

INTRODUZIONE ALL'ASSEMBLER

COD. 6028

PER IL
COMMODORE 64

 **commodore**
COMPUTER

EDIZIONE RIVEDUTA E CORRETTA
Ottobre 2024, mssmsoft/Ready64
=====

Il presente documento e' la versione restaurata delle scansioni pubblicate da Ready64.org del libro:

"Introduzione all' Assembler"
(C)1984 Commodore Italiana SpA

Si e' ritenuto di realizzare la presente SECONDA EDIZIONE (non ufficiale) a distanza di circa 40 anni dalla prima pubblicazione per correggere i numerosi errori presenti nel testo originale, preservando la forma, i contenuti, l'aspetto e l'impaginazione dell'opera.

La descrizione del processo di restauro e un elenco dettagliato delle correzioni si trova alla fine del volume.

Curiosita':

da un listino ufficiale Commodore del 1984, il prezzo del volume originale, corredato da disco o nastro, era di 75.000 lire + IVA.



INTRODUZIONE ALL'ASSEMBLER

COD. 6028

**PER IL
COMMODORE 64**

Commodore Italiana SpA

Via F.lli Gracchi, 48 - 20092 Cinisello Balsamo (Milano)
Tel. 02/618321

© 1984 Commodore Italiana SpA

Tutti i diritti riservati. Nessuna parte del manuale e dei programmi può essere duplicata, copiata, trasmessa o riprodotta in qualsiasi forma o con qualsiasi mezzo senza il preventivo consenso scritto della Commodore Italiana

Commodore Italiana SpA

Via F.lli Gracchi, 48 - 20092 Cinisello Balsamo
Tel. 02/618321

INTRODUZIONE

Questo manuale si propone di essere il primo di una serie di pubblicazioni autosufficienti per insegnare la programmazione in Assembler.

Per questo motivo e' corredato da una cassetta o da un disco contenente un programma ASSEMBLER insieme ad altre Utility e che presuppone l'uso di un computer che nel nostro caso e' il CBM 64.

Nessun manuale infatti, a nostro parere, puo' consentire di programmare fino a quando non si eseguono delle applicazioni pratiche ed e' per questo che e' stato ritenuto necessario dotare il volume del supporto costituito dal linguaggio.

Come ripeteremo anche oltre, la lettura e le possibilita' applicative di questo manuale non possono prescindere da una conoscenza anche se non necessariamente approfondita del linguaggio Basic implementato sul computer.

Indispensabile invece appare la conoscenza del Sistema Operativo del computer sul quale si opera anche se cio' avverra' come necessario complemento a questo corso.

La soluzione degli esercizi proposti costituirà un valido complemento allo studio in particolare se da parte dell'utente si cercherà di risolverli con altra metodologia.

Infatti un ALGORITMO informatico ammette di solito piu' di una soluzione e comunque sempre piu' di una metodologia risolutiva. Da qui il

nostro consiglio per cercare di risolvere i problemi posti con gli esercizi e possibilmente anche gli esempi in altro modo.

NOTA

In alcuni punti di questo manuale il simbolo # e' stato rappresentato con £.

CAPITOLO PRIMO

Nell' aver operato su di un computer vi sara' senza dubbio capitato di sentire le parole CODICE MACCHINA ed il linguaggio chiamato ASSEMBLER.

Daremo successivamente definizioni piu' approfondite di questi termini, tuttavia in modo semplice il CODICE MACCHINA e' il linguaggio, anzi il solo linguaggio che il microprocessore comprende.

Facciamo un semplice esempio di somma, aggiungendo il numero 1 al numero 2.

IN LINGUAGGIO CORRENTE DIREMO :

Aggiungo 1 a 2, quale e' il risultato?

IN BASIC SI POTREBBE IMPOSTARE:

```
10 A=1
20 B=2
30 C= A+B
40 PRINT C
```

IN CODICE MACCHINA (per il 6510):

A2 01
8E 84 03
A9 02
6D 84 03
8D 85 03
60

Cio' che non risulta molto intelleggibile.
Diamo di seguito lo stesso problema in ASSEMBLER
con un breve commento per ogni linea.

LDX £1 Carica il registro X con '1'
STX 900 Immagazzina il contenuto di X in 900
LDA £2 Carica '2' nell' Accumulatore
ADC 900 Aggiungi all' Accumulatore il contenuto
dell'indirizzo di memoria 900

STA 901 Metti il contenuto dell' Accumulatore
nella locazione di memoria 901
RTS Ritorno da Subroutine in L.M.

Ecco uno dei motivi per il quale si utilizza l'
ASSEMBLER che come possiamo vedere e' molto piu'
facile da leggere e da ricordare che non il
Linguaggio Macchina visto prima.

La funzione di tutti gli Assembler e' quindi
quella di tradurre un programma scritto in codici
mnemonici, e quindi semplici da ricordare, in
Codice Macchina.

E' possibile anche inserire codici macchina
direttamente in memoria e questo sara' visto piu'
avanti in altri capitoli di questo libro.

ACCUMULATORE

Il cuore del microprocessore 6510 e' un registro chiamato ACCUMULATORE ed abbreviato con la lettera A.

Si tratta di un registro ad 8 bit attraverso il quale passeremo quasi sempre e che quindi puo' immagazzinare numeri fino ad un massimo di 255.

Le istruzioni del 6510 consentono di scrivere direttamente entro questo registro.

Una delle istruzioni appena viste e':

LDA& Load Accumulator using Immediate Mode

cioe' carica l'Accumulatore usando il modo immediato.

NOTA

L' utilizzo del MODO (in questo caso IMMEDIATO) per questa istruzione e' dato dal segno &.

Un' altra istruzione vista consente di trasferire un numero immagazzinato nell' Accumulatore in una specificata locazione di memoria. Questa istruzione e':

STA Store contents of Accumulator in the address specified

cioe' immagazzina il contenuto dell' Accumulatore

in un dato punto della memoria.

NOTA

Nel caso del CBM64 se la locazione di memoria nello quale si copia il contenuto dell' Accumulatore e' compresa fra i valori decimali 1024 e 2023, allora il valore corrispondente di quel numero sara' visualizzato sullo schermo appunto perche' questa e' l' area riservata alla memoria di schermo.

NOTA II

Il comando STA esegue una vera e propria COPIA dell' Accumulatore, lasciando il contenuto dell' accumulatore stesso inalterato.

Creeremo ora un programma che immettera' un numero nell' Accumulatore e dopo lo visualizzera' nel punto piu' in alto sinistra dello schermo.

Prima pero' e' bene puntualizzare qualcosa circa l' Assembler che useremo.

Diamo quindi una prima veloce spiegazione di utilizzo del programma ASSEMBLER presente sul lato A della Cassetta.

Naturalmente questo discorso non e' valido per chi adoperasse la versione disco.

- 1) Controllare che la cassetta sia all' inizio cioe' che sia stata riavvolta completamente.
- 2) Eseguire il caricamento con LOAD.
- 3) Eseguire il comando RUN. Sul video appare il seguente Menu':

```
..... IMMISSIONE
..... CANC. LINEE
..... INSER. LINEE
..... LIST PROGRAMMA
..... MEMORIZZAZIONE
..... CARICAMENTO
..... COMPILAZIONE
..... RUN
..... NEW
..... MONITOR
```

In questo Menu' potete selezionare qualsiasi scelta per mezzo del cursore UP/DOWN.

Una volta scelta la funzione da eseguire premere il tasto Return.

Naturalmente la prima funzione da scegliere sara' IMMISSIONE.

- 4) Di conseguenza alla scelta di IMMISSIONE verra' chiesto il numero di linea da cui partire. E' utile notare che il numero di linea e' simile alla numerazione delle linee di un programma BASIC, ma questo concetto permette di ACCODARE semplicemente piu' programmi. E' inoltre in funzione la numerazione automatica con incremento

di 1 delle linee di programma.

5) di conseguenza compaiono 4 campi:

N. LINEA	LABEL	OPCODE	VALORE
----------	-------	--------	--------

NOTA

Quando si inizia a scrivere un programma in Assembler e' necessario conoscere in quale punto della memoria si desidera poi immagazzinare il programma stesso.

Nel CBM 64 ci sono numerosi punti della memoria disponibili per questa funzione di immagazzinamento, tuttavia per il momento e per brevi programmi, possiamo utilizzare il Buffer di cassetta grande 192 Bytes e che va dalla locazione decimale 828 a 1019.

La seconda annotazione sull' Assembler e' che per il momento utilizzeremo solo il formato decimale per i nostri numeri, mentre piu' in avanti vedremo come usare le altre notazioni numeriche disponibili.

6) E' necessario ora comunicare la locazione di inizio del nostro programma con:

1 * = 828

scriveremo quindi il nostro programma ed al termine, nella colonna delle LABEL, l' istruzione EXIT oppure premere F7.

La forma base di scrittura sara' quindi:

```
N. LINEA LABEL OPCODE VALORE
.....
.....
..... * = ..... 828
.....
.....
N. LINEA EXIT
```

NOTA

Prima di passare a scrivere il nostro primo programma e' opportuno notare che si possono eseguire delle correzioni anche semplicemente riposizionandosi con il cursore.

La prima e l' ultima linea mostrate in precedenza non hanno niente a che fare con il Programma in Codice Macchina. Danno semplicemente delle informazioni al programma traduttore.

Dopo aver inserito il programma questi deve essere ASSEMBLATO.

Si deve cioe' far eseguire l' opzione COMPILA scelta dal MENU' perche' cio' che abbiamo scritto in codice mnemonico (LDA, STA, ecc) possa essere convertito, cioe' tradotto in Codice Macchina che, come abbiamo detto in precedenza, puo' essere direttamente eseguito o processato dal computer.

Successivamente useremo l'opzione RUN del MENU' che ricordiamo e' un comando del Sistema operativo del linguaggio Basic che ordina al computer di iniziare l'esecuzione di un programma a partire da un dato indirizzo. Oppure useremo una SYS di chiamata all'indirizzo di partenza. Nel nostro caso quindi SYS 828.

NOTA

Ricordiamo che tutte e due i metodi devono comunicare al computer di ritornare in ambito Basic (almeno per il momento) perche' in caso contrario si entrerebbe in un LOOP o ciclo infinito per uscire dal quale occorrerebbe resettare il sistema intero.

Il comando che esegue questa funzione di ritorno e che quindi deve essere messo al termine di ogni programma Assembler e':

RTS ReTurn from Subroutine

cioe' ritorno da Subroutine.

Riassumendo, per inserire i dati e far funzionare il nostro programma dovremo eseguire i seguenti passi:

1- Comunicare all'Assembler che l'indirizzo di partenza che nel nostro caso abbiamo scelto nella locazione decimale 828

2- Caricare (LoaD) il numero '0' entro l' accumulatore A usando il Modo Immediato. Il codice mnemonico per fare questo e' LDA seguito dal numero che deve essere caricato, es. LDA& 0.

3- Immagazzinare (STore) in un dato indirizzo il contenuto dell' Accumulatore. Il codice Mnemonico e' STA, es. STA 1024.

4- Immagazzinare (STore) in un altro indirizzo il contenuto dell' accumulatore, es. STA 55296. Spiegheremo dopo il perche' di questa operazione.

5- Comunicare all' Assembler (non al 6510) la fine.

Programma 1.1

1				*	=	828
2	033C	A9	00		LDA	#0
3	033E	8D	00	04	STA	1024
4	0341	8D	00	D8	STA	55296
5	0344	60			RTS	

Per inserire questo programma vediamo con dettaglio i passi da fare naturalmente trovandosi davanti al computer:

a) Caricare l' Assembler.

b) Digitare RUN e Return

c) Lo schermo mostra il MENU

d) Selezionare l' opzione IMMISSIONE, cioe' per entrare nel modo programma. Sara' ora necessario comunicare lo:

INDIRIZZO DI PARTENZA

e) Apparira':

N. LINEA	LABEL	OPCODE	VALORE
----------	-------	--------	--------

daremo i sottoindicati valori

1	*		828
---	---	--	-----

avranno cioe' dato l' indirizzo di partenza.

f) Scrivere LDA Return 0 premere Return

g) Scrivere STA Return 1024 e Return

h) Scrivere STA Return 55296 e Return

i) Scrivere RTS e Return

ATTENZIONE!!!

Fare particolare attenzione all' incolonnamento e ricordarsi in particolare che per saltare da una colonna all' altra e' necessario il RETURN.

Ricordare che per il momento la colonna delle

Label non viene usata.

NOTA

E' probabile che per ragioni tipografiche le regole sulle spaziature appena riportate non siano sempre rispettate con precisione in tutti i programmi. Tuttavia Vi invitiamo ad attenervi strettamente ad esse.

A questo punto il programma su schermo dovrebbe essere il seguente:

```
1          *          =      828
2  033C A9 00          LDA  #0
3  033E 8D 00 04      STA  1024
4  0341 8D 00 D8      STA  55296
5  0344 60           RTS
```

Se tutto e' OK premere Return ed il programma ritornera' al MENU di scelta. In caso di errore ripetere la procedura dal punto d).

Dopo che siamo ritornati al MENU eseguire i passi descritti:

1) Selezionare l' opzione COMPILAZIONE per far eseguire la traduzione. In questa occasione viene

richiesto da video o stampa.

m) Selezionare l' opzione RUN, digitare 828 e Return. Dopo di cio' il programma per prima cosa pulira' lo schermo e fara' girare il programma stampando una a commerciale in nero sulla parte sinistra in alto dello schermo stesso.

n) Premere un tasto e si tornera' al MENU.

Dal MENU selezionare l'opzione LIST per listare il programma.

Verra' chiesto l' indirizzo di partenza e le linee da listare.

I REGISTRI INDICE

Oltre all' Accumulatore, il 6510 ha due registri detti REGISTRI INDICE:

REGISTRO INDICE X

REGISTRO INDICE Y

Ognuno di questi che d' ora in poi chiameremo semplicemente registro X o Y ha, come l' Accumulatore, la possibilita' di immagazzinare valori in un intervallo da 0 a 255 essendo registri da 8 bits.

NOTA

Ricordiamo ancora una volta che un registro e' una locazione di memoria nella quale puo' essere caricato un valore.

Questo valore di norma e' compreso, come abbiamo detto in un intervallo fra 0 e 255 per i registri da 8 bits e fra 0 e 65535 per i registri da 16 bits.

Per il momento inoltre i registri X e Y sono mostrati con funzionamento simile fra loro sebbene in effetti differiscano come comportamento come vedremo nel corso del volume.

Il grande vantaggio di questi registri indice e'

che il valore in essi contenuto puo' essere incrementato o decrementato (di 1 per volta). Naturalmente non sono solo questi i vantaggi e le possibilita' che vedremo oltre.

Altro punto da prendere in considerazione e' l' ALU o ARITHMETIC AND LOGIC UNIT cioe' unita' aritmetico-logica che si trova all'interno del microprocessore stesso ed e' da questi usata appunto per tutte le operazioni aritmetico logiche.

La ALU ha due ingressi per i dati sui quali esegue le operazioni ed un' uscita tramite la quale i risultati delle operazioni stesse vengono inviati all' Accumulatore.

Quindi tutti i dati sui quali opera il 6510 passano almeno una volta attraverso l' Accumulatore e da questo si intuisce l' importanza di questo Registro.

La strada sulla quale i dati passano si chiama:

DATA BUS

Mentre nel seguito di questo capitolo e dei prossimi vedremo come i dati passino da un registro all' altro e come si possa aver accesso alla memoria, esaminiamo ora i comandi per mettere in funzione questi due registri.

LDX Load index register X

Carica il registro indice X con i dati di una locazione di memoria. Per esempio:

```
LDX 900
```

Consente di immettere nel registro X il dato presente nella locazione di indirizzo decimale 900.

LDX differisce da LDA& perche', a parte il fatto che un' istruzione carica il registro X e l'altra l' Accumulatore, LDA& e' un comando IMMEDIATO.

In altre parole quando il 6510 trova l'equivalente in codice assemblato dell' istruzione LDA&, leggerà il valore immediatamente seguente il comando, (cioe' il suo parametro) e lo caricherà nell' Accumulatore.

Con il comando LDX invece il microprocessore prenderà il dato scritto immediatamente dopo l' istruzione e lo considererà come INDIRIZZO a cui dirigersi per caricare un dato.

Con la seguente istruzione pertanto:

```
LDX 900
```

il 6510 eseguirà la lettura della locazione di memoria 900, preleverà il dato ivi presente e lo caricherà nel registro indice x.

NOTA

Anche nel caso di questa istruzione ci troviamo di fronte ad una COPIA di un contenuto di memoria perche' cio' che era nella locazione di memoria 900 viene copiato nel registro X, ma lo stesso valore resta sempre anche all' indirizzo 900.

Poiche' e' necessario disporre spesso di questi registri liberi ecco un' istruzione che consente di immagazzinare invece il contenuto di uno di questi registri in una data locazione di memoria. L' istruzione e':

STX STore X in a address

Cioe' immagazzina il contenuto del registro X in una data zona di memoria. Per esempio:

STX 1024

Comunica al computer di immettere il contenuto del registro indice X nella locazione di memoria 1024.

Programma 1.2

1					*	=	828
2	033C	A9	01			LDA	#1
3	033E	8D	00	04		STA	1024
4	0341	8D	00	D8		STA	55296
5	0344	AE	00	04		LDX	1024
6	0347	8E	02	04		STX	1026
7	034A	8E	02	D8		STX	55298
8	034D	60				RTS	

- 1 Indirizzo di partenza
- 2 Carica 1 nell' Accumulatore
- 3 Carica il contenuto dell' Accumulatore in 1024.
- 4 Immagazzina il contenuto dell' Accumulatore nella locazione 55296 per visualizzare la rappresentazione dello schermo in bianco.
- 5 Carica nel registro X il contenuto di 1024 (1).
- 6 Immetti il contenuto di X in 1026
- 7 Immetti il contenuto di X in 55298 per visualizzare in BIANCO anche questo carattere.
- 8 Ritorno da subroutine.

Non appena ritornati al MENU selezionare l' opzione COMPILA e poi il RUN per far girare il programma.

Se tutto e' stato eseguito correttamente dovrebbero essere visualizzate:

A spazio A

entrambe le lettere in alto a sinistra in bianco.

NOTA

Le possibili soluzioni dei seguenti esercizi sono date al termine del manuale.

Abbiamo parlato di possibili soluzioni in quanto come abbiamo già detto, per risolvere un dato algoritmo sono possibili più soluzioni.

Esercizio 1.1

Caricare l' Accumulatore con 1 visualizzando questo nella locazione 1024 in verde.

Esercizio 1.2

Scrivere il vostro nome in alto a sinistra dello schermo.

Esercizio 1.3

Scrivere la lettera X in ognuno dei quattro angoli dello schermo.

IL REGISTRO Y

Dopo aver visto le istruzioni relative al registro X vediamo ora quelle del registro Y in molti casi del tutto eguali.

LDY Load register Y

cioe' carica il registro Y con il contenuto di un dato registro di memoria.

STY Store in Y

cioe' immagazzina i dati contenuti nel registro Y in un determinato indirizzo di memoria.

Per molte operazioni, MA NON PER TUTTE, i registri Y e X possono essere intercambiabili.

Per questo il programma 1.2 visto in precedenza che riportiamo qui:


```
*      =      828
LDA&1
STA 1024
STA 55296
LDX 1024
STX 1026
STX 55298
RTS
```

puo' essere scritto:

Programma 1.3

```
*      =      828
LDA&1
STA 1024
STA 55296
LDY 1024
STY 1026
STY 55298
RTS
```

La necessita' di girare o passare rapidamente i dati da un registro all' altro anche durante l' esecuzione di un programma evidenzia l' utilita' delle seguenti istruzioni:

TAX Transfer Accumulator in X.

Cioe' trasferisci il contenuto dell' Accumulatore

nel registro X.

Usando ad esempio questo comando si puo' riscrivere il programma 1.3 in questo modo:

```
*      =      828
LDA&1
STA 1024
STA 55296
TAX
STX 1026
STX 55298
RTS
```

Anche in questo caso avremo il risultato dell' esercizio precedente, cioe' la stampa di due lettere A separate da uno spazio bianco.

NOTA

Fino a questo momento sono state date le descrizioni complete dei registri, ma d' ora in poi, per brevitaa', li indicheremo con la sola lettera maiuscola e non piu' come REGISTRO INDICE X o Y, ma quindi come X o Y.

Per esempio l' ultima istruzione vista TAX sara' descritta come:

```
TAX    Trasferisci A in X.
```

Vediamo ora le altre istruzioni di trasferimento:

TAY Trasferisci A in Y

TXA Trasferisci X in A

TYA Trasferisci Y in A

ESERCIZI

Esercizio 1.4

Scrivere un programma che carichi una Z entro l' Accumulatore ed una A entro il registro X. Poi, senza utilizzare comandi in modo immediato, visualizzare la Z nella prima locazione di memoria e la A nell' ultima.

Esercizio 1.5

Scrivere un programma che carichi il simbolo dei quadri nell' Accumulatore, un asterisco in X ed una E in Y. Senza usare i comandi in modo immediato, portare la E nell' Accumulatore, il simbolo dei quadri in X e l' asterisco in Y. Visualizzare in alto a sinistra dello schermo il simbolo dei quadri, in basso a destra l' asterisco e nei rimanenti angoli liberi due E.

CAPITOLO SECONDO

I salti ed il Program Counter

In realta' quasi nessun programma procede attraverso una serie di passi ininterrotti, senza salti a subroutines, a Kernal routines o altro. Questo capitolo esamina questi comandi e il loro uso.

Vedremo poi i flags che consentono di controllare i salti e le diramazioni.

SALTI INCONDIZIONATI

Sono comandi che dicono al programma di saltare ad un certo indirizzo, semplicemente, cioe' senza condizioni.

Sul 6510 esistono solo due istruzioni di questo tipo. La prima che vediamo e':

`JMP JuMP to the specified address`

Cioe' salta ad un dato indirizzo

Per esempio, `JMP 834` ordina di saltare alla locazione di memoria 834.

Questo potrebbe, e spesso e' cosi', essere un modo di inserire pezzi di programmi dimenticati o parti di programma in aggiunta.

Vediamo ora il comportamento in un programma in cui sia stata effettivamente inserita l'istruzione detta.

Ora questo puo' essere scritto cosi':

Programma 2.1

1				*	=	828
2	033C	A9	01		LDA	#1
3	033E	4C	42	03	JMP	834
4	0341	60			RTS	
5	0342	8D	00	04	STA	1024
6	0345	8D	00	D8	STA	55296
7	0348	4C	41	03	JMP	833

Quando i salti sono usati in questo modo e' necessario dire al programma dove esattamente deve saltare, dando per questo un indirizzo come JMP 834.

Nel calcolo degli indirizzi di salto e' necessario tenere conto dell'occupazione di memoria derivata dall'uso dell'istruzione stessa ed eventualmente di quella degli operandi. Vediamo, guardando per questo al programma precedente, la successione di memoria occupata dalle istruzioni in modo da comprendere bene perche' per superare il RTS e' necessario saltare

a 834:

```
828 LDA&
829 1
830 JMP
831 - 832 (indirizzo 834)
833 RTS
834 STA
835 - 836 (indirizzo 1024)
837 STA
838 - 839 (indirizzo 55296)
840 JMP
841 - 842 (indirizzo 833)
```

Le parti seguenti i comandi, e che sono conosciute come operandi, determinano una qualche complicazione nel calcolo degli indirizzi.

Esiste una strada semplice o almeno relativamente semplice, per il calcolo degli indirizzi ed e' quella di adoperare le tavole in appendice dove sono riportati i codici delle istruzioni ed i bytes necessari.

JSR Jump to SubRoutine

Questo e' un altro comando che consente un salto ad una Subroutine.

Utilizzandolo insieme a RTS consente una funzione simile a GOSUB.... RETURN del Basic.

Vediamo un esempio comparativo:

BASIC	ASSEMBLER
-----	-----
10 GOSUB 200	830 JSR 834
...	...
...	...
...	...
...	...
200 REM SUB ROUTINE	834 STA 1024
...	...
...	...
...	...
300 RETURN	840 RTS

Proviamo a modificare il programma 2.1 usando l'istruzione appena vista in luogo di JMP.

Programma 2.2

1		*	=	828
2	033C	A9 01	LDA	#1
3	033E	20 42 03	JSR	834
4	0341	60	RTS	
5	0342	8D 00 04	STA	1024
6	0345	8D 00 D8	STA	55296
7	0348	60	RTS	

Il vantaggio di RTS su JMP e' dimostrato da questo programma dove con RTS non e' necessario calcolare l' indirizzo di salto per il ritorno al programma principale.

PROGRAM COUNTER (PC)

Questo e' un registro di 16 bit che contiene gli indirizzi del prossimo comando da eseguire.

In realta' si tratta di due memorie, di 8 bit ciascuna, inserite entro il 6510.

Quando si seleziona l' opzione RUN del menu' e si da' come locazione di partenza l' indirizzo 828, questa azione genera un comando che fissa il PC a 828 ed inizia quindi l' esecuzione da qui.

Il Program Counter si incrementa in rapporto al tipo di istruzione data in modo tale da puntare alla locazione di memoria che conterra' quindi i dati richiesti, ma che non sara' necessariamente la successiva.

Prendiamo per esempio le prime tre linee del precedente programma:

```
*      =      828  
LDA# 1  
JSR 834
```


e vediamo un sommario dei contenuti del PC alla esecuzione delle varie istruzioni:

PROGRAMMA	CONTENUTO DEL PROGR.	COUNTER
	PRIMA	DOPO
INIZIO 828	?	828
LDA& 1	828	830
JSR 834	850	834

Ricordiamo che questa area di memoria e' grande solo 16 bit per cui se e' necessario immagazzinare piu' di un indirizzo dovremo far ricorso ad un' area esterna di memorizzazione, lo STACK, che vedremo in seguito.

Esercizio 2.1

Scrivere un programma che metta un 3 nell' accumulatore. Il programma deve incominciare ad 828, saltare quindi (cioe' eseguire un JUMP) alle routine di indirizzo 900 che aggiunga un 3 al 3 gia' presente in A.

Ritornare quindi alla routine originale e stampare, in alto a sinistra dello schermo, il contenuto dell' Accumulatore, cioe' il risultato dello somma.

SALTI CONDIZIONATI

Fino a questo momento abbiamo sempre visto dei salti incondizionati, ma un qualsiasi programma che necessiti di un minimo di controllo avra' la necessita' di : SALTI CONDIZIONATI.

Per fare un' analogia con il Basic possiamo prendere il comando di condizionamento IF....THEN:

```
10 IF X=Y THEN 500
```

In questa linea i valori X e Y, che sono stati immagazzinati in memoria, sono confrontati fra loro e se si verifica la condizione di eguaglianza, almeno in questo caso, si salta alla linea specificata dopo il THEN.

Il 6510 puo' eseguire questo tipo di operazione in una molteplicita' di modi.

Uno di questi e' attraverso l' uso di un particolare registro chiamato REGISTRO DI STATO o STATUS REGISTER (SR) o anche conosciuto come PROCESSOR STATUS WORD.

Lo STATUS REGISTER e' un registro di 8 bits come l' Accumulatore, i registri X e Y ma viene usato in maniera differente.

Mentre gli altri registri sono usati per immagazzinare e manipolare Bytes, questo registro divide e quindi considera separatamente i suoi singoli BITS come FLAGS o segnali.

Di norma il 6510 manipola uno solo di questi flags per volta, sia fissandone il suo valore a 0 o a 1 sia controllando se il valore e' a 0

oppure a 1.

In altre parole su questi flags (che sono poi i singoli bits del SR) si puo', di volta in volta scrivere o leggere il valore relativo.

Un esempio di uno di questi flags e' il flag Z o flag ZERO.

Quando viene eseguito un programma o una manipolazione di dati che produca un risultato di 0 in un determinato registro (A,X o Y) allora il flag Z viene messo a 1. Se invece il risultato e' diverso da 0 allora lo Z flag viene messo a 0.

Altre istruzioni possono settare questo flag, una di queste e'!

DEX DEcrement the contents of register X

Cioe' decrementa il contenuto del registro X.

Questo pezzo di programma ne dimostra l' uso.

Programma 2.3

(Parte)

```
1 * = 828
2 LDX #100
3 DEX
```

Viene cioe' caricato X con 100 e poi decrementato.

Quando il contenuto di X sia 0 allora il Flag ZERO viene messo a 1.

Se vogliamo usare questa capacita' del Flag per eseguire dei controlli sul programma, dovremo usare un' istruzione che controlli il contenuto del Flag stesso e che quindi consenta di effettuare salti o deviazioni in dipendenza del fatto che il Flag sia a 0 o a 1.

Vediamone un' istruzione:

BEQ Branch if result was EQual to zero

Cioe' esegui il salto se il valore, per esempio del flag Z, e' eguale a 0.

E' necessario fare un po' di attenzione a questo punto per non confondersi.

Rileggendo quanto abbiamo detto in precedenza infatti, se il risultato dell' operazione e' ZERO allora il flag Z viene messo a 1, per cui il BEQ, cioe' la sua condizione, si verifica quando il flag Z e' =1.

L' operando, con questa istruzione e' solo di 1 Byte, per cui si possono manipolare numeri da 0 a 255.

Introduciamo ora, sebbene in modo preliminare il concetto di LABEL.

Per LABEL si intende un particolare indirizzo di memoria al quale abbiamo assegnato un nome.

Con l' intruzione BEQ PIPPO sara' effettuato un salto alla locazione di riferimento e questo senza bisogno di contare il punto preciso a cui saltare.

Programma 2.3

```
ROUT = $033E
FINE = $0344
1      *      =      828
2  033C A2 64      LDX #100
3  033E CA      ROUT  DEX
4  033F F0 03      BEQ  FINE
5  0341 4C 3E 03      JMP  ROUT
6  0344 8E 00 04  FINE  STX  1024
7  0347 8E 00 D8      STX  55296
8  034A 60      RTS
```

Quando questo programma viene eseguito, verra' visualizzata una A commerciale nera in posizione 1024.

Come molte istruzioni relative al registro X anche DEX ha la sua istruzione corrispondente per il registro Y:

DEY DEcrement the contents of register Y.

Cioe' decrementa il contenuto del registro Y.

Esercizio 2.2

Scrivere un programma simile al precedente ma che utilizzi pero' il registro Y.

Un' altra istruzione che controlla lo stato del Flag Z e' la seguente:

BNE Branch if Not Equal

Salta se non eguale

Questa istruzione e' esattamente il rovescio della BEQ ed esegue il salto se il Flag Z e' a 0, cioe' non e' settato.

Il seguente programma e' una modifica del programma 2.3.

Notate come questo programma sia un po' piu' corto del precedente che faceva uso dell' istruzione BEQ.

Programma 2.4

```
ROUT = $033E
FINE = $0341
1          *          = 828
2  033C A2 64          LDX #100
3  033E CA          ROUT  DEX
4  033F D0 FD          BNE ROUT
5  0341 8E 00 04  FINE  STX 1024
6  0344 8E 00 D8          STX 55296
7  0347 60          RTS
```

Il risultato sara' identico a quello ottenuto in precedenza con il programma 2.3.

