sono contenute in un'altra abella che inizia all'indirizzo 14336. Poiché, di solito, non abbiamo bisono di interferire con le tabelle degli sprite, le lasceremo in pace, per ora. .e tabelle dei colori sono assai complicate, poiché sono le tabelle dei rafici ad alta risoluzione, e sarà quanto mai opportuno lasciare in pace nche queste!

CAPITOLO 4

I dettagli dello Z80

I REGISTRI, IL PC E L'ACCUMULATORE

Un microprocessore è formato da un insieme di memorie, assai diverse dalla ROM o dalla RAM, chiamate *registri.* I registri possono essere collegati l'uno con l'altro ed anche ai piedini della MPU attraverso i circuiti chiamati "gates". Tutte le azioni della MPU sono eseguite facendo questi collegamenti.
In questo capitolo, esamineremo alcuni dei più importanti registri dello Z80 ed il modo in cui vengono usati. Un buon punto di partenza è il registro chiamato PC, abbreviazione di Program Counter, Contatore di Programma.
No, non conta i programmi; ciò che fa è contare i *passi* di un programma. Il PC è un registro a sedici bit (due byte) che può immagazzinare un indirizzo completo, fino a FFFFH (65535 decimale). Il suo scopo è d'immagazzinare un numero d'indirizzo, e il numero contenuto nel PC sarà incrementato (aumentato di 1) ogni volta che un'istruzione è stata portata a termine, o quando occorre un altro byte. Per esempio, se il PC contiene l'indirizzo 1F3AH, e questo indirizzo contiene un byte d'istruzione, allora il PC verrà incrementato a 1F3BH quando la MPU sarà pronta per un altro byte. Il byte successivo sarà quindi letto da questo nuovo indirizzo. Quando il computer viene acceso, l'indirizzo del PC deve essere quello della prima istruzione della ROM.
Ciò che rende così importante il PC è che esso costituisce il modo automatico mediante il quale la memoria è usata sia per la lettura che per la scrittura. Quando il PC contiene un numero d'indirizzo, i segnali elettrici che corrispondono agli 1 e agli 0 di quell'indirizzo appaiono su un insieme di collegamenti, chiamati globalmente il "bus degli indirizzi", che collega la MPU all'intera memoria, sia la RAM che la ROM. Il numero che si trova nel PC selezionerà un byte nella memoria, il byte che si trova a quel numero d'indirizzo. All'inizio di un'operazione di lettura, la MPU emetterà un segnale chiamato segnale di lettura su un'altra linea, e questo obbligherà la memoria a collegare la parte che è stata selezionata ad un altro insieme di linee, il "bus dei dati". I segnali sul bus dei dati, quindi, corrispondono allo schema di 0 e 1 immagazzinato nel byte di memoria selezionato dall'indirizzo contenuto nel PC. Lettura significa che questi segnali sono copiati in un registro all'interno della MPU. Ogni volta che il numero contenuto nel PC cambia, viene selezionato un altro byte di memoria, e questo è appunto il modo in cui la MPU può prelevare

automaticamente i bytes. Quando la MPU è pronta per un altro byte, il PC viene incrementato e un altro segnale di lettura viene emesso. Analogamente, per un'operazione di scrittura, il PC contiene il numero d'indirizzo, i segnali sul bus degli indirizzi selezionano il byte corretto nella memoria, e un altro registro dello Z80 divide i suoi segnali elettrici con le linee del bus dei dati, per cui la memoria è obbligata a copiare. Dopo aver fatto questo, il numero contenuto nel PC viene di nuovo incrementato per l'azione successiva.
Ci sono altri modi per cambiare il numero contenuto nel PC, ma per il momento sorvoleremo questa parte e passeremo ad esaminare un altro registro, l'*accumulatore*. L'accumulatore di un microprocessore è il principale registro "operativo" della MPU. Ciò significa che lo userete normalmente per immagazzinare un qualsiasi numero che desiderate trasferire da qualche parte, o addizionarlo ad un altro numero o eseguire un'altra operazione. Il nome accumulatore deriva dal modo in cui opera il registro. Se avete un numero contenuto nell'accumulatore e addizionate ad esso un altro numero, anche il risultato sarà immagazzinato nell'accumulatore. Per fare un parallelo con il BASIC, usiamo una variabile A e scriviamo la riga:

```
A=A+N
```

dove N indica una variabile numerica. Il risultato della riga BASIC è di aggiungere N al vecchio valore di A e di rendere A uguale a questo nuovo valore. Il vecchio valore di A viene perciò perduto. L'accumulatore agisce nello stesso modo, con l'importante differenza che l'accumulatore dello Z80 non può immagazzinare un numero maggiore di 255 (decimale).
Lo Z80 ha solo un registro accumulatore, denominato A. La sua importanza sta nel fatto che viene usato molto più degli altri registri, perché molte azioni possono essere eseguite più rapidamente, più convenientemente o talvolta unicamente nell'accumulatore. Quando leggiamo un byte dalla memoria, di solito lo mettiamo nell'accumulatore. Quando dobbiamo effettuare un'operazione logica od aritmetica, sarà svolta normalmente nell'accumulatore ed anche il risultato sarà immagazzinato nell'accumulatore.
A differenza dei primi microprocessori, lo Z80 ha un gran numero di altri registri, parecchi dei quali possono essere usati praticamente nello stesso modo dell'accumulatore, ma nessuno di questi ha una così vasta gamma di possibilità.

METODI D'INDIRIZZAMENTO

Quando programmiamo solo in BASIC, non dobbiamo preoccuparci affatto degli indirizzi di memoria, a meno che non usiamo le funzioni PEEK

o POKE. Il compito di trovare dove siano immagazzinati i bytes spetta al sistema operativo della macchina. Quando viene assegnato un valore ad una variabile, come, ad esempio, in una riga di programma tipo:

10 N%=12

non dobbiamo preoccuparci di sapere dove sia memorizzato il numero 12 o sotto quale forma. Analogamente, quando aggiungiamo la riga:

20 K%=N%

non dobbiamo preoccuparci di sapere dove sia stato memorizzato il valore di N% o dove verrà memorizzato il valore di K%. Se ricordate il nostro paragone con la costruzione di un muro, sarà logico supporre di dover specificare, nella programmazione in codice macchina, ogni numero che usiamo o, in alternativa, l'indirizzo a cui si trova memorizzato il numero. Questo modo di ottenere un numero, di trovare la posizione a cui si trova memorizzato, è detto "metodo di indirizzamento". Ciò che rende particolarmente importante la scelta di un metodo d'indirizzamento è il fatto che *ogni* diverso metodo d'indirizzamento richiede, per *ogni* comando, un diverso numero di codice. Cioè, ogni comando esiste in parecchie versioni, diverse l'una dall'altra, con un diverso codice per ogni metodo d'indirizzamento. Una lista di tutti i metodi d'indirizzamento dello Z80, in questo momento sarebbe assai inopportuna, e per questa ragione è stata inclusa nell'Appendice F. Ciò che faremo qui sarà di vedere degli esempi di alcuni metodi d'indirizzamento ed il modo in cui vengono scritti in linguaggio assembly.

IL LINGUAGGIO ASSEMBLY

Cercar di scrivere il codice macchina direttamente, come insieme di numeri, è un procedimento molto difficile e suscettibile di errori dall'inizio alla fine. Il modo più utile di cominciare a scrivere un programma è di scriverlo utilizzando il *linguaggio assembly* (o linguaggio assembler). Esso è un insieme di comandi rappresentati da parole abbreviate, chiamate *mnemoniche*, e di numeri, che sono i dati o i numeri d'indirizzo. I numeri possono essere esadecimali o decimali, purché siano forniti al calcolatore nella forma corretta. Ogni riga di un programma in linguaggio assembly indica un'azione del microprocessore, e questo insieme di istruzioni viene poi "assemblato" in codice macchina, da qui il nome. In questo modo, è la macchina ad eseguire tutte quelle azioni di "traduzione" che sono così noiose per l'uomo. Il programmatore dovrà solo pensare al programma e scriverlo in questo linguaggio assembly. C'è un diverso linguaggio assembly per ogni tipo di microprocessore.
Ogni riga di un programma in linguaggio assembly deve specificare l'azione e i dati o l'indirizzo necessari per eseguire quell'azione, nello

stesso modo in cui, quando usiamo TAB in BASIC, dobbiamo completare il comando con un numero. Come il BASIC, il linguaggio assembly deve essere scritto correttamente (con la corretta sintassi). La parte del linguaggio assembly che specifica ciò che deve essere fatto è chiamata *operatore,* e la parte che specifica l'oggetto dell'azione è chiamata *operando.* Alcune istruzioni non richiedono un operando, ne esamineremo qualcuna più avanti.
Un esempio chiarirà meglio le cose. Consideriamo questa riga in linguaggio assembly:

```
LD A,12H
```

L'operatore è LD, abbreviazione di LOAD (caricare), e indica che un byte deve essere copiato da una posizione ad un'altra. L'operando è formato da due parti, A e 12H. La A significa che l'accumulatore A deve essere caricato con un byte. L'altra parte dell'operando è 12H, dove H sta ad indicare che 12 è un numero esadecimale. È stato usato un numero di un solo byte per mostrare il metodo d'indirizzamento che deve essere impiegato, un metodo chiamato "indirizzamento immediato".
L'intera riga, quindi, avrà l'effetto di porre il numero 12H nel registro A. È l'equivalente, in codice macchina, dell'istruzione BASIC:

```
A=&H12
```

se immaginate che la memoria che contiene il numero, denominata A, si trova entro il microprocessore invece che nella memoria RAM.
Un comando come LD A,12H usa *un indirizzamento immediato,* perché il byte che viene caricato nell'accumulatore deve essere immagazzinato nel byte di memoria *il cui indirizzo segue immediatamente quello del byte d'istruzione.* C'è un numero di codice per indicare LD, A e questo byte è 3EH, per cui la sequenza esadecimale 3E12H in memoria rappresenta l'intero comando LD A,12H. È molto più facile ricordarsi il significato di LD A,12H piuttosto che interpretare i numeri 3EH e 12H immagazzinati in memoria: ecco perché usiamo il linguaggio assembly quanto più possibile.
L'indirizzamento immediato può essere conveniente, ma vi costringe all'uso di un numero definito. È come programmare in BASIC:

```
N=4*12+3
```

invece che

```
N=A*B+C
```

Nel primo esempio, N può essere solo 51, e potevamo anche scrivere: N=51. Il secondo esempio è molto più flessibile, e il valore di N dipende da quale valore scegliamo per le variabili A, B, e C.
Quando un programma in codice macchina è contenuto nella RAM, i

numeri caricati con questo metodo d'indirizzamento immediato possono essere cambiati, se dobbiamo cambiarli, solo cambiando il programma. Quando il programma è nella ROM, invece, non è possibile nessun cambiamento, e questa è un'altra ragione per sentire la necessità di altri metodi d'indirizzamento. Uno di questi metodi è l'*indirizzamento esteso*. L'*indirizzamento esteso* (talvolta chiamato "indirizzamento assoluto") usa come operando un indirizzo completo di due bytes. Questo crea parecchio lavoro per lo Z80, perché, quando ha letto il codice per l'operatore, dovrà leggere altri *due* bytes per trovare l'indirizzo di memoria a cui è immagazzinato il dato. Dovrà poi mettere questo indirizzo nel PC, leggere il byte di dati, eseguire l'operazione e poi immettere il successivo corretto indirizzo nel PC. La Fig. 4.1 mostra, in un diagramma, ciò che deve essere fatto.

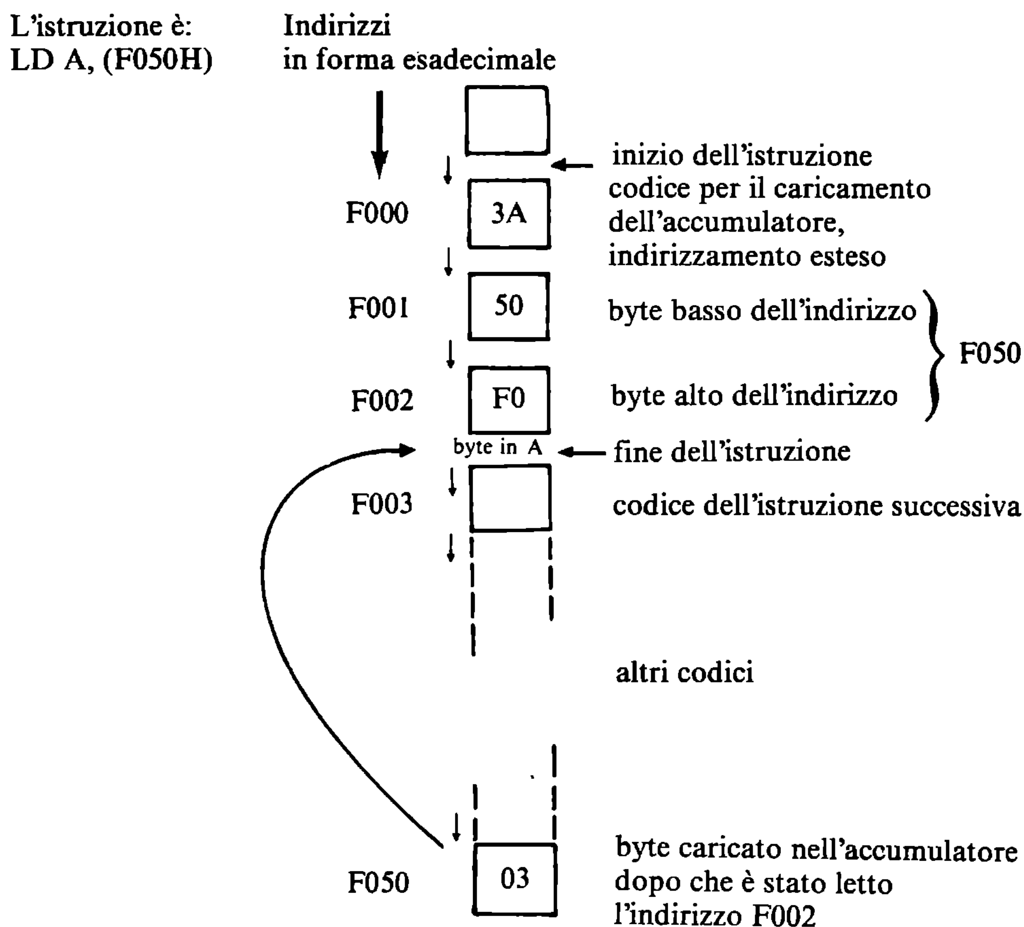


Fig. 4.1. Come funziona il metodo d'indirizzamento esteso (chiamato anche assoluto o diretto).

Un'operazione ad indirizzamento esteso è perciò molto più lenta nell'esecuzione di una ad indirizzamento immediato, ma, poiché ogni byte può essere immagazzinato all'indirizzo specificato, è facile modificare i dati, se ci occorre. Possiamo anche fare in modo che sia lo stesso programma a modificare i dati!

Prendiamo un esempio di indirizzamento esteso diretto, rappresentato dall'istruzione:

LD A,(EFFEH)

L'operatore è LD, che sta ad indicare che un caricamento deve essere effettuato, e l'operando è formato da A, l'accumulatore, e dall'indirizzo EFFEH. L'ordine di queste due parti dell'operando è importante. Mettendo A davanti a (EFFEH), ordiniamo al computer di caricare ciò che si trova all'indirizzo EFFEH *entro* l'accumulatore A.
Dovete ricordarvi che quanto viene immesso nell'accumulatore A non è EFFEH, che è un indirizzo di due bytes, ma *il byte di dati* che è in memoria a questo indirizzo. L'effetto dell'istruzione completa, quindi, è di porre una copia del byte immagazzinato ad EFFEH nell'accumulatore A dello Z80. Quando l'istruzione è stata completata, l'indirizzo EFFEH conterrà ancora la sua copia del byte, perché la lettura di una memoria non ne cambia assolutamente il contenuto. Sarà invece cambiato il contenuto dell'accumulatore, perché ora sarà il byte che si trovava all'indirizzo EFFEH. In un elenco delle istruzioni dello Z80, questo tipo d'indirizzamento apparirà come LD A,(NN) (o, come è mostrato nell'Appendice E, LD A,(0000)). L'uso delle parentesi davanti all'indirizzo non è solo una questione di sintassi. Le parentesi vogliono dire "contenuto di", e ogni volta che troverete le parentesi nel linguaggio assembly, avranno questo significato. Il significato dell'istruzione LD A,(EFFEH) sarà perciò "carica l'accumulatore con il contenuto dell'indirizzo EFFE esadecimale".
Possiamo anche usare il metodo d'indirizzamento esteso in un comando che immagazzinerà un byte nella memoria. È qui che l'ordine delle parti dell'operando diventa importante. Il comando:

LD (EFFFH) ,A

significa che il byte contenuto nell'accumulatore A deve essere copiato in memoria all'indirizzo EFFFH. Questa azione *cambia* il contenuto di questo indirizzo di memoria, ma l'accumulatore A conterrà sempre lo stesso byte dopo che l'operazione è stata portata a termine. Ancora una volta, le parentesi sono state poste davanti all'indirizzo per indicare "contenuto di" e l'ordine d'indirizzo, virgola, A mostra la direzione della copiatura. L'ordine, come potete vedere dai due precedenti esempi, è *sempre* destinazione, virgola, sorgente. Con l'eccezione dell'indirizzamento immediato, la maggior parte dei comandi LD possono essere scritti in modo da indicare l'una o l'altra direzione di copiatura. Altri microprocessori usano il comando ST (di solito immediatamente seguito dal nome del registro, come STA,STX) per l'azione di copiatura di un byte dall'accumulatore ad un indirizzo di memoria.

INDIRIZZAMENTO INDIRETTO TRAMITE REGISTRO

L'indirizzamento immediato e l'indirizzamento esteso (chiamato anche indirizzamento diretto o assoluto) sono ugualmente utili, ma lo Z80 consente di effettuare anche un altro tipo d'indirizzamento, chiamato *indirizzamento indiretto tramite registro*. Indirizzamento indiretto significa andare ad un indirizzo per trovarne un altro. Sembra molto strano e misterioso, ma in effetti è difficile quanto rivolgersi ad un'agenzia di viaggi per sapere l'indirizzo di un ristorante. Il metodo d'indirizzamento indiretto tramite registro ha questo nome perché un registro (più precisamente, una coppia di registri) sarà usato per contenere l'indirizzo. Questo tipo d'indirizzamento è una particolarità dello Z80 e si trova solo in pochi altri microprocessori.
Nello Z80, dipende dalla capacità di combinare certe coppie di registri ad otto bit e di utilizzarli come se fossero un registro a sedici bit. Questa capacità di usare i registri singolarmente o in coppia, come voi decidete, è proprio una delle caratteristiche che hanno fatto dello Z80 un microprocessore così popolare ed ampiamente usato. I principi della struttura dello Z80, a differenza di altri precedenti tipi di microprocessori, sono stati adottati anche dai più nuovi microprocessori a sedici bit. Imparando il codice macchina dello Z80, quindi, vi preparate anche per il futuro!
Ci sono tre insiemi di queste coppie di registri nello Z80 e, per convenienza, sono denominati HL, BC e DE. I registri singoli sono denominati H, L, B, C, D ed E, ma possiamo combinarli insieme solo in questi tre accoppiamenti. Singolarmente i registri possono essere usati come degli accumulatori, ma con un numero di azioni inferiore rispetto a quelle dell'accumulatore A. Fra queste tre coppie di registri, la coppia HL è quella usata più frequentemente. Possiamo caricare un indirizzo completo di sedici bit nella coppia di registri HL usando un comando scritto in linguaggio assembly, per esempio del tipo:

```
LD HL,EFFFH
```

Questo significa che il byte alto dell'indirizzo, EFH nel nostro esempio, sarà contenuto nel registro H (H dovrebbe ricordare più facilmente High, alto) e il byte basso nel registro L (L per Low, basso). Fra parentesi, potete cambiare o il byte alto o il byte basso, caricando separatamente il registro H o il registro L. Dopo aver messo questo indirizzo nella coppia di registri HL, possiamo utilizzare il byte che si trova all'indirizzo EFFFH con un comando tipo:

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

che significa che l'accumulatore A deve essere caricato con il byte il cui indirizzo si trova in HL. Nel nostro esempio, sarà il byte che si trova all'indirizzo EFFFH. Se cambiassimo l'indirizzo contenuto nei registri

HL, ovviamente caricheremmo un diverso byte da un diverso indirizzo. Nello stesso modo, possiamo facilmente immagazzinare a questo indirizzo un byte dell'accumulatore, invertendo l'ordine delle parti dell'operando. Per esempio, se usiamo:

LD(HL) ,A

allora il byte dell'accumulatore verrà copiato all'indirizzo EFFFH, supponendo che questo sia l'indirizzo ancora contenuto in HL. Ricordate che le parentesi vogliono dire "contenuto di", e che l'ordine delle parti dell'operando è destinazione, sorgente.
Ora, potreste pensare che questo è un modo piuttosto contorto di scrivere un comando come LD A,(EFFFH), ma c'è una differenza importante. Una volta che un numero d'indirizzo è stato immagazzinato nella coppia di registri HL possiamo incrementare o decrementare quell'indirizzo con un'istruzione di un solo byte. Se, per esempio, abbiamo caricato HL con l'indirizzo EFFFH, l'istruzione DEC HL uguaglierà il numero d'indirizzo che si trova in HL a EFFEH, uno in meno. Se usiamo di nuovo LD A,(HL), il caricamento verrà effettuato dall'indirizzo EFFEH, non EFFFH. Se un programma richiede che parecchi bytes siano caricati da un insieme consecutivo di indirizzi, questo metodo permette di eseguire l'azione in *un ciclo (loop),* semplicemente con un'istruzione LD A,(HL) ed un'istruzione DEC HL. Naturalmente, potreste usare INC HL nel ciclo in modo che il numero d'indirizzo contenuto in HL sia incrementato, anziché decrementato. Inoltre, potete incrementare o decrementare H o L separatamente, e potete usare i comandi INC(HL) e DEC(HL), che incrementeranno o decrementeranno il *byte* immagazzinato all'indirizzo contenuto in HL! Ma non andiamo troppo oltre, per il momento, e rimettiamo i piedi per terra!

INDIRIZZAMENTO RELATIVO

L'indirizzamento relativo è uno dei primi e più semplici metodi d'indirizzamento, ma oggi non è usato in molte istruzioni. Indirizzamento relativo significa che l'operando di un'istruzione può essere formato soltanto da un byte, e l'indirizzo che dovrà essere usato viene trovato aggiungendo questo numero (chiamato valore di spostamento o di offset) all'"indirizzo corrente", che è il numero contenuto nel Contatore di Programma. Il numero ottenuto come risultato di questo procedimento di addizione viene quindi usato nel caricamento o in qualsiasi altra operazione specificata. È un po' come le antiche mappe dell'Isola del Tesoro, che specificano "dieci passi a sinistra, tre in avanti, cinque indietro..." e così via. Non saprete dove questo vi porterà finché non sapete da dove partire, ma quando viene utilizzato un indirizzamento relativo in un microprocessore, il punto di partenza è *sempre* l'indirizzo contenuto nel PC. Lo Z80 usa gli

indirizzamenti relativi solo per un piccolo gruppo di comandi, i comandi JR di salto relativo (JR=Jump Relative).
Ogni comando di questo tipo userà un numero *con segno* di un solo byte, come offset. L'uso di un numero con segno di un solo byte significa che possiamo saltare ad un nuovo indirizzo che si trova 127 passi avanti o 128 passi indietro rispetto all'indirizzo attuale. I comandi di salto relativo sono l'equivalente in codice macchina dell'istruzione BASIC GOTO, con la differenza che possono dipendere da una condizione, ad esempio che l'accumulatore contenga zero. È come se ci fosse una sola istruzione BASIC per ottenere l'effetto di:

IF A=0 THEN GOTO...

L'istruzione completa JR è formata dall'operatore JR, e da un operando che può contenere una condizione ed un byte contenente il valore di offset, per indicare, appunto, la lunghezza del salto. In codice macchina, vengono usati solo due bytes per ogni tipo di comando JR. L'uso dell'istruzione senza un assemblatore richiede una certa attenzione (ed esperienza!). Questo perché lo spostamento (offset) viene considerato dallo Z80 come un numero *con segno*, cioè se il bit più significativo è 1 (per il numero decimale 128 o per valori superiori a 128), allora il byte è negativo. Se il byte è negativo, il salto avverrà all'indietro, ad un numero d'indirizzo inferiore a quello del PC. In numeri decimali, quindi, se il valore di offset è 127 o inferiore a 127, il salto è in avanti; se il numero è 128 o superiore a 128, il salto è all'indietro. Dopo un'istruzione di salto, il nuovo indirizzo contenuto nel PC sarà incrementato nel modo consueto, e la normale azione riprenderà da questo nuovo indirizzo. Anche in questo è simile all'azione di GOTO in BASIC.
Quando usiamo l'indirizzamento relativo, la cosa più complicata è di calcolare la grandezza del byte di spostamento (offset). Se usate l'assemblatore programmi ZEN per scrivere i vostri programi in codice macchina, il byte di offset verrà calcolato dall'assemblatore. Ma, se eseguite un assemblaggio "manuale", dovrete calcolarvelo per conto vostro. Il calcolo viene così effettuato:

1) Scrivete l'indirizzo a cui sarà immagazzinato il byte dell'operatore JR. Questo è chiamato "indirizzo sorgente".
2) Scrivete l'indirizzo a cui volete che il programma salti. Questo è chiamato "indirizzo di destinazione".
3) Sottraete l'indirizzo sorgente dall'indirizzo di destinazione e poi sottraete due da questo risultato. Il risultato finale è il valore di offset espresso nella notazione decimale. Se è positivo, usatelo direttamente. Se è negativo, sottraetelo da 256 e usate il risultato di quest'ultima operazione.

Perché sottrarre 2? Questo perché il valore di offset (la lunghezza del

salto) è sempre calcolato dall'indirizzo a cui è posto il byte dell'*operatore,* ma il salto non potrà essere effettuato finché non viene letto il byte dell'operando, poiché è l'operando che contiene il byte di spostamento (offset). Per leggere l'operando, il PC deve incrementarsi e inoltre il PC s'incrementerà *nuovamente* alla fine dell'istruzione. Sottraendo 2, permettiamo che avvengano queste due azioni d'incremento e otteniamo il valore corretto del byte di spostamento. La fig. 4.2 mostra come vengono calcolati alcuni valori di spostamento, positivi e negativi. È spesso più facile, quando scrivete i bytes di un programma assemblato "a mano", contare il numero dei bytes che si trovano tra l'indirizzo sorgente e quello di destinazione. Ricordate che la lunghezza del salto che è possibile effettuare con l'indirizzamento relativo non può essere superiore a 127 passi in avanti o a 128 passi indietro rispetto all'indirizzo contenuto nel PC, poiché questa è la gamma completa di variazione di un byte con segno.

Indirizzo (Decimale)

sorgente	61472	JR	
(S)	61473	..	← byte di spostamento qui (226)
	.		
	.		
	.		
	.		
destinazione	61700		← Salta qui
(D)			

(D)—(S) è 61700—61472=228
Poi sottrarre 2 per ottenere il risultato 226, che è il byte di spostamento

61470 JR
61471 ← byte di spostamento qui (110)

(La destinazione è 61362)

61362—61470 = —108
Sottrai 2 per il risultato —110, che è il byte di spostamento

---

Fig. 4.2. L'indirizzamento relativo effettuato con bytes di spostamento sia positivi che negativi.

INDIRIZZAMENTO INDICIZZATO

L'indirizzamento indicizzato è un metodo particolarmente utile per alcune applicazioni sullo Z80. Il principio è che un registro a sedici bit (uno dei due dello Z80) è usato per contenere un indirizzo, l'indirizzo di un byte nella memoria. Questo indirizzo può essere usato *direttamente, o può essere usato come indirizzo di base.* "Usare un indirizzo di base" significa che possiamo aggiungere un numero all'indirizzo prima di usarlo. Per esempio, supponiamo di avere il numero 9000H immagazzinato in un registro indice. Se vogliamo, possiamo usare questo per specificare l'uso dell'indirizzo 9000H, o in alternativa possiamo aggiungere un numero ad esso. Se aggiungiamo 4, per esempio, possiamo effettuare un caricamento da 9004H o immagazzinare a questo indirizzo. Il comando di caricamento indicizzato per l'accumulatore dello Z80 prende la forma:

LD A,(IX+d) o LD A,(IY+d)

I due registri indice sono denominati IX e IY. Ognuno di essi è un registro a sedici bit, che *non può* essere diviso in due registri a otto bit. Il "d" è il parametro del byte di spostamento (displacement) che è aggiunto all'indirizzo contenuto nel registro indice. L'operazione di caricamento ha luogo da questo numero d'indirizzo finale, ottenuto dall'addizione. L'uso dei registri indice è molto conveniente quando un insieme di valori (come un insieme di "simboli" del BASIC, per esempio) è memorizzato in forma di tabella. Se volete il cinquantacinquesimo simbolo di una tabella che inizia a 3000H, per esempio, usate LD A,(IX+54), supponendo che il registro indice IX sia stato precedentemente caricato con 3000H. Questo sarà stato fatto con il comando LD IX,3000H. Se vi chiedete perché usiamo IX+54 per ottenere il 55esimo simbolo, ricordate che l'indirizzo IX contiene il primo simbolo, IX+1 contiene il secondo e così via. Normalmente, i registri IX e IY, se devono essere usati in un programma, saranno stati caricati nella prima parte del programma e il contenuto verrà lasciato immutato. Notate che questo metodo è diverso dall'uso dell'indirizzamento indiretto tramite registro, tipo LD A,(HL), perché non viene stabilito di lasciare HL immutato e di aggiungere un numero all'indirizzo di HL solo quando viene usato. I registri indice sono ampiamente usati nelle routines ROM del vostro computer MSX, come avrete immaginato, ma non compaiono spesso nei brevi programmi di questo libro.

GLI ALTRI REGISTRI DELLO Z80

Abbiamo già menzionato alcuni registri dello Z80, e un breve richiamo può essere utile. Il PC è il registro d'indirizzamento, che tiene conto dell'indirizzo di ogni byte d'istruzione e di ogni byte di dati di un programma. È il registro del "dove siamo ora"? dello Z80. Quando un pro-

gramma in codice macchina deve essere eseguito, questo è fatto ponendo l'indirizzo del primo byte d'istruzione del programma nel PC. Il resto è automatico; quindi, dando per scontato che il programma sia stato scritto correttamente, la MPU procederà oltre. Per far girare un programma in codice macchina, abbiamo quindi bisogno di un comando che immetta l'indirizzo corretto nel PC: ne esamineremo i metodi più avanti.
L'accumulatore è il registro dove viene eseguita la maggior parte del lavoro. È così importante che dedicheremo molte pagine del prossimo capitolo alle azioni che possono essere portate a termine nell'accumulatore. Ci sono altri sei registri ad un solo byte, denominati B, C, D, E, H e L che possono essere usati nello stesso modo in cui usiamo l'accumulatore, sebbene nessuno di essi offra una così vasta gamma di azioni. Inoltre come abbiamo visto, questi registri possono essere raggruppati in HL,BC e DE per immagazzinare numeri d'indirizzo completo a sedici bit. È possibile eseguire un limitato numero di operazioni aritmetiche a sedici bit nella coppia di registri HL. Bisogna sottolineare che nessun costruttore di computer ha però mai sostenuto che lo Z80 sia un chip a sedici bit per via di questo!
Abbiamo visto brevemente anche i registri indice, IX e IY, ma con questi ultimi non abbiamo esaurito l'elenco dei registri dello Z80. Vi è un altro registro a sedici bit, denominato SP, abbreviazione di *Stack Pointer*, Puntatore di Stack. Come vedremo, lo "stack" è una parte di memoria usata per la memorizzazione temporanea durante l'esecuzione di un programma, e il registro del Puntatore di Stack viene usato per tener nota degli indirizzi contenuti in questo "stack". Solo per illustrarvi come viene usato, vi siete mai chiesti come mai un GOSUB in BASIC potesse andare ad una nuova riga, ma poi ritornare, con il comando RETURN, alla posizione corretta? Questo accade perché quando viene eseguita l'istruzione GOSUB, l'indirizzo corretto per il ritorno al programma principale è immagazzinato nella memoria dello Stack e il registro del puntatore di stack viene usato per far riferimento a questo indirizzo. Quando la subroutine è terminata, il puntatore di stack indica quale degli indirizzi contenuti nello stack debba essere usato per ottenere il corretto indirizzo di ritorno nel PC. Questa è una versione semplificata di quanto accade: vedremo l'azione molto più dettagliatamente nel Capitolo 7.
Ci sono anche due registri che usiamo molto raramente nei nostri programmi: *il registro del vettore di Interrupt (I) e il registro di Rinfresco (R).* I loro usi sono piuttosto specializzati. A meno che non vogliate scrivere programmi per cui il vostro computer deve essere controllato da dispositivi esterni come penne luminose o sensori, il registro di Interrupt potrà difficilmente interessarvi. Il registro R è usato dallo Z80 per "rinfrescare" la memoria RAM del computer. I computer moderni usano un tipo di memoria RAM chiamata RAM dinamica. Questa costa molto meno di altri tipi di memoria, ma perderà i suoi segnali dopo breve tempo se non

viene attivata molte volte al secondo da un segnale di "rinfresco". Lo Z80 fornisce automaticamente questo segnale. I computer che usano MPU di vecchia progettazione ricevono questo segnale di rinfresco da altri chip. Il contenuto del registro di rinfresco è talvolta usato nei programmi in codice macchina come sorgente di numeri casuali. *Un importante registro rimane, invece, il registro di stato.*

IL REGISTRO DI STATO

Il *registro di "stato"*, chiamato spesso il registro dei flag (e di solito indicato come Registro F) non è proprio un registro come gli altri. Non potete *fare* niente con i bit di questo registro se non "testarli", e la loro combinazione non equivale neppure ad un numero. Il registro di stato è usato come una specie di taccuino elettronico. Ogni bit del registro (ce ne sono otto) è usato per registrare che cosa è avvenuto al punto precedente del programma. Se il passo precedente era una sottrazione il cui risultato, contenuto nel registro A, è stato zero, allora uno dei bit del registro di stato passerà dal valore 0 al valore 1, appunto per portare questo all'attenzione della MPU. Se addizionate un numero al numero contenuto nell'accumulatore, e il risultato è formato da nove bit anziché otto (Fig. 4.3), allora un altro dei bit del registro di stato è "settato", cioè posizionato da zero a 1.

Numero contenuto nell'accumulatore	10110110
Numero addizionato	11000101
Risultato	101111011

Il risultato è di nove bit, mentre l'accumulatore può contenerne solo otto.

Il bit più significativo è trasferito al flag di carry del registro di stato.

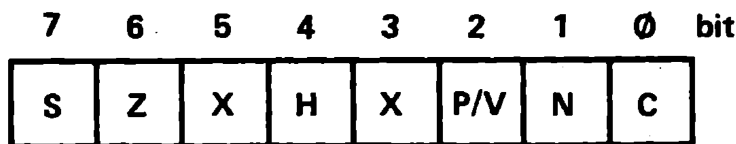
L'accumulatore ora contiene 01111011

Il bit di carry è posizionato a 1.

Fig. 4.3. Perché occorre il flag di carry.

Se il bit più significativo di un registro passa da 0 a 1 (il che potrebbe indicare un numero negativo), allora viene posizionato a 1 un altro dei bit stato. Ogni bit del registro di stato viene quindi usato per prendere nota di ciò che è appena avvenuto, in particolare di ciò che è avvenuto *nell'accumulatore*. Ciò che rende così importante il registro di stato è che potete far dipendere delle istruzioni di "branch" * dalla condizione che un bit di stato sia posto a 1 o sia posto a 0.

* Le istruzioni di salto (jump) condizionato, dipendenti cioè da una scelta logica nel programma, sono spesso chiamate "branch" (diramazione). Il nome deriva dall'analogia con l'albero, poiché implica una biforcazione nella rappresentazione del programma. (N.d.T.).



Flag di riporto (carry) — è posto a 1 se c'è un riporto, o un prestito, in un'operazione aritmetica.

N-Flag — è posto a 1 per le sottrazioni

Z-Flag — è posto a 1 se alcune operazioni danno zero come risultato

S-Flag del segno — è posto a 1 se il risultato è negativo

C—flag di riporto (carry)
N—flag di addizione-sottrazione
P/V—flag di parità
H—flag di riporto interno (half carry)
Z—flag di zero
S—flag del segno

(Gli altri flag hanno usi speciali)

Fig. 4.4. I bit del registro dei flag, o registro di stato. Di questi bit, soltanto tre, N, Z e C, sono i più ampiamente usati nei programmi.

La fig. 4.4. mostra come siano disposti i bit del registro di stato dello Z80. Di questi bit, i numeri 0, 6 e 7 sono quelli che userete di più all'inizio della vostra carriera di programmatori in codice macchina. L'uso degli altri è assai più specialistico di quanto sia, per il momento, necessario. Il bit 0 è il flag di Carry (riporto). È posizionato a 1 se in un'addizione c'è stato un riporto del bit più significativo di un registro. Se non c'è riporto, il bit rimane a 0. Quando viene eseguita una sottrazione (o un'operazione simile come il confronto), questo bit sarà usato per indicare se vi è stato un "prestito". Per alcuni scopi, può essere usato anche come nono bit dell'accumulatore, soprattutto nelle operazioni di rotazione e di spostamento (o scorrimento: shift), nelle quali i bit di un byte sono tutti spostati di un posto. (Fig. 4.5).

Il flag di Carry è usato per tutte le operazioni di addizione e sottrazione dello Z80, comprese anche quelle operazioni che non usano l'accumulato-

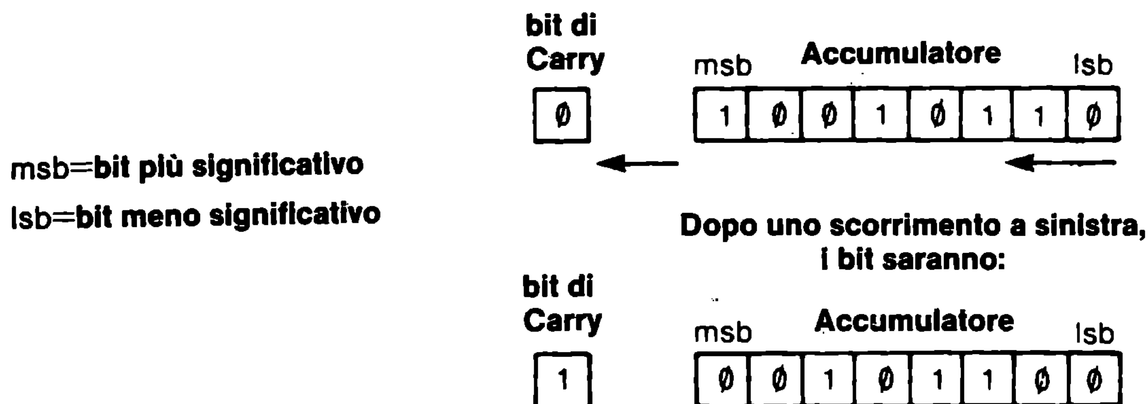


Fig. 4.5. L'uso del bit di carry in un'operazione di scorrimento, in cui tutti i bit di un byte sono spostati di una posizione a sinistra.

re. C'è però una cosa a cui dovete fare attenzione. Tutte le operazioni eseguite nell'accumulatore, ad eccezione del caricamento (in una direzione o nell'altra) modificheranno i flag, ma NON lo faranno molte operazioni svolte negli altri registri. Questa è la ragione per cui spesso si usa l'accumulatore invece di un altro registro che può svolgere la stessa funzione.
Il flag di Zero è il bit 6 del registro di stato. È posizionato a 1 se il risultato della precedente operazione era esattamente zero, ma sarà posizionato a 0 in caso contrario. È un utile modo di individuare se due bytes sono uguali, sottraendoli l'uno dall'altro, e se il flag di zero è posizionato a 1, allora i due bytes erano uguali. L'azione CP (ComPare=Confronto) posizionerà a 1 o a 0 questo flag *senza* eseguire effettivamente l'azione di sottrazione. L'istruzione CP può essere usata solo con l'accumulatore, e quindi i flag saranno sempre modificati. Il flag del Segno, il numero 7, è posto a 1 se il numero che risulta da un'azione svolta in un registro ha uguale a 1 il suo bit più significativo. Questo è il tipo di numero che potrebbe essere negativo se stiamo lavorando con numeri con segno. Questo bit è perciò ampiamente usato quando lavoriamo con numeri con segno.
Come molte altre MPU, lo Z80 non consente al programmatore di modificare facilmente il contenuto del registro di stato. Potete porre a 1 il flag di carry con il byte d'istruzione SCF (Set Carry Flag) e potete assicurarvi che il flag di carry sia posto a 0 usando XOR A (che azzera anche l'accumulatore). È anche possibile leggere il contenuto del registro dei flag nei registri C, L o E immagazzinando A ed F sullo stack e poi leggendo in BC, HL o DE. Normalmente, non avrete mai bisogno di modificare il contenuto del registro di stato.Molto raramente vi può interessare di sapere che cosa sia contenuto nel registro di stato: per voi esso è importante soprattutto perché controlla i salti. Per fare un altro paragone con il BASIC, supponiamo di aver programmato:

```
100 IF A=0 THEN 300
```

Un salto alla riga 300 verrà effettuato se la variabile A assume il valore zero. Quando il programma viene eseguito, possiamo non sapere in ogni particolare istante il valore di A, ma sappiamo che il salto verrà effettuato se la variabile A diventa 0. La versione in codice macchina di questa azione è la seguente:

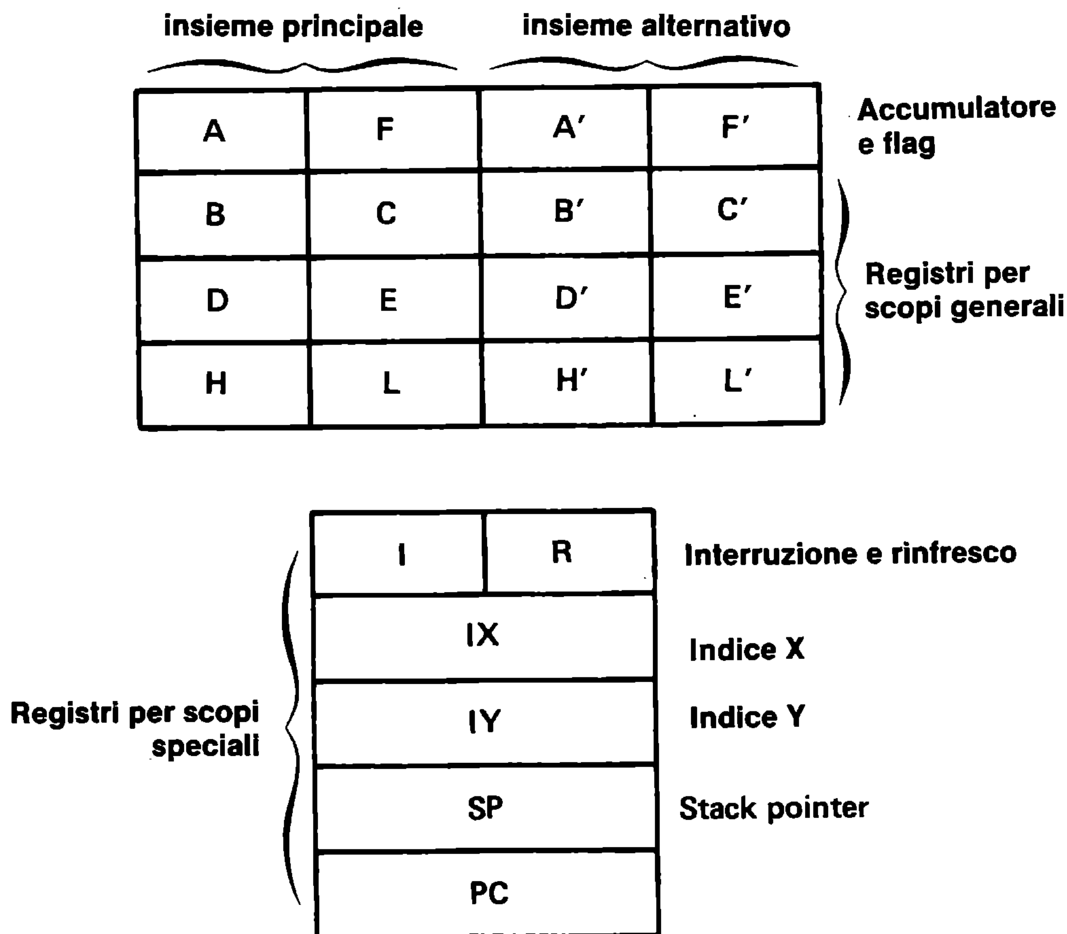
```
JR Z,d
```

dove JR è l'operatore, che indica il salto relativo, e l'operando è formato da Z e dal parametro d. Z (deve essere la *prima* parte dell'operando) significa che il flag di Zero deve essere testato. Il parametro "d" indica uno spostamento, dovrete scrivere qui un numero di un solo byte. Se il salto viene eseguito (quando l'accumulatore contiene il valore zero), il valore di spostamento sarà addizionato al numero contenuto nel PC per

dare il nuovo indirizzo a cui dovrà saltare il programma. Se Z=0, nel caso in cui l'accumulatore *non* contenga uno zero, il valore di spostamento sarà ignorato e il PC sarà incrementato nel modo consueto. Quando il microprocessore esegue quest'azione, il singolo byte di codice che rappresenta la parte Z dell'operando farà sì che lo Z80 controlli il flag di zero nel registro di stato.
Una particolarità dello Z80 è che solo certe azioni, principalmente quelle che modificano l'accumulatore, modificheranno i flag del registro di stato. Tuttavia, le operazioni di caricamento e di immagazzinamento *non* modificano mai, in alcun modo, i flag, per cui un flag che è stato posto a 1 prima di un'operazione di caricamento o d'immagazzinamento, sarà ancora posizionato a 1 dopo quell'operazione. Questo può talvolta dimostrarsi assai utile. Supponiamo, ad esempio, di avere questa parte di programma in linguaggio macchina:

```
LD A,(HL)
DEC A
LD A,(DE)
JR Z,d
```

I flag non vengono modificati dal passo LD A,(HL). Se il passo DEC A (byte di decremento nell'accumulatore) ha fatto diventare zero il contenuto dell'accumulatore, allora il flag di zero sarà posto a 1 a questo punto. Il flag di zero non sarà invece modificato dal passo LD A,(DE), anche se questo passo pone un nuovo byte nell'accumulatore, per cui non conterrà più zero. Il salto previsto dal passo JR Z,d (parametro indicante il valore di spostamento) verrà quindi effettuato, anche se l'accumulatore ora contiene un byte diverso da zero. Questo può essere assai utile perché può risparmiare la ripetizione di un passo di caricamento. Ma ne potrete apprezzare l'utilità fra un po' di tempo, quando avrete un po' più d'esperienza del codice macchina.
Per riassumere tutte le informazioni sui registri, la Fig. 4.6 mostra una "mappa" di tutti i registri dello Z80. Il registro di stato è indicato con F, per convenzione, e può essere denominato anche registro dei flag. Sono usati tutti e due i nomi, ed ecco perché ho continuato ad indicarlo come registro di stato in questo capitolo. I registri sono rappresentati sia come insieme principale che come insieme alternativo e, in ogni gruppo, i registri sono rappresentati in coppie. I raggruppamenti HL, BC e DE vi sono già familiari, ma vedrete che anche l'accumulatore A e il registro dei flag F sono raggruppati come AF.
Questo perché ci sono alcune operazioni che considerano questi due registri come se fossero un registro a sedici bit. Ci arriveremo più avanti. Nel frattempo, una parola sull'insieme alternativo. Normalmente, usiamo l'insieme principale dei registri dello Z80, ma ognuno dei registri principali è duplicato nell'insieme alternativo. Non potete usare l'insieme principa-



I registri principali e quelli alternativi possono essere scambiati dalle istruzioni EX AF,AF' e EXX.

Fig. 4.6. Una "mappa" dei registri dello Z80.

le e quello alternativo contemporaneamente; dovete passare all'uno o all'altro. Sono usate due istruzioni per far questo, EX AF,AF' e EXX. L'istruzione EX AF,AF' scambia i registri A e F dell'insieme principale con i registri alternativi, ma lascia immutati gli altri registri. Se, ad esempio, avete usato A e F, l'istruzione EX AF,AF' vi permette di usare i registri alternativi A e F, mentre i registri originali A ed F rimangono inalterati, con lo stesso contenuto che avevano prima dello scambio. Un'altra istruzione EX AF,AF' effettuerà un altro scambio. Gli altri registri B, C, D, E, H ed L saranno scambiati con l'istruzione EXX. Oltre questi registri, vi sono i registri speciali I, R, IX, IY, PC e SP. Ad eccezione di I ed R, questi sono tutti registri a sedici bit utilizzati per memorizzare indirizzi.

