precedenti, in modo che gli utenti potessero acquisire nuovi elaboratori senza dover riscrivere i programmi applicativi. La compatibilità è ottenuta facendo in modo che i vari modelli appartengano a una stessa *famiglia* di elaboratori, caratterizzati dall'adozione di una stessa architettura di base.

Gli elaboratori realizzati successivamente nell'ambito di una stessa famiglia sono via via più potenti, passando progressivamente a velocità di calcolo, ampiezza di memoria indirizzabile e grado di parallelismo sempre maggiori. Una famiglia di elaboratori molto diffusa è la serie IBM AS/400, presente sul mercato per applicazioni gestionali.

Infine, menzioniamo l'avvento dei *supercalcolatori;* con questo termine si indicano macchine di grandissima potenza di calcolo, ottenuta soprattutto mediante un gran numero di processori funzionanti in parallelo e con parole di memoria molto lunghe (64 bit). Per queste caratteristiche, i supercalcolatori sono dedicati soprattutto alle elaborazioni di tipo numerico. La Cray (che prende nome dal suo fondatore) è stata la prima azienda produttrice di supercalcolatori; ora è affiancata soprattutto da alcune case giapponesi (tra queste ricordiamo la Fujitsu).

Esercizi

3.1 Illustrare la relazione che intercorre fra il numero di linee del bus dati e la lunghezza di parola.

3.2 Facendo riferimento alle porte logiche AND, OR e NOT descritte nel Capitolo 2, progettare i circuiti che consentono di trasferire dati tra i registri: dati, indirizzi, istruzione corrente, a e b. Si supponga che esista uno specifico segnale di controllo per ciascun trasferimento dei dati fra i registri. Ovviamente, non è necessario che tutti i registri siano collegati tra loro; ad esempio, il registro a deve essere collegato solo al registro dati, consentendo trasferimenti bidirezionali. Si supponga che le istruzioni siano lunghe 4 bit, di cui i primi 2 bit denotano i codici operativi e i successivi 2 bit denotano indirizzi in una piccola memoria con 4 celle.

3.3 Descrivere le caratteristiche (collegamenti, tipo e lunghezza dei registri, dimensione del bus) di una macchina dotata di un video, una tastiera, una memoria centrale di 256 Kbyte con parole lunghe 16 bit e due tipi di memorie di massa (unità a dischetto e unità a disco rigido).

3.4 Spiegare il modo in cui, nell'architettura illustrata in questo capitolo, è possibile:

- confrontare due numeri e dire quale è il maggiore;
- confrontare due numeri e dire quando sono uguali;
- accorgersi di un overflow verificatosi nel corso di una operazione.

4

Il linguaggio del calcolatore

La macchina di von Neumann, descritta nel capitolo precedente, ci fornisce un modello molto semplificato di un calcolatore; in questo capitolo, si descriverà un linguaggio operativo che consente la programmazione della macchina di von Neumann. Il linguaggio che verrà illustrato appartiene alla categoria dei cosiddetti *linguaggi macchina*, cioè linguaggi direttamente eseguibili dagli elaboratori. Come nei capitoli precedenti, la descrizione del linguaggio è qualitativa, con drastiche semplificazioni rispetto ai linguaggi macchina reali. L'obiettivo di questo capitolo è di fornire un *modello operazionale* del calcolatore, cioè illustrare il modo in cui la macchina di von Neumann può effettivamente eseguire passo-passo un programma, visto come opportuna sequenza di istruzioni del linguaggio macchina. Non ci poniamo invece l'obiettivo, ben più ambizioso e fuori luogo per i lettori di questo testo, di insegnare a programmare in linguaggio macchina.

4.1 Formato delle istruzioni

Le istruzioni del linguaggio macchina si dividono in due parti, un *codice operativo* e uno o più *operandi*; il codice operativo è sempre presente, mentre gli operandi possono mancare. Il codice operativo specifica l'operazione da compiere; gli operandi specificano, in base a varie modalità, le locazioni delle celle di memoria cui ciascuna operazione si riferisce. In alcuni linguaggi sono possibili istruzioni con molti operandi (ad esempio, due o tre operandi); in questo capitolo descriviamo un semplice linguaggio in cui ciascuna istruzione ha al più un solo operando.

Dato che le istruzioni in linguaggio macchina devono essere memorizzate nelle celle di memoria su dispositivi fisici, esse sono codificate in binario. Indichiamo con s la lunghezza di un'istruzione; sia poi m il numero di bit dedicati al codice operativo e n il numero di bit dedicati all'eventuale operando, con $s = m + n$. Chiamiamo *set di istruzioni* del linguaggio l'insieme delle diverse istruzioni eseguibili, cioè dei codici operativi; la cardinalità del set di istruzioni sarà allora un numero inferiore a 2^m. Il valore n ha invece un legame con il numero 2^k di celle di memoria presenti; infatti, per assicurare la possibilità di indirizzare tutte le celle di memoria, dovrà essere $n \geq k$. Tuttavia, vedremo nel corso del capitolo che gli indirizzi di memoria possono anche essere rappresentati in forma sintetica, richiedendo un numero inferiore di bit.

La lunghezza delle istruzioni, s, è in relazione alla lunghezza h delle parole. In molti

casi, s e k coincidono: ogni istruzione è contenuta esattamente in una cella di memoria. È anche possibile che s sia un multiplo o sottomultiplo di k; in tal caso, ciascuna istruzione occupa varie celle di memoria, oppure una stessa cella di memoria ospita più istruzioni. È infine possibile che la lunghezza dell'istruzione dipenda dal codice operativo; ad esempio, supponendo che la maggior parte delle istruzioni occupi una cella di memoria, sarà possibile attribuire a specifiche istruzioni, individuate tramite il codice operativo, due celle di memoria successive. In questo capitolo faremo l'ipotesi semplificativa che ciascuna istruzione occupi esattamente una cella di memoria.

Riassumendo le ipotesi fin qui formulate, il formato delle istruzioni del linguaggio macchina ha un campo codice operativo di m bit e un solo indirizzo, talvolta assente, di n bit; ogni istruzione è contenuta in una cella di memoria.

4.2 Esecuzione delle istruzioni

Il funzionamento normale dell'elaboratore consiste nella lettura ed esecuzione delle istruzioni dei programmi, e avviene ripetendo una sequenza di operazioni nell'unità centrale. L'esecuzione di ogni istruzione richiede lo svolgimento di tre fasi: l'acquisizione dalla memoria centrale, l'interpretazione e l'esecuzione vera e propria.

La fase di acquisizione, detta anche *fase di fetch*, richiede l'esecuzione di quattro *micro-istruzioni*; ciascuna micro-istruzione corrisponde a un trasferimento di dati fra i registri dell'unità centrale oppure tra le unità funzionali collegate dal bus. La fase di fetch avviene nel seguente modo.

1. Il contenuto del registro contatore programma (PC) viene trasferito nel registro indirizzi (RI). Si noti che PC contiene l'indirizzo della prossima istruzione da eseguire, che in genere è l'istruzione successiva rispetto all'ultima eseguita.
2. Viene eseguita un'operazione di lettura. Il contenuto della cella di memoria corrispondente all'indirizzo contenuto in RI viene trasferito nel registro dati (RD).
3. Il contenuto del registro dati viene trasferito nel registro istruzione corrente (RIC).
4. Il valore del registro PC viene incrementato di uno. In questo modo, il registro PC contiene l'indirizzo della successiva istruzione rispetto all'istruzione attualmente caricata nel registro RIC. Viene cioè predisposta l'esecuzione della successiva fase di fetch. È tuttavia possibile che durante l'esecuzione dell'istruzione corrente venga inserito nel registro PC un indirizzo differente dal successivo, alterando così l'esecuzione "in sequenza" del programma; come vedremo, istruzioni di questo genere si dicono *salti* (jump).

