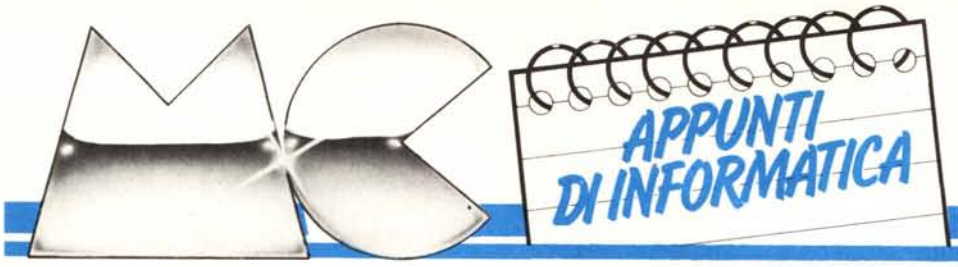

di Andrea de Prisco

Istruzioni, Registri, Operandi

Col termine linguaggio macchina, convenzionalmente, si usa indicare il linguaggio di programmazione «vero» dei computer. I più preparati sanno infatti che il Basic di qualsiasi Personal computer, sebbene «residente» in memoria, è completamente simulato via software dall'interprete contenuto nelle rom di sistema: «I Processori, si sa, capiscono solo in termini di 0 e 1 dei codici di linguaggio macchina...» E se vi dicessimo che spesso ciò non è vero, cosa rispondereste? Sapevate che a sua volta anche il più crudo dei linguaggi macchina può non essere eseguito direttamente dal Processore, ma ha bisogno di una ulteriore interpretazione da parte di un sottostante livello detto di microprogrammazione? Questo l'argomento scottante del prossimo mese: su questo numero, come introduzione all'argomento, solo un piccolo sguardo alle istruzioni, ai registri e agli operandi tipici del livello convenzionale di macchina.

Il linguaggio macchina

8 bit, 16 bit, z80, 68000, PDP-11: come vedete in quanto a numeri, non possiamo proprio lamentarci. E in effetti, un calcolatore calcola, e i calcoli si fanno coi numeri, salvo poche eccezioni di stampo lievemente più intelligente (anche se artificiale) del calcolo dei predicati dell'elaborazione dell'informazione non numerica nei sistemi esperti.

Tranquilli, non stiamo per spiccare il volo: è troppo presto. Restiamo ancora un po' vicino ai calcolatori tradizionali per vedere il linguaggio macchina di queste bestioline. Senza ovviamente fare un corso di Assembler, né puntare la nostra attenzione, come è consuetudine di «Appunti», su un particolare processore esistente.

Sappiamo infatti che ogni unità di elaborazione programmabile ha un proprio linguaggio macchina col quale è possibile specificare la sequenza di istruzioni da compiere per svolgere le funzioni volute. Istruzioni, dal canto loro, tutte piuttosto semplici: solo combinandole opportunamente con gli altrettanto semplici meccanismi di controllo, e possibile programmare tutto ciò che si desidera, ovviamente a patto che sia calcolabile.

Chi programma in Basic conoscerà le stringhe, le funzioni scientifiche, magari la doppia precisione: in linguaggio macchina non esiste niente di tutto ciò.

Eppure in Assembler si fanno i compilatori di linguaggi di livello ben più alto del Basic: senza contare che un programma scritto interamente in linguaggio macchina può anche essere migliaia di volte più veloce dello stesso programma scritto in Pascal, Algol o Fortran.

Il punto è che programmare in Assembler è assai più arduo che in qualsiasi altro linguaggio. Ciò essenzialmente perché il linguaggio macchina risente appunto della «macchina» che vogliamo programmare. Non possiamo ignorare la sua architettura interna, come funzionano e soprattutto come interagiscono le varie unità che la compongono, non senza avere una discreta conoscenza dell'aritmetica binaria, anche se, come più volte detto, rimane così simile a quella decimale (in termini più precisi «isomorfa») che basta avere solo le idee chiare sulle 4 operazioni insegnateci alle elementari sui numeri naturali, per fare fronte a qualsiasi situazione anche in base due.