CAPITOLO 5

Le azioni dei registri

LE AZIONI DELL'ACCUMULATORE

In questo capitolo, vedremo più dettagliatamente le azioni dei registri dello Z80. A questo punto, sarete diventati impazienti per il fatto che non avete ancora niente da provare sul vostro computer. Ma il linguaggio macchina non si può suddividere in facili stadi di apprendimento. Non ha senso fare un esempio da provare sul computer, se tutto il procedimento è un mistero e sarà così finché non saprete che cosa state facendo. Su, coraggio, ci siete quasi!
Poiché l'accumulatore è il principale registro ad un solo byte, possiamo fare un elenco delle sue azioni e poi descriverle dettagliatamente, dal momento che quanto vale per l'accumulatore dello Z80 sarà anche un'utile guida per gli altri registri da un byte. Di tutte le azioni dell'accumulatore, il semplice trasferimento di un byte è senza dubbio la più importante. Per esempio, non eseguiamo nessuna operazione aritmetica sui numeri di codice ASCII, per cui le principali azioni che si eseguono su questi bytes sono il caricamento e l'immagazzinamento. Carichiamo l'accumulatore con un byte copiato da un indirizzo di memoria, e lo immagazziniamo in un altro. Pochissimi sistemi operativi permettono di eseguire il trasferimento diretto di un byte da un indirizzo all'altro, per cui viene usato quasi esclusivamente il metodo di caricare da un indirizzo e di memorizzare ad un altro, anche se può sembrare piuttosto lento. Nel linguaggio assembly, usando un indirizzamento diretto, avrà questa forma:

```
LD A, (SORG)
LD (DEST),A
```

La prima riga copia un byte posto all'indirizzo "SORG" (sorgente) nell'accumulatore. Lo stesso byte viene poi copiato all'indirizzo "DEST" (destinazione) nella riga successiva. Ricordatevi dell'ordine dei termini dell'operando. Ho usato le parole SORG in luogo degli effettivi numeri d'indirizzo per due ragioni. La prima è che vi aiuta a ricordare che questi possono essere due qualsiasi numeri d'indirizzo. La seconda è dovuta al modo in cui si usa il linguaggio assembly, utilizzando parole come "label" (letteralmente: etichetta) ogni qualvolta sia possibile, piuttosto che numeri definiti di indirizzo. Se usate un numero, siete vincolati ad esso, ma se usate una parola, potete assegnarle un numero soltanto quando dovete

effettivamente inserire il programma nel computer. Questo rende molto più semplice la stesura e la modifica di un programma. È un po' come usare le variabili al posto delle costanti in BASIC. Per esempio, supponiamo di aver un programma BASIC che deve eseguire parecchi calcoli dell'IVA. Immettete istruzioni tipo X=A*.15?
Certamente no, se sapete cosa fare! Dovreste usare X=A*IV. In questo modo, la variabile IV contiene il valore dell'IVA. Ci sono due valide ragioni per procedere in questa maniera: una variabile occupa meno spazio in memoria, inoltre, basta assegnarle un valore solo una volta. Se la percentuale di IVA cambia al .18, tutto ciò che dovrete fare sarà di cambiare la riga IV=.15 in un'altra che indichi IV=.18. L'uso di una "label" (un nome di variabile) è assai più pratico dell'uso di un numero specifico, in questo esempio, e lo stesso vale anche per il linguaggio assembly.
L'altro importante gruppo di azioni dell'accumulatore è il gruppo aritmetico e logico, che comprende addizioni, sottrazioni, istruzioni INC, DEC, AND, OR e XOR. Possiamo aggiungerle alle azioni di scorrimento (SHIFT) e rotazione (ROTATE), che abbiamo esaminato brevemente nel capitolo precedente. La formazione di questo gruppo è però arbitraria, e molti testi vi inseriscono anche l'istruzione CP. Cominceremo con le azioni di addizione e sottrazione. Lo Z80 ha due tipi di questi comandi: la differenza fra l'uno e l'altro tipo è l'uso del bit di carry del registro F. Quando viene usato l'accumulatore, come accade nella maggior parte delle operazioni, l'accumulatore contiene un byte prima che l'azione incominci. Viene quindi addizionato un altro byte,da un registro diverso o da un indirizzo di memoria, e anche il *risultato* dell'azione (addizione o sottrazione) viene memorizzato nell'accumulatore. Utilizzando, per il momento, solo l'indirizzamento immediato, l'effetto di ADD A, 0AH sarà di aggiungere il numero dieci (0AH) al contenuto dell'accumulatore. Il flag di carry è ignorato durante l'esecuzione, ma potrebbe essere posizionato *dopo l'addizione.* Per esempio, se l'accumulatore conteneva 2BH (43 decimale) prima dell'istruzione ADD A, 0AH, dopo l'esecuzione l'accumulatore conterrà 2BH (53 decimale), e il bit di carry viene posto a 0, perché non c'é stato nessun riporto. D'altra parte, se l'accumulatore conteneva il numero F9H (249 decimale), allora l'effetto di ADD A, 0AH sarebbe stato di lasciare 03H nell'accumulatore e di posizionare a 1 il bit di carry. Perché? Perchè F9H+0AH=103H, e poiché l'accumulatore può contenere solo un byte (due cifre in meno di questo numero esadecimale), allora immagazzinerà 03H e l'"1" farà sì che il bit di carry sia "settato" (posto a 1). Ancora una volta, non importa se il bit di carry era posto a 1 o no *prima* dell'addizione.
È molto diverso se usiamo l'altro comando per l'addizione, ADC. ADC significa "addiziona con riporto" (add with carry), e addizionerà sempre il numero del flag di carry all'altro numero. Se, ad esempio, vi è il numero

2BH nell'accumulatore e usiamo ADC A, 0AH, allora il risultato sarà 35H, come prima, se il bit di carry era posto a 0 prima dell'addizione. Se il bit di carry era posto a 1, il risultato dell'addizione sarà, invece 36H, uno in più. La ragione di questi due tipi istruzione per l'addizione è che talvolta abbiamo bisogno di addizionare un riporto (aritmetica su numeri che usano più di un byte, per esempio), ma, altrettanto spesso, non ne dobbiamo far uso (aritmetica con numeri di un solo byte, per esempio). Avendo a disposizione questi due tipi di istruzione, non dobbiamo preoccuparci di sapere in anticipo se il bit di carry è posto a 1 o no. Molti microprocessori che non usano questi due comandi, devono avere delle istruzioni per posizionare a 1 o a 0 il bit di carry. Le istruzioni di sottrazione SUB (ignora il carry) e SBC (usa il carry), si comportano nello stesso modo. Quando viene usata l'istruzione SBC, viene sottrattto anche il valore del carry. Tutte queste istruzioni aritmetiche esistono per vari metodi d'indirizzamento, tuttavia ho usato l'indirizzamento immediato per maggior semplicità. ADD e SBC possono essere usate anche con alcuni registri a sedici bit, particolarmente HL. In tal caso, anche il risultato sarà contenuto nello stesso registro a sedici bit.
I comandi INC e DEC possono essere usati per incrementare o decrementare uno qualsiasi dei registri da un byte, quindi possiamo usare istruzioni tipo INC A, DEC C, INC H e così via. Queste istruzioni non modificano il bit di carry ma modificheranno *sempre* i bit Z e S del registro dei flag. INC e DEC possono essere usate anche con (HL), e con i registri indice, per incrementare o decrementare un byte contenuto in memoria, anziché nei registri. I flag Z e S saranno modificati da queste istruzioni, ma non il flag C. Vi sono i comandi INC e DEC anche per le coppie di registri (HL, BC, DE, IX, IY e SP). Questi passi INC e DEC *non* modificano nessuno dei flag del registro di stato.
Le istruzioni logiche AND, OR, XOR operano *sempre* sul byte contenuto nell'accumulatore, ed anche il risultato dell'operazione viene lasciato nell'accumulatore. Per esempio, supponete che l'accumulatore contenga il byte 3EH, equivalente al numero binario 00111110, e che il registro C contenga il byte ABH, 10101011 binario. Il risultato di AND C sarà perciò l'AND di 3EH e di ABH, che dà 2AH, 00101010 binario, e questo numero sarà memorizzato nell'accumulatore. Ritornate al Capitolo 1 se avete dimenticato come si svolge l'azione di AND. Le azioni OR e XOR usano i registri in modo analogo. Questa volta, ho preso un esempio che utilizza l'indirizzamento di registri anziché l'indirizzamento di memoria, ma tutte queste istruzioni possono usare l'intera gamma dei metodi d'indirizzamento.

SCORRIMENTO (SHIFT) E ROTAZIONE (ROTATE)

Gli effetti delle istruzioni di scorrimento (shift) e rotazione (rotate) dello

Z80, con la mnemonica del linguaggio assembly, sono mostrate nella Fig. 5.1 e nella Fig. 5.2.

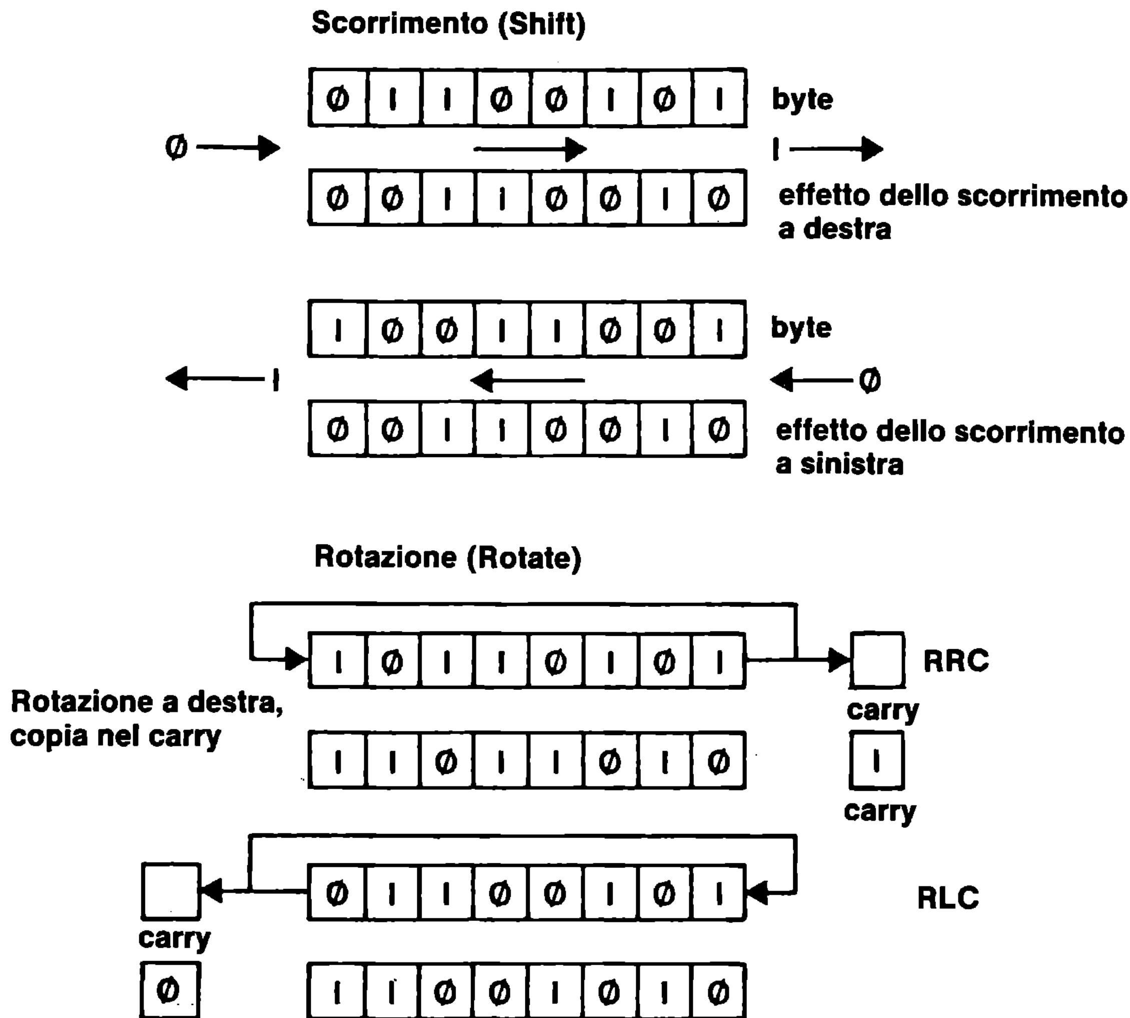


Fig. 5.1 L'effetto delle istruzioni di scorrimento (SHIFT) e rotazione (ROTATE). Questa è una versione semplificata, poiché queste istruzioni dello Z80 spesso coinvolgono sia il bit di carry che il contenuto dei registri.

Dopo uno scorrimento, un registro perde uno dei bit immagazzinati, ed esattamente quello posto all'estremità che è stata spostata. La maggior parte degli scorrimenti — ad eccezione dello scorrimento aritmetico a destra (SRA) — fanno sì che il registro prenda il valore zero all'estremità opposta. Il bit di carry viene usato come nono bit dell'accumulatore in tutte queste traslazioni. L'azione di scorrimento può essere eseguita sull'accumulatore, su un byte contenuto in uno qualsiasi degli altri otto registri a scopi generali, o su un byte immagazzinato in memoria e indirizzato da (HL), o usando l'indicizzazione. L'effetto dello scorrimento su un numero binario contenuto nel registro è di moltiplicare il numero per due se lo scorrimento è a sinistra, o di dividerlo per due se lo scorrimento è a destra (Fig. 5.3).

SLA	Scorrimento a sinistra, il bit più significativo nel flag di carry
SRA	Scorrimento a destra, il bit più significativo è immutato, il bit meno significativo nel flag del carry
SRL	Scorrimento a destra, il bit più significativo è posto a zero, il bit meno significativo nel flag di carry
RLD e RRD	non sono mai usati
RLCA	Rotazione a sinistra dell'accumulatore, il bit 7 è copiato nel flag di carry
RLA	Rotazione a sinistra dell'accumulatore, compreso il carry
RRCA	Rotazione a destra dell'accumulatore, il bit meno significativo nel carry
RRA	Rotazione a destra dell'accumulatore, compreso il carry
RLC	Rotazione a sinistra del registro, il bit più significativo è copiato nel carry
RL	Rotazione a sinistra del registro, compreso il carry
RRC	Rotazione a destra del registro, il bit meno significativo nel carry
RR	Rotazione a destra del registro, compreso il carry

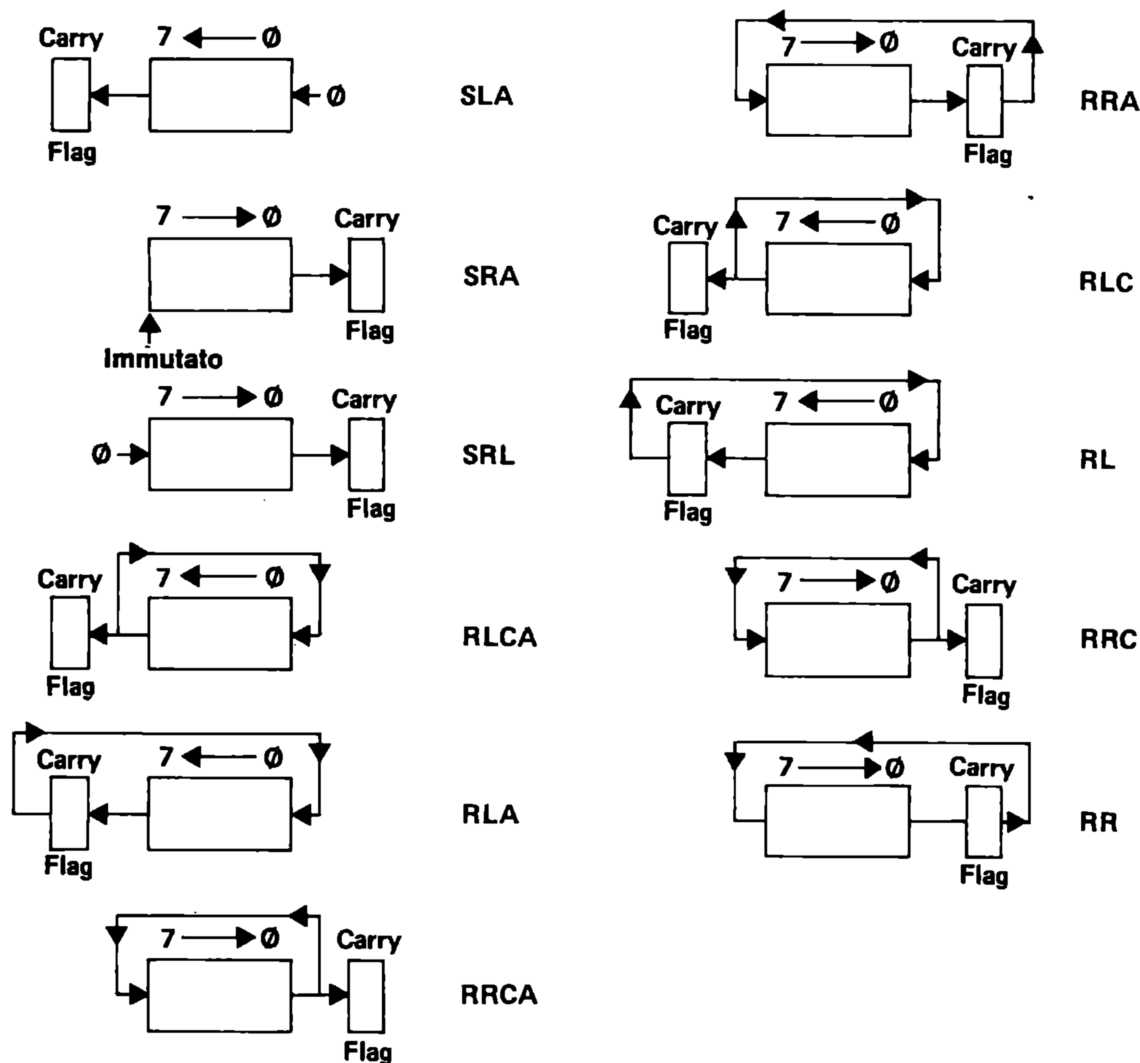


Fig. 5.2 I principali comandi di scorrimento e rotazione dello Z80, con i loro effetti e i codici mnemonici.

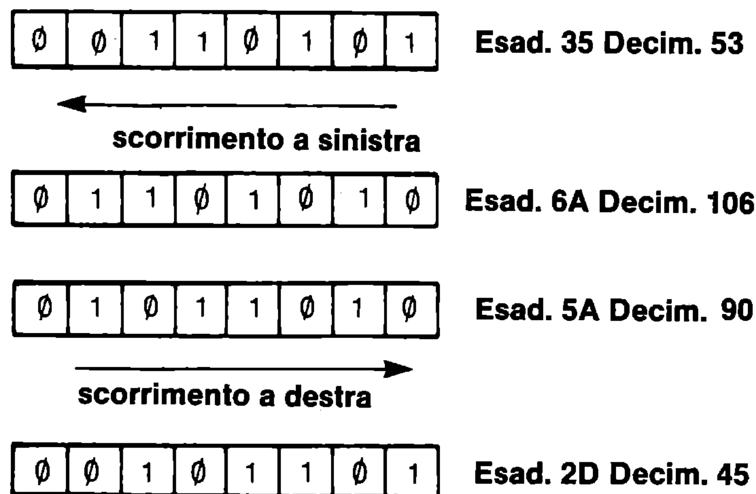


Fig. 5.3 L'effetto di uno scorrimento su un numero.

Una rotazione, al contrario, non modifica l'immagazzinamento dei bit di un registro, ma vengono cambiate le *posizioni* dei bit. Lo Z80 prevede due direzioni per la rotazione, sinistra e destra, e tre modi di usare il flag di carry, all'estremità meno significativa o a quella più significativa del byte ed anche come parte della rotazione o no. Ci sono otto istruzioni di rotazione, in tutto, ma non ne userete più di due nella maggior parte dei vostri programmi. Alcune delle rotazioni sono eseguite esclusivamente nell'accumulatore. Esse sono contraddistinte dall'uso di "A" nella mnemonica, quindi le rotazioni del tipo RLCA, RLA, RRCA, RRA sono disponibili solo per i bytes che si trovano nell'accumulatore. Le altre istruzioni di rotazione sono disponibili per una più ampia gamma di metodi d'indirizzamento.
L'istruzione CP (Compare=confronto) è un'istruzione particolarmente utile, che compare in quasi tutti i programmi. L'istruzione si applica *solo* ad un byte contenuto nell'accumulatore. Può usare qualsiasi dei metodi-standard d'indirizzamento di memoria, ed il suo effetto è di confrontare il byte copiato dalla memoria con il byte già contenuto nell'accumulatore. "Confrontare" significa sottrarre il byte copiato dalla memoria da quello contenuto nell'accumulatore. La differenza fra questa istruzione ed una vera sottrazione è che il risultato *non* viene memorizzato da nessuna parte! Il risultato della sottrazione viene utilizzato solamente per posizionare i flag, e il byte contenuto nell'accumulatore rimane immutato.Per esempio, supponete che l'accumulatore contenga il byte 4FH e di avere un byte della stessa grandezza memorizzato all'indirizzo 827FH, contenuto nella coppia di registri HL. Se usiamo l'istruzione:

```
CP (HL)
```

allora il flag di zero nel registro F sarà posto a 1, ma il byte dell'accumula-

tore sarà ancora 4FH, e anche il byte in memoria sarà ancora 4FH. Una sottrazione avrebbe lasciato il contenuto dell'accumulatore uguale a zero. Perché mai dovrebbe essere importante tutto questo? Bene, supponete di voler far eseguire qualcosa al programma se viene premuto il tasto "Y", e qualcosa di diverso se viene premuto il tasto "N". Se voi fate in modo che il programma in codice macchina immagazzini nell'accumulatore il codice ASCII per il tasto che viene premuto, potete effettuare il confronto. Confrontandolo con 4EH (il codice ASCII per "N"), possiamo sapere se è stato premuto il tasto "N". Se ciò è avvenuto, allora il flag di zero sarà posto a 1. In caso contrario, possiamo effettuare un'altra prova. Confrontando con 59H, possiamo sapere se è stato premuto il tasto "Y" — ancora una volta questo posizionerà a 1 il flag di zero. Se nessuno di questi due confronti ha posizionato a 1 il flag di zero, sappiamo che non è stato premuto né il tasto "Y" né quello "N", e possiamo tornare indietro e provare di nuovo. Se pensate che assomigli molto all'azione del ciclo di INKEY$ in BASIC, avete perfettamente ragione!
Infine, abbiamo il gruppo di azioni di salto. Queste permettono di "testare" i flag del registro F e, se un flag è stato posto a 1, faranno saltare il programma ad un nuovo indirizzo. Quale flag? Dipende dall'istruzione usata per il test del salto, perchè ce n'é una diversa per ogni flag e per ogni stato di un flag. Per esempio, considerate i due test, la cui mnemonica è JR Z,d e JR NZ,d. La prima, JR Z,d significa: "se il flag di zero è posizionato a 1, allora salta 'd' posti". Si avrà quindi un salto se il risultato di una sottrazione o di un confronto è zero. Il suo "opposto", JR NZ, d significa: " se il flag di zero *non* é posizionato a 1, allora salta". Ci sono, perciò, due istruzioni di "branch" (cioé, salto condizionato) che testano il flag di zero, ma in modi opposti. Lo stesso accade per quasi tutti gli altri flag. C'é anche un'istruzione di salto relativo, la cui mnemonica é JR, che non é seguita dal nome di nessun flag e pertanto non esegue nessun test, un po' come un'istruzione di GOTO in BASIC, senza nessun IF che la preceda. Oltre a queste istruzioni di salto JR, che usano l'indirizzamento relativo, ci sono delle istruzioni di salto JP, che usano altri metodi d'indirizzamento, come l'indirizzamento diretto, l'indirizzamento tramite HL, e l'indirizzamento indicizzato. Le istruzioni JP possono effettuare i test su un maggior numero di flag.
L'elenco completo di tutte le istruzioni di salto é riportato nella Fig. 5.4. Molte di queste sono istruzioni che probabilmente non userete mai, e quelle veramente importanti sono le istruzioni che usano i flag di zero, di segno e di carry. Per i programmi brevi, probabilmente userete solo le istruzioni JR, poiché esse vanno sempre bene per salti fino a 127 passi in avanti e 128 passi indietro. Le istruzioni JP sono usate nei programmi lunghi, o quando si deve effettuare un salto da un programma nella RAM ad una routine nella ROM *senza ritornare alla RAM.* Se invece volete effettuare un salto e ritornare, esiste un'istruzione CALL che cor-

risponde all'uso di GOSUB in BASIC. Ne parleremo più avanti, comunque.

JP ind	Salto all'indirizzo dato
JP c,ind	Salto all'indirizzamento dato se la condizione c è verificata.
JR d	Saltò relativo all'indirizzo del PC usando il parametro di spostamento d.
JR c,d	Salto relativo all'indirizzo del PC, usando il parametro di spostamento d se la condizione c è verificata.

CONDIZIONI PER JP

NZ - non zero
Z - zero
NC - nessun riporto
C - carry settato
PO - parità dispari
PE - parità pari
P - segno positivo
M - segno negativo

CONDIZIONI PER JR

NZ - non zero
Z - zero
C - riporto
NC - nessun riporto

Nota: C'é una versione di JP, JP(HL), che esegue un salto all'indirizzo contenuto nel registro HL.

Fig. 5.4. L'elenco completo delle istruzioni di salto dello Z80.

L'INTERAZIONE CON IL COMPUTER MSX

È finalmente arrivato il momento di iniziare una programmazione, anche se molto elementare, in codice macchina del vostro computer MSX. Non si tratta di digitare semplicemente delle righe di programma come se fossero righe di BASIC. A meno che non abbiate inserito il programma assemblatore ZEN, il computer MSX vi darà soltanto dei messaggi di "SYNTAX ERROR" quando cercherete di eseguire questi programmi. Dal momento che vogliamo cominciare su scala ridotta, ci dimenticheremo degli assemblatori, per ora, e assembleremo "manualmente". Questo

vuol dire che troveremo i bytes di codice macchina che corrispondono alle istruzioni in linguaggio assembly cercandoli in una tabella. Poi li immetteremo nella memoria del computer MSX con l'istruzione POKE, metteremo l'indirizzo del primo byte nel PC dello Z80, e staremo a vedere che cosa accade. Sembra semplice, ma c'é molto ragionamento da fare e un certo numero di precauzioni da prendere. Per cominciare, il computer MSX usa una buona parte della RAM per i suoi scopi, come abbiamo visto. Se ci limitiamo ad usare la funzione POKE per immettere in memoria un numero di bytes, senza fare attenzione a quale parte della memoria stiamo usando, vi è la possibilità o di sostituire i bytes che il computer MSX deve utilizzare, oppure i bytes del nostro programma saranno sostituiti dall'azione del computer MSX. Ci occorre, quindi una parte di memoria che sia sicuramente riservata al nostro personale uso.
Si può far questo usando il comando BASIC CLEAR. Quando utilizzate CLEAR nel modo consueto, tipo CLEAR 100, l'azione sarà di riservare 100 bytes agli indirizzi più alti della RAM per la memorizzazione di stringhe. CLEAR 100, per esempio, riserva spazio per 100 bytes di caratteri di stringa. Possiamo combinare questa istruzione con un altro tipo di operazione CLEAR. Se, ad esempio, digitate CLEAR 100, &HF000, in tal caso, oltre ad aver lasciato spazio per 100 bytes di caratteri di stringa, vi siete assicurati che il BASIC del computer MSX non possa usare nessun indirizzo maggiore di F000H (61440 decimale) per i suoi scopi. Il normale limite di memoria per questo uso è F380H (62336 decimale), quindi avrete gli indirizzi da 61440 a 62336 per i vostri programmi in codice macchina, o per memorizzare dei bytes senza rischiare di perderne il valore. Vi sono in totale 896 bytes, più che sufficienti per tutti i programmi che proveremo in questo libro. Notate che F380H è il più alto indirizzo della RAM che possa essere usato *in ogni caso* sul computer MSX. Se cercate di usare indirizzi sopra F380H, andrete probabilmente a toccare il sistema operativo e farete bloccare la macchina. L'istruzione CLEAR può usare sia numeri decimali che numeri esadecimali, ma dobbiamo specificare &H prima di ogni numero esadecimale.
Dopo aver protetto uno spazio in memoria in modo da immagazzinare i bytes di un programma, l'altro problema è di come mettere l'indirizzo di inizio del programma nel Contatore di Programma dello Z80. Fortunatamente, i progettisti dello Z80 sono stati gentili verso di voi. Ci sono due comandi BASIC, DEF USR (che può essere scritto anche DEFUSR) e USR, che faranno questo per voi. Il principio è che DEF USR deve essere seguito da un segno di uguaglianza, poi dal primo indirizzo del programma. Per esempio, se il vostro programma comincia all'indirizzo F001H, vi preparerete ad usarlo con il comando DEF USR=&HF001. Il comando DEF USR di solito usa numeri decimali, ecco perché bisogna usare &H. Però, potreste benissimo scrivere così questa istruzione: DEF USR=61441. Questo *prepara* l'immissione dell'indirizzo nel PC. Dico

"prepara", perché non esegue effettivamente l'azione. Il PC viene caricato e quindi il programma in codice macchina inizia quando usate, in BASIC, qualcosa del tipo A=USR(0) oppure PRINT USR(1). Il numero fra parentesi è essenziale per l'azione, quindi bisogna sempre immettere un numero anche se non viene usato. Questo è un esempio di numero "fittizio". La macchina può usare DEF USR e USR fino a dieci programmi in codice macchina, ponendo un numero di riferimento dopo DEF USR e USR, ma per ora lasciamolo da parte.
L'uso di DEF USR=&HF001, e poi di A=USR(0), farà sì che F001H venga usato come indirizzo del primo byte del vostro programma. Fra parentesi, io l'ho considerato come "byte di partenza". È però possibile scrivere programmi in cui i primi bytes siano dei dati, per cui il programma comincerà, diciamo, al decimo bytes. Questo non crea alcun problema, userete semplicemente l'indirizzo dell'effettivo byte di inizio come numero per DEF USR.
Infine, almeno per il momento, dovete assicurarvi che il vostro programma in codice macchina si fermi al punto giusto. Niente di ciò che abbiamo fatto fin qui indica allo Z80 del computer MSX dove termina il vostro programma. Quindi, lo Z80 continuerà a leggere bytes dopo la fine del vostro programma, finché incontrerà qualche byte che provoca un "fallimento". Per esempio, questo potrebbe essere un byte che genera un ciclo (loop) senza fine. Alcuni programmatori dubitano che ci siano bytes che non generano un ciclo senza fine in queste circostanze! Per ritornare correttamente al sistema operativo del computer MSX, dovrete terminare il vostro programma in codice macchina con un'istruzione di "ritorno dalla subroutine", la cui mnemonica è RET e il cui codice è C9H, 201 decimale. Se usate PEEK con le routine della ROM, vedrete questi bytes "201" sparsi dappertutto, per indicare la fine di ogni parte di programma.
C'é un altro rompicapo di cui non dobbiamo preoccuparci, per il momento. Quando eseguite un programma in codice macchina insieme ad un programma BASIC sul vostro computer MSX, usate lo stesso microprocessore Z80 per i due scopi. Se usate i registri dello Z80 nel vostro programma in codice macchina, come siete costretti a fare, allora dovete essere assolutamente certi di non distruggere informazioni che occorrono al programma BASIC. Per esempio, se nel momento in cui è partito il vostro programma in codice macchina i registri dello Z80 contenevano gli indirizzi di una parola riservata della ROM, allora ci sarà bisogno di questo indirizzo quando termina il programma in codice macchina. Quando il programma in codice macchina è richiamato dall'istruzione USR, questo avviene automaticamente. Il contenuto dei registri dello Z80 viene posto in una parte della memoria RAM chiamata "stack". Questa è quindi un'altra buona ragione per fare attenzione a dove immettere in memoria il vostro programma in codice macchina. Se cancellate lo stack, il computer MSX non sarà certamente contento! Quando viene trovata

l'istruzione RET, alla fine del programma in codice macchina, i bytes che erano stati immagazzinati nello "stack" sono di nuovo rimessi nei registri e così riprende la normale azione.
Se attivate un programma in codice macchina con qualsiasi altro metodo, senza usare USR, dovrete effettuare voi stessi questa operazione di salvataggio con un'istruzione inserita nel vostro programma in codice macchina, e questo implica l'uso delle istruzioni PUSH e POP — ma una cosa per volta.

FINALMENTE QUALCHE PROGRAMMA PRATICO

Terminati tutti questi preliminari, possiamo finalmente cominciare con qualche programma, molto semplice, ma utile per farvi prendere confidenza con il modo in cui i programmi vengono posti nella memoria del computer. Farete anche un po' d'esperienza nell'uso del linguaggio assembly e del codice macchina, e vedrete come può essere eseguito un programma in codice macchina.
Cominceremo con l'esempio più semplice possibile — un programma che immette solo un byte in memoria. In linguaggio assembly, avrà questa forma:

```
ORG 61441    ;inizia a porre qui i bytes
LD A,55H     ;metti 55H nell'accumulatore
LD(62000),A  ;memorizzali a 62000
RET          ;ritorna al BASIC
```

La prima riga contiene una mnemonica, ORG, che non avete visto prima. *Non* fa parte delle istruzioni dello Z80, ma è un'istruzione diretta all'assemblatore, che in questo caso siete voi! ORG è un'abbreviazione di origine, e vi ricorda che questo è il primo indirizzo che verrà usato per il vostro programma. È stato scelto un indirizzo che lascia spazio per programmi più lunghi di quelli che sono in questo libro, e si sarebbe potuto scegliere un numero più alto. Comunque, questo andrà benissimo, tanto più che lascia spazio anche per programmi più lunghi. Quando programmate con un assemblatore, scrivendo la prima riga, l'assemblatore immetterà automaticamente i bytes del programma in memoria a partire da questo indirizzo. Nel nostro caso, con l'assemblaggio "fatto a mano", ha solo la funzione di ricordarci quali indirizzi usare. Notate i commenti che seguono il punto e virgola. Il punto e virgola, in linguaggio assembly, ha la stessa funzione di REM in BASIC. Qualsiasi cosa segua il punto e virgola, è soltanto un commento che verrà ignorato dall'assemblatore, ma che il programmatore potrà trovar utile.
Ora dobbiamo vedere che cosa fa il programma. La prima vera istruzione è di caricare il numero 55H nell'accumulatore. Esso usa un indirizzamen-

to immediato, per cui il numero 55H dovrà essere scritto subito dopo la virgola. La riga seguente ordina che il byte contenuto nell'accumulatore (ora 55H) sia memorizzato all'indirizzo 62000. In esadecimale, corrisponde a F230H. È un indirizzo molto al di sopra di quelli che useremo per il programma. Ovviamente, non vogliamo usare un indirizzo che possa essere usato anche dai bytes del programma. Questa istruzione usa l'indirizzamento diretto: ci sono dei metodi più eleganti, ma non per i principianti! Infine, il programma termina con l'istruzione RET, essenziale per assicurarci che il computer MSX torni alla sua normale attività dopo che il nostro programma sarà finito.
Il passo successivo è di scrivere i codici. Bisogna cercare ogni codice, facendo attenzione a prendere il codice corretto per ogni metodo d'indirizzamento. Il codice per LD A con indirizzamento immediato è 3EH, 62 decimale, quindi questo è il primo byte del programma che verrà memorizzato all'indirizzo 61441 (decimale).
Possiamo cominciare una tabella di numeri d'indirizzo e di dati in questo modo:

```
61441    62
```

e poi andiamo avanti. Il byte che vogliamo caricare è 55H, 85 decimale, e questo deve essere immesso nel successivo numero d'indirizzo, perché questo è il modo in cui opera l'indirizzamento immediato. La tabella ora sarà:

```
61441    62
61442    85
```

Avremo ora bisogno del byte d'istruzione per LD in memoria, con l'indirizzamento esteso. Questo byte è 32H, 50 decimale, e deve essere seguito dai due bytes dell'indirizzo a cui vogliamo memorizzare i bytes. L'indirizzo 62000 equivale a F230H esadecimale, quindi possiamo usare i bytes 30 e F2 dopo l'istruzione LD. I loro corrispondenti valori decimali sono 48 e 242. L'ultimo codice deve essere il codice di RET, C9 esadecimale, 201 decimale; ora la tabella appare come quella della Fig. 5.5. Usa gli indirizzi da 61441 a 61446, sei bytes in tutto, e immetterà un byte all'indirizzo 62000, usando numeri decimali. Ora dobbiamo metterlo in memoria e farlo lavorare.

INDIRIZZO	BYTE
61441	62
61442	85
61443	50

61444	48
61445	242
61446	201

Fig. 5.5. Il programma codificato sotto forma di tabella, con indirizzi e bytes di dati espressi nella notazione decimale. Ogni byte d'istruzione è stato scelto dall'elenco dell'Appendice E.

Per far questo, occorre un programma BASIC che "ripulisca" la memoria, e vi immetta i bytes uno per uno. Il programma è mostrato nella Fig. 5.6

```
10 CLEAR 200,61440!
20 A!=61441
30 FOR N%=0 TO 5
40 READ D%:POKE A!+N%,D%
50 NEXT
60 DEF USR=61441!
70 A=USR(0)
80 DATA 62,85,50,48,242,201
```

Fig. 5.6. Il programma BASIC che immette i bytes in memoria.

Usando CLEAR 200,61440 ci assicuriamo che tutti gli indirizzi di memoria sopra 61440, cioè da 61441 a 62336 decimale, non siano utilizzati dal computer MSX. Alla variabile A! viene assegnato il valore 61441, in modo da poterlo usare nei comandi POKE. Le righe 30 e 40 immettono i numeri dei dati agli indirizzi che cominciano da 61441. Abbiamo usato POKE A!+N% e dal momento che A!=61441 e N%=0, il primo indirizzo deve essere per forza 61441. Abbiamo scritto tutto il programma usando numeri decimali, e questo ha causato delle complicazioni quando abbiamo dovuto scrivere dei numeri d'indirizzo di due bytes. Poiché dovrete usare la notazione esadecimale con l'assemblatore ZEN, tanto vale cominciare, prima possibile, a prendere dimestichezza con questi principi. Nei programmi successivi, perciò, sia gli indirizzi che i dati saranno espressi in valori esadecimali.Dovremo quindi usare VAL("&H"+D$) per trovare il valore numerico decimale dei codici esadecimali per poter usare la funzione POKE, che non accetta numeri esadecimali. Ad esempio, dovremo aggiungere &H al codice "3E" e poi VAL troverà il valore *decimale* per l'istruzione POKE.
La riga 60 contiene DEF USR=61441, che prepara il caricamento del PC dello Z80. L'ultima riga del programma, la riga 70, contiene A=USR(0). Sia la "A" che lo "0" sono valori fittizi, senza alcun significato nel programma, sebbene debbano essere presenti perché l'istruzione sia valida. Questa è l'istruzione BASIC che farà sì che il vostro programma

venga effettivamente eseguito, con l'indirizzo di inizio specifico nella parte DEF USR. La riga 80 contiene infine i sei bytes di dati che abbiamo prima individuato. Quando avete immesso il programma, battete ?PEEK(62000) per vedere cosa c'é all'indirizzo 62000: dovrebbe essere zero. Quando date il comando RUN, non vedrete nessun effetto, perché non potete vedere che cosa c'é all'indirizzo 62000. Se poi usate:

?PEEK(62000)

dovreste trovare il valore 85, la versione decimale di 55H, il numero immesso dal programma. Ora provate questo: digitate POKE 62000,255, e poi cancellate la riga 70 del vostro programma, cioé la riga USR. Date ancora il comando RUN e usate ?PEEK(62000) per vedere che cosa c'é a questo indirizzo: dovrebbe essere 255.
Ora digitate ?USR(0) e premete RETURN. Vedrete uno zero sullo schermo: ignoratelo. Se usate di nuovo ?PEEK(62000), dovrebbe darvi ancora il valore 85. Questo perché l'immissione in memoria dei bytes del programma non provoca l'esecuzione del programma, soltanto USR potrà farlo. Potete perciò immettere dei valori con l'istruzione POKE nella prima parte di un programma BASIC e poi usarli più avanti, ponendo dove volete l'istruzione EXEC. Anche se cancellate tutto il resto del programma BASIC, il codice macchina rimarrà nella memoria finché non usate POKE per immettere qualcos'altro a quegli indirizzi, o finché non spegnete il computer. Usando ?USR(0) oppure A=USR(0), questa parte di programma in codice macchina verrà sempre eseguita finché non userete DEF USR per assegnare un nuovo indirizzo di partenza. Possiamo utilizzare con profitto questo sistema per immettere in memoria una parte in codice macchina, cancellando poi il programma BASIC che ve lo ha immesso (basta mettere NEW invece di END alla fine del programma BASIC), per usare solamente il codice macchina. Potete registrare su nastro il programma in codice macchina, se volete, ma questa è una tecnica che vedremo più avanti. Un passo alla volta, per favore!
Il primo esempio di codice macchina non ha certo delle ambizioni: fa esattamente quanto farebbe POKE 62000, 85 in BASIC, ma è pur sempre un inizio. L'importante, a questo punto, é abituarsi al metodo per scrivere il codice macchina, immetterlo in memoria e farlo eseguire.
Ora proviamo qualcosa di molto più ambizioso rispetto al nostro uso del codice macchina — benché l'esempio sia assai semplice. La Fig. 5.7 mostra la versione in linguaggio assembly del programma.

```
1 ORG 0F000H
2 LD IX,0F020H
3 LD A,(IX+0)
4 SLA A
```

```
5 LD (IX+1),A
6 RET
7 END
```

Fig. 5.7. Il programma in linguaggio assembly per "moltiplica per due". Il listato mostra il linguaggio assembly come viene usato per l'assemblatore ZEN.

Ciò che faremo, sarà di caricare un byte nell'accumulatore, spostarlo di un posto a sinistra, e poi immetterlo in memoria ad un passo più alto dell'indirizzo da cui lo avevamo prelevato. Questa sembra anche una buona occasione per mostrare un esempio d'indirizzamento indicizzato, quindi cominceremo con l'immettere un indirizzo nel registro IX. Questo è il passo LD IX, F020H, ed il suo effetto sarà di porre l'indirizzo F020H nel registro IX. La riga successiva, LD A,(IX+0), significa che l'accumulatore deve essere caricato dall'indirizzo contenuto nel registro IX, aggiungendo 0 a questo numero, ovvero il caricamento avverrà da F020H, lo stesso indirizzo contenuto nel registro IX. Il terzo è SLA A, scorrimento aritmetico a sinistra, per cui i bit del byte vengono spostati a sinistra. Con il quarto passo, il byte viene immagazzinato all'indirizzo F021H usando LD (IX+1),A. Questa volta, viene aggiunto 1 al numero contenuto nel registro IX in modo che il byte sia memorizzato all'indirizzo F021H. Terminiamo, come sempre, con l'istruzione RET.
Ora possiamo tradurre questo programma nella forma codificata. L'istruzione LD IX,F020H è codificata come DD21, seguito dai due bytes dell'indirizzo, 20 e F0 *in quest'ordine*. L'istruzione LD A con indirizzamento indicizzato è codificata come DD 7E, ma non termina qui. Quando usate un'istruzione indicizzata di questo tipo, il byte d'istruzione deve essere seguito da un altro byte. Questo, chiamato lo spostamento, occorre per specificare il numero che deve essere aggiunto all'indirizzo contenuto nel registro indice. In questo esempio, usiamo numeri come 0 e 1, ma non possiamo omettere lo 0. Questo, perciò, segue il codice DD 7E per il caricamento indicizzato. I bytes di SLA A sono CB 27 (abbiamo proprio preso tutte le istruzioni di più bytes, questa volta!), e poi arriviamo a LD(IX+1). La parte dell'istruzione è DD 77 e deve essere seguita dal byte di spostamento 1. Ciò farà in modo che il caricamento venga effettuato ad un indirizzo che è l'indirizzo contenuto nel registro indice *più uno*. Infine, C9 è il comando RET.
Ora dobbiamo codificarlo in BASIC. Se scegliamo un numero piccolo da immettere in F020H, l'effetto dello scorrimento a sinistra sarà di raddoppiare il numero, perciò possiamo usarlo per fare un po' di magia aritmetica! Il programma BASIC è mostrato nella Fig. 5.8. Come al solito, cominciamo col ripulire la memoria. Non dovete preoccuparvi se c'era un altro programma in questa parte della memoria. Il nuovo programma lo sostituirà completamente e, se il vostro programma termina correttamen-

te con il byte d'istruzione RET, i bytes del vecchio programma non possono interferire con quelli del nuovo. I valori sono immessi in memoria con la funzione POKE, nel modo consueto, alle righe da 30 a 50. Alla riga 60, comunque, immettiamo un numero, 85 decimale, all'indirizzo F020H. Questo è l'indirizzo che sarà usato dal programma e il byte 55H, 01010101 nella forma binaria, sarà posto a questo indirizzo. Alla riga 70, A=USR(0) permette l'esecuzione del programma in codice macchina, che dovrebbe far scorrere a sinistra questo byte, facendolo diventare 10101010, il cui valore decimale è 170, il doppio di 85. La riga 80 contiene i bytes dei dati. Quando il programma sarà eseguito, potrete vederne il risultato con ?PEEK(&HF201). Potremmo abbellire il programma aggiungendo la riga 75:

```
75 ?"Due volte";PEEK(&HF020);"è uguale a";PEEK(&HF021)
```

Apparirà così il messaggio: Due volte 85 è uguale a 170
quando viene eseguito il programma.

```
10 CLEAR 200,&HF000
20 A!=&HF001
30 FOR N%=0 TO 12:READ D$
40 POKE A!+N%,VAL("&H"+D$)
50 NEXT:DEF USR=&HF001
60 POKE &HF020,&H55
70 A=USR(0)
80 DATA DD,21,20,F0,DD,7E,00,CB,27,DD,77,01,C9
```

Fig. 5.8. Il programma BASIC che immette i bytes in memoria e poi utilizza il programma in codice macchina.

È assai semplice, ma se non sapevate niente di codice macchina, vi starete chiedendo come diavolo ha fatto il numero ad essere moltiplicato per due. Ancora una volta, il programma non fa niente che non possa essere eseguito più facilmente, ed anche più rapidamente, in BASIC. La cosa importante, dal nostro punto di vista, è che ora avete usato un indirizzamento indicizzato ed un'istruzione di scorrimento, oltre ad aver acquisito più esperienza per immettere un programma in codice macchina nel vostro computer MSX con il metodo più difficile. Se, fra parentesi, avete fatto degli sbagli, soprattutto con i DATA, è probabile che il computer MSX cada in "trance" e rifiuti di fare qualsiasi cosa. Quando avete immesso un programma BASIC come questo, che pone dei bytes nella memoria, registrate *sempre* il programma BASIC prima di dare il comando RUN. In questo modo, se l'effetto di un byte non corretto è di

cancellarvi metà della RAM, potete spegnere, riaccendere e ricaricare il vostro programma. Se non l'avete registrato, dovrete ribatterlo nuovamente. Ed è dura!

ALTRI ESEMPI

A questo punto, è importante provare molti semplici programmi perché siate più sicuri nell'uso di questi metodi. Quando avrete finito il capitolo, avrete le idee molto più chiare sul modo di approcciare il codice macchina da un punto di vista pratico, e potrete fare degli altri esercizi per conto vostro. Il prossimo capitolo vi aiuterà nella progettazione dei programmi in codice macchina, che è la parte più difficile. Dopo questo, è tutta pianura!
Ritornando agli esempi, è il momento di fare un po' di aritmetica con il caricamento indiretto tramite registro. La Fig. 5.9 mostra la versione in linguaggio assembly, che inizia caricando un indirizzo, F030H, nella coppia di registri HL.

```
 1  ORG 0F000H
 2  LD HL,0F030H
 3  LD A,(HL)
 4  INC HL
 5  ADD A, (HL)
 6  INC HL
 7  ADD A,(HL)
 8  INC HL
 9  LD (HL),A
10  RET
11  END
```

Fig. 5.9. Un programma in linguaggio assembly che usa il caricamento indiretto tramite registro per eseguire delle addizioni.

Il passo successivo è di caricare l'accumulatore da questo indirizzo usando LD A,(HL). Ricordate che le parentesi significano "contenuto di". Ciò che vogliamo fare è caricare il byte memorizzato all'indirizzo F030H nell'accumulatore. Dopo aver fatto questo, il prossimo passo è INC HL. Questa istruzione incrementerà l'indirizzo a F031H. Notate che *non* usiamo INC (HL), perché questo incrementerebbe il byte posto all'indirizzo F030H, non l'indirizzo. Dopo aver incrementato l'indirizzo in HL, il byte che si trova al nuovo indirizzo F031H viene addizionato al byte contenuto nell'accumulatore. L'indirizzo viene ancora incrementato, ed un altro byte è addizionato all'accumulatore. L'accumulatore, come ricordate, contiene sempre il risultato dell'addizione, quindi sta accumu-

lando il numero per noi. Infine, l'indirizzo viene incrementato ancora una volta e l'accumulatore è caricato in memoria a F033H, dopo di che vi è l'istruzione RET.
La Fig. 5.10 mostra questo programma nella forma BASIC con l'uso dell'istruzione POKE. Le righe dalla 10 alla 50 preparano lo spazio, immettono i bytes del programma in memoria e pongono l'indirizzo di inizio nell'istruzione DEF USR. La riga 60 immette, con la funzione POKE, tre valori in posizioni consecutive della memoria in modo che il programma possa usarli. La riga 70 fa eseguire la parte del programma in codice macchina, e la riga 80 dimostra che tutto ha funzionato stampando la somma dei tre numeri. Potete provare a cambiare i numeri, ma ricordate che se la somma supera il valore decimale 255, verrà stampato solo il resto, gli otto bit più bassi che rimangono nell'accumulatore. Se la somma è 300, per esempio, quello che rimane sarà 300—256=44. Se la somma è 700, ciò che rimane sarà 700—(2*256)=188.

```
10  CLEAR 200,&HF000
20  A!=&HF001
30  FOR N%=0 TO 10
40  READ D$:POKE A!+N%,VAL("&H"+D$):NEXT
50  DEF USR=&HF001
60  POKE&HF030,18:POKE&HF031,12:POKE&HF032,27
70  A=USR(0)
80  PRINT"La somma è";PEEK(&HF033)
90  DATA 21,30,F0,7E,23,86,23,86,23,77,C9
```

Fig. 5.10. Il programma BASIC che immette in memoria il codice macchina.

Ora per un ultimo esempio di questo capitolo, consideriamo qualcosa di meno numerico. Non potrete mai allontanarvi dai numeri quando avete a che fare con il codice macchina, ma almeno questa volta non facciamo dell'aritmetica e inoltre useremo un ciclo (loop) in BASIC per richiamare ventisei volte il programma in codice macchina. L'ideale sarebbe stato di inserire il ciclo nel codice macchina, ma non siamo ancora del tutto pronti per questo. La Fig. 5.11 mostra la versione in linguaggio assembly di quello che viene fatto nella parte del programma in codice macchina.

```
1  ORG 0F000H
2  LD DE,0F030H
3  LD C,20H
4  LD A,(DE)
5  OR C
```

```
6 LD (DE),A
7 RET
8 END
```

Fig. 5.11 Un programma in linguaggio assembly per trasformare le lettere maiuscole in minuscole.