I registri indice X e Y sono stati decrementati o INDICIZZATI VERSO IL BASSO, cioe' verso un valore inferiore, con le istruzioni DEX e DEY e naturalmente e' possibile eseguire l'operazione opposta o indicizzarli verso l'alto, cioe' incrementarli, utilizzando le seguenti istruzioni:

INX INcrement the contents of X by 1

Cioe' incrementa il contenuto del registro X di 1

INY INcrement the contents of Y by 1

Cioe' incrementa il contenuto del registro Y di 1

ISTRUZIONI DI CONFRONTO

Utilizzando gli incrementi un vero controllo per il valore 0 non e' naturalmente possibile per cui i registri devono essere confrontati con un valore preventivamente immagazzinato da qualche parte.

Il 6510 ha 3 istruzioni per eseguire questo controllo.

CPX ComPare the contents of the specified memory address with the X register.

Cioe' confronta il contenuto di un dato indirizzo di memoria con il registro X.

Cio' viene fatto sottraendo il contenuto di memoria da X cosa che puo' dare un valore positivo, negativo o zero.

Percio' l'istruzione CPX 900 si comportera' nel modo seguente:

- 1) Legge il contenuto della locazione di memoria 900
- 2) Sottrae questo contenuto da quello del registro X
- 3) Mette a 1 (o setta) il flag Z se la risposta e' =0

NOTA

Ne il contenuto della locazione di memoria 900 ne il contenuto di X vengono pero' cambiati durante questa fase.

Per il momento noi siamo interessati alla condizione zero (infatti anche altri flags sono interessati da operazioni simili).

Per utilizzare questa istruzione possiamo mettere il registro X a zero ed immagazzinare un valore di confronto da qualche parte della memoria.

Vediamone un' applicazione in programma

commentata.

Programma 2.5

```
ROUT = $0343
FINE = $034C
1      *      = 828
2      033C A9 5A      LDA #90
3      033E 8D 7A 03   STA 890
4      0341 A2 00      LDX #0
5      0343 E8      ROUT INX
6      0344 EC 7A 03   CPX 890
7      0347 F0 03      BEQ FINE
8      0349 4C 43 03   JMP ROUT
9      034C 8E 00 04   FINE STX 1024
10     034F A9 01      LDA #1
11     0351 8D 00 D8   STA 55296
12     0354 60      RTS
```

Spiegazione del programma:

```
1      Indirizzo di partenza
2      Carica 90 in Accumulatore
3      Carica il contenuto di A a 890
4      Carica 0 nel registro X
5      Incrementa il registro X
6      Confronta il valore in X con quello in 890
7      Vai avanti di 3 Bytes se dal confronto
risulta CPX=0
8      Vai a ROUT
9      Immagazzina il contenuto di X in 1024
10     Carica 1 in Accumulatore
11     Immetti il valore dell' Acc. (1) in RAM per
dare il colore bianco.
12     Ritorno da subroutine.
```

Naturalmente l'istruzione CPX ha la sua corrispondente in Y:

CPY Compare the contents of specified location with those in the Y register.

Cioe' confronta il risultato di una specifica locazione di memoria con il valore contenuto nel registro Y.

Il cui risultato e commento e' quindi esattamente eguale a quello visto in precedenza per X.

Esercizio 2.3

Riscrivere il programma 2.5 usando il registro Y ed alla fine del ciclo far stampare a 1034 un cuore porpora.

Ricordarsi che il colore porpora viene dato dal valore 4 in RAM anziche' da 1 come per il bianco.

La terza istruzione di confronto e':

CMP CoMPare the contents of the specified location with the Accumulator.

Cioe' confronta il contenuto di una particolare locazione di memoria con l' Accumulatore.

Questa istruzione e' particolarmente utile perche' il risultato di tutte le operazioni aritmetiche e' depositato nell'Accumulatore e quindi CMP consente un confronto diretto fra un dato valore e la "risposta".
 Un esempio di cio' viene fornito nel programma seguente:

Programma 2.6

```
ROUT = $0340
1          *          = 828
2  033C A2 00          LDX #0
3  033E A9 53          LDA #83
4  0340 E8          ROUT INX
5  0341 8E 84 03      STX 900
6  0344 CD 84 03      CMP 900
7  0347 D0 F7          BNE ROUT
8  0349 8E 00 04      STX 1024
9  034C A9 01          LDA #1
10 034E 8D 00 D8      STA 55296
11 0351 60          RTS
```

- 1 Indirizzo di partenza
- 2 Carica 0 in X
- 3 Carica 83 (il cuore) in A
- 4 Incrementa X
- 5 Immagazzina X in 900
- 6 Confronta A con 900
- 7 Salta se diverso
- 8 Immagazzina X in 1024
- 9 Carica 1 in A
- 10 Immetti 1 in 55296 per il colore bianco
- 11 Ritorno da subroutine

I FLAGS DEL 6510

Fino a questo momento abbiamo lavorato solo su 1 dei 7 flags disponibili sul 6510.

Vediamo quali sono gli altri

BIT N.	7	6	5	4	3	2	1	0
FLAG	N	V	-	B	D	I	Z	C

Diamo ora un sommario di questi Flags che vedremo uno per uno, come abbiamo fatto per Z, nel corso del manuale.

N FLAG NEGATIVE.

Viene settato o messo a 1 quando il risultato di un' operazione aritmetica e' negativo

V OVERFLOW FLAG.

Viene settato quando i risultati di un' operazione aritmetica vanno in overflow dal bit 6 al 7.

B BREAK FLAG

Viene messo a 1 quando avviene un' interruzione di programma messa in funzione da un' istruzione BRK.

D DECIMAL FLAG

A 1 quando si opera in modo decimale

I INTERRUPT FLAG

Viene messo a 1 quando opera una sequenza di Interrupt.

Z ZERO FLAG

E' stato abbondantemente spiegato

C CARRY FLAG

Indica la presenza di un Carry, cioe' di un riporto durante un' operazione aritmetica.
Messo a 1 anche durante le operazioni di SHIFT o ROTATE per indicare la possibile perdita di un bit.

IL FLAG N

Questo Flag, cioe' il NEGATIVE FLAG, viene messo

a 1 quando la risposta a un' operazione e' un risultato negativo.

Puo' essere controllato da due istruzioni:

BMI Branch on Minus

Un' istruzione come BMI PIPPO eseguirà il controllo sul flag N e se questi e' a 1 allora salterà alla locazione la cui Label e' PIPPO.

Un esempio d' uso di BMI e' dato nel programma seguente, in cui il contenuto di Y e' incrementato fino a quando un comando CPY da' un meno e si passa all' istruzione BMI:

Programma 2.8

```
ROUT =    $0343
1          *          =    828
2    033C A9 5A          LDA    #90
3    033E 8D 84 03      STA    900
4    0341 A0 00          LDY    #0
5    0343 C8           ROUT    INY
6    0344 CC 84 03      CPY    900
7    0347 30 FA          BMI    ROUT
8    0349 8C 00 04      STY    1024
9    034C A9 01          LDA    #1
10   034E 8D 00 D8      STA    55296
11   0351 60           RTS
```

Vediamo ora un breve commento al programma.

- 1 Indirizzo iniziale
- 2 Carica 90 in A
- 3 Immagazzina il contenuto di A in 900
- 4 Carica 0 in Y
- 5 Incrementa Y
- 6 Confronta il contenuto di 900 con il contenuto di Y
- 7 Controlla il flag N
- 8 Immetti il contenuto di Y in 1024
- 9 Carica 1 in A
- 10 Metti il colore in RAM
- 11 Ritorno da subroutine

In aggiunta al comando BMI il flag N puo' essere controllato da:

BPL Branch on PLus

Un' istruzione come BPL PIPPO controllera' il contenuto del Flag N e se questi non e' a 1 eseguirà un salto a PIPPO.
Vediamone un' applicazione nel seguente programma:

Programma 2.9

```

ROUT =      $0343
1          *          =      828
2      033C A9 5B          LDA      #91
3      033E 8D 84 03          STA      900
4      0341 A0 64          LDY      #100
5      0343 88          ROUT      DEY
6      0344 CC 84 03          CPY      900
7      0347 10 FA          BPL      ROUT
8      0349 8C 00 04          STY      1024
9      034C A9 07          LDA      #7
10     034E 8D 00 08          STA      55296
11     0351 60          RTS

```

Vediamo la spiegazione linea per linea:

```

1      Indirizzo di partenza
2      Carica 91 in A
3      Immagazzina il contenuto di A in 900
4      Carica 100 in Y
5      Decrementa Y
6      Confronta il contenuto di 900 con il
contenuto di Y
7      Controlla il flag N
8      Immetti il contenuto di Y in 1024
9      Carica 1 in A
10     Metti il colore in RAM
11     Ritorno da subroutine

```

Il risultato dell' esecuzione di questo programma sarà un quadri (91) giallo (7) in 1024.

Esercizio 2.5

Scrivere un programma usando BPL per saltare

quando il registro X arriva a 0 essendo stato decrementato a partire da 90. A questo punto visualizzare l'attuale valore di X.

CAPITOLO TERZO

Uno dei vantaggi della programmazione in codice macchina e' la velocita' di esecuzione e questo naturalmente facilita la visualizzazione dei risultati.

Le animazioni, i giochi ad esempio, possono essere migliorate, come velocita' esecutiva, usando un comando come:

```
STA LOC,X  STore the contents of Accumulator in
the specified address index with the X register
```

Cioe' immagazzina il contenuto dell' Accumulatore in una locazione di memoria indicizzato dal contenuto del registro X.

Questo vuol dire che se X contiene 100 e A 90, l'istruzione:

```
..... STA 1024,X
```

immetterà il simbolo dei quadri in (1024+100).

Quando si usa con un' istruzione incrementale consente alla locazione di schermo di essere indicizzata.

Vediamone una dimostrazione con il programma 3.1

Programma 3.1

ROUT =	\$033E					
1				*	=	828
2	033C	A2	64		LDX	#100
3	033E	A9	5A	ROUT	LDA	#90
4	0340	9D	FF	03	STA	1023,X
5	0343	A9	01		LDA	#1
6	0345	9D	00	D8	STA	55296,X
7	0348	CA			DEX	
8	0349	D0	F3		BNE	ROUT
9	034B	60			RTS	

Vediamone la spiegazione

1	Indirizzo di partenza
2	Carica 100 in X
3	Carica 90 (DIAMOND) in A
4	Visualizza (DIAMOND) a (1023+X)
5	Carica 1 in A
6	Scrive il colore BIANCO a (55296+X)
7	Decrementa il valore di X
8	Salta se diverso
9	Ritorno da subroutine

Quando questo programma gira, mette il quadri nelle prime 100 locazioni di schermo.

Naturalmente il comando visto per X ha il suo corrispondente nell'uso del registro Y.

STA LOC,Y Store the contents of Accumulator in the specified address indexed with the Y register.

Cioe' immagazzina il contenuto dell' Accumulatore in una locazione di memoria indicizzata dal contenuto del registro Y.

Esercizio 3.1

Modificare il programma 3.1 usando il registro Y invece di X. Usare solo comandi diretti di POKE.

Esercizio 3.2

Stampare un asterisco nelle prime 100 locazioni di schermo usando un comando di incremento INX.

NOTA

Si consiglia di cercare di risolvere questo esercizio prima di proseguire.

Nell' esercizio 3.2 l' istruzione di BRANCH era attivata dallo zero generato da un comando di confronto.

Tuttavia se il registro X o Y e' incrementato oltre il 255 il suo valore torna a zero e resetta il flag Z.

Se e' stato fissato con un appropriato valore puo' essere usato per saltare senza confronto.

Il programma qui sotto (3.2) consente una funzione simile a quella vista con il programma

3.1 ma usa il INX, cioe' l' incremento, invece di DEX.

Questo programma di per se non offre particolari guadagni rispetto al precedente, ma in particolari situazioni puo' essere piu' vantaggioso.

Programma 3.2

```
ROUT = $033E
1      * = 828
2      033C A2 D8      LDX #216
3      033E A9 2A      ROUT LDA #42
4      0340 9D 28 04   STA 1064,X
5      0343 A9 01      LDA #1
6      0345 9D 28 D8   STA 55336,X
7      0348 E8        INX
8      0349 D0 F3      BNE ROUT
9      034B 60        RTS
```

Vediamo ora un programma che puo' essere usato anche come routine da aggiungere ad altri programmi, che serve a muovere un carattere sullo schermo.

Questo risultato puo' essere ottenuto in programmi Basic con una serie di istruzioni POKE.

Programma 3.3

INCR =	\$0348					
1				*	=	828
2	033C	A2	00		LDX	#0
3	033E	A0	20		LDY	#32
4	0340	8C	84	03	STY	900
5	0343	A9	5A		LDA	#90
6	0345	8D	85	03	STA	901
7	0348	9D	00	04	INCR	STA 1024,X
8	034B	A9	01		LDA	#1
9	034D	9D	00	D8	STA	55296,X
10	0350	98			TYA	
11	0351	9D	FF	03	STA	1023,X
12	0354	AD	85	03	LDA	901
13	0357	E8			INX	
14	0358	D0	EE		BNE	INCR
15	035A	60			RTS	

Quando gira, il programma esposto fa muovere il simbolo dei quadri, in bianco, lungo lo schermo fino a 1279.

Come abbiamo detto in precedenza, il programma 3.3, pur essendo uno dei tanti sistemi con il quale si puo' scrivere un programma, forse non e' il migliore, pur funzionando, cioe' risolvendo lo scopo per il quale e' stato scritto.

Successivamente in questo capitolo ne vedremo una versione migliore.

Vediamo ora invece un problema legato anche a questo tipo di programma: la temporizzazione.

LA TEMPORIZZAZIONE DEI PROGRAMMI

Il programma 3.3 mostra con notevole efficacia uno dei problemi della programmazione in codice macchina: la velocità.

Mentre, di solito in Basic non è quasi mai necessario diminuire la velocità che è già lenta di suo, altrettanto non può dirsi per quanto riguarda il Codice Macchina.

Il microprocessore 6510 ricava la sua VELOCITÀ OPERATIVA da un clock interno (un oscillatore al cristallo di quarzo molto preciso) che nel caso del CBM64 gira a 2 MHz (due MegaHertz) o due milioni di cicli al secondo.

Così ogni ciclo richiede mezzo-milionesimo di secondo e la velocità operativa delle varie istruzioni sarà riferita al numero di cicli necessari per la loro esecuzione.

Alcune di queste operazioni prendono un posto entro il microprocessore e sono eseguite in maniera molto più veloce di altre che invece devono andare a prendere dati dalla memoria.

Per esempio l'istruzione TAX prende 2 cicli, mentre per eseguire STA (OPER,X) ne sono necessari 6.

Naturalmente la conoscenza del tempo richiesto per l'esecuzione del ciclo di istruzione è importante per determinare la velocità operativa del programma e consente di usare correttamente il clock da 2MHz per i cicli e per i ritardi programmati.

Ritornando a vedere l'esecuzione del programma 3.3, può essere calcolato il tempo che il

simbolo (DIAMOND) rimane sullo schermo.

La tavola sotto riporta le istruzioni del programma, i cicli esecutivi e la sommatoria dei tempi, relativamente alla visualizzazione del carattere e quindi alla effettiva esecuzione di quella parte di programma.

COMANDO	TEMPO/CICLO	SOMMAT.TEMPI
STA 1024,X	-	0
LDA& 1	2	2
STA 55296,X	5	7
TYA	2	9
STA 1023,X	5	14
LDA 901	4	18
INX	2	20
BNE 243	2	22
STA 1024,X	5	27
LDA& 1	2	29
STA 55296,X	5	34
TYA	2	36
STA 1023,X	5	41

Così vediamo che dal momento dell'apparizione del carattere al momento in cui lo stesso è cancellato (o sovrascritto da un BLANK che è la stessa cosa), sono necessari o passano 41 cicli per un totale di 20,5 microsecondi.

I 256 caratteri saranno perciò scritti in 5248 micro-secondi o 5,2 circa milli-secondi. Un tempo chiaramente troppo breve perché l'occhio umano possa seguirlo.

Per avere quindi qualcosa di visibile è necessario programmare un ritardo (DELAY).

Il programma seguente mostra un semplice ciclo di ritardo.

Programma 3.4

```

DECRX = $033E
1          *          =      828
2  033C  A2  FA          LDX  #250
3  033E  CA          DECRX  DEX
4  033F  D0  FD          BNE  DECRX
5  0341  60          RTS

```

Cio' da' un ritardo di 5 cicli per giro (ignorando i 2 cicli per l'istruzione LDX) o $250 \times 5 = 1250$ cicli per esecuzione.

Dopo aver provato che giri correttamente si può aumentare il ritardo, al momento di 625 micro-secondi, inserendo in questo programma di ritardo altre istruzioni o congiungendolo con un altro programma di ritardo come mostrato di seguito.

Programma 3.5

```
RIT1 =    $033E
RIT2 =    $0340
1          *          =    828
2    033C A0  C8          LDY    #200
3    033E A2  FA          RIT1   LDX    #250
4    0340 CA              RIT2   DEX
5    0341 D0  FD          BNE    RIT2
6    0343 88              DEY
7    0344 D0  FB          BNE    RIT1
8    0346 60              RTS
```

Quando gira completamente la subroutine DEX dovrebbe dare un ritardo aggiuntivo di 200 x 625 micro-secondi o 1/8 di secondo.

Qualora si desideri usare il computer come temporizzatore di precisione o comunque quando si abbia necessita' di misurare il tempo con assoluta precisione e' chiaro che non si puo' mettere un ritardo qua e la' ne ignorare i 2 microsecondi come abbiamo fatto per semplificare il problema per l'istruzione LDX&.

Sara' invece vitale assicurarsi l' assoluta certezza del calcolo dei tempi.

In particolare e' necessario fare attenzione alle istruzioni di BRANCH.

L'istruzione BNE nel programma 3.5 normalmente necessita di tre cicli, per esempio quando il BRANCH ha successo o viene eseguita.

Tuttavia quando non viene eseguita e quindi il programma passa oltre sono necessari solo 2 cicli.

In altre condizioni se il BRANCH rimanda il programma ad altra pagina di memoria sono necessari due cicli addizionali.

Un altro problema poi tipico del CBM 64 e' che l'integrato VIC che controlla la visualizzazione di schermo puo' interferire con la temporizzazione. In particolare quando si manipola la grafica e specialmente gli SPRITES il VIC-II ha veramente molto lavoro da compiere. Spesso infatti deve prendere il controllo completo delle operazioni condotte ed allora al 6510 non resta che attendere.

Cio' porta alla conclusione che se realmente si desidera un' accurata temporizzazione e' necessario arrestare l'attivita' di visualizzazione del VIC-II, eseguire il lavoro o la parte di programma che necessita di tempi precisi e dopo fare ripartire il lavoro di visualizzazione.

Queste operazioni non sono pero' particolarmente difficili in Assembler.

Tuttavia in questo momento siamo interessati in modo particolare a ritardi di animazioni sullo schermo e continuiamo a vedere sia come operano sia come si inseriscono in programma.

Il programma 3.6 usa il precedente 3.3 come base di lavoro ed inserisce il ciclo di ritardo illustrato in 3.4 immettendo un tempo di 0.6 millisecondi fra la visualizzazione e la cancellazione del carattere.

Programma 3.6

```

INCREM = $034D
RIT = $035B
1
2 033C A0 00 * = 828
3 033E A9 5A LDY #0
4 0340 8D 84 03 LDA #90
5 0343 A9 01 STA 900
6 0345 8D 85 03 LDA #1
7 0348 A9 20 STA 901
8 034A 8D 86 03 LDA #32
9 034D A2 FA INCREM LDX #250
10 034F AD 84 03 LDA 900
11 0352 99 00 04 STA 1024,Y
12 0355 AD 85 03 LDA 901
13 0358 99 00 D8 STA 55296,Y
14 035B CA RIT DEX
15 035C D0 FD BNE RIT
16 035E AD 86 03 LDA 902
17 0361 99 00 04 STA 1024,Y
18 0364 C8 INY
19 0365 D0 E6 BNE INCREM
20 0367 60 RTS

```

Provate ad inserire questo programma ed a farlo girare. Non riuscite a vedere niente!!!

La verita' e' che 0.6 millisecondi non bastano.

Lo schermo televisivo ha un REFRESH di 1/50 di secondo (sistema europeo PAL) o di 1/60 di

secondo (sistema americano NTSC) così che per la scansione di schermo sono necessari 16/20 millisecondi.

Se il nostro carattere bianco è sullo schermo solo per un terzo di questo tempo, vorrà dire che vedremo SOLO un terzo delle immagini che si desiderava visualizzare.

NOTA

In effetti a causa del così detto INTERLACE la situazione è leggermente migliore, cioè si vedono un po' più di immagini di quelle teoricamente visibili. Tuttavia ciò non risolve il problema.

In ogni caso quindi il ritardo prodotto non è sufficiente e va aumentato. Come?

Il registro X può manipolare un massimo di 255 per cui l'unica soluzione è di far ricorso, allo stesso modo che si farebbe con il Basic, ad una SUBROUTINE DI TEMPORIZZAZIONE o, nel caso non sia sufficiente, a più temporizzazioni.

Programma 3.6A

```

INCLOC = $034D
LOOPA = $035E
LOOPB = $0360
1
2      033C A0 00      *      =      828
3      033E A9 5A      LDY #0
4      0340 8D 84 03   STA 900
5      0343 A9 01      LDA #1
6      0345 8D 85 03   STA 901
7      0348 A9 20      LDA #32
8      034A 8D 86 03   STA 902
9      034D AD 84 03   INCLOC LDA 900
10     0350 99 00 04   STA 1024,Y
11     0353 AD 85 03   LDA 901
12     0356 99 00 D8   STA 55296,Y
13     0359 8C 87 03   STY 903
14     035C A0 0F      LDY #15
15     035E A2 FA      LOOPA LDX #250
16     0360 CA      LOOPB DEX
17     0361 D0 FD      BNE LOOPB
18     0363 88      DEY
19     0364 D0 F8      BNE LOOPA
20     0366 AC 87 03   LDY 903
21     0369 AD 86 03   LDA 902
22     036C 99 00 04   STA 1024,Y
23     036F C8      INY
24     0370 D0 DB      BNE INCLOC
25     0372 60      RTS

```

Inserire questo programma e farlo girare.
Questo dimostra il motivo per cui programmi e giochi scritti in codice macchina funzionano in modo tanto migliore di quelli scritti in Basic.

MODI DI INDIRIZZAMENTO

Nel programma precedente abbiamo utilizzato l'istruzione STA (LOC,Y) riuscendo a far muovere il nostro carattere nelle prime locazioni dello schermo e cio' semplicemente incrementando il valore dell' indice Y.

In effetti il comando STA ha numerosi modi che dipendono dall' indirizzamento usato.

Si puo' quindi dire che l' indirizzamento e' una modifica del comando fatta per cambiare la sua funzione in modo particolare.

L' indirizzamento fa si che il 6510 punti (o sia puntato o indirizzi) ad una locazione di memoria sia direttamente che indirettamente.

La strada seguita dipende dal modo particolare di indirizzamento usato.

Gli indirizzamenti sono uniformi attraverso tutti i 64 K di memoria disponibile tranne che per i primi 256 Bytes di memoria (dalla locazione 0 alla 255).

Per indirizzare queste locazioni (o prima pagina di memoria) e' necessario solo 1 Byte mentre tutte le altre pagine necessitano di 2 Bytes.

NOTA

Ricordiamo che l' intera mappa di memoria del CBM64 puo' essere divisa in pagine ognuna delle quali di 256 Bytes e che la prima pagina e' chiamata appunto PAGINA ZERO che ha uno speciale modo di indirizzamento che vedremo nel seguito di questo capitolo.

Ricordiamo inoltre che quando a seguito di un comando si salta da una pagina all'altra a livello di temporizzazione verrà usato un ciclo addizionale.

INDIRIZZAMENTO IMPLICITO

Questo modo, chiamato qualche volta anche indirizzamento inerente, è probabilmente il più facile da usare in quanto è il 6510 ad eseguire tutto il lavoro.

Con questo modo possono essere utilizzate numerose istruzioni come TYA, TXA, RTS in quanto il 6510 stesso calcola gli indirizzi.

Fondamentalmente le istruzioni possono dividersi in due gruppi separati.

Nel primo gruppo possono essere messe le istruzioni che sono eseguite interamente entro il 6510 come TYA che trasferisce Y in A e quindi tutto avviene all'interno del microprocessore.

Nel secondo gruppo possiamo mettere invece le istruzioni dove è necessario un riferimento esterno come per esempio RTS.

Le istruzioni del primo gruppo sono:

DEX DEY INX INY TAX TXA TYA CLC CLD CLI CLV NOP
SEC SED SEI.

Quelle del secondo gruppo:

RTS BRK PHA PHP PLA PLP RTI.

INDIRIZZAMENTO ASSOLUTO

Le istruzioni usate in questo modo sono facili da comprendere in quanto l'operando dell'istruzione (il numero che viene accanto all'istruzione stessa) e' un numero di 2 Bytes che definisce appunto l'indirizzo in modo assoluto. In questo modo, per esempio, nel programma 3.6 l'istruzione STA 901 comunica al registro X ESATTAMENTE dove immagazzinare il suo contenuto.

Le istruzioni che utilizzano questa forma di indirizzamento sono elencate di seguito ed in parte sono gia' state viste mentre altre le vedremo in seguito:

ADC CMP CPX CPY JMP JSR LDA LDX LDY STA STX STY
AND EOR ORA SBC

INDIRIZZAMENTO IN PAGINA ZERO

Questa forma di indirizzamento e' in realta' una sotto-forma dell'indirizzamento assoluto solo che l'operando e' ristretto ad un Byte cioe' massimo 256 caratteri.

Il maggior vantaggio di questo tipo di indirizzamento e' la velocita' di esecuzione perche' le istruzioni sono eseguite in soli tre cicli invece che in quattro come nell'indirizzamento assoluto normale.

A causa della maggior velocita' di esecuzione la pagina zero e' adoperata quasi per intero dal

Sistema Operativo e dall' interprete BASIC per cui non e' realmente diponibile per l' Assembler. Al momento si possono utilizzare le locazioni di pagina zero da 251 a 254. Quando ne saprete di piu' sull' interpete Basic potrete utilizzare altre locazioni di questa pagina ed addirittura spostare la pagina zero con il suo contenuto da altre parti della memoria. Considerazioni piu' approfondite esulano pero' dagli scopi di questo manuale.

Malgrado sia pericoloso utilizzare le locazioni di questa pagina si possono pero' leggere ed utilizzare le informazioni qui contenute.

Tre locazioni utili di questa pagina possono essere la 160, 161 e 162 che contengono il valore del clock o JIFFIES CLOCK o OROLOGIO (espresso in ore, minuti e secondi) che si incrementa ogni 1/60 di secondo.

Il programma 3.7 e' un semplice programma che carica uno di questi valori in A e lo stampa sullo schermo.

Programma 3.7

1			*	=	828
2	033C	A5	A0	LDA	160
3	033E	8D	00 04	STA	1024
4	0341	A9	01	LDA	#1
5	0343	8D	00 D8	STA	55296
6	0346	60		RTS	

INDIRIZZAMENTO IMMEDIATO

Questo modo di indirizzamento consente che un numero sia caricato immediatamente entro un registro o per essere usato direttamente come termine di un confronto.

Tutti i comandi in modo immediato sono riconoscibili in questo volume perché nella istruzione viene aggiunto il suffisso & (Pound).

Fino ad ora sono stati visti numerosi esempi del modo di INDIRIZZAMENTO IMMEDIATO come per esempio nel programma 3.6.

In questo programma l' Accumulatore era caricato direttamente usando LDA& 32, mentre in altri programmi sia il registro X che il registro Y sono stati caricati con lo stesso sistema.

Anche altre istruzioni possono essere usate in modo immediato come vediamo nel programma di seguito.

Questo programma mostra inoltre l' uso di una nuova istruzione:

CPY& ComPare Y with value specified in Immediate Mode.

Cioe' confronta con il valore specificato in Modo Immediato.

Programma 3.8

```

INCLOC = $033E
1          *          =      828
2  033C A0 00          LDY  #0
3  033E 98          INCLOC TYA
4  033F C8          INY
5  0340 99 FF 03     STA  1023,Y
6  0343 A9 01          LDA  #1
7  0345 99 FF D7     STA  55295,Y
8  0348 C0 64          CPY  #100
9  034A D0 F2          BNE  INCLOC
10 034C 60          RTS

```

Commento

```

1  Inizio a 828
2  Carica Y con 0
3  Trasferisci Y in A
4  Incrementa Y
5  Immagazzina il contenuto di A in 1023+Y
6  Carica A con 1
7  Immagazzina in A 55295 + Y
8  Confronta Y con 100
9  Salta se il flag Z non e' stato settato
10 Ritorno da subroutine

```

Quando gira questo programma stampa i primi 100 caratteri del set di caratteri in memoria sulle prime 100 locazioni di schermo.

INDIRIZZAMENTO INDICIZZATO

In questo modo un indirizzo viene calcolato usando il contenuto di un registro aggiunto ad un dato indirizzo.

E' stato usato di frequente per stampare caratteri sullo schermo con la forma STA (LOC,X) e STA (LOC,Y).

Nel programma 3.8 STA (LOC,Y) era stato usato in questo modo con il comando STA 1024,Y.

Quando si usa questo sistema di indirizzamento bisogna fare attenzione perche' il modo di comportarsi del registro X rispetto al registro Y e' diverso.

Entrambi i registri possono essere usati con istruzioni di indicizzamento assoluto, per esempio operando con due Bytes.

ATTENZIONE !!!!

Eccezioni da ricordare:

1) STY non puo' essere indicizzato con X

2) ASL DEC LSR ROL ROR non possono essere indicizzati con Y.

3) La pagina ZERO non puo' essere indicizzata MAI con Y.

I codici mnemonici che devono essere usati in pagina ZERO devono avere l'indirizzo della pagina che sarà costituito da 1 solo Byte.

NOTA

Non è possibile ovviamente usare in questo modo i due comandi STX e LDX.

INDIRIZZAMENTO RELATIVO

Molti programmi usati fino a questo momento hanno utilizzato indirizzi relativi, nei quali un salto è stato definito relativamente all'attuale posizione del programma.

Per esempio l'operando che esprime la posizione desiderata.

Nel programma 3.8 l'istruzione BNE PIPPO è stata usata per controllare che il flag Z fosse fissato e di saltare qualora non fosse stata verificata quella condizione.

Tutte le istruzioni di salto usate in questo modo utilizzano l'indirizzamento relativo.

Il gruppo è composto dai seguenti comandi:

BCC BCS BEQ BMI BNE BPL BVC BVS

INDIRIZZAMENTO INDIRETTO

Questo e' allo stesso tempo il piu' complesso ed il piu' versatile di tutti i modi di indirizzamento.

Questo modo prende il nome di INDIRETTO dal fatto che l'operando e' un puntatore e non un indirizzo.

Ed e' questo puntatore che dirige il 6510 attraverso le locazioni di memoria che contengono l'indirizzo.

