HOME E PERSONAL COMPUTER

David Lawrence - Mark England

Linguaggio macchina per il C16

GRUPPO EDITORIALE JACKSON
DIVISIONE LIBRI

EDIZIONE ITALIANA

Linguaggio macchina per il C16

David Lawrence - Mark England

GRUPPO
EDITORIALE
JACKSON
Via Rosellini, 12
20124 Milano

TITOLO ORIGINALE: Beginning Machine Code on C16
TRADUZIONE: Enrico Gavinelli
SUPERVISIONE TECNICA: Angelo Cattaneo
GRAFICA E IMPAGINAZIONE: Francesca di Fiore
COPERTINA: Silvana Corbelli
FOTOCOMPOSIZIONE: Lineacomp S.r.l. - Via Rosellini, 12 - 20124 Milano
STAMPA: A. Matarelli S.p.A. - Stabilimento Grafico - Milano.

SOMMARIO

INTRODUZIONE

Questo libro non serve agli esperti in linguaggio macchina; essi, sfogliandolo, sorriderebbero certamente per la semplicità di qualche esempio, anche se forse potrebbe insegnare qualcosa anche a loro.

Il volume è rivolto a chi non sa il linguaggio macchina, ma desidera impararlo.

Abbiamo notato che molte persone indugiano attorno al linguaggio macchina, desiderose di iniziare, ma incapaci di arrivare in qualche modo alla comprensione dei termini e, più specificamente, del gergo. Spesso si ha la sensazione che alcuni testi entrino immediatamente nell'impenetrabile linguaggio del programmatore in codice macchina. Sembra quasi che le persone che scrivono programmi composti principalmente di numeri trovino non poche difficoltà quando è necessario comunicare con le parole.

In realtà, il linguaggio macchina è estremamente semplice. Esso consiste in un numero relativamente piccolo di istruzioni, che possono essere eseguite impiegando pochi numeri. Con questo libro dimostriamo, o cerchiamo di farlo nel miglior modo possibile, quanto sia facile apprendere il linguaggio macchina.

Pensiamo che quest'opera valga lo sforzo da noi sostenuto, infatti siamo convinti che se voi seguirete attentamente il libro da cima a fondo potrete capire il linguaggio macchina ed il comportamento del chip 7501 che gestisce il C16/Plus 4. Non pretendiamo certo di insegnarvi a scrivere programmi in linguaggio macchina, ma siamo sicuri che, giunti al termine del volume, non ne sarete più timorosi e ciò, siatene certi, è un gran passo avanti.

CAPITOLO 1

GUIDA AL 7501

Qualsiasi operazione in codice macchina viene eseguita dentro minuscole scatoline nere con 40 piedini, chiamate chip. Poichè non si può vedere cosa sta avvenendo dentro di essi, e poichè i risultati della loro attività possono essere eccessivamente complessi, si è portati automaticamente a credere che debba essere quasi impossibile comprenderli e molto difficile dargli istruzioni sotto forma di programmi.

Nella costituzione di un chip è necessario tener conto di ogni tipo di considerazioni tecniche. Le funzioni automatiche, delle quali l'utente non è mai consapevole, possono occupare mesi di ideazione e correzione. I componenti microscopici devono essere sincronizzati con tutti gli altri mentre operano ad incredibili velocità.

Ma tutto questo poco interessa chi si interessa del codice macchina, che è il linguaggio con il quale un chip può essere programmato. Come non è necessario essere capaci di ideare un'auto affinchè la si guidi, così non è necessario essere in grado di ideare un chip affinchè si sia in grado di programmarlo. Inoltre, come guidare un'auto è facile una volta che si prende l'abitudine e si comprende quanti elementi agiscano assieme, così programmare un chip è un mezzo attraverso cui sentire come un ristretto numero di azioni operino assieme a produrre il risultato voluto.

Se siete indotti a temere la programmazione in codice macchina, probabilmente vi può essere d'aiuto ricordare quanto siano limitate le azioni che il chip 7501, che è il cuore del Commodore 16, può attuare.Ecco l'impressionante lista:

1) Ha tre aree di immagazzinamento nelle quali può conservare numeri nella gamma da 0 a 255.
2) Ha sette aree di immagazzinamento in cui può conservare il risultato di semplici decisioni sì/no.
3) Può prendere valori dalla memoria e metterli in una delle tre aree di cui al punto 1.
4) Può sommare due numeri.
5) Può moltiplicare o dividere per due.
6) Può eseguire 56 semplici istruzioni per effettuare combinazioni di quanto detto sopra.

Essenzialmente è tutto. Naturalmente, queste poche funzioni sono abbinate in vari modi. Ed è ugualmente ovvio che, per ottenere qualcosa di utile durante la programmazione del chip, devono essere scritti dei listati di istruzioni relativamente lunghi. Ciò non può alterare il fatto che le basi sono

veramente semplici, persino banali,e non c'è ragione di essere intimoriti da esse.

Per impadronirsi del codice macchina, tutto quello che serve è tenere a mente le 6 funzioni sopra descritte ed iniziare col capire:

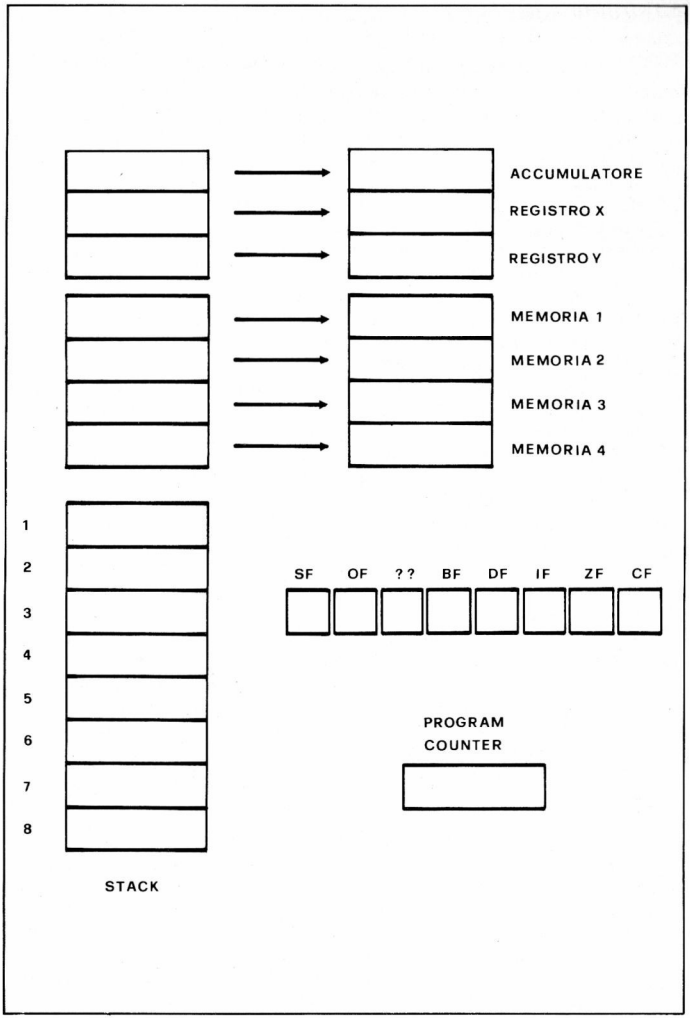
1) Come muovere numeri dentro e fuori le tre aree di immagazzinamento dentro il chip.
2) Come usare le sette decisioni si/no per controllare cosa sta facendo il chip.
3) Come individuare le aree di memoria dove collocare o trovare numeri.
4) I vari modi in cui si può usare la addizione ivi incluso l'uso dell'addizione per effettuare sottrazioni (oh certo,si può!).
5) Come moltiplicare o dividere semplicemente spostando le cifre dei numeri di posto in posto.
6) I nomi delle 56 istruzioni e come istruire il chip ad eseguirle nell'ordine voluto.

