exma

un assemblatore per VIC-20

di Andrea de Prisco

Seconda parte

Nel numero scorso, vi abbiamo presentato un potente assemblatore per il VIC-20 espanso con 16K. Col listato pubblicato, è possibile scrivere programmi in linguaggio macchina sfruttando etichette, notazione decimale, ottale, binaria nonché altre utility per rendere la vita un po' più facile a chi si occupa di questo genere di programmazione. Per semplificare ancora di più le cose, aggiungiamo al nostro assemblatore le macro istruzioni definibili dall'utente.

È il tema di questa puntata. Le linee Basic presenti su questo numero sono da aggiungere e/o sostituire al listato 2 del numero scorso. Loro compito è appunto quello di permettere la creazione di macro nonché, chiaramente, la loro utilizzazione nei programmi. L'assemblatore, tanto per cambiare, provvederà a sostituire ogni istruzione definita dall'utente, con il pacchetto di istruzioni elementari direttamente eseguibili dal microprocessore.

Parametri attuali e parametri formali

Prima di entrare nel merito di macro istruzioni e affini, è bene chiarire alcuni concetti riguardanti il passaggio di parametri. Tanto per restare in intimità, senza quindi scomodare linguaggi assai più evoluti come l'Algol e il Pascal, chiamiamo in causa il caro amico Basic, l'onnipresente.

Oltre alla semplicità d'uso, una delle caratteristiche più interessanti di questo linguaggio è la possibilità di definire funzioni tramite l'istruzione DEFFN. Supponiamo di aver bisogno di una funzione che, preso un qualunque N, restituisca la somma del suo quadrato e del suo doppio.

La definizione avviene col comando:

```
10 DEFFNF(N) = N*N + 2*N
```

Anche se qualcuno non se ne sarà mai accorto, la N che vediamo nella definizione di FNF non è la variabile N, che dal canto suo può tranquillamente esser usata in qualsiasi altra parte del programma. È un parametro formale che serve solo per descrivere la funzione: ciò che si deve fare col dato in ingresso.

Se alla linea 20 scriviamo:

```
20 N = 150:A = FNF(3)
```

dopo l'esecuzione, N conterrà ancora 150 e ad A sarà associato il valore 3*3 + 2*3 (= 15) e non N*N + 2*N come appare nella definizione. Il 3 di FNF(3) è il parametro attuale, quello con il quale viene chiamata la funzione FNF(N).

In Algol e Pascal, la cosa si fa ancora più interessante: la definizione di una funzione può anche essere lunga come un intero programma, e il numero di parametri "passabili" non è limitato a 1 come in Basic.

Quando si ha la possibilità di definire a piacimento procedure e funzioni, anche la programmazione cambia aspetto. Generalmente, risolvere con un programma un problema in Pascal, si riduce essenzialmente a scomporlo in sottoproblemi di minore difficoltà, definendosi facilmente tutte le procedure che interessano e, conseguentemente, limitandosi a scrivere il programma come semplici chiamate di quest'ultime.

Le Macroistruzioni

Programmando in assembler 6502, spesso capita di dover ripetere più volte una stessa sequenza di istruzioni. Tanto per citare qualche caso, l'incremento di un byte con relativo riporto nel byte successivo o semplicemente l'azzeramento di un determinato byte, sono sequenze, seppur molto brevi, praticamente onnipresenti in programmi in linguaggio macchina. Ed è un vero peccato che non siano disponibili al livello hardware del microprocessore.

Per non parlare poi di casi leggermente più raffinati, come la copia di un byte in un altro o lo scambio dei contenuti di due celle di memoria o la moltiplicazione 8 × 8 bit, che se fossero disponibili farebbero del 6502 una vera "bomba".