Oltre a questo, quando abbiamo scritto il nostro bravo programmino in Assembler, e mandatolo in esecuzione non otteniamo il voluto, non aspettiamoci nemmeno messaggi d'errore da parte del computer, dovremo sbrigarcela da soli, cercando un po' qua, un po' là, la causa del fallimento.

Bit, Bite, parole di memoria

Se un hobbynformatico dice a un altro: «sai, la Motorola ha fatto un nuovo Processore, il QT 54321...» potremmo ben scommettere che la prima domanda che gli sarà posta dal collega sarà: «A quanti bit?». Pare infatti che le CPU si misurano a bit: 8, 16, 32. Nessuno chiederà mai, di primo acchito, la Performance, misurata in milioni di operazioni al secondo, né la sua architettura interna, che certamente non è da meno in quanto a influenza sulle prestazioni.

Fra l'altro, dire che un processore è a 16 bit spesso non significa quello che vorremmo. In altre parole, cosa del nostro Processore, o meglio, del nostro calcolatore è a 16 bit?

Potrebbe essere di tale formato il Bus di indirizzamento memoria, i registri interni, le singole celle di memoria, il Bus dati. O tutto ciò insieme? Procediamo con ordine. Innanzitutto il Bus di indirizzamento è strettamente legato, come numero di bit, alla quantità di memoria di cui si dispone. Quindi 16 bit di tale Bus significa poter indirizzare solo 65536 celle (2 alla 16).

Il Bus dati, di contro, è misura delle dimensioni delle celle di memoria, in altre parole quanti bit di memoria vengono trasferiti in seguito a un solo accesso.

Chi storcerà il naso lo farà semplicemente per colpa di questa bizzarra rivoluzione informatica domestica (del registratore e del joystick) che sta invadendo questi tempi moderni: ebbene sì, non sta scritto da nessuna parte che le memorie dei calcolatori sono, per contratto, fatte da byte di 8 bit l'uno. Possono avere le singole celle di 16 bit, di 32, di 64, di un solo bit, così come possono disporre di formati variabili a seconda delle applicazioni. Uffa!

La convenzione comunque è che celle di 8 bit sono dette byte, se in formato minore nibble (4 bit), se in formato maggiore Parole o più comunemente Word. Esistono poi visioni più ibride in cui la memoria è comunque indirizzabile a byte anche se è possibile accedere simultaneamente a 2 o più celle consecutive (nel caso di due specificando sempre indirizzi pari, (fig. 1)) avendo così al contempo una visione a Word.

...oppure possono essere a 16 bit i registri interni, come dicevamo prima: di questo ne parleremo tra poco.