Ancora una volta tuttavia i meccanismi di indicizzazione di X e Y differiscono fra loro in misura considerevole e danno luogo a diversi modi di indirizzamento.

Tutte le istruzioni che utilizzano questo metodo sono riconoscibili in assembler perche' contengono o un suffisso (LOC,X) o (LOC,Y) ed hanno un operando di 1 Byte.

A causa di cio' possono puntare solo a locazioni in pagina zero e percio' sono sottoposte alle stesse restrizioni gia' viste per gli altri comandi in pagina zero.

USO DEL REGISTRO X

Con un indirizzamento indiretto che usi il registro X, l'operando e' indicizzato (aggiunto) con il contenuto dello stesso registro per produrre il puntatore.

Questa locazione e quella immediatamente successiva sono quindi esaminate ed i loro contenuti forniscono gli indirizzi per i data richiesti con l'ordine seguente:

Byte meno significativo (LSB)

Byte piu' significativo (MSB)

Questa tecnica e' utile per esaminare un particolare elemento in una tavola, essendo fissata l' attuale posizione della tavola dal valore del registro X.

Data la scarsa disponibilita' dello spazio sulla pagina zero sul CBM64 il modo di indirizzamento e' di uso limitato, tuttavia, a scopo dimostrativo, faremo vedere un programma dove viene usata una istruzione di questo modo.

L' istruzione e':

```
LDA LOC,X  Load A indirectly indexed with X
```

Cioe' carica l' Accumulatore con l' indirizzo indiretto indicizzato con il contenuto di X.

Nel nostro caso e' usata per trovare 4 bytes immagazzinati in pagina ZERO da 84 a 88.

Programma 3.8

```
LOOP =    $033E
1          *          =    828
2    033C A2 00          LDX    #0
3    033E B5 54          LDA    84,X
4    0340 9D 00 04      STA    1024,X
5    0345 E8          INX
6    0346 E0 04          CPX    #4
7    0348 D0 F6          BNE    LOOP
8    034A 60          RTS
```


Commento

```
1   Inizio
2   Carica X immediato con 0
3   Carica A indiretto 84+X
4   Immagazzina A in 1024+X
5   Incrementa X
6   Esegui un confronto immediato di X con 4
7   Vai a 241 se non uguale
8   Ritorno da Subroutine
```

Quando questo programma gira, saranno visualizzati 4 caratteri nelle prime 4 locazioni di schermo.

Questi caratteri differiranno in base a cio' che stava facendo il Basic per ultimo.

Quando si usa questa routine in un programma i quattro numeri dovrebbero formare due indirizzi con il seguente ordine:

Carattere 1	Indirizzo 1	LSB
" 2	" 1	MSB
" 3	" 2	LSB
" 4	" 2	MSB

Questo tipo di indirizzamento e' conosciuto come **INDIRIZZAMENTO INDICIZZATO INDIRETTO** o, molto piu' chiaramente **INDIRIZZAMENTO PREINDICIZZATO INDIRETTO**.

Infatti, come e' implicito nel nome stesso, questo indirizzamento e' preindicizzato poiche'

il valore di X e' aggiunto prima che il 6510 salti all' indirizzo.

USO DEL REGISTRO Y

Usando l' indirizzamento indiretto con il registro Y si opera in modo differente, poiche' l' istruzione operando punta direttamente ad una locazione di memoria in pagina zero.

Questa contiene il LSB dell' indirizzo e la successiva locazione di memoria contiene il MSB.

Finalmente il contenuto indicizzato del registro e' aggiunto a questo indirizzo per formare l' indirizzo finale indicizzato.

Non deve sorprendere quindi se questa forma e' chiamata anche **INDIRIZZO INDIRETTO**

POSTINDICIZZATO in quanto l' indicizzazione e' calcolata DOPO che l' indirizzo e' stato trovato.

L' interprete Basic ed il Sistema Operativo del CBM64 fanno un uso molto esteso di questa istruzione.

Quando avrete una maggiore confidenza con l' uso dell' Assembler potrete vedere come lavori il Basic e trarre notevole vantaggio nell' uso delle routines del Basic stesso ed in generale del Sistema Operativo del computer.

INDIRIZZAMENTO INDIRETTO ASSOLUTO

Questo modo di indirizzamento e' usato con una sola istruzione:

JMP (LOC) JuMP indirectly addressed

Cioe' salta ad un indirizzo indiretto

E' questa un' istruzione assoluta nel quale l'operando e' un indirizzo di 2 bytes e puo' quindi indirizzare una qualsiasi locazione di memoria. E' tuttavia indiretto in quanto a quella locazione ed a quella successiva trova l'indirizzo (prima LSB e poi MSB) per l'istruzione di salto.

CAPITOLO QUARTO

Nelle prime pagine ed in particolare nel programma 1.1 avevamo fatto un esempio di somma e di visualizzazione del risultato.

La semplicità del programma e dei numeri da sommare veniva dal fatto che erano numeri di un solo digit e che la risposta non richiedeva il riporto.

Quando è necessario operare su numeri di dimensioni maggiori allora il 6510 li manipola usando il suo riporto o CARRY o C FLAG.

Usando un Byte è possibile contare solo fino a 255, per cui se vogliamo contare oltre dobbiamo usare due Bytes.

Questi 16 Bits consentono allora di contare fino a 65535.

È possibile manipolare naturalmente numeri di dimensioni molto più grandi di questi, tuttavia per il momento ci limiteremo a descrivere operazioni con solo due Bytes che vengono chiamate:

OPERAZIONI IN DOPPIA PRECISIONE

Se due o più Bytes devono essere utilizzati per rappresentare uno stesso numero allora si deve creare un legame (LINK) fra il primo ed il secondo Byte tramite un meccanismo univoco.

Questa e' la funzione del CARRY.

Il suo funzionamento e' provato dall'istruzione seguente:

BCC Branch on Carry Clear

Questa istruzione controlla che il CARRY sia posto a 0 ed esegue un salto in caso positivo (cioe' se e' 0).

Tuttavia e' sempre bene osservare una precauzione quando si esegue un controllo di questo FLAG.

La precauzione e' di assicurarsi che il flag sia nello stato desiderato prima dell'operazione che eventualmente possa modificarlo.

L'istruzione per eseguire questa funzione e' la seguente:

CLC CLear the Carry

Cioe' PULISCI o metti a 0 il flag di CARRY o semplicemente Carry.

Programma 4.1

```
SOMMA1 = $033E
1          *          =      828
2      033C 18          CLC
3      033D A9 00          LDA #0
4      033F 69 01      SOMMA1  ADC #1
5      0341 90 FC          BCC SOMMA1
6      0343 8D 00 04      STA 1024
7      0346 A9 01          LDA #1
8      0348 8D 00 D8      STA 55296
9      034B 60          RTS
```

Quando gira, questo programma incrementa progressivamente il contenuto dell' Accumulatore di 1 fino a 255.

L' istruzione ADC& gira gli otto valori 1 in otto 0 e fissa il Carry a 1. Così che quando l' Accumulatore e' visualizzato con l' istruzione STA 1024 si vede il contenuto 0 (esempio una a commerciale bianca sullo schermo).

Il 6510 ha una seconda istruzione di controllo per il Carry:

BCS Branch on Carry Set

Questa istruzione controlla che il Carry sia settato o fissato, per esempio che contenga un 1, e se il controllo da' un risultato positivo, esegue un salto.

Il seguente programma illustra l' uso di questa istruzione:

Programma 4.2

```
SOMMA1 = $033E
FINE = $0345
1          *          =      828
2  033C A9 00          LDA  #0
3  033E 69 01          SOMMA1 ADC  #1
4  0340 B0 03          BCS  FINE
5  0342 4C 3E 03      JMP  SOMMA1
6  0345 8D 00 04      FINE  STA  1024
7  0348 A9 01          LDA  #1
```



```

8  034A 8D 00 D8          STA  55296
9  034D 60                RTS

```

Ancora una volta questo programma riempie gli otto bits dell' Accumulatore con dei valori 1, fissa il Carry e termina.

Al termine l' Accumulatore conterra' tutti 0 e per questo la solita a commerciale bianca sara' visualizzata sullo schermo.

Proviamo ora ad addizionare due numeri maggiori del valore 255 che sappiamo essere il massimo esprimibile con un solo byte.

Primo di tutto bisogna calcolare l' MBS e il LBS e per far questo e' necessario passare dal formato decimale a quello esadecimale, al quale per distinguerlo metteremo il prefisso \$.

Per far questo mostriamo un breve procedimento che fa uso della parte comandi Basic del Computer:

INT (1257/4096) = 0 I carattere = 0

INT (1257/256) = 4 II " = 4

INI (1257-4x256)/16 = 8 III " = 8

(1297-4x296-9x16) = 5 IV " = 5

Per cui sara':

1257 equivale in esa a 0485

di cui la parte:

MSB = 04

LSB = 85

Per sommare due valori 1257 dobbiamo per prima cosa addizionare i loro LSB, controllare se c'è un Carry (cioè un riporto) e dopo aggiungere l'MSB tenendo conto della presenza o della mancanza del carry.

```
..... 85+
..... 85
..... --
piu' il Carry 0A
```

16, 10 = Carry + 0A

Dopo si esegue l'addizione sugli MSB

```
..... 04
..... 04
..... --
..... 08
```

Dopo di che si aggiunge il Carry

```
..... 08 + Carry = 09
```

Nella spiegazione abbiamo omesso di dire "+

Carry" ed e' questa l' operazione che il Flag C esegue per conto del programmatore.

Il Flag infatti e' messo a 1 quando l' operazione ha un riporto.

La successiva operazione tiene allora conto di questo riporto e' aggiunge 1 alla somma.

Vediamo come si comportano con due brevi esempi i risultati di due somme con diversi valori nel Carry:

Con Carry a 0

$$04 + 04 = 08$$

Con Carry a 1

$$04 + 04 = 09$$

In questo modo la risposta all'esempio precedente e' in esa \$090A o:

$$9 \times 256 + 10 = 2314 \text{ in decimale}$$

Vediamo ora di rifare i calcoli invece che a mano con il computer.

Noi possiamo contare sulla capacita' di manipolazione, da parte del 6510 del Carry, ma non possiamo invece contare sulla sua capacita' di riconoscere quando usarlo.

Tutto il lavoro in doppia precisione e' eseguito prima LSB come durante l' operazione di somma ed

il Carry e' immagazzinato per la parte di addizione con il MSB.

Si deve ricordare che quando il 6510 e' usato in comandi di indirizzamento indiretto, questi immagazzina prima il LSB dell' indirizzo e poi il MSB. Questo e' l' ordine usato quando l' indice e' aggiunto al puntatore indirizzo.

Si potrebbe utilizzare questa forma noi stessi quando si immagazzinano NUMERI (naturalmente distinti dall' indirizzo).

Per assicurarsi che l' addizione LSB non sia variata dal valore del Carry e' importante far precedere la somma stessa dall' istruzione CLC (Clear Carry).

Prima di tutto dobbiamo calcolare il valore di MSB e LSB in decimale, poiche' entrambi i metodi di immissione dati in memoria lo richiedono.

Per LSB il suo valore decimale sara':

$$8 \times 16 + 5 = 133$$

mentre per MSB e':

$$0 \times 16 + 4 = 4$$

Ora scriviamo il programma, ma prima di questo introduciamo una nuova istruzione:

NOP No OPeration

Cioe' nessuna operazione.

Quando il 6510 incontra questa istruzione non viene eseguita nessuna operazione per due cicli

macchina.

Programma 4.3

1				*	=	828
2	033C	18			CLC	
3	033D	D8			CLD	
4	033E	A9	85		LDA	#133
5	0340	69	85		ADC	#133
6	0342	8D	02	04	STA	1026
7	0345	A2	01		LDX	#1
8	0347	8E	02	D8	STX	55298
9	034A	EA			NOP	
10	034B	A9	04		LDA	#4
11	034D	69	04		ADC	#4
12	034F	8D	00	04	STA	1024
13	0352	8E	00	D8	STX	55296
14	0355	60			RTS	

Dopo il RUN dovrebbero apparire le lettere I e J sullo schermo.

I passi di questo programma sono specificati e dimostrati nella tabella 4.1.

PASSO	ACCUM.	X	1026	1024	C
CLC	?	?	0	0	0
CLD	?	?	0	0	0
LDA& 133	133	?	0	0	0
ADC& 133	10	?	0	0	1
STA 1026	10	?	10	0	1
LDX& 1	10	1	10	0	1
STX 55298	10	1	10	0	1
NOP	10	1	10	0	1
LDA& 4	4	1	10	0	1
ADC& 4	9	1	10	0	0
STA 1024	9	1	10	9	0
STX 55296	9	1	10	9	0
RTS	9	1	10	9	0

Come mostra la tavola, all'istruzione ADC& 133, viene generato un riporto e il Flag C e' messo a 1 che ha effetto sul seguente ADC.

Altra cosa da notare e' che all'istruzione ADC& 4 non c' e' invece nessun riporto e percio' il Carry e' posto a 0.

Per controllare cio' potreste rimpiazzare il comando NOP con CLC che dovrebbe PULIRE il Flag prima che sia fissato e notare che la risposta data dovrebbe essere errata.

Cio' puo' essere fatto attraverso un comando POKE che immettera' nella locazione 842 il codice per CLC (24)

Programma 4.3a

POKE 842,24

Facendo ora girare il programma 4.3 modificato con 4.4 saranno visualizzate le lettere:

```
..... H J
```

A questo giro, il valore di J e' stato calcolato e quando il suo valore 266 passato, allora e' riportato il 256, il bit di Carry fissato e il valore 10 immagazzinato nell' Accumulatore.

INPUT IN ESADECIMALE

Il sistema esposto in precedenza per la conversione da decimale ad esadecimale puo' sembrare a qualcuno un po' empirico anche se e' sostanzialmente corretto.

Nel programma che mostriamo, invece di inserire numeri decimali immetteremo dei valori espressi in questa nuova notazione e che saranno preceduti dal simbolo del dollaro (\$).

Utilizzando come base il programma 4.3 otterremo questo nuovo listato:

Programma 4.3b

1				*	=	828
2	033C	18			CLC	
3	033D	D8			CLD	
4	033E	A9	85		LDA	#\$85
5	0340	69	85		ADC	#\$85
6	0342	8D	02	04	STA	1026
7	0345	A2	01		LDX	#1
8	0347	8E	02	D8	STX	\$D802
9	034A	EA			NOP	
10	034B	A9	04		LDA	#\$04
11	034D	69	04		ADC	#\$04
12	034F	8D	00	04	STA	\$0400
13	0352	8E	00	D8	STX	\$D800
14	0355	60			RTS	

Quando questo programma gira, da' lo stesso risultato del programma 4.3 visto in precedenza.

Esercizio 4.1

Usando come input valori esa, sommare \$1807 e \$2AFA. Verificare il programma con addendi e risultati in base 10.

Il 6510 possiede un'istruzione che consente la sottrazione con il Carry. Questa istruzione e':

SBC SuBtract from the accumulator with Carry the data at the specified memory location.

Cioe' sottrai dall' Accumulatore, con riporto il dato contenuto in uno specifico indirizzo di

memoria.

Per esempio:

..... SBC 891

e' un' istruzione che andra' a vedere il valore presente nella locazione di memoria 891 e sottrara' il numero ivi trovato dal valore contenuto nell' accumulatore.

Tuttavia, allo stesso modo in cui si rendeva necessario preparare il Flag di Carry per l'addizione mettendolo a 0, si rende necessario prepararlo per la sottrazione.

Tuttavia in questo caso sara' necessario invertire il valore del Carry mettendolo a 1 anziche' a 0.

La relativa istruzione e':

SEC SEt the Carry bit to 1

Cioe' metti il bit di Carry a 1

Vediamone ora un' applicazione in un programma che pero' non fara' uso dell' istruzione SBC ma carichera' i valori in modo diretto eseguendo $4 - 2$.

Programma 4.4

1				*	=	828
2	033C	38			SEC	
3	033D	A9	04		LDA	#4
4	033F	E9	02		SBC	#2
5	0341	8D	00	04	STA	1024
6	0344	A9	01		LDA	#1
7	0346	8D	00	08	STA	55296
8	0349	60			RTS	

Proponiamo un esercizio.

Esercizio 4.2

Scrivere un programma che sottragga 600 da 800 usando l'indirizzamento assoluto. Immagazzinare i dati a partire dalla locazione 890. Visualizzare il risultato in 1034.

Esercizio 4.3

Scrivere un programma che sottragga 500 dalla somma eseguita in programma stesso di 300 + 400 (tutti i valori in decimale). Visualizzare la risposta in 1040/1 con l'ordine LSB/MSB

LA MOLTIPLICAZIONE

Le istruzioni aritmetiche disponibili sul 6510

consentono addizioni e sottrazioni ma non, almeno direttamente, moltiplicazioni.

Questo, cioè la risoluzione di una moltiplicazione, viene fatta attraverso una serie di somme.

Per esempio 2×3 può essere espresso come $2+2+2$ e ciò è relativamente semplice da programmare.

Il processo da eseguire è quello di aggiungere all' Accumulatore il valore 2 tre volte e questo richiede che 3 sia fissato in un ciclo che definisce il numero di volte che sarà quindi necessario eseguire la somma.

Ricordiamo che l' Accumulatore all' inizio deve contenere un valore 0.

Vediamo un' applicazione.

Programma 4.5

```
SOMMA2 = $0341
1          *          =      828
2  033C 18          CLC
3  033D A0 03      LDY  #3
4  033F A9 00      LDA  #0
5  0341 69 02      SOMMA2 ADC #2
6  0343 88          DEY
7  0344 D0 FB      BNE  SOMMA2
8  0346 8D 00 04   STA  1024
9  0349 A9 01      LDA  #1
10 034B 8D 00 D8   STA  55296
11 034E 60          RTS
```

Eseguendo il RUN sarà visualizzata una F (che è il valore relativo di 6) in 1024.

All' interno di questo programma la chiave e':

```
ADC& 2  
DEY  
BNE 251
```

Questo ciclo che esegue il lavoro e' conosciuto in generale come ALGORITMO.

Naturalmente una limitazione di questo semplice algoritmo e' che puo' solo manipolare una risposta con valore non superiore a 255, dopo di che genera un Carry e il valore dell' accumulatore torna a 0.

Si rende necessario anche in questo caso passare al concetto di moltiplicazione in doppia precisione.

Cio' puo' essere ottenuto controllando il Carry dopo ogni somma e se e' stato generato un riporto, aggiungere 1 entro MSB.

Un sistema di controllare l' incremento generato dal ciclo e' di operare con la seguente istruzione:

```
INC INCRement the contents of the specified  
memory location
```

Cioe' incrementa il contenuto di una specifica locazione di memoria.

Il programma 4.6 mostra l' algoritmo precedentemente visto elaborato per registrare il

numero di riporti generati e per incrementare MSB
affinche' registri tutto questo.

Programma 4.6

```

SOMM16 = $0345
DECY = $034D
1
2 033C A0 00 * = 828
3 033E 8C 89 03 LDY #0
4 0341 A0 11 STY 905
5 0343 A9 00 LDY #17
6 0345 18 LDA #0
7 0346 69 10 SOMM16 CLC
8 0348 90 03 ADC #16
9 034A EE 89 03 BCC DECY
10 034D 88 INC 905
11 034E D0 F5 DECY DEY
12 0350 8D 02 04 BNE SOMM16
13 0353 A2 01 STA 1026
14 0355 8E 02 D8 LDX #1
15 0358 AD 89 03 STX 55298
16 035B 8D 00 04 LDA 905
17 035E 8E 00 D8 STA 1024
18 0361 60 STX 55296
RTS

```

Quando gira dovrebbe essere visualizzato A P o
256+16, esempio 16x17.

Nelle pagine successive vedremo un altro metodo
per le moltiplicazioni.

LA

DIVISIONE

Nello stesso modo che la moltiplicazione e' fatta per somme successive, cosi' la divisione deve essere eseguita per sottrazioni successive.

Questo concetto viene illustrato nel programma 4.6a nel quale 30 e' diviso 2.

In questo caso l' Accumulatore e' usato per immagazzinare cio' che resta da elaborare, per esempio partendo da 30 ed andando progressivamente verso 0 (30, 28, 26, 24, 22, ..4, 2, 0).

Il registro X e' usato per caricare il divisore in memoria mentre il registro Y memorizza il numero di volte che la sottrazione deve essere eseguita.

Programma 4.6a

```

SOTT2 =    $0345
1          *          =    828
2    033C  A0  00          LDY    #0
3    033E  A2  02          LDX    #2
4    0340  8E  84  03          STX    900
5    0343  A9  1E          LDA    #30
6    0345  38          SOTT2    SEC
7    0346  E9  02          SBC    #2
8    0348  C8          INY
9    0349  CD  84  03          CMP    900
10   034C  B0  F7          BCS    SOTT2
11   034E  8C  00  04          STY    1024
12   0351  A2  01          LDX    #1
13   0353  8E  00  D8          STX    55296
14   0356  8D  02  04          STA    1026
15   0359  8E  02  D8          STX    55298
16   035C  60          RTS

```


Dopo il RUN sara' visualizzato il quoziente 15 (come lettera 0) in 1024 ed il resto 0 (come a commerciale) in 1026.

CODICE DECIMALE BINARIO

In aggiunta ai numeri che possono essere rappresentati con la notazione binaria e con quella decimale esiste una forma ibrida o mista appunto la BCD o CODICE DECIMALE BINARIO.

Il BCD forma un ponte fra le due notazioni ed in molti casi facilita grandemente gli output.

Per fortuna il microprocessore puo' manipolare direttamente BCD ed e' messo in condizioni di operare in questo modo con l'istruzione:

SED SEt Decimal mode.

Questa istruzione fissa automaticamente il Flag D a 1 e percio' le operazioni sono date in BCD.

Quando questo modo di operare non e' piu' necessario allora il flag D e' rimesso a 0 con l'istruzione:

CLD CLear Decimal flag

Questa istruzione, riportando a 0 il flag D, consente di tornare ad operare in binario.

Un semplice programma per sommare 1 a 2 usando il modo BCD e' dato nel programma 4.7.

Quando gira, questo programma immette una C in

1024.

E' considerato normalmente una buona pratica eseguire un clear sul Flag D dopo ogni operazione di BCD.

L' esempio dato nel 4.7 e' in effetti identico ad una normale operazione aritmetica con l' eccezione che in BCD il riporto avviene dopo che ogni mezzo byte (NIBBLE) supera il 9.

Questo e' dimostrato nel programma 4.8 che aggiunge ancora due 6.

NOTA

Se il programma 4.7 fosse ancora in 828 allora il 4.8 puo' essere POKEGGIATO tramite:

```
..... POKE 831, 6
..... POKE 836, 6
```

Programma 4.7

1				*	=	828
2	033C	F8			SED	
3	033D	18			CLC	
4	033E	A9	02		LDA	#2
5	0340	8D	84	03	STA	900
6	0343	A9	01		LDA	#1
7	0345	6D	84	03	ADC	900
8	0348	8D	00	04	STA	1024
9	034B	A2	01		LDX	#1
10	034D	8E	00	D8	STX	55296
11	0350	D8			CLD	
12	0351	60			RTS	

Programma 4.8

1				*	=	828
2	033C	F8			SED	
3	033D	18			CLC	
4	033E	A9	06		LDA	#6
5	0340	8D	84	03	STA	900
6	0343	A9	06		LDA	#6
7	0345	6D	84	03	ADC	900
8	0348	8D	00	04	STA	1024
9	034B	A2	01		LDX	#1
10	034D	8E	00	08	STX	55296
11	0350	DB			CLD	
12	0351	60			RTS	

Quando gira il programma 4.8 immette una R bianca in 1024 (equivalente al 12).

Cio' deriva dal fatto che il BCD e' immagazzinato in memoria come NIBBLE, cioe' mezzo Byte.

La lettera R pero' viene come codice del CBM 64 con valore di 18. Come puo' succedere questo?

18 in binario e':

00010010

Tuttavia l' indirizzo di memoria e' immagazzinato in due NIBBLES:

00010010 e' in realta':

0001 cioè 1 (in decimale)

e

0010 cioè 2 (in decimale)

per cui il numero rappresentato dai due NIBBLES è:

$1 \times 10 + 2 \times 1 = 12$ (decimale)

L' esempio precedente enfatizza il problema che si presenta quando si pensa in modo decimale e si sta lavorando in binario.

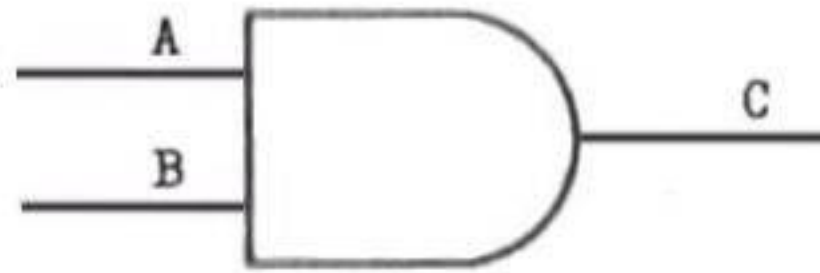
In rapporto a quanto abbiamo visto circa il programma precedente dobbiamo trovare una tecnica di manipolazione dei singoli bits entro il Byte. Per estrarre il Nibble basso da un numero binario è sufficiente cancellare il nibble alto, ad esempio immettendo tutti 0.

Questo può essere fatto con l' istruzione:

AND Esegue un AND logico entro l' Accumulatore.

Un AND è un operatore logico che confronta due stati logici e produce in uscita un risultato sulla base del confronto.

Se esaminiamo una porta logica AND come è usata in un circuito elettronico, si comprende bene anche la funzione assembler AND.



La figura mostra una porta AND con due ingressi A e B con un' uscita C.

La funzione di questo circuito consiste nel fatto che se entrambi gli ingressi sono a 1 allora anche C sarà a 1.

Se invece A o B o tutti e due sono a 0 allora C sarà 0.

Cio' è normalmente espresso in quella che è conosciuta come TAVOLA DELLA VERITA' (TRUTH TABLE) che mostriamo di seguito:

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

TAVOLA DELLA VERITA' PER AND

Fig 4.5

Esercizio 4.4

Usando la tavola della verita', calcolare l' output logico ottenuto con i seguenti input:

..... A = 1 AND B = 0

..... A = 0 AND B = 1

..... A = 1 AND B = 1

Quando viene eseguito un AND dal 6510, esso opera su tutti gli 8 bits dell' accumulatore contemporaneamente.

Per cui se su 255 viene eseguito un AND di 1 avremo:

..... 255 = 11111111

..... 1 = 00000001

cioe':

..... 1 1 1 1 1 1 1 1 Accumulatore

..... 0 0 0 0 0 0 0 1 AND 1

..... 0 0 0 0 0 0 0 1 Risultato

Il risultato crediamo che non abbia bisogno di commenti.

Esercizio 4.5

Quale risultato si ottiene eseguendo un AND fra i seguenti due numeri (base dieci) 149 e 52.

Come abbiamo appena visto nell' esercizio precedente l' istruzione AND puo' essere usata per togliere bits da un numero e potrebbe essere usata per convertire parte del BCD 12 dal programma 4.8.

Questa istruzione BCD 12 era stata immagazzinata come due NIBBLES in un Byte.

Se il Nibble piu' significativo o MSN puo' essere cambiato in 4 zeri allora il Byte potrebbe essere letto direttamente come Nibble Meno significativo o LSN.

In questo modo il mascheramento di bits puo' essere fatto usando un comando AND.

Vediamone il comportamento con BCD 12:

```
..... 0 0 0 1 0 0 1 0 ..... BCD 12
AND     0 0 0 0 1 1 1 1 ..... Binary 15
=       0 0 0 0 0 0 1 0 ..... " 2
```

Eseguendo cioe' l' AND fra BCD ed il numero decimale 15 (cioe' in binario 00001111) i quattro bits piu' significativi sono stati cancellati ed il numero convertito in LSN (in questo caso 2 decimale).

In un programma l'istruzione AND puo' essere usata con diversi modi d'indirizzamento. Vediamo un esempio con il modo immediato:

Programma 4.9

1				*	=	828
2	033C	A2	0F		LDX	#15
3	033E	8E	84	03	STX	900
4	0341	A9	12		LDA	#18
5	0343	2D	84	03	AND	900
6	0346	8D	00	04	STA	1024
7	0349	A2	01		LDX	#01
8	034B	8E	00	D8	STX	55296
9	034E	60			RTS	

Quando gira questo programma, sara' visualizzata una B bianca in 1024.

Usando la notazione binaria, il programma 4.9 puo' essere riscritto come segue:

Programma 4.9a

1				*	=	828
2	033C	A2	0F		LDX	##%00001111
3	033E	8E	84	03	STX	900
4	0341	A9	12		LDA	##%00010010
5	0343	2D	84	03	AND	900
6	0346	8D	00	04	STA	1024
7	0349	A2	01		LDX	#1
8	034B	8E	00	D8	STX	55296
9	034E	60			RTS	

ORA E EOR

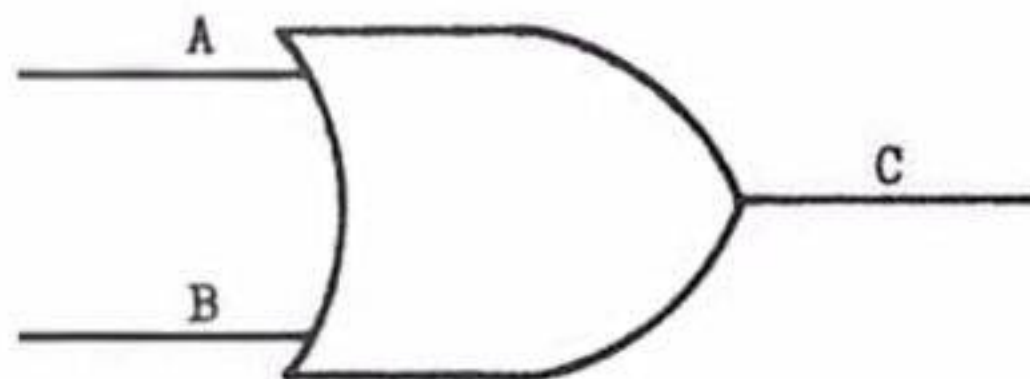
Il 6510 usa anche due altri operatori logici uno dei quali consente la funzione OR.

Il codice mnemonico usato in questo manuale per questa funzione e':

ORA Perform a logical inclusive OR between the Accumulator and the data specified.

Cioe' esegui un OR fra l' Accumulatore ed il dato specificato.

In un circuito elettrico la funzione OR viene simboleggiata come segue



Il suo modo di operare e' che se un "1" e' presente in A o (OR) in B allora l' uscita C e' messa a "1".

E' un po' il rovescio di AND il quale da' come risultato "1" solo se entrambi gli ingressi sono a "1", mentre OR da' "0" solo se entrambi gli ingressi sono a "0".

Di seguito la tavola della verita' o TRUTH TABLE.