Ogni volta che ci servono, siamo costretti a scrivere per intero la sequenza di istruzioni elementari che li descrivono. Più interessante sarebbe definire una volta per tutte queste sequenze standard e fare un semplice riferimento ad esse tutte le volte che sia necessario, eventualmente specificando i parametri su cui operare. In altre parole definirsi la macro istruzione che descrive l'istruzione assente a livello hardware.

Una macro altro non è che una piccola porzione di programma con in testa un nome e una lista di parametri formali. Ad esempio, la macro che descrive l'operazione di azzeramento di un determinato byte è:

```
MACRO :CLR  M
       LDA  #0
       STA  M
```

Come nel caso del Basic, la M presente nella dichiarazione è assolutamente formale: l'assemblatore, dopo questa dichiarazione, è informato dell'esistenza di questa nuova istruzione che si chiama CLR e che opera su un parametro.

Ogni volta che nel processo di assemblaggio viene incontrato un CLR di qualche byte di memoria, viene automaticamente sostituito con la sequenza di due istruzioni LDA #0 e STA [byte specificato]. Senza alcuna restrizione per il modo di indirizzamento. In questo caso, sono ammessi tutti i modi di indirizzamento concessi dall'istruzione STA (che nella dichiarazione usa il parametro M). Potremmo quindi usare "CLR $1000", "CLR (44),Y", "CLR (12,X)" etc.

Facciamo un discorso un po' più operativo. Supponiamo di aver già aggiunto al programma 2 del numero scorso le linee Basic presentate in quest'articolo. Caricato e fatto eseguire il programma DATA, diamo il RUN al secondo programma.

Con SHIFT e "1", si va in fase di Input dopo aver ripulito l'area di lavoro. Per far capire all'assemblatore che si sta definendo una Macro è obbligatorio scrivere "MACRO" nel campo Label della prima linea.

Si procede indicando, sempre nella prima linea, nel campo OPR il nome della Macro e nel campo Address la lista dei parametri, ognuno separato dal "Punto e virgola".

Fa seguito la porzioncina di programma che descrive la Macro da noi definita. Al termine, dopo essere tornati al MENU #1, bisogna assemblare la Macroistruzione in modo da poterla usare a nostro piacimento. Al termine di questa operazione, l'assemblatore, col solito trucchetto del [RETURN] forzato nel buffer di tastiera, inserisce fra le REM di testa la definizione cifrata della Macroistruzione.

Nel numero scorso, per non confondervi troppo le idee, oltre al "giallo della passata

Il minifloppy con il programma EXMA per VIC 20 pubblicato in questo numero e nel precedente (codice DVC/01) può essere acquistato presso la redazione al prezzo di lire 15.000 (compresa IVA e spedizione).
Per l'ordinazione inviare l'importo (a mezzo assegno, c/c o vaglia postale) alla Technimedia srl, Via Valsolda 135, 00141 Roma.
N.B.: EXMA non è disponibile su cassetta.

zero" (vedi riquadro per la soluzione), vi sono state nascoste altre due possibilità dell'EXMA. Una è il salto relativo, e si usa con i Branch condizionali. Si specifica nel campo Label il numero di istruzioni da saltare, in avanti col simbolo ">" o indietro col simbolo "<". L'istruzione:

```
BPL <$03
```

salta indietro di tre linee se il BPL ha dato esito vero. Sempre ad esempio:

```
BNE >$07
```

salta in avanti di 7 linee se è vero il BNE.

L'altra possibilità è la direttiva vuota. Si indica con ".CO" (dal fortran-iano CONTINUE) e quando l'assemblatore l'incontra, l'ignora del tutto e assembla la linea successiva. Sembra l'arte dei pazzi, ma non lo è. Specialmente l'ultima, è utile nelle definizioni macro, quando vi è un'uscita brutale dal corpo della definizione.