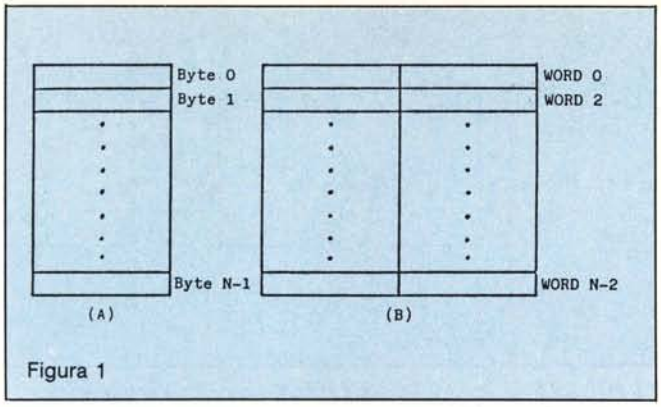

Figura 1

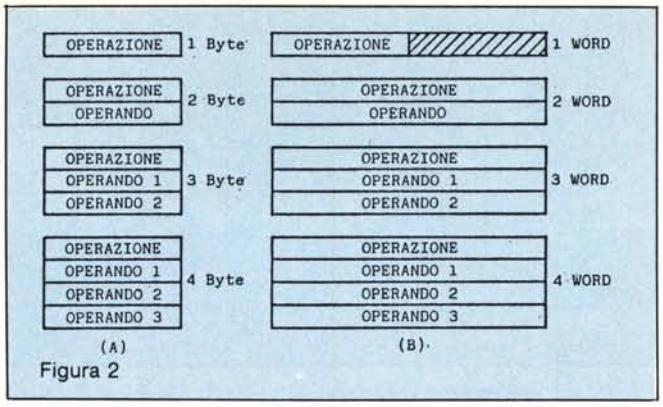

Figura 2

Figura 1 - Memoria organizzata a Byte (A) e a word di due Byte l'una (B). Figura 2 - Occupazione memoria a seconda del formato istruzione. (A) memoria a Byte, (B) memoria a word.

Operazioni & operandi

Entriamo ora nel cuore del problema: il linguaggio. Abbiamo già detto che le istruzioni con cui si programmano i processor sono abbastanza semplici. Infatti non andranno molto oltre la somma di due numeri, qualche operazione sui bit di una cella, spostare il contenuto di una cella di memoria in un'altra cella e poco altro. Volendo azzardare una classificazione delle istruzioni di un generico linguaggio macchina avremo un certo numero di operazioni per accedere alla memoria, operazioni per scrivere in memoria, operazioni aritmetiche, di test su celle di memoria e di salto condizionato e non, così come di chiamata e sottoprogramma.

Il formato di una generica istruzione sarà dunque del tipo

```
OP op1,...,opN
```

dove OP è l'operazione da compiere e op1...opN gli operandi su cui e seguire l'operazione di cui sopra. L'enne di opN ci indica il numero di operandi dell'istruzione: avremo istruzioni a 0 operandi, a un operando così come a due o a tre. Potremo ad esempio sommare 2 numeri con l'istruzione:

```
ADD 13,33
```

anche se in tal caso dovrebbe essere implicito dove mandare il risultato. Oppure disporre di istruzioni che richiedono esplicitamente una cella di memoria per il risultato, esempio:

```
ADD 13,33,$1000
```

in questo caso il risultato è posto nella cella 1000. Altre tipiche istruzioni di linguaggio macchina sono quelle di salto incondizionato (è un esempio di istruzione a un operando):

```
JMP $2000
```

che fa saltare alla cella 2000, dove si suppone sia memorizzata la continuazione del programma. O di salto condizionato:

```
BEQ $2000
```

che fa saltare alla cella 2000 se l'operazione precedente ha dato come risultato zero. Questo comunque lo vedremo meglio quando parleremo della Processus Status Word. Un esempio di istruzione a 0 operandi potrebbe essere un comunissimo:

```
RTS
```

per ritornare da un sottoprogramma attivato dall'istruzione:

```
JSR <indirizzo>
```

mentre, per finire, un'operazione di trasferimento potrebbe essere:

```
MOVE $100, $101
```

che come è facilmente intuibile nel nostro caso sposta il contenuto della cella 100 nella cella 101.

In figura 2 sono mostrati possibili modi di memorizzare le istruzioni di linguaggio macchina nel caso di memoria indirizzabile a byte e a word di due byte l'una. Ad esempio possiamo mettere il codice operativo di una istruzione a zero operandi in un byte o in mezza word (in tal caso l'altra mezza risulterebbe sprecata); se abbiamo un'istruzione a un solo indirizzo possiamo occupare due celle contigue, nella prima metteremo il codice operativo nella seconda l'indirizzo; mentre per istruzioni a più indirizzi possiamo occupare un numero maggiore di celle, come prima una per il codice e le altre per gli operandi.