	0	1
0	0	1
1	1	1

Nel programma il comando ORA ha il seguente effetto:

Binario di 149 1 0 0 1 0 1 0 1
 Binario di 52 0 0 1 1 0 1 0 0

..... ORA
 Binario di 181 1 0 1 1 0 1 0 1

Inserendo in un programma:

Programma 4.10

1		*	=	828
2	033C	A9 95	LDA	#149
3	033E	09 34	ORA	#52
4	0340	8D 00 04	STA	1024
5	0343	A2 01	LDX	#1
6	0345	8E 00 D8	STX	55296
7	0348	60	RTS	

Allo stesso modo di AND l'istruzione OR ha la possibilita' di numerosi modi d'indirizzamento.

Il terzo operatore logico e':

EOR Perform a logical Exclusive OR between the accumulator and the data specified.

Questa operazione e' probabilmente la piu' facile da comprendere ed e' illustrata dalla seguente tavola della verita':

	0	1
0	0	1
1	1	0

Un sistema di esprimere la funzione e' che l'uscita sara' "1" se l'uno o l'altro degli inputs e' "1" ma non entrambi.

Vediamo di usare questa istruzione con un esempio eseguendo un EOR di 149 con 52 (decimali):

```

149 in Binario 1 0 0 1 0 1 0 1
52      "      0 0 1 1 0 1 0 0
161     "      1 0 1 0 0 0 0 1

```

EOR

Il programma per provare cio' e' il seguente:

Programma 4.11

```

1          *          =      828
2  033C A9 95          LDA   #%10010101
3  033E 49 34          EOR   #%00110100
4  0340 8D 00 04      STA   1024
5  0343 A2 01          LDX   #1
6  0345 8E 00 D8      STX   55296
7  0348 60           RTS

```

Anche questo operatore logico ha diversi modi di indirizzamento per facilitare l' uso in programmi.

Esercizio 4.6

Calcolare i risultati delle seguenti operazioni logiche:

i) 100 AND 87

ii) 75 OR 27

iii) 99 EOR 57

iv) 94 EOR con il risultato di 100 AND 87.

Tutti i valori sono in decimale.

Scrivere un programma che verifichi ogni operazione.

ALTRE FORME DI MANIPOLAZIONE DEI BIT

Esistono altre istruzioni 6510 che consentono di manipolare bits entro un Byte.

Nell' ultimo esempio che usava BCD, l' istruzione AND era in grado di isolare LSN dal Byte.

Tuttavia non era possibile estrarre il MSN usando la logica disponibile.

Usando uno dei comandi di manipolazione bit che mostreremo cio' diventa ora possibile.

LSR Logical Shift of the specified contents one bit to the Right.

Cioe' esegui uno SHIFT, uno spostamento, di un bit verso destra.

Quando viene eseguito questo comando i bits sono spostati, TUTTI, di un posto verso destra. L'ultimo bit a destra viene immesso nel Carry e la prima posizione del Byte, che resterebbe vuota, viene riempita con uno 0.

Eseguendo una istruzione LSR sul numero 149 (decimale) avremo il seguente risultato:

149 1 0 0 1 0 1 0 1

 = 0 1 0 0 1 0 1 0

 con 1 nel Carry

Come per altri comandi anche LSR ha numerosi modi di indirizzamento e l'indirizzo particolare informa il 6510 dove si trova il dato che deve essere traslato. Così':

LSR A consente una traslazione a destra dei dati in Accumulatore.

LSR 900 come sopra dei dati di indirizzo di memoria 900.

Usiamo il modo Accumulatore di LSR per immettere e far girare il seguente programma:

Programma 4.12

1				*	=	828
2	033C	A9	95		LDA	#149
3	033E	4A			LSR	A
4	033F	8D	00 04		STA	1024
5	0342	A2	01		LDX	#1
6	0344	8E	00 D8		STX	55296
7	0347	60			RTS	

Che visualizza l' equivalente decimale di 74 in 1024.

Usando quattro volte lo spostamento il MSN viene messo al posto del LSN ed i 4 bits a sinistra riempiti con "0".

Cio' consente di isolare MSN in un calcolo BCD.

Tutto questo viene dimostrato nel programma seguente che usa il LSR in modo assoluto.

Programma 4.13

LOOP =	\$0343					
1				*	=	828
2	033C	A0	12		LDY	#18
3	033E	8C	84 03		STY	900
4	0341	A0	04		LDY	#4
5	0343	4E	84 03	LOOP	LSR	900
6	0346	88			DEY	
7	0347	D0	FA		BNE	LOOP
8	0349	AD	84 03		LDA	900
9	034C	8D	00 04		STA	1024
10	034F	A2	01		LDX	#1
11	0351	8E	00 D8		STX	55296
12	0354	60			RTS	

Esercizio 4.7

Supponiamo che la risposta ad un problema BCD sia 86.

Scriviamo allora un programma in codice macchina per decodificare questo e visualizziamo la risposta in decimale in 1024 e 1025.

Un'altra istruzione del set del 6510 è di muoversi di un bit a sinistra:

ASL Arithmetic Shift Left

Cioè sposta verso sinistra di un bit.

Anche in questo caso i bits del Byte considerato vengono spostati verso sinistra di una posizione. La posizione più a sinistra che rimarrebbe vuota viene riempita con uno "0". Il bit più a sinistra è immagazzinato nel Carry.

Eseguendo una istruzione ASL sul numero 149 (decimale) avremo il seguente risultato:

149 1 0 0 1 0 1 0 1

0 0 1 0 1 0 1 0 =

con 1 nel Carry

Come per altri comandi anche ASL ha numerosi modi

di indirizzamento e l' indirizzo particolare informa il 6510 dove e' il dato che deve essere traslato. Così':

ASL A consente una traslazione a sinistra dei dati in Accumulatore.

LSR 900 come sopra dei dati di indirizzo di memoria 900.

Usiamo il modo accumulatore per provare il seguente esempio:

Programma 4.14

1				*	=	828
2	033C	A9	95		LDA	#149
3	033E	A0			ASL	A
4	033F	8D	00	04	STA	1024
5	0342	A2	01		LDX	#1
6	0344	8E	00	D8	STX	55296
7	0347	60			RTS	

MOLTIPLICAZIONE BINARIA

Abbiamo visto, relativamente ai programmi 4.5 e 4.6 che si puo' eseguire una moltiplicazione usando un processo ripetitivo o RE-ITERATIVO, ma abbiamo anche visto che si tratta di un procedimento lungo e dispendioso.

Esistono pero', in particolare con le istruzioni viste e quelle che vedremo, altri e piu' veloci metodi. Vediamoli.

Vediamo prima di tutto il sistema convenzionale di operare.

Si abbia da moltiplicare 13 x 14. Normalmente si definisce 13 come MOLTIPLICANDO e 14 come MOLTIPLICATORE:

```

..... 13 x
..... 14
..... --
..... 52
..... 130
..... ---
..... 182
.....
```

In questo formato convenzionale, abbiamo per prima cosa eseguito la moltiplicazione del MOLTIPLICANDO per il digit piu' basso del moltiplicatore ed immagazzina questo come primo prodotto parziale $4 \times 13 = 52$.

Successivamente si moltiplica il moltiplicando per il secondo digit del moltiplicatore, 1×13 , e dopo si moltiplica questo per 10 per ottenere un secondo prodotto parziale, $13 \times 10 = 130$.

In questo modo la somma totale e' la somma delle parti, $52 + 130 = 182$.

E' possibile in modo semplice di attuare la stessa moltiplicazione usando numeri in formato

binario.

Per esempio moltiplicando 5 x 7 in binario:

5 = 0101 7 = 0111

per cui:

7 x 5 equivale a 0111 x 0101

Prodotto parziale 1	0111	=	0111
" "	00000	=	0111
" "	011100	=	100011
" "	0000000	=	100011

Risposta 100011

Cioe' 32 + 2 + 1 = 35

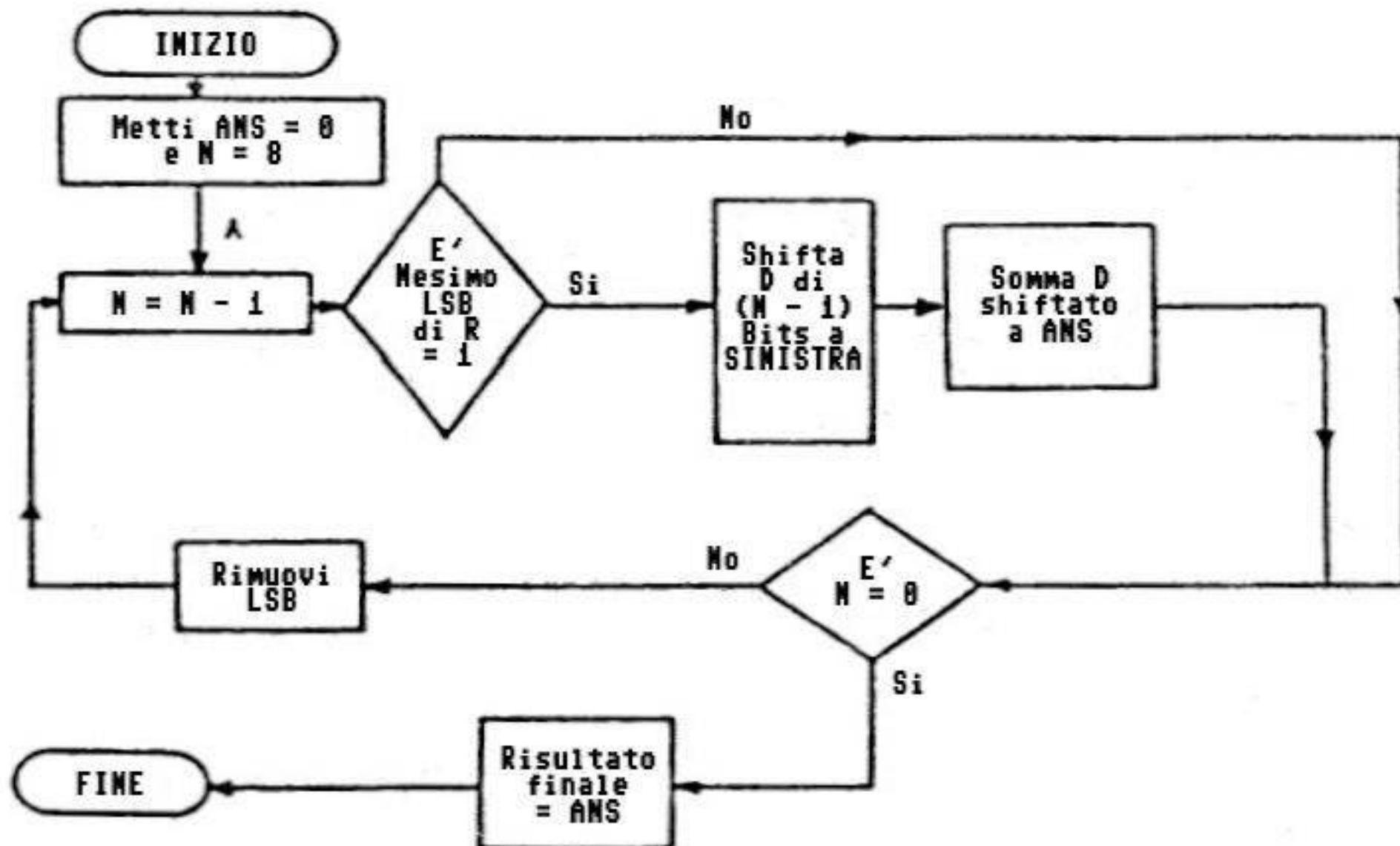
In questo caso il processo di moltiplicazione in binario si riduce ad una successiva addizione che segue il movimento a sinistra del moltiplicando.

MOLTIPLICAZIONE AD 8 BIT

Il diagramma a blocchi relativo a questo processo e' dato nella seguente figura dove:

ANS= risposta
D = moltiplicando
R = moltiplicatore

N = numero corrente del bit
 LSB= ultimo bit significativo del moltiplicatore;



Con questo programma operiamo una semplice moltiplicazione 2*2.

Programma 4.15

```

SALT01 = $0348
SALT02 = $0350
1          *          =      828
2  033C A2 02          LDX  #2
3  033E A0 08          LDY  #8
4  0340 8E 85 03      STX  901
5  0343 8E 86 03      STX  902
6  0346 A9 00          LDA  #0
7  0348 18            SALT01 CLC
8  0349 4E 86 03      LSR  902
9  034C 90 04          BCC  SALT02
  
```


10	034E	18			CLC	
11	034F	6D	85	03	ADC	902
12	0352	0E	85	03	SALTO2	ASL 901
13	0355	88			DEY	
14	0356	D0	F0		BNE	SALTO1
15	0358	8D	00	04	STA	1024
16	035B	A2	01		LDX	#1
17	035D	8E	00	D8	STX	55296
18	0360	60			RTS	

Esercizio 4.8

Riscrivere il precedente programma in maniera che moltiplichi due valori diversi.

Sfortunatamente il programma 4.15 e' solo una mezza verita' come routine di moltiplicazione ad 8 bit ed opera solo con numeri piccoli sia come moltiplicando che come moltiplicatore.

In una routine completa l'istruzione ASL moltiplica il moltiplicando otto volte per la base. Così con l'ottavo spostamento il bit piu' a destra dovrebbe trovarsi al margine sinistro del registro.

Infatti il secondo bit dovrebbe essere perso dopo il settimo spostamento.

Tuttavia cio' non ha effetto sul risultato complessivo perche' dopo due istruzioni di LSR di 00000010 tutti gli "1" sono stati cancellati e di conseguenza le susseguenti somme parziali saranno uguali a 0.

Se desideriamo usare il programma con numeri maggiori nella cui risposta finale e' presente

un riporto (CARRY), allora le risposte non sarebbero attendibili perche' l' ultimo bit a sinistra era significativo.

Fortunatamente l' ultimo bit a sinistra non viene perso nello spazio durante una operazione ASL ma viene inserito nel Carry.

Il problema allora e' di rintracciarlo e di riportarlo nell' MSB della risposta. Cio' puo' essere fatto usando il seguente nuovo comando:

ROL ROTate Left the contents of a specified address.

In questa operazione tutti i bits di uno specifico indirizzo eseguono una rotazione a sinistra ed il bit di Carry viene caricato nella cella del bit piu' a destra mentre il bit piu' a sinistra viene trasferito nel Carry.

7 6 5 4 3 2 1 0 schema del Byte

dopo l' esecuzione di ROL

6 5 4 3 2 1 0 C

nella posizione tenuta precedentemente dallo 0 c'e' ora il contenuto del Carry mentre nel Carry c'e' il contenuto dell' ottavo Bit (cioe' il bit n. 7).

Dato il numero effettivo di bits interessati questa operazione e' chiamata anche ROTAZIONE A 9

BITS.

Tuttavia il programma che ne deriva, in particolare per le moltiplicazioni ad 8 bits e' molto piu' complesso e vedremo come si usa con le labels nel capitolo seguente.

L' istruzione appena vista ne porta di conseguenza logica un' altra per la rotazione a destra:

ROR ROTate Right the contents of the specified address.

Cioe' esegui la rotazione a destra di un dato contenuto in uno specifico indirizzo.

Entrambe le istruzioni viste possono essere utilizzate in forme diverse come:

ROL A ROTate Left the contents of the Accumulator.

ROR A ROTate Right the contents of the Accumulator.

Che si spiegano da sole.

E' disponibile un' altra istruzione per la

manipolazione dei Bits:

BIT AND specified content's BIT's with accumulator.

Cioe' esegui un AND logico fra il contenuto di un Byte di locazione di memoria data con l' Accumulatore.

Ad esempio l' istruzione BIT 900 esegue un AND logico fra il contenuto dell' Accumulatore e il contenuto della locazione di memoria 900.

Mentre BIT consente la stessa funzione di AND, ne differisce in quanto lascia sia l'Accumulatore che la memoria come sono. Sono pero' modificati numerosi FLAGS nel PWS. Vediamo cosa accade:

1) Il flag Z viene messo a 1 se il risultato dell' AND e' zero e viceversa messo a 0 se il risultato e' diverso da zero.

2) Per il flag N invece e': il bit 7 della locazione che deve essere controllata e' copiato nel Processor Status register.

Questo e' un sistema molto conveniente di controllare quando il contenuto di una particolare locazione sia positiva o negativa senza la necessita' di caricarne il valore entro uno dei due registri.

3) Il Flag V (che non abbiamo ancora visto in dettaglio) e' il bit 6 del PSR. L' istruzione BIT

copia il bit 6 della locazione che deve essere controllata nel bit 6 del PSR. Cio' non e' utile come il Flag N visto prima in quanto il bit 6 normalmente non e' molto importante. Vedremo tuttavia che il Basic lo adopera molto spesso.

Usando queste istruzioni in binario, puo' essere messo in funzione un procedimento analogo a quello visto per la moltiplicazione vista in precedenza.

DIVISIONE BINARIA A 8 BIT.

Questo procedimento e' analogo a quello visto nella routine di moltiplicazione binaria dato che necessita solo di 8 RE-ITERAZIONI per manipolare un numero di 8 bits.

E' illustrato nel programma seguente dove il dividendo (nel caso 31) e' immagazzinato nella locazione 900 ed il divisore (2) in 901.

Il registro Y e' usato come contatore di ciclo per assicurarsi che l' algoritmo relativo venga eseguito 8 volte.

Tramite le istruzioni ASL e ROLA il RESTO della divisione e' inserito nell' accumulatore.

Programma 4.15a

SALT01 = \$034A

SALT02 = \$0359

1				*		=	828
2	033C	A2	1F			LDX	#31

3	033E	8E	84	03		STX	900
4	0341	A2	02			LDX	#2
5	0343	8E	85	03		STX	901
6	0346	A0	08			LDY	#8
7	0348	A9	00			LDA	#0
8	034A	0E	84	03	SALTO1	ASL	900
9	034D	2A				ROL	A
10	034E	CD	85	03		CMP	901
11	0351	90	06			BCC	SALTO2
12	0353	ED	85	03		SBC	901
13	0356	EE	84	03		INC	900
14	0359	88			SALTO2	DEY	
15	035A	D0	EE			BNE	SALTO1
16	035C	AE	84	03		LDX	900
17	035F	8E	00	04		STX	1024
18	0362	A0	01			LDY	#1
19	0364	8C	00	D8		STY	55296
20	0367	8D	02	04		STA	1026
21	036A	8C	02	D8		STY	55298
22	036D	60				RTS	

Quando questo programma gira verra' visualizzato il quoziente 15 (come una 0) in 1024 ed il resto 1 (come una A) in 1026.

CAPITOLO QUINTO

LE LABELS

L' uso delle labels consente al programma di dirigersi verso istruzioni con nome senza la necessita' di calcolare salti e relativi indirizzi.

Un termine usuale per Labels e' LABEL SIMBOLICHE perche' le labels stesse sono simboli di locazioni di memoria. Per esempio l' istruzione:

```
..... BNE LOOP1
```

comunica all' Assembler di costruire un codice macchina che comunichi al 6510 di saltare ad un' istruzione chiamata (o con label) LOOP1.

LOOP1 STA 1024,X crea un label chiamata LOOP1 il cui indirizzo e' lo stesso di "STA" nell' istruzione STA 1024,X.

Come abbiamo visto la LABEL deve essere messa nell' apposita colonna.

Per questo comunque l' inizio di LOOP1 dovrebbe essere immessa come:

```
..... LOOP1 STA 1024,X
```

La lunghezza della LABEL non puo' essere superiore ai 6 caratteri e NON deve contenere spazi.

Anche in questo caso non esiste ragione

particolare tranne che questo Assembler quando trova uno spazio dopo la label la considera conclusa.

Per questa stessa ragione la Label deve essere seguita da uno spazio e poi da una normale istruzione.

Quando ci si riferisce ad una Label in un'istruzione e' necessario solo di rimpiazzare l'operando dell'istruzione con la label stessa.

Per semplificare le cose passiamo a vedere come al solito un esempio applicativo del concetto appena esposto.

Il programma seguente usa due cicli (loops) chiamati LOOP1 e LOOP2 ed esegue alcuni salti non necessari a scopo dimostrativo.

Programma 5.2

```

LOOP1 =    $0343
LOOP2 =    $034E
FINE =    $0360
1          *          =    828
2    033C  A2  A0          LD% #160
3    033E  A0  01          LDY #1
4    0340  4C  4E  03      JMP  LOOP2
5    0343  A9  53          LOOP1 LDA #83
6    0345  9D  9F  04      STA  1183,X
7    0348  CA              DEX
8    0349  D0  F8          BNE  LOOP1
9    034B  4C  60  03      JMP  FINE
10   034E  A9  5A          LOOP2 LDA #90
11   0350  9D  FF  03      STA  1023,X
12   0353  98              TYA
13   0354  9D  FF  D7      STA  55296,X

```



```

14 0357 CA          DEX
15 0358 D0 F4      BNE LOOP2
16 035A A2 78      LDX #120
17 035C 88        DEY
18 035D 4C 43 03   JMP LOOP1
19 0360 60          FINE  RTS

```

Sebbene questo programma esegua dei salti a determinati punti, e' ancora relativamente facile da seguire.

Quando il processo di assemblaggio e' terminato il programma risiederà in memoria nello stesso identico formato di un qualsiasi altro programma.

Esercizio 5.1

Aggiungere un terzo ciclo LOOP3 al programma precedente. Riscrivere il programma facendo girare per primo il LOOP3 seguito dal LOOP1 e LOOP2.

Il LOOP3 dovrebbe far apparire sullo schermo 2 righe di asterischi rossi.

MEMORY LABELS

In aggiunta alle istruzioni LABEL, l'assembler consente anche la creazione di LABEL come locazioni di memoria.

Queste verranno impostate come la locazione di

inizio programma.

Es.

1 * = 828
2 DATO = 900

Con questa istruzione (DATO=900) l' Assembler quando durante la compilazione del programma verra' incontrata la LABEL DATO si riferira' alla locazione 900.

Il seguente programma illustra l' uso di labels di memoria in una somma in doppia precisione che addiziona due numeri a 16 bits.

..... Numero 1 = 2760

..... " 2 = 948

immessi in LSB1 e MSB1 e LSB2 e MSB2.

La risposta sara' immessa in ANSLSB1 e ANSMSB2

Programma 5.3

LSB1 = \$0384
 MSB1 = \$0385
 LSB2 = \$0386
 MSB2 = \$0387
 ANSLSB = \$0388
 ANSMSB = \$0389

1				*	=	828
2				LSB1	=	900
3				MSB1	=	901
4				LSB2	=	902
5				MSB2	=	903
6				ANSLSB	=	904
7				ANSMSB	=	905
8	033C	A9	0A		LDA	#10
9	033E	8D	85	03	STA	MSB1
10	0341	A9	08		LDA	#200
11	0343	8D	84	03	STA	LSB1
12	0346	A9	03		LDA	#3
13	0348	8D	87	03	STA	MSB2
14	034B	A9	B4		LDA	#180
15	034D	8D	86	03	STA	LSB2
16	0350	18			CLC	
17	0351	6D	84	03	ADC	LSB1
18	0354	8D	88	03	STA	ANSLSB
19	0357	8D	01	04	STA	1025
20	035A	AD	85	03	LDA	MSB1
21	035D	6D	87	03	ADC	MSB2
22	0360	8D	89	03	STA	ANSMSB
23	0363	8D	00	04	STA	1024
24	0366	A2	02		LDX	#2
25	0368	8E	00	D8	STX	55296
26	036B	8E	01	D8	STX	55297
27	036E	60			RTS	

ALTRE FUNZIONI

Vediamo una per una queste nuove opzioni rimandando al prossimo capitolo quella relativa al monitor.

INSERIMENTO LINEE

Selezionando la relativa richiesta si ha la possibilita' di inserire una nuova sezione di codici macchina entro un programma gia' esistente. Questa opzione e' particolarmente importante in quanto consente di correggere programmi gia' scritti o di effettuare aggiunte o variazioni.

E' possibile aprire un nuovo spazio nel quale inseriremo altre istruzioni usando poi per questo l' opzione INSERIMENTO LINEE del menu' principale.

Viene richiesta la linea da dove vorremmo iniziare l' inserimento ed il numero di linee da inserire. Quindi verranno spostate in avanti le linee di programma in modo che venga riservato uno spazio all' interno del programma precedentemente scritto.

CANCELLAZIONE LINEE

Supponendo di avere un programma gia' scritto che abbia una numerazione di linee da 1 a 15 e si desideri cancellare, perche' inutili o per altri motivi, le linee 4-5-6.

Sara' allora necessario selezionare l' upzione CANC. LINEE.

A questo punto ci verra' richiesto per primo da quale linea effettuare la cancellazione. Nel nostro caso partiremo dalla linea 4.

Successivamente verra' richiesto QUANTE linee cancellare e noi, sempre nell' esempio considerato, risponderemo con un 3 perche' si desidera cancellare le linee 4-5-6.

A cancellazione avvenuta il programma stesso verra' AUTOMATICAMENTE rinumerato tenendo conto dell' operazione effettuata.

LIST

Questa opzione, allo stesso modo dell'opzione precedente, consente di listare totalmente o parzialmente il programma,

MEMORIZZAZIONE

Con questa opzione e' possibile memorizzare su disco o su nastro, il programma attualmente in memoria.

Si tratta del programma sorgente, cioe' non ancora assemblato.

Viene creato un file sequenziale con il nome del programma che assegneremo in questa fase.

CARICAMENTO

Un programma salvato su periferica con l' opzione precedente puo' essere in qualsiasi momento ricaricato in memoria.

Ricordiamo che su disco il nome puo' essere abbreviato con un' asterisco, (ma attenzione a non fare confusione), mentre la stessa tecnica non puo' essere usata su cassetta.

NEW

Con questo comando si cancella solo il programma `SORGENTE` che si trova nella memoria del computer. Nessun effetto invece ha questo comando sul programma `ASSEMBLER` presente in memoria.

CONVERSIONE DI UN PROGRAMMA IN DATA

Un sistema conveniente di collegare un programma in codice macchina ad un programma in BASIC e' quello di convertire il programma in codice macchina in una serie di `DATA` e di aggiungerli al programma BASIC.

Nel programma BASIC sara' poi sufficiente inserire un ciclo di lettura e di `POKE` di questi `DATA` in una conveniente zona di memoria.

Vediamo ora come e' possibile convertire dei codici macchina in comandi `DATA`.

Il nostro `ASSEMBLER` non ha una funzione dedicata a questo proposito, in quanto la cosa avrebbe inutilmente appesantito il programma stesso.

A questo proposito riportiamo al termine il listato di un programma per questa operazione.

Vediamo ora di spiegare il funzionamento del programma.

Vengono inizialmente richieste la prima e l'ultima locazione di memoria da convertire in DATA.

La risposta agli indirizzi puo' essere data con valori decimali o esadecimali.

Nel secondo caso i valori devono essere di 4 DIGIT e preceduti dal segno \$ (dollaro).

Immediatamente il programma provvedera' alla conversione in DATA.

Come risultato avremo una serie di linee di programma BASIC con i DATA a partire dalla linea 1000, mentre nelle linee 20 e 30 troveremo la routine di caricamento dei DATA nelle locazioni di memoria specificate.

IL MONITOR

Introduzione

Questo programma Assembler offre un altro sistema, che potremo definire complementare rispetto a quanto visto fino a questo momento, per inserire e modificare i codici macchina.

Questo e' ottenuto con un MONITOR.

Per entrare in ambito Monitor (MACHINE LANGUAGE MONITOR o MLM) e' necessario selezionare l'opzione MONITOR del menu'.

Funzioni MONITOR

Questo capitolo e' diviso nelle sottoindicate sezioni:

PRIMA SEZIONE-INTRODUZIONE AL MONITOR

Questa parte descrive il VICMON in termini generali.

SECONDA SEZIONE-I COMANDI DEL MONITOR

In questa sezione e' spiegato dettagliatamente ogni comando di questa procedura, il suo formato,

il suo uso e sono riportati alcuni esempi. Questa sezione e' stata descritta in ordine alfabetico relativamente ai comandi usati.

LE FUNZIONI DEL MONITOR

Il MONITOR consente le seguenti funzioni:

- Visualizzare una scelta area di memoria.
- Cambiare i contenuti di locazioni di memoria.
- Muovere blocchi di memoria.
- Riempire blocchi di memoria selezionati.
- Ricerca in memoria un determinato valore.
- Esaminare e cambiare i registri principali.
- Immagazzinare e ricercare sulle periferiche dati e programmi.
- Eseguire i programmi a diverse velocita' e con diverse modalita' selezionabili.

INIZIO E PARTENZA DEL MONITOR

Selezionando l' opzione MONITOR seguita dal Return si fa eseguire al programma un' istruzione SYS, cioe' un salto ad una locazione di memoria.

Lo schermo del CBM 64 mostrerà ora i valori dei registri del 6510 a quella locazione di memoria nel seguente formato:

```
B      PC  SR  AC  XR  YR  SP
. ; 603E 33  00  63  00  F6
```

I registri visualizzati sono i seguenti:

PC = Program counter

SR = Stack register

AC = Accumulator

XR = X register

YR = Y register

SP = Stack pointer

Il Program counter riporta in esadecimale la locazione di memoria alla quale siamo saltati.

Riporta inoltre il contenuto dei Flag il cui significato deve essere visto nell'apposito capitolo.

FORMATO DEI COMANDI

Molti comandi del MONITOR sono di un singolo carattere alfabetico seguiti da un parametro se e' richiesto, o se serve, e sono spiegati in dettaglio nella seconda sezione.

I parametri possono includere l' indirizzo di partenza o l' indirizzo di partenza e di fine, il codice operativo o OP-CODE, gli operandi, i valori in esadecimale, ecc.

I comandi sono eseguiti immediatamente dopo aver premuto il tasto di RETURN.

E' da segnalare che rimane in funzione l' editing tipico del CBM 64 per correzioni ed aggiunte, per cui e' sufficiente riposizionarsi sopra i caratteri da correggere usando i cursori in modo diretto o con lo SHIFT, ma i comandi e le correzioni passano dal video al sistema operativo SOLO dopo il RETURN.

INDICAZIONI DI ERRORE

Qualsiasi errore nel quale siate incorsi durante la fase di INPUT sara' segnalato da un punto interrogativo (?) che segue la posizione dell' errore.

Come abbiamo detto si puo' correggere o riscrivere interamente facendo seguire da un colpo di RETURN.

SEZIONE SECONDA

I COMANDI DEL MONITOR

Introduzione

In questa sezione ogni comando del MONITOR viene presentato in ordine alfabetico e ne riportiamo un indice prima di addentrarci nell'esame dei singoli formati.

E' mostrato il formato richiesto, lo scopo e la funzione.

Sono inclusi inoltre un piccolo esempio, la risposta che se ne ottiene dal sistema ed una spiegazione del risultato.

Simbologia e convenzioni

I parametri nei formati comando sono rappresentati secondo il seguente schema:

INDIRIZZO = due Bytes in forma esadecimale es. 0400.

DEVICE o PERIFERICA = un singolo Byte in esadecimale es. 08.

CODICE OPERATIVO o OP-CODE = un codice operativo

in Assembler del 6502 es. LDA, JSR, ecc.

OPERANDO = un operando valido per la precedente istruzione del codice operativo es. \$01.

VALORE = Un singolo Byte contenente un valore esadecimale es. FF.