La coppia di registri DE (finalmente un cambiamento!) viene caricata con un indirizzo, F030H. Questo indirizzo sarà usato per memorizzare temporaneamente un codice ASCII per una lettera. Il registro C viene poi caricato con il numero 20H, usando il caricamento immediato. Il principio qui seguito è che, se viene applicata la funzione OR fra il codice ASCII per una lettera maiuscola e il numero 20H, il risultato sarà il codice ASCII per la stessa lettera minuscola. Lo stesso risultato può essere ottenuto aggiungendo 20H, ma questa sembra una buona occasione per vedere come opera l'azione di OR. La terza riga del programma in linguaggio assembly carica l'accumulatore dall'indirizzo contenuto in DE, F030H, per cui un codice ASCII sarà immesso nell'accumulatore. L'azione OR C viene quindi eseguita fra il byte contenuto nell'accumulatore ed il byte (20H) contenuto nel registro C e immette il risultato di nuovo nell'accumulatore. Quindi ricarichiamo questo byte all'indirizzo F030H di DE e ritorniamo al BASIC. Ora abbiamo bisogno solo di un programma BASIC per immettere in memoria i bytes del codice macchina, e di porre i bytes a F030H perché il programma in codice macchina possa utilizzarli.
La Fig. 5.12 mostra il programma BASIC. Le righe dalla 10 alla 50 seguono procedimenti che ormai dovrebbero esservi familiari, per cui non occorre nessun altro commento. La riga 60 ripulisce lo schermo e inizia un ciclo, usando J%=65 TO 90 per coprire i codici ASCII da A a Z, lettere maiuscole. La riga 70 stampa ogni lettera e la riga 80 immette, con l'istruzione POKE, il codice ASCII all'indirizzo F030H. Questo è l'indirizzo usato dal programma in codice macchina, e la riga 90 fa in modo che venga eseguito. Questo convertirà il numero di codice nel codice ASCII per le lettere minuscole, per cui A diventerà a, B diventerà b, e così via. La riga 100 stampa il risultato, trovando prima il numero immagazzinato a F030H e stampando poi CHR$ di questo numero. E' un programma semplice, ma di effetto — guardatelo in azione!

```
10 CLEAR 200,&HF000
20 A!=&HF001
30 FOR N%=0 TO 8
40 READ D$:POKE A!+N%,VAL("&H"+D$):NEXT
50 DEF USR=&HF001
60 CLS:FOR J%=65 TO 90
```

```
 70  PRINTCHR$(J%);
 80  POKE&HF030,J%
 90  A=USR(0)
100  PRINTCHR$(PEEK(&HF030));
110  NEXT
120  DATA 11,30,F0,0E,20,1A,B1,12,C9
```

Fig. 5.12. Il programma BASIC che esegue un ciclo per stampare le lettere maiuscole e richiama il codice macchina per trasformarle in lettere minuscole.

CAPITOLO 6

Allarghiamo la visuale

I semplici programmi che abbiamo visto nel Capitolo 5 non fanno molto, sebbene siano un'utile esercitazione per scrivere programmi in linguaggio macchina. È essenziale, a questo livello, esercitarsi a scrivere il linguaggio assembly e a convertirlo in codice macchina, perché potete scoprire più facilmente se avete fatto un errore quando i programmi sono così semplici. Non è così facile trovare un errore in un programma lungo in codice macchina, soprattutto quando state ancora lottando per imparare il linguaggio!

La maggior parte delle difficoltà per i principianti sorgono, anche se sembra assai strano, non perché il codice macchina sia difficile, ma perché è molto semplice. Infatti, proprio perché è così semplice, occorre un gran numero di passi di istruzione per raggiungere un minimo risultato, e quando un programma contiene molti passi di istruzione, è più difficile da stendere. La parte più difficile di questa stesura è il metter giù ciò che volete fare in un insieme di passi che possono essere tradotti in linguaggio assembly, e i diagrammi di flusso sono il metodo tradizionale per poterci orientare in modo corretto. Non ritengo affatto che i diagrammi di flusso siano adatti alla stesura di programmi BASIC, ma essi sono realmente utili per il codice macchina.

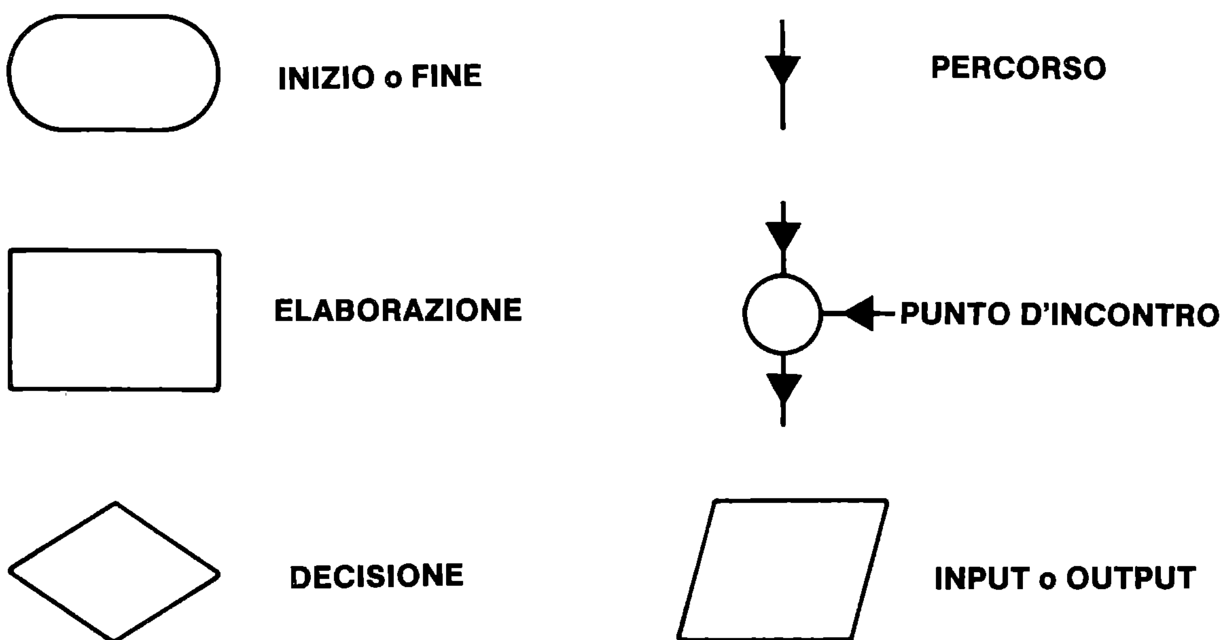


Fig. 6.1. Le principali figure-simbolo del diagramma di flusso.

DIAGRAMMI DI FLUSSO

I diagrammi di flusso stanno ai programmi come i diagrammi a blocchi stanno all'hardware, mostrano ciò che deve essere fatto (o che si cerca di fare) senza scendere in troppi dettagli, oltre il minimo essenziale. Un diagramma di flusso è formato da una serie di figure, e ognuna di esse è il simbolo di una determinata azione. La Fig. 6.1 mostra alcune delle figure più importanti per i diagrammi di flusso, almeno per i nostri scopi (riprese dal British Standard delle figure dei diagrammi di flusso). Queste sono il punto di inizio o di termine, i passi di entrata/uscita, di elaborazione (o azione) e di decisione. Dentro queste figure, possiamo scrivere delle brevi annotazioni dell'azione che desideriamo, ma anche qui senza dettagli.
Un semplice esempio è sempre il modo migliore di illustrarvi come viene usato un diagramma di flusso. Supponete di voler stendere un programma in codice macchina che prenda il codice ASCII per una lettera da un indirizzo in memoria, posizioni a 1 il bit 7, e poi reimmetta in memoria il numero. Posizionare a 1 il bit 7 (il bit più significativo) equivale ad aggiungere 128 al codice ASCII, e convertirà il codice della lettera nel codice di un "carattere speciale" o di un carattere grafico. Un diagramma di flusso per questa azione è mostrato nella Fig. 6.2.

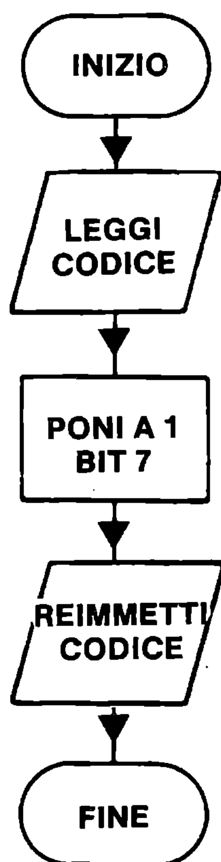


Fig. 6.2. Un diagramma di flusso per un programma che legge un byte, posiziona a 1 il bit 7, e reimmette in memoria il byte.

Il primo simbolo è quello del punto di INIZIO, perché ogni programma deve cominciare da un certo punto. La freccia mostra che si va poi al primo blocco di "input/output", entro cui è scritto "LEGGI CODICE". Questo descrive quello che vogliamo fare — prendere il numero di codice per un carattere immagazzinato in un indirizzo di memoria. Non sappiamo quale sia l'indirizzo, e non importa, per ora. Dopo aver prelevato il carattere, la freccia indica l'azione successiva, il posizionamento a 1 del bit 7 del numero di codice ASCII. Questo è rappresentato nel diagramma di flusso dal rettangolo di "azione", con l'azione effettiva scritta entro di esso. Poi, come mostra la freccia, il numero di codice modificato viene di nuovo posto in memoria allo stesso indirizzo.
Il simbolo di FINE ci ricorda poi che questa è la fine del programma.
Questo è un diagramma di flusso molto semplice, ma è sufficiente per illustrare il concetto. Le frecce sono *molto* importanti, perché mostrano la direzione del "flusso" (da cui il nome diagramma di flusso) del programma. Non è certo necessario che le frecce vi ricordino l'ordine delle azioni in un esempio così semplice come questo, ma man mano che i vostri programmi si faranno più audaci, queste frecce diventeranno più importanti. Notate, fra l'altro, che le descrizioni sono molto generali — non vengono mai inserite istruzioni in linguaggio assembly nelle figure del diagramma di flusso. Un diagramma di flusso dovrebbe essere scritto in modo da essere chiaro per chiunque lo legga, non dovrebbe mai esserci qualcosa comprensibile solo al programmatore e che potrebbe invece confondere le idee a qualsiasi altra persona. Un buon diagramma di flusso, infatti, dovrebbe poter essere usato da ogni programmatore per scrivere in ogni tipo di codice macchina — o in ogni altro "linguaggio", tipo BASIC, PASCAL, FORTH e così via. Ma, ahimé, molti diagrammi di flusso vengono fatti dopo che il programma è stato scritto (di solito attraverso molti errori e molti tentativi) nella sempre vana speranza di rendere più chiara l'azione, ma certamente voi non farete così, non è vero?
Una volta che avete il diagramma di flusso, potrete controllare che faccia ciò che volete, esaminandolo molto attentamente. In un esempio così semplice come il nostro, non c'è molto da fare, perché l'unica cosa che deve essere controllata è che l'ordine delle azioni sia corretto. Infatti, se scrivete esattamente il vostro diagramma di flusso, questo sarà praticamente tutto quello che avrete da fare! Questo perché scriverete un programma tracciando prima un diagramma di flusso delle azioni principali, e ci sono sorprendentemente poche azioni principali in un programma in codice macchina. La maggior parte dei programmi, infatti, hanno solo tre simboli nel primo diagramma di flusso: un input, un'elaborazione ed un output. Una volta fatto questo schema, tracciate dei diagrammi di flusso distinti per ognuna delle diverse parti. Ogni simbolo del diagramma potrebbe richiedere un altro diagramma di flusso e così via, finché non avrete un insieme di passi che possono essere tradotti facilmente in codice

macchina. Non dovrebbe essere una sorpresa per voi, se avete scritto dei programmi in BASIC, perché se leggete i testi adatti (il mio, spero) sul BASIC MSX, saprete già come stendere un programma suddividendolo in parti sempre più piccole. La differenza principale per i programmi in codice macchina è che queste parti sono molto più piccole!

I DIAGRAMMI DI WARNIER-ORR

Molti programmatori non si trovano bene con i diagrammi di flusso, e si sono rivolti ad un diverso modo di effettuare la stesura di un programma. Il nome, Warnier-Orr, deriva dall'inventore Warnier e da Orr, che ha contribuito a farlo conoscere. Uno dei vantaggi di questo metodo è che non dovete attenervi troppo strettamente ad un metodo prestabilito, a meno che non scriviate per una rivista professionale. Un diagramma di Warnier-Orr, nonostante il suo nome, si basa su delle parole per descrivere lo schema del programma, piuttosto che su dei simboli. Il suo grande vantaggio rispetto ai diagrammi di flusso è che si adatta meglio all'idea di progettazione "top-down" (dall'alto in basso) di un programma, stabilendo prima lo schema generale, e introducendo poi i vari particolari.
La Fig. 6.3 mostra un esempio molto semplice di un diagramma di Warnier-Orr relativo allo stesso problema della Fig. 6.2.
Questa volta, le parti principali dell'elaborazione vengono scritte sulla parte sinistra. Delle parentesi graffe vengono usate per mostrare come ogni azione sia suddivisa in altre, o per dare altre informazioni. Vorrei sottolineare che questo è un *adattamento* del metodo di Warnier-Orr;

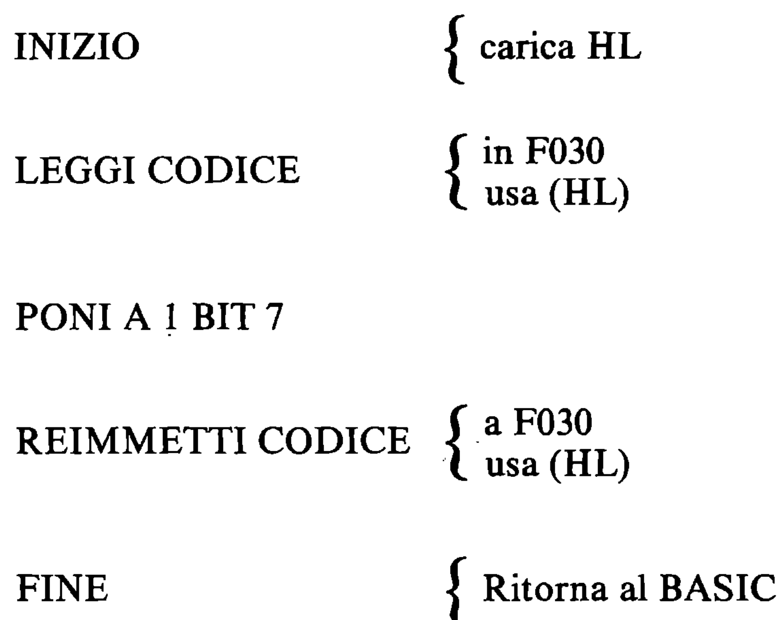


Fig. 6.3. Una versione semplificata di un diagramma di Warnier-Orr per il programma che posiziona a 1 il bit 7.

professori e altri puristi avrebbero da ridire. Tuttavia, va benissimo per me, e penso che possa andar bene anche per voi. In un progetto di programma semplice come questo, non c'è bisogno di cercare molti altri dettagli, ma, se ci occorresse, potremmo usare altri insiemi di parentesi graffe per ulteriori dettagli. In questo modo, il diagramma cresce dalla parte sinistra del foglio verso quella destra, con tutti i più sottili dettagli sulla destra. I grandi vantaggi di questo metodo sono di poter vedere sia lo schema generale che i dettagli su un foglio di carta. D'ora in poi, userò questo metodo di progettazione per illustrare i programmi di questo libro. Se vi trovate meglio con il diagramma di flusso, basta che tracciate le figure-simbolo accanto alle annotazioni.
Dopo aver mostrato in due modi diversi lo schema di questo programma, sarà meglio vedere come portarlo a termine. Se ci atteniamo allo schema delineato, arriveremo alla Fig. 6.4. L'indirizzo F030H è immesso in HL, poi l'accumulatore viene caricato da questo indirizzo, usando LD A,(HL). Il posizionamento del bit 7 viene fatto con il comando SET 7,A, e il byte viene poi di nuovo immesso in memoria con l'istruzione LD(HL),A.

```
1 ORG 0F000H
2 LD HL,0F030H
3 LD A, (HL)
4 SET 7,A
5 LD (HL),A
6 RET
7 END
```

Fig. 6.4. Implementazione dello schema in linguaggio assembly.

Questo programma è quanto l'esperienza fin qui acquisita ci permette di fare, ma lo Z80 prevede anche un metodo più semplice. Infatti, l'istruzione SET permette l'indirizzamento indiretto tramite registro. Al posto delle righe dalla 3 alla 5 comprese, possiamo sostituire l'unica istruzione SET 7,(HL). Non ci sarà più bisogno quindi, del caricamento dell'accumulatore e dall'accumulatore stesso. Poiché il motto di tutti i programmatori in codice macchina dovrebbe essere "semplificate e snellite i programmi", questa seconda versione sarà quella che useremo.
La Fig. 6.5 mostra un programma BASIC che usa quest'azione. Le righe dalla 10 alla 50 seguono la solita procedura, e l'azione comincia alla riga 60. Lo schermo viene ripulito, e vi si chiede di immettere il vostro nome. Alla riga 70, un ciclo INKEY$ aspetta che venga premuto un tasto. La riga 80 viene usata per individuare il tasto RETURN, che interrompe l'azione, eseguita nelle righe da 90 a 120. Il tasto che è premuto darà K$, e verrà trovato il codice ASCII per questo carattere. Il codice viene poi

immesso in memoria all'indirizzo F030H perché il codice macchina possa utilizzarlo. La riga 100 fa eseguire il codice macchina, che posiziona a 1 il bit 7 del codice ASCII e lo lascia all'indirizzo F030H. Quando il codice macchina ritorna al BASIC, sarà eseguita la riga 110, che stampa il carattere il cui codice è memorizzato ad F030H. Il punto e virgola fa in modo che la stampa continui sulla stessa linea e la riga 120 fa sì che l'azione venga ripetuta finché non viene premuto il tasto RETURN. La riga 130 poi vi dice che quello che vedete sullo schermo è il vostro codice. Certo, non assomiglierà al vostro nome! La cosa utile di tutto questo è che, anche per chi legge il listato, non sarà proprio ovvio quello che viene fatto dal programma.

```
 10 CLEAR 200, &HF000
 20 A!=&HF001
 30 FOR N%=0 TO 5
 40 READ D$:POKE A!+N%, VAL ("&H"+D$) :NEXT
 50 DEF USR=&HF001
 60 CLS:PRINT"Digitate il vostro nome, poi premete il ":PRINT
    "tasto RETURN."
 70 K$=INKEY$:IF K$=" " THEN 70
 80 IF K$=CHR$(13) THEN 130
 90 J%=ASC(K$):POKE&HF030,J%
100 A=USR(0)
110 PRINTCHR$(PEEK(&HF030));
120 GOTO 70
130 PRINT:PRINT "Ecco il vostro nome in codice!"
200 DATA 21, 30, F0, CB, FE, C9
```

Fig. 6.5. Il programma BASIC che immette in memoria i bytes del codice macchina ed attiva il programma.

PROVIAMO AD ESEGUIRE UN CICLO

Gli esempi che abbiamo visto finora riguardano programmi lineari, senza cicli (loop). Però, è molto raro far uso solo di programmi lineari in codice macchina: i cicli sono molto più comuni.
Un'azione ciclica in BASIC può essere molto lenta e potrete realmente apprezzare la velocità del codice macchina proprio in programmi che prevedono questi cicli. Se avete fatto qualcosa in più della più elementare programmazione BASIC, saprete già che cosa implica un ciclo. Un ciclo esiste quando una parte di programma verrà ripetuta continuamente, finché un certo test non ha esito positivo. In BASIC, un ciclo può essere originato da una riga come la seguente:

```
200 IF A=0 THEN GOTO 100
```

Essa contiene un test (è A=0?), e se il test ha esito positivo (si, A=0), allora il programma ritorna alla riga 100 e ripete tutti i passi da lì fino ad arrivare di nuovo alla riga 200. Questa specie di ciclo in BASIC assomiglia molto al modo in cui viene creato un ciclo in codice macchina. Invece di usare numeri di riga, però, usiamo numeri d'indirizzo. Invece di effettuare il test su una variabile "A", lo effettueremo sul contenuto di un registro che, nella maggior parte dei casi, sarà il registro A.
Cominciamo nel modo corretto con la stesura di un diagramma. La Fig. 6.6 mostra come potrebbe essere.

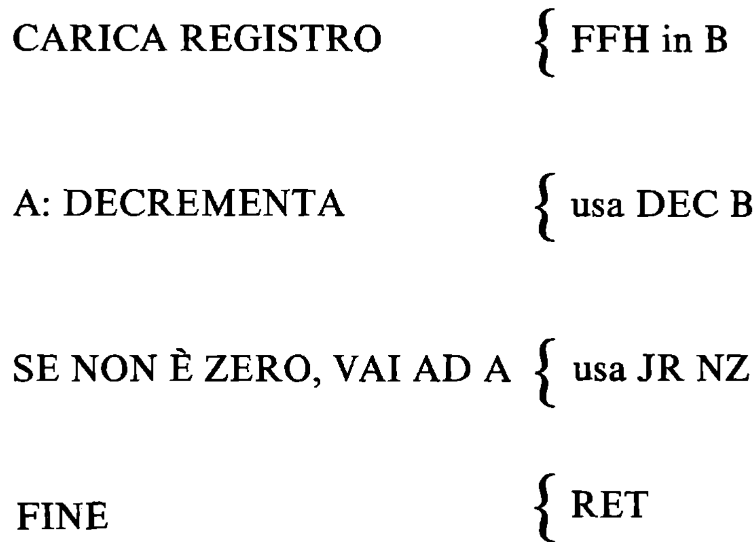


Fig. 6.6. Schema per un ciclo di conteggio di un solo byte.

Il primo passo è il caricamento di un registro. Il passo successivo è di decrementare il numero contenuto nel registro. Il terzo passo è di controllare quanto rimane. Se non è zero, dobbiamo ritornare alla fase di decremento. Io ho usato una lettera, A, come "label", per indicare dove il ciclo ritorna. Questa non è una notazione standard del diagramma di Warnier-Orr, ma è utile perché indica dove comincia il ciclo, una cosa che manca all'uso del GOTO in BASIC. Dopo aver messo giù questo semplice schema, useremo le parentesi graffe per specificarlo meglio. Alla fase di caricamento, metteremo il byte FFH nel registro B. Questo è il valore più grande che può assumere un byte singolo. Il passo di decremento sarà effettuato dall'istruzione DEC B, e il controllo userà JR NZ, e ritornerà al punto A, il passo di decremento.
L'azione consisterà nel caricamento immediato del registro B con il byte FFH, e poi il byte contenuto nel registro B viene controllato per vedere se è zero. Se lo è, ritorniamo al BASIC. Se non lo è (il che significa che il

conto alla rovescia non è completo), allora decrementiamo e proviamo ancora. Ora dobbiamo metter questo sotto forma di linguaggio assembly, e dovranno essere inseriti dei termini nuovi per voi. Per quanto ne sappiamo fin qui, potremmo scrivere questo programma in linguaggio assembly come segue:

```
ORG 0F00H
2 LD B,0FFH
3 LOOP: DEC B
4 JR NZ,LOOP
5 RET
6 END
```

e questo eseguirebbe l'azione desiderata. Lo Z80, però, è stato progettato in maniera da consentirvi modi più facili di programmare cicli di conteggio. Uno di questi modi è l'istruzione DJNZ. Le lettere DJNZ si riferiscono ad un'istruzione che automaticamente decrementa il registro B e salta ad un nuovo indirizzo se il byte contenuto nel registro B non ha raggiunto il valore di zero. Il nome DJNZ, infatti, sta per "Decrement and Jump if Not Zero" (decrementa e salta se non zero). Questa istruzione si applica *soltanto* al registro B.
La Fig. 6.7 mostra un programma in linguaggio assembly che usa questo utile passo DJNZ.

```
1 ORG 0F000H
2 LD B,0FFH
3 LOOP : DJNZ LOOP
4 RET
5 END
```

Fig. 6.7. Il linguaggio assembly per un ciclo di conteggio che usa DJNZ.

L'istruzione LD B,FFH carica il registro B con il byte più grande possibile. Il nuovo passo è il "ciclo DJNZ", ed esso ha la parola "LOOP:" messa davanti. Questa è una "label", usata al posto di un indirizzo e sta ad indicare l'indirizzo a cui comincia l'istruzione. Con "LOOP:" messo davanti all'istruzione del ciclo DJNZ, la parola Loop indica l'indirizzo a cui si trova il byte d'istruzione DJNZ. Usando le parole in questo modo, evitiamo di dover pensare ai numeri d'indirizzo finché non scriviamo effettivamente il codice macchina. Se usiamo un assemblatore. normalmente non dovremo preoccuparcene affatto, l'assemblatore immette automaticamente i numeri d'indirizzo al posto delle parole della label.
Nel linguaggio assembly, sembra tutto molto semplice ed immediato. In effetti. lo sarebbe se usassimo un assemblatore, ma quando facciamo

l'assemblaggio manuale, non è così semplice. La ragione è che dobbiamo far seguire all'istruzione DJNZ un singolo byte che darà l'indirizzo del byte d'istruzione di DJNZ. Questo è un indirizzamento relativo al PC, per cui dobbiamo usare un byte con segno che possa essere addizionato all'indirizzo contenuto nel contatore di programma, per dare l'indirizzo del passo LD. Ne abbiamo visto la formula in precedenza, ma rivediamone ancora il metodo. Supponendo di porre il primo byte del programma all'indirizzo F001H, l'indirizzo dell'istruzione DJNZ sarà F003H, e il byte di spostamento sarà a F004H. F003H è l'indirizzo di sorgente, da dove veniamo, e F003H è anche l'indirizzo di destinazione, dove vogliamo andare. Sottraendo i numeri della sorgente dalla destinazione, otteniamo 0, e sottraendo ancora 2 da questo risultato, otteniamo —2. La cifra —2 equivale a FE esadecimale, per cui questo è il byte di spostamento che deve seguire il codice d'istruzione DJNZ. Se avete dei dubbi su questi numeri d'indirizzo, scriveteli in forma decimale. I programmatori che hanno la mania di usare la codifica esadecimale (si, come me), comprano dei calcolatori tascabili speciali che eseguono le operazioni aritmetiche nella forma esadecimale, ma dovete proprio sentirne l'esigenza per comprarne uno, anche perché il loro costo è abbastanza elevato.
Ora è il momento di provare questo ciclo in un programma. La Fig. 6.8 mostra un programma che immette, con la funzione POKE, il codice macchina in memoria e poi esegue il ciclo del codice macchina. Esegue poi la stessa azione anche in BASIC. Quando farete girare il programma, vedrete che il codice macchina non sembra avere un vantaggio apprezzabile. Infatti, ambedue le versioni eseguono il conteggio da 255 (FFH esadecimale) a 0 abbastanza rapidamente. Ciò che mette in grado il BASIC di effettuare il conteggio assai rapidamente è l'uso dei numeri interi ma, in effetti, la maggior parte del tempo impiegato è dovuto alla stampa sullo schermo. I risultati sono quindi ingannevoli, e se vogliamo vedere quanto sia più veloce il codice macchina rispetto al BASIC, dobbiamo usare conteggi molto più lunghi, in modo che i tempi di ritardo dovuti alle azioni come la stampa non siano significativi. Prima di tutto, però, dobbiamo scoprire come eseguire un conteggio più lungo in codice macchina.

```
10 CLEAR 200,&HF000
20 A!=&HF001
30 FOR N%=0 TO 4:READ D$
40 POKE A!+N%,VAL("&H"+D$) :NEXT
50 DEF USR=&HF001
60 PRINT"INIZIO CONTEGGIO...";
70 A=USR(0) :PRINT"FINE"
80 PRINT"ORA LO STESSO CONTEGGIO IN BASIC..."
90 INPUT"PREMETE RETURN
   PER INIZIARE";A$:PRINT"INIZIO...";
```

```
100 FOR Q%=255 TO 0 STEP —1 :NEXT
110 PRINT"FINE"
120 DATA 06,FF,10,FE,C9
```

Fig. 6.8. Una versione in BASIC del programma di conteggio in codice macchina, insieme ad un conteggio analogo in BASIC. Il conteggio è troppo veloce per misurare il tempo impiegato in tutti e due i casi.

UNO SGUARDO AI CICLI PIU' LUNGHI

Il modo più ovvio di eseguire un conteggio alla rovescia più lungo, in codice macchina, è di caricare una delle coppie di registri ed eseguire su essa il conteggio. Sfortunatamente, non c'è nessuna istruzione equivalente a DJNZ per le coppie di registri. Altrettanto sfortunatamente, benché sia possibile usare l'istruzione DEC con questi registri, l'azione di DEC non modifica i flag del registro di stato, per cui non possiamo usare JR NZ dopo DEC perché il ciclo ricominci fino al termine del conteggio. Questa mancanza da parte dei progettisti dello Z80 può essere rimediata con un pizzico di astuzia nella programmazione. Se carichiamo il contenuto di una metà di una coppia di registri in A, ed eseguiamo l'azione OR con l'altra metà, soltanto una condizione darà zero, cioè quando tutte e due le metà della coppia di registri contengono zero. Per esempio, supponete di effettuare il conteggio alla rovescia in BC (la coppia di registri più adatta per questa azione).
Se eseguiamo LD A,B, l'accumulatore sarà caricato da B. Eseguendo poi OR C, se c'è un 1 in qualsiasi parte del registro A o del registro C, ci sarà un 1 nel risultato (immagazzinato in A). Soltanto se ambedue i registri sono zero il risultato sarà zero. L'importanza di tutto questo è che uno zero in A, dopo che è stata eseguita l'azione OR C, modificherà i flag.

```
1                          ORG   0F000H
2                          LOAD  0F000H
3 F000 01FFFF              LD    BC,0FFFFH
4 F003 0B        LOOP:     DEC   BC
5 F004 78                  LD    A,B
6 F005 B1                  OR    C
7 F006 20FB                JR    NZ,LOOP
8 F008 C9                  RET
9                          END
```

Fig. 6.9. Un programma di conteggio che usa la coppia di registri BC. Esso richiede maggior tempo per la sua esecuzione.

Possiamo quindi far seguire all'istruzione OR C l'istruzione JR NZ perché venga eseguita l'azione del ciclo.
Ora vediamo il ciclo, nella Fig. 6.9. La prima azione è di caricare la coppia di registri BC con il numero FFFFH, 65535 esadecimale. La label "LOOP" indica dove inizia il ciclo, e in questa riga la coppia di registri BC viene decrementata. Il risultato viene poi controllato usando LD A,B come avevamo descritto; segue poi JR NZ,LOOP perché il ciclo ricominci se il conteggio non ha raggiunto lo zero. Il programma ritorna al BASIC dopo la fine del conteggio alla rovescia. La Fig. 6.10 illustra il programma BASIC che immette i bytes in memoria con POKE. Questa volta, dovete premere RETURN per iniziare ogni conteggio, e noterete una netta differenza tra la versione in codice macchina e la versione in BASIC. La versione in codice macchina termina in circa mezzo secondo, è ancora quasi troppo veloce per misurarne il tempo d'esecuzione. La versione BASIC impiega circa 32 secondi, nonostante sia stato dato ad essa ogni vantaggio. Il conteggio BASIC va da —32768 a +32766 in modo da permettere l'uso degli interi, e il conteggio è inferiore di uno. Questo perché un conteggio fino a 32767 provocherebbe un messaggio d'errore del programma. In questo esempio, quindi, il codice macchina è circa 64 volte più veloce del BASIC! Se avessimo usato un conteggio BASIC di N!=65536 TO 0 STEP —1, il conteggio BASIC sarebbe stato molto più lento, circa 75 secondi. La versione in codice macchina sarebbe perciò circa 150 volte più veloce, in questo caso. Questo è il tipo di vantaggio in velocità d'esecuzione che potete aspettarvi quando avete a che fare con un programma che viene eseguito solo in codice macchina.

```
10 CLEAR 200,&HF000
20 A!=&HF001
30 FOR N%=0 TO 7:READ D$
40 POKE A!+N%,VAL("&H"+D$) :NEXT
50 DEF USR=&HF001
60 CLS:INPUT"Premete RETURN
   per incominciare..." ;A$:PRINT"INIZIO...";
70 A=USR(0) :PRINT"FINE"
80 PRINT"Ora lo stesso conteggio in BASIC"
90 INPUT"Premete RETURN
   per incominciare..." ;:PRINT"INIZIO"
100 FOR N%=—32768! TO +32766:NEXT
110 PRINT"FINE"
120 DATA 01,FF,FF,0B,78,B1,20,FB,C9
```

Fig. 6.10. Il programma BASIC che utilizza POKE per il conteggio alla rovescia della coppia di registri BC, ed un conteggio BASIC a confronto. È subito evidente la maggior velocità del codice macchina!

Se cercate di misurare il tempo di esecuzione della versione in codice macchina con un cronometro, avrete un risultato molto impreciso a causa del tempo richiesto per avviare e arrestare il cronometro. Possiamo tuttavia trovare *esattamente* quanto tempo impiega un conteggio in codice macchina. La frequenza di clock per il computer MSX è di 3,58 MHz, cioè 3,58 *milioni* di impulsi di clock al secondo, per cui il tempo fra i due impulsi è approssimativamente 0,28 *milionesimi* di secondo. Ora, per ogni istruzione dello Z80, occorre un tempo misurato in termini di numero di cicli di clock. Questi tempi sono mostrati dettagliatamente nell'Appendice G, e la Fig. 6.11 mostra quanto tempo occorre per ogni istruzione nel ciclo in codice macchina. Questo è approssimato, perché l'istruzione JR Z impiega meno tempo l'ultima volta che viene eseguita, quando non deve di nuovo effettuare il ciclo. Tuttavia, la correzione è *molto* piccola e la cifra di 26 cicli di clock per loop (ciclo) può essere ritenuta valida. 26 cicli di clock, con 0,28 milionesimi di secondo per ogni ciclo, ci dà un tempo di 7,28 milionesimi di secondo per ogni loop, per cui 65536 loop impiegano 0,47 secondi, meno di mezzo secondo. Loop di conteggio come questo possono essere impiegati per ottenere ben precisi ritardi, poiché la frequenza del clock è controllata da un cristallo di quarzo molto più preciso di quello usato nei moderni orologi. Questi ritardi sono ampiamente usati nelle routines della ROM. La routine del suono, BEEP, l'input e l'output da cassetta e le routines di stampa, tanto per citarne qualcuna, fanno uso di una precisa temporizzazione (timing) per portare a termine le loro azioni e i cicli (loop) come quello da noi usato sono la base di questa precisione.

Operazione	Tempo	Commento
DEC BC	6	Come per INC
LD A,B	4	
OR C	4	Come per ADD C
JR NZ	12	In ogni loop
TOTALE	26	Nel loop

Fig. 6.11. I tempi per ogni istruzione possono essere sommati per trovare quanto tempo occorre per ogni loop. I tempi sono espressi in cicli di clock.

L'USO DEI CICLI (LOOP)

Ora che abbiamo visto la velocità di un ciclo in codice macchina quando è usato per originare un ritardo, dobbiamo vedere delle altre applicazioni dei cicli. Una delle applicazioni più ovvie di un loop è per le operazioni di schermo, perché il BASIC impiega un tempo assai lungo per eseguire

delle azioni di schermo. Provate, per esempio, un programma BASIC che riempie lo schermo con la lettera "A", come mostra la Fig. 6.12. Questo programma in BASIC impiega circa sei secondi per riempire lo schermo con la lettera "A". Potremmo supporre che la versione in codice macchina dello stesso programma impieghi meno tempo, ma il problema è di come riuscire a farlo.

```
10 CLS
20 FOR ROW=0 TO 22
30 FOR COL=0 TO 36
40 LOCATE COL,ROW:PRINT"A";
50 NEXT:NEXT
```

Fig. 6.12. Un programma BASIC che riempie lo schermo con la lettera "A".

Ci sono due modi: uno difficile ed uno facile. Il modo difficile è di raccogliere sufficienti informazioni sul computer per usare direttamente gli indirizzi di schermo. Il modo facile è trovare la routine nella ROM che esegue l'azione di "scrivere sullo schermo". Ambedue i metodi richiedono delle informazioni, e la difficoltà del primo metodo consiste nel fatto che non sappiamo come usare la memoria di schermo, per il momento. Sappiamo che possiamo usare VPOKE in BASIC per inviare dei bytes alla memoria di schermo, ma non sappiamo come convertire tutto questo in codice macchina (ma vedete al Cap. 9!). Se memorizziamo semplicemente dei bytes a partire dall'indirizzo 0, non accadrà niente, perché l'indirizzo 0 è nella ROM. Prima di poter usare l'indirizzamento di schermo, allora, dobbiamo sapere come accedere agli indirizzi della RAM di schermo. Ecco che cosa rende difficile questo metodo! A prima vista, il metodo "facile" non sembra molto migliore, perché dovete conoscere un indirizzo e alcune informazioni sui registri. In effetti, ci sono alcune informazioni qui. La routine che comincia all'indirizzo 6CH ripulirà lo schermo, e la routine che comincia ad A2H scriverà un carattere sul primo spazio di schermo disponibile. Il codice ASCII del carattere deve essere nell'accumulatore prima di usare quest'ultima routine.
Questo è un passo avanti, ma ora dobbiamo sapere come usare queste routines. Il linguaggio assembly ha due istruzioni, CALL e RET, che sono praticamente l'esatto parallelo delle istruzioni BASIC GOSUB e RETURN. CALL deve essere seguita da un indirizzo d'inizio, che può essere nella ROM o nella RAM. Il suo effetto sarà di saltare a quell'indirizzo, mentre l'indirizzo corrente verrà immagazzinato nella memoria di "stack". La subroutine chiamata da CALL dovrebbe terminare in RET (o una variazione di RET come RETZ, che significa "ritorna se il flag di zero è posizionato a 1"). Quando viene eseguito il comando RET, lo Z80 prenderà l'indirizzo dallo stack, lo incrementerà e poi continuerà nell'ese-

cuzione del programma, come se non fosse accaduto niente. Quando dico che CALL e RET sono il parallelo di GOSUB e RETURN, fra parentesi, intendo un parallelo molto stretto, perché GOSUB/RETURN viene eseguita richiamando CALL e RET in codice macchina. L'interprete BASIC, come ricorderete, non è altro che un lunghissimo programma in codice macchina!
Con tutto questo da digerire, facciamo lo schema di un programma che riempia lo schermo con la lettera "A". La Fig. 6.13 mostra lo schema, con la delineazione dei punti principali sulla parte sinistra, come al solito.

```
INIZIO

RIPULISCI    { usa CALL 06CH

             ( metti 41 H in A
SCRIVI       | posiziona il contatore a 851 { usa BC=353H
851 VOLTE   <  X CALL 0A2H
ASC 65       | decrementa il cont. e salta  ( DEC & usa
             ( a X se non è zero            ( LD A,B OR C JR NZ

FINE         { RET
```

Fig. 6.13. Uno schema per un programma che userà il codice macchina per riempire lo schermo.

Vogliamo ripulire lo schermo, scrivere 851 volte il codice ASCII 65 sullo schermo, e ritornare al BASIC. La cifra 851 viene dalle righe e dalle colonne che usiamo. Ci sono 23 righe, ognuna di 37 caratteri nel normale modo di scrittura e 23X37 fa 851. La fase successiva della stesura mostra altri dettagli. Lo schermo sarà ripulito usando CALL 06CH. L'azione di porre la lettera "A" sullo schermo sarà fatta caricando l'accumulatore con il numero 65 (codice ASCII per "A"), posizionando un contatore a 851, stampando sullo schermo con CALL 0A2H, e poi decrementando ed effettuando un test sul contatore. Se il contatore non ha raggiunto zero, il programma ritorna al punto indicato da (X), la chiamata (CALL) per la stampa sullo schermo. Notate che dobbiamo caricare l'accumulatore ad ogni passo del loop, perché il numero 65 sarà sostituito quando viene eseguito il passo LD A (che serve a controllare il contatore). Normalmente, cercheremo di evitare di ricaricare un registro in un loop, perché si spreca del tempo. L'ulteriore dettaglio è poi mostrato nell'altro insieme di parentesi, che mostra i numeri in esadecimale, e ci ricorda di usare LD A,B e OR C seguiti da JR NZ per il controllo del ciclo. Ora possiamo scrivere il programma in linguaggio assembly.

```
1 ORG 0F000H
2 CALL 06CH; pulisci schermo
3 LD BC,0353H; contatore
4 LOOP:LD A,41H; lettera A
5 CALL 0A2H; stampala
6 DEC BC; decrem contatore
7 LD A,B; carica byte alto
8 OR C; OR con byte basso
9 JR NZ LOOP; ciclo fino a 0
10 RET; ritorna al BASIC
11 END
```

Fig. 6.14. Il programma in linguaggio assembly per riempire lo schermo.

Questo è riportato nella Fig. 6.14 e segue esattamente il nostro schema precedente, per cui non c'è bisogno di fare ulteriori commenti. L'unica novità è l'uso del punto e virgola nel linguaggio assembly, che corrisponde all'uso di REM in BASIC. Ciò che segue il punto e virgola sarà stampato nel listato, ma verrà ignorato dall'assemblatore. Poiché, per il momento, stiamo facendo un assemblaggio manuale il punto e virgola è solo un modo utile di ricordarci che cosa fa il programma. Mettete sempre questi commenti, perché quando esaminate un programma in linguaggio assembly, privo di ogni commento, dopo qualche settimana, non avrete il minimo indizio per sapere che cosa dovrebbe fare! Cercare di seguire un programma BASIC che avete scritto un anno fa, non è niente in confronto! La Fig. 6.12 mostra il programma nella versione BASIC, con l'uso di POKE, e con un passo INKEY$ che vi permette di vedere quanto sia veloce il codice macchina. Quando usate questo, la velocità con cui lo schermo viene riempito con la lettera è limitata solo dalla velocità della routine di stampa sullo schermo MSX. Non è proprio così veloce come potrebbe essere accedendo direttamente agli indirizzi di schermo, ma è molto più veloce del BASIC. Per molti scopi, questa velocità di riempimento dello schermo sarà più che sufficiente. Molte azioni sullo schermo in modo di scrittura possono essere organizzate con l'aiuto di questo tipo di routine, e possiamo ora vedere un'estensione di questo programma.

```
10 CLEAR200,&HEFFF
20 A!=&HF000
30 FOR N%=0 TO 16:READ D$
40 POKE A!+N%,VAL("&H"+D$) :NEXT
50 DEF USR=&HF000
60 PRINT"PREMI UN TASTO"
```

```
70 K$=INKEY$:IF K$=" " THEN 70
80 A=USR(0)
100 DATA CD,6C,00,01,53,03,3E,41,CD,A2,00,0B,78,B1,20,F6,C9
```

Fig. 6.15. La versione BASIC del programma che riempie lo schermo.

Che cosa ne direste se, invece di riempire lo schermo con la lettera "A", stampassimo tutto l'insieme dei caratteri? Come vedrete, anche questa è un'azione assai lenta se scrivete un programma BASIC per eseguirla. In linguaggio Assembly, non è molto diversa da quella che abbiamo appena visto. Questo è importante, perché se sviluppate un nuovo programma utilizzando quello che avete già provato, avrete buone probabilità che il vostro nuovo programma funzioni correttamente.

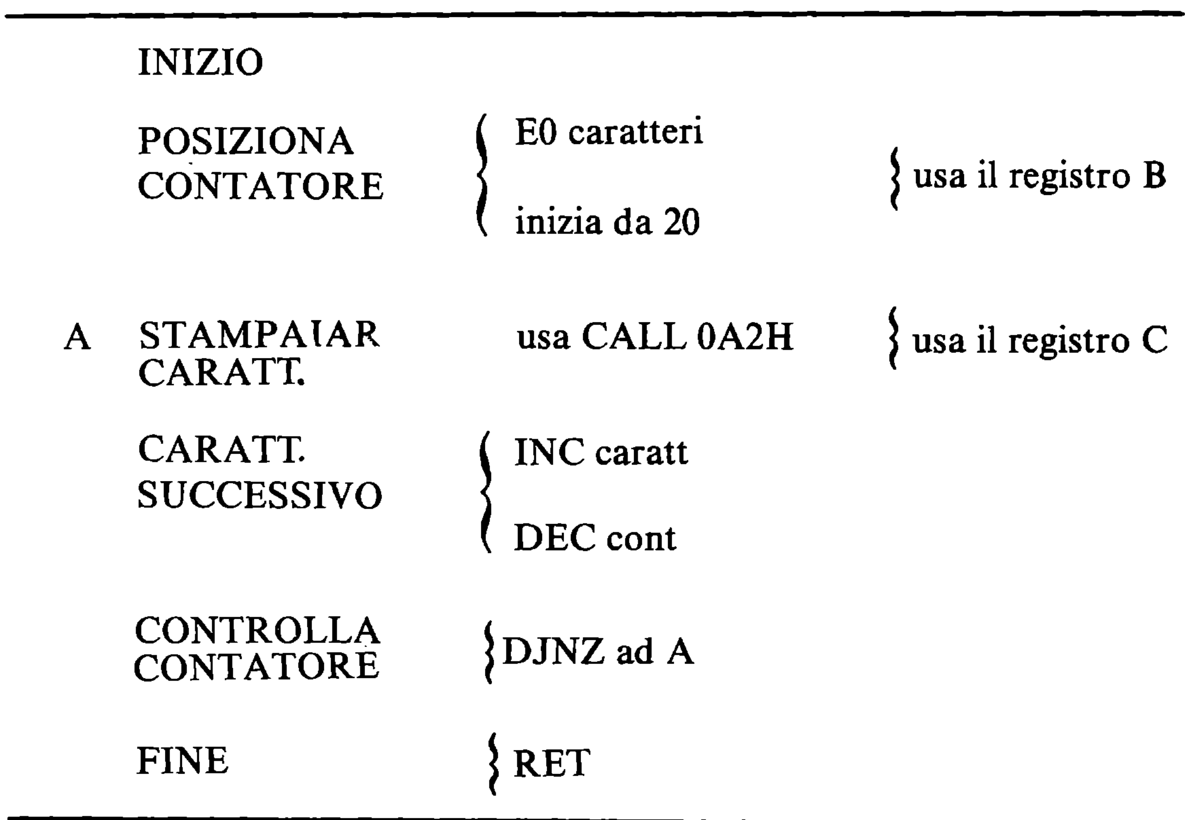


Fig. 6.16. Uno schema per stampare tutto l'insieme dei caratteri del computer MSX.

La Fig. 6.16 mostra la fase di progettazione di questo programma. Nello schema generale, dovremo posizionare un contatore, stampare un carattere, scegliere un altro carattere e ripetere tutto questo fino alla fine del conteggio. In maggior dettaglio, dovremo stampare E0H caratteri (224 decimale), usando un conteggio da DFH a zero. Il numero di codice del primo carattere è 20H, il carattere della barra spaziatrice. La stampa può essere fatta come al solito. con CALL 0A2H, e ad ogni ciclo dovremo incrementare il numero del carattere e decrementare il contatore. Il con-

trollo può essere fatto usando DJNZ, perché il conteggio è inferiore a FFH, per cui potrà essere utilizzato il registro B. Questo punto è annotato nella terza parte del dettaglio, e possiamo anche stabilire che un registro contenga il numero di codice del carattere. Potremmo immetterlo in A, ma se volessimo estendere questo programma in modo da coprire tutto lo schermo, non potremmo usare A, per cui useremo subito C. Questo schema di progetto conduce al programma della Fig. 6.17.

```
1 ORG 0F000H
2 LD B,0DFH; conteggio in B
3 LD C,20H; carattere in C
4 LOOP:LD A,C; carattere in A
5 CALL 0A2H; stampalo
6 INC C; carattere seguente
7 DJNZ LOOP; fino a zero
8 RET
9 END
```

Fig. 6.17. Il linguaggio assembly per lo schema di programmazione della Fig. 6.16.

Il programma BASIC, che utilizza POKE per immettere il codice macchina in memoria, è illustrato nella Fig. 6.18, ed eseguirà l'azione di stampare tutto l'insieme dei caratteri. È veloce e potete paragonarne la velocità a quella di un programma BASIC che esegua la stessa azione.

```
10 CLEAR 200,&HEFFF
20 A!=&HF000
30 FOR N%=0 TO 11:READ D$
40 POKE A!+N%,VAL("&H"+D$):NEXT
50 DEF USR=A!
60 A=USR(0)
70 DATA 06,DF,0E,20,79,CD,A2,00,0C,10,F9,C9
```

Fig. 6.18. Il programma BASIC che immette in memoria con POKE il programma della Fig. 6.16.

Un gran vantaggio di avere un programma in codice macchina è la sua indipendenza dal BASIC. Per esempio, supponete di sostituire la riga 60 del programma della Fig. 6.18 con la riga:

```
60 KEY 1,"A=USR(0)"+CHR$(13)
```

e di aggiungere la riga 65 NEW. L'effetto sarebbe di immettere il codice macchina in memoria e attivare il tasto funzione F1, per far eseguire il

programma. Il programma poi si autocancellerebbe, lasciando in memoria il codice macchina e il tasto funzione caricato. Potrete quindi caricare un altro programma BASIC, ma ogni volta che volete visualizzare tutto l'insieme dei caratteri, basterà premere il tasto F1. Magico!