Sintetizzando, la fase di fetch comporta le seguenti micro-istruzioni.

PC → RI
MEM[RI] → RD
RD → RIC
PC + 1 → PC

Ciascuna microistruzione corrisponde a un trasferimento di dati fra registri o specifiche parole di memoria; la notazione proposta indica un trasferimento dati dall'elemento a sinistra della freccia verso l'elemento a destra della freccia. La notazione MEM[RI] indica la particolare parola di memoria indirizzata dal registro RI.

La successiva fase di interpretazione delle istruzioni consiste nel comprendere, in base al contenuto del registro RIC, qual è l'istruzione corrente. In questa fase viene analizzato il solo codice operativo dell'istruzione corrente. Infine, la fase di esecuzione, diversa per ciascuna operazione, consiste nell'effettuare l'operazione stessa. Se l'operazione coinvolge un operando esso viene recuperato durante la fase di esecuzione. La descrizione della fase di esecuzione delle principali istruzioni sarà oggetto del prossimo paragrafo.

4.3 Principali istruzioni

Nel seguito, descriviamo le principali istruzioni del linguaggio macchina.

- L'istruzione load (caricamento) corrisponde a una lettura dalla memoria. Essa carica il contenuto di una cella di memoria in un opportuno registro. Il nostro linguaggio comprende due istruzioni, loada e loadb, relative rispettivamente ai registri A e B. In sistemi reali, altre istruzioni di caricamento fanno riferimento agli altri registri di lavoro. L'esecuzione dell'istruzione: loada ind1, ove ind1 rappresenta un opportuno indirizzo presente nella parte operando dell'istruzione, viene effettuata nel seguente modo.

  1. Il contenuto del campo operando del registro istruzione corrente (OP(RIC)) viene copiato nel registro indirizzi (RI).
  2. Viene eseguita un'operazione di lettura. Il contenuto della cella di memoria corrispondente all'indirizzo contenuto in RI viene copiato nel registro dati (RD).
  3. Il contenuto di RD viene copiato nel registro A.

  L'istruzione loada comporta perciò le seguenti micro-istruzioni (ove OP(RIC) indica il contenuto degli ultimi n bit del registro RIC, corrispondente al campo operando dell'istruzione corrente).

  OP(RIC) → RI
  MEM[RI] → RD
  RD → A

- L'istruzione store (salvataggio) corrisponde a un'operazione di scrittura. Essa assegna il contenuto di un registro a una cella di memoria. Nel nostro semplice linguaggio, considereremo le due istruzioni storea e storb, relativi ai registri della ALU. L'esecuzione dell'istruzione: storea ind1, ove ind1 rappresenta un opportuno indirizzo nella parte operando dell'istruzione, viene effettuata nel seguente modo.

  1. Il contenuto del registro A viene copiato nel registro dati (RD).
  2. Il contenuto del campo operando del registro istruzione corrente (OP(RIC)) viene copiato nel registro indirizzi (RI).

3. Viene eseguita un'operazione di scrittura. Il contenuto del registro RD viene copiato nella cella di memoria corrispondente all'indirizzo contenuto in RI.

L'istruzione storea comporta perciò le seguenti micro-istruzioni.

A → RD
OP(RIC) → RI
RD → MEM[RI]

- L'istruzione di *lettura da una periferica*, read ind1, ove l'operando ind1 rappresenta un indirizzo nella memoria centrale, acquisisce un valore dalla periferica (ad esempio, la tastiera di un terminale o di un personal computer) e lo inserisce nella cella di memoria indirizzata. L'esecuzione dell'istruzione viene effettuata nel seguente modo.

  1. Il contenuto del registro dati RDP presente nell'interfaccia della periferica viene trasferito nel registro dati (RD), utilizzando il bus dalla periferica all'unità di elaborazione.
  2. Il contenuto del campo operando del registro istruzione corrente (OP(RIC)) viene trasferito nel registro indirizzi (RI).
  3. Viene eseguita un'operazione di scrittura in memoria centrale.

  La prima micro-istruzione presuppone che il registro RDP contenga un dato proveniente dalla periferica; questa condizione può essere verificata dal programma utilizzando il registro di stato RSP sull'interfaccia della periferica, oppure il registro RINT nell'unità centrale. Sintetizzando, l'istruzione read comporta le seguenti micro-istruzioni.

  RDP → RD
  OP(RIC) → RI
  RD → MEM[RI]

- L'istruzione di *scrittura su una periferica*, write ind1, ove l'operando ind1 rappresenta un indirizzo nella memoria centrale, acquisisce un valore dalla cella di memoria indirizzata e lo presenta in output sulla periferica (ad esempio, sul video del terminale). L'esecuzione dell'istruzione viene effettuata nel seguente modo.

  1. Il contenuto del campo indirizzi del registro istruzione corrente (OP(RIC)) viene trasferito nel registro indirizzi (RI).
  2. Viene eseguita un'operazione di lettura. Il contenuto della cella di memoria corrispondente all'indirizzo in RI viene trasferito nel registro dati (RD).
  3. Il contenuto del registro RD viene trasferito, tramite il bus, nel registro dati RDP presente nell'interfaccia della periferica.

  Inoltre, l'unità di elaborazione attiva, con un opportuno comando, la visualizzazione sulla periferica del carattere caricato in RDP. In sintesi, l'esecuzione dell'istruzione write comporta le seguenti micro-istruzioni.

  OP(RIC) → RI
  MEM[RI] → RD
  RD → RDP

- Le istruzioni relative a *operazioni fra numeri interi*, rivolte alla ALU. Le istruzioni numeriche sono: add (somma), dif (differenza), mul (moltiplicazione), div (divisione). L'effetto di ciascuna di queste operazioni è di trasferire nel registro A il risultato dell'operazione, che si applica al contenuto dei due registri A e B; nel caso di divisione tra numeri interi, il risultato viene posto nel registro A e il resto nel registro B.

- Le *istruzioni di salto* (jump), che consentono di passare a una generica istruzione del programma. I salti, in pratica, modificano la normale esecuzione del programma, in cui le istruzioni vengono eseguite una dopo l'altra. Vi sono due tipi di salti.

  - Il *salto incondizionato*, jump ind1, fa proseguire l'esecuzione del programma dall'istruzione corrispondente all'indirizzo ind1. L'istruzione viene eseguita trasferendo il contenuto del campo operando del registro istruzione corrente (che vale ind1) nel registro contatore del programma (PC). In tal modo, la successiva fase di fetch preleverà l'istruzione contenuta nella cella ind1 invece dell'istruzione successiva all'ultima eseguita.
  - Il *salto condizionato*, jumpz ind1, fa avvenire il salto del programma all'indirizzo ind1 solo se il contenuto del registro A è nullo. In pratica, l'istruzione viene eseguita utilizzando il registro di stato (RS): il salto all'indirizzo ind1 avviene solo se il bit zero del registro RS è posto a 1.

- La *istruzione di pausa*, nop (*no operation*), viene utilizzata per far trascorrere un ciclo istruzione senza svolgere alcuna operazione; ciò consente di mantenere l'unità di elaborazione in attesa del verificarsi di eventi esterni.

- L'*istruzione di arresto*, halt, fa terminare l'esecuzione del programma.

Da questo elenco sono omesse tipologie importanti di istruzione, che non fanno pertanto parte del nostro semplice linguaggio. Tra di esse, sono significative le istruzioni per la *gestione dei sottoprogrammi* e per la *gestione dello stack*, che peraltro corrispondono a un modello operazionale più complesso, che verrà illustrato nella seconda parte. Vengono inoltre omesse tutte le operazioni che operano su specifici bit dei registri.