DATA = Una stringa di dati letterali racchiusa fra parentesi o un valore esadecimale. Successive voci devono essere separate da una virgola.

RIFERIMENTO = Un indirizzo di due Bytes es. 2000.

OFFSET o VALORE DI SALTO = Altro indirizzo di due Bytes.

I COMANDI : QUADRO RIASSUNTIVO

A = ASSEMBLE

D = DISASSEMBLE

F = FILL

G = GO

H = HUNT

L = LOAD

M = MEMORY

R = REGISTER DISPLAY

S = SAVE

T = TRANSFER

X = RETURN TO BASIC

I COMANDI PER ESTESO

A = ASSEMBLA

FORMATO : A (indirizzo)(op-code)(operando).

FUNZIONE : Assembla dei codici operativi partendo da un dato indirizzo.

Il comando consente di inserire, linea dopo linea, codici Assembler e di immagazzinarli in linguaggio macchina direttamente utilizzabile dal microprocessore.

L'indirizzo della successiva locazione di memoria disponibile oltre quello utilizzato dal codice operativo e dall'operando appena inseriti e' posto in attesa di un'altra istruzione.

Per far terminare la funzione A e' sufficiente premere il RETURN dopo l'inserimento dell'ultimo codice operativo.

Se viene inserito un codice operativo o un operando ILLEGALE il MONITOR visualizzera' un punto interrogativo (?) prima della quantita' illegale e ritornera' alla funzione generale del monitor scrivendo un punto (.) in una nuova e successiva linea.

Se si dimentica di specificare un codice operativo o un operando, allora il MONITOR

ignorerà la linea da assemblare e tornerà in ambito Monitor con un punto su una nuova linea.

NB. Ricordare che tutti gli operandi devono essere dati in esadecimale preceduti dal segno dollaro (\$).

ESEMPIO

Inserire il seguente gruppo di comandi:

```
..... LDA&$19
..... JSR$FFD2
..... RTS
```

con inizio all' indirizzo \$1000

COMANDO: A 1000 LDA&\$19 (RETURN)

```
SCHERMO: .A 1000 LDA&$19
..... .A 1002
```

COMANDO: JSR\$FFD2 (RETURN)

```
SCHERMO: .A 1000 LDA&$19
..... .A 1002 JSR $ FFD2
..... .A 1005
```

COMANDO: RTS (RETURN)

```
SCHERMO: .A 1000 LDA&$19
..... .A 1002 JSR $ FFD2
..... .A 1005 RTS
..... .A 1006
```


RISULTATO

L' equivalente in linguaggio macchina del programma Assembler appena descritto e' stato immagazzinato in memoria dalla locazione \$ 1000 alla \$ 1005 inclusa.

N.B. Facciamo notare che l' Assembler del MONITOR calcola automaticamente gli spazi necessari ad ogni codice operativo ed ai suoi operandi.

D = DISASSEMBLA

FORMATO: D(indirizzo)

oppure

D(indirizzo di partenza),(indirizzo di fine)

FUNZIONE: Questo comando serve per disassemblare programmi, routines o in generale gruppi di codici a partire da un certo punto della memoria oppure fra due indirizzi specificati nella seconda parte del comando.

Il comando D consente di riconvertire i codici presenti nella memoria del computer e quindi in

formato binario (anche se ricordiamo che vengono visualizzati byte per byte in forma esadecimale), nel corrispondente linguaggio ASSEMBLER.

Si puo' specificare un indirizzo di inizio, nel qual caso verra' disassemblata la linea corrispondente a quell' indirizzo.

In questo modo il sistema restera' in ambito del comando DISASSEMBLER e si potra' usare il cursore per disassemblare le altre linee.

Usando infatti la funzione CURSOR-DOWN saranno disassemblate le linee successive alla prima, mentre con il CURSOR-UP quelle precedenti.

E' tuttavia da notare un particolare e cioe' che queste funzioni NON inizieranno fin quando il cursore non si trovera' o in cima o in fondo allo schermo.

Questa funzione e' tipica del Sistema Operativo del CBM 64 ed infatti risultati simili, pur ovviamente con altri comandi, si ottengono anche in ambito BASIC.

ATTENZIONE

Facendo eseguire questi scrolling in alto o in basso con il cursore si possono NON ottenere dei risultati validi a causa dell' inaccurata traduzione dei codici dal linguaggio macchina all' ASSEMBLER. Cio' lo abbiamo notato in particolare usando la funzione SCROLL-UP.

In alternativa si puo' specificare la parte di memoria da disassemblare. In questo caso le linee specificate saranno visualizzate sullo schermo

una dopo l'altra.

Naturalmente se le linee da disassemblare sono troppe rispetto alla capacita' dello schermo, il relativo contenuto scorrera' verso l'alto.

Per fermare lo scrolling e' necessario premere il tasto di RUN/STOP.

Con questa operazione si resta in ambito DISASSEMBLER, infatti questa funzione puo' essere continuata con il tasto CURSOR-DOWN.

Quando ci troviamo in ambito Disassembler una linea di codice puo' essere modificata o riscritta usando l'editor del CBM 64, cioe' semplicemente riposizionandoci sopra e riscrivendola da capo. Ricordarsi poi di premere il RETURN.

Usando questo sistema si attiva automaticamente il comando A: per l'assemblaggio.

Qualora si sia entrati in modo Assembler il cursore rimane posizionato dopo l'indirizzo sulla linea seguente la linea corretta.

Per uscire dal modo Assembler eseguire un clear di schermo (ricordiamo che si fa premendo contemporaneamente il tasto di SHIFT e quello di CLR/HOME) e dopo premere il RETURN.

ESEMPIO

Si desidera disassemblare le linee di codice macchina inserite nell'esempio sull'utilizzo del comando ASSEMBLER visto in precedenza e cambiare l'indirizzo della seconda linea da FFD2 a FFD0.

COMANDO: D 1000,1005 (RETURN)

```
SCHERMO: . 1000 LDA £$19
          . 1002 JSR $ FFD2
          . 1005 RTS
```

AZIONE: posizionare il cursore in modo da essere sul 2 della scritta FFD2.

SCRIVERE: 0 (RETURN)

```
SCHERMO: . 1000 LDA £$19
          .A1002 JSR $FFD0
          .A1005 RTS
```

RISULTATO

Il codice macchina e' disassemblato dalla locazione \$1000 alla \$1005. E' stato eseguito il cambiamento richiesto e successivamente immesso in memoria con il tasto RETURN.

Come detto in precedenza si puo' a questo punto uscire dal modo ASSEMBLER.

F = FILL memory(riempi la memoria)

FORMATO: F(indirizzo di partenza),(indirizzo di fine),(valore)

FUNZIONE: Riempie la memoria contenuta fra due specificati indirizzi con un dato valore.

Il comando F consente di inserire un valore NOTO entro uno specifico blocco di memoria.

Cio' puo' essere utile per inizializzare una struttura di dati o per ripulire il contenuto di un' area di memoria.

Per questo motivo il comando F deve contenere tutti i parametri enunciati e cioe' l' indirizzo di partenza, l' indirizzo di fine ed il dato da caricare nella memoria compresa fra questi due indirizzi.

Il dato deve essere sempre espresso in forma esadecimale.

Naturalmente, dato che trattasi di un lavoro a blocchi non si devono usare le locazioni da \$0000 a \$01FF cioe' la pagina ZERO e UNO della memoria del CBM 64 senza usare particolari protezioni come ad esempio quella vista in precedenza.

ESEMPIO

Si desideri scrivere il dato \$EA (cioe' una istruzione cosi' detta NON OPERATIVA) dalla locazione \$1000 alla locazione \$2000 inclusa.

COMANDO: F 1000,2000,EA (RETURN)

RISULTATO

L'istruzione non operativa EA e' stata

immediatamente scritta nelle locazioni richieste.

G = GO (vai)

FORMATO: G

oppure

.....G(indirizzo)

FUNZIONE: Serve per far incominciare a girare un programma partendo dalla attuale locazione contenuta nel Program Counter, oppure, nel secondo formato, iniziando da un determinato indirizzo.

Il comando G puo' essere usato da solo o unitamente ad un indirizzo di partenza.

Nel primo caso il 64 eseguirà il programma in memoria o la subroutine del Sistema Operativo iniziando dalla locazione contenuta in quel momento nel Program Counter.

E' necessario fare attenzione nell' uso delle subroutines del Sistema Operativo perche' non conoscendone bene il loro uso e' facile entrare in un LOOP o ciclo infinito.

Per visualizzare il contenuto dei vari registri ed in questo caso ricordiamo che puo' essere importante vedere l' indirizzo contenuto nel

Program Counter usare il comando R come spiegato in seguito.

Nel caso invece si usi G seguito da un indirizzo, allora l' esecuzione del programma partirà dalla locazione di memoria specificata dall' indirizzo dell' istruzione stessa.

Il comando G riporta i registri al loro ultimo valore conosciuto.

Se il programma termina con un RTS (Return from Subroutine, cioè ritorno da subroutine) allora torneremo in ambito Assembler.

Se invece l' ultimo comando incontrato nel programma (che quindi non è necessariamente la fine del programma stesso) è un BRK (BReak), allora resteremo in ambito MONITOR.

Nel caso che non esista nessuna istruzione di termine programma o di STOP o comunque non si verifichi nessuna condizione di fine sarà necessario interrompere l' esecuzione altrimenti infinita, con il tasto di RUN/STOP e RESTORE.

Come detto prima usciremo dall'ambito MONITOR per tornare in ASSEMBLER, ma potremo anche rientrare in MONITOR senza perdere il programma.

NOTA

Se nel vostro programma ci sono state variazioni al colore di schermo o sia stato cambiato il colore delle lettere può verificarsi il caso che non riusciate a leggere i registri visualizzati o lo scritta READY.

Ritornandoci sopra con il cursore riusciremo però a visualizzarne il contenuto.

ESEMPIO

Ipotizziamo di avere un programma in memoria e di desiderare che esso vada in esecuzione a partire non dall'inizio ma dalla linea presente alla locazione \$2000.

COMANDO: G 2000(RETURN)

RISULTATO

I registri vengono ripristinati. Il Program Counter viene fissato a \$2000. Se e' stata scelta una pagina ZERO virtuale e' in questo momento che il Monitor effettua lo scambio.

Dopo aver fatto questo il programma inizia la sua esecuzione a partire dalla istruzione contenuta in \$2000.

H = HUNT (ricerca)

FORMATO: H(indirizzo di partenza), (indirizzo di fine),(dati).

FUNZIONE: Cerca in un blocco di memoria specificato dagli indirizzi dei dati o delle stringhe di caratteri.

Il comando HUNT localizza ogni selezionato gruppo di caratteri in memoria e li visualizza sullo

schermo.

Si puo' utilizzare questo comando per localizzare dati che devono essere espressi in forma esadecimale o per trovare stringhe di caratteri di una lunghezza massima di 88 caratteri.

Le stringhe devono essere specificate in forma letterale e precedute dal segno '(accento).

Le locazioni contenenti i dati o le stringhe saranno visualizzate con il relativo indirizzo.

Nel caso siano presenti piu' locazioni di quante ne possa contenere lo schermo, i dati visualizzati scorreranno in alto.

Per terminare lo scrolling dello schermo sia per interrompere la funzione H sara' necessario premere il tasto di RUN/STOP. In questo caso resteremo in ambito Monitor.

Per far scorrere lentamente lo schermo e' sufficiente premere il tasto di controllo.

I ESEMPIO

Ammettiamo che il gruppo di dati \$A9 2F 3C sia immagazzinato in memoria in una parte qualsiasi ma compresa fra gli indirizzi \$C000 e \$COFF.

Per localizzarla con i relativi indirizzi opereremo come segue:

COMANDO: H C000,COFF,A9,2F,3C (RETURN)

RISULTATO

Viene esaminata la memoria fra le locazioni assegnate e se il gruppo di dati richiesto e' presente viene visualizzato con l' indirizzo

accanto.

II ESEMPIO

Vogliamo cercare la locazione esatta della parola `COMMODORE` che sappiamo essere presente fra le locazioni di indirizzo `$2000` e `$3000`.

COMANDO: `H 2000,3000,'COMMODORE (RETURN)`

RISULTATO

Saranno visualizzate sullo schermo le locazioni di memoria ai cui indirizzi inizia la stringa richiesta.

Accanto a questi indirizzi verra' visualizzata la parola `COMMODORE` in reverse.

`L = LOAD (carica)`

FORMATO; `L "nome del file", (numero della periferica).`

FUNZIONE: Carica in memoria il contenuto del file da una data periferica.

Il comando `LOAD (L)` consente, nello stesso modo del Basic, di caricare un file di dati o un programma da un determinata periferica nella

memoria RAM del CBM 64.

Si possono quindi caricare files da disco o da cassetta.

Per i files disco, l' indirizzo della prima locazione RAM entro la quale il file dovra' essere letto deve essere costituita dai primi due Bytes del file.

I files provenienti da cassetta hanno come indirizzo di inizio parte dell' HEADER BLOCK iniziale.

ATTENZIONE

Con questo comando si possono caricare solo programmi o dati che siano stati precedentemente salvati su una periferica o con il comando S del MONITOR o con il comando SAVE del BASIC del CBM 64 mentre non si possono caricare dati o programmi da cartridge.

Il comando e' composto dalla lettera L, dal nome del file e dal numero di periferica da cui deve essere letto.

Il nome del file deve essere racchiuso fra apici o virgolette ("") e naturalmente si deve applicare la sintassi generale del Basic per questa operazione.

Ricordiamo che il numero di periferica per la cassetta e' 01 mentre quello del disco e' 08.

Quando viene usato il comando L, il file specificato fra virgolette e' letto fino a quando non si incontri un EOF (END OF FILE).

Nel caso non venga trovato il carattere di EOF

(o anche di EOT cioe' di END OF TAPE), e questo puo' accadere su ricerche da cassetta ed in particolare per file di dati, la funzione di LOAD non ha termine.

Anche questo classico esempio di LOOP infinito puo' essere arrestato con i tasti di RUN/STOP e di RESTORE.

Nel caso invece che il file non sia trovato sara' visualizzato il solito messaggio di errore ed il CBM 64 sara' riportato in ambito Assembler.

ESEMPIO

Ammettiamo di avere su disco un programma di nome TEST, che sia lungo 258 Bytes e che i primi due Bytes siano rispettivamente 00 e CA.

Si desidera caricare il file in memoria.

COMANDO: L"TEST",08 (RETURN)

RISULTATO

Il programma TEST presente e trovato sul dischetto e' caricato in memoria a partire dalla locazione \$CA00 alla locazione \$CB00 inclusa.

M = MEMORY

FORMATO: M (indirizzo)

oppure

M (indirizzo di partenza),(indirizzo di fine)

FUNZIONE: Visualizza i codici esadecimali contenuti in memoria.

Il comando M visualizza il contenuto della memoria dall' indirizzo di partenza specificato nel parametro all' indirizzo di fine incluso.

La visualizzazione mostrerà l' indirizzo in esadecimale ed il contenuto, sempre in esa, di 5 bytes di memoria.

Se invece di un blocco di memoria viene dato solo un indirizzo, allora saranno visualizzati i codici esadecimali (sempre 5 a riga) a partire da quell' indirizzo.

Gruppi di 5 bytes in più possono essere esaminati come al solito eseguendo lo scroll di schermo tramite l' uso dei tasti di controllo cursore.

Il contenuto della memoria può essere cambiato riscrivendo sopra al valore visualizzato e premendo successivamente il return.

L' indirizzo della prima locazione di memoria esaminata, relativamente al gruppo dei 5 bytes appare alla sinistra della riga.

Se si tenta di modificare i valori contenuti in zone di memoria riservate, come ad esempio i valori contenuti in ROM, allora accanto alla locazione di memoria immutabile apparirà un punto interrogativo (?) a segnalare l'

impossibilita' della operazione.

ESEMPIO

Si desidera visualizzare i cinque bytes di memoria con indirizzo di partenza \$1000 e variarne il contenuto.

COMANDO: M 1000 (RETURN)

SCHERMO: 1000 A0 00 EA EA FF

OPERAZIONE :Posizionare il cursore sopra il primo 0 della seconda locazione di memoria, cioe' quella che si trova a 00.
Digitare quindi FF e RETURN.

RISULTATO

I primi 5 Bytes di memoria con indirizzo alla locazione \$1000 si leggono ora:

A0 FF EA EA FF

R = REGISTER

FORMATO: R

FUNZIONE: Visualizza il contenuto dei registri.

Il comando R consente di vedere sullo schermo il contenuto dei seguenti registri del microprocessore 6510:

PC = Program Counter

SR = Status Register

AC = Accumulator

XR = Registro X

YR = Registro Y

SP = Stack Pointer

FLAGS

Questo comando puo' essere utile quando si sta provando un programma, perche' R vi consentira' di osservare se i registri contengono i valori che si desidera.

Si puo' cambiare il valore degli stessi registri quando ci si trova nel modo R, molto semplicemente riposizionandosi sopra i valori che appaiono sullo schermo, variandoli e premendo poi il RETURN.

ATTENZIONE

Quando si effettua un controllo dei registri e piu' ancora quando se ne cambia il contenuto di qualcuno sarebbe bene prendere nota scritta di quello che appare e di quello che si cambia.

I registri suddetti vengono automaticamente visualizzati quando entra in funzione il Monitor.

ESEMPIO

Visualizzare il contenuto dei registri

COMANDO:R (RETURN)

RISULTATO

Viene visualizzato il contenuto dei registri in questo formato:(I contenuti sono pero' ipotetici)

.R

	PC	SR	AC	XR	YR	SP
. ;	0401	33	00	63	00	F6

S = SAVE

FORMATO: S "nome del file", (numero della periferica), (indirizzo), (indirizzo)

FUNZIONE: Scrive il contenuto di una data zona di memoria su una particolare periferica che potrà essere disco o cassetta.

Il comando S (SAVE) consente di salvare un programma o un gruppo di dati su cassetta o su disco per utilizzarli poi in un secondo tempo.

I parametri del comando S consistono nel nome del FILE, nel numero della periferica (ricordiamo 01 per la cassetta e 08 per il disco) e negli indirizzi di inizio e fine dati che devono essere in questo caso specificati a differenza di quanto avviene per il simile comando SAVE del Basic. Gli indirizzi di inizio e fine sono naturalmente indirizzi esadecimali della memoria RAM nella quale è presente in quel momento il file da salvare.

Il nome del File deve essere racchiuso fra virgolette (") e deve obbedire alle regole di sintassi dei comandi per la gestione dei files del CBM 64.

Ricordiamo quindi che per esempio deve iniziare con un carattere alfabetico e non deve essere più lungo di 16 caratteri.

L'indirizzo iniziale deve essere quello della

locazione di memoria in cui incomincia il file, mentre quello finale deve essere di un Byte in piu'.

ATTENZIONE

- Se l' indirizzo finale non e' aumentato di un byte rispetto al reale, verra' perso l'ultimo carattere del file.

- Se la periferica specificata nel relativo parametro non e' presente sara' segnalato un messaggio di errore:

?DEVICE NON PRESENT ERROR

e torneremo in ambito Assembler.

ESEMPIO

Ipotizziamo di avere un programma in memoria dalla locazione \$1000 alla locazione \$10FF e si desidera scrivere il programma su disco con il nome di TEST 1.

COMANDO: S "TEST 1",08,1000,1100 (RETURN)

RISULTATO

Il file programma denominato TEST 1 e' salvato su

disco. Questo File conterra' il contenuto delle locazioni RAM dall' indirizzo \$1000 a \$10FF incluso.

T = TRANSFER

FORMATO: T(indirizzo),(indirizzo),(indirizzo)

FUNZIONE: Serve per trasferire i contenuti di un blocco di memoria RAM ad un' altra area di memoria.

Questo comando vi consente di rilocare un programma o dei dati in un' altra parte della memoria.

Questo puo' risultare utile qualora si desideri espandere un programma o se si vuole usare parti di un programma o di dati senza essere costretti a riscriverli.

Nel comando sono presenti 3 parametri relativi a tre diversi indirizzi.

I primi due delimitano il blocco di memoria che deve essere duplicato mentre il terzo indica l' indirizzo di inizio della copia.

Se il programma da trasferire contiene indirizzi assoluti o WORD TABLE, il trasferimento avverra' ma questi dati non avranno piu' una logica all' interno delle funzioni del programma.

ESEMPIO

Ipotizziamo di avere un blocco di dati qualsiasi (programma, subroutines o data) in memoria dalla locazione \$3000 alla locazione \$3500 e che si vogliano avere ANCHE a partire dalla locazione \$4000.

ATTENZIONE

La parola ANCHE usata nell' ultima riga sta a significare che si tratta di una vera e propria duplicazione e non di un trasferimento che nel senso stretto del termine lascerebbe la zona di memoria iniziale vuota.

COMANDO: T 3000,3500,4000 (RETURN)

RISULTATO

I dati sono ora presenti sia nelle locazioni di memoria da \$3000 a \$3500 che da \$4000 a \$4500.

FORMATO: X

FUNZIONE: Serve per uscire dall' ambito Monitor e tornare in Assembler.

L' uso di questo comando vi riporterà in ambiente Assembler.

Ricordiamo solo che l'eventuale programma in Linguaggio Macchina resta immagazzinato in memoria.

ESEMPIO

Si desidera tornare la Basic

COMANDO: X(RETURN)

RISULTATO

Si rientra così all'interno del programma Assembler e sarà visualizzato il menu.

CAPITOLO SESTO

Tutti i programmi in codice macchina sono stati inseriti tramite l'Assembler fino a questo momento.

Tuttavia, come abbiamo visto nell'ultimo capitolo, l'Assembler non e' la sola strada per inserire codici macchina che appunto possono essere caricati sotto forma di POKE come nel seguente esempio:

Programma 6.1

```
POKE 828,160
POKE 829,1
POKE 830,162
POKE 831,0
POKE 832,169
POKE 833,90
POKE 834,157
POKE 835,0
POKE 836,4
POKE 837,152
POKE 838,157
POKE 839,0
POKE 840,216
POKE 841,232
POKE 842,208
POKE 843,244
POKE 844,96
```


Non possiamo inserire questo programma in memoria mentre sta girando l' assembler.

Per prima cosa selezioniamo dal menu principale l' opzione MONITOR per uscire. Il CBM 64 andra' in READY e sara' pronto per ricevere i comandi.

Quando e' stato inserito questo piccolo programma, puo' essere fatto girare con un comando :

SYS 828

che fara' visualizzare 256 quadri bianchi sullo schermo.

Vediamo come si presenta in linguaggio Assembler questo programma.

Per prima cosa facciamo di nuovo girare l' Assembler e dopo aver selezionato L, facciamoci listare il programma a partire dall' indirizzo iniziale cioe' 828.

Vediamo di seguito il listato DISASSEMBLATO del programma inserito con i comandi POKE:

Programma 6.1a

```
CICLO = $0340
1          *          = 828
2  033C A0 01      LDY #1
3  033E A2 00      LDX #0
4  0340 A9 5A      LDA #90
5  0342 9D 00 04   STA 1024,X
6  0345 98          TYA
7  0346 9D 00 D8   STA 55296,X
8  0349 E8          INX
9  034A D0 F4      BNE CICLO
10 034C 60          RTS
```

Allo stesso modo che e' stato possibile inserire un programma senza l' aiuto dell' Assembler e' anche possibile farlo girare direttamente o da un programma Basic.

Come abbiamo visto per farlo girare direttamente e' sufficiente comunicare al Program Counter l' indirizzo di partenza con un SYS all' indirizzo stesso.

Proviamo ora a farlo girare DA un programma Basic.

Programma 6.2

```
20000 PRINT "clear" Far eseguire un Clear di
schermo
20010 SYS 828
20020 PRINT "prova"
```

digitiamo quindi:

```
RUN 20000
```

e vediamo che il programma girera' visualizzando i soliti 256 quadri seguiti pero' questa volta dalla scritta prova.

Cosa accade?

Linea 20000 Il programma Basic esegue un CLEAR di schermo

Linea 20010 Viene preso il controllo dal codice macchina a partire dalla locazione di memoria 828 e vengono eseguite le istruzioni relative fino a quando non si incontri un RTS che fara' ritornare nuovamente il sistema in ambito Basic.

Linea 20020 Il Basic fa stampare il messaggio "prova".

Come abbiamo visto far girare un programma direttamente e' facile ma l'inserimento con i POKE che abbiamo appena visto e' particolarmente noioso, mentre piu' semplice appare la via dei comandi DATA. Vediamone un esempio.

Programma 6.3

```
1 FOR X = 0 TO 16
2 READ A
3 POKE (828+X),A
4 NEXT X : RESTORE
5 DATA 160, 1, 162, 0, 169, 90, 157, 0, 4
6 DATA 152, 157, 0, 216, 232, 208, 244, 96
7 END
```

Questo programma gira come un normale programma Basic con un semplice RUN.

I COLORI NEL CBM 64

Una delle maggiori capacita' del CBM64 e' quella di visualizzare i colori.

Cio' e' disponibile e facilmente usabile sia operando in Basic che in codice macchina.

Il seguente programma mostra una combinazione di colori schermo/bordo.

Programma 6.6

```
VIDEO = $0340
BORDO = $0343
1          *          = 828
2  033C A0 0F          LDY #15
3  033E A2 0F          LDX #15
4  0340 8C 21 D0 VIDEO STY $D021
5  0343 8E 20 D0 BORDO STX $D020
6  0346 CA            DEX
7  0347 10 FA          BPL BORDO
8  0349 88            DEY
9  034A 10 F4          BPL VIDEO
10 034C 60            RTS
```

Sfortunatamente questo programma gira in 2600 cicli pari a 1300 microsecondi e diviene percio' non percettibile dai nostri occhi.

Per poterlo osservare sara' quindi necessario ricorrere a dei cicli di ritardo.

Nei precedenti capitoli abbiamo visto molte cose sia sui cicli sia sui ritardi.

Ricordiamo che agli indirizzi di memoria in pagina zero 160, 161 e 162 (\$A0, A1 e A2) esiste un orologio (JIFFY CLOCK) che funziona come un contatore binario.

Il byte di indirizzo 162 (\$A2) viene incrementato

di 1 ogni sessantesimo di secondo. Quando arriva a 256 scarica 1 sul byte 161 che a sua volta si incrementa fino ad arrivare a 256 dopo di che fara' scattare il contatore di locazione 160 di uno.

Tutto cio' ci consente di manipolare ritardi di qualsiasi misura come nel programma seguente:

Programma 6.7

```

BORDO =    $033E
VIDEO  =    $0343
LOOP   =    $034A
1      *      =    828
2      033C A0 0F      LDY    #15
3      033E 8C 20 D0    BORDO   STY    $D020
4      0341 A2 0F      LDX    #15
5      0343 8E 21 D0    VIDEO   STX    $D021
6      0346 A9 F6      LDA    #246
7      0348 85 A2      STA    162
8      034A A5 A2      LOOP   LDA    162
9      034C 30 FC      BMI    LOOP
10     034E CA          DEX
11     034F D0 F2      BNE    VIDEO
12     0351 88          DEY
13     0352 D0 EA      BNE    BORDO
14     0354 60          RTS

```

Anche sui singoli caratteri sullo schermo possono essere controllati i codici di colore come del resto abbiamo fatto nel nostro ultimo programma. Se all' indirizzo di memoria 55296 (\$D800) c'e' un "2" allora la locazione di memoria 1024, cioe' la prima locazione a sinistra in alto sullo schermo, sara' rossa. 0 meglio il carattere che verra' visualizzato in quella locazione di

memoria sara' rosso.

Ecco la tabella completa dei colori:

CODICE	COLORE
0	Black
1	White
2	Red
3	Cyan
4	Purple
5	Green
6	Blue
7	Yellow
8	Orange
9	Brown
10	Light red
11	Dark grey
12	Mid grey
13	Light green
14	Light blue
15	Light grey

Il programma 6.8 mostra come il colore dei blocchi di schermo possa essere definito per mezzo dei codici di colore dello schermo stesso.

Programma 6.8

```
LOOP = $033E
1 * = 828
2 033C A2 C8 LDX #200
```


3	033E	A9	08		LOOP	LDA	#8
4	0340	9D	FF	D7		STA	\$D7FF,X
5	0343	A9	A0			LDA	#160
6	0345	9D	FF	03		STA	1023,X
7	0348	CA				DEX	
8	0349	D0	F3			BNE	LOOP
9	034B	60				RTS	

CAPITOLO SETTIMO

Per utilizzare il 6510 il nostro computer ha immagazzinato in ROM (Read Only Memory) una serie di routines di controllo.

Queste routines consentono al CBM 64 di riconoscere ed utilizzare i comandi Basic che fanno parte di un programma, tutti gli input e gli output, intesi qui nel senso di ingresso e uscita dati, e tutte le procedure comunque necessarie al suo funzionamento.

Le ROM che manipolano tutti i comandi Basic sono localizzate in memoria fra gli indirizzi \$A000 e \$BFFF.

Le ROM che si occupano di tutte le altre routines del Sistema Operativo del computer, chiamate KERNAL dalla Commodore sono fra \$E000 e \$FFFF.

Abbiamo quindi una suddivisione, abbastanza elastica da considerare fra i due grandi blocchi di istruzione:

IL BASIC

IL SISTEMA OPERATIVO

In aggiunta alle locazioni in ROM sia il Basic che il S.O. utilizzano parte della memoria RAM di indirizzi fra \$0000 e \$03FF, cioè le prime quattro pagine.

In particolare come abbiamo già detto in precedenza viene largamente utilizzata la pagina

zero.

Buona parte di questo utilizzo e' per l'immagazzinamento di dati transienti, cioe' temporanei o che si modificano in continuazione come per esempio abbiamo visto il JIFFY CLOCK, che ci dice anche da quanto tempo e' stato acceso il computer.

Altre zone RAM sono usate per dati con maggiore stabilita' o utilizzate come le locazioni da 43 a 56 (\$2B fino a \$38) che ci indicano l'area di memoria usata dai programmi Basic e le relative aree di dati.

Alcune di queste indicazioni si servono di due o tre Bytes mentre altre, come per esempio il Buffer di cassetta, che viene utilizzato durante il trasferimento di dati da e per la cassetta, sono di dimensioni maggiori.

Infatti il Buffer di cassetta occupa ben 192 Bytes.

L'aspetto piu' difficile nell'uso delle routines inserite (BUILT-IN SUBROUTINES) e' di conoscere da che parte arrivano a loro le informazioni e dove esse depositano i dati che producono. Cio' e' particolarmente vero per quanto riguarda le routines che appartengono al gruppo dell'Interprete Basic.

Al termine di questo manuale si trovano delle tavole, abbastanza complete, che descrivono gli indirizzi e le azioni delle routines Commodore.

Tuttavia molte funzioni saranno esemplificate e commentate nel presente e nei successivi capitoli.

Per prima cosa diamo un'occhiata ai contenuti

dell' Accumulatore durante l' uso di una subroutine kernal.

Nei primi capitoli abbiamo visto l' Accumulatore durante l' uso di un comando STA per spostare una copia dello stesso in una locazione di memoria, es. STA 1024.