Il chip Lawland 10000

Per aiutarvi vogliamo iniziare questo testo non con listati in codice macchina, ma con la descrizione di un semplice chip che ha delle somiglianze con il 7501, sul quale il vostro C16 è basato. Fortunatamente, comunque, il chip che ci apprestiamo a descrivere è certo più semplice del 7501 e, per rendere le cose più facili, invece di lavorare con numeri binari, che esamineremo più tardi, esso lavora con i buoni vecchi numeri decimali, come tutti noi. Il nome di questo meraviglioso chip è Lawland 10000, ideato da due giganti della computerizzazione non sconosciuti agli autori di questo testo. Benchè non ci siano ancora esemplari funzionanti del chip, esso può essere descritto nei dettagli e possiamo iniziare a conoscere alcuni dei modi con cui il chip può operare.

A voi il diagramma funzionale delle architetture interne del chip Lawland 10000: (vedi figura).

Non preoccupatevi se, a prima vista, non riconoscete tutte le parti.In questo capitolo passeremo attraverso le principali funzioni del chip ed il disegno diventerà un pò più chiaro per tutti. Per seguire il capitolo con efficacia, fate delle copie del diagramma, sulle quali segnare a matita i valori che si muovono e si manipolano. Il diagramma è pensato in modo tale che nelle più importanti aree di immagazzinamento, possiate scrivere i valori e poi, nella seconda serie di spazi, scrivere i risultati di qualsiasi operazione che state eseguendo.


ACCUMULATORE
REGISTRO X
REGISTRO Y
MEMORIA 1
MEMORIA 2
MEMORIA 3
MEMORIA 4
1
2
3
4
5
6
7
8
STACK
SF
OF
??
BF
DF
IF
ZF
CF
PROGRAM
COUNTER

La struttura del chip

Il nostro chip immaginario, come potete vedere, ha una struttura molto semplice, anche se, tuttavia, abbiamo incluso alcuni blocchi che non sono esattamente parte del chip ma parti del computer del quale il chip è il cuore.

a) I registri. Questi sono locazioni di memoria contenute nel chip, ognuna delle quali può contenere un singolo numero compreso tra 0 e 9999, in altre parole ogni numero composto da 4 cifre o meno. Ogni operazione che il chip esegue, la fa sul numero inserito in quel registro detto "accumulatore". Affinchè si possano impiegare i vari registri, il chip accetta istruzioni per trasferire numeri direttamente tra i registri, anche se per trasferire un numero dal registro X al registro Y, questo dovrà essere mosso attraverso l'accumulatore.

b) Memoria. La maggior parte dei numeri con i quali opererà non sono immagazzinati nel chip, ma in altre aree del computer, del quale è una parte.Queste aree sono chiamate memoria perchè, mentre il computer è acceso, qualsiasi dato situato dentro esse sarà "ricordato" e potrà essere richiamato. La memoria è composta da locazioni numerate, ognuna capace di immagazzinare un numero da 0 a 9999, come i registri del chip. Poichè la memoria non fà parte del chip, esso trova molta difficoltà a manipolare i numeri immagazzinati nelle locazioni di memoria, e quindi per eseguire qualche cosa con o ad un numero immesso nella memoria, il chip copierà quel numero in un registro. La maggior parte del lavoro del chip consisterà nel prendere numeri dalla memoria, fare qualcosa con essi e poi riporli nuovamente in memoria. Il nostro chip immaginario è dotato di quattro locazioni di memoria da usare.

c) Flag. In precedenza abbiamo detto che il chip 7501 è fornito di 7 locazioni usate per immagazzinare semplici risposte si/no su operazioni in corso nel chip, in realtà esse sono otto, anche se una di esse non viene usata. Queste sono chiamate "flag" e i flag del Lawland 10000 sono rappresentati dagli otto riquadri sulla destra del diagramma.

d) Program counter. Più tardi vedremo come sia in grado di accettare istruzioni da un programma riposto in memoria. Il programma verrà presentato sottoforma di numeri, che il chip identificherà di conseguenza. Il compito del program counter è ricordare al chip dove sarà collocata la prossima istruzione.

e) Stack. Ogni scrivania necessita di un blocco per appunti, il luogo dove ssono essere registrati e poi scartati messaggi,fatti veloci calcoli o annotati dei fatti prima che essi siano dimenticati. Per il Lawland 10000 questo blocco per appunti è conosciuto come stack. Infatti lo stack è una parte di memoria, usata in un modo speciale, più precisamente come i fermacarte da scrivania. Mentre sta esaminando lo stack, il chip può solamente vedere cosa c'è sopra il blocco degli appunti. Se le notizie sono immagazzinate sullo stack nell'ordine 1, 2, 3, solo il 3 sarà visibile alla

fine del processo e per riavere indietro le informazioni il chip dovrà richiederle nell'ordine 3, 2, 1. Poichè lo stack è una parte di memoria, tutto quello che puòessere registrato in esso è un numero da 0 a 10000.

Programmazione del Lawland 10000

Ora che avete una certa infarinatura delle strutture fondamentali del chip, proseguiamo. Il resto di questo capitolo è dedicato a una serie di semplici esercizi di programmazione del nostro immaginario chip, ed ai tipi di istruzione che esso può comprendere. Se avete già ricopiato il diagramma del chip, vi potranno essere d'aiuto anche una matita e una gomma.

Determinazione di un'area in memoria

Per programmare il chip dobbiamo essere in grado di comunicargli dove è situato il numero su cui si deve operare, il suo "indirizzo" in memoria. Questo potrebbe sembrare banale, siccome, per iniziare, abbiamo solo quattro locazioni di memoria, ma in realtà è probabilmente l'aspetto più difficile del codice macchina, specialmente per i principianti.