La Tabella 4.1 riassume l'elenco delle istruzioni del linguaggio macchina presentato in questo paragrafo. Dato che il linguaggio comprende 14 diversi tipi di istruzioni, sono sufficienti 4 bit ($14 \leq 2^4$) per codificare il campo codice operativo del nostro esecutore; anche i codici binari sono riportati nella Tabella 4.1.

In genere, le macchine hanno un numero ben maggiore di istruzioni (ad esempio, il VAX della Digital ha 304 istruzioni differenti). Recentemente, una nuova impostazione si è imposta all'attenzione dei progettisti di architetture: sono state proposte le macchine RISC (*Reduced Instruction Set Computer*), caratterizzate dal disporre di un ridotto set di istruzioni, con formati regolari. Queste istruzioni sono però le più frequentemente usate e vengono attentamente ottimizzate, utilizzando un ampio numero di registri di lavoro; secondo i proponenti delle architetture RISC, le prestazioni così ottenute sono migliori di quelle di una macchina con molte istruzioni, la maggior parte delle quali è scarsamente utilizzata.

0000	LOADA
0001	LOADB
0010	STOREA
0011	STOREB
0100	READ
0101	WRITE
0110	ADD
0111	DIF
1000	MUL
1001	DIV
1010	JUMP
1011	JUMPZ
1100	NOP
1101	HALT

Tabella 4.1 Elenco delle istruzioni del linguaggio macchina

4.4 Rappresentazione dei dati in memoria centrale

L'esecuzione di un programma richiede, in genere, la presenza di dati in memoria su cui operare; infatti, ogni programma definisce una trasformazione da un insieme iniziale di dati, detti *dati di input*, a un insieme finale di dati, detti *dati di output*. I dati di un programma hanno, in genere, una vita che coincide con la durata del programma. Durante l'esecuzione viene poi gestito un insieme di dati in memoria centrale, detti *dati del programma*.

I dati di input vengono acquisiti tramite operazioni di lettura (dalle periferiche o dalla memoria di massa); i dati di output vengono progressivamente calcolati dall'elaboratore e vengono anche normalmente presentati sui dispositivi periferici, in modo che l'utente conosca il risultato dell'elaborazione. Al termine dell'elaborazione, i dati del programma cessano di esistere; è possibile però trascrivere i dati di output sulla memoria di massa, rendendoli così disponibili anche al termine della computazione del programma.

I dati di un programma hanno anch'essi un proprio formato: il calcolatore rappresenta in modo diverso, in memoria centrale, un numero intero, un numero reale o un carattere. D'altra parte, abbiamo visto nel Capitolo 2 che la codifica binaria di interi, reali e caratteri è diversa. Nel nostro linguaggio macchina, siamo in grado di manipolare soltanto dati di tipo intero; in un linguaggio di programmazione reale, esiste un certo numero di *tipi base* presenti nel linguaggio, e ciascun dato elementare manipolato da un programma appartiene a uno dei tipi base disponibile nel linguaggio.

Per manipolare dati nel linguaggio macchina qui descritto, dobbiamo perciò semplicemente predisporre alcune celle di memoria a contenere dati di tipo intero. Tali celle, ovviamente dotate di un proprio indirizzo in memoria centrale, possono essere in una generica posizione rispetto al programma. Nelle istruzioni load, store, read e write, che fanno riferimento ai dati contenuti in memoria, il campo operando dovrà contenere locazioni di memoria di celle predisposte ad accogliere i dati del programma.

Invece nelle istruzioni jump e jumpz il campo operando dovrà contenere locazioni di memoria di istruzioni del programma.

4.5 Alcuni esempi di programmi

Siamo finalmente in grado di descrivere un programma. Esso consiste di due parti, le *istruzioni* e i *dati*. In questo capitolo, per semplicità, disporremo la prima istruzione del programma nella prima cella di memoria e disporremo le istruzioni prima dei dati. La memoria risulta così partizionata in due zone: la prima contiene le istruzioni del programma, la seconda contiene i dati su cui il programma opera. Le due parti sono separate dall'ultima istruzione del programma, che dovrà essere l'istruzione halt. In generale, i programmi in linguaggio macchina vengono *caricati in memoria* da un dispositivo del sistema, a partire da un'opportuna cella di memoria (non necessariamente la prima). I programmi possono anche essere spostati nella memoria, occupando posizioni differenti in istanti di tempo successivi nel corso dell'esecuzione. Questo problema verrà discusso nel Capitolo 12.

L'esecutore così costruito è molto limitato; in pratica, è in grado soltanto di eseguire programmi elementari di tipo numerico. Tuttavia, è a questo punto possibile seguire l'esecuzione di questi programmi passo-passo, determinando le micro-istruzioni necessarie a eseguire ciascuna istruzione, i registri coinvolti, e il flusso dei dati fra i registri e lungo il bus di sistema.

Un esempio di programma completo, che verrà ampiamente discusso nel seguito, è descritto nella Figura 4.1. Il programma, semplicissimo, acquisisce due numeri interi dal terminale e stampa sul video il valore del loro prodotto. Le istruzioni del programma occupano le prime 8 celle di memoria (celle 0-7); le celle 8 e 9 sono predisposte a contenere dati di tipo intero. Per rendere il programma leggibile, usiamo i nomi delle istruzioni e i numeri decimali per far riferimento, rispettivamente, ai codici operativi e agli indirizzi del programma. Inoltre, indichiamo a sinistra di ciascuna istruzione l'indirizzo della cella di memoria che la contiene. Infine, per ricordare che le celle 8 e 9 dovranno contenere un numero intero, poniamo la parola INT in corrispondenza alle linee 8 e 9 del programma.

```
0    READ     8
1    READ     9
2    LOADA    8
3    LOADB    9
4    MUL
5    STOREA   8
6    WRITE    8
7    HALT
8    INT
9    INT
```

Figura 4.1 Programma moltiplicazione

Possiamo a questo punto simulare l'esecuzione del programma. Esso svolge inizialmente due operazioni di read, che caricano in memoria, nelle posizioni 8 e 9, due dati letti dalla tastiera del terminale. Supponiamo ad esempio che i numeri immessi dall'utente siano 8 e 4 (trascuriamo la problematica del riconoscimento di numeri costituiti da più cifre, e analogamente trascuriamo la conversione dei dati numerici, che vengono normalmente letti come sequenze di caratteri e successivamente interpretati come numeri). Le successive due istruzioni di load caricano i due numeri nei registri A e B dell'unità di elaborazione. Il successivo comando mul opera la moltiplicazione; pertanto, il registro A viene caricato con il valore 32. La successiva istruzione di store trasferisce il valore 32 dal registro A alla cella 8 di memoria; infine, l'istruzione write scrive il valore contenuto nella cella 8 di memoria sul video del terminale, ottenendo il risultato voluto. Il lettore attento avrà notato che questo programma richiede un notevole numero di spostamenti apparentemente superflui; ad esempio, il valore 32 del prodotto deve essere trascritto in memoria prima di essere passato alla periferica. In effetti, l'esecuzione qui descritta è resa necessaria da una limitazione intrinseca della nostra macchina, o meglio del suo set di istruzioni, molto ridotto.