Registri generali e strutture dati

Detto questo addentriamoci maggiormente nel merito, illustrando le strutture dati disponibili in linguaggio macchina. Ogni processore dispone infatti di un certo numero di registri, una struttura dati LIFO detta Stack più una manciata di registri di uso più particolare che vedremo nel prossimo paragrafo.

I registri di uso generale servono principalmente per non scomodare di continuo la memoria del calcolatore (che seppur dell'ordine di milionesimi di secondo ha tempi d'accesso tutt'altro che trascurabili) per le «variabili di comodo» usate dai programmi. Supponiamo ad esempio di dover scambiare il contenuto di due locazioni di memoria: in un qualsiasi linguaggio di programmazione che non dispone di un tale comando, come è noto, ci occorrerà una variabile temporanea per effettuare lo scambio: potremmo usare ad esempio (caso sconsigliato) un'altra cella di memoria, la 1000.

Scambiamo allora la cella 1111 con la cella 2222:

```
MOVE $1111, $1000
MOVE $2222, $1111
MOVE $1000, $2222
```

effettuando la bellezza di 6 accessi in memoria. Usando un registro interno al processore (qui li indicheremo con r0...rN) possiamo risparmiare due accessi in memoria, risparmiando così sul tempo totale dell'intera operazione:

```
MOVE $1111, r0
MOVE $2222, $1111
MOVE r0,    $2222
```

Se qualche milionesimo di secondo in più o in meno vi fa sorridere, non dimenticate che il tempo perso è di solito cumulabile, quindi posto di dover scambiare centomila celle di memoria tra loro, centomila milionesimi di secondo in alcuni casi possono anche farsi notare.

Oltre a questo, useremo i registri interni anche per le operazione aritmeti-

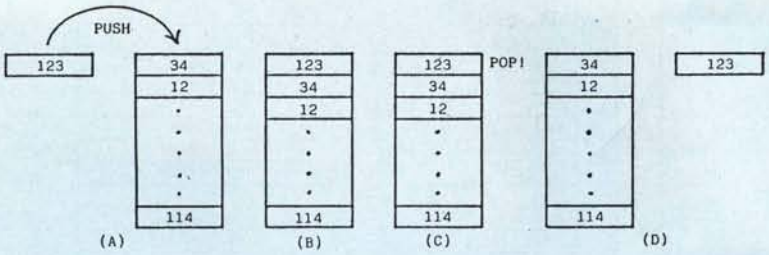

Figura 3

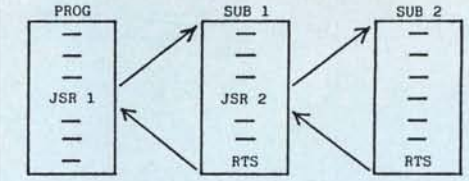

Figura 4

Figura 3 - (A) inserimento di un elemento, (B) stato dello Stack dopo (A), (C) richiesta di POP, (D) Stack dopo (C). Figura 4 - Chiamate nidificate di subroutine.

che, specialmente quando il nostro processore non dispone di istruzioni a più di un operando: in tale caso, si parla di registro Accumulatore. Spieghiamoci meglio: immaginiamo di dover sommare il contenuto di due celle di memoria e scrivere il risultato del calcolo in una terza cella, ad esempio $1001+$1002 in $1003. Come detto, non possiamo specificare più di un operando: le nostre istruzioni riferiranno implicitamente al registro accumulatore. Scriveremo:

```
LDA $1001
ADD $1002
STA $1003
```

La prima istruzione (LoaD in Acc.) carica nell'accumulatore la cella di memoria $1001, la seconda gli somma il contenuto della cella $1002 e la terza (STore Acc.) memorizza l'accumulatore nella cella $1003.