Facciamo un esempio: definiamo una macro che pone nell'accumulatore il massimo tra due oggetti. La definizione è:

```
MACRO  :MAX   ALFA;BETA
        LDA   ALFA
        CMP   BETA
        BPL   FINE
        LDA   BETA
FINE   :.CO
```

La direttiva ".CO" è stata necessaria dato che la label FINE (così come qualsiasi altra label) non può esistere se il campo

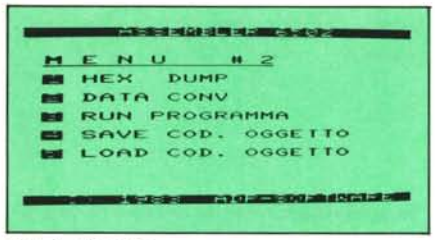

Foto 1 - Menu #2.

Foto 2 - Definizione della Macro SWP.

OPR non è occupato da qualcosa. La possibilità di definire i Branch relativi è sfruttata dall'assemblatore stesso nella fase di Macro Expansion. È questa fase che precede l'assemblaggio: vengono sostituite a tutte le Macro usate in un programma, le relative sequenze di istruzioni elementari (foto 2, 3 e 4).

A titolo di esempio, vediamo ora qualche Macro di uso più o meno comune:

```
MACRO  :SWP   I;J
        LDA   I
        PHA
        LDA   J
        STA   I
        PLA
        STA   J
```

Scambia (SWAP) il contenuto di due celle di memoria.

È importante notare che nella definizione di una Macro, possono starci sia istruzioni semplici, sia altre Macro purché già definite. Supponiamo di definire una istruzione che ordina in modo crescente due byte. Algoritmicamente ciò significa che se ALFA e BETA sono due byte e ALFA >= BETA non bisogna far nulla; se ALFA < BETA, bisogna scambiare i contenuti di ALFA e di BETA. In termini di Macro definizione:

```
MACRO  :ORD   ALFA;BETA
        LDA   ALFA
        CMP   BETA
        BPL   EXIT
        SWP   ALFA;BETA
EXIT   :.CO
```

Facciamo ora un esempio di Macro a tre parametri. Questa istruzione pone in un determinato byte (RE) il resto della divisione tra un byte dividendo (DD) e un byte divisore (DR):

```
MACRO  :RES   DD;DR;RE
        LDA   DD
LOOP   :STA   RE
        SEC
        SBC   DR
        BPL   LOOP
```

Notare che tanto DD quanto DR possono essere celle di memoria o numeri. RE deve essere necessariamente una cella di memoria.

Potremmo ad esempio usare:

```
RES $2234; #33,3
```

che pone nella cella 3 il resto della divisione tra il contenuto di $2234 e il numero 33; così come:

```
RES #50; $5; $200
```

pone nel byte $200 il resto tra #50 e la cella di memoria 5.

Nulla ci vieta di essere ancora più contorti:

```
RES ($41), Y;(54,X); 482,Y
```

pone in "482,Y" il resto della divisione tra "($41), Y" e "(54,X)".

In casi come questo, sorge però un piccolo problema: a causa del limitato numero di colonne del VIC, può capitare che una determinata chiamata di Macro non entri in una linea di schermo per i troppi (o troppo contorti) parametri passati.