Quando parleremo del chip 7501 potremo vedere che è capace di comprendere 13 diversi modi di specificare l'indirizzo al quale un numero si può trovare. In questa semplice introduzione consideriamo solo alcune possibilità per darvi alcune idee sulla varietàdi quello che si può intendere col termine "modo di indirizzamento". In ogni caso ,lo scopo è specificare un numero sul quale devono essere eseguite delle operazioni, l'unica differenza è dove il chip dovrà cercare quel numero.

a) Inserendo l'indirizzo dentro lo stesso comando: questa è la più semplice forma di istruzione. Per esempio,il chip è capace di muovere un numero dal registro x all'accumulatore. Per fare questo, non dobbiamo fare altro che dare un singolo comando (trasferire X all'accumulatore). Poichè il chip conosce tutto dei suoi registri, non serve niente altro. Ricordate che i numeri possono essere trasferiti sia dal registro X che da quello Y all'accumulatore o viceversa, ma non potete trasferirli direttamente tra il registro X e quello Y.
b) Specificando il numero nel comando: qui il comando deve assumere la forma "carica il numero 123 nell'accumulatore". Ognuno dei tre registri può essere "caricato" in questo modo.
c) Specificando un indirizzo in memoria al quale un numero può essere trovato: supponiamo, per esempio che la locazione di memoria 1 contenga il numero 123. Dovremmo dare una istruzione nella forma "carica il registro X con il contenuto della locazione di memoria uno". Tutti i tre registri possono essere caricati in questo modo.
d) Prendendo un indirizzo in memoria da un altro indirizzo. Questo modo è più complicato. Supponiamo di avere il numero 1 nella locazione di

memoria 4 e lasciamo il 123 usato nell'ultima sezione nella locazione di memoria 1. Ora diamo una istruzione della forma: "prendi l'indirizzo del numero dalla locazione di memoria 4 e poi prendi il numero all'indirizzo che hai avuto e caricalo nel registro Y". Il risultato di questo è che il chip esplora prima la locazione di memoria 4. Da essa prende un numero, che, nel nostro caso, è il numero uno. Esso tratta l'1 come un indirizzo e immediatamente prende il numero da quell'indirizzo. L'indirizzo trovato è, naturalmente, 1, così esso prende il contenuto della locazione di memoria 1, il numero 123, e carica quel numero dentro il registro Y.

e) Specificando una posizione in un blocco. Supponiamo che qualcuno vi dia informazioni di una casa in una città sconosciuta. Le informazioni sono le seguenti: "Andate lungo questa strada fino ad arrivare alla quinta traversa sulla destra, la casa è la quarta sulla sinistra lungo quella strada". Non c'è alcuna grossa difficoltà in ciò: vi viene dato un punto chiaramente riconoscibile per iniziare a contare e poi un numero di case da contare da quel punto. Il chip è provvisto di un metodo di indirizzamento che è molto simile. Un esempio per verificare questo: il registro X contiene 1 e la locazione di memoria 2 contiene 234. Noi emettiamo l'istruzione "carica nell'accumulatore il numero che si trova ad un indirizzo uguale a 1 più il contenuto del registro X". Il risultato dell'addizione tra il contenuto del registro X e 1 è 2, così il chip prende il valore 234 nella locazione di memoria 2 e lo carica nell'accumulatore.

Questi sono tutti i modi di indirizzamento disponibili sul Lawland 10000, e sebbene in realtà il 7501 ne ha alcuni di più potrete riconoscere, quando arriveremo a loro, che essi sono tutti basati su quello che si è appena detto.

Uso dello stack

Il modo più semplice per vedere se avete capito cosa sia lo stack è porvi un test. Immaginate che le quattro locazioni di memoria 1-4 contengono i numeri 1-4, ma nell'ordine inverso, così che la locazione 1 contenga 4 e così via. Il vostro compito è usare l'accumulatore e lo stack per mettere le notizie nell'ordine giusto, con 1 nella locazione di memoria 1 ecc. Avete appena conosciuto cosa il chip è capace di fare in termini di trasferimento di numeri tra i registri e la memoria; tutto quello che dovete conoscere in più è che un numero può essere trasferito dall'accumulatore allo stack e viceversa. Se ogni trasferimento tra qualsiasi due locazioni è un'istruzione, vi dovrebbero bastare 16 istruzioni per invertire l'ordine dei numeri in memoria.

La soluzione,se ne avete bisogno, consiste nel caricare ognuna delle locazioni di memoria da 1 a 4 nell'accumulatore a turno, poi collocarle nello stack. Poi riprenderle semplicemente dallo stack e, a turno, rimetterle in memoria, ma iniziando ad inserirle dalla locazione 4 e andando all'indietro. Il compito, ovviamente, è banale e ci sono modi più rapidi per realizzare ciò, ma è un utile esempio dello stack usato come blocco-appunti e del modo col quale lo stack rovescia l'ordine dei dati.

Perchè i flag

Gli otto piccoli riquadri riportati nel vostro diagramma del chip rappresentano i "flag", che come è già stato detto, immagazzinano decisioni del tipo si/no riguardanti i compiti che il chip si assume. Non entriamo nei dettagli che vedremo più tardi, ma un paio di informazioni vi daranno un'idea di come i flag siano essenziali.

a) Numeri fuori del range del chip. Supponiamo che il nostro chip riceva una istruzione per sommare un numero a quello che è nell'accumulatore. Un tale comando potrebbe chiaramente avere un largo numero di usi nella realtà, ma può portare con esso alcuni problemi. Il nostro chip Lawland 10000 è limitato nel registrare numeri composti da non più di 4 cifre, oltre è impossibile. Così cosa accade se il numero nell'accumulatore è 9999 e diciamo al chip di addizionare uno ad esso? Fedele ad ogni comando il chip addiziona uno e dà come risultato "0000". Naturalmente la risposta dovrebbe essere 10000 ma il chip può solo mostrare le prime quattro cifre, leggendo dalla destra. Il risultato è un errore, ed un errore che potrebbe avere serie ripercussioni quando si trattano forme complesse di calcolo. E qui subentrano i flag, o almeno uno di essi. Il riquadro CF rappresenta il "flag di carry" ed indica se, nel processo di esecuzione dell'ultimo comando, si è creato un numero che è fuori della gamma di valori che il chip solitamente tratta. In questo caso, se avessimo osservato il flag di carry dopo eseguita l'addizione, esso avrebbe contenuto "1", che significa "sì c'era un 'carry'", ed avremmo immediatamente conosciuto che il vero risultato dell'addizione non era "0000" ma "10000" ed avremmo agito conseguentemente.

b) Controllare un risultato nullo. Scoprire un risultato nullo è una parte importante della programmazione. Poniamo che si usi uno dei registri per contare un numero di attività. Mettiamo il numero di volte che vogliamo ripetere una azione in uno dei registri e poi, ogni volta che l'azione è eseguita, sottraiamo uno dal totale del registro. Alla fine si giungerà a zero e il compito è stato eseguito il corretto numero di volte. Ma come possiamo conoscere che il registro ha raggiunto lo zero? La risposta è nuovamente un flag, questa volta ZF ovvero il "flag di zero". Questi è ideato in modo tale che se il risultato dell'ultima operazione eseguita su uno dei registri o locazioni di memoria vale zero, si setta a uno per indicare "sì, il risultato di quest'ultima operazione è uno zero".