In Figura 4.2 rappresentiamo il codice macchina in binario. Ne siamo in grado: basta convertire gli indirizzi e i codici operativi in binario. Assumiamo che sia disponibile una memoria di 4 Kbyte, con celle di memoria di 16 bit. In tal caso, $k = s = 16$, $m = 4$ $n = 12$. La lunghezza delle celle di memoria è tale da poter contenere numeri naturali compresi fra 0 e $2^{16} - 1$, oppure numeri interi (dotati di segno) compresi fra -2^{15} e $2^{15} - 1$. Ovviamente, il calcolatore deve conoscere il tipo delle variabili memorizzate nelle celle di memoria 8 e 9, cioè deve sapere se esse contengono numeri naturali oppure numeri interi, per interpretare correttamente il significato delle variabili.

Siamo anche in grado di seguire l'evoluzione del programma a livello di micro-istruzioni; l'elenco delle micro-istruzioni corrispondenti a ciascuna istruzione del programma è riportato nelle Figure 4.3 e 4.4. Si noti che ciascuna istruzione corrisponde a 4 micro-istruzioni fisse, relative alla fase di fetch, e a un numero variabile di micro-istruzioni relative alla fase di esecuzione. Prima di iniziare l'esecuzione del programma, si suppone che nel registro del contatore venga messo il valore 0, corrispondente alla prima istruzione del programma. Non viene tradotta la parte esecuzione dell'istruzione 7, che fa arrestare la macchina.

numero cella di memoria	contenuto della cella di memoria
0	0100000000001000
1	0100000000001001
2	0000000000001000
3	0001000000001001
4	1000000000000000
5	0010000000001000
6	0101000000001000
7	1101000000000000
8	0000000000000000
9	0000000000000000

Figura 4.2 Programma moltiplicazione codificato

0	fetch	PC → RI
		MEM[RI] → RD
		RD → RIC
		PC + 1 → PC
	esecuzione	RDP → RD
		OP(RIC) → RI
		RD → MEM[RI]
1	fetch	PC → RI
		MEM[RI] → RD
		RD → RIC
		PC + 1 → PC
	esecuzione	RDP → RD
		OP(RIC) → RI
		RD → MEM[RI]
2	fetch	PC → RI
		MEM[RI] → RD
		RD → RIC
		PC + 1 → PC
	esecuzione	OP(RIC) → RI
		MEM[RI] → RD
		RD → A
3	fetch	PC → RI
		MEM[RI] → RD
		RD → RIC
		PC + 1 → PC
	esecuzione	OP(RIC) → RI
		MEM[RI] → RD
		RD → B
4	fetch	PC → RI
		MEM[RI] → RD
		RD → RIC
		PC + 1 → PC
	esecuzione	MUL
5	fetch	PC → RI
		MEM[RI] → RD
		RD → RIC
		PC + 1 → PC
	esecuzione	A → RD
		OP(RIC) → RI
		RD → MEM[RI]

Figura 4.3 Micro-istruzioni corrispondenti alle prime sei istruzioni del programma moltiplicazione

6	fetch	PC → RI
		MEM[RI] → RD
		RD → RIC
		PC + 1 → PC
	esecuzione	OP(RIC) → RI
		MEM[RI] → RD
		RD → RDP
7	fetch	PC → RI
		MEM[RI] → RD
		RD → RIC
		PC + 1 → PC

Figura 4.4 Micro-istruzioni corrispondenti alle ultime due istruzioni del programma moltiplicazione

4.6 Modi di indirizzamento

Finora abbiamo ipotizzato che il campo operando di un'istruzione contenga l'*indirizzo assoluto* della parola di memoria in cui è memorizzato l'operando. Questo modo di indirizzamento dell'operando (e della memoria che lo contiene) è detto *diretto*. Esso richiede alla macchina un accesso alla memoria per il recupero dell'operando e presenta due inconvenienti: innanzitutto la presenza di un indirizzo assoluto rende lo spostamento di un programma più oneroso (perché gli indirizzi assoluti devono essere ricalcolati a ogni spostamento); in secondo luogo, se la memoria è molto grande, i campi indirizzi diventano anch'essi molto lunghi (si ricordi a tale proposito il legame esistente tra la dimensione del campo operando e il numero di parole di memoria indirizzabili). Esistono altri modi di indirizzamento dell'operando coinvolto in un'operazione. In questo paragrafo ne presentiamo tre: quello *indiretto*, quello *relativo a un registro indice* e quello *immediato*.

- Nel caso di *indirizzamento indiretto*[1] (o differito), l'indirizzo effettivo dell'operando è contenuto nella parola di memoria indirizzata dal campo operando dell'istruzione. L'indirizzo effettivo dell'operando può così essere rappresentato in h bit e non solo in n come avviene per l'indirizzamento diretto. Il recupero di un operando richiede però, nel caso di indirizzamento indiretto, un accesso in più alla memoria rispetto a quanto richiesto da un indirizzamento diretto. Infatti nell'indirizzamento indiretto, una volta recuperato l'indirizzo presente nel campo operando dell'istruzione che si sta eseguendo, bisogna accedere alla memoria per il recupero dell'indirizzo dell'operando e accedere nuovamente alla memoria per il recupero dell'operando. È infine importante notare che la cella di memoria che contiene l'indirizzo dell'operando contiene l'indirizzo assoluto dello stesso. Vedremo invece che nel prossimo modo

1. Questo modo di indirizzamento consente l'implementazione di variabili puntatore messe a disposizione del programmatore dai linguaggi di alto livello. Le variabili puntatore sono trattate nel Capitolo 7 con riferimento al linguaggio C.

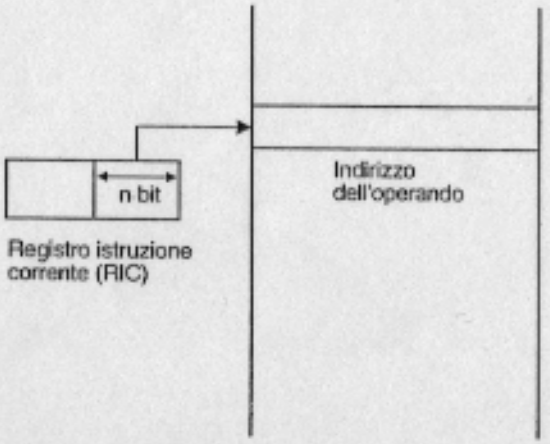

Figura 4.5*a* Indirizzamento diretto

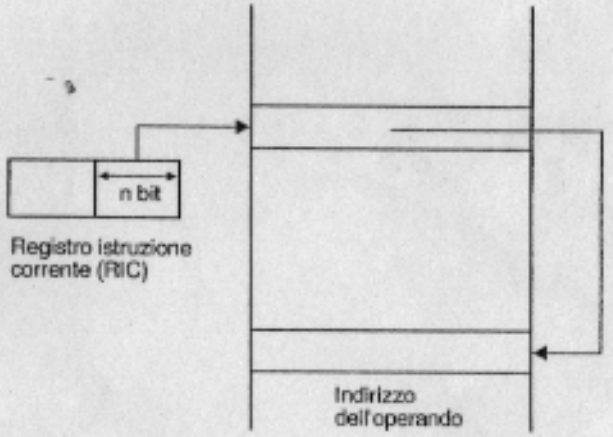

Figura 4.5*b* Indirizzamento indiretto

di indirizzamento l'indirizzo assoluto deve essere calcolato dalla macchina. In Figura 4.5*a* e 4.5*b* sono mostrate rispettivamente le modalità di indirizzamento diretto e indiretto.

- L'indirizzamento *tramite registro indice*[2] consente di accedere a un indirizzo che si ottiene sommando, in modo algebrico, il contenuto di un registro particolare, detto registro indice (I), all'indirizzo presente nel campo operando dell'istruzione. Il

2. Dopo la lettura del Capitolo 7 si ritorni su questo modo di indirizzamento per considerare il legame esistente tra esso e la trattazione di vettori e di variabili indice nei linguaggi di alto livello.