Niente paura: si può usare il campo Label della linea seguente mettendo 3 puntini sospensivi nel campo OPR. La chiamata di Macro sopra descritta, di fatto, va inserita in memoria sotto forma di due linee; precisamente:

```
RES ($41), Y; (54,
...X); 482,Y cioè ...; (54,
... X); 482, Y
```

```
1285 IFA$(0,0)="MACRO"THENFF=1:N$=A$(0,1):AR$=A$(0,2)
1725 IFFF=1THEN3000
3000 H=PEEK(44)*256+PEEK(43)
3010 LH=PEEK(H+3)*256+PEEK(H+2)+1:H=PEEK(H+1)*256+PEEK(H):IFPEEK(H+4)=143THEN301
0
3020 A$=MID$(STR$(LH),2)+"REM*"+N$
3030 FORI=1TOT:X$=A$(I,1):IFX$=".CO"THEN3040
3035 X=VAL(X$):GOSUB1910:A$=A$+RIGHT$(X$,2):GOSUB3100
3040 AD$=A$(I,2):IFAD$=""THEN3090
3042 X=VAL(AD$):IFX=0ANDLEFT$(AD$,1)<>"0"THEN3050
3044 GOSUB1910:X$=RIGHT$(X$,2-2*(LEFT$(X$,2)<>"00")):Y$=RIGHT$(A$,1)
3045 IFI%(I)=5THENA$=A$+"#":GOTO3049
3046 IFLEN(X$)=2ANDY$<>")"ANDY$<>"<"THENA$=A$+"@"
3047 IFLEN(X$)=4THENA$=A$+"£"
3049 A$=A$+X$:GOTO3090
3050 J=1:K=0:LE=LEN(AD$)
3060 IFMID$(AR$,J,1)=";"THENK=K+1
3070 IFMID$(AR$,J,LE)<>AD$THENJ=J+1:GOTO3060
3080 A$=A$+CHR$(36+K)
3090 NEXT:PRINT"■■■■"+A$:PRINT"RUN■";:POKE198,2:POKE631,13:POKE632,13:END
3100 IF(VAL(A$(I,1))AND31)<>16THENRETURN
3110 X=VAL(A$(I,2)):II=0
3120 II=II+1:IFX<>A%(II)THEN3120
3130 IFII>ITHENA$=A$+")":A$(I,2)=STR$(II-I):RETURN
3140 A$=A$+"(":A$(I,2)=STR$(I-II):RETURN
3500 I=FF-1:DD=T
3502 I=I+1:IFI>DDTHEN3710
3505 HH=20486+I*24:FORH=1TO3:POKE831+H,PEEK(HH+H):NEXT:POKE835,1
3510 J=PEEK(832):IFJ=36ORJ=46THEN3502
3520 POKE0,180:POKE1,20:SYS19550:H=PEEK(836):IFH=3THEN3502
3530 H=PEEK(0)+PEEK(1)*256+3:IFA$(I+1,1)<>"..."THEN3535
3532 A$(I,2)=A$(I,2)+A$(I+1,2)
3533 L=20480+24*(I+1):POKE168,L/256:POKE167,(L/256-PEEK(168))*256:SYS19859
3535 AR$=A$(I,2)+";":LA$=A$(I,0):LL=I
3540 IFPEEK(H)=0THEN3650
3550 L=20480+24*I-1:POKE168,L/256:POKE167,(L/256-PEEK(168))*256:SYS19812:DD=DD+1
3560 HH=20486+24*I
3565 FORJ=1TO2:POKE166+J,PEEK(H+J-1):NEXT:SYS19894
3567 FORJ=0TO2:POKEHH+J+1,PEEK(253-J):NEXT:H=H+2:P=PEEK(H)
3568 IFP<>60ANDP<>62ANDP<>35ANDP<>92ANDP<>64THEN3575
3569 POKEHH+6,P:POKEHH+7,36:IFP=92ORP=64THENPOKEHH+6,32
3570 RORJ=1TO2-2*(P=92):POKEHH+7+J,PEEK(H+J):NEXT:I=I+1:H=H+3-2*(P=92):GOTO3540
3575 IFPEEK(H)>42ORPEEK(H)=0THENI=I+1:GOTO3540
3580 S=PEEK(H)-36:D=0:T=0
3590 T=T+1:IFMID$(AR$,T,1)<>";"THEN3590
3600 IFS>0THEND=T:S=S-1:GOTO3590
3610 B$=MID$(AR$,D+1,T-D-1):FORJ=1TOLEN(B$):POKEHH+5+J,ASC(MID$(B$,J,1)):NEXT
3620 H=H+1:I=I+1:GOTO3540
3650 L=20480+24*I:POKE168,L/256:POKE167,(L/256-PEEK(168))*256:SYS19859:FORI=1TOL
EN(LA$)
3660 POKE20479+LL*24+I,ASC(MID$(LA$,I,1)+" "):NEXT:SYS19992:T=PEEK(673)-1:GOTO35
00
3710 SYS19992:T=PEEK(673)-1:RETURN
```