Tralasciamo gli altri cinque flag (uno degli otto non è usato) per il momento, vedremo più tardi che essi sono parti vitali della programmazione in codice macchina. Per il momento è sufficiente che capiate che un flag è un indicatore che risponde a specifiche domande: se la risposta a quella domanda è sì il flag avrà il valore "1", diversamente il flag conterrà zero.

Formato delle istruzioni in codice macchina

Fino ad ora abbiamo usato, molto vagamente, frasi come "diamo una istruzione al chip", tralasciando come impartire queste istruzioni.

La risposta è che noi usiamo delle locazioni di memoria per immagazzinare un programma. I programmi in codice macchina sono costituiti da numeri, pertanto la programmazione in codice macchina consiste nel depositare numeri dentro la memoria nel giusto ordine. Nelle tavole distribuite in tutto il libro, potrete trovare precisi dettagli dei numeri che avrete bisogno di usare per impartire comandi al chip 7501, ma, per il momento, immaginiamo alcune semplici istruzioni per il Lawland 10000 per ottenere alcuni degli obiettivi che abbiamo appena esaminato.

1) Trasferire un numero dalla memoria ad un registro. Supponiamo che il nostro obiettivo sia caricare l'accumulatore con il contenuto del registro X; quale forma prenderà questa istruzione? Se riguardate la descrizione dei modi di indirizzamento, potrete vedere che non abbiamo bisogno di un indirizzo per fare questo, così si è creato un solo comando che dice al chip di prendere qualsiasi cosa sia nel registro X ed immetterla nell'accumulatore. Il chip è ideato per riconoscere questo specifico comando sotto la forma del numero 1000.
2) Trasferire un determinato numero all'accumulatore. L' obiettivo è caricare l'accumulatore con il numero 123. Chiaramente non possiamo cavarcela qui con un'istruzione composta da un singolo numero. Il chip avrà bisogno di un numero che verrà riconosciuto come istruzione per caricare l'accumulatore con un numero che abbiamo scelto (2000), e poi un altro numero che è il valore da caricare. Per impartire questa istruzione al chip, gli inviamo quindi due numeri: 2000, 123. Nella ideazione del chip ci saremo dovuti assicurare che, quando esso incontra il comando 2000, sappia che deve aspettare un altro numero seguente che è il numero da collocare nell'accumulatore.
3) Caricamento di un numero dalla memoria all'accumulatore. Una diversa strada va seguita per caricare il contenuto della locazione di memoria 1 nell'accumulatore. Anche qui dobbiamo impartire l'istruzione in due parti, la prima parte che dice al chip il tipo di istruzione e la seconda parte che dà l'indirizzo dal quale un numero deve essere preso. Notate qualcosa di molto importante: quando descriviamo l'obiettivo da eseguire come "caricare l'accumulatore", usiamo la stessa frase nelle sezioni 2 e 3. In una carichiamo un numero direttamente e nell'altra carichiamo un numero da una locazione di memoria. Il nostro cervello è perfettamente capace di capire la distinzione tra le due, sebbene metà delle parole siano le stesse. Il chip non è capace di fare tali sottili distinzioni, deve prendere i due tipi di istruzioni totalmente separate. Per caricare un numero dalla memoria abbiamo bisogno di una istruzione distinta, 2001. Per impartire

tale istruzione al chip gli inviamo: 2001, 1, che gli comunica di caricare il contenuto della locazione di memoria 1 all'accumulatore.

Questi tre semplici esempi immaginari spiegano una importante lezione. Non esiste necessariamente alcun collegamento concreto tra il modo con cui una istruzione è scritta in Inglese e il modo in cui è scritta in codice macchina. Molti principianti si confondono quando un testo di codice macchina dice loro che esiste una istruzione chiamata "caricare l'accumulatore" e che si completa con un numero di diversi "modi di indirizzamento" o metodi per specificare dove sia il numero da caricare. Il vero è che tutte le istruzioni sono differenti. Se ci sono otto differenti modi di specificare il numero da caricare nell'accumulatore ci saranno anche otto diverse istruzioni. In questo e in altri testi le troverete come "LDA" che è l'abbreviazione di "load accumulator". Spiegare in dettaglio ogni istruzione ed il suo modo di indirizzamento richiederebbe un libro di centinaia di pagine, per cui basti sapere che LDA funziona in otto diversi modi di indirizzamento, e che ci sono otto differenti istruzioni per caricare l'accumulatore.

La seconda lezione da trarre dagli esempi è che non c'è una lunghezza standard per un'istruzione in codice macchina. Il numero di locazioni di memoria che una istruzione richiede dipende dal suo modo di indirizzamento. L'effettiva istruzione da sola, sia sul Lawland 10000 sia sul 7501, non prenderà mai più di una sola locazione di memoria ma ci sono anche i byte necessari per specificare il numero o l'indirizzo su cui operare. Nella programmazione in codice macchina troverete che una istruzione completa può avere 1, 2, 3, o più byte. Questo non è un problema per il chip, se esso è ideato per conoscere esattamente quanti altri numeri seguono una istruzione. Il problema si pone quando l'utente invia al chip una istruzione che richiede due altri numeri, ma ne fornisce uno solo. Quello che accadrà poi è che il chip tratterà qualsiasi cosa gli si invii come parte della precedente istruzione, che sia vostra intenzione o meno.

Il programma e il program counter

Dunque un programma in codice macchina è fatto da numeri immagazzinati in memoria, e quei numeri formano differenti istruzioni che possono avere diverse lunghezze. Fin qui tutto bene, ma come possono formare un programma? Quale è la differenza tra una collezione casuale di numeri e un programma che il chip seguirà?

La risposta a ciò è che il programma segue il percorso indicato dal "program counter", il registro che vedete riprodotto in basso, alla destra del diagramma del chip. Il program counter contiene sempre un indirizzo, e quell'indirizzo è la posizione della prossima istruzione che il chip si appresta ad eseguire. Se, per esempio, siete in BASIC ed inserite SYS 4567, il comando per eseguire un programma in codice macchina, tutto ciò che accade è che il chip 7501, che era alle prese con l'Interprete BASIC, vede nel suo program counter il numero 4567 ed immediatamente comincia ad obbedire all'istru-

zione che trova in esso.

Con il nostro modello di chip, si dovrebbe operare così:
L'istruzione per "trasferire il contenuto del registro X all'accumulatore" è 2500.

L'istruzione per "immagazzinare il contenuto dell'accumulatore in un indirizzo specificato" è 3001, seguita da un altro numero specificante l'indirizzo.

Istruite il chip come segue:

123 nel registro X
234 nell'accumulatore
2500 nella locazione di memoria 1
3001 nella locazione di memoria 2
4 nella locazione di memoria 3
1 nel program counter.