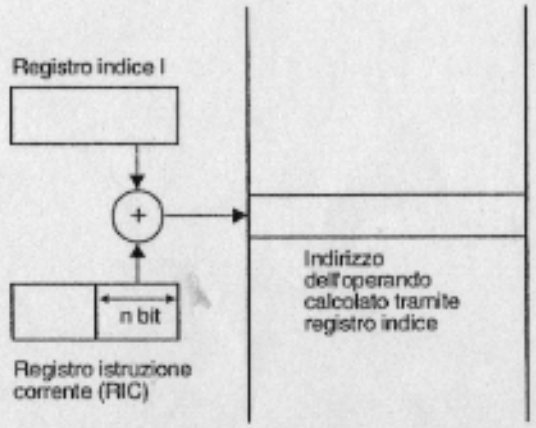

Figura 4.6 Indirizzamento tramite registro indice

registro I è uno dei registri presenti nella CPU. Questo modo di indirizzamento, che richiede un unico accesso in memoria per il recupero dell'operando, è particolarmente utile per accedere alle n parole che seguono una particolare parola di memoria (che chiamiamo parola base). Per ottenere questo effetto è infatti sufficiente ripetere n volte un'istruzione che abbia nel suo campo operando l'indirizzo della parola base, avendo l'accortezza di incrementare il registro I prima della successiva esecuzione dell'istruzione. In Figura 4.6 è mostrata la modalità di indirizzamento a indice. Anche questo modo di indirizzamento consente il riferimento a un numero maggiore di celle della memoria rispetto a quanto consentito dal modo di indirizzamento diretto. L'indirizzo assoluto dell'operando si ottiene infatti sommando il contenuto del registro indice (che potrebbe essere costituito da h bit) al contenuto del campo operando dell'istruzione (composto da n bit).

- L'indirizzamento *immediato* prevede che il valore dell'operando sia già presente nel campo operando dell'istruzione. Non è pertanto richiesto in questo caso nessun ulteriore accesso alla memoria in fase di esecuzione dell'istruzione per il recupero dell'operando. Il valore dell'operando è in questo caso limitato dal numero n di bit riservati al campo operando.

Il Paragrafo 4.8 contiene alcuni esempi di programmi, scritti in linguaggio assemblatore, che usano le modalità di indirizzamento presentate in questo paragrafo.

Le quattro modalità di indirizzamento illustrate devono poter essere riconosciute dall'unità di controllo che deve provvedere a un corretto recupero dell'operando. Per consentire il riconoscimento si potrebbero definire nel linguaggio macchina operazioni con nome diverso (e quindi codice operativo diverso) in corrispondenza di diversi modi di indirizzamento. Questo significherebbe, per esempio, avere a disposizione a livello di linguaggio macchina le seguenti istruzioni.

loada	ind1	se l'indirizzamento è diretto
loaddifa	ind1	se l'indirizzamento è differito (indiretto)

loadinda	ind1	se l'indirizzamento è tramite registro indice
loadimma	ind1	se l'indirizzamento è immediato

Questo comporterebbe però un grande aumento del numero delle istruzioni del linguaggio macchina (analogamente, infatti, si dovrebbero definire più istruzioni storea, loadb, storeb, read e write)

Riprendendo in considerazione il formato delle istruzioni è però possibile adottare una soluzione migliore che consente ugualmente all'unità di elaborazione di riconoscere il modo di indirizzamento coinvolto e di eseguire correttamente il recupero dell'operando. È possibile infatti introdurre, nell'istruzione stessa, una parte che indichi la modalità di indirizzamento dell'operando. Consideriamo quindi le istruzioni in linguaggio macchina divise in tre parti: il codice operativo, la modalità di indirizzamento dell'operando e l'operando. Se s è la lunghezza dell'istruzione, m il numero di bit dedicati al codice operativo, i il numero di bit dedicati alla modalità di indirizzamento dell'operando e n il numero di bit dedicati all'operando (con $s = m + i + n$), i modi di indirizzamento riconoscibili dalla macchina saranno in numero inferiore o uguale a 2^i. Avendo introdotto quattro modi di indirizzamento possiamo supporre che essi vengano rappresentati su due bit nel modo seguente:

00	indirizzamento diretto
01	indirizzamento indiretto
10	indirizzamento tramite registro indice
11	indirizzamento immediato

Se, ad esempio, la prima istruzione del programma moltiplicazione codificato in Figura 4.2 coinvolgesse un indirizzamento tramite registro indice, essa si presenterebbe nel modo seguente (i valori relativi al modo di indirizzamento sono evidenziati in neretto):

0100 **10**00 0000 1000

4.7 Il linguaggio assemblatore

I programmi in linguaggio macchina utilizzano codici operativi e indirizzi in formato binario, come illustrato dalla Figura 4.2; è evidente che leggere un programma e comprenderne il significato pone non poche difficoltà.

Per venire incontro a questi problemi, sono stati inventati linguaggi di programmazione *di alto livello*; con questo termine, si indica l'esigenza di frapporre una grande distanza fra il linguaggio macchina, che opera a diretto contatto della macchina (ad esempio, manipola il contenuto dei registri fisici) e i linguaggi di programmazione (che non dipendono dall'architettura della macchina e quindi dalla presenza dei registri). Tramite i linguaggi di alto livello, il programmatore *astrae* da una serie di dettagli e si concentra sulla produzione di programmi leggibili e corretti.

D'altra parte, quale che sia il livello di astrazione cui si pone l'utente, il codice eseguito è sempre scritto in linguaggio macchina. Tanto maggiore è il livello di astrazione, tanto maggiore sarà la complessità dei programmi che operano la traduzione dal *programma sorgente*, scritto dall'utente in un linguaggio ad alto livello, al *programma oggetto*, scritto in linguaggio macchina. I programmi che operano la traduzione (compilatori e interpreti) fanno parte dell'*ambiente di programmazione*,

discusso brevemente nel Paragrafo 1.5.5.

Tuttavia, è talvolta opportuno programmare in un linguaggio che sia molto vicino al linguaggio macchina. Infatti, i programmi prodotti traducendo linguaggi di alto livello possono essere relativamente inefficienti rispetto a programmi scritti direttamente in linguaggio macchina. L'esigenza di un linguaggio di programmazione di basso livello è soddisfatta dai cosiddetti *linguaggi assemblatori*.

Le istruzioni di un linguaggio assemblatore corrispondono in modo biunivoco alle istruzioni di un linguaggio macchina; pertanto, la capacità espressiva del linguaggio è esattamente la stessa. I linguaggi assemblatori differiscono dai linguaggi macchina per poche ma significative caratteristiche.

- Innanzitutto, i codici delle istruzioni vengono espressi tramite parole chiave; pertanto, è lecito nel linguaggio assemblatore scrivere le istruzioni LOADA oppure WRITE senza dover ricordare la loro codifica binaria.
- In secondo luogo, i riferimenti alle istruzioni del programma o alle celle di memoria contenenti i dati sono fatti tramite etichette, cioè in modo *simbolico*. In questo modo, non è necessario indicare nei programmi specifici indirizzi di celle di memoria; è sufficiente fare riferimento a etichette poste nel programma in corrispondenza ad alcune sue posizioni.
- Infine i modi di indirizzamento che il programmatore utilizza vengono espressi in modo simbolico indipendentemente dalla loro codifica binaria.