Ancora:

```
MACRO :MCD  I;J
PIPPO :ORD  I;J
       RES  I;J;I
       LDA  I
       BNE  PIPPO
       LDA  J
```

pone nell'accumulatore il Massimo Comun Divisore tra due celle di memoria. Provare per credere! Dulcis in fundo:

```
MACRO :MUL  C;D
       LDA  #0
       STA  $0
       LDX  #8
LOOP   :LSR  C
       BCC  SKIP
       CLC
       ADC  D
SKIP   :ROR
       ROR  $0
       DEX
       BNE  LOOP
       LDY  $0
```

esegue la moltiplicazione fra due byte, ponendo il risultato a 16 bit nell'accumulatore (parte alta) e nel registro Y (parte bassa). In questo caso, essendo presente nella dichiarazione un LSR C, si impone che il primo parametro nella chiamata di questa Macro sia una cella di memoria e non un numero puro.

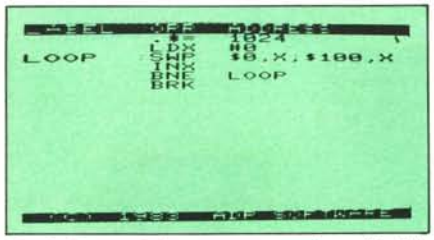

Foto 3 - Questo miniprogramma esegue lo scambio della pagina zero con la pagina 1, sfruttando la Macro SWP.

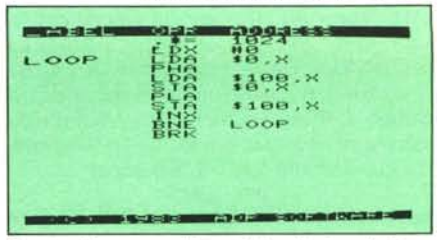

Foto 4 - Programma di foto 3 dopo la fase di Macro Expansion.

Avvisi e consigli

Ricordarsi che ogni definizione Macro è stivata in memoria sotto forma di linea Basic, grazie a un [RETURN] forzato nel buffer di tastiera. Ciò implica due considerazioni: primo, se si definisce una nuova Macro, bisogna salvare nuovamente il programma su nastro o su disco. Con l'inserimento di una nuova Macro, conterrà una linea in più.

Secondo, non è illimitata la lunghezza di una definizione. Oltre 16 o 17 linee, potrebbe non entrare, in forma cifrata, in una linea Basic (lunghezza 88 chr).

I parametri di una Macro devono essere sempre separati da "punto e virgola". Nei programmi, usare le Macro sempre con lo stesso numero di parametri usato nella definizione.

È inutile dire che non è lecito un:

```
LDX 1428, X
```

o peggio, un gustosissimo

```
STA $48
```

In altre parole, controllare che in una chiamata di Macro i parametri usati non combinino guai come sopra.

Notare che nelle Macro presentate come esempio in quest'articolo, accumulatore e registri indice vengono "sporcati". Chi non desidera ciò, salvi nello Stack detti registri prima di usare una determinata Macro.

Speriamo di aver detto tutto! Arrivederci. Se avete problemi vi preghiamo di non telefonare, ma scrivere! MC

Bibliografia

A.S. Tanembaum: Computer Structure Organization.
Digital Equipment Corporation: PDP 11/40 Processor Hand-book.
R. Zaks: Programmazione del 6502.