LEGGERE LA TASTIERA

La routine che si trova a 0A2H è molto utile per la stampa di un carattere, ma potrebbe essere ancora più utile se potessimo utilizzare anche la tastiera. In effetti, c'è una routine nella ROM per fare proprio questo. È a 09FH, ed è la routine di INKEY$. In altre parole, quando la richiamate, leggerà la tastiera e poi il programma proseguirà. Se un tasto viene premuto in quel momento, il codice ASCII per il carattere corrispondente a quel tasto sarà posto nell'accumulatore. Se non viene premuto nessun tasto, allora l'accumulatore contiene uno zero e il programma continua. Proprio come con INKEY$, quindi, dovremo porre questa CALL in un ciclo se vogliamo utilizzarla. Un ciclo di questo tipo viene chiamato "ciclo di attesa", perché tiene la macchina in uno stato di attesa finché non fate qualcosa, in questo caso, premete un tasto.
Cominciamo nel modo giusto facendo uno schema del programma. Questo è mostrato nella Fig. 6.19.

INIZIO		
PRELEVA CARATTERE	(X) controlla tastiera	usa 09FH
	se zero, allora (X)	OR A per posizionare i flag
STAMPALO	usa CALL 0A2H	
FINE	RET	

Fig. 6.19. Uno schema per la lettura della tastiera e la stampa del carattere corrispondente al tasto premuto.

Il primo passo è di prelevare il carattere, il secondo è di stamparlo, Ora vediamo in dettaglio ognuno di questi passi. Prelevare il carattere significa controllare la tastiera e poi ritornare al ciclo se il numero ottenuto è zero. Ancora più dettagliatamente, useremo la routine a 09FH per controllare la tastiera, e dovremo eseguire l'istruzione OR A in modo da posizionare i flag prima di effettuare un test con il passo JR Z. La parte di "Stampa del Carattere" viene eseguita con la routine 0A2H come prima. La Fig. 6.20 mostra la versione in linguaggio assembly di tutto questo.

```
1 ORG 0F000H
2 LOAD 0F000H
3 CALL 06CH; CLS
4 TASTO:CALL 09FH; controllo tasti
5 OR A; imposta i flag
6 JR Z,TASTO; controllo carattere
7 CALL 0A2H; stampalo
8 RET
9 END
```

Fig. 6.20. Il linguaggio assembly che usa le chiamate alle routine ROM.

La routine a 06CH è usata per ripulire lo schermo, e poi inizia il ciclo. Nel ciclo, CALL 09FH controllerà la tastiera e metterà un byte nell'accumulatore. Questo byte sarà zero se non viene premuto nessun tasto, altrimenti l'accumulatore conterrà il codice ASCII per quel tasto. Abbiamo bisogno del passo OR A perché l'azione di caricamento dell'accumulatore non modifica i flag, e poi possiamo effettuare il controllo sull'accumulatore e ricominciare il ciclo se il byte è zero. Quando viene trovato un byte diverso da zero, viene stampato con la routine CALL 0A2H.
La Fig. 6.21 mostra il programma sotto forma di un insieme di istruzioni BASIC POKE.

```
10 CLEAR 200,&HEFFF
20 A!=&HF000
30 FOR N%=0 TO 12:READ D$
40 POKE A!+N%,VAL("&H"+D$):NEXT
50 DEF USR=&HF000
60 A=USR(0)
70 DATA CD,6C,00,CD,9F,00,B7,28,FA,CD,A2,00,C9
```

Fig. 6.21. La versione con le istruzioni BASIC POKE.

Quando il programma viene eseguito, la pressione di un tasto farà apparire una lettera o un altro carattere sullo schermo. Non tutti i tasti daranno

un risultato (trovate quelli che non lo danno), e alcuni tasti produrranno degli effetti inaspettati. Provate il tasto ESC, per esempio, e provate anche l'effetto dato dal tasto SHIFT premuto insieme ad altri tasti. Utilizzando queste routines della ROM, possiamo eseguire un sorprendente numero di azioni in codice macchina che richiedono un input o un output. Già che siamo in argomento, sarà utile vedere in che cosa consistano gli indirizzi di queste CALL. Le effettive routines che ripuliscono lo schermo, prelevano il carattere e lo stampano *non* sono agli indirizzi 06CH,09FH e 02AH. Ciò che si trova a questi indirizzi è un'istruzione JP. JP è l'altra forma del salto dello Z80 e richiede un indirizzo completo di due bytes dopo di essa. Se usate PEEK per vedere che cosa c'è in memoria attorno a questi indirizzi, troverete un altro indirizzo che segue il codice C3H (195 decimale) di JP. La routine inizia appunto a quest'altro indirizzo. La ragione di tutto questo è che è possibile, in tal modo, eseguire eventuali cambiamenti nella ROM. Se viene cambiata una ROM, occorre la certezza che le macchine che la usano possano ancora usare correttamente le nuove routines.

Per esempio, il JP a 06CH è JP 0E 05, all'indirizzo 050EH, 1294 decimale. Supponiamo che la ROM sia cambiata, in modo che questa routine cominci a 051AH. Ogni salto al vecchio indirizzo 050EH sarebbe sbagliato ma, se fossero cambiati anche i bytes a 06DH e 06EH in modo da contenere il nuovo indirizzo, tutte le chiamate a 06CH individuerebbero ancora l'indirizzo corretto per la routine. Questo viene spesso denominato "blocco terminale", facendo il paragone con i circuiti elettrici, dove i collegamenti possono essere fatti ad un blocco terminale e cambiati in quel punto, se necessario. Un blocco terminale della ROM non è, però, così utile come uno nella RAM. Un blocco terminale della RAM ci permette di immettere una nostra istruzione di JP, per cui la normale azione della macchina viene deviata ad una routine di nostra scelta. Ma è qualcosa che vedremo più avanti.

Nel frattempo, che ne dite di sviluppare questo frammento di azione di programma in qualcosa di più utile? Supponiamo di voler stampare sullo schermo tutto quanto è stato immesso prima di un RETURN. Ciò significa usare due cicli, uno per controllare la tastiera e ripetere il ciclo finché non viene premuto un tasto e l'altro per controllare il carattere del tasto per vedere se è il codice di un RETURN (0DH, 13 decimale). La Fig. 6.22 mostra lo schema per quest'azione, che contiene una novità.

Sull'ultima parte delle caratteristiche generali vedrete un simbolo che sembra un + racchiuso da un cerchietto. Questo significa OR Esclusivo, e quando viene usato in uno schema di programma, significa che o un'affermazione o l'altra deve essere vera. Non possono essere vere tutte e due, e nemmeno ambedue false. Ne consegue che il programma dovrà ripetere il ciclo o terminare.

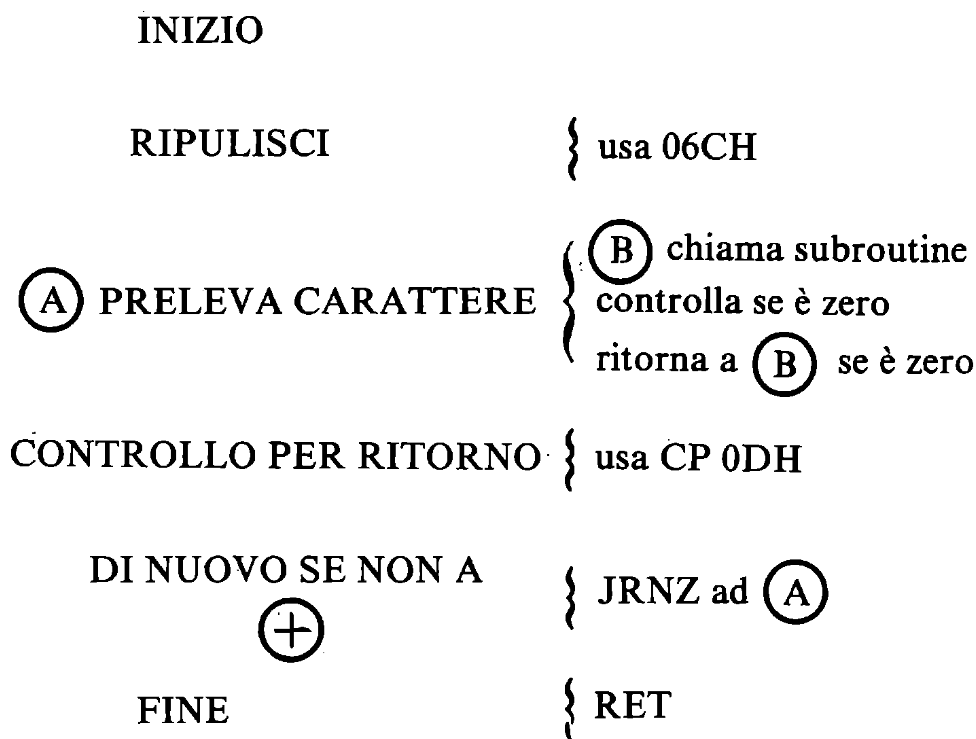


Fig. 6.22. Uno schema per controllare la tastiera e leggere ogni tasto finché non viene premuto il tasto RETURN.

La Fig. 6.23 mostra il linguaggio assembly che corrisponde a quest'azione, che è diretta.

```
 1 ORG 0F000H
 2 CR: EQU 0DH
 3 CALL 06CH;CLS
 4 TASTO:CALL 09FH;controllo tasti
 5 OR A;imposta i flag
 6 JR Z, TASTO;controlla carattere
 7 CALL 0A2H;stampalo
 8 CP CR;è RETURN?
 9 JR NZ,TASTO;indietro se non è
10 RET;ritorna al BASIC
11 END
```

Fig. 6.23. La versione in linguaggio assembly, che è uno sviluppo della Fig. 6.20.

La versione BASIC della Fig. 6.24 usa DEF USR=A! alla riga 50. Poiché A! è stata assegnata a &HF000 alla riga 20 possiamo usare A! per rappresentare questo numero nel corso del programma.

```
10 CLEAR 200,&HEFFF
20 A!=&HF000
30 FOR N%=0 TO 16:READ D$
40 POKE A!+N%,VAL("&H"+D$):NEXT
50 DEF USR=A!
60 A=USR(0)
70 DATA CD,6C,00,CD,9F,00,B7,28,FA,CD,A2,00,FE,
   0D,20,F3,C9
```

Fig. 6.24. Il programma BASIC che utilizza POKE per la Fig. 6.23.

Quando il programma viene eseguito, lo schermo è ripulito, e vedrete il cursore all'angolo in alto a sinistra. Potete poi digitare quello che volete, e apparirà sullo schermo. Quando premete RETURN, non apparirà nessun "Syntax error", come vi potreste aspettare di vedere, immettendo una qualsiasi cosa diversa da un comando BASIC, ed apparirà invece il messaggio "OK" sull'angolo in alto a sinistra, al posto di quello che avevate scritto. Questo accade perché avete battuto le lettere usando il codice macchina. Quando usate il codice macchina, "bypassate" completamente il BASIC. Poiché quello che avete battuto non è stato controllato dall'interprete BASIC della ROM, non ci può essere nessun messaggio di "Syntax error". Nello stesso modo, anche le routines BASIC che controllano dove debba apparire "OK", sono state bypassate e il messaggio appare quindi alla posizione "home", cioè nell'angolo in alto a sinistra. Normalmente, quando usate una routine di questo tipo, non state ritornando al BASIC in questa fase, per cui questo non rappresenta alcun problema.

CAPITOLO 7

Cassette e parametri

SALVATELO

A questo punto, ora che i nostri programmi stanno diventando un po' più lunghi ed interessanti, è opportuno vedere come si salva su nastro un programma in codice macchina. Non siete tuttavia obbligati a farlo. Tutti i nostri programmi, finora, sono stati espressi nella forma di un programma BASIC che assembla in memoria i numeri del codice macchina. Ovviamente, potete salvare semplicemente questo programma BASIC, ed esso creerà il codice macchina per voi quando lo desiderate. Quando usate un "misto" di BASIC e codice macchina, questo è il modo ideale di salvare e caricare i bytes del codice macchina. Ci sono tuttavia delle volte in cui avete bisogno di tutta la memoria possibile, e un altro programma BASIC sarebbe il benvenuto come un elefante in una capsula spaziale. Potreste anche voler mettere su cassetta un programma che è un insieme di BASIC e di codice macchina, rendendone difficile la copiatura. In un caso o nell'altro, vorrete quindi registrare direttamente i bytes immagazzinati in memoria. Le normali istruzioni BASIC del computer MSX possono far questo ma, come vedremo, in un modo assai diverso da LOAD e SAVE.
Come molte altre macchine, il computer MSX ha delle speciali istruzioni BASIC per salvare o caricare un programma in codice macchina. Ci sono dei notevoli vantaggi in questa procedura. Uno di questi è che i bytes del codice macchina occupano molto meno memoria delle istruzioni assembler o delle righe BASIC che generano il codice. Se generate il codice separatamente, usando una parte in linguaggio assembly, poi registrate il codice (come parte distinta dalla registrazione della versione in linguaggio assembly), potrete porre quel codice in un qualsiasi programma BASIC. Questo vi darà il vantaggio di avere una parte in codice macchina, ma senza occupare la quantità di spazio richiesta dal linguaggio assembly. Un altro vantaggio è che il codice macchina occupa pochissimo spazio sulla cassetta, ed il tempo di caricamento è molto breve. Un terzo vantaggio è che il vostro codice macchina è molto più al sicuro da sguardi indiscreti —un veloce LIST durante una serata con gli amici non rivelerà molto all'aspirante copiatore! Niente è mai totalmente sicuro, ma l'uso del codice macchina direttamente dalla cassetta renderà il programma meno facile da copiare in fretta, a differenza dell'uso di un programma BASIC che immette il codice macchina in memoria con POKE. Nel prossimo capitolo, vedremo come si genera il codice macchina dal linguaggio assembly

usando l'assemblatore ZEN. Se usate un programma BASIC POKE, potete fare in modo che si cancelli da solo, mettendo NEW all'ultima riga invece di END. In questo modo, il codice sarà immesso in memoria da POKE, ma il programma che usava la funzione POKE scomparirà, insieme ad ogni traccia di indirizzi e di dati.
Al lavoro, dunque. Il codice macchina su cassetta (o su disco, quando è possibile), usa i comandi modificati BSAVE e BLOAD (B per Byte). Cominceremo con l'istruzione BSAVE, ed userò come esempio il programma che abbiamo visto nel capitolo precedente — quello che stampa tutti i caratteri sullo schermo (Fig. 6.17). Come è logico supporre per un'operazione in codice macchina, BSAVE richiede molte più informazioni del normale comando CSAVE in BASIC. Dopo BSAVE, dovete immettere un "nome di file", un indirizzo d'inizio e un indirizzo di fine. Questi sono i requisiti *minimi;* vedremo più avanti le altre possibilità! Il nome di file deve cominciare con CAS: e può accettare altri sei caratteri, proprio come qualsiasi nome di file su cassetta in BASIC.
Una virgola deve seguire le virgolette di chiusura del nome di file, e poi dovete mettere l'indirizzo d'inizio. L'indirizzo d'inizio, decimale o esadecimale, è l'indirizzo del primo byte del vostro programma in codice macchina. Se usate la forma esadecimale, dovete mettere &H davanti al numero esadecimale. Questo è seguito poi da un'altra virgola, e dall'indirizzo di fine, sempre decimale o esadecimale. Questo è l'indirizzo dell'ultimo byte del programma. Il programma della Fig. 6.17 usa &HF000 come indirizzo d'inizio e &HF00B come indirizzo di fine, per cui potremmo registrarlo dando l'istruzione:

BSAVE"CAS:CHARS",&HF000,&HF00B

Ma non c'é niente di meglio, per imparare, che provare per conto vostro. Caricate il programma BASIC che origina i caratteri e togliete la riga A=USR(0). Date il comando RUN, in modo che il codice macchina si assembli, poi digitate l'istruzione BSAVE come è mostrato qui sopra. Assicuratevi che gli indirizzi corrispondano agli indirizzi del *vostro* listato, se avete usato dei numeri d'indirizzo diversi. È molto importante immettere correttamente questi indirizzi, specialmente l'indirizzo d'inizio. Se l'indirizzo di fine si trova diversi bytes oltre la fine del programma, ciò non ha importanza. Ma se l'indirizzo di fine è inferiore al valore reale, vi potreste trovare senza RET, o senza altre parti vitali, e questo vi causerebbe dei guai in seguito.
Il comando BSAVE permette una piccola, ma utile variazione. Infatti, potete usare un altro numero d'indirizzo dopo il numero d'indirizzo finale. Questo è "l'indirizzo di esecuzione", e può essere incluso se il programma in codice macchina non comincia con il primo byte. Al momento, non ci riguarda, ma potreste trovare programmi che, come primi

bytes, hanno dei bytes di dati, e il vero inizio potrebbe essere più avanti. L'indirizzo di esecuzione è l'indirizzo dell'inizio effettivo del programma. Se il programma comincia con il primo byte, non c'é alcun bisogno di includere questo indirizzo.
Porvate l'istruzione BSAVE, allora, sul vostro programma "CHARS", e poi eseguite l'ultima prova. Spegnete la macchina. Dovete farlo, perché questo è il modo più semplice di cancellare il codice macchina immagazzinato in memoria. Riaccendete, e poi digitate:

BLOAD"CAS:CHARS"

Questo è tutto, senza indirizzi. Quando premete RETURN, comparirà il consueto messaggio di caricamento da cassetta e vedrete che il caricamento è *molto* veloce, molto più veloce del caricamento del programma BASIC POKE. Potete poi dare l'istruzione di esecuzione del codice macchina battendo DEF USR=&HF000 e poi A=USR(0).
Non potete però dare il comando LIST, perché non c'é nessun programma BASIC in memoria.
Qualora pensiate di sapere ormai tutto, c'é ancora qualcosa da dire sull'istruzione BLOAD. Normalmente, quando usate BLOAD seguita solo dal nome di file, il codice macchina verrà caricato all'indirizzo che è stato specificato nell'istruzione BSAVE. Tuttavia, potete spostare un programma. Questo significa che potete caricarlo ad un diverso indirizzo di memoria. Lo spostamento dei bytes del programma *non* garantisce che il programma venga eseguito correttamente al nuovo indirizzo,ma in genere questo avviene per molti programmi; essi sono chiamati "programmi riallocabili". Per esempio, supponete di voler spostare il programma "CHARS" in modo che cominci all'indirizzo F020H. L'istruzione sarà la seguente:

BLOAD"CAS:CHARS",&H20

Questa istruzione caricherà semplicemente il programma; per farne iniziare l'esecuzione dovrete usare DEF USR=&HF020 e A=USR(0). Potete però usare una variante di BLOAD che lo caricherà e ne farà iniziare automaticamente l'esecuzione: basta far seguire il nome di file da una virgola e dalla lettera R. Per esempio, BLOAD"CAS:CHARS",R caricherà e farà eseguire il programma CHARS. Potete farlo seguire da un'altra virgola e aggiungere il numero di spostamento in memoria. In questo modo, BLOAD"CAS:CHARS",R,&H20 sposterebbe tutti gli indirizzi di 20H e poi farebbe eseguire il programma.

E ORA PRENDIAMO UN MESSAGGIO....

Dopo questo intermezzo sul salvataggio e il caricamento del codice macchina, ritorniamo ai programmi. Nel Capitolo 6, ricordate, avevamo scritto il programma che stampava i caratteri sullo schermo. Ora è il momento di trovare il modo per immettere qualcosa di più interessante sullo schermo, e le parole sembrano un inizio ragionevolmente semplice per questo tipo di programmazione. Che cosa dobbiamo fare? Bene, per cominciare, dobbiamo immagazzinare dei codici ASCII per le lettere da qualche parte nella memoria; non possiamo usare semplicemente una variabile di stringa come faremmo in BASIC. Dovremo conoscere l'indirizzo a cui viene immagazzinata la prima lettera, e dobbiamo anche sapere come arrestare l'elaborazione. Dopo aver fatto questo, dovremmo essere capaci di stabilire un ciclo che prenda un byte dallo "spazio di testo" (dove sono immagazzinati i codici delle lettere) e lo passi alla subroutine che stampa il carattere. Cominciamo, come al solito, con uno schema di programma. Non è così facile questa volta, perché abbiamo bisogno di terminare il ciclo in modo diverso. Potremmo contare il numero di lettere che vogliamo porre sullo schermo, ma voglio illustrarvi una tecnica diversa, questa volta —l'uso di un terminatore. Probabilmente vi è già familiare quest'idea dai programmi BASIC. Un "terminatore" è un byte che il programma può riconoscere come carattere speciale: uno che, per esempio, non fa parte di un messaggio. Un terminatore adatto per i numeri è 0, ma per le parole è più utile il codice 0DH, "ritorno carrello", per cui useremo questo. Possiamo farlo controllando l'accumulatore dopo che è stato usato per la routine di stampa. Dopo, notate bene, non prima, perché il ritorno carrello è un carattere valido, il cui effetto ci occorre nella stringa

INIZIO	
PRELEVA INDIRIZZO LETR	metti in HL
LEGGI & STAMPA	carica A da (HL) controlla terminatore incrementa HL stampa lettera
RIPETI SE NON TERMINATORE	finisci se terminatore
FINE	RET

Fig. 7.1. Lo schema di programma per stampare un messaggio sullo schermo.

di caratteri. Abbiamo bisogno anche del carattere dell'alimentazione riga (line—feed), 0AH, per assicurarci che il cursore si sposti di una riga dopo aver stampato la frase.
Il nostro schema è mostrato nella Fig. 7.1.
Ciò che dobbiamo fare è memorizzare un indirizzo per il messaggio che vogliamo stampare sullo schermo. Questo sarà l'indirizzo di una stringa di bytes dei codici ASCII. Ho scelto LETR come label di questa parte di memoria. Se usassimo un assemblatore, questa parte di linguaggio assembly potrebbe essere usata quasi così com'é. Nel linguaggio dell'assemblatore ZEN, DB significa "definisci i bytes", ed è usato per posizionare i bytes in memoria. Questo *non* fa parte del linguaggio assembly standard, semplicemente è una conveniente azione di questo assemblatore. Avete bisogno delle virgolette attorno alla stringa, come fareste quando definite una variabile di stringa, ma in questo caso usiamo non i doppi apici, ma gli apici semplici. Mettendo la label LETR davanti all'istruzione, ci assicuriamo che sia posta al corretto indirizzo d'inizio, F030H.
Dal momento che usiamo un programma BASIC con la funzione POKE per l'immissione in memoria, il problema che ora dobbiamo affrontare è di come immagazzinare i bytes a partire da questo indirizzo, usando solo il BASIC. Questo è assai semplice, con POKE, e lo vedremo poi. Ritornando al linguaggio assembly, la Fig. 7.2 mostra ciò di cui abbiamo bisogno.

```
 1  ORG 0F000H
 2  LD HL,LETR
 3  LOOP:LD A,(HL)
 4  INC HL
 5  CALL 0A2H
 6  CP 0DH
 7  JR NZ,LOOP
 8  RET
 9  LETR:DB 'ECCO IL TESTO',0AH
10  DB 0DH
11  END
```

Fig. 7.2. Il programma in linguaggio assembly per stampare il messaggio. DB viene usato solo con l'assemblatore ZEN.

La coppia di registri HL è caricata con l'indirizzo LETR. Nel ciclo, l'accumulatore viene caricato da (HL), cioé copia il primo codice ASCII del testo. L'indirizzo di HL è incrementato, poi viene richiamata la subroutine di stampa per stampare il carattere. Il codice del carattere viene poi confrontato con 0DH, il terminatore, e se il test ha esito negativo (se non è 0DH), il programma ricomincia il ciclo. Quando viene letto il carattere

0DH, viene stampato e il programma termina, ritornando al BASIC. La Fig. 7.3 mostra la versione BASIC con l'uso di POKE. Le righe dalla 30 alla 60 definiscono una stringa, con i caratteri 0AH e 0DH alla fine, e la immettono in memoria con POKE. Questo viene fatto con un ciclo che comincia all'indirizzo &HF000+&H30+0, e immette qui il primo codice ASCII. Incrementando l'indirizzo ad ogni passo, ed usando il contatore di ciclo (più 1) come numero da scegliere in MID$, possiamo immettere in memoria, in sequenza, ogni codice ASCII della stringa a partire dall'indirizzo &HF030. Così la stringa di bytes è pronta per il codice macchina. Possiamo poi immettere i bytes del codice macchina con DEF USR, per scegliere l'indirizzo e richiamare il codice macchina. L'azione CLS è eseguita in BASIC questa volta.

```
 10 CLEAR 200,&HEFFF
 20 A!=&HF000
 30 T$="ECCO IL TESTO"+CHR$(11)+CHR$(13)
 40 FOR N%=0TO LEN(T$) —1
 50 POKE A!+&H30+N%,ASC(MID$(T$,N%+1,1))
 60 NEXT: FOR N%=0 TO 12:READ D$
 70 POKE A!+N%,VAL("&H"+D$):NEXT
 80 CLS:DEF USR=A!
 90 A=USR(0):PRINT
100 DATA 21,30,F0,7E,23,CD,A2,00,FE,0D,20,F7,C9
```

Fig. 7.3 Il programma BASIC POKE. Ricordate che, una volta eseguito, il codice macchina può essere salvato separatamente.

Sembra tutto assai lento a causa delle dimensioni del programma BASIC, ma ricordate come si può fare la registrazione. Se usate BSAVE per registrare il codice macchina *e* i codici del testo, può essere eseguito tutto da nastro senza bisogno del BASIC. Una volta che avete creato un programma in codice macchina, può essere salvato e caricato senza bisogno che il BASIC sia presente. Inoltre, se avete usato CLEAR per riservare uno spazio di memoria, potete caricare con BLOAD un programma in codice macchina *senza modificare un programma BASIC che è in memoria.* Questo significa che il vostro programma BASIC può prevedere delle istruzioni BLOAD in modo da caricare il codice macchina quando ce n'é bisogno. Se registrate su nastro un programma che ha una riga come questa:

```
BLOAD "CAS:CHARS"
```

allora, a questo punto del programma BASIC, il registratore si metterà in funzione (se il tasto PLAY è ancora abbassato), e sarà caricato un programma in codice macchina chiamato "CHARS". Non verrà però eseguito, a meno che il vostro programma BASIC non abbia una riga con l'istruzione USR(0).

TRASFERIMENTO DEI VALORI

L'istruzione A=USR(0) è stata usata fin qui solo per dare inizio all'esecuzione di un programma in codice macchina. Tuttavia, può fare di più. Tutto ciò che é contenuto fra le parentesi può essere trasferito ad un programma in codice macchina. Per "trasferito", intendo che se usate, ad esempio, A=USR(5), la cifra 5 sarà posta ad un indirizzo di memoria che può essere usato da un programma in codice macchina. Inoltre, un "codice del tipo" viene posto in un'altra locazione della memoria. Questo codice del tipo è 2 per un intero, 3 per una stringa, 4 per un numero a precisione singola e 8 per un numero a doppia precisione. Il codice del tipo, come avrete notato, è uguale al numero dei bytes che sono usati per immagazzinare quel termine.Nel caso delle stringhe, i tre bytes sono la lunghezza e l'indirizzo da due bytes del primo carattere della stringa. L'istruzione fa un'altra cosa. Quando usate A=USR(X%),per esempio, il numero X% viene posto in memoria e può essere utilizzato dal codice macchina. Se fate in modo che il vostro codice macchina immetta un altro numero nella stessa parte di memoria, quel numero verrà assegnato alla variabile A. Per esempio. se usate A=USR(5) seguita da PRINT A, e il programma in codice macchina legge il "5" dalla memoria e mette "25" allo stesso indirizzo, allora l'istruzione PRINT A darà il numero 25! In questo modo, quindi, potete far sì che delle routines in codice macchina interagiscano con i vostri programmi BASIC.
Un esempio chiarirà meglio le cose. Date un'occhiata allo schema di programma della Fig. 7.4. Questa configurazione propone di controllare il tipo della quantità che deve essere trasferita, preleva il suo valore, fa il complemento (forma negativa), e poi lo rimette all'indirizzo originale. Più in dettaglio, cercheremo un numero intero, rifiutando tutto ciò che non è un intero. Il valore è ottenuto da (HL+2), il complemento con l'uso di NEG, e il valore viene reimmesso in (HL+2) per essere rimandato al BASIC. Occorre dare qualche spiegazione rispetto all'uso di A ed HL. Quando si usa l'istruzione USR(N), questa coinvolge sia i registri che la memoria. Il numero di tipo del dato (2, 3, 4 o 8) viene immagazzinato nell'accumulatore ed anche all'indirizzo F663H. Il valore è memorizzato in un insieme di indirizzi che cominciano a F7F6H per i numeri a precisione doppia e singola. Il registro HL memorizza l'indirizzo F7F6H come valore d'inizio, sebbene i numeri interi usino solo F7F8H e F7F9H. Se avete trasmesso un valore di stringa, allora due bytes sono immagazzinati

INIZIO

CONTROLLA TIPO	A= 2 se corretto ⊕ ritorna se non è
PRELEVA VALORE	in (HL+2) o indirizzo
COMPLEMENTA	usa NEG
REIMMETTI	in (HL+2) o indirizzo
STOP	RET

Fig. 7.4. Uno schema che implica il trasferimento di una quantità numerica dal BASIC ad una routine in codice macchina.

a partire da F7F8H come prima, ed anche nella coppia di registri DE. Questi due bytes danno l'indirizzo dell'inizio della VLT di stringa. Possiamo ottenere i nostri valori usando i registri o gli indirizzi direttamente, e i valori di ritorno usando gli indirizzi.

La Fig. 7.5 illustra il linguaggio assembly per l'azione che è stata indicata nello schema della Fig. 7.4. Ancora una volta, si tratta di un'azione semplice che poteva essere eseguita altrettanto facilmente con il BASIC, ma un esempio di questo tipo è breve e facile da battere, mentre un esempio più complicato avrebbe richiesto più tempo per la battitura (e più errori!) e sarebbe stato pieno di termini non necessari per comprendere i principi basilari.

```
1 ORG 0F000H
2 CP 02H;          test per intero
3 JR NZ,EXIT;      esci se non è
4 INC HL
5 INC HL
6 LD A,(HL);       preleva
7 NEG;             complementa
```

```
 8 LD (HL),A;       rimetti in A
 9 LD A,2;          codice tipo dato
10 EXIT:RET
11 END
```

Fig. 7.5. La routine in linguaggio assembly per implementare lo schema della Fig. 7.4.

Il programma in linguaggio assembly inizia con CP 02H e JR NZ, EXIT. Si vuole qui controllare il tipo del dato che è stato trasferito, e uscire dal programma se non è corretto. Il controllo dei dati è importante, perché un programma in codice macchina non si arresta con un messaggio d'errore se trova un dato sbagliato. Dovete quindi immettere voi stessi questi controlli. Se il dato è corretto, le due istruzioni INC HL avranno l'effetto di fare in modo che la coppia di registri HL porti (o *punti*) il corretto indirizzo d'inizio del numero. Renderemo le cose più semplici limitando il numero ad un solo byte. Questo verrà fatto nella parte BASIC del programma. L'istruzione LD A,(HL) leggerà il numero contenuto nell'accumulatore e NEG farà il complemento del numero. Il risultato viene poi reimmesso all'indirizzo di memoria F7F8H, contenuto in HL, e l'accumulatore è caricato con 2 per indicare un intero. Al passo RET, i valori dovrebbero essere nuovamente trasferiti al BASIC.

```
 10 CLEAR 200, &HEFFF
 20 A!=&HF000
 30 FOR N%=0 TO 12: READ D$
 40 POKE A!+N%,VAL("&H"+D$):NEXT
 50 DEF USR=A!
 60 CLS:PRINTTAB(14)"NEGATIVI"
 70 PRINT:PRINT "Inserire un numero minore di 256"
 80 INPUT X%: IF X%>255 OR X%<0 THEN PRINT
    "ERRATO - RIPETERE":GOTO 70
 90 A%=USR(X%)
100 PRINT "La forma negativa è";A%; "( ";HEX$(A%);
    "in esadecimale)"
110 DATA FE,02,20,08,23,23,7E,ED,44,77,3E,02,C9
```

Fig. 7.6 La versione BASIC POKE di questo programma.

Nel programma BASIC della Fig. 7.6, il codice macchina assemblato viene immesso in memoria nel modo consueto con la funzione POKE, e il programma BASIC che inizia alla riga 60 usa il codice macchina.

Il numero immesso deve essere minore di 256 in modo da poter essere contenuto in un byte. Quando è stato immesso il numero come variabile X%, la riga 90 lo trasferisce al programma in codice macchina usando A%=USR(X%). Il programma in codice macchina preleverà questo valore, complementerà il numero, e lo trasferirà alla variabile A%. La riga 100 stampa il valore di A% nella forma decimale ed esadecimale. Avreste ottenuto lo stesso risultato se aveste usato PRINT(256—X%), ma almeno serve come dimostrazione.
Se trasferite un valore intero completo, che usa due bytes, allora il byte più basso viene immagazzinato all'indirizzo F7F8H e il byte più alto all'indirizzo F7F9H. La Fig. 7.7 mostra come vengono trasferiti i valori a singola e doppia precisione:

NUMERI A PRECISIONE SINGOLA
Il numero del tipo, 4, è all'indirizzo F663H e nell'accumulatore.
I bytes del valore, in forma BCD, agli indirizzi da F7F6H a F7F9H —l'indirizzo F7F6H è in HL.
F7F6H contiene la posizione della virgola decimale (codificata)
NUMERI A DOPPIA PRECISIONE
Il numero del tipo, 8, è all'indirizzo F663H e nell'accumulatore.
I bytes del valore, in forma BCD, agli indirizzi da F7F6H a F7FDH —l'indirizzo F7F6H è in HL.
F7F6H contiene la posizione della virgola decimale (codificata)

Fig. 7.7. Come vengono trasferiti i valori di numeri a precisione doppia e singola a un programma in codice macchina e viceversa.

Questi non usano direttamente i numeri binari, ma quelli chiamati BCD (si veda Appendice B). Potete eseguire, su un numero BCD, una qualsiasi delle operazioni aritmetiche svolte nell'accumulatore, *ma* dovete poi usare l'istruzione DAA (decimal adjust accumulator = adattamento decimale dell'accumulatore) per assicurarvi che la risposta sia in forma BCD.
La Fig. 7.8 dà un esempio di questo tipo di tecnica.

```
1                       ORG   0F000H
2                       LOAD  0F000H
3 F000 23               INC   HL
4 F001 0603             LD    B,03H
5 F003 7E      LOOP:    LD    A,(HL)
6 F004 C622             ADD   A,22H
7 F006 27               DAA
8 F007 77               LD    (HL),A
```

```
 9 F008 23                INC   HL
10 F009 10F8              DJNZ  LOOP
11 F00B 3E04              LD    A,04H
12 F00D 3263F6            LD    (0F663H) ,A
13 F010 21F6F7            LD    HL,0F7F6H
14 F013 C9       EXIT:    RET
15                        END
```

Fig. 7.8. Un esempio di trasferimento di un numero che viene manipolato dal codice macchina e poi di nuovo trasferito.

Si vuole trasferire il numero 12,3456 alla routine in codice macchina. Questo sarà memorizzato dall'indirizzo F7F6H in poi sotto forma dei bytes (esadecimali) 42 12 34 45. Il primo byte è "la potenza di dieci" più 40H: la potenza di dieci è quindi 2, che indica uno spostamento di due posti decimali a destra. Il numero viene memorizzato come 123456, e questo significa 0,123456. Lo scorrimento di due posti a destra dà 12,3456. Ora, se aggiungiamo 22H ad ognuno dei bytes del *numero* (escluso quindi il primo byte), avremo immagazzinato 34 56 78, che dovrebbe ritornare come 34,5678. Per far questo, dobbiamo incrementare HL, in modo da ignorare il primo byte, poi eseguire il ciclo per tre volte, aggiungendo 22H al numero caricato da (HL). Dopo l'addizione, il passo DAA controlla che il numero sia corretto in BCD (in questo esempio, DAA non è strettamente necessario). Il risultato viene poi ricaricato in memoria e il puntatore è incrementato dal caricamento successivo fino alla fine del conteggio. La Fig. 7.9 ne dà la versione BASIC. Il codice macchina non usa controlli (come CP 04H) all'inizio per assicurarsi che il numero sia a precisione singola. In questo esempio, il programma BASIC controlla che il numero sia a precisione singola, ma se il programma avesse usato numeri immessi da tastiera, sarebbero stati necessari i passi CP 04H e JR NZ,EXIT nella versione in linguaggio assembly.

```
 10 CLEAR 200,&HEFFF
 20 Q!=&HF000:DEF USR= Q!
 30 FOR N%=0 TO 19:READ D$
 40 POKE Q!+N%,VAL("&H"+D$):NEXT
 60 X!=12.3456
 70 A!=USR(X!)
 80 PRINTA
100 DATA 23,06,03,7E,C6,22,27,77,23
110 DATA 10,F8,3E,04,32,63,F6,21,F6
120 DATA F7,C9
```

Fig. 7.9. La versione BASIC della Fig. 7.8.

TRASFERIMENTO DI STRINGHE

Il trasferimento di un valore di stringa ad una routine in codice macchina implica un po' più di lavoro. Quello che è immagazzinato ai due indirizzi F7F8H e F7F9H è l'indirizzo contenuto nella VLT, non l'indirizzo della stringa. Dobbiamo prelevare l'indirizzo da F7F8H e F7F9H, poi vedere che cosa è memorizzato a *questo* nuovo indirizzo per ottenere la lunghezza della stringa, poi prelevare gli altri due bytes per l'indirizzo della stringa. Possiamo illustrare tutto questo con un programma che "codificherà" un messaggio di stringa. L'idea è di aggiungere per ogni byte 3 al suo codice ASCII — non sarà esattamente un codice segreto, ma sono i principi che vogliamo illustrare. La Fig. 7.10 mostra il progetto per quest'azione.

INIZIO		
CONTROLLA TIPO	usa CP 03 esci se non è	
PRENDI INDIRIZZO VLT	HL+2 > HL+3 =indirizzo	leggi da (HL) in DE reimmetti in HL
LEGGI STRINGA	1° byte = lunghezza gli altri 2 = indirizzo inizia ciclo	leggi lunghezza in B metti indirizzo in DE
AGGIUNGI 3 A OGNUNO	in A	da (DE)
REIMMETTI	nell'indirizzo di stringa	a (DE)
RITORNA	al BASIC	
FINE		

Fig. 7.10. Uno schema di programma per il trasferimento di un valore di stringa.

In questo schema, poiché voglio enfatizzare lo svolgimento dell'azione, non è tutto molto semplice da seguire, almeno all'inizio. Per quanto riguarda gli indirizzi, viene indicato come VLT l'indirizzo, contenuto nella VLT, della lunghezza e della situazione della stringa, e come STR l'indirizzo della stringa stessa.
La prima azione è di controllare che sia corretto il tipo di variabile, codice 3. Questo è abbastanza diretto, per cui passiamo a PRENDI INDIRIZZO VLT. Questo, come ricorderete, è contenuto agli indirizzi F7F8H e F7F9H, e possiamo ottenerlo assai facilmente incrementando HL due volte, perché HL contiene *sempre* l'indirizzo F7F6H quando c'é una chiamata con USR. Ora vogliamo leggere questi due bytes per formare un indirizzo, e sembra perciò ragionevole leggerli in una coppia di registri. Li metteremo in DE. DE ora contiene l'indirizzo della VLT. È qui che si presenta un problema, perché non possiamo leggere così facilmente da DE, se non nell'accumulatore. Usando l'istruzione EX DE,HL, però, possiamo mettere l'indirizzo della VLT in HL, e l'indirizzo F7F9H in DE. Ora possiamo usare l'indirizzo della VLT che è in HL. Il primo byte è il byte di lunghezza e lo vogliamo in B. Perché proprio in B? Perché B é usato per l'azione del contatore di DJNZ, e se carichiamo da (HL) in B, avremo il conteggio dei caratteri pronto per partire! Leggiamo poi gli altri due indirizzi della VLT e li mettiamo in DE.
Questa coppia di bytes costituisce l'indirizzo STR, e questa volta possiamo lasciarlo in DE. Possiamo ora formare un ciclo che usa DE per l'indirizzamento, legge da (DE) nell'accumulatore, aggiunge 3, lo rimette a posto, e ripete finché il registro B non sia a zero. Questo è tutto, gente! La Fig. 7.11 mostra il programma in linguaggio assembly.

```
 1                          ORG   0F000H
 2                          LOAD  0F000H
 3  F000 FE03               CP    03H       ; è una stringa?
 4  F002 2012               JR    NZ,EXIT   ; esci se non è
 5  F004 23                 INC   HL
 6  F005 23                 INC   HL        ; inizia
 7  F006 5E                 LD    E,(HL)    ; preleva indirizzo
 8  F007 23                 INC   HL        ; ora alto
 9  F008 56                 LD    D,(HL)    ; in HL
10  F009 EB                 EX    DE,HL     ; metti in HL
11  F00A 46                 LD    B,(HL)    ; lunghezza stringa
12  F00B 23                 INC   HL        ; ora basso
13  F00C 5E                 LD    E,(HL)    ; in E
14  F00D 23                 INC   HL        ; per byte alto
15  F00E 56                 LD    D,(HL)    ; in D
16  F00F 1A         LOOP:   LD    A,(DE)    ; preleva cod. car
17  F010 C603               ADD   A,03H     ; aggiungi 3
```

```
18  F012  12              LD    (DE),A   ; reimmetti
19  F013  13              INC   DE       ; il prossimo
20  F014  10F9            DJNZ  LOOP     ; fino in fondo
21  F016  C9      EXIT:   RET
22                        END
```

Fig. 7.11. Il linguaggio assembly per lo schema di programma della "codifica di un messaggio".

Notate che, nonostante ci siano molti passi nel linguaggio assembly, essi originano soltanto 23 bytes di codice macchina, meno di una normale riga in BASIC! La Fig. 7.12 mostra la versione BASIC del programma che immetterà il codice macchina in memoria e lo controllerà con una stringa. Anche qui, non viene fatto niente di più di quanto potreste fare solo col BASIC, ma vi mostra come posizionare in memoria e usare una stringa, che era appunto lo scopo dell'esercizio. È anche il più lungo programma in linguaggio assembly che abbiamo finora incontrato, e nel prossimo capitolo cominceremo a vedere l'uso di un assemblatore per creare programmi molto più lunghi. Nel frattempo, faremo una piccola digressione su alcuni argomenti che devono essere affrontati per fare altri progressi nel codice dello Z80.

```
10 CLEAR 200,&HEFFF
20 Q!=&HF000:DEFUSR1=Q!
30 FOR N%=0 TO 22:READ D$
40 POKE Q!+N%,VAL("&H"+D$):NEXT
50 CLS
60 T$="ECCO UNA PROVA"
70 B$=USR1(T$)
80 PRINTB$
100 DATA FE,03,20,12,23,23,5E,23,56
110 DATA EB,46,23,5E,23,56,1A,C6,03
120 DATA 12,13,10,F9,C9
```

Fig. 7.12. La versione BASIC della Fig. 7.11.

LO STACK

Lo STACK fa parte della memoria RAM del computer, e l'unica cosa che lo rende speciale é solo il fatto di essere usato in un modo particolare. All'inizio di questo libro, abbiamo esaminato la VLT che si trova proprio sopra un programma BASIC. Questo era un esempio di una parte di memoria riservata all'uso di un programma BASIC. Lo stack é un'altra parte della RAM, ma, a differenza della VLT, il suo indirizzo d'inizio è fisso, a meno che non venga modificato da una vostra istruzione. Questo

non significa che sarà *sempre* nella stessa posizione. Se osserviamo una "mappa" della cima della memoria (Fig. 7.13), possiamo vedere che la memoria sopra F380H è riservata come "area di lavoro" alla ROM BASIC — abbiamo già visto alcuni di questi indirizzi.
Da F380H in giù la memoria è usata per il blocco di controllo dei file, l'area di stringa, e poi lo stack. Ora l'ampiezza del blocco di controllo dei file dipende dal comando MAXFILES in BASIC. Quando accendete il computer, viene riservato lo spazio per il blocco di controllo di un file e, a meno che non usiate i dischi, questo dovrebbe bastarvi. L'area di stringa è posta a 200 bytes inizialmente, e potete modificarla con l'istruzione CLEAR. La posizione d'inizio dello stack quando viene acceso il compu-

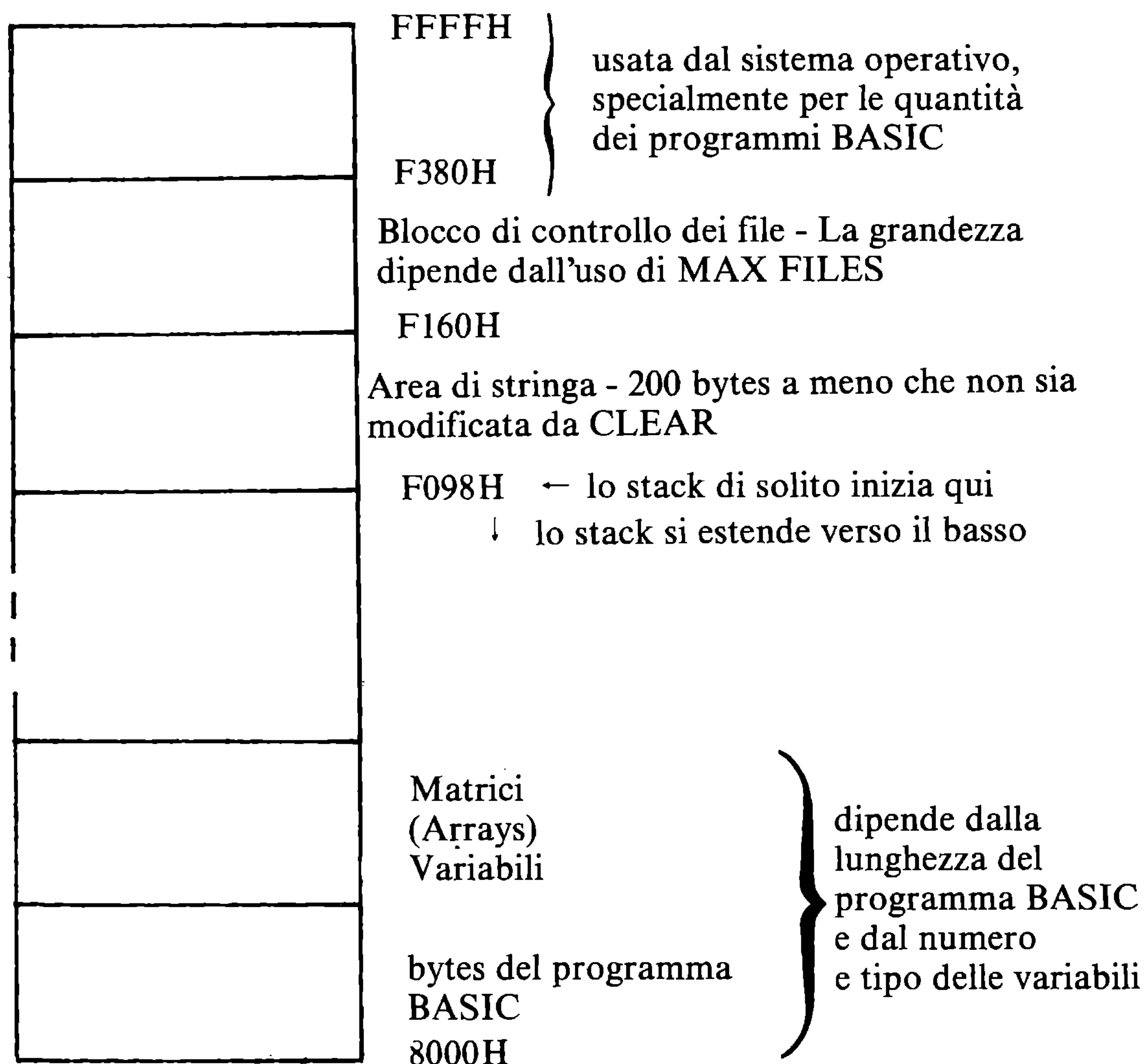
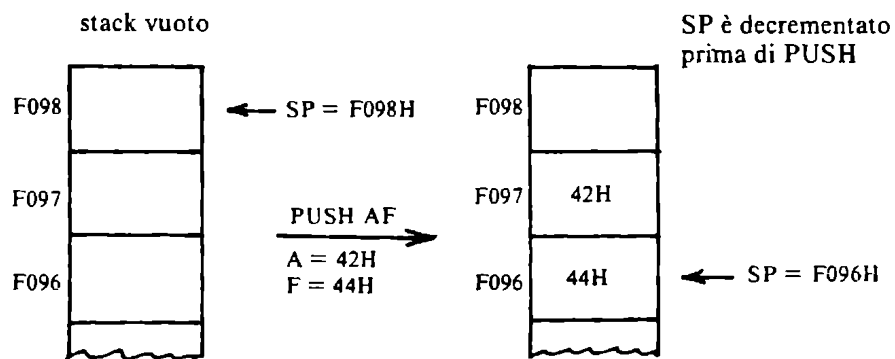


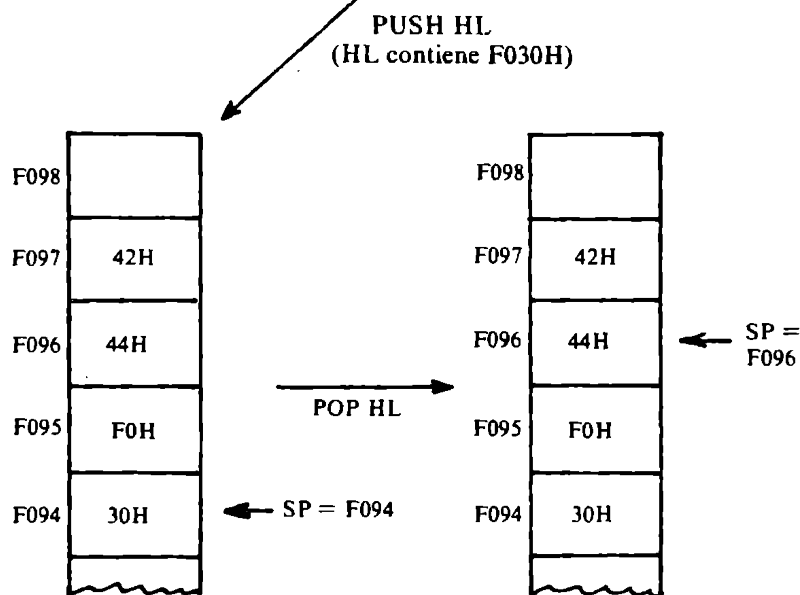
Fig. 7.13. Una "mappa" della parte alta della memoria, che mostra come é posizionato lo stack.

ter é F098H, ma sarà spostata ad un indirizzo più basso se usate MAX-FILES, o CLEAR per una più vasta area di stringa. Al contrario, se definite un'area di stringa più piccola, ad esempio con CLEAR50, lo stack si sposterà verso l'alto in memoria. La posizione dello stack, comunque, *deve rimanere fissa* per la durata di un programma. Se cercate di modificarla durante un programma, causerà senza alcun dubbio un fallimento del programma stesso. Quando lavorate in BASIC, non potete fare una cosa del genere, perché il programma si arresterà con un messaggio di errore, se avete delle istruzioni che potrebbero scrivere dei dati su quelli dello stack, ma in codice macchina non avete questa protezione.
I nostri programmi finora hanno usato gli indirizzi da F000H in avanti. Quando diciamo che l'indirizzo dello stack é a F098H, vogliamo dire che questo è l'inizio della memoria di stack *quando accendete il computer.* Quando usate CLEAR 200, &HEFFF, lo stack viene spostato a ED0DH, ben al sicuro. Lo stack "cresce verso il basso", cioé, quando immagazzinate i bytes sullo stack, l'indirizzo ad essere successivamente usato sarà un indirizzo *più basso.* Per esempio, se l'indirizzo dello stack è F098H e viene memorizzata una coppia di bytes, l'indirizzo cambia in F096H, due bytes più basso. Questo indirizzo del primo posto disponibile sullo stack è contenuto in un registro dello Z80 che è chiamato "puntatore di stack". Caricando questo indirizzo con altri valori, possiamo cambiare la posizione dello stack. L'assemblatore ZEN, per esempio, posiziona il puntatore di stack in modo che siano usati gli indirizzi subito sotto il programma principale dello ZEN (da A21EH in giù). Per i vostri programmi, a meno che non vi sia qualche pressante ragione per spostare il puntatore di stack, *lasciatelo in pace!* Il posizionamento del puntatore di stack stabilito per il BASIC può normalmente essere usato senza alcun problema per i vostri programmi in codice macchina. Se lo spostate, dovete immagazzinarne il valore nella RAM, e ripristinare questo valore prima che il vostro programma in codice macchina ritorni al BASIC.