Poiché il program counter indica la locazione di memoria 1, il chip vi cercherà una istruzione. Trova 2500, così trasferisce il 123 dal registro X all'accumulatore. Ora avanza alla locazione di memoria 2. Qui trova 3001, che riconosce come una istruzione che richiede un ulteriore numero, quindi prende il "4" in locazione 3. Con questi due valori che ha prelevato esso sa che deve inserire il contenuto dell'accumulatore nella locazione 4, che finisce contendo 123.

Se per cominciare, avessimo caricato 2 nel program counter il trasferimento del 123 all'accumulatore non avrebbe avuto luogo, poiché il chip non avrebbe visto quella istruzione, e la locazione di memoria 4 avrebbe contenuto 234.

Il program counter è dunque fondamentale per ogni programmazione in linguaggio macchina. Più avanti nel libro scoprirete che i GOTO e GOSUB del BASIC sono paragonabili, in codice macchina ad istruzioni che pongono nuovi valori nel program counter, costringendo così il chip a saltare all'interno di un programma.

Codice macchina e linguaggio assembly

In tutte le cose viste in precedenza abbiamo evitato una importante fonte di confusione di cui ancora potete non esservene accorti. Abbiamo parlato diverse volte a proposito dei nomi dati a singole istruzioni in codice macchina e pure a riguardo dei numeri, rappresentanti essi stessi istruzioni comprensibili dal chip. Nella sezione precedente abbiamo visto anche che un programma in codice macchina non sia niente di più di una serie di numeri registrati in memoria. Un programma sarà portato a termine, a patto che il program counter indichi l'inizio di una serie di istruzioni valide in codice macchina.

Nella descrizione dei modi di indirizzamento abbiamo anche detto che i neofiti del codice macchina vanno spesso in confusione per il fatto che le istruzioni sono raccolte in gruppi, come le istruzioni "carica l'accumulatore", sebbene ogni diverso metodo di caricamento dell'accumulatore sia una istruzione interamente differente.

La confusione non sussiste se programmassimo totalmente solo nella forma di codice macchina puro e semplice ponendo cioè numeri in memoria. Il problema consiste nel fatto che mentre i chip trovano naturale comunicare tramite numeri, l'uomo normalmente no.

Per questa ragione, è stato sviluppato un apposito linguaggio per aiutare chi vuole programmare in codice macchina: il linguaggio assembly. Il linguaggio assembly è a volte definito come una forma di stenografia per il codice macchina ma questo può anche non essere vero, in quanto esso prende più spazio del solo codice macchina. Il linguaggio assembly è un modo di scrivere e leggere il codice macchina in modo tale che possa essere compreso da chiunque voglia dedicare un pò di tempo e concentrazione allo scopo. È una convenienza e niente più. I chip non capiscono il linguaggio assembly, infatti essi non sono concepiti per questo, così ogni volta che un programma è scritto in linguaggio assembly, esso è tradotto in codice macchina prima che al chip venga chiesto di eseguirlo.

Abbiamo già accennato alla forma di alcune istruzioni in linguaggio assembly, ma ecco un piccolo esempio per darvene il tono: LDA 100 è una istruzione per caricare il numero 100 nell'accumulatore. LDA 100 fa caricare il contenuto della locazione di memoria 100 nell'accumulatore. LDA 100,X fa caricare nell'accumulatore il contenuto dell'indirizzo ottenuto sommando il contenuto del registro X e 100.

Non è necessario scervellarsi su questi esempi poichè tratteremo queste istruzioni più avanti. La cosa importante da notare è che nel linguaggio assembly le diverse istruzioni per caricare un numero nell'accumulatore iniziano tutte con lo stesso termine "LDA" cioè (L)oa(D) (A)ccumulator. Quale delle differenti istruzioni di caricamento dell'accumulatore si intenda si scoprirà osservando la forma di quello che segue l'istruzione. Se volete spendere del tempo operando con i linguaggi codice macchina/assembly imparerete molto velocemente a riconoscere le forme con cui sono espressi i 14 diversi "modi di indirizzamento".

CAPITOLO 2

I MISTERI DEL CHIP ARITHMETIC

Nel capitolo precedente abbiamo iniziato a vedere in termini molto grossolani alcune delle capacità ed i problemi presentati da un chip immaginario chiamato Lawland 10000. In questo capitolo continueremo con questa meraviglia tecnologica per iniziare a vedere il modo con cui un chip esegue calcoli aritmetici.

A questo punto non bisogna restare impantanati. Molte persone non superano la descrizione del sistema di numeri usato dal chip, non comprendendo che, per una larga serie di usi della programmazione, non è necessaria una profonda conoscenza. Con tutto ciò, se cominciate a prendere padronanza del modo col quale un chip tratta il calcolo aritmetico, e perchè, potrete evitare più avanti di sbagliare.

Poichè alcuni potrebbero creare un pò di confusione, abbiamo deciso in questo libro di adottare un nuovo metodo di approccio. Avete già appreso, dal capitolo precedente, che il nostro chip Lawland 10000 opera con numeri decimali, il sistema numerico basato sulle potenze di dieci che usiamo nella vita di tutti giorni. Altresì probabilmente saprete che i chip normalmente non operano in decimale, ma in binario, il sistema numerico basato sulle potenze di due. Il perchè di questo, e quali difficoltà o possibilità esso susciti, lo lasceremo al capitolo successivo. Abbandoneremo la numerazione decimale nella speranza che, una volta impadroniti dei principi, avrete molte meno difficoltà nel comprendere cosa accade in un vero chip che usa numeri binari.

Il range di un chip

Nel capitolo precedente abbiamo visto che il Lawland 10000 era capace di immagazzinare, in una sola locazione di memoria, qualsiasi numero nella gamma 0-9999, cioè un numero che richiede non più di quattro cifre. Questo è un importante fattore per stabilire le capacità di un chip poichè il limite imposto dal numero di cifre, siano 4, 8 o, nel caso di chip più recenti e molto potenti, 16 o 32 cifre, è assoluto. Ogni cosa che vogliamo fare va fatta dentro quelle cifre e ogni operazione con numeri più larghi può essere effettuata solo usando speciali tecniche, che dividono quei numeri in parti. Questa limitazione ha conseguenze importanti per il modo con cui il chip è ideato e quello in cui è usato, come potremo vedere quando metteremo il chip ad operare su uno dei compiti per cui è indicato, sommare due numeri.

L'addizione e l'uso del flag di carry

Nello studiare l'addizione non analizzeremo ciò che sta accadendo "meccanicamente" dentro il chip, annoteremo puramente il modo con cui l'addizione procede, che è semplicemente il processo che tutti imparammo (si spera) a scuola, di sommare due singole cifre e riportare ogni "carry".

Facciamo l'esempio di 4567 + 5678, possiamo dividere il processo in quattro fasi:

(4)	(3)	(2)	(1)
4	5	6	7
+	+	+	+
5	6	7	8
+	+	+	
CARRY	CARRY	CARRY	

La fase (1) porta all'addizione di 7 e 8 e produce 15, dove 5 è registrato e 1 è chiamato "carry" perchè deve essere riportato nella prossima colonna alla sinistra. La fase (2) ora comprende l'addizione di 6, 7 e 1, e produce 14, dove quattro è registrato e uno è riportato avanti. La fase (3) consiste nell'addizione di 5, 6 e 1 dando 2 da registrare e 1 da riportare. La fase finale (4) esegue l'addizione di 4, 5 e 1. Il risultato è 10 e, poichè non ci sono più altre colonne alla sinistra, il risultato finale delle quattro fasi è 10345.