In un programma scritto in linguaggio assemblatore potremmo trovare quindi le seguenti istruzioni (NUM è l'etichetta associata a un cella di memoria contenente un dato):

```
LOADA     NUM
```

che provoca l'assegnamento al registro A del valore contenuto nella cella etichettata con NUM (il modo di indirizzamento coinvolto è quello diretto)

```
LOADA     @NUM
```

che provoca l'assegnamento al registro A del valore contenuto nella cella di memoria il cui indirizzo è contenuto nella cella etichettata con NUM (il modo di indirizzamento coinvolto è quello indiretto)

```
LOADA     NUM(I)
```

che provoca l'assegnamento al registro A del valore contenuto nella cella di memoria il cui indirizzo si ottiene sommando al valore NUM il valore contenuto nel registro I (il modo di indirizzamento coinvolto è quello tramite registro indice).

```
LOADA     #NUM
```

che provoca l'assegnamento al registro A del valore dell'etichettata NUM (il modo di indirizzamento coinvolto è quello immediato).

La traduzione in linguaggio macchina delle precedenti istruzioni è la seguente (assumendo che l'etichetta NUM corrisponda all'indirizzo 9)

```
LOADA     NUM:     0000  0000  0000  1001
LOADA     @NUM:    0000  0100  0000  1001
LOADA     NUM(I):  0000  1000  0000  1001
LOADA     #NUM:    0000  1100  0000  1001
```


La traduzione in linguaggio macchina delle istruzione riporta il codice binario dell'indirizzo 9 nei 10 bit meno significativi, il codice binario del rispettivo modo di indirizzamento nei due bit successivi e il codice binario dell'operazione LOADA (0000) nei quattro bit più significativi.

La Figura 4.7 illustra la versione del programma moltiplicazione in linguaggio assemblatore. Si noti l'uso delle parole READ, LOADA, LOADB, MUL, STOREA, WRITE e HALT per rappresentare le varie istruzioni; questo uso era stato fatto anche nel linguaggio macchina ma in modo improprio. Inoltre, X e Y sono le etichette corrispondenti a due celle di memoria, che vengono referenziate nel programma.

Un programma scritto in linguaggio assemblatore, prima di poter essere eseguito, deve essere tradotto in linguaggio macchina; questa funzione è svolta da un traduttore specifico, detto *assemblatore*, che trasforma le parole chiavi in codici operativi e le etichette in indirizzi.

Si noti che l'uso di nomi simbolici come etichette, al posto degli indirizzi assoluti o relativi delle celle di memoria, facilita il caricamento del programma in una generica posizione della memoria, e non necessariamente in una posizione fissa. Questa operazione di caricamento prende il nome di *rilocazione* e viene effettuata dal sistema operativo (si veda il Capitolo 12).

4.7.1 Variabili e costanti

Le etichette assegnate alle celle di memoria devono ovviamente essere tutte distinte; quando usiamo un'etichetta facendo riferimento a una cella di memoria della zona dati, in realtà diamo un *nome simbolico* a una *variabile* del programma. In generale, in un qualsiasi linguaggio di programmazione i dati di un programma vengono classificati in *costanti* (che non cambiano durante la computazione) e *variabili* (che possono cambiar valore); ciascuna costante o variabile ha un proprio *tipo* (ad esempio, intero oppure reale) e un proprio *nome simbolico*.

Nell'esempio di Figura 4.7, i nomi simbolici X e Y corrispondono perciò a due variabili di tipo intero. A X e Y corrispondono anche due celle di memoria predispo-

```
        READ    X
        READ    Y
        LOADA   X
        LOADB   Y
        MUL
        STOREA  X
        WRITE   X
        HALT
X       INT
Y       INT
```

Figura 4.7 Programma moltiplicazione in linguaggio assemblatore

ste per contenere numeri interi. In questo esempio, la corrispondenza fra variabili e celle di memoria è biunivoca (a ogni variabile corrisponde una cella di memoria e viceversa); in generale, questa corrispondenza può essere più complessa (possono ad esempio essere necessarie più celle di memoria per una stessa variabile). Nel seguito, vedremo anche celle di memoria riservate per contenere delle costanti, cioè dei valori che non cambiano durante la computazione.

4.7.2 Schemi a blocchi

Per illustrare la logica di programmi più complessi, è opportuno darne una prima rappresentazione tramite uno *schema a blocchi*. Lo schema a blocchi è un primo, rudimentale strumento per descrivere un *algoritmo*, cioè una procedura che porta alla soluzione di uno specifico problema applicativo. Il concetto di algoritmo, che qui ritroviamo utilizzato, è stato introdotto nel Capitolo 1 e verrà ripreso nel prossimo capitolo.

In questo testo, lo schema a blocchi è soprattutto visto come strumento per descrivere il flusso di esecuzione di un programma e le manipolazioni che esso opera sulle variabili, piuttosto che come strumento per descrivere algoritmi. Lo schema a blocchi si pone a un livello più *astratto* del linguaggio assemblatore perché opera sulle variabili prescindendo dal modo in cui le variabili vengono manipolate a livello di registri; i blocchi rappresentano quindi operazioni logiche (sulle variabili) e non fisiche (sui registri). Gli elementi costitutivi degli schemi a blocchi sono rappresentati in Figura 4.8.

- Il blocco di *assegnamento* assegna a una variabile il risultato di un'espressione. Una freccia da destra a sinistra separa la variabile (a sinistra) dall'espressione (a destra).
- Il blocco di *test* contiene un'espressione logica; il risultato della valutazione dell'espressione è un valore booleano (*vero* o *falso*). Il diagramma a blocchi ha due vie di uscita, etichettate con *sì* e *no*; l'esecuzione prosegue sul ramo *sì* se la valutazione dell'espressione da il risultato *vero*, sul ramo *no* se il risultato è *falso*. Si noti che nel nostro semplice linguaggio assemblatore non siamo in grado di effettuare tutti i

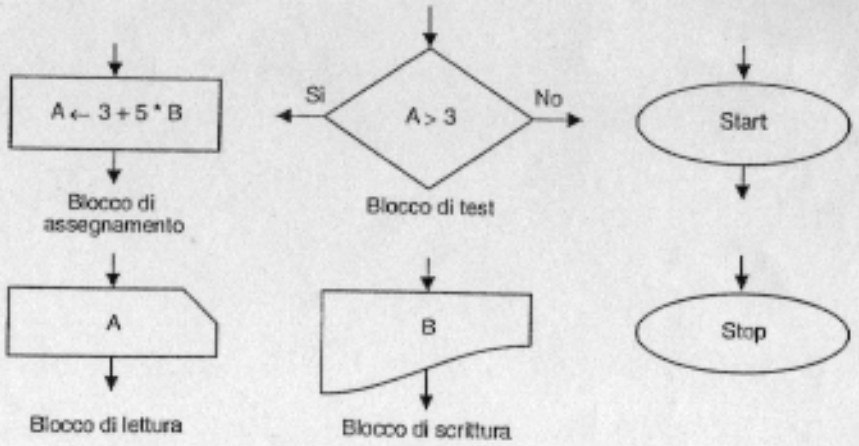

Figura 4.8 Elementi degli schemi a blocchi

possibili confronti, ma solo quelli di uguaglianza. Il confronto tra due dati numerici (variabili e constanti) avviene caricandoli nei registri A e B, valutando la loro differenza, e eseguendo l'istruzione jumpz; se i due numeri sono diversi il programma prosegue con l'istruzione successiva, altrimenti prosegue con un salto.

- Il blocco di *lettura* assegna a una variabile un valore in input (digitato sulla tastiera di una periferica).
- Il blocco di *scrittura* presenta sull'output (ad esempio, sul video di un terminale) il valore contenuto in una variabile.
- Il blocco di inizio (start) e quello di termine (end) della computazione delimitano lo schema a blocchi.