COME SI USA LO STACK

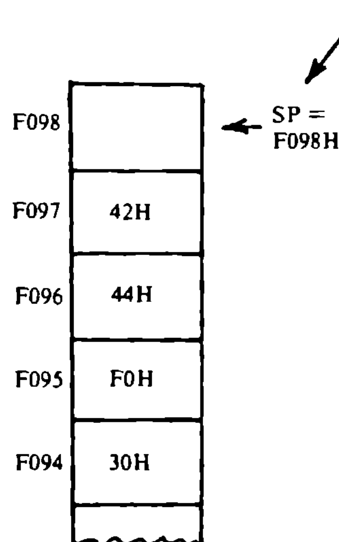
Potete usare lo stack nei vostri programmi senza dovervi preoccupare di dove sia posizionato in memoria, a meno che non ci sia pericolo che lo stack invada l'area del programma. Se usate un'istruzione CLEAR, tipo CLEAR 200,&HEFFF, e ponete i vostri programmi solo in indirizzi che si trovano al di sopra di quest'area riservata (da F000H in su, nel nostro esempio), non toccherete lo stack e potrete usarlo come desiderate. Lo Z80 usa lo stack in due modi, uno automatico e l'altro manuale. L'uso automatico si ha attraverso istruzioni come CALL 0C9H. Appena prima della sua esecuzione, il PC conterrà l'indirizzo dell'istruzione che dovrà essere eseguita successivamente. L'esecuzione dell'istruzione CALL fa sì che il puntatore dello stack (STACK Pointer = SP) sia decrementato ed il





HL contiene ora F030H.
I bytes sullo stack
non sono cancellati

POP AF



AF ora contiene 4244H

Lo stack ora è "vuoto", anche se i bytes
sono ancora immagazzinati.
Ogni bytes immesso sullo stack
andrà a sostituire qualcuno
dei bytes immagazzinati.

Fig. 7.14. Un diagramma che illustra come viene usato lo stack. L'indirizzo corrispondente alla "sommità dello stack" non è mai usato.

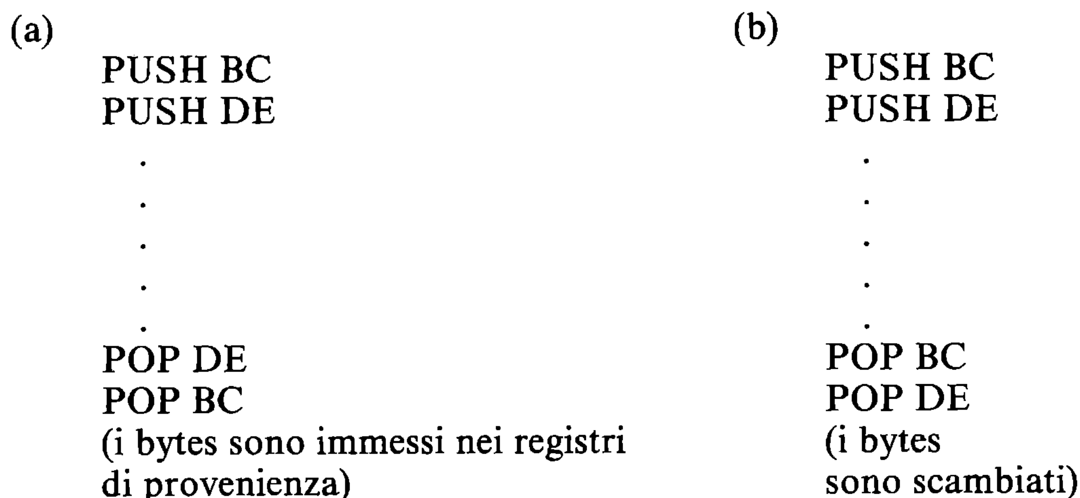
byte alto del PC venga memorizzato all'indirizzo contenuto in SP. SP viene poi di nuovo decrementato e verrà ora memorizzato il byte basso del PC. Infine, il puntatore dello stack viene lasciato con l'indirizzo a cui è stato memorizzato, nello stack, il byte basso del PC. Il nuovo indirizzo di CALL viene quindi caricato nel PC e la subroutine viene eseguita. Dopo l'istruzione RET, viene caricato il byte dall'indirizzo contenuto in SP nel byte del PC, SP viene incrementato e il byte che si trova a questo nuovo indirizzo viene caricato nel byte alto del PC. Tutto questo ripristina l'indirizzo corretto del PC, quello cioé che vi si trovava prima di eseguire CALL, ed il puntatore dello stack viene poi incrementato. Questa posizione corrispondente alla "sommità dello stack" non è mai usata; viene tenuta libera per indicare la posizione della sommità della memoria di stack. La Fig. 7.14 riassume questo processo.
Oltre a questo uso automatico dello stack, attraverso l'istruzione CALL ed altri tipi di istruzione (come RES), possiamo avere delle istruzioni direttamente rivolte allo stack, quali PUSH e POP. PUSH significa mettere il contenuto di un registro doppio sullo stack, usando due bytes della memoria di stack. In un'istruzione PUSH, il puntatore dello stack viene decrementato, il byte alto posto nell'indirizzo dello stack, e quindi SP è di nuovo decrementato per memorizzare il byte basso. Questo è lo stesso ordine di memorizzazione usato nell'istruzione di chiamata CALL. L'azione POP segue l'azione RET, il byte basso viene restituito, il puntatore dello stack incrementato, viene restituito il byte alto ed SP è di nuovo incrementato. Gli operatori PUSH e POP possono essere usati con gli operandi AF,BC,DE,HL,IX e IY, per cui si possono salvare in questo modo tutti quei registri che sono importanti per un programma. Ora è il momento di vedere *perché* ci occorrono queste istruzioni e come dobbiamo usarle.
Una necessità molto comune è il salvataggio di BC a causa di un conteggio DJNZ. Supponete di avere un ciclo che sia stato programmato caricando il registro B con 0FFH, ed usando poi DJNZ. Ciò, normalmente, ci impedirebbe di usare in un qualsiasi altro modo la coppia di registri BC subito prima o subito dopo il ciclo. Per esempio, non potreste fare quanto è mostrato nella Fig. 7.15(a), dove un indirizzo viene caricato in BC subito prima di un ciclo, e poi usato in seguito. Infatti, con l'istruzione DJNZ verrà cambiato il numero contenuto nel registro B, per cui BC certamente non conterrà più l'indirizzo F100H dopo il ciclo; conterrà 0000H perché B è stato portato a zero. Il programma—campione della Fig. 7. 15(b) aggira questo ostacolo ponendo BC sullo stack, con l'istruzione PUSH, subito prima del ciclo, e poi prelevandolo dallo stack subito dopo.
Un altro uso delle istruzioni PUSH e POP è per conservare i flag. Supponete di aver appena eseguito un'azione di confronto CP che posiziona a 1 i

```
(a)         LD BC, 0F100H
            LD B, 0FFH           ← sostituisce il byte F1H
                                   in 'B'
    LOOP:       .
                .
                . ( istruzioni del ciclo )
                .
            DJNZ LOOP            ← fine del ciclo
            LD A, (BC)           ← istruzioni successive

(b)         LD BC, 0F100H
            PUSH BC              ← salvataggio sullo stack
            LD B, 0FFH
    LOOP:       .
                . ( istruzioni del ciclo )
                .
                .
                .
                .
            DJNZ, LOOP           ← fine del ciclo
            POP BC               ← riprende il contenuto
                                 di BC dallo stack
            LD A,(BC)            ← istruzione successiva
```

Fig. 7.15(a) Programma non corretto, perché è ricaricato il byte contenuto in B. (b) Come può essere usato lo stack per salvare i valori corretti e ripristinarli in seguito.

flag, ma di voler eseguire ancora altre due azioni prima di effettuare il controllo per il salto. Se queste azioni modificano i flag, il salto non sarà corretto, ma se usate l'istruzione PUSH AF subito dopo l'istruzione CP, e POP AF prima del passo JR, potete ripristinare i valori presenti in A e in F dopo che era stata eseguita l'azione CP, a prescindere da quanti altri passi siano stati portati a termine fra PUSH e POP.
Come avrete immaginato, PUSH e POP devono essere usate con una certa attenzione, particolarmente quando l'istruzione PUSH viene applicata a più di un insieme di registri. Un'istruzione come POP AF, per esempio, leggerà due bytes dallo stack e li metterà nei registri AF, *sia che appartengano o no a tali registri.* Per esempio, supponete di avere una parte di programma come quella della Fig. 7.16.



(a)

PUSH BC
PUSH DE
.
.
.
.
.
POP DE
POP BC
(i bytes sono immessi nei registri
di provenienza)

(b)

PUSH BC
PUSH DE
.
.
.
.
.
POP BC
POP DE
(i bytes
sono scambiati)

Fig. 7.16.(a) Il normale uso last-in-first-out (ultimo ad entrare, primo ad uscire) dello stack. (b) L'uso di un ordine diverso per spostare due bytes in una diversa coppia di registri.

Questo programma ha immesso BC sullo stack e poi DE. Quando arriva il momento di applicare l'istruzione POP, i bytes che vengono restituiti per primi sono quelli di DE. Lo stack si attiene strettamente alla regola "ultimo ad entrare, primo ad uscire" (last-in, first-out) e DE è stato l'ultimo ad entrare. Usando POP DE e poi POP BC, i bytes saranno reimmessi nel registro da cui provenivano. Se invece usate la sequenza della Fig. 7.16(b), i bytes saranno scambiati fra i registri. La coppia di registri BC conterra ciò che in origine era in DE, e DE conterrà ciò che era in BC. Questo *può* essere un modo di scambiare il contenuto dei registri.
Un uso molto comune dello stack è collegato alle istruzioni CALL. Quando fate una chiamata alla ROM o alle vostre subroutines della RAM, queste subroutines possono alterare i registri che state usando. Per esempio, potreste avere un indirizzo importante contenuto in HL quando richiamate una subroutine. Se la subroutine carica ed usa HL, sarete nei guai, a meno che non usiate PUSH HL prima dell'istruzione CALL e POP HL subito dopo. Dal momento che la coppia di registri HL è usata così spesso in molte routines, questa evenienza è assai frequente. Un altro uso abbastanza comune è PUSH AF e POP AF, poiché molte subroutine modificano sia il registro A che il registro F.

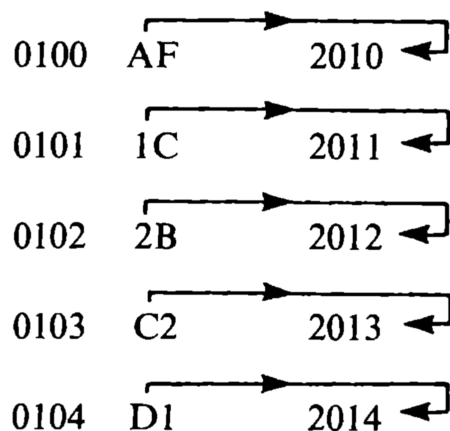
LE ISTRUZIONI DEI BLOCCHI

Dal momento che stiamo passando in rassegna una serie di argomenti che non avevamo trattato prima, vale la pena di menzionare le istruzioni dei blocchi dello Z80, poiché le useremo nel Cap. 9. Abbiamo già visto un tipo di istruzione di blocco, l'istruzione DJNZ, ma ce ne sono molte altre.

L'istruzione DJNZ è semplicemente un comodo contatore di un byte, che ci risparmia l'uso delle istruzioni DEC e CMP. Dovete programmare per conto vostro che cosa immettere in questi cicli. Invece, gli altri comandi di blocchi contengono istruzioni di caricamento e trasferimento dei registri che vengono eseguite automaticamente. Uno dei gruppi più importanti di questi comandi riguarda i registri HL e DF. La Fig. 7.17 riporta un sommario dei comandi di spostamento di blocchi dello Z80.

Mnemonica	Azione
LDI	Pone il contenuto di HL nell'indirizzo di DE, incrementa HL e DE, decrementa BC
LDIR	Come LDI, ma ripete finché BC non raggiunge 0
LDD	Come per LDI, ma i registri HL e DE sono decrementati
LDDR	Come LDD, ma ripete finché BC raggiunge 0

Un altro esempio.



HL -- 0100
DE -- 2010
BC -- 0005

LDIR esegue il trasferimento dei bytes dagli indirizzi da 0100 in poi a 2010 in poi, come indicato dalle frecce.
Nota: tutti i numeri sono in forma esadecimale.

Fig. 7.17. I comandi di trasferimento di blocchi dello Z80.

Immaginate che l'indirizzo contenuto in HL sia 23500 e l'indirizzo contenuto in DE sia 23600 e che B contenga 5.
L'azione comincia copiando il byte dall'indirizzo 23500 all'indirizzo 23600, poi gli indirizzi sono incrementati da 23601, ed il numero in BC è decrementato a 4. Il trasferimento è ripetuto, per cui il byte memorizzato a 23501 è copiato a 23601 e BC è decrementato a 3. L'azione termina quando il contenuto di BC è stato ridotto a 0 e sono stati copiati 5 bytes.
Le parole trasferimento di blocchi sintetizzano la funzione di queste istruzioni, cioé la loro capacità di spostare un blocco di dati da una parte all'altra della memoria, semplicemente ponendo nei registri le locazioni di memoria ed usando una sola istruzione in linguaggio assembly. Le istruzioni per il trasferimento di blocchi esistono in due forme principali, a ripetizione automatica o programmata, e con incremento o decremento di ognuna di queste due forme. Esamineremo ciascuna di esse, una per volta. LDI significa Carica, Decrementa, Incrementa. Quando viene usata questa istruzione, le coppie di registri HL e DE conterranno un indirizzo ciascuna, e BC è usato come contatore. Con l'istruzione LDI, il byte all'indirizzo (HL) è caricato all'indirizzo (DE) ed il contatore, BC, è decrementato. I registri HL e DE vengono poi incrementati: sta al programma stabilire quando il conteggio deve terminare, o dove deve ritornare il ciclo. LDD è un'altra versione di questa istruzione, con cui viene eseguito il trasferimento da (HL) a (DE), ma poi le due coppie di registri ed il contatore BC saranno decrementati. Aggiungendo "R" (Repeat = ripeti) ad ognuna di queste istruzioni, si avrà l'automatica ripetizione dell'azione: ciò significa, però, che non potete scegliere l'indirizzo cui dovrebbe tornare il ciclo. LDIR, per esempio, eseguirà il trasferimento di (HL) a (DE), incrementerà tutte e due le coppie di registri e decrementerà BC. Quest'azione, tuttavia, continuerà finché BC non raggiunge zero. Nello stesso modo, LDDR trasferirà, e poi decrementerà HL e DE, ed al contempo anche BC, finché BC non raggiungerà zero.
Le istruzioni per il trasferimento di blocchi sono particolarmente utili per spostare un intero gruppo di bytes da una locazione all'altra di memoria. Per esempio, potete spostare un intero programma BASIC in modo da caricare, con CLOAD, un altro programma BASIC, per eseguire quest'ultimo programma e poi riprendere il primo! Potete anche memorizzare la VLT di un programma BASIC per caricare, sempre con CLOAD, un altro programma che usi i valori delle variabili del primo programma. Tutte e due queste azioni possono essere eseguite riservando la memoria alla sommità della gamma di indirizzi (per esempio, usando CLEAR 200,&HA000) ed usando un breve programma in codice macchina per trasferire i bytes in quest'area protetta. Un altro programma in codice macchina potrà poi ritrasferire i bytes. Vedremo un esempio di quest'azione nel Cap. 9.
Nel frattempo, occorre considerare un altro tipo di azione per i blocchi: è

l'azione di "confronto di blocchi", tipica dell'istruzione CPI. Per portare a termine questa istruzione, dovete avere un indirizzo caricato in HL, un numero per il conteggio in BC, ed un byte nell'accumulatore. Quando viene eseguita l'istruzione CPI, il byte in (HL) è confrontato con il byte nell'accumulatore. Se i due sono uguali, viene posto a 1 il flag di zero, altrimenti il flag rimane a zero. L'indirizzo in HL viene poi incrementato e BC viene decrementato. Sta a voi usare il passo JR Z o JR NZ per creare un ciclo. Se usate l'istruzione CPIR, l'azione di confronto, incremento di HL e decremento di BC sarà ripetuta finché BC non raggiunge zero, *oppure* finché il confronto non risulta positivo. Questo è un ottimo sistema per la ricerca nella memoria di qualche specifico byte. Altre due istruzioni appartenenti a questo gruppo sono CPD e CPDR. Nell'azione CPD, i bytes sono confrontati come prima, ma la coppia di registri HL viene decrementata, ed anche il contatore BC é decrementato. L'azione CPDR è identica a quella di CPD, soltanto l'azione continuerà finché BC non raggiunge zero o il confronto dà esito positivo. Dovete però essere in grado di sapere quale di queste due eventualità si sia verificata, e in ambedue le istruzioni vengono usati sia il flag Z che il flag P/O (il flag numero 2). Il flag Z indica se il confronto è stato positivo o no, ed il flag P/O segnala se il conteggio è giunto al termine. Il flag P/O è infatti posto a 1 fintanto che il conteggio viene eseguito, ma viene poi posto a 0 quando il conteggio è completo . Se trovate che Z=1, P/O=1, allora il confronto è stato positivo. Se Z=0, P/O=0, il confronto è stato negativo. Dovete però fare attenzione al fatto che un'istruzione JR può controllare il flag Z, ma solo un'istruzione JP può controllare il flag P/O.

CAPITOLO 8

Correzione, messa a punto, controllo e ZEN

LE DELIZIE DELLA CORREZIONE

Ora che avete provato alcune delle delizie della programmazione in codice macchina, mi sembra onesto menzionarne anche qualche retroscena. Uno di questi è la correzione e la messa a punto di un programma (debugging)*

È facile a dirsi, lo so, ma la prima parte della messa a punto sta nella prevenzione. Controllate lo schema del vostro programma o il suo diagramma di flusso per essere sicuri che descriva realmente ciò che volete far eseguire al programma. Una volta soddisfatti dello schema, esaminate il linguaggio assembly per assicurarvi che esegua le istruzioni dello schema stesso. Quando sarete soddisfatti anche di questo punto, controllate che i bytes che volete immettere in memoria con l'istruzione POKE siano i bytes corrispondenti alle istruzioni del linguaggio assembly. Dovete controllare molto attentamente di aver inserito il codice corretto per il metodo d'indirizzamento usato. Se volete usare l'indirizzamento diretto, ma avete invece usato il byte d'istruzione per l'indirizzamento indicizzato, non potete aspettarvi che il programma funzioni. Ricordatevi che quando un programma in codice macchina non funziona, il risultato più probabile sarà quello di far bloccare la macchina. Non avrete nessuno dei cortesi messaggi del BASIC! Se controllate così ogni fase dello sviluppo di un programma, eliminerete molti errori prima che essi possano provocare dei guai. Non pensate di essere un assoluto incapace se a questo punto il vostro programma non funziona ancora — a meno che un programma in codice macchina non sia molto semplice, è assai probabile che ci sia un errore da qualche parte. Accade a tutti — ed è solo attraverso l'esperienza che potrete riuscire ad ottenere un programma con pochi errori, e facili da individuare.

Se usate un assemblatore, una fonte di errori scomparirà. La fragilità umana implica che il procedimento di conversione delle istruzioni in linguaggio assembly in codice macchina sia proclive all'errore. Infatti, questo procedimento consiste nell'effettuare una ricerca su delle tabelle ed ogni cosa che comporta la trascrizione da un pezzo di carta ad un altro è

*Nota: Bug, letteralmente "cimice", indica, nel gergo del computer, un errore del programma.

altamente suscettibile di introdurre errori. Descriverò brevemente l'azione dell'assemblatore ZEN più avanti, in questo capitolo. Nel momento in cui scrivo, lo ZEN è l'unico programma assemblatore disponibile per i computer MSX. Ho usato lo ZEN per alcuni anni su altri computer, e posso giurare (mano sul cuore) che è proprio il mio ideale di programma assemblatore. Se il codice macchina ha conquistato la vostra immaginazione, e volete veramente dedicarvi ad un lavoro più impegnativo di quello che può trovar spazio in questo libro, allora un buon programma assemblatore/editor/monitor non è un lusso, ma diventa essenziale. Se, però, intendete restare solo un dilettante, prendendo le parti più curiose in codice macchina di tanto in tanto, allora i metodi per immettere i bytes in memoria con POKE, che abbiamo finora usato, saranno perfettamente adeguati. L'uso di questi metodi, tuttavia, vuol dire che ci saranno errori nascosti in ogni angolo del codice macchina. La causa principale di questi errori è la noia. Convertire un programma in linguaggio assembly in bytes esadecimali e poi scriverli sotto forma di righe di DATA per un programma BASIC che usa la funzione POKE è un lavoro noioso, e tutti i lavori noiosi producono errori. I metodi d'indirizzamento errati sono un risultato comune della noia, e un altro è lo scrivere il codice sbagliato. Una notevolissima fonte di guai è data dagli spostamenti nell'istruzione JR. Potete sbagliare facilmente quando dovete sottrarre gli indirizzi e convertire un numero (specialmente un numero negativo) in forma esadecimale. Un altro problema sorge quando modificate un programma ed aggiungete delle altre istruzioni in codice macchina fra un'istruzione di salto e la sua destinazione, e poi dimenticate di modificare l'ampiezza del byte di spostamento! Questo è un problema che non si presenta usando un assemblatore. Un salto non corretto quasi sempre bloccherà il computer, per cui dovrete spegnerlo e poi riaccenderlo. In tal caso, sarete felici di esservi ricordati di registrare il vostro programma prima di eseguirlo! Un'altra forma di salto non corretto è scrivere l'opposto di quello che volevate, per esempio, usando JR Z invece di JR NZ o viceversa. Ma se considerate attentamente l'azione del salto per le diverse grandezze di bytes, dovreste eliminare questo tipo d'errore.

Come ho già detto, molti problemi possono essere eliminati da un controllo meticoloso, e vale la pena essere più che attenti riguardo agli spostamenti nelle istruzioni di salto o al contenuto iniziale dei registri. Un errore molto comune è usare i registri come se contenessero zero, all'inizio del programma. Non ne potete mai essere veramente certi. È più sicuro, infatti, presumere che ogni registro contenga un valore che farà impazzire il computer se viene usato. Però, dopo aver detto tutto questo ed averci messo tutta la buona volontà di questo mondo, che cosa si può fare se il programma non funziona ancora?

Non esiste una risposta semplice ed unica. Può darsi che il vostro schema di programma non faccia quello che voi volevate, e se non avete messo giù

uno schema, allora avete quello che vi meritate. Può darsi che cerchiate di usare una routine della ROM MSX e che essa non operi come vi aspettavate.
Finché non viene pubblicata una lista completa delle routines della ROM e delle condizioni (come ciò che deve essere in ciascun registro) che devono sussistere quando viene richiamata una routine, non resta altro che provare, e fare errori. Ciò che posso fare qui è darvi un'indicazione generale per eliminare gli errori da un programma che sembra ben costruito, ma non esegue l'azione che dovrebbe, come era stato invece previsto nello schema.
La prima regola d'oro è di non cercar mai di fare qualcosa di nuovo nel bel mezzo di un programma assai lungo. Teoricamente, il vostro programma in codice macchina dovrebbe consistere di varie subroutines su nastro, ognuna delle quali sia stata completamente controllata prima di assemblarla in un programma lungo. In pratica, ciò non è così facile, soprattutto quando le subroutines esistono solo come righe di DATA per i programmi BASIC che usano POKE. Come al solito, gli utilizzatori di un assemblatore sono favoriti, perché possono memorizzare le istruzioni in linguaggio assembly come i programmi BASIC, e richiamarle in memoria e modificarle come preferiscono. In alternativa al tenere su nastro una libreria di routines, la miglior cosa è di fare delle ampie note di commento alle subroutines. Oltre alle vostre subroutines, potete includere anche delle routines che avete trovato su delle riviste. Se poi usate una nuova subroutine in programma, è ragionevole provarla prima da sola, in modo che possiate essere sicuri di che cosa deve essere in un registro prima di richiamare la routine, e di che cosa vi sarà in seguito.
Se procedete in questo modo, dovreste riuscire ad eliminare parecchi errori, ma se vi trovate ancora davanti ad un programma che non funziona, e che non volete mettere da parte, allora dovrete usare un punto d'arresto (breakpoint).
Un punto d'arresto, per quanto riguarda il sistema operativo MSX, è il byte C9H (201 decimale). Questo è il byte RET, e il suo effetto è di ritornare al BASIC. Una volta che siete in BASIC, potete esaminare il contenuto della memoria con l'istruzione PEEK. La regola da seguire, per inserire dei punti di arresto, è di scegliere un punto del programma che preveda l'immissione in memoria di qualcosa. Se inserite un byte C9H subito dopo questo punto, il programma, quando verrà eseguito, ritornerà al BASIC appena incontrerà l'istruzione C9H. Usando la funzione PEEK, potrete quindi controllare che quanto è stato caricato in memoria sia proprio quello che volevate. In caso contrario, dovreste sapere dove cercare l'errore. Se invece va tutto bene, sostituite allora C9H con il byte che originariamente doveva essere al suo posto e ponete C9H subito dopo un'altra istruzione di memorizzazione. Dovete solo fare attenzione al fatto che C9H deve sostituire un byte di *istruzione,* e non di dati.

L'errore più noioso da individuare, con questo o con un altro metodo, è un ciclo non corretto. Infatti, esso provocherà sempre il blocco del computer e quindi dovrete spegnere e ricaricare il programma. Se usate un sistema a dischi, ciò può ancora essere tollerabile, ma se usate le cassette, potrà tradursi in una notevole perdita di tempo. Una delle cause principali di errore nel ciclo è che esso ritorni alla posizione sbagliata. Per esempio, se una parte del programma fosse la seguente:

```
LD B,FFH
LOOP:DJNZ
LD A,F050H
```

potremmo avere dei problemi. Supponete di averlo assemblato manualmente, e di aver fatto tornare il ciclo all'istruzione LD B anziché all'istruzione DJNZ. In questo caso, il registro B avrebbe sempre il valore FFH e non sarebbe mai decrementato a zero, per cui il ciclo non potrebbe mai terminare. Un errore come questo può essere facilmente individuato nel linguaggio assembly, perché è facile controllare la posizione della "label". Ma quando avete davanti solo i bytes del codice macchina, sarà molto più difficile da individuare. Come sempre, far attenzione ai cicli è l'unica risposta ed il metodo mostrato in questo libro per calcolare e controllare gli spostamenti sarà una buona precauzione.

MONITOR

Ho accennato brevemente ai monitor all'inizio del capitolo. Non hanno niente a che fare con i monitor televisivi, che sono una specie di schermo TV di qualità superiore per i segnali. Un monitor, nella terminologia "software", è un programma che esegue un controllo (monitor) di ogni azione di un programma in codice macchina. Un monitor è (o dovrebbe essere) un programma in codice macchina che può essere immesso in memoria ad una serie di indirizzi che si suppone non dobbiate usare per nient'altro. Una volta in memoria, un monitor vi permette di visualizzare il contenuto di ogni parte della memoria (in forma esadecimale), di modificare il contenuto di una qualsiasi parte della RAM, e di ispezionare o di modificare il contenuto dei registri dello Z80. Queste sono le più elementari azioni del monitor, ed è assai utile poter eseguire una parte del programma, inserire dei punti di arresto ed ispezionare i registri di un programma in azione. Il monitor ideale è quello che può eseguire, uno per volta, i passi di un programma in codice macchina, visualizzando, ad ogni passo, il contenuto della memoria e dei registri. Un monitor di questo tipo era disponibile per il vecchio TRS—80 Mk.1, e sarebbe il benvenuto, per i programmatori che utilizzano essenzialmente il codice macchina, se fosse disponibile anche per altri computer, in particolare per l'MSX. Per il

momento, tuttavia, l'assemblatore ZEN svolge anche la funzione di monitor in modo assai soddisfacente: la esamineremo in seguito.
L'assemblatore ZEN è un vasto programma in codice macchina, posto in memoria dall'indirizzo A000H a BB5CH. Quando comprate lo ZEN, il programma è su cassetta, con le istruzioni per caricarlo ed il manuale. Il manuale è assai conciso, ed è semplice solo se avete già usato un assemblatore. Ma non leggereste questo libro se l'aveste usato, per cui sarà meglio esaminarne in dettaglio i procedimenti. Il manuale vi dà anche un elenco dei codici d'istruzione dello Z80 e perfino un listato dello ZEN. A differenza di molti programmi, protetti fino al punto di renderli inutilizzabili per l'appassionato, potete avere un listato completo dello ZEN che vi permetterà di vedere, una volta acquistata una padronanza del codice macchina, come sia ben scritto questo programma. Potrete vedere anche le routines previste dallo ZEN per eventualmente adattarle ai vostri scopi. Un listato come questo è molto raro ai nostri giorni, e rende lo ZEN un prezioso strumento per l'appassionato. Cercate, però, di non abusare della fiducia dimostrata dalla Avalon Software facendo delle copie illegali dello ZEN per venderlo. A parte tutto, il programma è così noto che una copia pirata sarebbe subito riconosciuta....!
Ma una cosa è scrivere le istruzioni — un'altra, ben diversa, lavorare alla tastiera e farle funzionare — così vediamo di fare, per prima cosa, una vostra copia personale dello ZEN. È importante farla, perché la copia che avete comprato è stata registrata, per garantire una maggior affidabilità, alla velocità più bassa della cassetta. Dal momento che, durante le vostre prime esperienze, dovrete spesso spegnere il computer e ricaricare il programma, sarà utile fare una copia dello ZEN che si carichi più velocemente, poiché lo ZEN, in fin dei conti, è un lungo programma ed impiega abbastanza tempo a caricarsi alla velocità più bassa. Non c'é perciò niente di male a fare una copia dello ZEN per il vostro uso personale.
Cominciate ad accendere il vostro computer MSX. Quando compare la scritta "Ok", battete:

```
CLEAR 200,&H9FFF   (premete RETURN)
```

e poi:

```
BLOAD "ZEN"        (premete RETURN)
```

Ora premete il tasto PLAY del registratore e aspettate finché lo ZEN non è stato caricato. Se il volume del registratore non è adeguato, comparirà il messaggio consueto.
Quando lo ZEN è stato caricato, senza che siano comparsi messaggi

d'errore, si trova nella memoria del computer e non può essere danneggiato dal BASIC. Infatti, l'istruzione CLEAR 200,&H9FFF impedisce che il BASIC usi indirizzi sopra 9FFFH e lo ZEN comincia a A000H! Potete quindi stabilire la più alta velocità di scrittura della cassetta usando SCREEN,,,2. Non fate errori con le virgole — devono esserci! Ora potete registrare su un'altra cassetta. Riavvolgete il nastro all'inizio e premete i tasti di RECORD e PLAY del registratore. Ora digitate:

BSAVE"CAS:ZEN",&HA000,&HBB5C

e premete RETURN: la cassetta sarà registrata con lo ZEN alla più alta velocità di scrittura. Potrete usare questa cassetta, preferibilmente con altre due registrazioni dello ZEN, per i vostri programmi. Conservate l'originale in un posto sicuro e fresco come matrice per altre copie.

L'USO DELLO ZEN

Ora che avete una copia dello ZEN più veloce a caricarsi, possiamo vedere lo ZEN in azione. Non dovete far niente di particolare per caricare lo ZEN, solo accertarvi di aver riservato un sufficiente spazio di memoria. Vedremo ora come si usa l'assemblatore ZEN per generare il codice per un semplice programma. Una volta visto questo, potrete ripetere l'applicazione dello ZEN ad altri programmi finché non avrete preso abbastanza confidenza con questo eccellente assemblatore. È un po' come comprare il vostro primo Black & Decker, dopo averlo usato vi chiedete come avete potuto farne a meno prima! Caricate lo ZEN, con l'istruzione CLEAR 200,&H9FFF e poi BLOAD"ZEN",R. La "R" farà eseguire immediatamente il programma, senza bisogno delle istruzioni DEF USR e USR. Quando il nastro avrà terminato il caricamento, se tutto è andato bene vedrete sullo schemo l'indicazione:

ZEN>

con il cursore posizionato subito dopo la freccia >. Questa freccia > è sempre usata con il significato di "istruzione, prego"; è il segnale che lo ZEN è in attesa di un vostro comando. Se, a questo punto, premete semplicemente il tasto RETURN, lo schermo verrà ripulito e potrete usare tranquillamente lo ZEN. Supponiamo, quindi, di prendere il programma della Fig. 6.9, il ciclo di ritardo. È assai semplice come inizio e sappiamo già che funziona, quindi stiamo seguendo la principale regola della programmazione in codice macchina — non sperimentare troppe cose nuove tutte insieme!
Per immettere queste istruzioni nell'assemblatore ZEN, digitiamo prima E (per ENTER) e premiamo il tasto RETURN. Tutte le istruzioni dello

ZEN richiedono l'uso del tasto RETURN, per cui potete apportare delle correzioni usando il tasto BS se avete dei ripensamenti. Dopo aver premuto il tasto RETURN, vedrete che Z> si sposta verso l'alto ed appare un "1", in attesa che venga immessa la prima riga del programma in linguaggio assembly, che dovrebbe essere ORG 0F000H. Questa istruzione indica che il codice assemblato deve essere posto in memoria. Tutta la memoria sopra A000H è stata protetta, e gli indirizzi da A000H a BB5CH sono usati dallo ZEN, per cui siete liberi di usare gli indirizzi da BB5CH in poi. Questo è più che sufficiente per le vostre necessità, ed in questo libro continueremo ad usare F000H come indirizzo di inizio i nostri programmi. *Dovete ricordare* che in un programma in linguaggio assembly lo ZEN richiede che un numero esadecimale cominci con una cifra normale (da 0 a 9) *e* che sia seguito da H. Non accetterà &H davanti al numero, come invece richiede il computer MSX nelle istruzioni PEEK e POKE, né accetterà numeri come F000H. Quando avrete battuto l'istruzione ORG 0F000H, premete RETURN per immetterla in memoria. Essa viene posta in una parte di memoria (chiamata "buffer", o memoria di transito, che inizia all'indirizzo C000H) controllata dallo ZEN. In questa fase, non viene caricato niente nella parte di memoria che comincia all'indirizzo F000H, perché lo ZEN *non* assembla (non crea codice macchina) finché non gli dite di farlo.
Una volta che avrete immesso questa riga, lo schermo mostrerà un "2" sotto l'"1". Lo ZEN aspetta che immettiate la riga successiva, LOAD 0F000H e premiate (RETURN). Anche qui, lo zero davanti al numero è essenziale, a meno che non usiate numeri decimali. È praticamente come scrivere un programma BASIC con la numerazione automatica (AUTO) delle righe. Questa riga dice all'assemblatore di immettere in memoria il codice macchina, a partire da F000H. Perché le istruzioni ORG e LOAD? Potreste, infatti, voler assemblare ad un indirizzo ed immettere il codice ad un altro. Questo non vi interessa, al momento, ma se voleste scrivere un programma che viene eseguito agli indirizzi della ROM, non potreste certo assemblarlo nella ROM! L'uso delle istruzioni separate ORG e LOAD permette allo ZEN di assemblare il codice che potrebbe essere poi usato ad una diversa locazione di memoria, perfino su un diverso tipo di macchina!
Alla riga 3 immettete la prima vera istruzione in linguaggio assembly, LD BC,0FFFFH. Fate attenzione agli spazi — lo ZEN richiede sempre uno spazio fra LD e BC, in modo da poter distinguere fra operatore e operando. Anche qui, *dovete* mettere uno zero davanti al numero. Continuate pure a digitare, ma fate ben attenzione agli spazi e alle virgole. La "label" deve essere seguita da due punti, così come facciamo in linguaggio assembly. L'ultima linea *deve* essere END, in modo che l'assemblatore possa riconoscere la fine delle istruzioni in linguaggio assembly. Dopo aver digitato la riga END, ed aver premuto RETURN, comparirà un

altro numero di riga. Possiamo segnalare allo ZEN che vogliamo fermarci digitando un punto (.) e premendo RETURN. Lo ZEN tornerà nella sua condizione di attesa, e sarà di nuovo visualizzato Z>.
Cosa dobbiamo fare ora? Abbiamo una serie di caratteri in una parte di memoria — il "buffer" — e basta. Possiamo riesaminare questi caratteri da ORG a END, modificarli, aggiungerne degli altri, come vogliamo. Possiamo scegliere una o più righe da rivedere ed apportare i cambiamenti che desideriamo, proprio come in un programma BASIC; l'unica differenza è che le istruzioni sono diverse.
Lo ZEN termina sempre annotandosi, nella sua memoria, l'ultima riga che è stata immessa o che è stata visualizzata (la riga corrente), per cui, se vogliamo rivedere la prima riga, dobbiamo cambiare l'indirizzo della riga corrente. Per ora, essa è quella DOPO la riga END. Ritorniamo alla prima riga, digitando T seguito da RETURN. Verrà allora visualizzata la prima riga, la riga ORG. L'annotazione tenuta dallo ZEN, il suo "puntatore", sta infatti indicando questa riga e quindi essa verrà stampata. Provate a digitare P5(RETURN). Questo significa: "stampa cinque righe a partire dalla posizione corrente del puntatore". Il puntatore indica la prima riga quando date questa istruzione, quindi verranno stampate sullo schermo le prime cinque righe del programma, ma dopo l'esecuzione di questa istruzione, il puntatore si troverà alla quinta riga. Se vedete comparire il messaggio EOF dopo aver premuto RETURN, significa che siete giunti alla FINE DEL FILE (END OF FILE); non ci sono altre righe del programma da visualizzare. La Fig. 8.1 riassume alcune delle istruzioni dello ZEN che intervengono sul programma nella memoria di transito.

LE ISTRUZIONI DEL BUFFER DELLO ZEN

Dopo aver battuto una serie di istruzioni in linguaggio assembly, questo "testo" è memorizzato sotto forma di codice ASCII nel buffer (memoria di transito), una parte della RAM che inizia a C000H. Se il programma non è troppo lungo, potete usare gli indirizzi sopra questa memoria di transito per immagazzinare il codice macchina, come abbiamo fatto in questo libro.
Il metodo più semplice di controllare lo spazio occupato dal programma è di immettere il linguaggio assembly *senza l'istruzione LOAD.* Quando il codice è assemblato, non verrà caricato in memoria. Potrete allora usare il comando "H" per vedere quanto spazio della memoria di transito sia stato occupato e quale sia lo spazio ancora disponibile della memoria. Se lo spazio sopra il buffer è insufficiente, potete immettere il codice macchina *sotto lo ZEN,* a partire dall'indirizzo 8000H, dove si trova normalmen-

te un programma BASIC. Il metodo è sicuro, purché non ritorniate al BASIC e non scriviate delle righe BASIC. Una volta trovato uno spazio sicuro, potete aggiungere LOAD al testo e procedere come al solito. Quando il testo in linguaggio assembly è nel buffer, può essere richiamato usando le istruzioni del buffer stesso. Lo ZEN mantiene un'indicazione della "riga corrente", ed il richiamo verrà eseguito su questa riga corrente. Se, per esempio, digitate P(RETURN), la riga corrente sarà stampata sullo schermo. Tutti i comandi qui elencati operano sulla riga corrente o cambiano la riga corrente. La maggior parte dei comandi può usare un numero decimale dopo la lettera, e *tutti* richiedono l'uso del tasto RETURN.
D...(Down) Sposta la riga corrente verso il basso. Se la riga corrente è 3, per esempio, D la sposta a 4. D3 la sposterà da 3 a 6.
E...(Enter) Sposta la riga corrente verso il basso, la rinumera, e visualizza il vecchio numero. Potete ora immettere nuove istruzioni. Al termine, dovrete immettere un punto (.). Per esempio, se la riga corrente è 6, E darà a questa riga il numero 7, e potete immettere una nuova riga 6. Se immettete diverse righe, tutte quelle seguenti avranno una nuova numerazione.
H...(Howbig) Visualizza la quantità di memoria occupata dal buffer. Vengono visualizzati l'inizio del buffer (sempre C000H), la fine e l'indirizzo massimo di memoria utilizzabile.
K...(Kill) Cancella il file, in modo da immetterne un altro. Come regola generale, potete solo inserire qualcosa in un file. Se cancellate un file involontariamente, potete recuperarlo se conoscete l'indirizzo della fine del buffer. Questo indirizzo deve essere immesso in memoria con POKE agli indirizzi A10BH, A10CH per ripristinare il file. Per esempio, se cancellate un file che terminava a C29FH, allora dovete immettere 9FH all'indirizzo A10BH, e C2H all'indirizzo A10CH per ripristinare il file. Ciò è possibile solo se non avete già inserito qualcos'altro nel file.
N...(New) Richiama la riga corrente e la pone sullo schermo, con il cursore alla fine della riga. Potete aggiungere qualcosa, o cancellare con il tasto BS prima di cambiare la riga. I normali tasti—funzione non sono attivati.
P...(Print) Stamperà la riga corrente. Potete far seguire a P un numero per stampare un dato numero di righe a partire da quella corrente. Per esempio, P5 stamperà 5 righe. P99 viene usato di solito per stampare tutto il file, se é breve. Se il vostro comando indica un numero di riga superiore a quelle del programma, l'ultima riga che è visualizzata è sempre EOF —Fine del File.
T...(Target) Significa "prendi una nuova riga corrente". T da solo sceglie sempre la prima riga del file. T seguito da un numero sceglie la riga con quel numero — ma ricordate che il numero cambia se aggiungete o togliete qualcosa dal file.
U...(Up) Sposta la riga corrente verso l'alto. Per esempio, se la riga cor-

rente è 6, con U diventerà 5. Usando U3 la riga corrente si sposterà 3 posti più in alto.
Z...(Zap) Cancellerà la riga corrente. Z2 cancellerà 2 righe, a partire da quella corrente. Tutte le restanti righe vengono numerate di nuovo.

Fig. 8.1. Le istruzioni dello ZEN che agiscono sul "buffer del testo".

Supponiamo, quindi, che sia tutto a posto. Vediamo ora quanto è grande il "file di testo". Digitate H, e premete RETURN. Sul video comparirà qualcosa tipo:

```
C000    C04F    F37F
```

C000 è l'indirizzo. in forma esadecimale, a cui inizia la memoria di transito. Le parole di linguaggio assembly che avete digitato sono memorizzate come codici ASCII, a partire da questo indirizzo. Il file ASCII si estende fino all'indirizzo C04FH, e la sommità della memoria è F37F; *non dobbiamo* usare un indirizzo oltre quel limite. Ora, che cosa dobbiamo fare se vogliamo che lo ZEN generi il codice macchina ed il programma sia eseguito? Il primo passo è di fare in modo che lo ZEN assembli il programma, cioè converta le istruzioni che abbiamo digitato in codice macchina. Il comando dello ZEN per questa azione è A (Assemble), seguito sempre da RETURN. Ma l'azione non avviene subito: apparirà, infatti, la parola OPTION?, che vi chiede dove volete che venga riportato il testo assemblato — sul video (V) o su una stampante (P) o su un altro sistema esterno (E). Potete anche ignorare l'opzione e premere solo RETURN. Questa è l'opzione più veloce, e non verrà visualizzato niente sullo schermo, a meno che non ci sia un errore. Tuttavia, finché siete ancora alle vostre prime esperienze, sarà meglio che scegliate l'assemblaggio sullo schermo con V (RETURN), in modo da vedere che cosa accade. Le righe d'istruzione, insieme al codice macchina generato, appariranno sullo schermo, come mostrato nella Fig. 8.2. Vedremo poi le altre opzioni.

```
1                          ORG   0F000H
2                          LOAD  0F000H
3 F000 01FFFF              LD    BC,0FFFFH
4 F003 0B        LOOP:     DEC   BC
5 F004 78                  LD    A,B
6 F005 B1                  OR    C
7 F006 20FB                JR    NZ,LOOP
8 F008 C9                  RET
9                          END
```

Fig. 8.2. Un tipico assemblaggio su video.

Dal momento che usate lo ZEN e avete immesso l'istruzione LOAD 0F000H, il codice macchina generato dall'assemblatore sarà immesso in memoria a partire da questo indirizzo, e possiamo controllare che ci sia effettivamente. Prima di far questo, esaminiamo ancora il programma assemblato. Notate che non è stato creato un codice macchina corrispondente alle istruzioni ORG, LOAD e END. Questo perché sono istruzioni rivolte al programma ZEN e quindi non fanno parte del nostro programma che deve essere assemblato. Gli indirizzi sono mostrati nel listato assemblato, ma senza un'istruzione LOAD nessun byte di codice macchina viene effettivamente immesso a questi indirizzi, per cui potreste in tal caso indicare anche degli indirizzi inesistenti o appartenenti alla ROM. Controlliamo ora che i bytes siano effettivamente in memoria. Normalmente non lo si fa, ma quando provate un nuovo programma, è sempre una buona idea provare tutto prima, in modo da saper poi che cosa fare quando ne avrete bisogno. Annotatevi quali sono i bytes del codice macchina e poi premete Q (Query = richiesta) e fatela seguire (senza virgola né spazio) da F000H (RETURN). In questo comando NON dovete usare lo zero prima del numero. Il comando Q richiede allo ZEN di stampare sullo schermo, in forma esadecimale, il contenuto della memoria a partire da un indirizzo specifico, e visualizzando otto righe di otto bytes ciascuna. Usando Q in questo esempio, dovreste ottenere quanto mostrato dalla Fig. 8.3.

```
F000 01 FF FF 0B 78 B1 20 FB.x1 {
F008 C9 22 D2 FD FB C9 FE 3A I"R}{I~:
F010 C0 E5 C5 F5 21 00 00 01 @eEu!...
F018 18 28 CD EC 07 E5 E1 DB .(Ml.ea[
F020 98 E6 7F FE 20 38 03 CD f.~ 8.M
F028 A5 00 23 10 ED 3E 0A CD %.#.m>.M
F030 A5 00 3E 0D CD A5 00 06 %.>.M%..
F038 28 0D 20 DE F1 C1 E1 C9 (. ^qAaI
```

Fig. 8.3. L'uso del comando "Q" per stampare il contenuto di una data parte della memoria.

La parte sinistra indica l'indirizzo di inizio di ciascuna riga, seguito dagli otto codici esadecimali della riga, poi dai caratteri ASCII che corrispondono ai codici esadecimali, con un trattino al posto di ogni carattere che non può essere stampato.
La ragione dell'uso di questa ulteriore visualizzazione ASCII è che in tal modo possiamo facilmente riconoscere quando un gruppo di bytes esadecimali è stato usato per un messaggio, o quando invece sono stati usati i

caratteri grafici. Senza queste colonne extra avremmo difficoltà a separare questi caratteri dai codici d'istruzione del programma. Al momento, tuttavia, ci interessano solo i bytes che iniziano a F000H. Dovrebbero essere i bytes del codice macchina generati dal programma e caricati in questi indirizzi di memoria — controllateli. Non fatevi confondere da ciò che segue i bytes del programma. Se avete appena acceso il vostro computer MSX per far eseguire questo programma, allora gli indirizzi di memoria che non avete usato saranno occupati da bytes 00 e FF alternati. Se avevate dei programmi in questo spazio di memoria prima di caricare lo ZEN ed iniziare l'assemblaggio, può darsi che questi indirizzi siano stati usati e quindi vi troverete degli altri bytes. Non considerateli, non possono interferire con il vostro programma, a meno che allo Z80 non sia consentito di usarli come istruzioni del programma stesso. Poiché il nostro programma termina con RET, lo Z80 non potrà leggere altri indirizzi dopo F008H.
Ora che sappiamo con certezza che il nostro programma esiste in memoria (e, crediatelo o no, alcuni programmi assemblatori richiedono la registrazione su cassetta e poi il caricamento in memoria per effettuare questo controllo!), possiamo passare alla sua esecuzione. Attualmente, dopo l'esecuzione del programma, ritorniamo al BASIC, per cui dobbiamo vedere il da farsi per far invece ritornare il programma allo ZEN.
Il comando dello ZEN equivalente a "RUN" è G (GOTO) indirizzo (RETURN), per cui digitiamo GF000H (RETURN). Sullo schermo apparirà la parola BRKPNT (Breakpoint), che vuol dire "volete interrompere il programma in qualche punto per vedere che cosa ha fatto fin lì?". In effetti, ci occorre questa facilitazione perché, se facciamo eseguire tutto il programma, esso ritornerà al BASIC (a causa dell'istruzione RET), e in tal caso il computer tornerà nella condizione iniziale, cioé apparirà la scritta del copyright Microsoft, come se aveste perso tutto. In effetti, non è così, e la Fig. 8.4 vi mostra come ritornare allo ZEN in questa evenienza, ma non è proprio il metodo ideale dover fare tutto questo.