Un sistema semplice e piu' facile e' quello di usare una routine Kernal che si chiama CHROUT di indirizzo 65490 (\$FFD2). Questa routine consente di visualizzare il contenuto dell' Accumulatore a partire dall' attuale posizione del cursore.

Vediamone un' applicazione:

Programma 7.1

1				*	=	828
2	033C	92	2A		LDA	#42
3	033E	8D	F4	05	STA	1524
4	0341	A2	01		LDX	#1
5	0343	8E	F4	D9	STX	55796
6	0346	20	D2	FF	JSR	\$FFD2
7	0349	60			RTS	

Dopo il RUN saranno visualizzati due asterischi. Nel mezzo dello schermo un asterisco bianco che e' stato immesso direttamente.

Avremo inoltre un' altro asterisco probabilmente in 1024 e probabilmente in bleu chiaro.

Questo e' il risultato dell' uso della routine

CHROUT. Notiamo che questa subroutine non ha specificato ne' la posizione ne' il colore.

I due maggiori vantaggi di questa subroutine sono che per prima cosa localizza automaticamente la posizione del cursore e la incrementa, sempre automaticamente, tutte le volte che la routine stessa e' chiamata.

Secondo viene immagazzinato l' attuale colore nell' appropriata posizione RAM colore.

Se facciamo girare il programma, ad esempio quello precedente, partendo con l' assembler in memoria allora la corrente posizione del cursore sara' a 1024, perche' l' Assembler stesso prima di girare esegue un CLEAR di schermo e fissa come colore il bleu chiaro.

Naturalmente se si desidera si puo' diversamente fissare, tramite il programma che scriveremo, una diversa posizione del cursore e un colore diverso.

Fissare un' altro colore e' relativamente facile. Infatti il colore attuale (del cursore) e' localizzato in 646 (\$0286) con i soliti valori: 0 per il nero, 1 per il bianco, ecc.

In questo modo basta immettere il colore desiderato ed e' tutto cio' che e' necessario fare.

Il posizionamento del cursore e' un po' piu' complesso.

Fortunatamente una routine, chiamata giustamente PLOT di indirizzo 65520 (\$FFF0) esegue la maggior parte del lavoro.

PLOT puo' leggere o fissare l' attuale posizione

del cursore usando i registri X e Y.

Se facciamo entrare questa routine con il flag di Carry settato (cioe' a 1), allora PLOT riporterà l' attuale posizione del cursore nei registri X e Y, dove X conterra' il numero della riga (da 1 a 24) e Y il numero della colonna (da 0 a 39).

Se questa routine e' inserita con il flag di Carry CLEAR (cioe' a 0), allora il valore che e' stato immagazzinato in X e Y sara' usato per posizionare il cursore.

Vediamo nel seguente esempio come si puo' usare la subroutine PLOT per immettere un asterisco giallo all' inizio della decima linea di schermo.

Programma 7.2

1				*	=	828
2	033C	18			CLC	
3	033D	A2	09		LDX	#9
4	033F	A0	00		LDY	#0
5	0341	20	F0	FF	JSR	\$FFF0
6	0344	A9	07		LDA	#7
7	0346	8D	86	02	STA	646
8	0349	A9	2A		LDA	#42
9	034B	20	D2	FF	JSR	\$FFD2
10	034E	60			RTS	

Vediamone un breve commento

- 1 Esegui il CLEAR
- 2 Carica 9 in X (per la decima linea)
- 3 Carica 0 in Y (per la prima colonna)
- 4 Vai a PLOT per posizionare il cursore.
- 5 Carica il valore 7 per il colore giallo.
- 6 Immetti il contenuto di A (7) come colore corrente in 646
- 7 Carica l' asterisco
- 8 Salta a CHROUT per stampa.
- 9 Ritorno da Subroutine

Se avessimo desiderato inserire il nostro asterisco nella diciottesima posizione della decima linea avremmo dovuto immettere il valore 17 nell' istruzione LDY (la prima) avendo così il seguente programma:

Programma 7.3

1		*	=	828
2	033C	18	CLC	
3	033D	A2 09	LDX	#9
4	033F	A0 11	LDY	#17
5	0341	20 F0 FF	JSR	\$FFF0
6	0344	A9 07	LDA	#7
7	0346	8D 86 02	STA	646
8	0349	A9 2A	LDA	#42
9	034B	20 D2 FF	JSR	\$FFD2
10	034E	60	RTS	

Per illustrare come la routine CHROUT manovra il

cursore in modo tale che sia spostato alla
 posizione successiva dopo ogni chiamata di
 routine, proviamo a modificare il programma 7.3
 con:

Programma 7.3a

```

LOOP =      $033C
  1          *          =      828
  2    033C  0A  04          LDY  #4
  3    033E  20  D2  FF    LOOP  JSR  $FFD2
  4    0341  88          DEY
  5    0342  D0  FA          BNE  LOOP
  6    0344  60          RTS
  
```

per cui il nuovo programma sarà:

Programma 7.3b

```

LOOP =      $034D
  1          *          =      828
  2    033C  18          CLC
  3    033D  A2  09          LDX  #9
  4    033F  A0  11          LDY  #17
  5    0341  20  F0  FF          JSR  $FFF0
  6    0344  A9  07          LDA  #7
  7    0346  8D  86  02          STA  646
  8    0349  A9  2A          LDA  #42
  9    034B  A0  04          LDY  #4
 10    034D  20  D2  FF    LOOP  JSR  $FFD2
 11    0350  88          DEY
 12    0351  D0  FA          BNE  LOOP
 13    0353  60          RTS
  
```


Quando gira il programma 7.3b visualizzerà quattro asterischi gialli nella linea 10 rispettivamente nelle colonne 18, 19, 20 e 21.

Notare che non è stato necessario ricaricare l'Accumulatore con il valore 42 tutte le volte che è entrato in funzione il ciclo (LOOP) perché la CHROUT non ne altera il valore.

È anche chiaro che il registro Y non è stato cambiato. In caso contrario infatti il conteggio dei 4 asterischi non avrebbe funzionato.

Come abbiamo detto infatti la routine CHROUT non varia né A, Y o X. E questa è un'ottima qualità di questa utilissima routine.

Purtroppo non tutte le routines di inserimento si comportano in questo modo e spesso sarà necessario tenere in considerazione la possibilità che invece i registri, uno o più di uno, siano cambiati durante il loro funzionamento.

Molti programmi Basic usano il comando GET, che accetta un singolo Byte di ingresso entro il Buffer di tastiera. Prende in pratica un carattere per volta.

Per questa funzione il GET usa una delle routines Kernal chiamata GETIN di indirizzo 65508 (\$FFE4).

Quando entra in funzione questa routine o quando viene chiamata, GETIN rintraccia un carattere dalla coda di tastiera e lo immette, come valore ASCII, nell'Accumulatore.

Se la coda di tastiera (KEYBOARD QUEUE) è vuota, GETIN non attende, ma riporta un valore zero.

Il programma 7.4 mostra una operazione con GETIN:

Programma 7.4

```
LOOP =      $033E
1          *          =      828
2  033C A9 00          LDA  #0
3  033E 20 E4 FF LOOP JSR  $FFE4
4  0341 F0 FB          BEQ  LOOP
5  0343 20 D2 FF      JSR  $FFD2
6  0346 60            RTS
```

Quando gira questo programma fissa il ciclo:

```
LOOP      JSR $FFE4
          BEQ LOOP
```

che resta in attesa di un input.

Quando viene effettuato l'ingresso dati, cioè l'INPUT, il programma passa attraverso l'istruzione BEQ ed esegue il resto del programma stesso, visualizzando il carattere ottenuto per mezzo di GETIN.

Notare che è stato scelto di usare la subroutine CHROUT per immettere il risultato nello schermo invece di porcelo direttamente.

Sia GETIN che CHROUT operano con il codice ASCII invece del codice schermo CBM64.

Ricordiamo che solo nel caso dei numeri i due codici corrispondono, altrimenti operando in un modo o nell'altro dovremo ricercare i relativi valori nelle tavole per visualizzare gli stessi risultati.

La routine CHRIN di indirizzo 65487 (\$FFCF) è una alternativa a GETIN.

Quando si esegue un input da tastiera, la sua azione è simile a quella del comando INPUT del

Basic.

Per esempio, la prima volta che si chiama GETIN, il cursore scompare dallo schermo e non riappare fino a quando non si digiti RETURN (CHR\$ (13)).

I caratteri che erano stati immessi sono immagazzinati nel buffer del Basic che inizia da 512 (\$0200). Vengono inoltre accettati tutti i comandi di edit compresi gli INSERT ed i DELETE.

Tuttavia non abbiamo la necessita' di organizzare la ricerca di questi caratteri nel Buffer del Basic, perche' i caratteri stessi saranno restituiti in sequenza dopo ogni salto o chiamata di CHRIN.

Non e' neppure necessario organizzare la visualizzazione di questi caratteri perche' anche questa parte del lavoro e' organizzata da CHRIN.

Tutto questo lavoro puo' essere quindi sintetizzato in un breve programma:

Programma 7.5

```
..... JSR $FFCF  
..... RTS
```

La routine GETIN puo' essere usata per mettere a punto una vostra routine di INPUT.

Si potrebbe usare GETIN per immettere un carattere per volta, eseguire il controllo di entrata per un certo numero di caratteri oppure per l'inserimento di un certo carattere di termine che non debba essere necessariamente un RETURN.

Si potrebbe anche utilizzare la routine per

controllare ogni carattere immesso attraverso l' attuale comando di INPUT e dare una segnalazione di ATTENZIONE se e' un carattere non valido.

Il seguente programma mostra un esempio di quanto detto con controllo di inserimento di una virgola (ASCII 44):

Programma 7.6

```

LOOP =      $0341
1          *          =      828
2      033C A2 2C          LDX  #44
3      033E 8E 84 03      STX  900
4      0341 20 E4 FF      JSR  $FFE4 LOOP
5      0344 F0 FB          BEQ  LOOP
6      0346 20 D2 FF      JSR  $FFD2
7      0349 CD 84 03      CMP  900
8      034C D0 F3          BNE  LOOP
9      034E A9 0D          LDA  #13
10     0350 20 D2 FF      JSR  $FFD2
11     0353 60          RTS

```

Questo programma simula una routine di INPUT che termina con una virgola invece che con un RETURN. Per usare un carattere di termine diverso dalla virgola, basta cambiare l'operando della prima istruzione in modo tale che il valore corrispondente al carattere che si desidera sia immagazzinato alla locazione 900.

Esercizio 7.1

Modificare il programma 7.6 in modo tale che sia accettato un input che termini con uno spazio. Usare un comando POKE per ottenere questa variazione.

Come abbiamo detto in precedenza, uno dei maggiori problemi che si possono incontrare nell'uso di queste routine e' che esse fanno, di solito, un largo uso sia del 6510 che dei relativi registri o flags.

Per questo, dopo aver eseguito un'istruzione JSR non sara' ragionevole, al ritorno, pensare o supporre che non sia successo niente nei registri del 6510.

Per esemplificare quanto detto, osserviamo il seguente programma che e' stato messo a punto per eseguire un input di una stringa di 4 caratteri da tastiera.

Per prima cosa viene fissato un contatore di ciclo con valore 4 nel registro X. Sono usate successivamente le routines GETIN e CHROUT.

Di ritorno da queste routines e' decrementato X e controllato se il flag Z e' settato.

Programma 7.8

```
LOOP =    $033E
1          *          =    828
2    033C  A2  04          LDX  #4
3    033E  20  E4  FF  LOOP  JSR  $FFE4
4    0341  F0  FB          BEQ  LOOP
```


5	0343	20	D2	FF	JSR	\$FFD2
6	0346	CA			DEX	
7	0347	D0	F5		BNE	LOOP
8	0349	60			RTS	

Tuttavia quando questo programma gira, viene eseguito un RTS dopo che un solo carattere e' stato immesso.

Cio' suggerisce che una delle subroutines stia usando il registro X.

Per verificare cio' possiamo mettere il programma in modo che stampi il contenuto del registro X ai vari stadi di passaggio.

Questa operazione e' fatta nel programma seguente in cui il contenuto del registro X e' esaminato immediatamente dopo il ritorno da subroutine (RTS).

Programma 7.9

LOOP =	\$033E					
1					*	= 828
2	033C	A2	04			LDX #4
3	033E	20	E4	FF	LOOP	JSR \$FFE4
4	0341	F0	FB			BEQ LOOP
5	0343	8E	00	04		STX 1024
6	0346	A0	01			LDY #1
7	0348	8C	00	D8		STY 55296
8	034B	20	D2	FF		JSR \$FFD2
9	034E	8E	02	04		STX 1026
10	0351	A0	01			LDY #1
11	0353	8C	02	D8		STY 55298
12	0356	CA				DEX
13	0357	D0	E5			BNE LOOP
14	0359	60				RTS

Quando gira, questo programma si comporta come il precedente stampando il carattere in INPUT ma visualizza anche 2 A alle locazioni 1024 e 1026. Poiche' la prima A e' stampata immediatamente dopo l' uscita dalla routine di GETIN (JSR \$FFE4) e' chiaro che GETIN modifica il registro X. Come abbiamo appena scoperto CHROUT non esegue modifiche in questo caso.

Infatti poiche' il registro X viene messo a 1 da GETIN ecco spiegato il comportamento delle istruzioni DEX/BNE che ordinano di lasciare il programma dopo l' esecuzione del primo ciclo.

Per superare questo problema e' necessario che il valore del registro X sia immagazzinato da qualche parte prima di accedere alla subroutine e ripristinato prima che si passi a decrementarlo. Il programma 7.10 mostra questo processo in cui X e' temporaneamente immagazzinato nella locazione 900 durante l' esecuzione della subroutine.

Programma 7.10

```

SALVAX = $033E
GET = $0341
1          *          = 828
2  033C A2 04          LDX #4
3  033E 8E 84 03 SALVAX STX 900
4  0341 20 E4 FF GET   JSR $FFE4
5  0344 F0 FB          BEQ GET
6  0346 20 D2 FF      JSR $FFD2
7  0349 AE 84 03      LDX 900

```



```

8   034C CA           DEX
9   034D D0 EF       BNE SALVAX
10  034F 60           RTS

```

L' uso di questa tecnica se da una parte risolve il problema, dall' altra e' particolarmente laboriosa.

Per fortuna il 6510 e' in grado di eseguire automaticamente parte di queste operazioni.

LO STACK

Lo stack (S) e' un blocco di memoria, sul CBM64 localizzabile da 511 (\$01FF) INDIETRO fino a 256 (\$0100) che e' in grado di manipolare 255 Bytes.

E' usato per un trasferimento rapido dei dati che vengono immessi a partire dalla locazione 511 all' INDIETRO, mentre l' indirizzo della prima locazione libera viene immagazzinato nello STACK POINTER (SP).

Quando qualcosa deve essere ritrovato dallo STACK, solo l' ultima registrazione e' accessibile.

Di norma viene usata un' analogia con una catasta di piatti nella quale solo l' ultimo piatto della catasta e' accessibile.

Il termine tecnico per questo procedimento e' :

TECNICA LIFO

LIFO vuol dire LAST-IN FIRST-OUT, cioè che l'ultimo dato inserito è quello che esce per primo.

Una delle funzioni dello Stack è di ricordare gli indirizzi durante i salti a subroutines, cosa che viene fatta automaticamente.

Quando il 6510 trova un'istruzione come:

```
JSR 50000
```

deve per prima cosa tenere a mente dove è la prossima istruzione in modo tale che possa trovare il suo nuovo indirizzo dopo che la subroutine è stata eseguita e quindi piazza 50000 nel Program Counter.

La procedura è esaminata qui di seguito con un segmento di programma, il 7.11, proveniente dal programma 7.10

Programma 7.11

```
PASSO 1. 828 ($033C) STX 900 ($0384)
```

```
PASSO 2. 831 ($033F) JSR 65508 ($FFE4)
```

```
PASSO 3. 834 ($0342) BEQ 831 ($033F)
```

```
PASSO 1.
```

..... i) Calcola l' indirizzo della prossima
istruzione (831)

..... ii) Metti l' indirizzo nel PC

..... iii) Esegui l' istruzione STX 900

..... iv) Ricava l' indirizzo per la prossima
istruzione dal PC e che sara' \$033F.

PASSO 2.

..... v) Vai a cercare l' indirizzo della
prossima istruzione a \$033F.
L' istruzione da eseguire e' JSR \$FFEA

..... vi) Ripristina l' indirizzo per la
prossima istruzione in programma, \$0342, ed
immettilo nello Stack.

NOTA

L' immissione nello STACK sara' fatta a partire
da 511 in questo modo:

Indirizzo da immettere \$0342

..... 509
..... 510 \$03 (MSB)
..... 511 \$42 (LSB)

vii) Registra che la prima locazione disponibile nell' area Stack e' la 509. Lo Stack Pointer sara' quindi a 509.

viii) Carica \$FFE4 nel PC

ix) Esegui il salto a \$FFE4

x) Esegui la subroutine prima di trovare RTS

xi) Vai allo Stack Pointer per trovare l' ultimo dato. Poiche' SP e' = 509 i dati saranno a 510 e 511.

xii) Estrai i dati da 510 e 511 (\$0342) caricali nel PC, esegui il reset dello SP a 511

xiii) Salta a \$0342

PASSO 3.

xiv) Vai a cercare la prossima istruzione e prosegui con il programma.

Non e' molto semplice.

Fortunatamente le operazioni dello stack relative alla memorizzazione di indirizzi durante l' esecuzione di subroutines sono automatiche lasciando al programmatore il tempo ed il modo di occuparsi d' altro.

Tuttavia, come abbiamo visto a proposito delle routines di inserimento, lo Stack non puo'

immagazzinare automaticamente il contenuto di tutti i registri, ma deve essere programmato per questo.

Esistono solo due istruzioni per immagazzinare i dati dei registri, nessuna delle quali purtroppo per i due registri che dovranno quindi essere salvati tramite il passaggio attraverso l'Accumulatore.

PHA Push contents of Accumulator onto stack

Cioe' immetti il contenuto dell' Accumulatore nello Stack.

contenuto che potra' essere ricercato con l'istruzione:

PLA PuLl top of stack into Accumulator.

Usando queste istruzioni, il programma 7.8 puo' essere riscritto per trasferire il registro X entro lo Stack e ritrovarlo quando necessita.

Programma 7.12

SALVAX = \$033E

GET = \$0340

1			*	=	828
2	033C	A2	04	LIX	#4
3	033E	8A	SALVAX	TXA	
4	033F	48		PHA	

5	0340	20	E4	FF	GET	JSR	\$FFE4
6	0343	F0	FB			BEQ	GET
7	0345	20	D2	FF		JSR	\$FFD2
8	0348	68				PLA	
9	0349	AA				TAX	
10	034A	CA				DEX	
11	034B	D0	F1			BNE	SALVAX
12	034D	60				RTS	

Quando questo programma gira, accettera' quattro caratteri in INPUT e li visualizzera' sullo schermo.

Altre due istruzioni che consentono di salvare e ritrovare dati sullo Stack sono:

PHP Push Processor status register on stack

Per caricare lo SR nell' area di Stack, e:

PLP PuLl stack to Processor status register

Per rintracciare i dati.

Nel programma 7.12 lo SR non era stato salvato nell' area di Stack, perche' prima di controllare il flag Z con BNE, l' istruzione DEX lo aveva resettato.

Tuttavia in altre circostanze puo' rendersi necessario salvare il contenuto di SR.

Esercizio 7.2

Riscrivere il programma 7.12 in modo da salvare SR nello Stack prima della subroutine e ritrovarlo dopo l'esecuzione della stessa.

Quando si usa lo stack, la maggiore preoccupazione deve essere quella di controllare l'ordine di entrata e di uscita dei dati ricordandosi che il metodo di ricerca e salvataggio è il LIFO.

SCHEMA DI FUNZIONAMENTO

OPERAZ N. 1 SALVA A
" 2 SALVA SR
" 3 SALVA Y
" 4 SALVA X
" 5 RICARICA X
" 6 RICARICA Y
" 7 RICARICA SR
" 8 RICARICA A

CAPITOLO OTTAVO

INTERRUPTS, NUMERI E VARIABILI

Quando si esegue un lavoro, non si desidera essere interrotti fino a quando non sia stato terminato.

Anche il 6510 possiamo dire che si comporta allo stesso modo.

Durante l' esecuzione di un programma il microprocessore ha il controllo dell' Accumulatore, dei registri X e Y e tutti i flags sono appropriatamente settati.

Così all' atto dell' interruzione e' necessario che tutti i registri siano salvati, normalmente nell' area STACK, e dopo l' interruzione, ritrovati e ripristinati.

L' interruzione o INTERRUPT e' infatti solo una subroutine che interrompe il lavoro del 6510 quando si desidera.

Questo consente di affermare che l' interrupt e' generato FUORI dal sistema di funzionamento del 6510 sia da una periferica esterna che, per esempio, da tastiera.

La manipolazione dell' interrupt deve essere preparata da programma perche' in casi particolari non sono consentite interruzioni.

Se, per esempio, un' altra periferica sta inviando dati alla memoria, allora viene messa in funzione una procedura di HAND-SHAKING fra le due macchine.

Questa procedura, in termini semplici, si comporta in questo modo.

C'è uno scambio di messaggi lungo la linea di trasmissione, del tipo:

"Sono pronto per inviare dati. Sei pronto per ricevere?"

"Sì"

"Questi sono i dati..... Fine dati"

"Grazie, OK"

Se avviene un Interrupt, è probabile che i dati siano troncati e quindi senza nessun valore.

Durante questi periodi, cioè quando non devono essere effettuate interruzioni, il programma può bloccare molte interruzioni - non tutte!! - per consentire che un particolare processo sia completato.

L'istruzione che consente questo blocco delle interruzioni è:

SEI SET Interrupt disable flag.

per prevenire quindi le interruzioni.

Stranamente la prima azione che è necessario fare per ottenere un INTERRUPT è quella di settare il Flag I (interrupt disable) per mezzo dell'uso dell'istruzione SEI.

Per prevenire Interrupts di troppo, almeno per il

momento, dobbiamo effettuare un controllo per vedere se e' il NOSTRO interrupt, cioe' quello che si desidera o si produce con una nostra istruzione, dato che potrebbero esserci altri interrupts.

Quando ci saremo assicurati che l' interruzione occorsa e' la nostra, cioe' quella che desideravamo, allora possiamo eseguire un CLEAR sul Flag di Interrupt o Flag I (cioe' metterlo a 0) con l' istruzione:

CLI CLear Interrupt disable flag

Cioe' consenti gli interrupt

Se si pensa a quanto detto, capiremo che dobbiamo consentire agli interrupts di essere interrotti.

Non tutti gli interrupts possono essere bloccati settando il Flag I.

Molti casi possono aversi sia durante il lavoro del sistema sia per squilibri o perdite di tensione in rete che richiedono un' azione immediata.

Per consentire al 6510 di distinguere fra questi due tipi di condizioni esistono sull' integrato stesso due PINS uno per ogni tipo di interrupt.

Uno di questi e' il NMI o Non Maskable Input pin che non puo' essere bloccato.

L' altro e' il IRQ o Interrupt ReQuest pin che

puo' essere coperto dal Flag I.

Quando il 6510 riceve un segnale di Interrupt, completa sempre l'istruzione che sta eseguendo prima di fare qualsiasi altra cosa.

Nel caso di un IRQ interrupt dovrebbe, dopo l'ultima istruzione, controllare se e' a 0 il Flag I (CLEAR) e, nel caso non sia a zero continuare fino a quando il programma non esegua il Clear.

Successivamente, prima di entrare nella procedura di Interrupt, il contenuto del PC e' salvato nello Stack e successivamente viene salvato SR.

Questo salvataggio di registri e' in realta' una mezza misura in quanto, quasi sicuramente, si avranno modifiche ai registri X, Y ed all'Accumulatore.

Successivamente il Flag I e' messo a 1 per prevenire altri Interrupts ed allora l'apposito indirizzo della relativa routine di interrupt e' caricato nel PC.

Questo indirizzo viene trovato alle locazioni 65530 e 65531 (\$FFFA e \$FFFB) per NMI e a 65534 e 65535 (\$FFFE e \$FFFF) per l'IRQ.

Queste routines di interrupt devono terminare con:

RTI ReTurn from Interrupt

Trovando questa istruzione il 6510 esegue tre funzioni:

1) Ripristina ai valori precedenti l' interrupt

il SR.

2) Resetta il flag I con un CLI automatico.

3) Guarda nello stack per l'indirizzo di ritorno e lo ripristina come in un RTS automatico.

Come abbiamo detto in precedenza il 6510 esegue solo un mezzo lavoro per cui sia A che X e Y devono essere salvati nell'area Stack.

Ricordiamo che per X e Y devono essere prima trasferiti sull'Accumulatore, quindi immessi nello stack (con PHA) e quando si esce dalla routine di interrupt si richiamano i valori per Y e X nell'Accumulatore (con PLA) e quindi si rimetteranno nei rispettivi registri con TAX e TAY.

Il 6510 ha un'altra istruzione di interrupt:

BRK BReak

Quando il 6510 incontra questa istruzione per prima cosa resetta il PC indicizzandolo di una posizione (in modo tale che il PC punti al byte seguente l'istruzione BRK), immagazzina questo indirizzo nello Stack.

Setta poi il Flag di Break (B Flag) che è il bit 4 dell'SR ed immagazzina anche questo nello Stack.

Dopo di cio' il 6510 eseguirà un normale interrupt IRQ usando i vettori IRQ presenti (come abbiamo visto) in \$FFFE (LSB) e \$FFFF (MSB).

Successivamente la routine IRQ controllerà da cosa derivi questo interrupt, cioè se è un vero IRQ o una istruzione BRK.

Di norma se la routine di IRQ scopre che era un'istruzione BRK, si verrà riportati in ambito Basic, avremo un clear di schermo e apparirà il solito Ready.

Usando però il nostro programma Assembler otterremo un ingresso nel monitor.

Per controllare ciò eseguire il seguente programma :

Programma 8.1

```
LDA £ 90  
STA 1024  
LDA £ 1  
STA 55296  
BRK
```

Dopo aver stampato un quadri bianco in 1024 questo Programma salterà al Monitor.

Usando il nostro programma Assembler per disassemblare i codici di indirizzo \$FFFE, \$FFFF dovrete ormai essere capaci di trovare questi

indirizzi e di seguire quindi le operazioni di questa routine.

L' elemento essenziale e' l' istruzione JMP IA 790 che troverete in 65368 (\$FF58).

Prima di caricare il programma Assembler gli indirizzi 790 e 791 contenevano gli indirizzi della routine che torna al Basic con READY, mentre invece ora contengono gli indirizzi di entrata al monitor.

Quindi questi indirizzi ci sono stati messi dall' Assembler.

NUMERI CON SEGNO

In tutti gli esercizi matematici visti fino a questo momento, i numeri usati sono stati trattati come numeri positivi.

Perciò ogni processo aritmetico e' stato preso in considerazione come trattamento di insiemi o stringhe di 8 bit.

Tuttavia, nel caso che debba essere usato un intero negativo, uno di questi bit deve essere utilizzato per indicare che stiamo rappresentando un numero negativo.

Il bit più a sinistra nella struttura del Byte, cioè il bit 7, e' utilizzato per questo motivo.

Viene cioè messo a 0 se il numero che deve essere rappresentato e' positivo e a 1 se questo numero e' negativo.

Usando questo metodo potremo, con i restanti 7 bits, rappresentare valori compresi tra +127 e -128.

Uno dei problemi che si pongono con l'uso di questo metodo e' che, in teoria, si possono avere due rappresentazioni per lo 0, cioe' -0 e +0:

..... +0 = 00000000

..... -0 = 10000000

Per superare questo problema i numeri negativi sono rappresentati nella forma:

COMPLEMENTO A DUE

che potra' sembrare una forma strana ma che funziona. Vediamo come.

Prendiamo il numero 38 (decimale) che in binario e' 00100110.

Per convertirlo nella sua forma negativa complemeto a due e' necessario cambiare tutti gli 0 in 1 e viceversa. Cioe' nel suo complemento:

00100110 diviene 11011001

Questa forma si chiama anche COMPLEMENTO A 1.

Successivamente si aggiunge 1:

..... 11011001 +
..... 1

```
..... -----  
..... 11011010
```

Infatti:

-38 = 11011010

Per comprendere il significato di questo modo di lavorare vediamo alcuni esempi:

1) 38 - 38 che dovrebbe dare 0. Infatti:

```
..... -38 = 11011010  
..... +38 = 00100110  
..... -- -----  
..... 00 00000000
```

2) 43 - 38

```
..... -38 = 11011010  
..... +43 = 00101011  
..... -- -----  
..... 5 00000101
```

3) 24 - 38

```
..... +24 = 00011000  
..... -38 = 11011010  
..... -- -----
```


.... -14 ... 11110010

Nell' ultima risposta abbiamo un 1 nel bit 7 per cui siamo in presenza di un numero negativo in complemento a due. Per convertirlo, per prima cosa troviamo il complemento a 1:

..... 11110010 ... 00001101

Aggiungiamo 1:

```
..... 00001101
.....          1
..... -----
..... 00001110 ... = -14
```

GLI OVERFLOW

Nei Bytes che rappresentano numeri con segno, come abbiamo visto in precedenza, possiamo immagazzinare un valore non superiore a +127.

Perciò ogni tentativo di immettere un numero di dimensioni maggiori provocherà un riporto entro il bit 7 (CARRY) o, come si dice in questo caso un OVERFLOW.

Prendiamo la somma $100 + 30$

```
.... 100   = 01100100
....  30   = 00011110
.... ---   -----
.... 130   = 10000010
```

Ma per quanto abbiamo visto 10000010 è un numero negativo, perché nel bit 7 è presente un 1.

Effettuare quindi il complemento a 1 e poi aggiungere 1.

Il 6510 manipola questo tipo di situazione

eseguendo il MONITORING dell' Accumulatore e, quando siamo in presenza di un OVERFLOW, settando il Flag di Overflow o FLAG V.

Questo flag puo' essere controllato dalle seguenti istruzioni:

BVC Branch on overflow Clear

e

BVS Branch on overflow Set

BVC controlla il contenuto del Flag di overflow e se non e' settato (cioe' $V=0$) esegue un salto.

BVS le stesse operazioni di cui sopra solo che esegue il salto se $V=1$.

Quando e' necessario eseguire processi aritmetici in precisione multipla con interi con segno, il bit 7 deve essere trattato come un Carry interno. Allora se ci troveremo in presenza di un Overflow, questo dovra' essere trasferito nel Byte piu' significativo.

Il seguente programma illustra l' uso di BVC per il controllo di Overflow.

Programma 8.2

SOMMA =	\$033E				
1			*	=	828
2	033C	18		CLC	
3	033D	A9	01	LDA	#1
4	033F	69	01	SOMMA	ADC #1
5	0341	50	FC	BVC	SOMMA
6	0343	8D	00 04	STA	1024
7	0346	A2	01	LDX	#1
8	0348	8D	00 D8	STA	55296
9	034B	60		RTS	

Quando questo programma gira il contenuto dell' Accumulatore e' incrementato progressivamente fino a quando i 7 bits piu' a destra non sono riempiti con 1.