Passiamo allo Stack. Come abbiamo già detto, esso è una struttura dati con politica LIFO. Tale acronimo sta per Last In First Out che letteralmente vuol dire: il primo ad entrare è l'ultimo ad uscire. Pensate a una pila di piatti: dovendo aggiungere un piatto lo metteremo in cima, così come per prenderne uno prenderemo quello più in alto: il primo dei piatti posto in pila, sarà l'ultimo ad essere usato (se mai ci ridurremo ad aver sporcato tutti gli altri). In un calcolatore lo stack (fig. 3) funziona nello stesso modo e serve per parcheggiare momentaneamente dati che non useremo subito e ci servono un po' di registri liberi per compiere nuove operazioni. Infileremo i contenuti dei registri nello stack per poi riprenderli a tempo debito.

Esisteranno di conseguenza due operazioni, normalmente denominate PUSH e POP, che permettono di mettere qualcosa nello stack e di riprenderlo. Facciamo un esempio: ci servono liberi i registri r1, r2, r3, scriveremo:

```
PUSH r1
PUSH r2
PUSH r3
```

che mettono in «pila» i tre registri nell'ordine indicato. Nella fattispecie, in cima allo stack c'è l'ex contenuto di r3. Per ripristinare i registri al loro stato prima dei PUSH, scriveremo:

```
POP r3
POP r2
POP r1
```

si noti come l'ordine di recupero risulta capovolto a causa proprio del fatto che... «l'ultimo ad entrare sarà il primo ad uscire».

A questo punto una domanda: perché politica LIFO?

Sembrerebbe infatti che per salvare registri momentaneamente non serva tale convenzione, ed è vero. Si usa LIFO a causa del meccanismo dei sottoprogrammi che per natura hanno un comportamento di questo tipo. In figura 4 è mostrato un programma che a un certo punto chiama la subroutine 1 all'interno della quale vi è una chiamata alla subroutine 2. Immaginiamo allora che nel programma, prima di ogni chiamata sottoprogramma si salvano sullo stack i tre registri di prima, per poi ripristinarli non appena si ritorna: ciò avverrebbe, nel nostro esempio, in due punti: nel programma principale, in occasione del JSR 1 e nella subroutine 1, presso il JSR 2. È ovvio che quando ripristiniamo al ritorno da Sub 2, dobbiamo immettere in r1, r2, r3 gli ultimi tre valori da essi denotati e non quelli del programma principale, che saranno «ricatturati» a tempo debito: al ritorno da Sub 1.

Registri speciali

Oltre ai registri di uso generale, ogni CPU possiede almeno altri tre registri di uso più particolare: lo Stack Pointer, il Programma Counter e la Processus Status Word. Il primo di questi, come dice il suo nome, serve per implementare lo stack in una qualsiasi zona di memoria. Infatti, il togli e metti di cui sopra, è realizzato dal processore usando un pezzo di memoria come stack e un apposito registro che indica quale cella di memoria corrisponde alla testa di tale struttura dati: quale è l'ultimo elemento «push-ato».

Quando faremo un inserimento incrementeremo lo stack pointer e occuperemo la cella puntata da questo: diversamente, per l'operazione di pop, preleveremo l'elemento puntato per poi decrementare lo stack pointer che punterà all'elemento precedente. Semplice, no?

Il Program Counter, più semplicemente, contiene l'indirizzo della prossima istruzione di macchina da eseguire: esso è automaticamente incrementato dopo aver prelevato l'istruzione e viene modificato dal processore quando occorre saltare da un punto all'altro del programma in seguito a un GOTO (condizionato e non), GOSUB o RETURN da subroutine. In altre parole, il Programm Counter contiene costantemente il prossimo indirizzo da inviare alla memoria per ricevere da questa la cella contenente l'istruzione da eseguire.