	DIGITARE	PREMERE
	DEFUSR=&H A000	RETURN
*	A=USR(0)	RETURN

* Potete usare KEY1, "A=USR(0)"+CHR$(13) (RETURN) e poi premere F1

Fig. 8.4. Come ritornare allo ZEN dal BASIC.

Quando compare la richiesta BRKPNT, digitate F008H (in modo da sostituire a RET il ritorno allo ZEN), premete RETURN, e dopo mezzo secondo comparirà di nuovo sullo schermo Z>, perché il programma ha

causato un mezzo secondo di ritardo per l'esecuzione del ciclo. Dopo l'esecuzione, il punto d'arresto (breakpoint) è sostituito dal byte originario di RET, per cui il programma ora è nuovamente nel suo stato originario.
Ora, come facciamo a sapere che il programma è stato eseguito? Premete X (RETURN) per esaminare (eXamine) i registri dello Z80. Li troverete come mostrato nella Fig. 8.5 e sotto le lettere PC dovreste trovare F008. Questo è l'indirizzo a cui il programma è ritornato allo ZEN, dal momento che il punto d'arresto era stato posto qui.

HL	DE	BC	AF	RI	
0000	0000	0000	0044	2700	
0000	0000	0000	0000		(a)
IX	IY	SP	PC		
0000	0000	A166	F008		

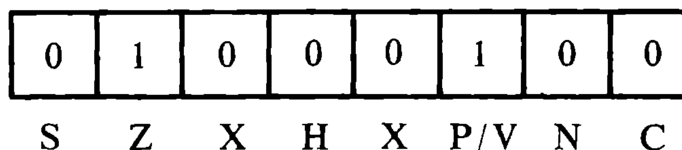
HL	DE	BC	AF	RI	
0000	0000	FFFE	FF44	5000	
0000	0000	0000	0000		(b)
IX	IY	SP	PC		
0000	0000	A166	F005		

Fig. 8.5. L'uso di X per esaminare i registri dello Z80. Il contenuto dei registri alternativi è mostrato sotto il contenuto dell'insieme principale. (a) interrompendo all'indirizzo F008H; (b) interrompendo all'indirizzo F005H.

L'unico registro che contiene qualcosa d'interessante è AF, in cui si trova 0044. Ciò significa che il registro A contiene 00 e il registro F 44. Sono numeri esadecimali, ricordatevelo. La cifra 44H nel registro dei flag significa che i flag 2 e 6 sono posti a 1 (vedi Fig. 8.6).

44H= 01000100 in binario

Registro dei Flag



Flag di zero posto a 1
Flag P/V posto a 1
Tutti gli altri flag sono a zero

Fig. 8.6. Analisi del byte contenuto nel registro del flag.

Il flag 6 è il più importante per noi — è il flag di zero, ed è posto a 1 perché abbiamo eseguito un programma di conteggio alla rovescia che terminava quando uno zero era il risultato dell'azione OR C.
Ora usiamo nuovamente G, ma quando compare la richiesta BRKPNT, rispondiamo con F005H. Il primo punto d'arresto a F008H è stato sostituito dall'originario codice di RET, come ricorderete. Questa volta non ci sarà ritardo: infatti il ciclo di ritardo non viene eseguito perché abbiamo immesso il punto d'arresto subito dopo il passo LD A,B. Usando X per esaminare i registri dello Z80, troveremo un quadro diverso. Questa volta, BC contiene FFFEH, come ci aspettavamo (era stato caricato con FFFFH e poi decrementato). Il registro A contiene FF, come dovrebbe essere dopo LD A,B. Il registro F contiene ancora 44 perché le azioni di caricamento non modificano i flag.

---

1. Usate CLEAR con un numero d'indirizzo inferiore di uno all'indirizzo d'inizio
   (Se il vostro programma inizia a F000H, CLEAR 200,&HEFFF)

2. Assegnate una variabile al primo indirizzo del vostro programma
   Esempio: A!=&HF000

3. Digitate una riga di DATA con i bytes in forma esadecimale
   Esempio: 100 DATA FE,03,20,12,23,23,5E,23,23,56,EB,C9

4. Contate i termini della riga DATA, a partire da O,1,2,...etc.
   Esempio: i termini della riga qui sopra sono 11.

5. Effettuate un ciclo per il caricamento, usando una variabile intera
   Esempio: 20 FOR N%=0 TO 11

6. Usate VAL("&H"+D$) per leggere i valori e immetterli in memoria
   Esempio: 30 READ D$
   40 POKE A!+N%,VAL("&H"+D$)
   50 NEXT

7. Stabilite l'indirizzo di DEFUSR
   Esempio: 60 DEFUSR=A!

8. Effettuate la chiamata con USR
   Esempio: 70 A=USR(0)

Fig. 8.7. Un sommario dei passi da seguire per immettere i bytes in memoria con un programma BASIC

Ora, sia che usiate un programma breve e semplice come questo, o uno che abbia invece centinaia di istruzioni, i passi per caricare lo ZEN, digitare il programma in linguaggio assembly ed immetterlo nel buffer, eseguire l'assemblaggio e poi controllarlo, sono sempre gli stessi. Quello che cambia in modo significativo è l'effettivo linguaggio assembly, per cui ci concentreremo su quello d'ora in poi, senza più utilizzare programmi BASIC che usano la funzione POKE. Se non avete lo ZEN, potete sempre leggere i listati in forma esadecimale, che ho stampato con lo ZEN, e ormai ne sapete abbastanza per immettere per conto vostro i bytes esadecimali in memoria, con la funzione POKE. I passi illustrati nella Fig. 8.7 dovrebbero aiutarvi a ricordare ciò che deve essere fatto.

Se invece tenete il computer acceso, con lo ZEN in memoria potrete effettuare diversi assemblaggi senza cambiare nulla. Quando avete finito un assemblaggio, basta digitare K (Kill), premere RETURN e lo ZEN sarà pronto a ricevere altre istruzioni in linguaggio assembly. Se, com'é naturale, volete conservare i vostri primi tentativi in linguaggio assembly, lo ZEN vi permette di far questo. Prima di cancellare il programma con K, inserite una cassetta nel registratore e riavvolgete il nastro. Premete i tasti RECORD e PLAY, poi digitate W (RETURN). Sullo schermo apparirà la parola NAME, che vi chiede di immettere un nome di file per questo programma. Potete usare un nome qualsiasi, purché non ecceda i 6 caratteri, poi premete RETURN. Il registratore partirà, registrerà tutto il programma in linguaggio assembly e poi si fermerà. In ogni momento, se avete lo ZEN in memoria, potrete ricaricare questo programma digitando R, seguito da RETURN, e poi il nome del file e (RETURN), oppure semplicemente RETURN. Notate però, che questa registrazione può essere letta SOLO dallo ZEN e NON è una registrazione dei bytes in codice macchina — questa è fatta in modo diverso. La registrazione effettuata con W e letta da R è la registrazione del testo del linguaggio assembly, ed è in codice ASCII. Potete leggerla con l'istruzione LOAD *(non* CLOAD) del computer MSX. Il salvataggio di questo testo vi permette di ricaricare il vostro linguaggio assembly in ogni momento, in modo da modificarlo in un diverso programma, e dovreste sempre ricordarvi di salvare ogni vostro testo in linguaggio assembly.

Potete usare lo ZEN anche per creare un nastro di codice macchina. Questa è una registrazione solo dei bytes del programma in codice macchina, il tipo di programma, cioé, che può essere caricato da BLOAD in BASIC. Per far questo, dovete conoscere l'indirizzo d'inizio del programma che è in memoria, l'indirizzo di fine e l'indirizzo di esecuzione. L'indirizzo di esecuzione è l'indirizzo della prima istruzione del programma, e per molti programmi coinciderà con l'indirizzo d'inizio. Tuttavia, molti programmatori preferiscono porre all'inizio dei loro programmi una tabella di bytes, e quindi l'indirizzo di esecuzione può essere spostato più avanti. Se l'indirizzo di esecuzione coincide con quello d'ini-

zio, non deve essere specificato. Inoltre, lo ZEN vi chiederà un indirizzo di LOAD, se i bytes devono essere caricati ad un indirizzo diverso da quello usato per l'assemblaggio.
Per provare questo modo di scrittura, usate lo stesso programma del ciclo di ritardo e assemblatelo. Dovete conoscere gli indirizzi di inizio e di fine dei bytes del codice macchina, che potrete leggere dal listato dell'assemblatore. In questo caso, l'indirizzo d'inizio è F000H e quello di fine è F008H. Gli indirizzi di LOAD (caricamento)ed EXEC (esecuzione) sono gli stessi, F000H. Quando il programma è stato assemblato, preparate una cassetta con del nastro libero. Digitate WB e poi RETURN e immettete le cifre per gli indirizzi di START, STOP, LOAD ed EXEC, tutti in forma esadecimale con H alla fine del numero. Se dimenticate la "H", lo ZEN risponderà con "HUH?" e dovrete ricominciare da capo. Alla fine, vi sarà richiesto un nome e potete usare fino a 6 caratteri — provate RITAR. Premete i tasti RECORD e PLAY del registratore *prima* di premere RETURN dopo aver immesso il nome del file. Il programma in codice macchina verrà quindi registrato. Per controllarlo, dovete cancellare il codice macchina in memoria e ricaricarlo dal BASIC. Digitate allora F (fill). Vi saranno richieste le informazioni per START, STOP e DATA. Immettete F000H per START, F008H per STOP e 00H per DATA. Quando premete RETURN, questa istruzione riempirà la memoria fra gli indirizzi START e STOP con i bytes di dati specifici. Questo procedimento ripulisce la memoria fra gli indirizzi F000H e F008H compresi. Potete ora tornare al BASIC digitando B e poi premendo RETURN. Riavvolgete il nastro, premete PLAY e digitate BLOAD "RITAR" — o qualsiasi altro nome abbiate usato per il file. Il nastro si avvierà e quando si arresta, dovete ritornare allo ZEN. Se avete già immesso DEF USR=&HA000, dovete solo digitare A=USR(0). Se non avete immesso prima la parte DEF USR, fatelo ora e con A=USR(0) tornerete allo ZEN. Quando vedrete il messaggio dello ZEN, usate QF000H per vedere che cosa è stato immesso in memoria: dovreste vedere il vostro programma. Questo è il metodo che avete usato per immettere in memoria programmi in codice *senza* lo ZEN, ma con lo ZEN potete usare il comando RB (Read Byte) invece di BLOAD.
Inoltre, con lo ZEN potete usare RB per caricare un programma in codice macchina che non avete creato voi, se non occupa nessuno degli indirizzi usati dallo ZEN. Una volta in memoria, potete usare il comando "d" per lo ZEN (*dovete* usare la lettera minuscola "d", e usare poi le maiuscole per altre parti dell'istruzione). Il comando "d" significa "disassembla", e farà sì che lo ZEN converta il codice macchina in linguaggio assembly *per quanto possibile.* Se il codice macchina non ha tabelle di bytes di dati, il disassemblaggio sarà perfetto e potrà essere stampato sullo schermo, sulla stampante o su un'altra periferica. Quando vi si richiede di scegliere l'opzione, OPTION, potete però premere semplicemente RETURN: in tal

caso il codice disassemblato viene *aggiunto* al vostro file di testo. Se non c'era, verrà creato ora. In questo modo, potete crearvi un file di testo in linguaggio assembly da una parte di programma in codice macchina. Questo vi permette di vedere come funziona il codice macchina, farvi delle aggiunte, modificarlo, prenderne delle parti, quello che volete, insomma. È un'altra caratteristica che rende lo ZEN un indispensabile aiuto per il programmatore in codice macchina.

ALTRI COMANDI DELLO ZEN

Lo ZEN è un programma così completo che richiede parecchio spazio per descrivere tutte le altre sue caratteristiche, oltre quelle essenziali che abbiamo visto finora. Possiamo iniziare con le istruzioni di EDITOR, di cui abbiamo finora usato solo E (Enter) ed A (Assemble). Potremmo aggiungere S (Sort), che visualizzerà un elenco in ordine alfabetico delle label usate nel programma assembly (file di testo). L'Editor, come potete supporre, ha delle istruzioni che permettono di controllare il vostro file in linguaggio assembly e cambiarlo come volete. La funzione di richiamo usa i numeri che appaiono sullo schermo, ma questi numeri non sono fissi e non sono registrati con il file. Ogni modifica sui numeri, come l'inserimento o la cancellazione di una riga, farà sì che lo ZEN rinumeri le righe *automaticamente.* Lo ZEN, come abbiamo visto, ha un "puntatore" sulla riga corrente, e molti di questi comandi riguardano infatti il puntatore. Il comando D, per esempio, lo sposterà verso il basso e potete specificare di quante righe. D5, ad esempio, sposterà il puntatore verso il basso di 5 righe. Il puntatore può essere spostato verso l'alto con U, e U3 lo sposterà di due righe più su. Se volete andare ad una data riga, usate T (target). T25 vi darà la riga 25 — se esiste. Se con questo comando chiedete un numero di riga che vi porterebbe oltre la fine del file, avrete il messaggio EOF (End Of File) ed il puntatore si troverà alla fine del file. T, usato senza numeri di riga, vi porterà all'inizio del file. Quando avete posizionato il puntatore. potete usare P pre stampare le righe. P da solo stamperà la riga corrente, quella a cui si trova il puntatore, ma potete dare anche il comando P10, che stamperà dieci righe a partire da quella corrente.
Potete cancellare delle righe del file con Z (Zap). Z5, per esempio, cancellerà 5 righe, a partire dalla riga a cui si trova il puntatore. Le restanti righe saranno numerate di nuovo. Se volete cancellare tutto il file, usate K (Kill). Vi potrebbe però capitare di usare K per sbaglio. Questo comando, comunque, *non* cancella la memoria, e potrete ancora ripristinare il vostro file! Dovete conoscere l'indirizzo di fine del file, per cui è sempre bene usare il comando H e segnarsi gli indirizzi dopo aver assemblato un programma. Supponiamo che l'indirizzo della fine del file sia C15AH. Per recuperare il vostro file, dovete immettere questo indirizzo, prima il byte basso e poi il byte alto, agli indirizzi A10BH e A10CH. Fatelo con il

comando M (Modify=Modifica memoria). Digitate MA10BH (RETURN). Probabilmente apparirà il numero 00. Digitate 5AH, il byte basso dell'indirizzo, nel nostro esempio, e premete RETURN. Ora apparirà il byte C0. Digitate C1H (il byte alto) e di nuovo RETURN. Ora digitate un punto (.) e premete ancora RETURN. Avrete così immesso l'indirizzo C15AH nello ZEN. Con T, e poi P, potrete vedere che il vostro file di testo è stato recuperato.
Supponete di voler trovare qualcosa nel vostro file, ad esempio dove avete usato la label LOOP7. È semplice, con lo ZEN; usate T per andare all'inizio del file, poi digitate LLOOP7 (L per Locate, posiziona) e premete RETURN. La riga che usa LOOP7 sarà la riga corrente e verrà stampata sullo schermo. Se volete aggiungere delle righe di testo al vostro file, spostate il puntatore in modo che la riga visualizzata, la riga corrente, sia quella che seguirà immediatamente la vostra nuova riga (o righe). Poi usate E (Enter) ed immettete il nuovo testo, usando sempre il punto per terminare l'immissione. Tutte le righe saranno quindi rinumerate. Se volete cambiare una riga, spostate il puntatore, usando T, U, D o L e poi digitate N (RETURN). La riga verrà visualizzata ed il tasto BS potrà essere usato per cancellare la riga in modo che possiate ribatterla. Non è un sistema così pratico come quello del BASIC, ma è perfettamente adeguato alle brevi righe del linguaggio assembly.

I COMANDI DELLA FUNZIONE DI MONITOR DELLO ZEN

I comandi della funzione di monitor (controllo) dello ZEN vi permettono di lavorare con un programma in codice macchina come parte separata da quello che avete in linguaggio assembly. Molto spesso, potete avere un lungo programma in codice macchina di cui volete cambiare solo alcuni bytes. Si perderebbe molto tempo a riportarlo in linguaggio assembly e poi a riassemblarlo. Tuttavia, ci sono dei comandi appositamente previsti per lavorare con il codice macchina invece che con il file di testo. Fra essi, Q ed M sono quelli che probabilmente userete più spesso. Q deve essere seguito da un indirizzo (senza spazio), e quando premete RETURN, vedrete i bytes del programma in forma esadecimale. M significa Modifica (Modify) e ne abbiamo appena visto un esempio. Quando digitate M seguita da un indirizzo (senza spazio) e poi premete RETURN, vedrete il byte che si trova a questo indirizzo, potete quindi digitare un nuovo byte (ricordate di porre "H" alla fine se è in forma esadecimale) e premendo RETURN, il byte sarà cambiato con quello nuovo. In alternativa, potete premere solo RETURN, il che lascerà immutato il byte, e si sposterà al byte dell'indirizzo successivo. Potrete cambiarlo o lasciarlo immutato e spostarvi all'altro byte successivo. L'azione di M termina digitando un punto e premendo RETURN.
Altri due comandi sono utili per i blocchi di bytes. Se volete trasferire un

blocco di bytes, per esempio dalla ROM alla RAM, potete usare C (Copy). Vi si richiederà allora l'indirizzo d'inizio, START, del blocco da copiare, poi l'indirizzo di fine, STOP. Infine, vi verrà chiesta la destinazione, DESTINATION, l'indirizzo a partire dal quale volete la copia. Naturalmente, potete usare solo "d" per disassemblare il codice macchina e aggiungerlo ad un file di testo, e questo è spesso più utile. È più pratico, invece, usare C quando si deve spostare una tabella di valori. L'altro utile comando per i blocchi è F (Fill), che abbiamo già usato. Vi permette di riempire una serie d'indirizzi della RAM con un byte di dati, ed è particolarmente utile per azzerare un blocco di memoria. Lavorando con il codice macchina dello Z80, è utile azzerare la memoria libera, perché il byte di 0 è il codice di "nessuna operazione" per lo Z80. La lettura di una serie di zeri non farà compiere nulla di strano allo Z80 — ma semplicemente lo Z80 andrà a vedere che cosa c'é alla fine degli zeri! Un'ultima parola. Quando usate lo ZEN con una stampante, vedrete che ogni assemblaggio stampato inizia con un salto a pagina nuova. Se vedete che ciò causa uno spreco di carta, potete fare una versione modificata dello ZEN, eliminando questa azione. Ecco come: digitate MA6DFH (RETURN). Vedrete 00, allora digitate FEH (RETURN), 0CH (RETURN), C8H (RETURN), e infine (RETURN). Questo immette una modifica in una parte libera della RAM entro lo ZEN che impedisce che il codice 0CH del salto pagina sia trasmesso alla stampante. Dopo aver fatto questo, controllate che funzioni e tornate al BASIC, con B. Ora registrate il vostro nuovo ZEN, alla velocità più alta con SCREEN,,,2; poi con BSAVE" CAS:NEWZEN",&HA000,&HBB5C per salvare lo ZEN. Se vi occorre, potete eliminare anche il salto riga (se la vostra stampante stampa due spazi per ogni riga) usando la stessa istruzione, ma sostituendo 0AH a 0CH. Potete usare anche FE 0C C8 FE 0A C8 per limitare sia i caratteri del salto riga che del salto pagina. Lo ZEN è stato progettato in modo da consentirvi di modificarlo facilmente, e questo contribuisce a farne indubbiamente uno dei migliori programmi assemblatori.

CAPITOLO 9

L'ultima battuta

Questo capitolo è dedicato ai programmi, per lo più scritti in linguaggio assembly, ma alcuni dei più particolari sono in BASIC. Hanno tutti un fattore in comune, usano la conoscenza del funzionamento interno del computer MSX per raggiungere il risultato desiderato. Sono quindi tutti utili, anche se non vi interessa l'azione svolta dal programma. Ognuno di essi mostra come si può eseguire qualche azione utile o come si possono estrarre dei dati utili. Per questo motivo, probabilmente scoprirete che i programmi che voi volete scrivere possono essere un adattamento dei programmi qui presentati. Ho cercato di variare il più possibile questi programmi, ma quando un sistema è abbastanza nuovo, come l'MSX nel momento in cui scrivo, non è facile sapere proprio tutto su di esso. Quando il sistema è ormai usato da anni, è sempre molto più facile sapere, dalle riviste, dalle lettere degli utilizzatori e dai libri, quali routines siano nella ROM, dove siano i loro indirizzi di inizio, e come esse usino i registri dello Z80. Ecco perché i computer più "vecchi" sono sempre il sogno dei programmatori in codice macchina.

SPOSTARE IL PROGRAMMA

Nel Cap. 7, ho menzionato la possibilità di spostare un programma scritto in BASIC in un'altra parte della memoria. Fatto questo, potete caricare ed eseguire un altro programma BASIC e poi ritornare a quello originale. Questo procedimento è molto più veloce che non registrare il primo programma su nastro, e poi ricaricarlo, perché il nastro è lento e non sempre così affidabile come vorremmo. Oltre a salvare le righe del BASIC, si può salvare anche la VLT, e quindi le variabili del programma. Potete così riprendere il programma, se è stato interrotto, purché usiate GOTO (con un numero di riga) invece di RUN per ricominciare l'esecuzione. Se usate RUN, tutti i valori delle variabili sono inizializzati ed il programma ricomincia da capo. Ricordatevi, fra parentesi, che potete usare BSAVE per salvare un programma BASIC che è stato fermato a metà dell'esecuzione. Se usate BSAVE da &H8000 fino alla fine della VLT, quando il programma verrà ricaricato, sarà possibile riprenderne l'esecuzione con GOTO. Dovete però far attenzione al fatto che "l'indirizzo di esecuzione" per un programma BASIC *non è* &H8000. Dovete cercare nella ROM questo indirizzo, denominato indirizzo di "inizio a caldo" per il BASIC. L'indirizzo di "inizio a freddo" è 0000, ma questo ripulirà la parte più bassa della memoria. Attualmente, l'indirizzo &H3297 sembra funzionare, an-

```
 1                     ORG    0F000H
 2                     LOAD   0F000H
 3  F000   010800      LD     BC,08H        ;8 bytes di puntatori
 4  F003   21C0F6      LD     HL,0F6C0H     ;inizio puntatori
 5  F006   1100D0      LD     DE,0D000H     ;destinazione
 6  F009   EDB0        LDIR                 ;trasferimento
 7  F00B   D5          PUSH   DE            ;salva indirizzo
 8  F00C   2B          DEC    HL            ;indietro di uno
 9  F00D   56          LD     D,(HL)        ;da 0F6C7
10  F00E   2B          DEC    HL
11  F00F   5E          LD     E,(HL)        ;da 0F6C6
12  F010   210080      LD     HL,8000H      ;inizio BASIC
13  F013   E5          PUSH   HL            ;salva
14  F014   EB          EX     DE,HL         ;scambia
15  F015   ED52        SBC    HL,DE         ;contatore in HL
16  F017   4D          LD     C,L
17  F018   44          LD     B,H           ;ora anche in C
18  F019   E1          POP    HL            ;riprende 8000
19  F01A   D1          POP    DE            ;riprende D007
20  F01B   79          LD     A,C           ;cont. byte basso
21  F01C   12          LD     (DE),A        ;in memoria
22  F01D   13          INC    DE
23  F01E   78          LD     A,B           ;cont. byte alto
24  F01F   12          LD     (DE),A        ;in memoria
25  F020   13          INC    DE            ;pronto al trasferimento
26  F021   EDB0        LDIR                 ;esegui!
27  F023   C9          RET
28  F024   010800      LD     BC,08H        ;cont. puntatori
29  F027   2100D0      LD     HL,0D000H     ;da D000H
30  F02A   11C0F6      LD     DE,0F6C0H     ;a F6C0H
31  F02D   EDB0        LDIR                 ;esegui trasf.
32  F02F   4E          LD     C,(HL)
33  F030   23          INC    HL
34  F031   46          LD     B,(HL)        ;memorizza cont.
35  F032   23          INC    HL
36  F033   110080      LD     DE,8000H      ;inizio BASIC
37  F036   EDB0        LDIR                 ;trasferimento
38  F038   C9          RET
39                     END
```

Fig. 9.1. Il programma di salvataggio del BASIC in linguaggio assembly, stampato dallo ZEN. Dal momento che sono indicati anche i bytes, potete facilmente immetterli in memoria con un altro programma BASIC che usa POKE se non avete lo ZEN.

che se dà il messaggio “Syntax error”. Il linguaggio assembly per il programma di salvataggio del BASIC è mostrato nella Fig. 9.1
Questo programma, come tutti gli altri programmi in linguaggio assembly di questo capitolo, è stato scritto usando lo ZEN ed è stato stampato con

l'opzione P per l'assemblaggio. Il programma è commentato, in modo che sia un po' più facile da seguire, ma se anche voi usate lo ZEN, non è strettamente necessario includere i commenti. Il codice macchina è assemblato a F000H, come al solito, e consiste di 39H bytes; è quindi di dimensioni abbastanza ridotte per essere contenuto in una qualsiasi parte libera della memoria, se volete immetterlo ad altre locazioni. Ci sono alcune parti della memoria alta non utilizzate dall'attuale generazione dei computer MSX, per cui potreste usare queste in modo da non aver bisogno di un'altra istruzione CLEAR quando caricate il programma. Vedremo poi i metodi per iniziare l'esecuzione di questi programmi premendo semplicemente il tasto ESC, ma potete sempre assegnare i tasti funzione alle istruzioni USR. Potete usare KEY1,"A=USR1(0)"+CHR$(13) KEY2,"A=USR2(0)"+CHR$(13) e così via, in modo che i tasti funzione facciano iniziare le vostre routines in codice macchina. Dovrete assegnare degli indirizzi per DEFUSR1 e DEFUSR2 in questo esempio, ed il modo più semplice è di preparare un nastro "ad hoc". Si può scrivere un piccolo programma in BASIC per assegnare gli indirizzi DEFUSR, poi esso dovrà caricare con BLOAD le routines in codice macchina, ed infine potrà terminare con NEW, invece che con END, in modo da auto-cancellarsi. Notate, per inciso, che se ridefinite i tasti funzione, potete usare BSAVE per conservare i codici nella RAM in modo da caricarli da nastro.
Ritornando al programma, le righe dalla 3 alla 6 riguardano il salvataggio dei puntatori dei dati della VLT. Questi puntatori sono posizionati da F6C0H in avanti, e se non sono ripristinati dopo che avrete ripristinato le righe del vostro programma, otterrete il messaggio "Syntax error" se cercherete di far eseguire di nuovo il programma. Il registro BC è usato come contatore, per il conteggio di otto, HL è caricato con l'indirizzo d'inizio F6C0H, e DE con il primo indirizzo di destinazione, D000H. Questo indirizzo è stato scelto arbitrariamente: potreste usare un qualsiasi indirizzo che sia stato prima protetto da un'istruzione CLEAR e la vostra scelta dipenderà solo dalle dimensioni del programma BASIC che volete salvare. Se volete salvare dei programmi molto lunghi, può darsi che dobbiate arrivare a 9FFFH o anche più in basso. Naturalmente, se un programma BASIC occupa più della metà della RAM disponibile, non potete usare questo metodo. Quando scrivete, ed utilizzate, programmi così lunghi, è logico supporre che usiate i dischi: in tal caso, il salvataggio su disco è altrettanto semplice quanto il salvataggio in memoria. L'istruzione LDIR della riga 6 esegue il trasferimento dei bytes del puntatore.
Dopo LDIR, il registro DE punta al primo indirizzo libero fra quelli che usiamo per salvare il programma BASIC. Con PUSH DE, salviamo l'indirizzo sullo stack, per usarlo successivamente. Ora dobbiamo vedere la lunghezza del programma, più quella della VLT. Leggiamo allora l'indirizzo della fine della VLT dall'ultima coppia di bytes del puntatore. Dal momento che l'istruzione LDIR ha lasciato HL che puntava ad un indi-

rizzo posto dopo F6C7, abbiamo bisogno di DEC HL per ripristinare l'indirizzo corretto, e possiamo leggere i due bytes dell'indirizzo in D e in E, assicurandoci che il byte basso sia in E e il byte alto in D. La riga 12 carica poi l'indirizzo d'inizio del BASIC, 8000H, in HL e questo indirizzo è salvato sullo stack con l'istruzione PUSH HL della riga 13. La riga 14 scambia gli indirizzi contenuti in HL e in DE. Questo è necessario, perché vogliamo sottrarre e l'unica sottrazione a 16 bit che usa questi registri sottrae l'indirizzo contenuto in DE dall'indirizzo in HL, ponendo il risultato in HL. La sottrazione è eseguita alla riga 15; le righe 16 e 17 immettono il risultato in BC. Questo numero ora può essere usato come conteggio per il trasferimento di tutti i bytes del programma BASIC e della VLT.
Ora prepariamoci a spostare i bytes del BASIC. Con POP HL, rimettiamo l'indirizzo dell'inizio del BASIC, 8000H, in HL, e con POP DE avremo l'indirizzo della prima parte disponibile di memoria libera. Ora dobbiamo salvare il numero del conteggio, perché ne avremo bisogno per la restituzione dei bytes del BASIC nella seconda parte del programma. Non abbiamo un'istruzione per caricare all'indirizzo di DE da B o da C, solo A. Perciò, trasferiamo da C ad A, e quindi a (DE), incrementiamo DE, poi trasferiamo B ad A, e a (DE) e incrementiamo di nuovo. Sembra un procedimento lungo ma occorrono solo pochi bytes di codice macchina. Alla riga 25 è stato tutto eseguito e ora ogni cosa è pronta per il trasferimento. L'istruzione LDIR alla riga 26 esegue il trasferimento e il programma ritorna al BASIC.
Potete ora dare l'istruzione NEW, oppure caricare un altro programma con CLOAD. Quando volete tornare al programma originario, dovete stabilire un'istruzione DEFUSR per l'indirizzo F024H e usare USR per attivarlo. Questo darà l'avvio alla seconda parte dell'esecuzione del programma. Inizia (riga 28) trasferendo i bytes salvati della tabella del puntatore agli indirizzi da F6C0 in avanti. Si può paragonare ad un'immagine speculare della prima parte del programma, con gli indirizzi contenuti in HL e in DE invertiti. L'istruzione LDIR alla riga 31 esegue il trasferimento. Il registro BC viene poi caricato con i bytes del conteggio, che era stato salvato a 0D07H e 0D08H. Fatto questo, con gli indirizzi contenuti in HL, questa volta, DE viene caricato con l'indirizzo di destinazione, 8000H, e un'altra istruzione LDIR esegue il trasferimento. Ora ritorniamo ancora al BASIC, e un'istruzione LIST rivelerà il programma originario.

UN PROGRAMMA PER OTTENERE UNA LISTA DI VARIABILI

Questo è uno dei pochi esempi di programmi BASIC contenuti in questo capitolo. Ma è molto più facile usare il BASIC per questo scopo particolare, mentre non varrebbe la pena usare il codice macchina. Dobbiamo stampare i nomi di variabili usati in un programma BASIC, che si trova

anch'esso in memoria. Per usare il programma della Fig. 9.2, dovete accertarvi che il programma delle cui variabili volete fare un elenco, sia in memoria, e che non abbia numeri di riga superiori a 9999, eventualmente potete rinumerarle.

```
10010 CLS:PRINTTAB(14)"VARIABILI":PRINT:PRINT
10020 VS!=PEEK(&HF6C2)+256*PEEK(&HF6C3)
10030 VF!=PEEK(&HF6C4)+256*PEEK(&HF6C5)—14
10040 VP%=PEEK(VS!):VT$=" "
10050 IF VP%=2 THEN VT$="%"
10060 IF VP%=3 THEN VT$="$"
10070 IF VP%=4 THEN VT$="!"
10080 FOR VN%=1 TO 2
10090 PRINTTAB(VN%+5)CHR$(PEEK(VS!+VN%));:NEXT
10100 PRINTVT$:VS!=VS!+VP%+3
10110 IF VS!<VF! THEN 10040
10120 END
```

Fig. 9.2. Un programma, che usa il BASIC, per ottenere una lista di variabili.

Se avete salvato il programma per ottenere la lista di variabili con CSAVE, potete caricarlo con MERGE, e in tal modo questo programma sarà unito alla fine del programma principale. Date ora il comando RUN in modo da stabilire le variabili nella VLT, e poi potete usare GOTO 10000 per richiamare il programma per la lista di variabili. Vengono elencate solo le variabili semplici, non gli array, ma dal momento che quasi tutti i programmi usano molte variabili semplici e pochi array, questa non è una grave perdita. Le variabili usate dal programma "elencatore" *non* sono mostrate, perché gli indirizzi sono stati scelti proprio in modo da evitare che questo avvenga.
Il principio seguito per la stesura del programma è semplice. Gli indirizzi F6C2, F6C3 danno l'indirizzo d'inizio della VLT, alla quale è stato assegnato il nome di variabile VS!. Analogamente, gli indirizzi F6C4 e F6C5 danno l'indirizzo della fine della VLT, a cui è stato assegnato il nome VF!. Alla riga 10030, il numero 14 viene sottratto da questi indirizzi, in modo che le variabili usate dal nostro programma elencatore non compaiono. È stato usato 14 perché, quando saranno state eseguite le assegnazioni di VF!, dei cambiamenti dovuti all'assegnazione di altre variabili non faranno cambiare il valore di VF!. Cioè, sono stati assegnati 4 bytes ciascuna per VS! e VF!, in tutto 8 bytes. Inoltre, ogni variabile usa altri tre bytes per il codice del tipo ed il nome, in tutto 6 bytes per le due variabili. Ecco i 14 bytes che devono essere sottratti dall'indirizzo finale perché questi nomi non compaiano nell'elenco. Le ultime assegnazioni, VP%,VT$ e VN%, non compaiono in ogni caso, anche omettendo 14.

Alla riga 10040, a VP% viene assegnato il numero del tipo per la prima variabile della tabella, e VT$ sarà usata per il simbolo del tipo, e le righe da 10050 a 10070 definiscono questo simbolo, a seconda del numero di codice del tipo. Non viene assegnato niente se il tipo è la doppia precisione (codice del tipo: 8), assunto automaticamente. La riga 10090 stampa il nome della variabile, fino a 2 caratteri, e il simbolo del tipo. La riga 10100 sposta il numero d'indirizzo di un certo numero di bytes, cioè il numero del tipo più tre (due per il nome, uno per il tipo), e la riga 10110 controlla che la routine sia giunta al termine. Se non è stato ancora raggiunto l'indirizzo di fine, il programma ritorna alla riga 10040 per trovare il numero del tipo della variabile successiva.
Questo è un buon esempio di programma che usa la funzione PEEK in BASIC, ma per il quale sarebbe praticamente inutile usare il codice macchina. Infatti, per scriverlo in codice macchina dovreste fare uno schema molto accurato e, soprattutto, dovreste scrivere routines più elaborate per assegnare i simboli del tipo. Tuttavia, non è poi particolarmente difficile e potreste ugualmente provare a scriverlo, come esercizio. Vedrete, però, che la versione in codice macchina non è più veloce del BASIC, almeno in modo apprezzabile, per cui non vale la pena usarla. Non è detto che dobbiate per forza programmare in codice macchina perché ora sapete farlo!

PER TROVARE LA RIGA

Ecco ora una routine che può essere facilmente scritta tutta in BASIC (provate!), ma che qui viene presentata come un insieme di BASIC e di linguaggio assembly. Si vuole trovare dove sia memorizzata una riga di programma BASIC. Inserite il numero di riga, come vi viene richiesto, ed il codice macchina troverà l'indirizzo d'inizio della riga. Ci *deve* essere uno STOP fra il programma BASIC di cui controllate le righe, e la parte di programma BASIC della Fig. 9.3.

```
10000 STOP
10010 PRINT"NUMERO DI RIGA?"
10020 INPUT D%:DEFUSR1=&HF000
10030 X%=USR1 (D%) : IF X%=0 THEN 10060
10040 PRINT "LA RIGA COMINCIA A" ;65536!+X%;
      "ESADECIMALE ";HEX$(65536!+X%)
10050 END
10060 PRINT "QUESTA RIGA NON ESISTE"
10070 END
```

Fig. 9.3. Il programma che individua una riga. Consiste di un insieme di BASIC e di codice macchina, e qui è mostrata la parte in BASIC.

Al numero di riga che sceglierete è assegnato il nome di variabile D%, il che limita questo numero a 32767, poiché il computer MSX non accetta un numero maggiore, come intero. L'uso di un intero, però, rende più semplice il trasferimento del valore al codice macchina. Il numero di riga viene restituito sotto forma di un altro intero, X%. A causa del modo in cui il computer MSX usa gli interi, questo sarà *sempre negativo* e la riga 10040 è usata per convertirlo in un numero positivo che potrete usare con la funzione PEEK. Come ho già sottolineato, non è però essenziale, perché potete usare anche i numeri negativi con l'istruzione PEEK. Se la riga non esiste, verrà restituito il numero 0; la riga 10030 controlla: se è 0, verrà stampato il messaggio della riga 10060 al posto della riga 10040. Il linguaggio assembly è mostrato nella Fig. 9.4.

```
 1                              ORG   0F000H
 2                              LOAD  0F000H
 3  F000 210080                 LD    HL,8000H
 4  F003 ED5BC0F6               LD    DE,(0F6C0H)
 5  F007 3AF8F7       LOOP:     LD    A,(0F7F8H)
 6  F00A BE                     CP    (HL)
 7  F00B 2815                   JR    Z,TSTHI
 8  F00D 23           BACK:     INC   HL
 9  F00E E5                     PUSH  HL
10  F00F ED52                   SBC   HL,DE
11  F011 E1                     POP   HL
12  F012 38F3                   JR    C,LOOP
13  F014 210000                 LD    HL,00H
14  F017 1805                   JR    EXIT
15  F019 110300       GOTIT:    LD    DE,03H
16  F01C ED52                   SBC   HL,DE
17  F01E 22F8F7       EXIT:     LD    (0F7F8H),HL
18  F021 C9                     RET
19  F022 23           TSTHI:    INC   HL
20  F023 3AF9F7                 LD    A,(0F7F9H)
21  F026 28F1                   JR    Z,GOTIT
22  F028 18DD                   JR    LOOP
23                              END
```

Fig. 9.4. Il linguaggio assembly per la routine che individua una riga.

La coppia di registri HL è caricata con 8000H, l'inizio del BASIC, e la coppia di registri DE con i bytes da F6C0H a F6C1H. Notate la differenza fra un'istruzione del tipo LD HL,8000H e LD DE,(0F6C0H). La prima

pone l'indirizzo di due bytes, 8000H, in HL, per cui H contiene 80H ed L contiene 00H.
L'istruzione LD DE,(0F6C0H) pone in E il byte immagazzinato a F6C1H. Questo significa che vengono letti *due* indirizzi. Alla riga 5, un ciclo inizia, caricando il byte dall'indirizzo F7F8H in A. Questo è il byte basso del numero di riga trasferito dal BASIC, e viene confrontato con il byte contenuto all'indirizzo di HL. Se sono uguali, occorre fare un'altra prova, perché il numero di riga è corretto solo se sono uguali sia il byte basso che il byte alto. Con il passo JR Z,TSTHI si salterà al secondo controllo se il confronto è positivo per il byte basso. Se il confronto è negativo, la riga 8 incrementa HL per il nuovo indirizzo. Il valore di HL viene salvato sullo stack in modo che l'indirizzo di DE possa essere sottratto dall'indirizzo di HL. Questo risultato sarà sempre negativo, a meno che i due indirizzi non siano uguali, o HL non contenga un indirizzo più alto di DE. Un risultato negativo posizionerà a 1 il flag di carry e dopo aver ripristinato il valore originario di HL (riga 11), il passo JR C,LOOP farà ripetere l'azione dalla riga 5.
Notate che è stato controllato il flag di carry invece del flag del segno, poiché il controllo sul flag di carry può essere eseguito con JR, mentre il flag del segno richiede l'uso di JP.
Nel frattempo, se il confronto per il byte basso è stato positivo, il programma si sposta dalla riga 7 alla riga 19. HL viene incrementato, A è caricato da F7F9H (è qui che è memorizzato il byte alto), e avviene il confronto fra i due. Se il confronto è positivo, allora HL contiene l'indirizzo per avere il primo byte della riga, e ciò viene fatto con il salto alla riga 15. Il valore che risulta, contenuto in HL, viene poi ritrasferito agli indirizzi F7F8H e F7F9H. Notate che anche questo è fatto con una sola istruzione, LD(0F7F8H) ,HL. Ora il programma può tornare alla parte BASIC. Che cosa accade se il confronto è stato negativo? In tal caso la riga 22 rimanda al ciclo in modo che il byte basso possa essere controllato al nuovo indirizzo. Il ciclo che va fino alla riga 12 sarà ripetuto finché gli indirizzi di HL e DE sono gli stessi. Se a questo punto il confronto dà ancora esito negativo, il numero di riga non esiste. La riga 13 allora carica zero in HL, e il salto ad EXIT fa in modo che sia caricato zero anche nei due indirizzi usati dalla parte BASIC del programma. Apparirà così il messaggio NON ESISTE QUESTA LINEA nella parte BASIC.
Potreste pensare a come migliorare questo programma. Da una parte, è un po' tortuoso dover continuare a caricare da F7F8H ad ogni ciclo. Poiché non viene usata la coppia di registri BC, non sarebbe più semplice porre all'inizio un'istruzione LD BC,(0F7F8H) e poi prendere i bytes da B o da C quando è necessario? Si può far funzionare il programma usando meno bytes, oppure in modo più veloce. Oppure, potreste trasformarlo in un programma scritto tutto in BASIC, usando POKE per immettere in memoria il codice macchina dalle righe DATA. Quando ho provato que-

sto programma, ho scoperto che gli indirizzi della "RAM riservata" da F970H a FB48H non erano usati, per cui potreste provare ad usarli voi per immettervi il codice macchina, in modo da non aver bisogno di un'istruzione CLEAR. Però,non sorprendetevi se avrete dei guai, perché può darsi che questi indirizzi siano effettivamente riservati per qualcosa. Se è qualcosa che non usate, come un sistema a dischi, allora non c'è problema. Potreste poi provare a scrivere il programma tutto in codice macchina. È abbastanza facile stampare dei messaggi sullo schermo; il problema sorge quando dovete immettere i numeri di riga. Infatti, essi verranno interpretati come codici ASCII e dovranno essere convertiti in numeri binari di due bytes. Analogamente, quando l'indirizzo della riga viene ritrasferito, dovrà essere convertito in codice ASCII. Troverete delle routines per questo tipo di operazione in alcuni testi: non preoccupatevi, però, di capire come funzionano queste routines, a meno che non siate degli esperti in matematica binaria: accontentatevi di usarle e di non doverle creare per conto vostro!

Vorrei spiegare qui la mia preferenza per i salti JR piuttosto che per JP. La ragione sta nel fatto che JR serve a rendere il codice macchina "riallocabile", un programma "riallocabile" è quello i cui bytes di codice macchina possono essere immessi, con la funzione POKE, a qualsiasi indirizzo della RAM, ed il programma funzionerà ugualmente. Ora, se avete delle istruzioni JP o CALL in un programma, che saltano ad indirizzi, o richiamano degli indirizzi *del programma stesso*, il programma non potrà essere riallocato. Per esempio, immaginate di avere JP F050H in un programma posto fra F000H e F0F0H. Se avete fatto iniziare questo programma ad A000H, l'istruzione JP F050H provocherà sempre un salto a F050H, ma questo non è più l'indirizzo corretto (dovrebbe essere A050H). Le istruzioni JR specificano solo un numero di passi, avanti o indietro, perciò, se il vostro programma usa solo i salti JR, e le istruzioni di CALL o JP si riferiscono ad indirizzi della ROM, esso è riallocabile e potete immetterlo in memoria dove preferite. Al contrario, se il programma non è riallocabile, dovrete riassemblarlo col nuovo indirizzo di ORG prima di poter far funzionare il programma.

UNA STRAMBERIA: LE RIGHE TUTTE ZERO

Possiamo utilizzare la nostra conoscenza del metodo per numerare le righe per far diventare zero tutti i numeri di riga di un programma BASIC. In BASIC, ciò richiede parecchio tempo, ma in codice macchina è questione di un attimo, ed è molto facile da programmare. La Fig. 9.5 mostra il metodo, usando il linguaggio assembly. Il listato è stato ottenuto da una stampa con lo ZEN.

```
 1                            ORG    0F9A0H
 2                            LOAD   0F9A0H
 3  F9A0  210180              LD     HL,8001H    ;inizio BASIC
 4  F9A3  5E        BACK:     LD     E,(HL)      ;conserva il prossimo
 5  F9A4  23                  INC    HL          ;indirizzo in
 6  F9A5  56                  LD     D,(HL)      ;HL per il ciclo successivo
 7  F9A6  0E00                LD     C,0         ;azzera C
 8  F9A8  23                  INC    HL          ;preleva byte basso riga
 9  F9A9  71                  LD     (HL),C      ;azzeralo
10  F9AA  23                  INC    HL          ;preleva byte alto
11  F9AB  71                  LD     (HL),C      ;azzeralo
12  F9AC  EB                  EX     DE,HL       ;indirizzo riga successiva
13  F9AD  7C                  LD     A,H         ;controllo
14  F9AE  B5                  OR     L           ;per fine
15  F9AF  20F2                JR     NZ, BACK    ;ripeti se non è
16  F9B1  C9                  RET
17                            END
```

Fig. 9.5. Come rendere ogni numero di riga di un programma BASIC uguale a zero!

Cominciamo caricando l'indirizzo d'inizio del BASIC, 8001H, nella coppia di registri HL. Nel ciclo principale, il byte a questo indirizzo viene caricato in E ed il byte al primo indirizzo più alto viene caricato in D. In questo modo, DE contiene l'indirizzo d'inizio della successiva riga del programma. Il registro C viene azzerato, e il valore zero contenuto in questo registro viene caricato negli altri due indirizzi, che sono gli indirizzi del numero di riga. I numeri contenuti in DE e in HL sono scambiati, per cui HL contiene l'indirizzo d'inizio della riga successiva. Questo indirizzo deve poi essere controllato per vedere se c'è una "riga successiva". Se HL contiene zero, abbiamo raggiunto la fine del programma, altrimenti, si ritorna al ciclo. È semplice e di effetto, provatelo!

LA PORTA VIDEO

Nel Cap. 3 abbiamo visto l'uso del comando VPOKE che ci permetteva di immettere dei bytes nella memoria video. Non importa quale modo di schermo sia usato, VPOKE consiste sempre di un indirizzo e di un byte. Il problema è di eseguire questa azione in codice macchina, perché è l'ultima parte importante del sistema operativo che esamineremo in questo libro. In effetti, non è poi troppo difficile trovare l'equivalente di VPOKE in codice macchina. La soluzione sta nel manuale MSX, che elenca "porte". Una porta (ricordate?) è dove un byte può essere immesso o emesso, e l'indirizzo della porta che ci interessa è 98H. Questo viene individuato nel manuale come Porta della RAM Video, e quindi potremmo avere l'equi-

valente di un'istruzione VPOKE se emettiamo dei bytes verso questa porta. Il linguaggio assembly dello Z80 usa dei comandi speciali per le porte, IN e OUT. Ognuno dei due comandi richiede un indirizzo della porta (fra parentesi) e il nome di un registro. L'effetto di OUT(98H),A, per esempio, è di inviare il byte contenuto nell'accumulatore alla porta numero 98H.
Qualche indagine è sempre utile, ed occorre un breve programma BASIC. Usando il programma della Fig. 9.6 possiamo sapere dove si trovano i bytes D3H 98H nella ROM.

```
10 CLS
20 FOR N%=0 TO 32766
30 IF PEEK(N%)=&HD3 AND PEE
   K(N%+1)=&H98 THEN PRINT N%
40 NEXT
```

Fig. 9.6. Un programma BASIC che ricerca nella ROM i bytes che usano la porta video.