4.8 Esempi di programmi

Vediamo due esempi di piccoli programmi scritti nel nostro linguaggio assemblatore: il *calcolo di moltiplicazioni come sequenze di somme* (Esempio 4.1) e l'*inversione di una sequenza di numeri* (Esempio 4.2).

Esempio 4.1 Moltiplicazione come sequenza di somme

Questo programma rappresenta una variazione sul tema rispetto al programma già discusso. In esso, si calcola nuovamente il prodotto di due numeri (supposti positivi), ma per ragioni didattiche si suppone che il calcolatore non sia in grado di eseguire il prodotto; pertanto, la moltiplicazione viene eseguita come un'opportuna sequenza di somme. Lo schema a blocchi del programma è illustrato in Figura 4.9.

Il programma utilizza tre variabili: X, Y e Z. X e Y, come in precedenza, contengono i valori in input al programma. Al termine del programma, Z conterrà il valore di output prodotto dal programma: sarà Z = XY. Per ottenere il risultato, è necessario incrementare Z di Y ad ogni iterazione di un ciclo che deve essere eseguito esattamente X volte. Per questo motivo, X viene decrementata di uno a ogni iterazione; la condizione di fine ciclo è che X sia uguale a zero.

Il passaggio dallo schema a blocchi al programma richiede di trasformare gli assegnamenti in opportune operazioni numeriche, precedute da operazioni di load e seguite da istruzioni di store; i test devono anche essere trasformati in modo da poter essere eseguiti tramite istruzioni di salto condizionato.

Per la corretta esecuzione di questo programma, è necessario disporre di due costanti: il valore 0, per inizializzare la variabile Z, e il valore 1, per decrementare la variabile X. Tali costanti vengono poste in opportune celle di memoria ed etichettate con le parole ZERO e UNO, che ci ricordano il valore assunto dalle costanti.

Il programma così ottenuto è illustrato in Figura 4.10. Invitiamo il lettore a seguire l'esecuzione del programma passo-passo in corrispondenza a specifici valori di input (ad esempio, 3 e 5) in modo da rendersi conto dell'effetto delle singole istruzioni e dell'intero programma nel suo complesso. Per seguire l'evoluzione del programma, è

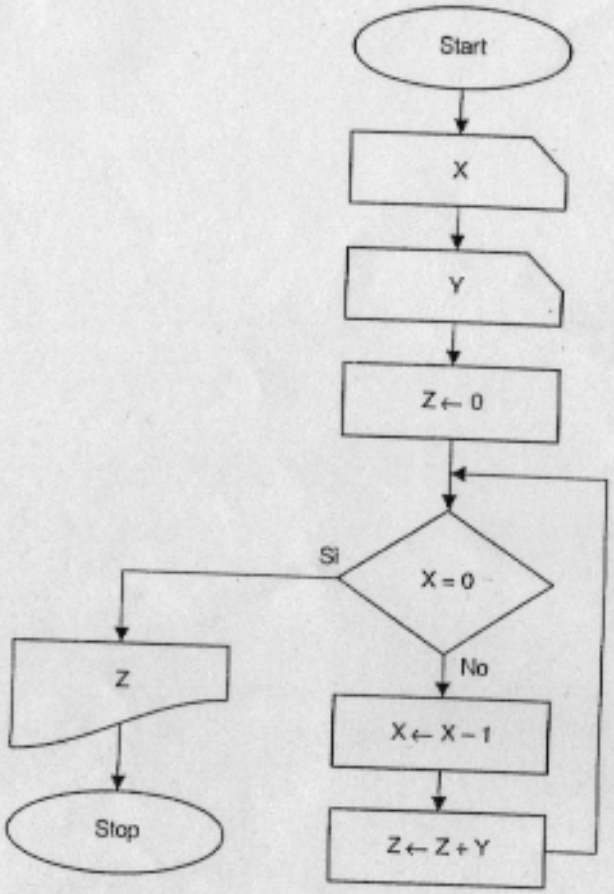

Figura 4.9 Schema a blocchi del programma che calcola il prodotto come sequenza di somme

opportuno seguire l'evoluzione del contenuto delle celle di memoria corrispondenti alle variabili X, Y e Z. È anche possibile tradurre il programma, o alcune sue istruzioni, in una sequenza di micro-istruzioni.

Esempio 4.2 Inversione di una sequenza di numeri

Vogliamo scrivere un programma che legge una sequenza di numeri interrotta da uno zero e li stampa in ordine opposto a quello di lettura.

La prima versione del programma mostra l'utilizzo di modalità di indirizzamento tramite registro indice

Nel programma sono presenti le istruzioni STOREA NUM(I) e LOADA NUM(I) che utilizzano il modo di indirizzamento tramite registro indice. Esse agiscono sulla parola


```
        READ    X
        READ    Y
        LOADA   ZERO
        STOREA  Z
TEST    LOADA   X
        JUMPZ   FINE
        LOADB   UNO
        DIF
        STOREA  X
        LOADA   Y
        LOADB   Z
        ADD
        STOREA  Z
        JUMP    TEST
FINE    WRITE   Z
        HALT
ZERO    0
UNO     1
X       INT
Y       INT
Z       INT
```

Figura 4.10 Programma moltiplicazione ottenuto tramite somme

p di memoria il cui indirizzo si ottiene sommando algebricamente il valore dell'etichetta NUM (corrispondente a un indirizzo) al valore contenuto nel registro I; la LOADA copia nel registro A il contenuto della parola *p*, la STOREA copia nella parola *p* il contenuto del registro A. Nel programma è presente anche l'istruzione LOADI N che assegna al registro I il valore contenuto nella parola di memoria di indirizzo N. LOADI va a estendere l'insieme di istruzioni ammesse per il nostro linguaggio assemblatore la sua codifica in codice binario (quindi la sua espressione in linguaggio macchina) è 1111. Il programma inizia leggendo un numero dal terminale; seguono poi due cicli:

- nel primo ciclo i numeri vengono letti e disposti in memoria in successione, nelle n celle che seguono la cella NUM; il ciclo termina se il valore letto è zero;
- nel secondo ciclo gli *n* numeri vengono scritti partendo però dall'ultima cella e risalendo verso la prima;
- la cella NUM contiene il numero appena letto o il numero da scrivere; la cella N contiene il valore del contatore che viene incrementato durante il ciclo di lettura e decrementato durante il ciclo di scrittura.

Il diagramma a blocchi del programma è illustrato in Figura 4.11. Si noti che nello schema a blocchi le operazioni di lettura e scrittura fanno riferimento a un indirizzo che si ottiene come risultato di un'operazione di somma tra l'indirizzo di una cella (fisica) di memoria e il contenuto (variabile) del registro I.