La Processus Status Word (per semplicità PSW, mostrata in fig. 5), riassume lo stato del processo in corso per quanto riguarda alcune situazioni. Per esempio, se sommando due numeri otteniamo come risultato 0 il bit marcato Z si porrà a 1. Analogamente se l'ultima operazione effettuata ha dato come risultato un numero negativo o vi è stato un overflow aritmetico (la somma di due numeri ha superato come risultato la capacità di un registro o di una cella di memoria) si setteranno rispettivamente i bit N o V della PSW. Infine si setterà il bit C se c'è stato un riporto nell'ultima somma o il bit I se c'è stato un Interrupt o altre cose del genere a seconda del caso.

Insomma, ispezionando la PSW si può controllare un po' di roba, eventualmente prendendo le decisioni del caso. Infatti le istruzioni di salto condizionato non fanno altro che accedere ai bit della PSW, avendo a seconda di questa un comportamento o un altro: ad esempio, col BEQ $2000 visto prima il processor non fa altro che controllare il bit Z e se questo è 1 salta a $2000 (mettendo $2000 nel pro-

gramm counter) o procedendo l'elaborazione se Z=0. Alla stessa maniera avremo istruzioni per saltare se N=0 o N=1 (BGT o BNE), se V=0 o V=1 (BVC o BVS) e se C=0 o C=1 (BCC o BCS).

Modi di indirizzamento

Finora abbiamo parlato di operandi di istruzioni senza fare differenza tra numeri, indirizzi di memoria o cose più complicate. Tale argomentazione riguarda i modi di indirizzamento di cui un processore dispone, che ne fanno per l'appunto una macchina più o meno flessibile e potente. In questa sede vedremo 7 modi di indirizzare dati che rappresentano il minimo indispensabile per non fare salti mortali coi giri e rigiri di una programmazione contorta.

Il primo modo di indirizzare un operando di una istruzione è l'indirizzamento «immediato»: l'istruzione dispone subito del suo operando, senza andarselo a pescare chissaddove. È l'esempio tipico, già visto, delle costanti. Ad esempio:

```
ADD 13,33
```

che esegue la somma del numero 13 e del numero 33. Se invece scriviamo:

```
ADD $1000, $1001
```

l'indirizzamento non sarà immediato in quanto i due operandi dovremo prelevarli nelle celle $1000 e $1001: in questo caso si parla di indirizzamento diretto. L'indirizzamento indiretto, di contro, prevede un ulteriore livello di ricerca dell'operando. Scrivendo:

```
JMP ($4000)
```

intendiamo saltare alla cella di memoria indicata nella cella $4000. Ciò significa che per eseguire tale istruzione dobbiamo prima di tutto avere «in mano» il $4000, accedere a tale cella, leggere il contenuto di essa e finalmente effettuare il salto all'indirizzo così ottenuto.

Esiste poi l'indirizzamento implicito, che riguarda istruzioni che implicitamente si riferiscono a un particolare registro, ad esempio:

```
PHA
```

del microprocessore 6502 (e gentile famiglia) inserisce nello stack il contenuto dell'accumulatore (detto anche registro A).

Per quanto riguarda gli indirizzamenti relativi a insiemi di celle cui fare capo all'interno di un loop con un indice, esistono alcuni modi di indirizzamento indiciato. Il primo, indiciato e basta, si effettua indicando un indirizzo di base più un registro che contiene il Displacement, ad esempio:

```
MOVE $1000+r0,r1
```

mette nel registro r1 il contenuto della cella il cui indirizzo è ottenuto sommando $1000 e r0. Così se r0 vale 10 metteremo in r1 il contenuto della cella $1010, se vale 50, la cella $1050 e così via.