L'importanza di questi due bytes è dovuta al fatto che costituiscono l'istruzione OUT(98H),A. Ciò significa "fai uscire il byte contenuto nell'accumulatore verso la porta numero 98H", e poiché la porta 98H è la porta video, questo comando può essere usato per ottenere l'equivalente di VPOKE e VPEEK. Il programma ci darà degli indirizzi, e possiamo disassemblare il codice macchina attorno a questi indirizzi con il comando "u" dello ZEN. Come al solito, chi utilizza lo ZEN è facilitato! Non vi mostrerò ogni passo di questo gioco investigativo (il gioco più interessante che possiate fare con qualsiasi computer), perché ritengo che vi divertiate a farlo per conto vostro. Il sommario è comunque importante. Se vogliamo eseguire l'azione VPOKE, dobbiamo porre l'indirizzo della parte di memoria video che vogliamo usare in HL e poi richiamare la subroutine 07DFH. Questa emetterà l'indirizzo in due bytes verso la porta 99H. Ora ci occorrono delle istruzioni fra questo comando e un qualsiasi uso di OUT(98H) ,A. Bisogna infatti dare alla porta il tempo di agire, il codice macchina della ROM esegue due caricamenti fittizi per perdere un po' di tempo qui. Dopo aver selezionato l'indirizzo, possiamo caricare l'accumulatore con qualsiasi byte che vogliamo inviare alla memoria video, e poi possiamo usare OUT(98H) ,A per inviarlo. Se usiamo il comando OUT in un ciclo, il sistema operativo prenderà automaticamente un nuovo spazio di schermo per ogni byte. La Fig. 9.7 mostra un semplice programma che riempirà lo schermo con la lettera A. L'abbiamo già fatto in BASIC ed anche in codice macchina con una CALL alla ROM. Sarà interessante confrontare i tempi che avevamo annotato con il tempo impiegato da questo programma.

```
 1                              ORG   0F000H
 2                              LOAD  0F000H
 3 F000  210000                 LD    HL,0000H
 4 F003  CDDF07                 CALL  07DFH
 5 F006  E5                     PUSH  HL
 6 F007  E1                     POP   HL
 7 F008  01E803                 LD    BC,03E8H
 8 F00B  3E41       LOOP:       LD    A, 41H
 9 F00D  D398                   OUT   (98H),A
10 F00F  0B                     DEC   BC
11 F010  78                     LD    A,B
12 F011  B1                     OR    C
13 F012  20F7                   JR    NZ,LOOP
14 F014  C303A0                 JP    0A003H
15                              END
```

Fig. 9.7. Un programma che riempie lo schermo con la lettera "A", usando la porta video.

Le righe 3 e 4 caricano HL con zero, e chiamano 07DFH. Questo indicherà 000H come indirizzo d'inizio nella RAM video. Le istruzioni PUSH HL e POP HL servono a perdere un po' di tempo, in modo che la porta venga inizializzata prima delle successive azioni. Carichiamo poi BC con 03E8H, il più alto numero di bytes in uno schermo in Modo 0, ed iniziamo un ciclo. Nel ciclo, carichiamo A con 41H (ASCII "A") e lo inviamo alla RAM video con OUT(98H) ,A. Usiamo, come al solito, BC come contatore, il che significa rimettere il byte in A. In questo semplice esempio, è più pratico ricaricare ogni volta piuttosto che usare PUSH AF e POP AF. Il programma continua il ciclo finché BC non ha raggiunto zero, e l'ultima istruzione, JP 0A003H, ritorna allo ZEN. Se usate questo programma dal BASIC, mettete RET qui. Quando lo assemblerete e lo farete "girare", vedrete che riempirà lo schermo ancor più velocemente di quanto farebbe usando la routine CALL 0A2H in un ciclo. Se immettete un numero diverso in HL, potete variare la posizione in cui lo schermo comincia a riempirsi. Notate che la dimensione dello schermo è la larghezza di 40 caratteri, invece di quella di 37 caratteri normalmente usata.

ED ORA, UN PO' DI MOVIMENTO!

Riempire uno schermo con un carattere non è di grande utilità per molti programmi; vedremo ora come far muovere un oggetto sullo schermo in modo di scrittura. Ho scelto il modo di scrittura per due ragioni: una, perché è molto più facile lavorare con questo piuttosto che con gli schermi in modo grafico; due, perché gli schermi in modo grafico possono tranquillamente utilizzare gli sprite che sono controllati da comandi BASIC.

Dal momento che non è prevista l'utilizzazione degli sprite con lo schermo in Modo 0, sembra una buona idea aggiungere quest'azione a quelle del computer MSX. Cominceremo prima a vedere come far muovere un oggetto sullo schermo.
Lo schema è abbastanza semplice. Scegliamo un carattere, in questo esempio la forma del codice grafico 2 (*non* è uguale al codice ASCII 2). È la forma del "volto sorridente", ma ricordate che queste forme sono controllate da numeri memorizzati nella RAM, e se volete potete cambiarle, usando una griglia 7×8. Per spostare il volto attraverso lo schermo su ogni riga dobbiamo stamparlo alla prima posizione dello schermo, e poi aspettare. Dobbiamo poi cancellare il volto, scegliere la nuova posizione e ripetere finché non abbiamo raggiunto l'ultima posizione dello schermo. Una piccola complicazione sorge se usiamo CALL 07DFH per ottenere l'indirizzo del volto, perché dovremo richiamare questa routine con lo stesso numero in HL per stampare uno spazio. Se non usiamo più l'indi-

```
 1                        ORG   0F000H
 2                        LOAD  0F000H
 3 F000 210000            LD    HL,0000
 4 F003 CDDF07            CALL  07DFH
 5 F006 E5                PUSH  HL
 6 F007 01E803            LD    BC,3E8H
 7 F00A 3E02     LOOP:    LD    A,02H
 8 F00C D398              OUT   (98H),A
 9 F00E 11FF1F            LD    DE,01FFFH
10 F011 1B       DELY:    DEC   DE
11 F012 7A                LD    A,D
12 F013 B3                OR    E
13 F014 20FB              JR    NZ,DELY
14 F016 E1                POP   HL
15 F017 CDDF07            CALL  07DFH
16 F01A 23                INC   HL
17 F01B E5                PUSH  HL
18 F01C 3E20              LD    A,32
19 F01E D398              OUT   (98H),A
20 F020 0B                DEC   BC
21 F021 78                LD    A,B
22 F022 B1                OR    C
23 F023 20E5              JR    NZ,LOOP
24 F025 E1                POP   HL
25 F026 C303A0            JP    0A003H
26                        END
```

Fig. 9.8. L'animazione di un carattere sullo schermo. Questo implica l'usare due volte ogni indirizzo.

rizzo, l'indirizzo dello schermo verrà automaticamente incrementato, e lo spazio sarà stampato nello spazio *successivo,* anziché andare a sostituire il volto. Al lavoro, dunque, con la Fig. 9.8.
La prima azione è di immettere in HL l'indirizzo di inizio della RAM video. Alla riga 4, la chiamata a 07DFH emette questo indirizzo verso la porta, e il contenuto di HL viene messo sullo stack. La coppia di registri BC viene caricata con 03E8H in modo da poter essere usata in un conteggio alla rovescia per l'intero schermo. La riga 7 inizia il ciclo che renderà possibile l'animazione. L'accumulatore è caricato con il byte 2H, e questo viene inviato verso la porta per essere immesso sullo schermo. La coppia di registri DE viene poi usata in un conteggio di ritardo, usando 1FFFH per il conteggio: esso controllerà la velocità del movimento. Dopo il ritardo, il contenuto di HL viene prelevato dallo stack. Non era necessario salvare HL sullo stack, ma, come ricorderete, ci occorreva un po' di tempo fra CALL 07DFH e OUT(98H) ,A, e PUSH HL, insieme al caricamento di BC, serve egregiamente a questo scopo. Dopo aver ripristinato l'indirizzo di HL, viene di nuovo richiamata la routine che si trova a 07DFH per ottenere la stessa posizione di schermo. HL viene incrementato e di nuovo messo sullo stack, ed A è caricato con lo spazio — 32 in codice ASCII. Questo viene poi inviato alla porta per cancellare il volto. BC viene decrementato e controllato per vedere se abbiamo raggiunto l'ultimo indirizzo di schermo. In caso contrario, il ciclo ricomincia. HL viene poi incrementato e l'indirizzo corretto di HL è sullo stack. Quando il ciclo termina, HL sarà ancora sullo stack, e *lo stack deve essere azzerato prima di ritornare al BASIC.* Se ve ne dimenticate, sarà un metodo più che sicuro per far fallire il vostro programma! Il programma, in questo caso termina con un'istruzione di JP all'indirizzo di "inizio a caldo" dello ZEN, 0A003H, ma se lo avete richiamato dal BASIC, dovrete sostituirvi RET.
In azione, questo programma fa muovere ad una discreta velocità il volto. L'animazione è ben lungi dall'essere perfetta per vari motivi. Uno è che ci dovrebbe essere un ritardo fra la cancellazione di un volto e la stampa di un altro e questo si può facilmente aggiungere. Un problema più complicato sta nel fatto che il movimento avviene a scatti, perché il volto si sposta di una rilevante distanza, un quarantesimo dell'ampiezza dello schermo, ad ogni passo e questo è inevitabile, usando lo schermo in modo di scrittura. Un movimento più omogeneo potrebbe essere ottenuto con un metodo elaborato che comporta lo scorrimento a destra dei bit dell'indirizzo del carattere, e la stampa su due indirizzi di schermo ogni volta, ma è molto più sensato usare lo schermo in modo grafico se volete fare cose del genere. Con questa esperienza, possiamo però cercare di stendere un programma di sprite per lo schermo in modo di scrittura. Faremo un programma che possa essere usato anche dal BASIC, immettendo i numeri in memoria con POKE.

LO SPRITE CON LO SCHERMO IN MODO DI SCRITTURA

Non si possono creare sprite con lo schermo in modo di scrittura. Questa affermazione è una vera sfida per il programmatore in codice macchina, e vedremo ora un primo tentativo di creare questo sprite. La Fig. 9.9 illustra come dovrebbe essere un programma BASIC per creare questo tipo di sprite.

```
10 DEFUSR1=&HF000
20 P1!=&HF02D:CH1!=&HF02F:S1!=&HF030
30 POKE CH1!,2
40 POKE S1!,0:POKE S1!+1,8
50 FOR N%=0 TO 1000:H%=N%/256:L%=N%—256*H%
60 POKE P1!,L%:POKE P1!+1,H%
70 A=USR1(0)
80 NEXT
```

Fig. 9.9. Un programma BASIC che usa la funzione POKE per controllare uno sprite. I bytes della forma, della posizione e della velocità possono essere immessi in memoria sempre con POKE.

La prima riga stabilisce l'indirizzo di USR che preleverà la routine in codice macchina, e la riga 20 definisce i parametri. P1! è l'indirizzo per il byte basso della posizione dello sprite, e P1!+1 è l'indirizzo del bite alto. CH1! è l'indirizzo del carattere che verrà usato come sprite, un qualsiasi carattere compreso tra 1 e 255. S1! è l'indirizzo per il byte basso della velocità, S1!+1 è l'indirizzo del byte alto. Il programma immetterà dei numeri in memoria a questi indirizzi, con POKE, per specificare la posizione dello sprite, la forma e velocità di movimento. Al codice macchina viene lasciato il compito di far apparire lo sprite. Il ciclo che fa muovere lo sprite è perciò in BASIC invece che in codice macchina. Il linguaggio assembly per il codice macchina è riportato nella Fig. 9.10.
Gli indirizzi immessi in memoria dal BASIC sono alla fine del programma, ed hanno, come label, gli stessi nomi usati in BASIC. L'uso di DW e DB è nuovo, per voi. Queste non sono mnemoniche, solo delle istruzioni rivolte all'assemblatore ZEN. DW significa "fai una parola", ed assegnerà due bytes ad un numero. In questo caso il numero è zero, e quindi ambedue i bytes sono zero. DB significa "assegna un byte" ad un numero e, anche qui, ho usato zero. Questa parte del codice macchina non fa altro che azzerare questi spazi di memoria *quando il programma viene assemblato.* Non ha effetto su questi indirizzi quando il programma viene eseguito.

In dettaglio, la riga 3 carica il contenuto di P1, P1+1 in HL. La parte BASIC del programma avrà già immesso in queste locazioni i due bytes di

```
 1                                ORG   0F000H
 2                                LOAD  0F000H
 3  F000 2A2EF0                   LD    HL,(P1)
 4  F003 CDEC07                   CALL  07ECH
 5  F006 E5                       PUSH  HL
 6  F007 E1                       POP   HL
 7  F008 DB98                     IN    A,(98H)
 8  F00A 4F                       LD    C,A
 9  F00B CDDF07                   CALL  07DFH
10  F00E E5                       PUSH  HL
11  F00F E1                       POP   HL
12  F010 3A30F0                   LD    A,(CH1)
13  F013 D398                     OUT   (98H),A
14  F015 CD24F0                   CALL  DELY
15  F018 CDDF07                   CALL  07DFH
16  F01B E5                       PUSH  HL
17  F01C E1                       POP   HL
18  F01D 79                       LD    A,C
19  F01E D398                     OUT   (98H),A
20  F020 CD24F0                   CALL  DELY
21  F023 C9                       RET
22  F024 ED5B31F0   DELY:         LD    DE,(S1)
23  F028 1B         BACK:         DEC   DE
24  F029 7A                       LD    A,D
25  F02A B3                       OR    E
26  F02B 20FB                     JR    NZ,BACK
27  F02D C9                       RET
28  F02E 0000       P1:           DW    0
29  F030 00         CH1:          DB    0
30  F031 0000       S1:           DW    0
31                                END
```

Fig. 9.10. La parte in linguaggio assembly della routine per lo sprite. Essa vi permette di avere uno sprite con lo schermo in Modo 0!

un indirizzo di schermo, e questi bytes ora saranno in HL. L'istruzione CALL07ECH non era stata usata finora. È il modo di inviare un indirizzo alla porta video come preparazione per la *lettura* della porta, l'equivalente di un'operazione VPEEK. PUSH HL e POP HL servono, come al solito, a perdere un po' di tempo, e poi la porta viene letta con IN(98H) ,A. Viene così letto un qualsiasi carattere presente sullo schermo a questa posizione,

e la riga 8 trasferisce poi questo carattere al registro C per usarlo in seguito. In questo modo lo sprite si sposta attraverso lo schermo senza cancellare i caratteri, almeno apparentemente, mentre si muove. Per dirla in un'altra maniera, sembrerà che passi davanti ai caratteri.
Il passo successivo è di chiamare la routine a 07DFH, per poter inviare dati allo schermo, alla riga 9. Questa volta, caricheremo l'accumulatore dall'indirizzo CH1, che è stato usato dal BASIC per memorizzare il numero di codice di un carattere. L'istruzione OUT(98H) ,A alla riga 13 posizionerà il carattere sullo schermo e la posizione è indicata dal numero in HL. Una volta posizionato il carattere, viene richiamata una subroutine di ritardo, DELY, che è stata posta in un punto qualsiasi del programma. Dopo l'esecuzione di questa subroutine, deve essere reimmesso il carattere originario, e questo viene fatto prendendo ancora un indirizzo immesso con POKE, trasferendo il byte contenuto nel registro C ancora nel registro A e usando un'altra istruzione diretta alla porta 98H alla riga 19. Segue poi un'altra subroutine di ritardo, ma questa può essere omessa, perché anche la parte BASIC del programma farà effettuare un ritardo a questo punto. Il programma ritorna poi al BASIC per prelevare l'altro gruppo di bytes agli indirizzi P1, CH1 e S1. È uno sprite molto semplice, ma i migliori computer del passato hanno usato degli schemi altrettanto semplici! Ciò che ho voluto dimostrare è come programmare uno sprite con uno schermo in modo di scrittura, un procedimento che in BASIC si può fare solo con VPEEK e VPOKE.

COME STAMPARE CIO' CHE COMPARE SULLO SCHERMO

Terminerò con una routine che, lo ammetto, sarà inutile per molti lettori. In compenso sarà molto apprezzata da altri e, in ogni caso, vi illustrerà ancora delle altre possibilità del codice macchina, anche se non la userete. Il programma intende permettervi di inviare ogni normale carattere che si trova sullo schermo ad una stampante. È particolarmente utile con lo ZEN, perché lo ZEN non prevede la stampa del file di testo, solo del linguaggio assembly. Con questa routine, sarà possibile stampare tutti i caratteri che usano i codici ASCII da 32 a 128 (decimali). Il problema principale sta nel fatto che le stampanti di solito usano 80 o più caratteri per riga, e lo schermo usa un massimo di 40 caratteri per riga. Potete predisporre molte stampanti, in particolare le Epson, in modo che stampino solo 40 caratteri per riga, e in questa maniera il semplicissimo programma della Fig. 9.11 sarà perfettamente adeguato.
Viene ancora usato l'equivalente di VPEEK in codice macchina per leggere i bytes dallo schermo, filtrarli, e poi trasmetterli alla stampante. Ora "filtrare" è un termine nuovo, e vorrei spiegarlo. Può darsi che ci sia un codice di carattere in memoria che faccia impazzire la stampante. I codici inferiori a 32, per esempio, hanno un significato del tutto diverso per le

stampanti, ed anche i codici superiori a 127 possono avere degli effetti strani. Che cosa otterrete, dipende da quale stampante usate, ma di solito non sarà certo la stampa di quello che vedete sullo schermo! Un "filtro" è

```
 1                             ORG   0F000H
 2                             LOAD  0F000H
 3 F000 210000                 LD    HL,0000
 4 F003 01E803                 LD    BC,03E8H
 5 F006 CDEC07     LOOP:       CALL  07ECH
 6 F009 E5                     PUSH  HL
 7 F00A E1                     POP   HL
 8 F00B DB98                   IN    A,(98H)
 9 F00D E67F                   AND   7FH
10 F00F FE20                   CP    20H
11 F011 3803                   JR    C,NEXT
12 F013 CDA500                 CALL  0A5H
13 F016 23         NEXT:       INC   HL
14 F017 0B                     DEC   BC
15 F018 78                     LD    A,B
16 F019 B1                     OR    C
17 F01A 20EA                   JR    NZ,LOOP
18 F01C C9                     RET
19                             END
```

Fig. 9.11. Una routine per la stampa del contenuto dello schermo, utile se la vostra stampante prevede la stampa di 40 caratteri per riga.

una parte di programma che elimina questi caratteri indesiderati. Ci sono due tipi principali che vogliamo eliminare. Il primo tipo è un qualsiasi carattere che abbia il bit 7 posto a 1. Esso corrisponde ad un numero uguale o maggiore di 128 (decimale). L'altro tipo è un qualsiasi carattere con un codice inferiore a 32. Possiamo azzerare il bit 7 prendendo il codice del carattere contenuto nell'accumulatore e poi eseguendo AND 7F (scrivete 7F in forma binaria, e vedrete perché). Questo convertirà i codici compresi tra 80H e FFH in caratteri stampabili. Si può anche usare CP 20H, effettuando un salto JR C per evitare il passo della stampa se viene usato un codice inferiore a 20H.

La Fig. 9.12 carica l'inizio della memoria video (schermo in Modo 0) in HL, usa BC come contatore e inizia un ciclo. Il ciclo richiama la routine VPEEK, e legge poi il byte che si trova a questo indirizzo. Questo byte viene "filtrato", e se è 32H o maggiore di 32H, viene inviato alla stampante con CALL 0A5H, l'indirizzo della routine di stampa. Alla label NEXT, l'indirizzo di HL viene incrementato, il contatore è decrementato e controllato prima che il ciclo ricominci. Quando il programma è assemblato, richiede solo 29 bytes e praticamente può essere immesso in una qualsiasi

locazione di memoria disponibile. Se la vostra stampante è una Epson (serie MX o RX), potete ritornare al BASIC e digitare:

LPRINT CHR$(27) ;"Q" ;CHR$(40)

perché la stampante stampi solo 40 caratteri per riga. Poi dovete assegnare l'indirizzo per DEFUSR e, da qui in poi, dare l'appropriato comando USR che riprodurrà lo schermo sulla stampante, come dimostra la Fig. 9.12, stampata durante lo sviluppo di questo programma.

```
 1 ORG 0F000H
 2 LOAD 0F000H
 3 LD HL,0000
 4 LD BC,03E8H
 5 LOOP:CALL 07ECH
 6 PUSH HL
 7 POP HL
 8 IN A,(98H)
 9 AND 7FH
10 CP 20H
11 JR C,NEXT
12 CALL 0A5H
13 NEXT:INC HL
14 DEC BC
15 LD A,B
16 OR C
17 JR NZ,LOOP
18 RET
19 END
```

Fig. 9.12. Un esempio dell'output per stampare quello che compare sullo schermo.

Non è proprio così semplice se volete usare una stampante che *non* permette di usare 40 caratteri per riga. In questo caso, il codice macchina deve inviare i caratteri di salto riga (0AH) e ritorno carrello (0CH) alla stampante dopo 40 caratteri. Come si può fare? La risposta più semplice è di usare in modo più raffinato il contatore BC. Possiamo mettere in B il numero dei caratteri della riga, e in C il numero di righe per schermo. In tal modo possiamo usare DJNZ per il conteggio dei caratteri, poi inviare i codici del ritorno carrello e salto riga, decrementare C, ripristinare B e ricominciare il ciclo. Il programma completo in linguaggio assembly è mostrato nella fig. 9.13.
All'inizio, si attiene alla vecchia routine, tranne che per il caricamento di BC con 2818H. Questo si suddivide in 28H (40 decimale) per B e 18H (24

decimale) per C, perché ora usiamo B e C come *contatori separati.* La routine dalla riga 5 alla 13 è identica a quella precedente, ma DJNZ

```
 1                          ORG   0F000H
 2                          LOAD  0F000H
 3 F000 210000              LD    HL,0000
 4 F003 011828              LD    BC,2818H
 5 F006 CDEC07    LOOP:     CALL  07ECH
 6 F009 E5                  PUSH  HL
 7 F00A E1                  POP   HL
 8 F00B DB98                IN    A,(98)
 9 F00D E67F                AND   7FH
10 F00F FE20                CP    20H
11 F011 3803                JR    C,NEXT
12 F013 CDA500              CALL  0A5H
13 F016 23        NEXT:     INC   HL
14 F017 10ED                DJNZ  LOOP
15 F019 3E0A                LD    A,0AH
16 F01B CDA500              CALL  0A5H
17 F01E 3E0D                LD    A, 0DH
18 F020 CDA500              CALL  0A5H
19 F023 0628                LD    B,28H
20 F025 0D                  DEC   C
21 F026 20DE                JR    NZ,LOOP
22 F028 C9                  RET
23                          END
```

Fig. 9.13. Una routine modificata per la stampa del contenuto dello schermo che può essere usata con una qualsiasi stampante a 80 colonne.

LOOP della riga 14 interrompe il ciclo dopo aver inviato 40 caratteri alla stampante. Le righe dalla 15 alla 18 inviano i caratteri di salto riga e ritorno carrello, dopo di che il registro B deve essere ripristinato alla riga 19. Il registro C viene poi decrementato e controllato per vedere se abbiamo terminato l'ultima riga. In caso contrario, il programma ripete dalla label LOOP.

GLI "AGGANCI"

Questo è veramente l'ultimo argomento, ma è assai importante. Finora, avete sempre iniziato l'esecuzione di un programma in codice macchina con l'istruzione USR, che poteva essere assegnata anche ad uno dei tasti funzione. In quasi tutti i casi, questo è assai soddisfacente, ma chi utilizza

una stampante, avrà notato un intoppo nel programma precedente. Infatti, se assegnate un'istruzione USR, del tipo A=USR1(0) ad un tasto funzione, quando verrà usato questo tasto, l'istruzione A=USR1(0) verrà stampata sullo schermo e, se inviate alla stampante quanto si trova sullo schermo, verrà inviata anche questa istruzione. Inoltre, ci sarà uno scorrimento (scroll) dello schermo, e potreste perdere quello che c'era in cima allo schermo stesso.
Termineremo, perciò, con un metodo automatico per iniziare l'esecuzione di un programma. Questo vi porta veramente addentro al sistema operativo e quando cominciate ad usare questi metodi di programmazione, siete decisamente ad un buon punto della vostra carriera di programmatori in codice macchina! Ci basiamo sul principio che il computer MSX è stato progettato per essere adattabile, per cui molte routines della ROM fanno delle chiamate alla RAM. Quando dico "fanno delle chiamate", lo intendo in senso letterale. Infatti, in molti punti della ROM troverete istruzioni come questa:

```
CALL 0FDAEH
CALL 0B9FH
RET NC
```

Una delle chiamate è rivolta ad un indirizzo che si trova nella più alta parte finale della RAM, da "FD" in poi. Vi è un insieme completo di questi indirizzi, che comincia da FD9AH e termina a FFC9H, e se li esaminate, vedrete che ognuno contiene lo stesso byte, C9H. Questo è il codice d'istruzione per RET, per cui, ogni volta che la ROM fa una chiamata a questa parte della RAM, tornerà immediatamente alla ROM. Che azione esegue? Risposta: nessuna!
La domanda potrebbe essere meglio formulata: "che azione *potrebbe* eseguire?". Poiché questi indirizzi che contengono i bytes C9H sono nella RAM, essi possono essere modificati. Invece di C9 C9 C9 potremmo mettere, per esempio, CD 00 F0. Questo è il codice macchina per CALL F000H e l'effetto sarà di inviare a F000H ogni chiamata, anziché tornare alla ROM. Questi indirizzi, quindi, permettono di "agganciare" delle nostre personali routines, in modo che vengano eseguite automaticamente ogni volta che la ROM fa una chiamata alla RAM. Dobbiamo poi vedere come vengono fatte queste chiamate di "aggancio". Ho scritto un breve programma BASIC che individua ogni chiamata agli indirizzi FD per trovarne qualcuna. Molte non sono di grande interesse, se non per usi specialistici, ma alcune sono invece assai utili. Per ogni chiamata dalla ROM vengono assegnati 5 bytes della RAM e sono riempiti con C9H. La presenza di 5 bytes ci permette di inserire i codici per le istruzioni JP, o CALL, come preferiamo, purché lo si faccia con attenzione. La Fig. 9.14

Indirizzo	Effetto su
FDDBH	tasto RETURN
FDE5H	Numerazione AUTO
FDAEH	Cancella cursore
FDB3H	Visualizzazione tasti funzione
FD9FH	Interruzione
FDD1H	Qualsiasi tasto
FDC2H	Prelevamento carattere

Fig. 9.14. Alcuni degli indirizzi di "aggancio" (HOOK) che possono essere usati.

riporta alcune di queste chiamate di "aggancio" e illustra come vengono usate. Per dimostrarne l'azione, ne esamineremo una, la chiamata a FDD1H.
La chiamata a FDD1H viene effettuata ogni volta che un tasto è premuto, per cui, se modifichiamo il byte di codice macchina contenuto a questo indirizzo, possiamo far eseguire una nostra routine ogni volta che viene premuto un tasto. Non è proprio così semplice come sembra, però, perché il codice contenuto nell'accumulatore, quando si verifica questa chiamata, *non* è un codice ASCII. Tuttavia, questo non ha importanza per l'uso che voglio descrivere, perché il punto principale è che la pressione del tasto ESC dà il codice 3AH (58 decimale). Se creiamo una routine che controlli questo byte, sarà allora possibile escludere gli altri tasti e far eseguire la routine solo quando viene premuto il tasto ESC. Ho scelto ESC, perché normalmente non viene usato per nient'altro.
Che cosa dobbiamo fare, allora? Abbiamo bisogno di un programma in linguaggio assembly che, prima di tutto immetterà un byte per l'istruzione CALL (CDH) al primo indirizzo dell'"aggancio", poi l'indirizzo della routine che vogliamo usare. Dopo aver caricato questi numeri, possiamo tornare al BASIC. All'indirizzo d'inizio della nostra routine, possiamo mettere un filtro. Questo, per esempio, potrebbe controllare uno specifico byte e ritornare se il byte non fosse nell'accumulatore. Il resto della routine consisterà di istruzioni che dovranno essere eseguite se il byte indicato *è* presente. Nel nostro esempio, il byte prescelto è 3AH, il byte che è presente nell'accumulatore quando viene premuto il tasto ESC, la routine terminerà poi con RET, per cui il successivo byte C9H della RAM rimanderà il programma alla ROM. Avremmo potuto usare anche un byte di JP, anziché di CALL, e l'effetto sarebbe stato lo stesso. La Fig. 9.15 mostra la routine di stampa del contenuto dello schermo realizzata con questo metodo.
La riga 3 è molto importante. DI significa "disabilita le interruzioni", per cui niente interromperà il programma quando vengono cambiati gli indi-

rizzi. Questo per evitare la catastrofe se qualcosa dovesse interrompere il programma, lasciando gli indirizzi cambiati per metà e si avrebbe, ovviamente, un immediato fallimento del programma. Non sono del tutto convinto che questo passo sia strettamente necessario per il computer MSX, ma, dal momento che lo è per altri computer, è meglio agire sul sicuro! L'accumulatore viene poi caricato con CDH, il byte dell'istruzione CALL, e questo è caricato all'indirizzo FDD1H. Questo è un indirizzo che viene richiamato ogni volta che viene premuto un tasto. Le righe 6 e 7

```
 1                            ORG   0F000H
 2                            LOAD  0F000H
 3 F000  F3                   DI
 4 F001  3ECD                 LD    A,0CDH
 5 F003  32D1FD               LD    (0FDD1H),A
 6 F006  210EF0               LD    HL,STRT
 7 F009  22D2FD               LD    (0FDD2H),HL
 8 F00C  FB                   EI
 9 F00D  C9                   RET
10 F00E  FE3A     STRT:       CP    58
11 F010  C0                   RET   NZ
12 F011  E5                   PUSH  HL
13 F012  C5                   PUSH  BC
14 F013  F5                   PUSH  AF
15 F014  210000               LD    HL,0000
16 F017  011828               LD    BC,2818H
17 F01A  CDEC07   LOOP:       CALL  07ECH
18 F01D  E5                   PUSH  HL
19 F01E  E1                   POP   HL
20 F01F  DB98                 IN    A,(98H)
21 F021  E67F                 AND   7FH
22 F023  FE20                 CP    20H
23 F025  3803                 JR    C,NEXT
24 F027  CDA500               CALL  0A5H
25 F02A  23       NEXT:       INC   HL
26 F02B  10ED                 DJNZ  LOOP
27 F02D  3E0A                 LD    A,0AH
28 F02F  CDA500               CALL  0A5H
29 F032  3E0D                 LD    A,0DH
30 F034  CDA500               CALL  0A5H
31 F037  0628                 LD    B,28H
32 F039  0D                   DEC   C
33 F03A  20DE                 JR    NZ,LOOP
34 F03C  F1                   POP   AF
```

```
35 F03D C1          POP   BC
36 F03E E1          POP   HL
37 F03F C9          RET
38                  END
```

Fig. 9.15. La chiamata ad una routine con il tasto ESC. In questo esempio, la routine è quella della stampa del contenuto dello schermo.

prelevano l'indirizzo d'inizio della routine di stampa della "videata" e lo pongono dopo il byte CDH. Una volta fatto questo, le interruzioni possono essere riabilitate e la routine ritorna al BASIC. Quando verrà richiamato, solo una volta, da un'istruzione USR in BASIC, cambierà l'indirizzo in modo che la routine che si trova all'indirizzo STRT venga erseguita ogni qualvolta viene premuto un tasto. Ora dobbiamo sistemare la routine di stampa in modo che sia eseguita solo quando viene premuto il tasto ESC, e dobbiamo anche proteggere i registri. Non sappiamo quali registri saranno usati dalla macchina quando la routine della pressione del tasto richiama la RAM. Per evitare ogni problema, è meglio salvare sullo stack ogni registro che viene utilizzato dalla routine di stampa.
Questa routine inizia (riga 10) con CP 58. Questa istruzione fa un'indagine sul tasto ESC, e se il codice non è presente, l'istruzione RET NZ (ritorna se non zero) effettuerà il ritorno alla ROM. Se il byte è presente, i passi successivi saranno di salvare HL, BC e AF sullo stack. In questo modo saremo sicuri che nessuno di questi registri sia modificato dalla routine di stampa. Le righe dalla 15 alla 33 seguono la routine di stampa della "videata" che abbiamo visto in precedenza. Ecco dove potete immettere la vostra routine, che volete far eseguire premendo il tasto ESC. Dopo che la routine è stata eseguita, i registri sono ripristinati e l'istruzione RET rimanda lo Z80 alla ROM. La routine di stampa della "videata" è ora pronta per l'"aggancio". Se avete usato lo ZEN per l'assemblaggio, ritornate al BASIC e usate DEFUSR e USR pe far eseguire la prima parte del programma. Questa si aggancerà alla seconda parte del programma, ponendo il suo indirizzo a FDD2H e FDD3H. La seconda parte del programma ora verrà eseguita ogni volta che viene premuto il tasto ESC. Questo accadrà sia che usiate il BASIC o che usiate lo ZEN, o qualsiasi altro linguaggio, finché gli indirizzi da FDD1H a FDD3H non vengano modificati da qualcos'altro. Come risultato, potrete usare normalmente il vostro computer, ma la stampante si avvierà automaticamente premendo ESC e il testo sullo schermo sarà copiato sulla stampante. È molto utile, perché non avete bisogno di programmare tutta una serie di istruzioni PRINT e LPRINT se volete un output sia sullo schermo che sulla stampante. Dal momento che questa routine è richiamata dalla pressione di ESC, che non fa apparire niente sullo schermo, avrete una stampa del testo sullo schermo così come esso si presenta, senza che sia

modificato da un'istruzione USR o da uno scroll. Non avete una stampante? Se fate seriamente della programmazione, l'avrete presto, ma nel frattempo questo metodo può essere ugualmente utile. Per esempio, potreste stampare un carattere con il tasto ESC, per cui dovrete sostituire una routine di stampa di un carattere al posto di quella di stampa di una videata. Potreste volere l'emissione di un suono quando viene premuto un tasto qualsiasi: eliminate quindi la parte relativa al tasto ESC e immettete una routine del suono nel programma.
Potete fare in modo che il tasto ESC ripulisca lo schermo, lo riempia con un carattere, stampi un insieme di caratteri — insomma, tutto quello che può essere programmato in codice macchina. Ecco di che cosa si occupa la programmazione in codice macchina.
Siamo arrivati alla fine del nostro cammino. È la fine dell'inizio, perché ora dovreste avere un'idea di come si scrive un programma in codice macchina e che cosa si può ottenere. Ma, ed è ancora più importante, ora conoscete molto di più il funzionamento del vostro computer MSX e sapete come scoprirne degli altri segreti. I vostri prossimi passi dipendono dai vostri interessi. Molti libri sulla programmazione in codice macchina sono disponibili in commercio e, benché siano assai più complessi di questo libro, ora sono alla vostra portata.

APPENDICE A

ELENCO DELLE PAROLE CHIAVE,
DEGLI INDIRIZZI E DEI BYTES DI SPOSTAMENTO

14968	246	BS	15132	209	EF
14971	6	TN	15135	151	SKI$
14974	14	SC	15140	234	SKF
14977	21	TTR$	15144	38	RAW
14982	233	ASE	15148	190	LSE
14987	201	SAVE	15153	161	ND
14992	208	LOAD	15156	129	RASE
14997	207	EEP	15161	165	RROR
15001	192	IN$	15166	166	RL
15005	29	ALL	15169	225	RR
15010	202	LOSE	15172	226	XP
15015	180	OPY	15175	11	OF
15019	214	ONT	15178	43	QV
15023	153	LEAR	15181	249	OR
15028	146	LOAD	15185	130	IELD
15033	155	SAVE	15190	177	ILES
15038	154	SRLIN	15195	183	N
15044	232	INT	15197	222	RE
15048	30	SNG	15200	15	IX
15052	31	DBL	15203	33	POS
15056	32	VI	15207	39	OTO
15059	40	VS	15212	137	O TO
15062	41	VD	15217	137	OSUB
15065	42	OS	15222	141	ET
15068	12	HR$	15225	178	EX$
15072	22	IRCLE	15230	27	NPUT
15078	188	OLOR	15236	133	F
15083	189	LS	15238	139	NSTR
15086	159	MD	15243	229	NT
15089	215	ELETE	15246	5	NP
15096	168	ATA	15249	16	MP
15100	132	IM	15252	250	NKEY$
15103	134	EFSTR	15258	236	PL
15109	171	EFINT	15261	213	ILL
15115	172	EFSNG	15267	212	EY
15121	173	EFDBL	15270	204	PRINT
15127	174	SKO$	15277	157	LIST

15282	158	POS
15286	28	ET
15289	136	OCATE
15295	216	INE
15299	175	OAD
15303	181	SET
15307	184	IST
15311	147	FILES
15317	187	OG
15320	10	OC
15323	44	EN
15326	18	EFT$
15331	1	OF
15334	45	OTOR
15340	206	ERGE
15345	182	OD
15348	251	KI$
15352	46	KS$
15356	47	KD$
15360	48	ID$
15364	3	AX
15367	205	EXT
15372	131	AME
15376	211	EW
15379	148	OT
15382	224	PEN
15387	176	UT
15390	156	N
15392	149	R
15394	247	CT$
15398	26	FF
15401	235	RINT
15407	145	UT
15410	179	OKE
15414	152	OS
15417	17	EEK
15421	23	SET
15425	194	RESET
15431	195	OINT
15436	237	AINT
15441	191	DL
15444	36	AD
15447	37	LAY
15451	193	RETURN
15459	142	EAD
15463	135	UN
15466	138	ESTORE
15473	140	EM
15476	143	ESUME
15482	167	SET
15486	185	IGHT$
15492	2	ND
15495	8	ENUM
15500	170	CREEN
15507	197	PRITE
15513	199	TOP
15517	144	WAP
15521	164	ET
15524	210	AVE
15528	186	PC(
15532	223	TEP
15536	220	GN
15539	4	QR
15542	7	IN
15545	9	TR$
15549	19	TRING$
15556	227	PACE$
15562	25	OUND
15567	196	TICK
15572	34	TRIG
15577	35	HEN
15582	218	RON
15586	162	ROFF
15591	163	AB(
15595	219	O
15597	217	IME
15601	203	AN
15604	13	SING
15610	228	SR
15613	221	AL
15617	20	ARPTR
15623	231	DP
15626	200	POKE
15631	198	PEEK
15636	24	IDTH
15642	160	AIT
15646	150	OR

APPENDICE B

CODIFICA DEI NUMERI IN BCD

BCD significa Decimale in Codice Binario, ed è un modo di scrivere i numeri decimali in codice binario, diverso dalla *scala* binaria. In un numero BCD, ogni cifra decimale è convertita in un numero binario di quattro bit. Questo implica che si possono usare solo gli equivalenti binari dei numeri da 0 a 9, e che un byte può contenere solo un numero decimale di due cifre. Per esempio, il numero 47 viene convertito in 01000111, i codici binari per "4" e "7". Numeri binari di questo tipo sono molto più facili da convertire in codici ASCII, e le operazioni aritmetiche sono precise, anche se talvolta lente.
Un'azione dello Z80 che rende preferibile l'uso dei numeri BCD è il comando DAA. Lo Z80, come ogni altro microprocessore, esegue le azioni aritmetiche solo in forma binaria. Ciò significa che se addizionate 10010111 a 01000011, avrete 11011010. Secondo il codice binario senza segno, questo significa addizionare il numero decimale 151 a 67 ed avere come risultato 218. In BCD i numeri sono invece 97 e 43 e la loro somma dovrebbe essere 140. Se viene usata l'istruzione DAA dopo ogni addizione, il risultato contenuto nell'accumulatore viene convertito in BCD, e il carry avrà un aggiustamento, se necessario. DAA agisce su ogni gruppo di quattro bit del risultato. Se il gruppo è uguale o inferiore a 9, non c'è alcun effetto. Se il gruppo di quattro ha un valore superiore a 9, allora viene addizionato 6, e il risultato è dato in forma binaria, con un riporto al bit successivo. Nel nostro esempio, il semi-byte (o "nibble") inferiore è 1010, dieci in forma decimale. Se si aggiunge sei, diventa sedici, 10000 in forma binaria. Quindi il semi-byte inferiore sarà uguale a 0000, con un riporto. Il semi-byte superiore è 1101. Aggiungendo sei, e il riporto, diventa 20, cioè 10100 in forma binaria, con il bit più significativo nel carry. Il risultato sarà quindi 101000000, 140 in BCD, ed è infatti il risultato corretto.

APPENDICE C

MEMORIZZAZIONE DELLE VARIABILI

La Fig. C.1 mostra come i numeri assegnati alle variabili siano memorizzati in righe, in modo diverso dalla memorizzazione nella VLT. Questo metodo si discosta dai metodi utilizzati da altri computer. Nella Fig. C.1(a), il numero 5 assegnato alla variabile intera A% è codificato sotto forma delle cifre esadecimali 16. Nella Fig. C.1(b), il numero 32001 assegnato ad A% è codificato sotto forma delle cifre esadecimali 1C 01 7D. La parte EFH è il "simbolo" (token) per il segno di uguaglianza, e i numeri che seguono esprimono il tipo e il valore. Per i numeri fino a 9 decimali, viene addizionato il numero 11H, 17 decimale, al numero intero

```
10 A%=5
20 A!=55.5555                      (a)
30 A#=5.555555555#

8000 00 0A 80 0A 00 41 25 EF ....A%o
8008 16 00 17 80 14 00 41 21 .....A!
8010 EF 1D 42 55 55 55 00 28 o.BUUU.(
8018 80 1E 00 41 23 EF 1F 41 ..A#o.A
8020 55 55 55 55 55 00 00 00 UUUUU...
8028 00 00 08 41 00 00 00 00 ...A....
8030 00 00 00 00 00 1E 00 20 .......
8038 DA 20 91 20 4E 25 00 45 Z  N%.E
```

```
10 A%=32001
20 B!=123.456                      (b)
30 C#=1234.5678#

8000 00 0C 80 0A 00 41 25 EF ....A%o
8008 1C 01 7D 00 19 80 14 00 ..}.....
8010 42 21 EF 1D 43 12 34 56 B!o.C.4V
8018 00 2A 80 1E 00 43 23 EF .*..C#o
8020 1F 44 12 34 56 78 00 00 .D.4Vx..
8028 00 00 00 00 08 41 00 00 .....A..
8030 00 00 00 00 00 00 00 0A ........
8038 DA 20 91 20 4E 25 00 45 Z  N%.E
```

Fig. C.1. Esempi di righe BASIC e del modo in cui sono memorizzate. Potete vedere come sono stati codificati i numeri. (a) Il numero 5 assegnato alla variabile intera A% è codificato sotto forma delle due cifre esadecimali 16. (b) Il numero 32001 assegnato ad A% è codificato nelle cifre esadecimali 1C 01 7D.

per la codifica del valore. Per i numeri da 10 a 255, viene posto il codice 0F davanti al valore, per cui A%=10 diventa 0F 0A (esadecimale). Questo schema dell'uso di 0F come prefisso, viene seguito per i valori inferiori o uguali a 255, codificato come 0F FF. Per i valori superiori a 255, vengono usati due bytes per la codifica del valore del numero, nel consueto ordine, prima basso, poi alto, e il prefisso è ora 1C.
Per i numeri a precisione doppia e singola, il valore è codificato in BCD, in tal modo il loro prelevamento è reso più facile. Nella Fig. C.1(a), per esempio, le cifre del numero 55,5555 sono facilmente suddivise agli indirizzi da 8013H a 8015H. Il prefisso per questo numero a precisione singola è 1D, e 42 esprime la posizione della virgola decimale. Se immaginate il numero scritto come una frazione moltiplicata per una potenza di dieci, allora 55,5555 diventa 0,555555 moltiplicato 10 elevato a 2. Il 2 *viene* poi aggiunto a 40H per ottenere 42 e le cifre 555555 vengono poste dopo questo codice. Potete vedere dalla Fig. C.1(b) che il numero 123,456 è stato codificato come 0,123456 moltiplicato 10 elevato a tre, per cui si ha il codice 43 12 34 56. I numeri a doppia precisione usano 1F come prefisso, e lo stesso metodo generale di codifica. Il BCD è usato per scrivere le cifre, e la potenza di 10 viene poi aggiunta a 40H e posta davanti ai codici delle cifre.

APPENDICE D

CONVERSIONE TRA DECIMALE ED ESADECIMALE
DA DECIMALE A ESADECIMALE

(a) Bytes singoli — numeri inferiori a 256 decimale
Esempio: convertire 153 in esadecimale
153/16=9,5625 per cui 9 è la cifra superiore; la cifra inferiore è 0,5625*16=9. Il numero è quindi 99H.

Esempio: convertire 58 in esadecimale
58/16=3,625, per cui 3 è la cifra superiore; la cifra inferiore è 0,625*16=10, A in esadecimale. Il numero è quindi 3AH.

(b) Bytes doppi — numeri compresi tra 256 e 65535
Esempio: convertire 23815 in esadecimale
23815/16=1488,4375. 0,4375*16=7, è la cifra più bassa.
1488/16=93 con resto 0. 0 è la cifra successiva.
93/16=5,8125. 0,8125*16=13, cioé D esadecimale.
L'ultima cifra è 5, per cui il numero completo è 5D07H.

DA ESADECIMALE A DECIMALE

(a) Bytes singoli
Esempio: convertire 3DH in decimale
Il valore è 3*16+13 (valore decimale di D) che fa 61 decimale

Esempio: convertire A8 in decimale
Il valore è 10*16+8, cioé 168 decimale

(b) Bytes doppi
Esempio: convertire 2CA5 in decimale
La prima cifra dà 2*4096=8192. La seconda cifra dà 12*256=3072, e la terza cifra dà 16*10=160.
L'ultima cifra è 5, per cui si avrà 8192 + 3072 + 160 + 5 = 11429.

Esempio: convertire F3DBH in decimale

Prima cifra	15*4096	=	61440
Seconda cifra	3*256	=	768
Terza cifra	13*16	=	208
Quarta cifra	11	=	11
TOTALE		=	62427

APPENDICE E

I CODICI OPERATIVI DELLO Z80, CON MNEMONICHE, CODICI ESADECIMALI, CODICI DECIMALI E CODICI BINARI

Il seguente elenco di tutti i codici operativi dello Z80, con mnemoniche, codici esadecimali, codici decimali e codici binari, è formato da quattro colonne. Quando un codice è formato da più di un byte, i bytes sono stati sistemati verticalmente, per evitare confusione. Quando occorre inserire un byte di dati, di spostamento o di indirizzo, questo viene indicato con 00 per il byte di dati o di spostamento, o 0000 per indirizzo. Di solito, in questi elenchi si usa N per un byte singolo e NN per un indirizzo, ma, dal momento che è stato usato un programma per convertire un valore esadecimale in decimale e in binario per questo elenco, non si sono potute usare lettere come la N.
Il lavoro non indifferente richiesto per questo elenco comporta inevitabilmente il fare degli errori durante la sua compilazione. Ho cercato di eliminarli tutti, per quanto mi è stato possibile, ma ciò non esclude che ve ne sia ancora qualcuno.