Stabilito che stiamo addizionando soltanto due numeri l'ammontare che deve essere riportato alla sinistra non sarà mai superiore a 1; infatti 9 + 9 è la somma più grande di due singole cifre e dà 18. Quindi il riporto dentro il numero non è una cosa complessa. Il chip, che può addizionare solo due numeri alla volta deve semplicemente essere in grado di riconoscere quando una particolare addizione ha dato una somma maggiore di 9 e quindi sommare 1 ai prossimi due valori alla sinistra .

Secondariamente, bisogna notare che qualcosa in sovrappiù è apparso alla sinistra del nostro numero. Sappiamo cosa sia poichè abbiamo brevemente visto il problema nel capitolo precedente. Il numero che abbiamo creato è troppo grande per essere contenuto in 4 cifre, e la cifra in più alla sinistra del numero deve essere collocata nel "flag di carry" che esiste proprio per questa funzione. Più avanti nel capitolo vedrete che una lunga serie di calcoli aritmetici dipende dall'uso del flag di carry per riprendere cifre potenzialmente perdute.

Programma BASIC per dimostrare una semplice addizione.

```
10 SCNCLR
20 INPUT "NUMBER 1 (0-9999):";N1
30 N1$ = RIGHT$("0000" + MID$(STR$(N1),2),4)
```

```
40 INPUT "[CD]NUMBER 2 (0-9999):";N2
50 N2$ = RIGHT$("0000"+MID$(STR$(N2),2),4)
60 BASE = 10
70 PRINT
80 FOR I = 1 TO 4
90 PRINT TAB(I*8-2) BASE "^" 4-I
100 PRINT "[CD][CD]" TAB(I*8+4) MID$(N1$,I,1)
110 PRINT TAB(I*8+4) MID$(N2$,I,1)
120 PRINT TAB(I*8-2) "--------"
130 PRINT TAB(I*8-2) "[CD]--------"
140 PRINT "[CU][CU][CU][CU][CU][CU][CU] [CU][CU]"
150 NEXT I
160 PRINT "[CD][CD][CD][CD][CD][CD]"
170 CARRY = 0
180 FOR I = 4 TO 1 STEP -1
190 T1 = ASC(MID$(N1$,I,1))-48
200 T2 = ASC(MID$(N2$,I,1))-48
210 PRINT "[CU][CU]"
220 SUM = T1+T2+CARRY
230 CARRY=0
240 IF SUM>=BASE THEN SUM = SUM-BASE : CARRY = 1
250 PRINT TAB(I*8+3) SUM
260 IF CARRY=1 THEN PRINT TAB(I*8-8) "[CD]CARRY[CU][CU]"
270 GETKEY T$
280 NEXT I
290 PRINT "[CD][CD][CD][CD]"
300 END
```

IMPORTANTI NOTE sul formato del programma BASIC:
I programmi BASIC in questo testo sono stati elaborati per renderli più leggibili. Le maggiori variazioni sono:
a) I caratteri di controllo, come quelli per lo spostamento del cursore, sono stati sostituiti dalle abbreviazioni tra parentesi quadre:
[CD] = CURSOR DOWN
[CU] = CURSOR UP
[CL] = CURSOR LEFT
[CR] = CURSOR RIGHT
[RVS ON] = REVERSE ON
b) I caratteri grafici sono sostituiti con i seguenti codici:
[B] = tasto B + Shift
[C= B] = tasto B + Commodore

L'obiettivo dell'inclusione nel testo di programmi BASIC come questo non è inserire una grossa quantità di commenti ma semplicemente procurarvi

alcuni strumenti per analizzare procedure che a volte mentalmente si visualizzano con difficoltà. Il programma in esame accetta due numeri privi di segno (cioè non possono essere negativi) e, quando si preme la barra spaziatrice, mostra come le cifre sono sommate assieme, con ogni riporto risultante. Se avete qualche difficoltà nella visualizzazione del modo in cui avvengono queste semplici addizioni, utilizzate il programma diverse volte fino a che avete chiare nella vostra mente le regole.

Semplici moltiplicazioni

Abbiamo notato che il Lawland 10000 è un chip con una mente veramente semplice. Tuttavia, è capace di eseguire forme molto grossolane di moltiplicazione e di divisione, cioè moltiplicazioni o divisioni per 10. Fà ciò semplicemente muovendo le cifre di un numero a sinistra o a destra. Se il numero iniziale fosse 1234, per esempio, il flag di carry e l'accumulatore mostrerebbero questo:

CF	ACCUMULATORE
0	1234

Se le cifre fossero spostate verso sinistra,il risultato sarebbe:

CF	ACCUMULATORE
1	2340

La divisione viene eseguita semplicemente spostando le cifre nell'altro senso:

CF	ACCUMULATORE
0	0123

Quello che succede alle cifre che sono spostate fuori dal numero dipenderà dal tipo di istruzione usata, come vedrete quando si arriverà al capitolo 13.

Trattamento dei numeri più grandi

9999 non è, dopo tutto ciò che è stato detto e fatto, un numero molto grande, specialmente se deve essere il limite assoluto di operazioni per un modesto computer. Fortunatamente, sebbene il Lawland 10000 sia limitato ai numeri con quattro cifre, il programmatore può facilmente manipolare valori più grossi, dividendoli in sezioni. Usando due sezioni, per esempio,

sarebbe possibile trattare valori fino a 99.999.999. Il metodo per fare ciò, che sarà trattato per il 7501, è usare due locazioni di memoria per ogni numero. Ogni locazione di memoria, conosciuta come "byte, immagazzinerà una sezione del numero, una chiamata "high byte (byte alto) e l'altra, ovviamente, "low byte (byte basso). L'high byte è espresso in unità maggiori di uno del più grande numero memorizzabile in una singola locazione di memoria. Nel caso del Lawland 10000 il più grande numero esprimibile con un singolo byte è 9999, così l'high byte sarà in unità di 10000. Con un esempio pratico, se vogliamo immagazzinare il numero 12345678, noi metteremo 5678 nel low byte, e le 1234 unità di 10000 nell'high byte. Se volessimo potremmo operare con più di due byte; tre byte ci permetterebbero di operare con numeri tipo 999.999.999.999.

Chiaramente le tecniche per lavorare con tali numeri non sarebbero esattamente così semplici come quelle per i numeri memorizzabili in un singolo byte. I problemi, comunque, non sono grandi, principalmente si cerca di assicurare che ogni riporto da un byte sia fornito al byte successivo. Nei capitoli seguenti vi sarà mostrato in dettaglio come operare con numeri a due byte sul 7501.