Al successivo incremento il settimo bit e' resettato a 0 e viene generato un Carry che entra nel settimo bit.

Cio' setta il bit di riporto ed arresta il salto, consentendo al programma di girare fino a RTS.

Allo stesso modo che accade con altri Flags esistono provvedimenti per il controllo del flag di Overflow che puo' essere messo a 0 (CLEAR) con l' istruzione:

CVL CLear the oVerflow flag

Tuttavia, diversamente dagli altri Flag, il flag di Overflow non puo' essere messo a 1 (cioe' settato).

VISUALIZZAZIONE DEI NUMERI

In tutti gli esempi numerici visti fino a questo momento, le uscite su schermo sono state su codice CBM 64.

Mentre e' possibile interpretare queste visualizzazioni ricorrendo ad una tavola, e' ovviamente necessario in programma visualizzare numeri come numeri in base 10.

La maggiore complicazione che questa procedura comporta sta nel fatto che mentre il codice di visualizzazione del CBM64 e' una effettiva rappresentazione in base 256, per cui per rappresentare numeri fra 0 e 255 e' necessario un solo Byte, quando si desidera visualizzare in base 10 sono necessari tre caratteri (o Bytes) per rappresentare lo stesso valore.

Il programma seguente serve per eseguire questo tipo di conversione.

Per prima cosa controlla se il numero e' maggiore di 200 (se il primo digit e' 2) o se e' minore di 200 ma maggiore di 100.

Esegue poi lo stesso controllo per le decine e le unita' ed usa lo stack per immagazzinare il resto del numero mentre aggiunge la costante di conversione (48) all' accumulatore per cambiare il valore binario nell' equivalente valore da visualizzare.

Nel seguente esempio il numero che deve essere visualizzato (152) e' caricato in accumulatore all' inizio del programma.

Programma 8.3

	LDX	£ 1	Carica il colore bianco in X
	LDA	£ 152	Immetti il numero
	CMP	£ 200	Confronta A con 200
	BCC	PIPPO	Salta se e' inferiore a 200
	SBC	£ 200	Rimuovi il digit piu' a sinistra
	PHA		Immagazzina il resto nello stack
	LDA	£ 50	Carica A con "2" per 2 x 100
	STA	1024	Visualizza A
	STX	55296	Carica il bianco in RAM
	PLA		Ritrova A dallo Stack
	JMP	TENS	Salta alla routine
PIPPO	CLC		Clear del Carry
	CMP	£ 100	Confronta A con 100
	BCC	TENS	Salta se inferiore a 100
	SBC	£ 100	Rimuovi il digit piu' a sinistra
	PHA		Immetti il resto nello Stack
	LDA	£ 49	Carica A con "1" per 1 x 100
	STA	1024	Stampa A sullo schermo
	STX	55296	Carica il colore bianco in RAM
	PLA		Ritrova A dallo stack
TENS	CLC		Clear del Carry
	LDY	£ 0	Fissa Y a 0
	CMP	£ 9	Confronta A con 9

LOOP	BCC TENS0	Salta se A minore di 9
	INY	Incrementa Y
	SBC £ 10	Sottrai 10 da A
	CMP £ 9	Confronta A con 9
	BCS LOOP	Salta se A e' piu' grande di 9
TENS0	PHA	Carica A nello Stack
	TYA	Trasferisci Y in A
	ADC £ 48	Aggiungi la costante di conversione ad A
	STA 1025	Visualizza A
	STX 55297	Carica il bianco nella RAM colore
	PLA	Ritrova A dallo stack
	ADC £ 48	Aggiungi la costante di conversione ad A
	STA 1026	Visualizza A
	STX 55298	Immetti il bianco nella RAM colore
	RTS	

Quando questo programma gira sara' visualizzato, come al solito in alto a sinistra dello schermo un 152 in bianco.

In linea generale questo programma puo' essere usato come subroutine di un programma piu' generale per visualizzare il contenuto dell' accumulatore.

NUMERI IN VIRGOLA MOBILE

Quando si lavora in Basic le costanti binarie in

virgola mobile hanno 10 digits di precisione e sono visualizzate in 9 digits.

I loro esponenti hanno un range da -38 a +37.

Ogni valore e' immagazzinato in 6 bytes consecutivi e, per facilitarne la manipolazione, esistono due "ACCUMULATORI" nelle locazioni di memoria da 97 a 102 (\$61 a \$66) e da 105 a 110 (\$69 a \$6E). Questi sono normalmente conosciuti come:

FAC Floating Point Accumulator

e

AFAC Alternative Floating Point Accumulator.

Per immagazzinare un numero fino a 10 digits, quando e' visualizzato in base 10, il FAC usa solo sei Bytes.

Quando un programma Basic gira, potrete osservare che numeri molto grandi o molto piccoli sono espressi in forma esponenziale o notazione scientifica.

Percio' 4079.013 puo' essere espresso come 0.4079013E+ 4 oppure 0.4049013 x 10 alla quarta.

0.0000417 puo' scriversi come 0.417E- 4 oppure come 0.417 x 10 alla -4.

Questo tipo di rappresentazione contiene due parti.

Prendendo il primo caso, la prima parte, 0.4079013, e' chiamata MANTISSA e la seconda, il +4 di 10 alla quarta, ESPONENTE.

Queste due parti sono immagazzinate in binario nel FAC nella seguente maniera:

a) La MANTISSA BINARIA e' immagazzinata nei quattro Bytes centrali di FAC e AFAC.

Il segno della mantissa e' immagazzinato nel sesto Byte in cui a "1" nel bit 1 corrisponde una mantissa negativa e a "0" nello stesso bit corrisponde una mantissa positiva.

b) L' ESPONENTE BINARIO e' immagazzinato nel primo Byte di FAC e AFAC.

Poiche' e' necessario disporre della possibilita' di immagazzinare sia esponenti negativi che positivi e' necessario eseguire il complemento a 128.

Per questo motivo un esponente di 20 sara' immesso come $128+20 = 148$, mentre un esponente negativo, come -20 , sara' $128-20 = 108$.

Quando carica un numero in virgola mobile il Basic normalizza la sua rappresentazione binaria e quindi il suo digit piu' a sinistra e' sempre 1.

Prendiamo per esempio il numero +1400 (\$0578), espresso in binario e':

0000 0101 0111 1000

In questa forma il numero binario ha un esponente di 2 ed un punto binario implicito (naturalmente un numero binario ha un punto binario invece di

un punto decimale ed e' conosciuto anche come RADIX) alla destra del digit meno significativo:

0000 0101 0111 1000.

IL COMANDO USR

Questo comando consente il trasferimento di dati tra il FAC ed un programma in codice macchina.

Per esempio, la linea $B=USR(Q)$ in un programma Basic consente al sistema di immettere il valore di Q entro il FAC.

Questo allora salta alla routine in codice macchina il cui indirizzo e' trovato in 785 (\$0311) come LSB e 786 (\$0312) come MSB.

Si presume che abbiate immesso una routine in codice macchina in memoria che inizi a quell'indirizzo e usi i valori di 785 e 786 per puntare alla routine.

Quando la vostra routine usa FAC utilizzerà il valore che era in Q alla chiamata di USR.

Quando il particolare processo aritmetico e' stato terminato (cioe' ne siamo usciti), la vostra routine dovrebbe lasciare la risposta in FAC e questo sarà immesso in B dal Basic.

Naturalmente si può usare la funzione USR con lo stesso metodo e sistema di altre funzioni.

Per esempio $PRINT USR (P+2)$ visualizzerà il risultato della routine in codice macchina che sarà stata iniziata con il FAC che conteneva il

valore immagazzinato in P aumentato di 2.

Per controllare questo processo, puo' essere messo un numero in FAC per mezzo della funzione USR e dopo visualizzato.

Lo faremo con i due programmi seguenti:

Programma 8.6a

```
20000 PRINT "clear di schermo"  
20010 POKE 785,60  
20020 POKE 786,3  
20030 B=1400  
20040 Q=USR(B)  
20050 PRINT"Q=";Q
```

NOTA

L' indirizzo \$033C e' ottenuto con :

```
60=$3C  
3=$03
```

Programma 8.6b

828 RTS

Al RUN, l' indirizzo \$033C (828) viene caricato

nelle locazioni 785 e 786 dalle linee 20010 e 20020.

Quando viene eseguita la linea 20040, l'argomento B (1400) e' caricato nel FAC. Quindi il controllo viene preso dal programma in codice macchina a \$033C.

La routine in codice macchina non puo' a questo punto modificare il valore di FAC, ma eseguendo RTS sara' restituito il controllo al Basic che a sua volta immettera' il contenuto di FAC in Q.

La linea 20050 stampa il valore immagazzinato in Q che non e' stato modificato dalla routine in codice macchina.

Questa routine offre un sistema di caricare un numero qualsiasi, che sia valido in Basic, entro il FAC.

Offre anche un sistema per esaminare il contenuto di FAC.

Questo non e' cosi' chiaro come si potrebbe credere, perche' molti comandi Basic usano FAC durante la loro esecuzione, cosi' anche un comando PEEK cambia il suo contenuto.

Tuttavia, se il contenuto e' esaminato in codice macchina, immediatamente dopo essere stato settato, puo' essere visto prima che il Basic ci rimetta le mani.

Per far questo il programma 8.6b dovrebbe essere modificato per esaminare le locazioni da \$61 a \$66 (da 97 a 102) per dopo visualizzarle. Cio' viene fatto nel programma 8.7.

Programma 8.7


```

PIFFO = $033E
1          *          =      828
2  033C A2 06          LDY   #6
3  033E A0 02          LDY   #2
4  0340 B5 60          LDA   96,X
5  0342 9D 90 05       STA   1424,X
6  0345 98             TYA
7  0346 9D 90 D9       STA   55696,X
8  0349 CA             DEX
9  034A D0 F2          BNE   PIFFO
10 034C 60            RTS

```

Stampa del contenuto di FAC sullo schermo.

Come il programma appena visto visualizza il contenuto in codice CBM, lo stesso programma potrebbe essere modificato per decodificare questo codice.

Programma 8.8

```

20000 PRINT "home"
20010 POKE 785,60
20020 POKE 786,3
20050 B=1400
20040 A=USR(B)
20050 PRINT"A=";A
20060 FORX=0TO5
20070 PRINT PEEK (1425+X);" ";
20080 NEXT X

```

Questo programma semplicemente guarda (PEEKs) le

locazioni che visualizzano il contenuto e stampano le risposte.

NOTA

Da osservare che il contenuto fra parentesi della linea 20000 sta ad indicare che deve essere premuto il tasto CLR/HOME, ma senza lo SHIFT,

Quando gira sarà visualizzato:

..... A = 1400

ed i contenuti:

... 139 175 0 0 0 47

SUBROUTINES PER VIRGOLA MOBILE

Vediamo ora cosa ci offre il sistema operativo della Commodore per manovrare con maggiore facilità queste complesse operazioni a sei Bytes.

Per usare queste routines prima di tutto è necessario sapere dove sono, cioè a quale indirizzo trovarle.

A oggi esse hanno avuto quattro indirizzi:

OLD ROM

BASIC 2.0

BASIC 4.0

BASIC V2

le prime due sono delle serie PET/CBM e l'ultima per il VIC-20.

Sono state rilocate ancora una volta per il CBM64.

Gli indirizzi riportati in questo libro sono per i modelli CBM64 attualmente sul mercato.

Tuttavia, dato le precedenti esperienze, nulla vieta che in futuro la Commodore, e sempre per il CBM64, decida di cambiare piu' o meno profondamente il Basic e rilocarle da altra parte, magari uguali e con le stesse funzioni, ma con indirizzi diversi.

In appendice e' riportata una tavola, che crediamo utilissima, che riporta l'indirizzo ed il contenuto di queste routines, le applicazioni, i registri interessati al loro uso, gli errori nei quali eventualmente possiamo incorrere durante l'esecuzione di esse oltre alle routines preparatorie e conseguenti.

Ancora qualcosa prima di addentrarci nella spiegazione di alcune di queste routines.

Si deve anche conoscere da dove queste routines raccolgono i loro dati e dove depositano poi il risultato se si desidera usarle con sicurezza.

Molte di loro iniziano con una piccola sezione di

inizializzazione che prepara i dati e li deposita nella giusta posizione per lavoro.

La subroutine che trasferisce i dati dalla memoria nell' AFAC, per esempio, incomincia con il trasferire i suoi indirizzi di data in 31 (\$1F) e 35 (\$23) dall' Accumulatore e dal registro Y. Per cui quando parte da \$BA8C attende di trovare gli indirizzi in A e Y.

A \$BA90 (47760) incomincia la routine vera e propria e allora ricava i suoi indirizzi dati da 34 (\$22) e 35 (\$23).

Allora puo' inserirsi facilmente con gli indirizzi in A e Y, oppure un po' di bytes dopo con i suoi indirizzi in \$22 e \$23.

Un interessante esercizio potrebbe essere quello di disassemblare questa routine, con l' opzione L del nostro Assemblatore e indirizzo di partenza 47756, e studiarne il funzionamento attraverso i vari passi.

Esercizio 8.1

Scrivere un programma per inserire i numeri 1.047 e 4038.22 in un programma in codice macchina.

Eseguire una moltiplicazione fra i due numeri ed effettuare la radice quadrata della somma.

Visualizzarne le risposte in Basic.

Non e' facile come sembra!!!

```
20000 PRINT " "  
20010 POKE 785,60
```



```
20020 POKE 786,3
20030 INPUT B
20040 A=USR (B)
```

```
..... Metti (B) in FAC
```

```
..... 828 JSR $BC0C
..... RTS
```

```
20050 POKE 785,72
20060 POKE 786,3
20070 INPUT D
20080 C=USR (D)
```

```
..... Metti (D) in FAC
```

```
..... 832 JSR $BA2B
..... 835 JSR $BF71
..... 838 RTS
```

```
20090 PRINT "C=";C
```

Questo era il piano, tuttavia non funziona.
Perche?

Non funziona perche' il resto del Programma ritiene che il contenuto di FAC resti fermo, mentre cambia continuamente mentre il programma

Basic gira.

Dopo la linea 20040 FAC contiene B e dopo JSR \$BC0C sia FAC che AFAC contengono B.

Tuttavia nell' esecuzione delle linee da 20050 a 20080 il Sistema Operativo utilizza FAC e AFAC e quindi ne cambia i contenuti. Per questo quando viene chiamata in funzione la routine JSR \$BA2B il contenuto di FAC e AFAC non e' quello che si aspettava.

Il problema puo' essere superato salvando AFAC in memoria mentre si ritorna al Basic.

Cio' puo' essere fatto per mezzo della subroutine di indirizzo \$BBC7.

Questa subroutine copia il contenuto di AFAC nei 5 bytes di memoria che iniziano all' indirizzo immagazzinato a \$49 e \$4A.

Il seguente programma lo illustra immettendo AFAC da \$0384 in poi.

Programma 8.9

```
LDA £132
STA $49
LDA £3
STA $4A
JSR $BBC7
RTS
```


COPIA DI AFAC IN MEMORIA

Questa azione puo' essere controllata facendo girare un programma diretto come il seguente:

```
FOR X=0 TO 5:PRINT PEEK(900+X); : NEXT X
```

Per usare questa subroutine nell' esercizio 8.1 e' necessario rilocare i dati in AFAC. Cio' puo' essere fatto usando la subroutine \$BA8C (47756).

Per operare la subroutine ha bisogno di sapere dove trovare i dati e cio' viene fatto caricando l' indirizzo del primo byte dati nell' accumulatore (LSB) e nel registro Y (MSB). Percio' un programma di "RELOAD AFAC" per i dati da 900 in poi potrebbe essere il seguente:

Programma 8.10

```
LDA# 132    Carica LSB dell' indirizzo
```

```
LDY  3      Carica MSB      "
```

```
JSR  $BA8C  Carica AFAC dalla memoria
```

```
RTS
```

A questo punto e' possibile effettuare l' esercizio 8.1 di cui una possibile soluzione e'

nel capitolo relativo alla soluzione degli esercizi.

ADDIZIONE

Vediamo, iniziando da questa, altre subroutines. Usando la routine di indirizzo \$B86A (47210) i numeri nel formato in virgola mobile (FLOATING POINT) in FAC ed AFAC sono sommati fra loro ed il risultato della somma caricato in FAC.

Facciamo un esempio con i seguenti programmi:

Programma 8.11

```
20000 PRINT "clear"  
20010 POKE 785,60  
20020 POKE 786,3  
20030 INPUT B  
20040 A=USR (B)  
20050 RUN 20060  
20060 POKE 785,72  
20070 POKE 786,3  
20080 INPUT D  
20090 C=USR(D)  
20100 PRINT "C=";C
```

Programma 8.12


```

828 LDA& 132
    STA 73
    LDA& 3      Immag. FAC in memoria
    STA 74
    JSR $BBC7
    RTS

840 LDA& 132    Ritrova i dati dalla memoria
    LDY 3
    JSR $BA8C    Immagazz. in AFAC

847 JSR $B86A    Rout di somma
    RTS

```

Il programma richiede l' input di due numeri e li restituisce addizionati.

SOTTRAZIONE

I programmi precedenti possono essere usati per dimostrare questa operazione inserendo la subroutine di indirizzo \$B853 (47187) a 848 e 849:

Programma 8.13 (parziale)

847 JSR \$B853

Anche questo dopo il RUN 20000 porra' due richieste di input. Le risposte saranno la sottrazione del secondo valore inserito dal primo.

DIVISIONE

Usiamo ancora una volta i programmi 8.11 e 8.12 per dimostrare l'uso della routine di divisione di indirizzo \$BB12 (47890) rimpiazzando 848 e 849 come sopra.

Programma 8.14

847 JSR \$BB12

eseguire RUN 20000.

Il programma eseguirà la divisione fra il primo dato inserito ed il secondo.

POTENZE

La routine di elevamento a potenza e' di indirizzo \$BF7B (49019). Utilizzare i programmi

precedenti rimpiazzando ancora una volta 848 e 849.

Programma 8.15

```
847 JSR $BF7B
```

Dopo il RUN 20000 il primo numero inserito sarà elevato alla potenza del secondo.

Altre routine necessitano di un solo input come:

LOG

La subroutine è a \$B9EA (47594) e calcola il logaritmo naturale o in base E. I programmi seguenti ne dimostrano l'uso.

Programma 8.16

```
828 JSR $B9EA  
831 RTS
```

Questo programma è richiamato dal seguente:

Programma 8.17

```
20000 PRINT"clear"  
20010 POKE 785,60  
20020 POKE 786,3  
20030 INPUT B  
20040 A=USR(B)  
20050 PRINT "A=";A
```

Con un RUN 20000 si attivano entrambe le routines che consentiranno di stampare il logaritmo in base E del numero inserito con l' input di 20030.

ESERCIZIO N. 1-1

1					*	=	828
2	033C	A9	01			LDA	#1
3	033E	8D	00	04		STA	1024
4	0341	A9	05			LDA	#5
5	0343	8D	00	D8		STA	55296
6	0346	60				RTS	

ESERCIZIO N. 1-2

1					*	=	828
2	033C	A9	06			LDA	#6
3	033E	8D	00	04		STA	1024
4	0341	A9	00			LDA	#0
5	0343	8D	00	D8		STA	55296
6	0346	A9	12			LDA	#18
7	0348	8D	01	04		STA	1025
8	034B	A9	00			LDA	#0
9	034D	8D	01	D8		STA	55297
10	0350	A9	05			LDA	#5
11	0352	8D	02	04		STA	1026
12	0355	A9	00			LDA	#0
13	0357	8D	02	D8		STA	55298
14	035A	A9	04			LDA	#4
15	035C	8D	03	04		STA	1027
16	035F	A9	00			LDA	#0
17	0361	8D	03	D8		STA	55299
18	0364	60				RTS	

ESERCIZIO N. 1-3

1					*	=	828
2	033C	A9	18			LDA	#24
3	033E	8D	00	04		STA	1024
4	0341	8D	27	04		STA	1063
5	0344	8D	C0	07		STA	1984
6	0347	8D	E7	07		STA	2023
7	034A	A9	0C			LDA	#12
8	034C	8D	00	D8		STA	55296
9	034F	8D	27	D8		STA	55335
10	0352	8D	C0	DB		STA	56256
11	0355	8D	E7	DB		STA	56295
12	0358	60				RTS	

ESERCIZIO N. 1-4

1					*	=	828
2	033C	A9	1A			LDA	#26
3	033E	A2	01			LDX	#1
4	0340	8E	84	03		STX	900
5	0343	AA				TAX	
6	0344	AD	84	03		LDA	900
7	0347	8D	00	04		STA	1024
8	034A	8E	E7	07		STX	2023
9	034D	8D	00	D8		STA	55296
10	0350	8D	E7	DB		STA	56295
11	0353	60				RTS	

ESERCIZIO N 1-5

1					*	=	828
2	033C	A9	5A			LDA	#90
3	033E	A2	2A			LDX	#42
4	0340	A0	05			LDY	#5
5	0342	8E	84	03		STX	900
6	0345	AA				TAX	
7	0346	98				TYA	
8	0347	AC	84	03		LDY	900
9	034A	8D	00	04		STA	1024
10	034D	8D	27	04		STA	1063
11	0350	8E	00	07		STX	1984
12	0353	8C	E7	07		STY	2023
13	0356	A9	01			LDA	#1
14	0358	8D	00	D8		STA	55296
15	035B	8D	27	D8		STA	55335
16	035E	8D	00	DB		STA	56256
17	0361	8D	E7	DB		STA	56295
18	0364	60				RTS	

ESERCIZIO N. 2-1

1					*	=	828
2	033C	A9	06			LDA	#3
3	033E	20	84	03		JSR	900
4	0341	8D	00	04		STA	1024
5	0344	A9	01			LDA	#1
6	0346	8D	00	D8		STA	55296
7	0349	60				RTS	

1					*	=	900
2	0384	8D	B6	03		ADC	#3
3	0386	60				RTS	

ESERCIZIO N. 2-2

ROUT =	\$033E						
FINE =	\$0344						
1					*	=	828
2	033C	A0	64			LDY	#100
3	033E	88			ROUT	DEY	
4	033F	F0	03			BEQ	FINE
5	0341	4C	3E	03		JMP	ROUT
6	0344	8C	00	04	FINE	STY	1024
7	0347	8C	00	D8		STY	55296
8	034A	60				RTS	

ESERCIZIO N. 2-3

INCREM =	\$0343						
PIFFO =	\$034C						
1					*	=	828
2	033C	A9	53			LDA	#83
3	033E	8D	7A	03		STA	890
4	0341	A0	00			LDY	#0
5	0343	C8			INCREM	INY	
6	0344	CC	7A	03		CPY	890
7	0347	F0	03			BEQ	PIFFO
8	0349	4C	43	03		JMP	INCREM
9	034C	8C	0A	04	PIFFO	STY	1034
10	034F	A9	04			LDA	#4
11	0351	8D	0A	D8		STA	55306
12	0354	60				RTS	

ESERCIZIO N. 2-5

DECRX =	\$033E						
1					*	=	828
2	033C	A2	5A			LDX	#90
3	033E	CA			DECRX	DEX	
4	033F	10	FA			BPL	DECRX
5	0341	8E	00	04		STX	1024
6	0344	8E	00	D8		STX	55296
7	0347	60				RTS	

ESERCIZIO N. 3-2

ROUT =	\$0343				*	=	828
1							
2	033C	A2	64			LDX	#100
3	033E	8E	84	03		STX	900
4	0341	A2	00			LDX	#0
5	0343	A9	2A		ROUT	LDA	#42
6	0345	9D	00	04		STA	1024,X
7	0348	A9	01			LDA	#1
8	034A	9D	00	03		STA	55296,X
9	034D	E8				INX	
10	034E	EC	84	03		CPX	900
11	0351	D0	F0			BNE	ROUT
12	0353	60				RTS	

ESERCIZIO N. 4-1

					*	=	828
1							
2	033C	18				CLC	
3	033D	D8				CLD	
4	033E	A9	07			LDA	#7
5	0340	69	FA			ADC	#\$FA
6	0342	8D	02	04		STA	1026
7	0345	A2	01			LDX	#1
8	0347	8E	02	0B		STX	55298
9	034A	A9	18			LDA	#\$18
10	034C	69	2A			ADC	#\$2A
11	034E	8D	00	04		STA	1024
12	0351	8E	00	0B		STX	55296
13	0354	60				RTS	

ESERCIZIO N. 4-2

1					*	=	828
2	033C	D8				CLD	
3	033D	38				SEC	
4	033E	A9	20			LDA	#32
5	0340	E9	58			SBC	#88
6	0342	8D	0C	04		STA	1036
7	0345	A2	01			LDX	#1
8	0347	8E	0C	D8		STX	55308
9	034A	A9	03			LDA	#3
10	034C	E9	02			SBC	#2
11	034E	8D	0A	04		STA	1034
12	0351	8E	0A	D8		STX	55306
13	0354	60				RTS	

ESERCIZIO N. 4-3

1					*	=	828
2	033C	D8				CLD	
3	033D	18				CLC	
4	033E	A9	2C			LDA	#\$2C
5	0340	69	90			ADC	#\$90
6	0342	8D	84	03		STA	900
7	0345	A9	01			LDA	#\$01
8	0347	69	01			ADC	#\$01
9	0349	8D	85	03		STA	901
10	034C	38				SEC	
11	034D	AD	84	03		LDA	900
12	0350	E9	F4			SBC	#\$F4
13	0352	8D	11	04		STA	1040
14	0355	A2	01			LDX	#1
15	0357	8E	11	D8		STX	55313
16	035A	AD	85	03		LDA	901
17	035D	E9	01			SBC	#\$01
18	035F	8D	10	04		STA	1041
19	0362	8E	10	D8		STX	55312
20	0365	60				RTS	

ESERCIZIO N. 4-7

PIFFO = \$034C

1				*	=	828
2	033C	A9	86		LDA	#134
3	033E	29	0F		AND	#15
4	0340	8D	01	04	STA	1025
5	0343	A2	01		LDX	#1
6	0345	8E	01	D8	STX	55297
7	0348	A0	04		LDY	#4
8	034A	A9	86		LDA	#134
9	034C	4A		PIFFO	LSR	A
10	034D	88			DEY	
11	034E	D0	FC		BNE	PIFFO
12	0350	8D	00	04	STA	1024
13	0353	8E	00	D8	STX	55296
14	0356	60			RTS	

ESERCIZIO N. 4-8

PLUTO = \$034B

PIFFO = \$0354

1				*	=	828	
2	033C	A2	03		LDX	#3	
3	033E	8E	85	03	STX	901	
4	0341	A2	04		LDX	#4	
5	0343	8E	86	03	STX	902	
6	0346	A0	08		LDY	#8	
7	0348	A9	00		LDA	#0	
8	034A	18			CLC		
9	034B	4E	86	03	PLUTO	LSR	902
10	034E	90	04		BCC	PIFFO	
11	0350	18			CLC		
12	0351	6D	85	03	ADC	901	
13	0354	0E	85	03	PIFFO	ASL	901
14	0357	88			DEY		
15	0358	D0	F1		BNE	PLUTO	
16	035A	8D	00	04	STA	1024	
17	035D	A2	01		LDX	#1	
18	035F	8E	00	D8	STX	55296	
19	0362	60			RTS		

ESERCIZIO N. 5-1

LOOP1 =	\$0343						
LOOP2 =	\$0352						
LOOP3 =	\$0363						
FINE =	\$0375						
1				*	=	828	
2	033C	A2	50		LDX	#80	
3	033E	A0	02		LDY	#2	
4	0340	4C	63	03	JMP	LOOP3	
5	0343	A9	53		LOOP1	LDA	#83
6	0345	9D	9F	04	STA	\$049F,X	
7	0348	98			TYA		
8	0349	9D	9F	D8	STA	\$D89F,X	
9	034C	CA			DEX		
10	034D	D0	F4		BNE	LOOP1	
11	034F	4C	75	03	JMP	FINE	
12	0352	A9	5A		LOOP2	LDA	#90
13	0354	9D	FF	03	STA	\$03FF,X	
14	0357	98			TYA		
15	0358	9D	FF	D7	STA	\$D7FF,X	
16	035B	CA			DEX		
17	035C	D0	F4		BNE	LOOP2	
18	035E	A2	78		LDX	#120	
19	0360	4C	43	03	JMP	LOOP1	
20	0363	A9	2A		LOOP3	LDA	#42
21	0365	9D	DF	05	STA	\$05DF,X	
22	0368	98			TYA		
23	0369	9D	DF	D9	STA	\$D9DF,X	
24	036C	CA			DEX		
25	036D	D0	F4		BNE	LOOP3	
26	036F	A2	A0		LDX	#160	
27	0371	88			DEY		
28	0372	4C	52	03	JMP	LOOP2	
29	0375	60			FINE	RTS	

ESERCIZIO N. 7-1

PIFFO = \$0341

1					*	=	828
2	033C	A2	20			LDX	#32
3	033E	8E	84	03		STX	900
4	0341	20	E4	FF	PIFFO	JSR	65508
5	0334	F0	FB			BEQ	PIFFO
6	0346	20	D2	FF		JSR	65490
7	0349	CD	84	03		CMP	900
8	034C	D0	F3			BNE	PIFFO
9	034E	A9	0D			LDA	#13
10	0350	20	D2	FF		JSR	65490
11	0353	60				RTS	

ESERCIZIO N. 7-2

PIFFO = \$033E

PLUTO = \$0341

1					*	=	828
2	033C	A2	04			LDX	#4
3	033E	8A			PIFFO	TXA	
4	033F	48				PHA	
5	0340	08				PHP	
6	0341	20	E4	FF	PLUTO	JSR	\$FFE4
7	0344	F0	FB			BEQ	PLUTO
8	0346	20	D2	FF		JSR	\$FFD2
9	0349	28				PLP	
10	034A	68				PLA	
11	034B	AA				TAX	
12	034C	CA				DEX	
13	034D	D0	F2			BNE	PIFFO
14	034F	60				RTS	

ESERCIZIO N. 8-1

```

20000 PRINT "(clear)"
20010 POKE 785,60
20020 POKE 786,3
20030 INPUT B
20040 A=USR(B)
20050 RUN 20060
20060 POKE 785,72
20070 POKE 786,3
20080 INPUT D
20090 C=USR (D)
20100 PRINT "C=";C

```

1				*	=	828
2	033C	A9	84		LDA	#132
3	033E	85	49		STA	73
4	0340	A9	03		LDA	#3
5	0342	85	4A		STA	74
6	0344	20	C7	BB	JSR	\$BBC7
7	0347	60			RTS	
8	0348	A9	84		LDA	#132
9	034A	A0	03		LDY	#3
10	034C	20	8C	BA	JSR	\$BA8C
11	034F	20	2B	BA	JSR	\$BA2B
12	0352	20	71	AF	JSR	\$BF71
13	0355	60			RTS	