ADC A,(HL)	BE	142	10001110
ADC A,(IX+00)	DD	221	11011101
	BE	142	10001110
	00	0	00000000
ADC A,(IY+00)	FD	253	11111101
	BE	142	10001110
	00	0	00000000
ADC A,A	8F	143	10001111
ADC A,B	88	136	10001000
ADC A,C	89	137	10001001
ADC A,D	8A	138	10001010
ADC A,00	CE	206	11001110
	00	0	00000000
ADC A,E	8B	139	10001011
ADC A,H	8C	140	10001100
ADC A,L	8D	141	10001101
ADC HL,BC	ED	237	11101101
	4A	74	01001010
ADC HL,DE	ED	237	11101101
	5A	90	01011010
ADC HL,HL	ED	237	11101101
	6A	106	01101010
ADC HL,SP	ED	237	11101101
	7A	122	01111010
ADD A,(HL)	86	134	10000110
ADD A,(IX+00)	DD	221	11011101
	86	134	10000110
	00	0	00000000
ADD A,(IY+00)	FD	253	11111101
	86	134	10000110
	00	0	00000000
ADD A,A	87	135	10000111
ADD A,B	80	128	10000000
ADD A,C	81	129	10000001
ADD A,D	82	130	10000010
ADD A,00	C6	198	11000110
	00	0	00000000
ADD A,E	83	131	10000011
ADD A,H	84	132	10000100
ADD A,L	85	133	10000101
ADD HL,BC	09	9	00001001
ADD HL,DE	19	25	00011001
ADD HL,HL	29	41	00101001
ADD HL,SP	39	57	00111001
ADD IX,BC	DD	221	11011101
	09	9	00001001
ADD IX,DE	DD	221	11011101
	19	25	00011001
ADD IX,IX	DD	221	11011101
	29	41	00101001
ADD IX,SP	DD	221	11011101
	39	57	00111001
ADD IY,BC	FD	253	11111101
	09	9	00001001
ADD IY,DE	FD	253	11111101
	19	25	00011001
ADD IY,IY	FD	253	11111101
	29	41	00101001
ADD IY,SP	FD	253	11111101
	39	57	00111001
AND (HL)	A6	166	10100110
AND (IX+00)	DD	221	11011101
	A6	166	10100110
	00	0	00000000
AND (IY+00)	FD	253	11111101
	A6	166	10100110
	00	0	00000000
AND A	A7	167	10100111
AND B	A0	160	10100000
AND C	A1	161	10100001
AND D	A2	162	10100010
AND 00	E6	230	11100110
	00	0	00000000
AND E	A3	163	10100011
AND H	A4	164	10100100
AND L	A5	165	10100101
BIT 0,(HL)	CB	203	11001011
	46	70	01000110
BIT 0,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	46	70	01000110
BIT 0,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	46	70	01000110
BIT 0,A	CB	203	11001011
	47	71	01000111
BIT 0,B	CB	203	11001011
	40	64	01000000
BIT 0,C	CB	203	11001011
	41	65	01000001
BIT 0,D	CB	203	11001011
	42	66	01000010
BIT 0,E	CB	203	11001011
	43	67	01000011
BIT 0,H	CB	203	11001011
	44	68	01000100
BIT 0,L	CB	203	11001011
	45	69	01000101
BIT 1,(HL)	CB	203	11001011
	4E	78	01001110
BIT 1,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	4E	78	01001110
BIT 1,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	4E	78	01001110
BIT 1,A	CB	203	11001011
	4F	79	01001111
BIT 1,B	CB	203	11001011
	48	72	01001000
BIT 1,C	CB	203	11001011
	49	73	01001001
BIT 1,D	CB	203	11001011
	4A	74	01001010
BIT 1,E	CB	203	11001011
	4B	75	01001011
BIT 1,H	CB	203	11001011
	4C	76	01001100
BIT 1,L	CB	203	11001011
	4D	77	01001101
BIT 2,(HL)	CB	203	11001011
	56	86	01010110
BIT 2,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	56	86	01010110

BIT 2,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	56	86	01010110
BIT 2,A	CB	203	11001011
	57	87	01010111
BIT 2,B	CB	203	11001011
	50	80	01010000
BIT 2,C	CB	203	11001011
	51	81	01010001
BIT 2,D	CB	203	11001011
	52	82	01010010
BIT 2,E	CB	203	11001011
	53	83	01010011
BIT 2,H	CB	203	11001011
	54	84	01010100
BIT 2,L	CB	203	11001011
	55	85	01010101
BIT 3,(HL)	CB	203	11001011
	5E	94	01011110
BIT 3,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	5E	94	01011110
BIT 3,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	5E	94	01011110
BIT 3,A	CB	203	11001011
	5F	95	01011111
BIT 3,B	CB	203	11001011
	58	88	01011000
BIT 3,C	CB	203	11001011
	59	89	01011001
BIT 3,D	CB	203	11001011
	5A	90	01011010
BIT 3,E	CB	203	11001011
	5B	91	01011011
BIT 3,H	CB	203	11001011
	5C	92	01011100
BIT 3,L	CB	203	11001011
	5D	93	01011101
BIT 4,(HL)	CB	203	11001011
	66	102	01100110
BIT 4,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	66	102	01100110
BIT 4,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	66	102	01100110
BIT 4,A	CB	203	11001011
	67	103	01100111
BIT 4,B	CB	203	11001011
	60	96	01100000
BIT 4,C	CB	203	11001011
	61	97	01100001
BIT 4,D	CB	203	11001011
	62	98	01100010
BIT 4,E	CB	203	11001011
	63	99	01100011
BIT 4,H	CB	203	11001011
	64	100	01100100
BIT 4,L	CB	203	11001011
	65	101	01100101
BIT 5,(HL)	CB	203	11001011
	6E	110	01101110
BIT 5,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	6E	110	01101110
BIT 5,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	6E	110	01101110
BIT 5,A	CB	203	11001011
	6F	111	01101111
BIT 5,B	CB	203	11001011
	68	104	01101000
BIT 5,C	CB	203	11001011
	69	105	01101001
BIT 5,D	CB	203	11001011
	6A	106	01101010
BIT 5,E	CB	203	11001011
	6B	107	01101011
BIT 5,H	CB	203	11001011
	6C	108	01101100
BIT 5,L	CB	203	11001011
	6D	109	01101101
BIT 6,(HL)	CB	203	11001011
	76	118	01110110
BIT 6,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	76	118	01110110
BIT 6,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	76	118	01110110
BIT 6,A	CB	203	11001011
	77	119	01110111
BIT 6,B	CB	203	11001011
	70	112	01110000
BIT 6,C	CB	203	11001011
	71	113	01110001
BIT 6,D	CB	203	11001011
	72	114	01110010
BIT 6,E	CB	203	11001011
	73	115	01110011
BIT 6,H	CB	203	11001011
	74	116	01110100
BIT 6,L	CB	203	11001011
	75	117	01110101
BIT 7,(HL)	CB	203	11001011
	7E	126	01111110
BIT 7,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	7E	126	01111110
BIT 7,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	7E	126	01111110
BIT 7,A	CB	203	11001011
	7F	127	01111111
BIT 7,B	CB	203	11001011
	78	120	01111000

BIT 7,C	CB	203	11001011
	79	121	01111001
BIT 7,D	CB	203	11001011
	7A	122	01111010
BIT 7,E	CB	203	11001011
	7B	123	01111011
BIT 7,H	CB	203	11001011
	7C	124	01111100
BIT 7,L	CB	203	11001011
	7D	125	01111101
CALL 0000	CD	205	11001101
	00	0	00000000
	00	0	00000000
CALL C,0000	DC	220	11011100
	00	0	00000000
	00	0	00000000
CALL M,0000	FC	252	11111100
	00	0	00000000
	00	0	00000000
CALL NC,0000	D4	212	11010100
	00	0	00000000
	00	0	00000000
CALL NZ,0000	C4	196	11000100
	00	0	00000000
	00	0	00000000
CALL P,0000	F4	244	11110100
	00	0	00000000
	00	0	00000000
CALL PE,0000	EC	236	11101100
	00	0	00000000
	00	0	00000000
CALL PO,0000	E4	228	11100100
	00	0	00000000
	00	0	00000000
CALL Z,0000	CC	204	11001100
	00	0	00000000
	00	0	00000000
CCF	3F	63	00111111
CP(HL)	BE	190	10111110
CP(IX+0)	DD	221	11011101
	BE	190	10111110
	00	0	00000000
CP(IY+0)	FD	253	11111101
	BE	190	10111110
	00	0	00000000
CP A	BF	191	10111111
CP B	B8	184	10111000
CP C	B9	185	10111001
CP D	BA	186	10111010
CP 00	FE	254	11111110
	00	0	00000000
CP E	BB	187	10111011
CP H	BC	188	10111100
CP L	BD	189	10111101
CPD	ED	237	11101101
	A9	169	10101001
CPDR	ED	237	11101101
	B9	185	10111001
CPI	ED	237	11101101
	A1	161	10100001
CPIR	ED	237	11101101
	B1	177	10110001
CPL	2F	47	00101111
DAA	27	39	00100111

DEC(HL)	35	53	00110101
DEC(IX+0)	DD	221	11011101
	35	53	00110101
	00	0	00000000
DEC(IY+0)	FD	253	11111101
	35	53	00110101
	00	0	00000000
DEC A	3D	61	00111101
DEC B	05	5	00000101
DEC BC	0B	11	00001011
DEC C	0D	13	00001101
DEC D	15	21	00010101
DEC DE	1B	27	00011011
DEC E	1D	29	00011101
DEC H	25	37	00100101
DEC HL	2B	43	00101011
DEC IX	DD	221	11011101
	2B	43	00101011
DEC IY	FD	253	11111101
	2B	43	00101011
DEC L	2D	45	00101101
DEC SP	3B	59	00111011
DI	F3	243	11110011
DJNZ,00	10	16	00010000
	00	0	00000000
EI	FB	251	11111011
EX(SP),HL	E3	227	11100011
EX(SP)IX	DD	221	11011101
	E3	227	11100011
EX(SP)IY	FD	253	11111101
	E3	227	11100011
EX AF,AF'	08	8	00001000
EX DE,HL	EB	235	11101011
EXX	D9	217	11011001
HLT	76	118	01110110
IM 0	ED	237	11101101
	46	70	01000110
IM 1	ED	237	11101101
	56	86	01010110
IM 2	ED	237	11101101
	5E	94	01011110
IN A,(C)	ED	237	11101101
	78	120	01111000
IN A,PORT	DB	219	11011011
	00	0	00000000
IN B,(C)	ED	237	11101101
	40	64	01000000
IN C,(C)	ED	237	11101101
	48	72	01001000
IN D,(C)	ED	237	11101101
	50	80	01010000
IN E,(C)	ED	237	11101101
	58	88	01011000
IN F,(C)	ED	237	11101101
	70	112	01110000
IN H,(C)	ED	237	11101101
	60	96	01100000
IN L,(C)	ED	237	11101101
	68	104	01101000
INC(HL)	34	52	00110100
INC(IX+00)	DD	221	11011101
	34	52	00110100
	00	0	00000000
INC(IY+00)	FD	253	11111101
	34	52	00110100
	00	0	00000000

Mnemonic	Hex	Dec	Binary
INC A	3C	60	00111100
INC B	04	4	00000100
INC BC	03	3	00000011
INC C	0C	12	00001100
INC D	14	20	00010100
INC DE	13	19	00010011
INC E	1C	28	00011100
INC H	24	36	00100100
INC HL	23	35	00100011
INC IX	DD	221	11011101
	23	35	00100011
INC IY	FD	253	11111101
	23	35	00100011
INC L	2C	44	00101100
INC SP	33	51	00110011
IND	ED	237	11101101
	AA	170	10101010
INDR	ED	237	11101101
	BA	186	10111010
INI	ED	237	11101101
	A2	162	10100010
INIR	ED	237	11101101
	B2	178	10110010
JP(HL)	E9	233	11101001
JP(IX)	DD	221	11011101
	E9	233	11101001
JP(IY)	FD	253	11111101
	E9	233	11101001
JP 0000	C3	195	11000011
	00	0	00000000
	00	0	00000000
JP C,0000	DA	218	11011010
	00	0	00000000
	00	0	00000000
JP M,0000	FA	250	11111010
	00	0	00000000
	00	0	00000000
JP NC,0000	D2	210	11010010
	00	0	00000000
	00	0	00000000
JP NZ,0000	C2	194	11000010
	00	0	00000000
	00	0	00000000
JP P,0000	F2	242	11110010
	00	0	00000000
	00	0	00000000
JP PE,0000	EA	234	11101010
	00	0	00000000
	00	0	00000000
JP PO,0000	E2	226	11100010
	00	0	00000000
	00	0	00000000
JP Z,0000	CA	202	11001010
	00	0	00000000
	00	0	00000000
JR C,00	38	56	00111000
	00	0	00000000
JR 00	18	24	00011000
	00	0	00000000
JR NC,00	30	48	00110000
	00	0	00000000
JR NZ,00	20	32	00100000
	00	0	00000000
JR Z,00	28	40	00101000
	00	0	00000000
LD(0000),A	32	50	00110010
	00	0	00000000
	00	0	00000000
LD(0000),BC	ED	237	11101101
	43	67	01000011
	00	0	00000000
	00	0	00000000
LD(0000),DE	ED	237	11101101
	53	83	01010011
	00	0	00000000
	00	0	00000000
LD(0000),HL	ED	237	11101101
	63	99	01100011
	00	0	00000000
	00	0	00000000
LD(0000),HL	22	34	00100010
	00	0	00000000
	00	0	00000000
LD(0000),IX	DD	221	11011101
	22	34	00100010
	00	0	00000000
	00	0	00000000
LD(0000),IY	FD	253	11111101
	22	34	00100010
	00	0	00000000
	00	0	00000000
LD(0000),SP	ED	237	11101101
	73	115	01110011
	00	0	00000000
	00	0	00000000
LD(BC),A	02	2	00000010
LD(DE),A	12	18	00010010
LD(HL),A	77	119	01110111
LD(HL),B	70	112	01110000
LD(HL),C	71	113	01110001
LD(HL),D	72	114	01110010
LD(HL),00	36	54	00110110
	00	0	00000000
LD(HL),E	73	115	01110011
LD(HL),H	74	116	01110100
LD(HL),L	75	117	01110101
LD(IX+00),A	DD	221	11011101
	77	119	01110111
	00	0	00000000
LD(IX+00),B	DD	221	11011101
	70	112	01110000
	00	0	00000000
LD(IX+00),C	DD	221	11011101
	71	113	01110001
	00	0	00000000
LD(IX+00),00	DD	221	11011101
	36	54	00110110
	00	0	00000000
	00	0	00000000
LD,(IX+00),D	DD	221	11011101
	72	114	01110010
	00	0	00000000
LD(IX+00),E	DD	221	11011101
	73	115	01110011
	00	0	00000000
LD(IX+00),H	DD	221	11011101
	74	116	01110100
	00	0	00000000
LD(IX+00),L	DD	221	11011101
	75	117	01110101
	00	0	00000000

LD(IY+00),A	FD	253	11111101
	77	119	01110111
	00	0	00000000
LD(IY+00),B	FD	253	11111101
	70	112	01110000
	00	0	00000000
LD(IY+00),C	FD	253	11111101
	71	113	01110001
	00	0	00000000
LD(IY+00),D	FD	253	11111101
	72	114	01110010
	00	0	00000000
LD(IY+00),00	FD	253	11111101
	36	54	00110110
	00	0	00000000
	00	0	00000000
LD(IY+00),E	ED	237	11101101
	73	115	01110011
	00	0	00000000
LD(IY+00),H	FD	253	11111101
	74	116	01110100
	00	0	00000000
LD(IY+00),L	FD	253	11111101
	75	117	01110101
	00	0	00000000
LD A,(0000)	3A	58	00111010
	00	0	00000000
	00	0	00000000
LD A,(BC)	0A	10	00001010
LD A,(DE)	1A	26	00011010
LD A,(HL)	7E	126	01111110
LD A,(IX+00)	DD	221	11011101
	7E	126	01111110
	00	0	00000000
LD A,(IY+00)	FD	253	11111101
	7E	126	01111110
	00	0	00000000
LD A,A	7F	127	01111111
LD A,B	78	120	01111000
LD A,C	79	121	01111001
LD A,D	7A	122	01111010
LD A,00	3E	62	00111110
	00	0	00000000
LD A,E	7B	123	01111011
LD A,H	7C	124	01111100
LD A,I	ED	237	11101101
	57	87	01010111
LD A,L	7D	125	01111101
LD A,R	ED	237	11101101
	5F	95	01011111
LD B,(HL)	46	70	01000110
LD B,(IX+00)	DD	221	11011101
	46	70	01000110
	00	0	00000000
LD B,(IY+00)	FD	253	11111101
	46	70	01000110
	00	0	00000000
LD B,A	47	71	01000111
LD B,B	40	64	01000000
LD B,C	41	65	01000001
LD B,D	42	66	01000010
LD B,00	06	6	00000110
	00	0	00000000
LD B,E	43	67	01000011
LD B,H	44	68	01000100
LD B,L	45	69	01000101
LD BC,(0000)	ED	237	11101101
	4B	75	01001011
	00	0	00000000
	00	0	00000000
LD BC,0000	01	1	00000001
	00	0	00000000
	00	0	00000000
LD C,(HL)	4E	78	01001110
LD C,(IX+00)	DD	221	11011101
	4E	78	01001110
	00	0	00000000
LD C,(IY+00)	FD	253	11111101
	4E	78	01001110
	00	0	00000000
LD C,A	4F	79	01001111
LD C,B	48	72	01001000
LD C,C	49	73	01001001
LD C,D	4A	74	01001010
LD C,00	0E	14	00001110
	00	0	00000000
LD C,E	4B	75	01001011
LD C,H	4C	76	01001100
LD C,L	4D	77	01001101
LD D,(HL)	56	86	01010110
LD D,(IX+00)	DD	221	11011101
	56	86	01010110
	00	0	00000000
LD D,(IY+00)	FD	253	11111101
	56	86	01010110
	00	0	00000000
LD D,A	57	87	01010111
LD D,B	50	80	01010000
LD D,C	51	81	01010001
LD D,D	52	82	01010010
LD D,00	16	22	00010110
	00	0	00000000
LD D,E	53	83	01010011
LD D,H	54	84	01010100
LD D,L	55	85	01010101
LD DE,(0000)	ED	237	11101101
	5B	91	01011011
	00	0	00000000
	00	0	00000000
LD DE,0000	11	17	00010001
	00	0	00000000
	00	0	00000000
LD E,(HL)	5E	94	01011110
LD E,(IX+00)	DD	221	11011101
	5E	94	01011110
	00	0	00000000
LD E,(IY+00	FD	253	11111101
	5E	94	01011110
	00	0	00000000
LD E,A	5F	95	01011111
LD E,B	58	88	01011000
LD E,C	59	89	01011001
LD E,D	5A	90	01011010
LD E,00	1E	30	00011110
	00	0	00000000
LD E,E	5B	91	01011011
LD E,H	5C	92	01011100
LD E,L	5D	93	01011101

LD H,(HL)	66	102	01100110
LD H,(IX+00)	DD	221	11011101
	66	102	01100110
	00	0	00000000
LD H,(IY+00)	FD	253	11111101
	66	102	01100110
	00	0	00000000
LD H,A	67	103	01100111
LD H,B	60	96	01100000
LD H,C	61	97	01100001
LD H,D	62	98	01100010
LD H,00	26	38	00100110
	00	0	00000000
LD H,E	63	99	01100011
LD H,H	64	100	01100100
LD H,L	65	101	01100101
LD HL,(0000)	ED	237	11101101
	6B	107	01101011
	00	0	00000000
	00	0	00000000
LD HL,(0000)	2A	42	00101010
	00	0	00000000
	00	0	00000000
LD HL,0000	21	33	00100001
	00	0	00000000
	00	0	00000000
LD I,A	ED	237	11101101
	47	71	01000111
LD IX,(0000)	DD	221	11011101
	2A	42	00101010
	00	0	00000000
	00	0	00000000
LD IX,0000	DD	221	11011101
	21	33	00100001
	00	0	00000000
	00	0	00000000
LD IY,(0000)	FD	253	11111101
	2A	42	00101010
	00	0	00000000
	00	0	00000000
LD IY,0000	FD	253	11111101
	21	33	00100001
	00	0	00000000
	00	0	00000000
LD L,(HL)	6E	110	01101110
LD L,(IX+00)	DD	221	11011101
	6E	110	01101110
	00	0	00000000
LD L,(IY+00)	FD	253	11111101
	6E	110	01101110
	00	0	00000000
LD L,A	6F	111	01101111
LD L,B	68	104	01101000
LD L,C	69	105	01101001
LD L,D	6A	106	01101010
LD L,00	2E	46	00101110
	00	0	00000000
LD L,E	6B	107	01101011
LD L,H	6C	108	01101100
LD L,L	6D	109	01101101
LD R,A	ED	237	11101101
	4F	79	01001111
LD SP,(0000)	ED	237	11101101
	7B	123	01111011
	00	0	00000000
	00	0	00000000

LD SP,0000	31	49	00110001
	00	0	00000000
	00	0	00000000
LD SP,HL	F9	249	11111001
LD SP,IX	DD	221	11011101
	F9	249	11111001
LD SP,IY	FD	253	11111101
	F9	249	11111001
LDD	ED	237	11101101
	A8	168	10101000
LDDR	ED	237	11101101
	B8	184	10111000
LDI	ED	237	11101101
	A0	160	10100000
LDIR	ED	237	11101101
	B0	176	10110000
NEG	ED	237	11101101
	44	68	01000100
NOP	00	0	00000000
OR(HL)	B6	182	10110110
OR(IX+00)	DD	221	11011101
	B6	182	10110110
	00	0	00000000
OR(IY+00)	FD	253	11111101
	B6	182	10110110
	00	0	00000000
OR A	B7	183	10110111
OR B	B0	176	10110000
OR C	B1	177	10110001
OR D	B2	178	10110010
OR 00	F6	246	11110110
	00	0	00000000
OR E	B3	179	10110011
OR H	B4	180	10110100
OR L	B5	181	10110101
OTDR	ED	237	11101101
	BB	187	10111011
OTIR	ED	237	11101101
	B3	179	10110011
OUT(C),A	ED	237	11101101
	79	121	01111001
OUT(C),B	ED	237	11101101
	41	65	01000001
OUT(C),C	ED	237	11101101
	49	73	01001001
OUT(C),D	ED	237	11101101
	51	81	01010001
OUT(C),E	ED	237	11101101
	59	89	01011001
OUT(C),H	ED	237	11101101
	61	97	01100001
OUT(C),L	ED	237	11101101
	69	105	01101001
OUT00,A	D3	211	11010011
	00	0	00000000
OUTD	ED	237	11101101
	AB	171	10101011
OUTI	ED	237	11101101
	A3	163	10100011
POP AF	F1	241	11110001
POP BC	C1	193	11000001
POP DE	D1	209	11010001
POP HL	E1	225	11100001
POP IX	DD	221	11011101
	E1	225	11100001

Instruction	Hex	Dec	Binary
POP IY	FD	253	11111101
	E1	225	11100001
PUSH AF	F5	245	11110101
PUSH BC	C5	197	11000101
PUSH DE	D5	213	11010101
PUSH HL	E5	229	11100101
PUSH IX	DD	221	11011101
	E5	229	11100101
PUSH IY	FD	253	11111101
	E5	229	11100101
RES 0,(HL)	CB	203	11001011
	86	134	10000110
RES 0,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	86	134	10000110
RES 0,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	86	134	10000110
RES 0,A	CB	203	11001011
	87	135	10000111
RES 0,B	CB	203	11001011
	80	128	10000000
RES 0,C	CB	203	11001011
	81	129	10000001
RES 0,D	CB	203	11001011
	82	130	10000010
RES 0,E	CB	203	11001011
	83	131	10000011
RES 0,H	CB	203	11001011
	84	132	10000100
RES 0,L	CB	203	11001011
	85	133	10000101
RES 1,(HL)	CB	203	11001011
	8E	142	10001110
RES 1,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	8D	141	10001101
RES 1,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	8D	141	10001101
RES 1,A	CB	203	11001011
	8F	143	10001111
RES 1,B	CB	203	11001011
	88	136	10001000
RES 1,C	CB	203	11001011
	89	137	10001001
RES 1,D	CB	203	11001011
	8A	138	10001010
RES 1,E	CB	203	11001011
	8B	139	10001011
RES 1,H	CB	203	11001011
	8C	140	10001100
RES 1,L	CB	203	11001011
	8D	141	10001101
RES 2,(HL)	CB	203	11001011
	96	150	10010110
RES 2,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	96	150	10010110
RES 2,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	96	150	10010110
RES 2,A	CB	203	11001011
	97	151	10010111
RES 2,B	CB	203	11001011
	90	144	10010000
RES 2,C	CB	203	11001011
	91	145	10010001
RES 2,D	CB	203	11001011
	92	146	10010010
RES 2,E	CB	203	11001011
	93	147	10010011
RES 2,H	CB	203	11001011
	94	148	10010100
RES 2,L	CB	203	11001011
	95	149	10010101
RES 3,(HL)	CB	203	11001011
	9E	158	10011110
RES 3,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	9E	158	10011110
RES 3,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	9E	158	10011110
RES 3,A	CB	203	11001011
	9F	159	10011111
RES 3,B	CB	203	11001011
	98	152	10011000
RES 3,C	CB	203	11001011
	99	153	10011001
RES 3,D	CB	203	11001011
	9A	154	10011010
RES 3,E	CB	203	11001011
	9B	155	10011011
RES 3,H	CB	203	11001011
	9C	156	10011100
RES 3,L	CB	203	11001011
	9D	157	10011101
RES 4,(HL)	CB	203	11001011
	A6	166	10100110
RES 4,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	A6	166	10100110
RES 4,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	A6	166	10100110
RES 4,A	CB	203	11001011
	A7	167	10100111
RES 4,B	CB	203	11001011
	A0	160	10100000
RES 4,C	CB	203	11001011
	A1	161	10100001
RES 4,D	CB	203	11001011
	A2	162	10100010
RES 4,E	CB	203	11001011
	A3	163	10100011
RES 4,H	CB	203	11001011
	A4	164	10100100

Instruction	Hex	Dec	Binary
RES 4,L	CB	203	11001011
	A5	165	10100101
RES 5,(HL)	CB	203	11001011
	AE	174	10101110
RES 5,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	AE	174	10101110
RES 5,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	AE	174	10101110
RES 5,A	CB	203	11001011
	AF	175	10101111
RES 5,B	CB	203	11001011
	A8	168	10101000
RES 5,C	CB	203	11001011
	A9	169	10101001
RES 5,D	CB	203	11001011
	AA	170	10101010
RES 5,E	CB	203	11001011
	AB	171	10101011
RES 5,HCBAC	CB	203	11001011
	AC	172	10101100
RES 5,L	CB	203	11001011
	AD	173	10101101
RES 6,(HL)	CB	203	11001011
	B6	182	10110110
RES 6,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	B6	182	10110110
RES 6,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	B6	182	10110110
RES 6,A	CB	203	11001011
	B7	183	10110111
RES 6,B	CB	203	11001011
	B0	176	10110000
RES 6,C	CB	203	11001011
	B1	177	10110001
RES 6,D	CB	203	11001011
	B2	178	10110010
RES 6,E	CB	203	11001011
	B3	179	10110011
RES 6,H	CB	203	11001011
	B4	180	10110100
RES 6.L	CB	203	11001011
	B5	181	10110101
RES 7,(HL)	CB	203	11001011
	BE	190	10111110
RES 7,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	BE	190	10111110
RES 7,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	BE	190	10111110
RES 7,A	CB	203	11001011
	BF	191	10111111
RES 7,B	CB	203	11001011
	B8	184	10111000
RES 7,C	CB	203	11001011
	B9	185	10111001
RES 7,D	CB	203	11001011
	BA	186	10111010
RES 7,E	CB	203	11001011
	BB	187	10111011
RES 7,H	CB	203	11001011
	BC	188	10111100
RES 7,L	CB	203	11001011
	BD	189	10111101
RET	C9	201	11001001
RET C	D8	216	11011000
RET M	F8	248	11111000
RET NC	D0	208	11010000
RET NZ	C0	192	11000000
RET P	F0	240	11110000
RET PE	E8	232	11101000
RET PO	E0	224	11100000
RET Z	C8	200	11001000
RETI	ED	237	11101101
	4D	77	01001101
RETN	ED	237	11101101
	45	69	01000101
RL(HL)	CB	203	11001011
	16	22	00010110
RL(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	16	22	00010110
RL(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	16	22	00010110
RL A	CB	203	11001011
	17	23	00010111
RL B	CB	203	11001011
	10	16	00010000
RL C	CB	203	11001011
	11	17	00010001
RL D	CB	203	11001011
	12	18	00010010
RL E	CB	203	11001011
	13	19	00010011
RL H	CB	203	11001011
	14	20	00010100
RL L	CB	203	11001011
	15	21	00010101
RLA	17	23	00010111
RLC(HL)	CB	203	11001011
	06	6	00000110
RLC(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	06	6	00000110
RLC(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	06	6	00000110
RLC A	CB	203	11001011
	07	7	00000111
RLC B	CB	203	11001011
	00	0	00000000
RLC C	CB	203	11001011
	01	1	00000001

RLC D	CB	203	11001011
	02	2	00000010
RLC E	CB	203	11001011
	03	3	00000011
RLC H	CB	203	11001011
	04	4	00000100
RLC L	CB	203	11001011
	05	5	00000101
RLCA	07	7	00000111
RLD	ED	237	11101101
	6F	111	01101111
RR(HL)	CB	203	11001011
	1E	30	00011110
RR(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	1E	30	00011110
RR(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	1E	30	00011110
RR A	CB	203	11001011
	1F	31	00011111
RR B	CB	203	11001011
	18	24	00011000
RR C	CB	203	11001011
	19	25	00011001
RR D	CB	203	11001011
	1A	26	00011010
RR E	CB	203	11001011
	1B	27	00011011
RR H	CB	203	11001011
	1C	28	00011100
RR L	CB	203	11001011
	1D	29	00011101
RRA	1F	31	00011111
RRC(HL)	CB	203	11001011
	0E	14	00001110
RRC(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	0E	14	00001110
RRC(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	0E	14	00001110
RRC A	CB	203	11001011
	0F	15	00001111
RRC B	CB	203	11001011
	08	8	00001000
RRC C	CB	203	11001011
	09	9	00001001
RRC D	CB	203	11001011
	0A	10	00001010
RRC E	CB	203	11001011
	0B	11	00001011
RRC H	CB	203	11001011
	0C	12	00001100
RRC L	CB	203	11001011
	0D	13	00001101
RRCA	0F	15	00001111
RRD	ED	237	11101101
	67	103	01100111
RST 00	C7	199	11000111

RST 08	CF	207	11001111
RST 10	D7	215	11010111
RST 18	DF	223	11011111
RST 20	E7	231	11100111
RST 28	EF	239	11101111
RST 30	F7	247	11110111
RST 38	FF	255	11111111
SBC A,(HL)	9E	158	10011110
SBC A,(IX+00)	DD	221	11011101
	9E	158	10011110
	00	0	00000000
SBC A,(IY+00)	FD	253	11111101
	9E	158	10011110
	00	0	00000000
SBC A,A	9F	159	10011111
SBC A,B	98	152	10011000
SBC A,C	99	153	10011001
SBC A,D	9A	154	10011010
SBC A,00	DE	222	11011110
	00	0	00000000
SBC A,E	9B	155	10011011
SBC A,H	9C	156	10011100
SBC A,L	9D	157	10011101
SBC HL,BC	ED	237	11101101
	42	66	01000010
SBC HL,DE	ED	237	11101101
	52	82	01010010
SBC HL,HL	ED	237	11101101
	62	98	01100010
SBC HL,SP	ED	237	11101101
	72	114	01110010
SCF	37	55	00110111
SET 0,(HL)	CB	203	11001011
	C6	198	11000110
SET0,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	C6	198	11000110
SET0,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	C6	198	11000110
SET0,A	CB	203	11001011
	C7	199	11000111
SET0,B	CB	203	11001011
	C0	192	11000000
SET0,C	CB	203	11001011
	C1	193	11000001
SET0,D	CB	203	11001011
	C2	194	11000010
SET0,E	CB	203	11001011
	C3	195	11000011
SET0,H	CB	203	11001011
	C4	196	11000100
SET0,L	CB	203	11001011
	C5	197	11000101
SET1,(HL)	CB	203	11001011
	CE	206	11001110
SET1,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	CE	206	11001110
SET1,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	CE	206	11001110

SET1,A	CB	203	11001011
	CF	207	11001111
SET1,B	CB	203	11001011
	C8	200	11001000
SET1,C	CB	203	11001011
	C9	201	11001001
SET1,D	CB	203	11001011
	CA	202	11001010
SET1,E	CB	203	11001011
	CB	203	11001011
SET1,H	CB	203	11001011
	CC	204	11001100
SET1,L	CB	203	11001011
	CD	205	11001101
SET2,(HL)	CB	203	11001011
	D6	214	11010110
SET2,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	D6	214	11010110
SET2,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	D6	214	11010110
SET2,A	CB	203	11001011
	D7	215	11010111
SET2,B	CB	203	11001011
	D0	208	11010000
SET2,C	CB	203	11001011
	D1	209	11010001
SET2,D	CB	203	11001011
	D2	210	11010010
SET2,E	CB	203	11001011
	D3	211	11010011
SET2,H	CB	203	11001011
	D4	212	11010100
SET2,L	CB	203	11001011
	D5	213	11010101
SET3,(HL)	CB	203	11001011
	DE	222	11011110
SET3,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	DE	222	11011110
SET3,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	DE	222	11011110
SET3,A	CB	203	11001011
	DF	223	11011111
SET3,B	CB	203	11001011
	D8	216	11011000
SET3,C	CB	203	11001011
	D9	217	11011001
SET3,D	CB	203	11001011
	DA	218	11011010
SET3,E	CB	203	11001011
	DB	219	11011011
SET3,H	CB	203	11001011
	DC	220	11011100
SET3,L	CB	203	11001011
	DD	221	11011101
SET4,	CB	203	11001011
	E6	230	11100110

SET4,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	E6	230	11100110
SET4,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	E6	230	11100110
SET4,A	CB	203	11001011
	E7	231	11100111
SET4,B	CB	203	11001011
	E0	224	11100000
SET4,C	CB	203	11001011
	E1	225	11100001
SET4,D	CB	203	11001011
	E2	226	11100010
SET4,E	CB	203	11001011
	E3	227	11100011
SET4,H	CB	203	11001011
	E4	228	11100100
SET4,L	CB	203	11001011
	E5	229	11100101
SET5,(HL)	CB	203	11001011
	EE	238	11101110
SET5,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	EE	238	11101110
SET5,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	EE	238	11101110
SET5,A	CB	203	11001011
	EF	239	11101111
SET5,B	CB	203	11001011
	E8	232	11101000
SET5,C	CB	203	11001011
	E9	233	11101001
SET5,D	CB	203	11001011
	EA	234	11101010
SET5,E	CB	203	11001011
	EB	235	11101011
SET5,H	CB	203	11001011
	EC	236	11101100
SET5,L	CB	203	11001011
	ED	237	11101101
SET6,(HL)	CB	203	11001011
	F6	246	11110110
SET6,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	F6	246	11110110
SET6,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	F6	246	11110110
SET6,A	CB	203	11001011
	F7	247	11110111
SET6,B	CB	203	11001011
	F0	240	11110000
SET6,C	CB	203	11001011
	F1	241	11110001
SET6,D	CB	203	11001011
	F2	242	11110010

SET6,E	CB	203	11001011
	F3	243	11110011
SET6,H	CB	203	11001011
	F4	244	11110100
SET6,L	CB	203	11001011
	F5	245	11110101
SET7,(HL)	CB	203	11001011
	FE	254	11111110
SET7,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	FE	254	11111110
SET7,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	FE	254	11111110
SET7,A	CB	203	11001011
	FF	255	11111111
SET7,B	CB	203	11001011
	F8	248	11111000
SET7,C	CB	203	11001011
	F9	249	11111001
SET7,D	CB	203	11001011
	FA	250	11111010
SET7,E	CB	203	11001011
	FB	251	11111011
SET7,H	CB	203	11001011
	FC	252	11111100
SET7,L	CB	203	11001011
	FD	253	11111101
SLA(HL)	CB	203	11001011
	26	38	00100110
SLA(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	26	38	00100110
SLA(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	26	38	00100110
SLA A	CB	203	11001011
	27	39	00100111
SLA B	CB	203	11001011
	20	32	00100000
SLA C	CB	203	11001011
	21	33	00100001
SLA D	CB	203	11001011
	22	34	00100010
SLA E	CB	203	11001011
	23	35	00100011
SLA H	CB	203	11001011
	24	36	00100100
SLA L	CB	203	11001011
	25	37	00100101
SRA(HL)	CB	203	11001011
	2E	46	00101110
SRA(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	2E	46	00101110
SRA(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	2E	46	00101110
SRA A	CB	203	11001011
	2F	47	00101111

SRA B	CB	203	11001011
	28	40	00101000
SRA C	CB	203	11001011
	29	41	00101001
SRA D	CB	203	11001011
	2A	42	00101010
SRA E	CB	203	11001011
	2B	43	00101011
SRA H	CB	203	11001011
	2C	44	00101100
SRA L	CB	203	11001011
	2D	45	00101101
SRL(HL)	CB	203	11001011
	3E	62	00111110
SRL(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	3E	62	00111110
SRL(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	3E	62	00111110
SRL A	CB	203	11001011
	3F	63	00111111
SRL B	CB	203	11001011
	38	56	00111000
SRL C	CB	203	11001011
	39	57	00111001
SRL D	CB	203	11001011
	3A	58	00111010
SRL E	CB	203	11001011
	3B	59	00111011
SRL H	CB	203	11001011
	3C	60	00111100
SRL L	CB	203	11001011
	3D	61	00111101
SUB(HL)	96	150	10010110
SUB(IX+00)	DD	221	11011101
	96	150	10010110
	00	0	00000000
SUB(IY+00)	FD	253	11111101
	96	150	10010110
	00	0	00000000
SUB A	97	151	10010111
SUB B	90	144	10010000
SUB C	91	145	10010001
SUB D	92	146	10010010
SUB 00	D6	214	11010110
	00	0	00000000
SUB E	93	147	10010011
SUB H	94	148	10010100
SUB L	95	149	10010101
XOR(HL)	AE	174	10101110
XOR(IX+00)	DD	221	11011101
	AE	174	10101110
	00	0	00000000
XOR(IY+00)	FD	253	11111101
	AE	174	10101110
	00	0	00000000
XOR A	AF	175	10101111
XOR B	A8	168	10101000
XOR C	A9	169	10101001
XOR D	AA	170	10101010
XOR 00	EE	238	11101110
	00	0	00000000
XOR E	AB	171	10101011
XOR H	AC	172	10101100

APPENDICE F

METODI DI INDIRIZZAMENTO DELLO Z80

Vi sono alcune differenze nei nomi usati per i metodi di indirizzamento dello Z80, a seconda dei vari autori di libri sulla programmazione in codice macchina dello Z80.
Indirizzamento implicito: l'indirizzamento è implicito nell'istruzione, e non è richiesto alcun riferimento di memoria. Esempio: RET.
Indirizzamento immediato: l'indirizzo dei dati è l'indirizzo di memoria che segue l'indirizzo del byte d'istruzione.
Indirizzamento diretto: l'indirizzo completo (di due bytes) della memoria è contenuto nei due bytes che seguono il byte d'istruzione.
Indiretto tramite registro: l'indirizzo dei dati è contenuto in una coppia di registri, di solito HL.
Indirizzamento indicizzato: l'indirizzo viene trovato aggiungendo un byte di spostamento ad un "indirizzo base" contenuto in uno dei registri indice.
Indirizzamento relativo: l'indirizzo dei dati è ottenuto aggiungendo un byte di spostamento con segno all'indirizzo del Contatore di Programma.
Indirizzamento di stack: il byte è memorizzato sullo stack e può essere prelevato con un comando POP, posto dopo un comando PUSH.

APPENDICE G

I TEMPI DI ESECUZIONE DELLE ISTRUZIONI

Questa tabella mostra i tempi in termini di impulsi di clock per alcune istruzioni. Poichè il "clock" dell'MSX opera a 3,58 MHz, il tempo di un impulso di clock è di 0,2793296 microsecondi. Il microsecondo è un milionesimo di secondo.

OPERAZIONE	TEMPO	COMMENTO
LD A,r	4	carica il reg. A da qualsiasi altro
LD r,n	7	caricamento immediato, per tutti i registri
LD r, (HL)	7	caricamento indiretto, per tutti i registri
LD A, (NN)	13	indirizzamento diretto
LD RR,NN	10	caricamento di una coppia di registri
LD HL,(NN)	16	HL da due bytes di memoria
PUSH RR	11	salva sullo stack una coppia di registri
POP RR	10	preleva dallo stack una coppia di registri
ADD A,r	4	addizione
INC r	4	incrementa il registro
RLA	4	rotazione dell'accumulatore
JP NN	10	salto all'indirizzo NN
JR spost	12	salto relativo, incondizionato
JR cond	12	condizione non verificata
	7	condizione verificata
CALL NN	17	chiamata della subroutine
RET	10	ritorno

Le istruzioni di spostamenti di blocchi e di confronto non sono mostrate qui, perché il tempo impiegato dipende da quante volte si ripete l'istruzione.

APPENDICE H

IL CODICE MACCHINA ESPRESSO CON UNA RIGA DATA

Ci sono varie occasioni in cui volete includere una parte in codice macchina in un programma BASIC. Può darsi che non abbiate scritto voi la parte in codice macchina, e può darsi che non la comprendiate affatto, ma dovete usarla. Il BASIC è adatto per molte operazioni di calcolo, specialmente per l'aritmetica o la matematica. Per le immagini che si muovono velocemente, però, il BASIC è molto meno utile. Se scrivete le vostre routines in codice macchina con l'aiuto di un assemblatore, terminerete con il codice macchina in memoria, ma senza un metodo semplice a disposizione per unirlo ad un programma BASIC. Se usate altri programmi che comprendono delle parti in codice macchina, e vorreste usarle nei vostri programmi, nemmeno questo è particolarmente facile.
Questo programma, però, cambia completamente la situazione. Ho scritto la prima versione di questo programma nel 1979 per un TRS-80, ed è stato così che ho potuto unire parti in codice macchina a programmi BASIC e utilizzarli. Il TRS-80 permetteva il caricamento del codice macchina direttamente da nastro, ma in molti casi questo significava caricare il codice macchina e poi caricare il BASIC. Quello che occorre è la trasformazione dei bytes del codice macchina in numeri utilizzati nelle righe di DATA. Potete quindi scrivere un breve programma "caricatore" che immetta questi bytes in memoria, così che possono essere utilizzati quando occorre.
Una volta ho trascritto i bytes del codice macchina su carta, e poi ho scritto delle righe di DATA. Se sono pochi bytes, questo è ancora tollerabile, ma se avete dei programmi in codice macchina di 20 o più bytes, diventa noioso e soprattutto è facile sbagliare, oltre che essere una perdita di tempo, perché il computer può farlo al vostro posto!

```
100 CLS:PRINT TAB(13) "CERCA DATI"
110 PRINT:PRINT" INDICA IL PUNTO
DI PARTENZA (decimale)"
120 INPUT S!
130 PRINT :PRINT"INDICA L'INDIRIZZO DI FINE"
140 INPUT F!
150 CLS:LOCATE 10,4:PRINT"...ATTENDERE..."
160 H!=F!—S! :A1!=&H8001:IF SGN(H!)=—1
THEN GOSUB 500:GOTO 100
170 T!=PEEK(A1!+2)+256*PEEK(A1!+3)
180 IF T!<>1000 THEN A1!=PEEK(A1!) +25
```

```
6*PEEK(A1!+1) :GOTO 170
190 Q!=A1+4
200 Z%=0:FOR N!=0 TO H!—1
210 L$=STR$(PEEK(S!+N!)) :LL%=LEN(L$)
220 FOR X%=2 TOLL%:Y%=ASC(MID$(L$,X%
,1)) :Z%=Z%+1
230 POKE Q!+Z%,Y%:NEXT X%:Z%=Z%+1:POK
E Q!+Z%,44
240 IF Z%<62 THEN NEXT N!
250 IF N!>=H! THEN 290
260 POKE Q!+Z%,32:FOR D%=1 TO 3:IF PE
EK(Q!+Z%+D%)=38 THEN POKE Q!+Z%+D%,32
270 NEXT D%
280Q!=Q!+71:Z%=0:NEXT N!
290 CLS:PRINT:PRINT“TRASFERIMENTO DI” ;
H!;“BYTES ,COMPLETATO”
300 PRINT:PRINT“LA RIGA 1000 ORA CONTIENE
I DATI PER IL TUO PROGRAMMA IN LINGUAG
GIO MACCHINA”
310 DELETE 100—520
500 CLS:LOCATE0,5
510 PRINT“L’INDIRIZZO DI FINE È PIU’ GRANDE
DI QUELLO D’INIZIO”:PRINT“RIPROVA”
520 FOR X=1 TO 1500:NEXT:RETURN
1000 DATA &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
1010 DATA &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
1020 DATA &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
1030 DATA &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
1040 DATA &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
1050 DATA &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
1060 DATA &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
1070 DATA &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
1080 DATA &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
1090 DATA &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
```

COME SI USA

Questo programma legge i bytes di un programma in codice macchina, e li riscrive sotto forma di righe di DATA. Ciò significa che il programma cambia se stesso durante l'esecuzione, perciò dovreste tenere diverse copie del programma nella sua forma originale. Quando passate all'esecuzione del programma, vi si chiederà di fornire, in numeri decimali, l'indirizzo di inizio e l'indirizzo di fine del codice macchina. L'indirizzo di fine deve essere maggiore di quello d'inizio, altrimenti comparirà un messaggio di errore! Questo è assai facile se avete scritto voi il codice macchina con un assemblatore, perché in tal caso avrete stabilito voi gli indirizzi. L'assemblatore li fornisce in forma esadecimale, ma la conversione è abbastanza semplice. Non è così semplice se volete usare una routine in codice macchina che fa parte di un altro programma. Un disassemblatore, come quello nello ZEN, vi può essere d'aiuto, ma dovete avere una discreta conoscenza del codice macchina per usarlo correttamente. Se non avete altre indicazioni, potete cercare di indovinare la lunghezza, provando numeri come 25, 50 100 e così via, finché il codice macchina non funziona con la vostra versione. Una volta trovata una lunghezza che funziona, potete provare degli indirizzi di fine più bassi finché non ne trovate uno che non va bene. Un indizio può essere dato dal fatto che spesso l'ultimo byte del programma è 201 (il comando di ritorno dalla subroutine in codice macchina), o tre bytes che cominciano con 195 (l'istruzione di salto). Potete perciò cercare questi numeri nelle vostre righe di DATA.
Una volta che avete immesso l'indirizzo di fine, il programma stampa il messaggio "ATTENDERE, PREGO" e si mette al lavoro. Le righe del programma dalla 1000 alla 1090 contengono il segno &, ripetuto 65 volte per ogni riga. Questo serve a riservare spazio in memoria per i bytes DATA. Le dieci righe di questo programma sono generalmente sufficienti per dei brevi programmi in codice macchina, ma se volete usare dei programmi extra-lunghi in codice macchina, avrete bisogno di aggiungere delle altre righe di DATA. In tal caso, ricordatevi che ogni riga deve contenere l'esatto numero di simboli & perché il programma funzioni correttamente. Quando il programma termina, vedrete un messaggio sullo schermo che vi dice quanti bytes di codice macchina sono stati letti e vi ricorda che i bytes sono ora in forma di numeri DATA alle righe dalla 1000 in avanti. Potete quindi cancellare le righe del programma dalla 10 alla 510 con l'istruzione DELETE 10-510. Potete anche cancellare le righe DATA che contengono solo il simbolo &. A questo punto avrete delle righe contenenti dei numeri e, di solito, una sola riga che termina con una virgola ed alcuni simboli &. Potete usare le eccellenti istruzioni di richiamo delle righe del vostro computer MSX per cancellare i simboli & in eccesso e l'ultima virgola. Ora avete un insieme di righe di DATA per un programma in codice macchina.

Per usare queste righe, dovrete scrivere il vostro programma "caricatore". Questo significa semplicemente che dovrete leggere i numeri e immetterli in memoria. Questo programma "caricatore" può essere come quello presentato nella Fig. 8.7. Notate che potete scegliere l'indirizzo di memoria per l'inizio del programma, ma a meno che non conosciate perfettamente il programma, è meglio utilizzare lo stesso indirizzo dal quale lo leggete. Niente vi impedisce di leggere i bytes dagli indirizzi da 60152 a 60182 e poi scrivere un programma caricatore che li immetta in memoria ad altri indirizzi. Però, potreste scoprire che (a) questi indirizzi sono stati usati per qualcos'altro, o (b) il programma in codice macchina non è riallocabile. Un programma non riallocabile deve essere immesso agli stessi indirizzi ogni volta, non può essere spostato senza modificare i bytes del programma. Alcuni programmi presentati in questo libro sono riallocabili, per cui potete immetterli dovunque in memoria. Dovete solo accertarvi che quegli indirizzi non siano usati dal BASIC per immagazzinare qualcos'altro.

COME FUNZIONA

L'indirizzo d'inizio viene assegnato alla variabile S! e l'indirizzo di fine alla variabile F!. La riga 160 assegna l'indirizzo d'inizio per il BASIC al nome di variabile A1!. Nella stessa riga, la lunghezza del programma in codice macchina, H!, è calcolata e, se è negativa, viene stampato il messaggio d'errore delle righe 500 e 510. Le righe 170 e 180 eseguono un'azione di ricerca di riga, cercando la riga 1000, la prima riga DATA. Quando essa viene trovata, il numero A! è l'indirizzo d'inizio della riga 1000.
Alla riga 190, Q!=A!+4 stabilisce l'indirizzo a cui è assegnato il primo codice d'istruzione della riga. Questo sarà il codice per DATA. Ora dobbiamo leggere i bytes del codice macchina, convertirli in numeri di codice ASCII, e immetterli in memoria. Per esempio, se leggiamo un byte 146, deve essere convertito nei codici ASCII 49, 52, 54, perché le righe DATA devono contenere solo codici ASCII. La riga 200 si prepara a far questo ponendo a zero una variabile Z%, e iniziando un ciclo con N!. N! è la variabile che conterrà i bytes del programma in codice macchina da 0 a H!. La variabile Z% è usata per scegliere l'indirizzo nella riga DATA a cui verrà immesso ogni codice ASCII.
La riga 210 legge dalla memoria il byte del codice macchina dall'indirizzo S!+N!, e converte questo numero in una stringa. La conversione in stringa significa che questo numero ora esiste come insieme di codici ASCII, e la seconda parte della riga fa in modo che LL% contenga la lunghezza della stringa. Ora dobbiamo far attenzione, a questo punto. Quando si usa STR$, la stringa che viene formata inizia *sempre* con uno spazio, per riservare la possibilità di inserire un segno + o —. La cifra della lunghezza perciò è superiore di un'unità al numero di cifre della stringa. Per esem-

pio, se avete il numero 143, la lunghezza di questa stringa verrà data come di quattro caratteri, non di tre. Quando la leggiamo, non vogliamo usare lo spazio, per cui cominceremo la lettura dal secondo carattere, non dal primo.
Questo viene fatto alla riga 220, con X%=2 TO LL%. Ogni codice ASCII è assegnato a Y%, con ASC(MID$(L$,X%,1)) per trovare il carattere e il suo codice ASCII. La variabile Z% viene incrementata, e poi la riga 230 immette in memoria Q!+Z% (la riga DATA in memoria) con il codice ASCII, Y%. Incrementando Z% prima di usare POKE, vi assicurate che l'indirizzo Q! non sia immesso in memoria — conteneva il codice di DATA, come ricorderete. NEXTX% assicura che tutti i caratteri del byte siano immessi in memoria, incrementando Z% ogni volta, poi Z% è ancora incrementato e 44 è immesso in memoria. 44 è il codice ASCII per la virgola, quindi a questo punto abbiamo terminato di immettere in memoria i codici ASCII per il primo byte del codice macchina.
Dobbiamo quindi controllare la fine di una riga. Non dobbiamo immettere più di 65 caratteri in una riga DATA in ogni caso. Altrimenti, l'indirizzo della "riga successiva" sarà alterato e il programma funzionerà a modo suo. La riga 240 permette di continuare l'immissione in memoria solo se Z% è minore di 62, lasciando un margine di sicurezza. La riga 250 controlla la lettura dopo la fine dei bytes in codice macchina. Se sono stati immessi 62 o più caratteri in una riga, viene eseguita la riga 260. Questa immette uno spazio, e controlla il resto della riga: se è presente un simbolo & (ASCII 38), viene sostituito da uno spazio, finché non ci sono più simboli &. La riga 280 sposta il valore di Q% al codice DATA per la successiva riga DATA, e il ciclo che inizia alla riga 200 continua. Alla fine del programma, alla riga 280, tutti i bytes del codice macchina sono stati trasformati in numeri DATA per essere utilizzati nei vostri programmi BASIC.

Programmare in codice macchina ti permette di ottenere alte velocità di esecuzione, una grafica migliore ed effetti speciali, oltre ad un completo e più efficace uso della memoria del tuo computer MSX. In genere, chi usa un computer MSX e sa già scrivere ed utilizzare programmi BASIC, è spesso riluttante ad intraprendere il passo successivo, cioé imparare a servirsi del linguaggio macchina. Questa riluttanza è comprensibile in quanto spesso l'argomento è trattato in maniera poco comprensibile ai non esperti.
Questo libro è diretto a chi possiede un computer MSX, sa programmare in BASIC, ma non conosce il funzionamento del suo computer e soprattutto non sa servirsi del linguaggio macchina.
Nel testo si descrive il funzionamento del microprocessore Z80, si indica quali indirizzi possono essere utilizzati in RAM e quali routines possono essere richiamate dalla ROM.

ISBN 88-7708-020-5
Cod. 9503

L. 25.000

MSX: LINGUAGGIO MACCHINA E ASSEMBLY
di IAN SINCLAIR

95