Il programma corrispondente allo schema a blocchi di Figura 4.11 è illustrato in Figura 4.12. Si prevede che *NN* celle dopo la cella NUM siano predisposte per

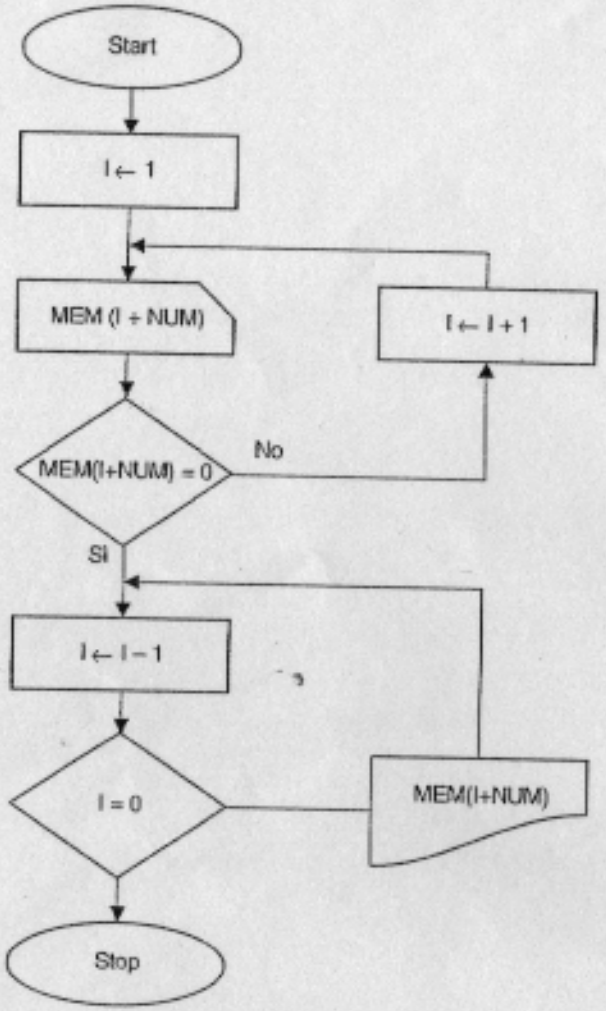

Figura 4.11 Schema a blocchi che inverte una sequenza di *N* numeri

accogliere numeri interi; perciò il programma funziona correttamente solo se il numero di interi letti è inferiore a *NN*. Nel primo ciclo (ciclo di lettura) i dati vengono letti nella cella NUM e successivamente trasferiti nelle celle immediatamente successive a NUM, in modo progressivo; il ciclo è interrotto dalla lettura di uno zero. Nel secondo ciclo (di scrittura) i dati vengono trasferiti nella cella NUM a partire dall'ultimo e "risalendo" verso il primo, così producendo la sequenza inversa. La variabile N contiene il numero di dati letti dal programma e viene progressivamente incrementata nel primo ciclo e decrementata nel secondo; essa corrisponde alla variabile I nello schema a blocchi di Figura 4.11. Come già osservato, se il linguaggio fosse più potente si potrebbero evitare inutili trasferimenti. In particolare, sarebbe utile per questo


```
        LOADA   UNO
        STOREA  N

CICLO1  READ    NUM
        LOADA   NUM
        JUMPZ   CICLO2
        LOADI   N
        STOREA  NUM(I)
        LOADA   N
        LOADB   UNO
        ADD
        STOREA  N
        JUMP    CICLO1

CICLO2  LOADA   N
        LOADB   UNO
        DIF
        JUMPZ   FINE
        STOREA  N
        LOADI   N
        LOADA   NUM(I)
        STOREA  NUM
        WRITE   NUM
        JUMP    CICLO2

FINE    HALT

UNO     1
N       INT
NUM     INT
NUM+1   INT
....
NUM+NN  INT
```

Figura 4.12 Prima versione del programma inversione di *N* numeri

esempio estendere il set di istruzioni con l'incremento o il decremento unitario del contenuto del registro indice.

La seconda versione del programma mostra l'utilizzo di modalità di indirizzamento indiretto e immediato

In Figura 4.13 è illustrata la nuova versione del programma. La parola di memoria etichettata con X viene inizializzata con il valore (costante) NUM+1[3] corrispondente

3. Si noti che la stringa NUM + 1 indica un valore costante e non un'espressione aritmetica.

```
CICLO1    READ     NUM
          LOADA    NUM
          JUMPZ    CICLO2
          STOREA   @X
          LOADA    X
          LOADB    #1
          ADD
          STOREA   X
          JUMP     CICLO1

CICLO2    LOADA    X
          LOADB    #(NUM+1)
          DIF
          JUMPZ    FINE
          LOADA    X
          LOADB    #1
          DIF
          STOREA   X
          WRITE    @X
          JUMP     CICLO2

FINE      HALT

X         NUM+1
NUM       INT
NUM+1     INT
....
NUM+NN    INT
```

Figura 4.13 Seconda versione del programma inversione di *N* numeri

all'indirizzo della prima parola di memoria che conterrà i valori letti. La memorizzazione dell'ultimo numero letto e la stampa del valore che si sta considerando nella scansione sfruttano pertanto il modo di indirizzamento indiretto appoggiandosi sulla parola etichettata con X.

```
STOREA   @X
```

memorizza nella cella il cui indirizzo è contenuto in X il valore contenuto in A questo provoca la memorizzazione del valore contenuto in A nella parola di indirizzo NUM+1. L'incremento graduale subito da X consente l'assegnamento delle celle successive con i numeri letti in ingresso.

L'esempio non utilizza, come quello precedente, una cella N per il conteggio degli elementi letti. Durante il secondo ciclo, per sapere se sono stati stampati tutti i numeri letti nel primo ciclo, vengono confrontati il valore di X con l'indirizzo della parola contenente il primo numero letto (NUM+1). Quando X torna a valere NUM+1 il ciclo di stampa termina.

L'esempio utilizza anche il modo di indirizzamento immediato

```
LOADB  #(NUM+1)   carica nel registro B il valore costante NUM+1.
```

mentre

```
LOADB  #1         carica nel registro B il valore costante 1
```

Esercizi

4.1 Si descriva, in base alla struttura dei registri dell'unità di elaborazione descritti nel capitolo precedente, la modalità di esecuzione delle seguenti istruzioni:

- jumpgtz ind1, salto all'indirizzo ind1 condizionato dalla presenza di un numero positivo nel registro A;
- jumpgez ind1, salto all'indirizzo ind1 condizionato dalla presenza di un numero positivo o nullo nel registro A.

4.2 Spiegare come, con un linguaggio dotato di istruzioni con 3 operandi, è possibile eseguire l'istruzione jumpgt ind1 ind2 ind3, che provoca il salto all'indirizzo ind3 se il numero contenuto all'indirizzo ind1 è più grande del numero contenuto all'indirizzo ind2.

4.3 Modificare il programma che inverte *N* numeri in modo da arrestare la sua esecuzione se *N* > *NN*, ove *NN* indica il numero di celle di memoria riservate dopo la cella X.

4.4 Scrivere un programma che svolge la divisione di due numeri interi; il programma si arresta senza calcolare la divisione se il secondo operando è nullo.

4.5 Scrivere un programma che acquisisce 10 numeri interi dal terminale e ne stampa la somma.

4.6 Scrivere un programma che acquisisce un numero variabile di numeri interi dal terminale e ne stampa la somma. L'immissione dei dati termina quando l'utente immette uno zero.

4.7 Scrivere un programma che acquisisce un numero variabile di numeri interi dal terminale e stampa la somma dei soli numeri pari. Per determinare se un numero è pari, si utilizzi il resto di una divisione per 2. L'immissione dei dati termina quando l'utente immette uno zero.

4.8 Scrivere un programma che esegue un prodotto come sequenza di somme, assumendo in input due numeri interi anche negativi. Si supponga disponibile l'istruzione jumpgtz descritta nell'Esercizio 4.1.

4.9 Scrivere un programma che trova il massimo di 10 numeri, letti dal terminale. Anche per questo programma, si suppongano disponibili le istruzioni jumpgtz e jumpgez descritte nell'Esercizio 4.1.

4.10 Progettare un linguaggio assemblatore in cui le istruzioni hanno tre indirizzi (ad esempio, sum ind1 ind2 ind3, che esegue la somma dei numeri memorizzati nelle celle ind1 e ind2, depositando il risultato nella cella ind3). Mostrare le micro-istruzioni corrispondenti a ciascuna istruzione del linguaggio e riprogrammare alcuni degli esercizi precedenti con questo linguaggio più potente.