LE ROUTINES KERNAL

NOME	INDIRIZZO		FUNZIONE
	HEX	DEC	
ACPTR	\$FFA5	65445	INGRESSO BYTE DALLA PORTA SERIALE
CHKIN	\$FFC6	65478	APRE UN CANALE PER INPUT
CHKOUT	\$FFC9	65481	APRE UN CANALE PER OUTPUT
CHRIN	\$FFCF	65487	INGRESSO DI UN CARATTERE DA UN CANALE
CHROUT	\$FFD2	65490	OUTPUT DI UN CARATTERE SU UN CANALE
CIOUT	\$FFA8	65448	OUTPUT DI UN BYTE ALLA PORTA SERIALE
CINT	\$FF81	65409	INIZIALIZZA L'EDITOR DI SCHERMO
CLALL	\$FFE7	65511	CHIUDI TUTTI I CANALI E I FILES
CLOSE	\$FFC3	65475	CHIUDI UN FILE LOGICO
CLRCHN	\$FFCC	65484	CHIUDI I CANALI IN I/O
GETIN	\$FFE4	65508	RICEVE UN CARATT. DALLA CODA DI TAST.
IOBASE	\$FFF3	65523	RIPORTA INDIRIZZO DI BASE PERIF. I/O
IOINIT	\$FF84	65412	INIZIALIZZA INPUT/OUTPUT
LISTEN	\$FFB1	65457	COMANDO AL BUS SERIALE PER LISTEN
LOAD	\$FFD5	65493	CARICA RAM DA PERIFERICA
MEMBOT	\$FFC9	65463	LEGGE/FISSA IL MINIMO DI MEMORIA
MEMTOP	\$FF99	65433	LEGGE/FISSA IL MASSIMO DI MEMORIA

Appendice 1

OPEN	\$FFC0	65472	APRE UN FILE LOGICO
PLOT	\$FFF0	65520	LEGGE/FISSA POSIZIONE DEL CURSORE
RAMTAS	\$FF87	65415	INIZ. RAM; DISPONE PER BUFFER NASTRO
RDTIM	\$FFDE	65502	LEGGE OROLOGIO IN TEMPO REALE
READST	\$FFB7	65463	LEGGE IN I/O LO STATO DELLA PAROLA
RESTOR	\$FF8A	65418	REINTEGRA ASSENZA DEI VETTORI DI I/O
SAVE	\$FFD8	65496	SALVA RAM SU PERIFERICA
SCNKEY	\$FF9F	65439	SCANSIONE DI TASTIERA
SCREEN	\$FFED	65517	RIPORTA ORGANIZZ. SCHERMO DI X, Y
SECOND	\$FF93	65427	INVIA L'INDIRIZO SECON. DOPO LISTEN
SETLFS	\$FFBA	65466	FISSA INDIR LOGICO E SECONDARIO
SETMSG	\$FF90	65420	CONTROLLO MESSAGGI KERNAL
SETNAM	\$FFBD	65469	FISSA IL NOME DEL FILE
SETTIM	\$FFDB	65499	FISSA L'OROLOGIO (R.T.CLOCK)
SETTMO	\$FFA2	65442	FISSA IL TIMEOUT (fuori tempo) SUL BUS.
STOP	\$FFE1	65505	SCANSIONE DEL TASTO DI STOP
TALK	\$FFB4	65460	COM. SUL BUS SER. PER TALK
TKSA	\$FF96	65430	INVIA IND. SEC. DOPO TALK
UDTIM	\$FFEA	65514	INCREMENTA OROLOGIO
UNLSN	\$FFAE	65454	COMANDO AL BUS SERIALE PER UNLISTEN
UNTLK	\$FFAB	65451	COMANDO AL BUS SERIALE PER UNTALK
VECTOR	\$FF8D	65421	LEGGE/FISSA GLI I/O VETTORIZZATI

Appendice 2

TAVOLE DELLE ISTRUZIONI DEL 6510

ADC

Mnem.	COD. OPER		N.B.	N.CY.	MODI INDIRIZZAMENTO
	DEC	HEX			
ADC & OPER	105	69	2	2	Immediato
ADC OPER	101	65	2	3	Pagina
ADC OPER,X	117	75	2	4	Pagina
ADC OPER	109	6D	3	4	Assoluto
ADC OPER,X	125	7D	3	4*	Assoluto,X
ADC OPER,Y	121	79	3	4*	Assoluto,Y
ADC (OPER,X)	97	61	2	6	(Indiretto,X)
ADC (OPER),Y	113	71	2	5*	(Indiretto),Y

* Aggiungere 1 ciclo se salta pagina

AND

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
AND & OPER	41	29	2	2	Immediato
AND OPER	37	25	2	3	Pagina
AND OPER,X	53	35	2	4	Pagina
AND OPER	45	2D	3	4	Assoluto
AND OPER,X	61	3D	3	4*	Assoluto,X
AND OPER,Y	57	39	3	4*	Assoluto,Y
AND (OPER,X)	33	21	2	6	(Indiretto,X)
AND (OPER),Y	49	31	2	5	(Indiretto),Y

* Aggiungere 1 ciclo se salta pagina

ASL

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
ASL A	10	0A	1	2	Accumulatore
ASL OPER	6	06	2	5	Pagina
ASL OPER,X	22	16	2	6	Pagina
ASL OPER	14	0E	3	6	Assoluto
ASL OPER,X	30	1E	3	7	Assoluto,X

BCC

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
BCC OPER	144	90	2	2*	Relativo

- * Aggiungere 1 ciclo se il salto e' nella stessa pagina
- * Aggiungere 2 cicli se il salto e' a pagina diversa

BCS

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
BCS OPER	176	B0	2	2*	Relativo

- * Aggiungere 1 ciclo se il salto e' nella stessa pagina
- * Aggiungere 2 cicli se il salto e' a pagina diversa

BEQ

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
BEQ OPER	240	F0	2	2*	Relativo

- * Aggiungere 1 ciclo se il salto e' nella stessa pagina
- * Aggiungere 2 cicli se il salto e' a pagina diversa

BIT

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
BIT OPER	36	24	2	3	Pagina
BIT OPER	44	2C	3	4	Assoluto

BMI

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
BMI OPER	48	30	2	2*	Relativo

* Aggiungere 1 ciclo se il salto e' nella stessa pagina

* Aggiungere 2 cicli se il salto e' a pagina diversa

BNE

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
BNE OPER	208	D0	2	2*	Relativo

* Aggiungere 1 ciclo se il salto e' nella stessa pagina

* Aggiungere 2 cicli se il salto e' a pagina diversa

BPL

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
BPL OPER	16	10	2	2*	Relativo

* Aggiungere 1 ciclo se il salto e' nella stessa pagina

* Aggiungere 2 cicli se il salto e' a pagina diversa

BRK

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
BRK	0	00	1	7*	Implicito

BVC

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
BVC OPER	80	50	2	2*	Relativo

* Aggiungere 1 ciclo se il salto e' nella stessa pagina

* Aggiungere 2 cicli se il salto e' a pagina diversa

BVS

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
BVS OPER	112	70	2	2*	Relativo

* Aggiungere 1 ciclo se il salto e' nella stessa pagina

* Aggiungere 2 cicli se il salto e' a pagina diversa

CLC

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
CLC	24	18	1	2	Implicito

CLD

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
CLD	216	D8	1	2	Implicito

CLI

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
CLI	88	58	1	2	Implicito

CLV

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
CLV	184	B8	1	2	Implicito

CMP

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
CMP & OPER	201	C9	2	2	Immediato
CMP OPER	197	C5	2	3	Pagina
CMP OPER,X	213	D5	2	4	Pagina
CMP OPER	205	CD	3	5	Assoluto
CMP OPER,X	221	DD	3	4*	Assoluto,X
CMP OPER,Y	217	D9	3	4*	Assoluto,Y
CMP (OPER,X)	193	C1	2	6	(Indiretto,X)
CMP (OPER),Y	209	D1	2	5*	(Indiretto),Y

* Aggiungere 1 ciclo se salta pagina

CPX

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
CPX & OPER	224	E0	2	2	Immediato
CPX OPER	228	E4	2	3	Pagina
CPX OPER	236	EC	3	4	Assoluto

CPY

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
CPY & OPER	192	C0	2	2	Immediato
CPY OPER	196	C4	2	3	Pagina
CPY OPER	204	CC	3	4	Assoluto

DEC

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
DEC OPER	198	C6	2	5	Immediato
DEC OPER,X	214	D6	2	6	Pagina
DEC OPER	206	CE	3	6	Assoluto
DEC OPER,X	222	DE	3	7	Assoluto,X

DEX

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
DEX	202	CA	1	2	Implicito

DEY

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
DEY	136	88	1	2	Implicito

EOR

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
EOR & OPER	73	49	2	2	Immediato
EOR OPER	69	45	2	3	Pagina
EOR OPER,X	85	55	2	4	Pagina
EOR OPER	77	4D	3	4	Assoluto
EOR OPER,X	93	5D	3	4*	Assoluto,X
EOR OPER,Y	89	59	3	4*	Assoluto,Y
EOR (OPER,X)	65	41	2	6	(Indiretto,X)
EOR (OPER),Y	81	51	2	5*	(Indiretto),Y

* Aggiungere 1 ciclo se salta pagina

INC

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
INC OPER	230	E6	2	5	Immediato
INC OPER,X	246	F6	2	6	Pagina
INC OPER	238	EE	3	6	Assoluto
INC OPER,X	254	FE	3	7	Assoluto,X

INX

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
INX	232	E8	1	2	Implicito

INY

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
INY	200	C8	1	2	Implicito

JMP

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
JMP OPER	76	4C	3	3	Assoluto
JMP (OPER)	108	6C	3	5	Indiretto

JSR

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
JSR OPER	32	20	3	6	Assoluto

LDA

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
LDA & OPER	169	A9	2	2	Immediato
LDA OPER	165	A5	2	3	Pagina
LDA OPER,X	181	B5	2	4	Pagina
LDA OPER	173	AD	3	4	Assoluto
LDA OPER,X	189	BD	3	4*	Assoluto,X
LDA OPER,Y	185	B9	3	4*	Assoluto,Y
LDA (OPER,X)	161	A1	2	6	(Indiretto,X)
LDA (OPER),Y	177	B1	2	5*	(Indiretto),Y

* Aggiungere 1 ciclo se salta pagina

LDX

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
LDX & OPER	162	A2	2	2	Immediato
LDX OPER	166	A6	2	3	Pagina
LDX OPER, Y	182	B6	2	4	Pagina
LDX OPER	174	AE	3	4	Assoluto
LDX OPER, Y	190	BE	3	4*	Assoluto, X

* Aggiungere 1 ciclo se salta pagina

LDY

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
LDY & OPER	160	A0	2	2	Immediato
LDY OPER	164	A4	2	3	Pagina
LDY OPER, X	180	B4	2	4	Pagina
LDY OPER	172	AC	3	4	Assoluto
LDY OPER, X	188	BC	3	4*	Assoluto, X

* Aggiungere 1 ciclo se salta pagina

LSR

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
LSR A	74	4A	1	2	Accumulatore
LSR OPER	70	46	2	5	Pagina
LSR OPER,X	86	56	2	6	Pagina
LSR OPER	78	4E	3	6	Assoluto
LSR OPER,X	94	5E	3	7	Assoluto,X

NOP

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
NOP	234	EA	1	2	Implicito

ORA

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
ORA & OPER	9	09	2	2	Immediato
ORA OPER	5	05	2	3	Pagina
ORA OPER,X	21	15	2	4	Pagina
ORA OPER	13	0D	3	4	Assoluto
ORA OPER,X	29	1D	3	4*	Assoluto,X
ORA OPER,Y	25	19	3	4*	Assoluto,Y
ORA (OPER,X)	1	01	2	6	(Indiretto,X)
ORA (OPER),Y	17	11	2	5*	(Indiretto),Y

* Aggiungere 1 ciclo se salta pagina

PHA

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
PHA	72	48	1	2	Implicito

PHP

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
PHP	8	08	1	2	Implicito

PLA

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
PLA	104	68	1	2	Implicito

PLP

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
PLP	40	28	1	2	Implicito

ROL

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
ROL A	42	2A	1	2	Accumulatore
ROL OPER	38	26	2	5	Pagina
ROL OPER,X	54	36	2	6	Pagina
ROL OPER	46	2E	3	6	Assoluto
ROL OPER,X	62	3E	3	7	Assoluto,X

ROR

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
ROR A	106	6A	1	2	Accumulatore
ROR OPER	102	66	2	5	Pagina
ROR OPER,X	118	76	2	6	Pagina
ROR OPER	110	6E	3	6	Assoluto
ROR OPER,X	126	7E	3	7	Assoluto,X

RTI

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
RTI	64	40	1	6	Implicito

RTS

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
RTS	96	60	1	6	Implicito

SBC

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
SBC & OPER	233	E9	2	2	Immediato
SBC OPER	229	E5	2	3	Pagina
SBC OPER,X	245	F5	2	4	Pagina
SBC OPER	237	ED	3	4	Assoluto
SBC OPER,X	253	FD	3	4*	Assoluto,X
SBC OPER,Y	249	F9	3	4*	Assoluto,Y
SBC (OPER,X)	225	E1	2	6	(Indiretto,X)
SBC (OPER),Y	241	F1	2	5*	(Indiretto),Y

* Aggiungere 1 ciclo se salta pagina

SEC

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
SEC	56	38	1	2	Implicito

SED

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
SED	248	F8	1	2	Implicito

SEI

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
SEI	120	78	1	2	Implicito

STA

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
STA OPER	133	85	2	3	Pagina
STA OPER,X	149	95	2	4	Pagina
STA OPER	141	8D	3	4	Assoluto
STA OPER,X	157	9D	3	4*	Assoluto,X
STA OPER,Y	153	99	3	4*	Assoluto,Y
STA (OPER,X)	129	81	2	6	(Indiretto,X)
STA (OPER),Y	145	91	2	5*	(Indiretto),Y

* Aggiungere 1 ciclo se salta pagina

STX

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
STX OPER	134	86	2	2	Pagina
STX OPER, Y	150	96	2	4	Pagina
STX OPER	142	8E	3	4	Assoluto

STY

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
STY OPER	132	84	2	2	Pagina
STY OPER, X	148	94	2	4	Pagina
STY OPER	140	8C	3	4	Assoluto

TAX

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
TAX	170	AA	1	1	Implicito

TAY

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
TAY	168	A8	1	2	Implicito

TSX

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
TSX	186	BA	1	2	Implicito

TXA

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
TXA	138	8A	1	2	Implicito

TXS

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
TXS	154	9A	1	2	Implicito

TYA

Mnem.	COD. DEC	OPER HEX	N.B.	N.CY.	MODI INDIRIZZAMENTO
TYA	152	98	1	2	Implicito

PROGRAMMA PER LA CONVERSIONE IN DATA

Il modo di utilizzo del sottoindicato programma e' esposto nelle pagine 122 e segg.

```
0 PRINT"␣PRIMO ED ULTIMO INDIRIZZO␣":PRI
NT"IN DECIMALE 0 4 DIGIT HEX CON '$'Q"
1 INPUTA$(1),A$(2):FORI=1TO2:IFLEFT$(A$(
I),1)<>"$"THENA(I)=VAL(A$(I)):GOTO3
2 FORJ=1TO4:K=ASC(MID$(A$(I),J+1,1))-48:
A(I)=A(I)+(K-7*(1+(K<=9)))*16^(4-J):NEXT
J
3 NEXT:S=A(1):F=A(2):FORI=0TO9:POKE631+I
,13:NEXT:PRINT"␣QQ"0:PRINT1:PRINT2:PRINT
4
4 PRINT"3S="S":F="F:PRINT"10S ="S":F ="F
:PRINT"RUN␣":POKE198,7:END
5 PRINT"␣QQ4X="X+1:PRINT1000+X;"DATA ";;
FORY=0TO15:Z=S+16*X+Y:IFZ>=FTHEN7
6 PRINTMID$(STR$(PEEK(Z)),2);",":NEXT:P
RINT"␣":PRINT"RUN␣":POKE198,3:END
7 PRINTMID$(STR$(PEEK(Z)),2):Y=15:NEXT:P
RINT"RUN8␣":POKE198,3:END
8 PRINT"␣Q":FORI=3TO9:PRINTI:NEXT:PRINT"
?";CHR$(34);POKE198,8
9 PRINTCHR$(147)"IL PROGRAMMA IN FORMA D
I DATA SALVALO";CHR$(34);"␣":END
20 FOR I=S TO F:READ A:POKE I,A:NEXT
30 PRINT"QQSYS";S;"PER RUN M/L PROGRAM.Q
"
90 NEW
```

```
"␣=(CLEAR)
"␣=(CURSOR DOWN)
"␣=(HOME)
"␣=(CURSOR LEFT)
```


IL RESTAURO

Per prima cosa si e' provveduto al restauro della copertina , della pagina 1 (fronte e retro) e della pagina finale (ultima di copertina). Le parti mancanti sono state ritagliate e copiate da altre pubblicazioni Commodore del periodo.

Vista la buona riuscita, si e' cominciato a fare lo stesso con le altre pagine, attingendo dalle pagine leggibili parole o singole lettere per ricostruire le pagine illeggibili.

Le modifiche pero' risultavano troppo evidenti e il risultato comunque mediocre. In piu' l' operazione richiedeva molto tempo: trovare parole o lettere e copiarle posizionandole in modo da non essere distinguibili da quelle presenti non era sempre facile. In piu' le pagine del libro non erano omogenee: alcune erano state ingrandite e quindi il font in queste pagine risultava piu' grande e non poteva essere utilizzato nelle pagine con font piu' piccolo (e viceversa).

Dopo aver "restaurato" una quarantina di pagine in questo modo, si e' deciso per una diversa strategia: estrarre tutte le lettere dalle pagine "buone" creando un font di immagini con cui ricomporre da zero tutte le pagine del libro, partendo da una pagina bianca.

Sono stati quindi estratti 4 font:

- quello principale
- due font diversi per le parti in assembler
- un font per i numeri di pagina

In piu' un mini-font con i caratteri semigrafici

per costruire le tavole relative alle istruzioni (quelle che stanno in fondo al libro).

Restavano fuori le figure, che sono state ritagliate e restaurate a parte.

Fatto questo, bisognava recuperare il testo completo e leggibile dell'intero volume e un programma che permettesse di creare file grafici da linea di comando: imageMagick.

Per il formato testo si è partiti da una versione ricavata via ocr trovata in rete.

Ovviamente il testo ricavato dalle scansioni era monco, spesso errato e privo di tutti gli esempi e le parti in assembly.

Quindi è stato integrato "decodificando" le pagine e mantenendo l'impaginazione del volume originale.

Una volta ottenuto il file di testo, è stato sviluppato un programma in grado di "pilotare" imageMagick, inserendo nel file di testo delle chiavi che il programma poteva interpretare per decidere come formattare il testo, quale font usare, se le righe andavano o meno "giustificate" e dove inserire le figure.

Insomma, una specie di word processor.

Quindi, partendo da un'immagine completamente bianca di 1024x1416 pixel, il programma ha ricreato le 254 pagine del libro.

Anche i nuovi capitoli sono stati riprodotti con lo stesso font originale del libro.

NOTA SUL .D64 ALLEGATO

Il .D64 contiene:

- 1) Tutti i programmi di esempio descritti nel volume, salvati in codice macchina o in Basic vanno caricati con il comando:

```
LOAD "NOMEPROGRAMMA", 8, 1
```

e mandati in esecuzione con SYS 828 o RUN (vedere le indicazioni nel testo).

- 2) Tutte le soluzioni degli esercizi proposti, salvate in codice macchina o in Basic, che vanno caricate con il comando:

```
LOAD "NOMEESERCIZIO", 8, 1
```

e mandate in esecuzione con SYS 828 o RUN.

- 3) Il programma per la conversione in data di programmi in codice macchina riportato a pag. 250.

ERRATA CORRIGE

Valido per tutto il testo:

- Rimossi tutti gli apostrofi errati (tipo: un' altro).
- Rimossi i numerosi errori di ortografia e/o di battitura.
- Tutte le virgole sono state legate alla parola che le precede.
- Tutte le intestazioni "Programma X.Y" sono state scritte in minuscolo.

*Pag. 18:

Il programma ha un errore. La linea 7 dovrebbe contenere STX 55298 e non STX 55296, come infatti viene correttamente indicato nella pagina successiva.

*Pag. 21:

Il programma 1.3 e' lo stesso di 1.2. Si elimina il titolo e si rinumerava 1.3a in 1.3.

*Pag. 26:

Rimossa linea con \$0341 solitario.

*Pag. 27:

Nella descrizione di JSR sostituita "SubRoutins" con "SubRoutine".

*Pag. 41:

Spostata la linea 11 del programma di pagina 40 a pagina 40. In questo modo si comincia la pagina 41 con un titolo.

*Pag. 48:

Corretta la spiegazione della linea 6 in "Scrivi il colore BIANCO a (55296+X) al posto di "552956 Si assicura che il colore sia BIANCO".

Cambiato "Branch" in "Salta" (per omogeneita' col resto del testo).

Corretta linea 6 programma 3.1: il valore 55296 deve essere 55295, altrimenti non si colora il primo dei quadri.

*Pag. 51:

Il programma e' errato: la linea 4 invece che sul video finisce per sovrascrivere il programma stesso e la linea 6 pasticcia la zona riservata al SID.

*Pag. 52:

L'istruzione STAX ovviamente non esiste. Si e' modificato il riferimento con STA(OPER,X) che richiede appunto 6 cicli.

*Pag. 57:

Rimossa la riga con "START ADDRESS?" prima del programma.

*Pag. 59:

Il programma e' errato: l'indice dell'istruzione alla righe 12 e 22 deve essere Y e non X.

*Pag. 69:

Il programma 3.8 e' un rebus: l'istruzione LDA (84),X alla riga 3 non esiste. Secondo l'esadecimale utilizzato dovrebbe essere LDA (84,X) ma anche cosi' non farebbe quello che e' indicato nel testo precedente. In piu' la riga 5 non serve assolutamente a nulla. Quindi modifichiamo il programma in modo che esegua quanto scritto.

*Pag. 73:

Corretto errore: se un byte consente di contare da 0 a 255, allora due bytes consentono di contare fino a 65535 (e non 65536).

*Pag. 76:

Corretto errore: il valore massimo di un byte e' 255, non 256.

*Pag. 80:

Il programma 4.3 dovrebbe avere solo valori in notazione decimale, in quanto a pagina 83 viene riportato lo stesso programma (identico) dopo aver parlato della notazione esadecimale.

*Pag. 81:

Poiche' la tabella mostra l' effetto dell' istruzione, la riga relativa all' istruzione ADC£ 133 dovrebbe avere 1 nella colonna del Carry (non 0). Come infatti viene indicato nel testo successivo.

Rinumerato in 4.3a il programma 4.4 (il 4.4 e' a pagina 84).

*Pag. 82:

Rinumerato in 4.3b il programma 4.3a (per mantenere la logica della numerazione).

Rimosse le inutili righe contenenti CLC e CLD in fondo alla pagina. Sono le prime righe del programma 4.3b della pagina 83.

*Pag. 92:

Rimossa inutile riga contenente RTS in cima alla pagina. E' l' istruzione dell' ultima riga del programma 4.7 della pagina precedente.

Corretto NYBBLE/S in NIBBLE/S (per omogeneita' con pag. 91).

*Pag. 93:

Corretto NYBBLE in NIBBLES (per omogeneita' con pag. 92).

*Pag. 97:

I programmi 4.9 e 4.9a sono identici e usano entrambi l' indirizzamento immediato. La differenza e' che gli operandi di LDX e LDA sono espressi in notazione binaria. Si corregge la descrizione testuale.

*Pag. 105:

La spiegazione di ASL (Arithmetic Shift Left) e' errata: i bit vengono spostati a SINISTRA e non a destra.

*Pag. 108:

La somma progressiva dell' esempio di 7x5 e' errata. Dopo il prodotto parziale 3 abbiamo gia' il risultato 100011 che rimane tale dopo il prodotto parziale 4 (che e' composto di soli 0).

In piu' si fa riferimento a uno schema a blocchi della "MULTIPLICAZIONE A 8 BIT" di cui non c'e' traccia (recuperato e inserito).

*Pag. 109:

La linea 8 del programma 4.15 e' errata.

L' istruzione ASL va applicata all' indirizzo 902, non all' accumulatore.

*Pag. 115:

Il quoziente del risultato del Programma 4.15a e' una 0, non uno 0.

*Pag. 117:

Il programma 5.2 e' errato: l'istruzione alla linea 8 fa si che il secondo loop non venga mai eseguito. Modificato il programma.

*Pag. 118:

C'e' una riga "o realmente disponibili." di troppo. Rimossa.

*Pag. 135:

"Si puo' specificare un linguaggio di inizio".
Non "linguaggio" ma "indirizzo".

*Pag. 156:

"l' opzione" e' ripetuto due volte. Rimosso.

*Pag. 165:

Corretto "usare una kernal" con "usare una routine kernal".

*Pag. 169:

Spostata label LOOP da linea 2 a linea 3 per evitare un loop infinito.

*Pag. 170:

Aggiunto spazio dopo "(LOOP)" e spostato "la" alla linea successiva per mantenere la lunghezza della linea.

Relativamente a GETIN, se la coda di tastiera e' vuota (non piena, come scritto) la procedura ritorna 0 e non attende.

*Pag. 178:

Il passo 3 del programma 7.11 e' errato: il branch va fatto a 831 (non 251).

Rimossa inutile linea "ISTRUZIONE STX 900" in fondo alla pagina, visto che per gli altri due passi si indica solo "Passo n"

*Pag. 184:

Lo "SCHEMA DI FUNZIONAMENTO" dello stack e' errato: mostra una sequenza FIFO anziche' LIFO. Corretto.

*Pag. 195:

Spostato fine esempio nella pagina precedente, mezza vuota, in modo da iniziare il capitolo "GLI OVERFLOW" a inizio pagina.

*Pag. 197:

Spostato titolo "VISUALIZZAZIONE DEI NUMERI" nella pagina successiva.

*Pag. 199:

Corretta indentazione commenti per rendere il programma leggibile.

*Pag. 200:

Essendo una Label, rimosso spazio in "TENS 0". Corretto ultimo STX 55296 in STX 55298.

*Pag. 201:

Inserito spazio in "a+37".

*Pag. 202:

Nel paragrafo b), quando si parla di "complemento a 128", corretto valore 120 in 128 (che 120 e' errato lo si deduce anche dal risultato).

*Pag. 205:

L' indirizzo dell' istruzione STA in linea 5 del programma 8.7 e' errato. Dovrebbe essere 1024, visto che l' istruzione STA alla riga 6 ha lo scopo di colorare di rosso la locazione schermo scritta sopra. Oppure va modificato l' indirizzo 55296 in 55696. Corretto in questo secondo modo. Inoltre manca l'RTS per ritornare al Basic. Aggiunto.

*Pag. 213:

Il programmi 8.11 e 8.12 contengono diversi errori che mandano il C64 in crash. Corretti.

*Pag. 215:

I programmi 8.13 e 8.14 riportano un errato indirizzo. Quello corretto e' nel testo.

*Pag. 216:

Il programma 8.15 riporta un errato indirizzo. Quello corretto e' nel testo.

*Pag. 218:

L'esercizio 1-1 richiede di stampare IN VERDE. La soluzione proposta era incompleta. Corretta.

*Pag. 221:

La soluzione 2-4 si riferisce all' esercizio 2.5 di pag.45. L'esercizio 2.4 non esiste.

*Pag. 223:

L' esercizio 4.3 richiede la memorizzazione del risultato LSB/MSB a partire dalla locazione 1040. La soluzione proposta memorizza invece MSB/LSB. Corretta.

*Pag. 225:

La soluzione dell' esercizio 5.1, oltre ad essere illeggibile, e' completamente errata. A pag.118 viene chiesto di aggiungere un ciclo LOOP3 al programma 5.2 di pag.117. Tale programma usa correttamente le istruzioni STA con indice X (hex 9D). La soluzione usa invece, erroneamente, l' istruzione STA non indicizzata (hex 8D), rendendo inutili i cicli. Alla riga 5 l' indirizzo di partenza dovrebbe essere 1184 (anziche' 1183) altrimenti l' ultimo carattere viene stampato in fondo alla riga precedente. Si e' preferito spostare le due linee di asterischi a partire dalla locazione schermo 1503 (colore da 55775) per rendere piu' evidente l' effetto del codice aggiunto.

*Pag. 226:

La soluzione proposta e' errata poiche' nei cicli successivi al primo non torna a salvare i valori di stack ma comunque li ripristina. La soluzione corretta prevede due label.

*Pag. 227:

La soluzione 8-1 presenta errori sia nel Basic che nell' assembly: nella linea 20060 il valore da inserire in 785 e' 72 e non 70.

Nell' assembly sono errati tutti gli indirizzi di JSR.

*Pag. 242:

L'istruzione LSR A occupa un byte e non due come indicato. Inoltre la colonna dei cicli e' completamente errata. Corretto.

*Pag. 244:

E' una copia della pagina 242.

*Pag. 245:

E' una copia della pagina 243.

Le pagine 244-245 sono una copia delle pagine 242-243. Questo fa si che manchino le pagine contententi le tavole delle istruzioni: ROL, ROR, RTI, RTS, SBC, SEC. Ricostruite le pagine mancanti.

*Pag. 250:

Il programma termina con "syntax error". In effetti contiene diversi errori. Corretto in modo che funzioni come descritto.

INDICE

Introduzione	1
CAPITOLO PRIMO	3
Il linguaggio macchina	3
L' Accumulatore	5
Il programma Assembler	6
I registri indice	15
CAPITOLO SECONDO	25
I salti ed il PC	25
Salti condizionati	26
Il Program Counter	29
Istruzioni di confronto	36
I Flags	41
CAPITOLO TERZO	47
La temporizzazione	52
I modi di indirizzamento	60
Indirizzamento implicito	61
Indirizzamento assoluto	62
Indirizzamento in pag. Zero	62
Indirizzamento immediato	64
Indirizzamento relativo	67
Indirizzamento indiretto	68
Indirizzamento indiretto assol.	71
CAPITOLO QUARTO	73

Operazioni in doppia precis.	73
Input in esadecimale	82
La divisione	88
Codice decimale binario	90
AND e OR	92
ORA e EOR	98
Altre forme di manipolazione	102
Moltiplicazione binaria	106
Moltiplicazione a 8 bit	108
CAPITOLO QUINTO	116
Le Labels	116
Memory labels	118
Altre funzioni	120
Conversione in DATA	123
Il Monitor	125
CAPITOLO SESTO	155
Comando POKE	156
I colori	158
CAPITOLO SETTIMO	163
Le Routines	169
I programmi	172
Lo Stack	177
Tecnica LIFO	178
Metodi di programmazione	181
Schema di funzionamento	184
CAPITOLO OTTAVO	185
Gli interrupts	186

Numeri con segno	191
Gli overflow	195
Visualizzazione dei numeri	198
Numeri in virgola mobile	200
Il comando USR	205
Subroutines in virgola mob.	207
Le operazioni decimali	213
SOLUZIONE DEGLI ESERCIZI	218
LE ROUTINES KERNAL	228
TAVOLE DELLE ISTRUZIONI	230
IL RESTAURO	251
ERRATA CORRIGE	254

Commodore Italiana SpA

Via F.lli Gracchi, 48 - 20092 Cinisello Balsamo (Milano)