Il problema dei numeri negativi

Operare con i numeri negativi non è la più semplice delle cose. Se trovate difficile quanto segue, vi capiamo, dato che molti programmatori evitano di lavorare con numeri negativi. La maggior parte dei programmi può operare perfettamente senza mai ricorrere ai numeri negativi eccetto per il calcolo degli indirizzi conosciuti come "salti relativi.

Stabilito l'effettivo range del Lawland 10000, c'è la sgradevole sorpresa di scoprire che il suo range è molto più piccolo. La ragione di ciò è che dobbiamo considerare pure i numeri negativi. Un numero come -9999, può sembrare un numero negativo a quattro cifre, ma è un inganno. Per esprimere -9999 sulla carta abbiamo bisogno di CINQUE simboli, i quattro nove PIÙ il segno meno. Siccome il nostro chip è limitato a quattro cifre, non è in grado di registrare un numero tipo -9999, per cui qualcosa non va. Naturalmente, sarebbe possibile inserire un valore in qualche altro luogo, magari in un 'flag'. Il problema è che in un vero computer, tutta la memoria viene riempita di numeri ed è impossibile inserire una registrazione per dire se ognuno di essi è positivo o negativo. Quello che dobbiamo essere in grado di fare, guardando ogni numero, è dire istantaneamente se è positivo o negativo, e ciò può essere fatto solamente se il segno positivo o negativo è parte integrante del numero stesso.

Infatti,come avrete senza dubbio supposto, quello che si perderà è una delle cifre. Se vogliamo essere in grado di trattare numeri negativi, come, a volte, facciamo nella realtà tutto quello che possiamo manipolare con le

nostre quattro cifre è compreso tra +999 e -1000 (spiegheremo i cinque caratteri di -1000 in un attimo). Il segno meno prende il posto di una delle cifre, in realtà della prima di esse.

Nella addizione,il chip è incapace di comprendere il simbolo "- e cercherà di rappresentare il segno meno con qualcosa che PUÒ capire. Per questa ragione, quando lavora con numeri che possono essere sia positivi che negativi, il Lawland 10000 tratterà ogni numero con uno zero nella colonna più a sinistra come positivo e ogni numero con un nove nella colonna più a sinistra come negativo. Mettendo assieme tutto quello che è stato detto, si trova che il nostro chip decimale può interpretare 0999 come + 999 e 9999 come -999. Il meno 1000 nominato in precedenza deriva dal fatto che possiamo avere apparentemente due versioni di zero, nominalmente 0000 (più zero) e 9000 (meno zero). Chiaramente non servono due forme di zero così possiamo istruire il chip a trattare 9000 come un numero maggiore, di uno, di 9999. In tutti i chip che incontrerete nella realtà, il più grosso numero negativo che il chip sarà in grado di trattare sarà maggiore di uno del più grosso numero positivo. Si vede dunque che, adottando questo semplice metodo, se vogliamo operare con numeri che sono sia positivi che negativi possiamo avere un effettivo range che va da +999 a -1000.

I numeri negativi e la sottrazione

Quindi,si è visto,per tutti gli scopi pratici, che, sebbene abbiamo perso parte del range del chip, abbiamo infine risolto il problema della memorizzazione dei numeri negativi.

Sfortunatamente, non è esatto quanto è semplice. Immagazzinare numeri negativi è una cosa, ma operare con questi è un'altra. La ragione di questo è che operare con numeri negativi implica sottrazione e il Lawland 10000, come molti altri chips, incluso il 7501, è incapace di sottrarre! Partendo dalle cifre più piccole di due numeri, è programmato per sommarli assieme, ricordando ogni riporto, poi ad agire sulla prossima coppia di cifre, sommandole, più ogni riporto dalle precedenti coppie e così via. I metodi richiesti per la sottrazione, mentre alla nostra mente sembrano uguali, sono in realtà assai differenti ed al di fuori di quelli utilizzati dal chip. Per sopperire a questa limitazione abbiamo adottato un nuovo sistema di numerazione chiamato "complemento a 10. In qualche modo è simile alla semplice forma di immagazzinamento dei numeri positivi/negativi visti in precedenza ma il suo vantaggio è che permette di eseguire sia l'addizione che la sottrazione con l'unico metodo di addizione.

Consideriamo il seguente esempio:

Addizioniamo: 9999
0001

La risposta: 10000

Ma quale sarebbe la risposta per quanto riguarda il Lawland 10000? Può solo manipolare quattro cifre, così mette la quinta cifra nel flag di carry e registra il risultato principale come zero.
Ora provate questo:

0100
9950

Il vero risultato è 10050 ma al chip risulta 50 (scartando la quinta cifra).

Quello che abbiamo fatto,in entrambi i casi, è prendere un numero positivo con uno zero nella colonna più a sinistra e apparentemente sommargli un numero negativo che ha un nove nella colonna più a sinistra. Se osservate più da vicino i due numeri negativi impiegati vedrete, comunque, che 9999 è 10000 meno uno e 9950 è 10000 meno 50. Quando addizioniamo 10000-1 e 1, il chip registra la risposta come zero. Quando addizioniamo 10000-50 e 100, la risposta registrata è 50. Se ci pensate, vedrete che prendendo il numero che vogliamo sottrarre, sottraendolo prima da 10000 e poi addizionandolo al numero da cui vogliamo sottrarlo ci compare la risposta giusta. Provatelo con i seguenti esempi:

888
-777
500
-500

Basandoci su questa scoperta abbastanza eccezionale, possiamo vedere come la sottrazione può essere semplicemente eseguita tramite l'addizione. Se vogliamo creare numeri negativi, non dobbiamo semplicemente mettere un "9" all'inizio di essi, ma dobbiamo sottrarre il numero desiderato da 10000, mettendo automaticamente il "9" nella sua corretta posizione e creando una strana forma di numero negativo, che però funziona.

Quando chiediamo al Lawland 10000 di sottrarre, quello che farà è eseguire la stessa trasformazione sul numero che deve essere sottratto e poi sommare i due numeri. Sfortunatamente, manca una pagina dalla nostra copia del manuale di descrizione tecnica del Lawland 10000 e non possiamo descrivere come il chip effettivamente crei il numero "offset (equivalente) quando non può sottrarre. Quando osserveremo il 7501, comunque, troverete che non ci sono misteri nel suo sistema di numerazione binariaa.

Nonostante bisogna prendere ciò un pò alla cieca, alla fine possiamo vedere che è possibile per il nostro chip eseguire sottrazioni senza tuttavia comprendere cosa stia facendo.

Quando un numero negativo non è un numero negativo

Avendo imparato tutto dei numeri negativi è importante ricordarsi cosa abbiamo fatto e cosa NON abbiamo fatto.

Non abbiamo istruito il chip a funzionare in un modo speciale affinché tratti i numeri con segno, numeri che possono essere sia positivi che negativi, piuttosto che numeri senza segno. Il chip opererà esattamente nello stesso modo per entrambi. Quello che abbiamo fatto è alterare il formato dei numeri in modo che i risultati che vengono dati dal chip abbiano un senso entro i limiti del formato.