Il Miniminiquiz (del numero scorso): la soluzione

Cosa avviene prima dell'assemblaggio? Dando un'occhiata alla linea 1280, troviamo un DIM A$ (170,2), una SYS 19992 e subito dopo il programma di assemblaggio vero e proprio che continuamente fa riferimento al contenuto di A$(I,J), apparentemente non inizializzato (nessuna istruzione di assegnamento è presente in queste linee).

In una primitiva versione dell'EXMA, prima di iniziare la fase di assemblaggio, una routine Basic trasferiva il contenuto dell'area di lavoro nell'array A$ (I,J) grazie ad un semplice FOR e a delle istruzioni di PEEK. Lo svantaggio, tanto per cambiare, era appunto l'esasperante lentezza: per assemblare un qualsiasi programma, il 60% del tempo totale era perso per inizializzare A$ (I,J).

La routine in linguaggio macchina posta all'indirizzo 19992, pone rimedio all'inconveniente, risolvendo il tutto in pochi decimi di secondo.

Per capire meglio il funzionamento, vediamo come il VIC organizza all'interno della sua memoria, la gestione di un array di tipo stringa. L'Array Header è il "descrittore" dell'array, ed è creato in memoria all'atto del dimensionamento. Nel caso di matrici a due indici (la nostra è 171 × 3), è composto da 9 byte ed è immediatamente seguito da tanti Array Elements quanti sono gli elementi dell'array (fig. 1). Ognuno di questi elementi è composto a sua volta da tre byte, dei quali il primo indica la lunghezza e gli altri due l'indirizzo dove la stringa è "stivata". Modificando opportunamente ogni Array Element (all'atto del DIM tutti a zero), si ha l'effetto, ritornando al Basic, che ogni stringa non è vuota ma contiene il corrispondente "atomo" di programma da assemblare.

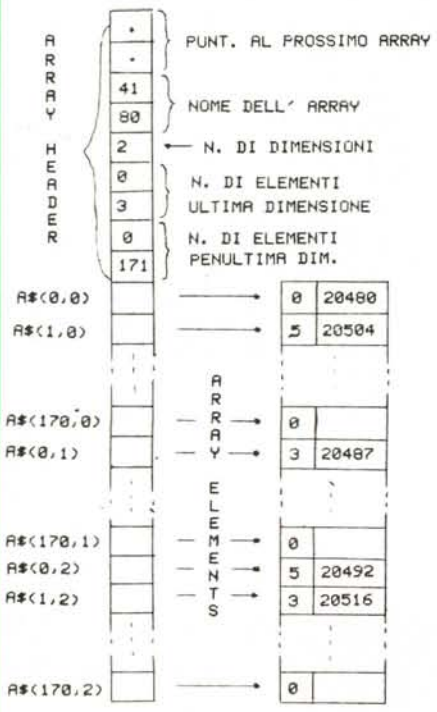

Figura 1 - Organizzazione di un Array bidimensionale di stringhe nella memoria del VIC.

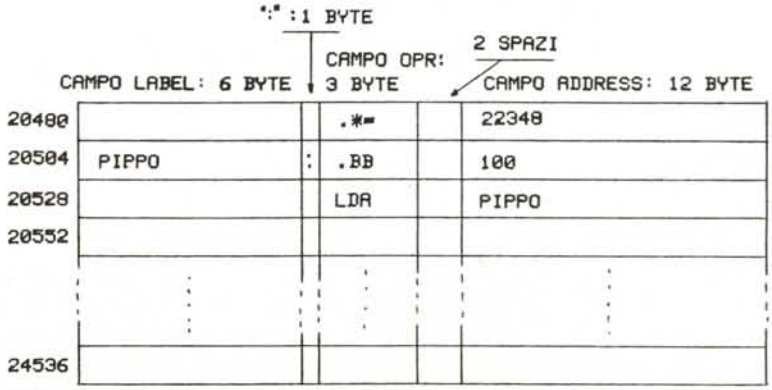

Figura 2 - Area di lavoro dell'assemblatore.