Combinazione dei modi precedenti si hanno con gli indirizzamenti indiciato-indiretto e indiretto-indiciato. Anche qui avremo un indirizzo base più un registro: nel primo caso l'indirizzo è calcolato sommando tra loro la base e il displacement. Facciamo un

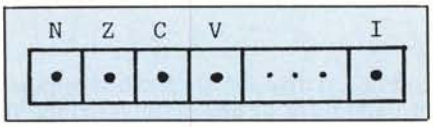

Figura 5 - Processus Status Word.

esempio, abbiamo:

```
MOVE ($1000+r0),r1
```

accediamo come prima alla cella 1000+r0, ad esempio la $1010, per prelevare l'indirizzo effettivo: supponiamo che in $1010 ci sia il valore 2000. La cella $2000 conterrà l'operando cercato, che metteremo in r1 come da istruzione.

Nel secondo caso, l'indirizzamento indiretto-indiciato, si fa prima l'indirezione e poi si somma l'indice per ottenere l'indirizzo finale, sia:

```
MOVE ($1000)+r0,r1
```

si noti la differenza di posizione delle parentesi rispetto al caso precedente. Supponiamo che r0 valga, 10, e la cella $1000 contenga 2000. L'indirizzo finale è ottenuto accedendo alla cella $1000, dalla quale otteniamo 2000, a questo indirizzo sommiamo 10 (r0) ottenendo $2010 il cui contenuto sarà copiato in r1.

Assembler e MacroAssembler

Ovviamente qualche sforzo è stato fatto per rendere meno faticosa la programmazione in linguaggio macchina. Il primo passo è stato quello dei nomi simbolici per indicare indirizzi di salto (le etichette) senza stare a contorcersi il cervello con somme e sottrazioni esadecimali per ottenere l'indirizzo effettivo. Si etichetta un punto del programma con un nome (ad esempio pippo, che è così facile da digitare al terminale) e per saltare lì scriveremo roba del tipo:

```
JMP PIPPO
BNE PIPPO
```

o similari. Poi è venuta la volta dei nomi simbolici anche per le celle che useremo nel programma per i calcoli, in particolar modo per i vettori di celle: indichiamo ad esempio con la parola TOP la locazione $1000. Dopo aver preservato da intrusioni le celle seguenti potremo fare accessi del tipo:

```
MOVE TOP+r0,r1
```

Il passo successivo è segnato dalla comparsa delle macroistruzioni definibili (da non confondere con le microistruzioni del prossimo mese) che permettono di creare nuove istruzioni a partire da istruzioni più semplici e/o da macro già definite. Se, ad esempio, il nostro processore non dispone di una istruzione che azzera il contenuto di una cella o di un registro, potremmo definirla così:

```
MACRO CLR M
      LDA 0
      STA M
      endMacro
```

la prima linea definisce il nome della nuova operazione e su quanti operandi agisce, nel nostro caso 1. La seconda e terza linea compongono il corpo della macro e in particolare cosa si deve fare una volta ottenuto il parametro M (la cella da azzerare): mettiamo in A il numero 0 e poi scriviamo A nella cella passataci. EndMacro. Da questo momento in poi, potremo considerare CLR come una nuova istruzione del linguaggio usandola come ci pare, con qualsiasi modo di indirizzamento. Ad esempio potremo scrivere:

```
CLR $1000
CLR $1203+r0
CLR ($1234)
```

eccetera. Di fatto, però, non è avvenuto nulla di strano: semplicemente l'assemblatore (che si preoccupa di trasformare programmi mnemonici in codice eseguibile dal processore) sostituirà ad ogni occorrenza di CLR, la sequenza di istruzioni specificate nella dichiarazione di Macro, facendo corrispondere ad ogni parametro formale (la M di cui sopra) il parametro attuale con cui si invoca l'espansione macro, Nella fattispecie l'assemblatore, ai tre CLR appena mostrati, sostituirà:

```
LDA 0
STA $1000
LDA 0
STA $1203+r0
LDA 0
STA ($1234)
```

che corrisponde (a meno di una indegna ridondanza) al programma in istruzioni naturali che azzera le tre celle di memoria. Tutto qui.

Ah! dimenticavo: in che cosa consiste «l'indegna ridondanza»? MC