Sommando -100 (9900) e + 50 (0050), non si produrrà un risultato di -50 in qualche parte dentro il chip, ma invece si otterrà il risultato 9950. Se vogliamo considerare 9950, per qualche scopo, come "realmente" significante -50, sta a noi programmare secondo questa assunzione, per il chip non c'è nessuna differenza nel trattamento dei numeri.

Alcune peculiarità dei numeri negativi

Fin qui va bene, ma c'è molto da dire se volete usare i numeri negativi in modo attendibile. Per il momento indicheremo semplicemente alcuni problemi, lasciando ai prossimi capitoli la spiegazione delle tecniche precise da impiegare per trattare con essi.

a) Il formato per i numeri negativi è sempre lo stesso. Mettendo un nove all'inizio di un numero come 100 non lo si tramuterà in meno 100, ma in 900 (1000-100). In altre parole più grosso sembra a prima vista un numero negativo, più piccolo è in realtà.

b) Regole diverse si applicano quando si forma un "carry" mentre si addizionano numeri negativi. Per esempio, se addizioniamo 9001 e 9001 (meno 999 più meno 999), il risultato sarà 18002, con l'"1" che entra nel flag di carry e, come risultato principale 8002. Secondo le nostre regole, questo non è un numero negativo, quindi qualcosa deve essere sbagliato. Il guaio è che la parte principale del numero usa solo le prime tre cifre dalla destra e ogni carry dentro o fuori le quattro cifre può guastare il formato del numero così che non siamo più tanto sicuri del segno del numero. Ciò è trattato dal flag OF, visibile sul vostro diagramma, "flag di overflow". Il compito di questo flag è di registrare se viene fatto un carry dalle tre prime cifre alla quarta cifra. Nei prossimi capitoli vedrete che il flag è usato in unione con il flag di carry per scoprire se dall'addizione risulta, nella cifra del segno, un numero sbagliato.

Una ulteriore facilitazione è data dal "flag di sign" (SF sul diagramma), che registra alla fine di ogni operazione se il risultato nell'accumulatore ha un "9" nella posizione della quarta cifra. I flag lavorano in continuazione, indifferentemente se si stanno usando numeri negativi, il che è per lo più irrilevante, ma può essere molto utile quando addizioniamo o sottraiamo numeri con segno.

Dimostrazione BASIC di addizione che impiega numeri con segno

```
10 SCNCLR
20 INPUT "NUMBER 1 (-1000 TO 999):";N1
30 IF N1<0 THEN N1 = 10000+N1
40 N1$ = RIGHT$("0000"+MID$(STR$(N1),2),4)
50 INPUT "[CD]NUMBER 2 (-1000 TO 999):";N2
60 IF N2<0 THEN N2 =10000 +N2
70 N2$ = RIGHT$("0000"+MID$(STR$(N2),2),4)
80 BASE = 10
90 PRINT
100 FOR I = 1 TO 4
110 PRINT TAB(I*8-2) BASE "^" 4-I
120 PRINT "[CD][CD]" TAB(I*8+4) MID$(N1$,I,1)
130 PRINT TAB(I*8+4) MID$(N2$,I,1)
140 PRINT TAB(I*8-2) "--------"
150 PRINT TAB(I*8-2) "[CD]--------"
160 PRINT "[CU][CU][CU][CU][CU][CU][CU] [CU][CU]"
170 NEXT I
180 PRINT "[CD][CD][CD][CD][CD][CD]"
190 CARRY = 0
200 FOR I = 4 TO 1 STEP -1
210 T1 = ASC(MID$(N1$,I,1))-48
220 T2 = ASC(MID$(N2$,I,1))-48
230 PRINT "[CU][CU]"
240 SUM = T+T2+CARRY
250 LASTCARRY = CARRY
260 CARRY = 0
270 IF SUM>=BASE THEN SUM = SUM-BASE : CARRY = 1
280 OVERFLOW = LASTCARRY<>CARRY
290 PRINT TAB(I*8+3) SUM
300 IF CARRY=1 THEN PRINT TAB(I*8-8) "[CD]CARRY[CU][CU]"
310 GETKEY T$
320 NEXT I
330 PRINT "[CD][CD][CD][CD]"
340 IF OVERFLOW THEN PRINT "SIGN DIGIT IN ERROR"
350 END
```

Questo secondo programma dimostrativo presenta l'immagine dei numeri "offset" permettendovi di specificare i numeri negativi. Il programma crea automaticamente il numero offset nella forma con la quale dovrebbe essere usata sul Lawland 10000, e poi illustra il processo di addizione quando premete la barra spaziatrice.

Dimostrazione BASIC della sottrazione che impiega numeri con segno.

```
10 SCNCLR
20 INPUT "NUMBER 1 (-1000 TO 999):";N1
30 IF N1<0 THEN N1 = 10000+N1
40 N1$ = RIGHT$("0000"+MID$(STR$(N1),2),4)
50 INPUT "[CD]NUMBER 2 (-1000 TO 999):";N2
60 IF N2<0 THEN N2 = 10000+N2
70 N2$ = RIGHT$("0000"+MID$(STR$(N2),2),4)
80 BASE = 10
90 PRINT
100 FOR I = 1 TO 4
110 PRINT TAB(I*8-2) BASE "^" 4-I
120 PRINT "[CD][CD]" TAB(I*8+4) MID$(N1$,I,1)
130 PRINT TAB(I*8+4) MID$(N2$,I,1)
140 PRINT TAB(I*8-2) "--------"
150 PRINT TAB(I*8-2) "[CD]--------"
160 PRINT "[CU][CU][CU][CU][CU][CU][CU][CU][CU]"
170 NEXT I
180 PRINT "[CD][CD][CD][CD][CD][CD]"
190 CARRY = 1
200 FOR I = 4 TO 1 STEP -1
210 T1 = ASC(MID$(N1$,I,1))-48
220 T2 = ASC(MID$(N2$,I,1))-48
230 PRINT "[CU][CU]"
240 SUM = T1+(9-T2)+CARRY
250 LASTCARRY = CARRY
260 CARRY = 0
270 IF SUM>=BASE THEN SUM = SUM-BASE : CARRY = 1
280 OVERFLOW = LASTCARRY<>CARRY
290 PRINT TAB(I*8+3) SUM
300 IF CARRY=1 THEN PRINT TAB(I*8-8) "[CD]CARRY[CU][CU]"
310 GETKEY T$
320 NEXT I
330 PRINT "[CD][CD][CD][CD]"
340 IF OVERFLOW THEN PRINT "SIGN DIGIT IN ERROR"
350 END
```

Il programma finale di questo capitolo vi permette di inserire due numeri con segno, sia positivi che negativi, e poi visualizza le procedure per sottrarre il secondo dal primo quando la barra spaziatrice è premuta.

CAPITOLO 3

NUMERI BINARI

Ora è tempo di affrontare la realtà. Quello che si è visto dovrebbe avervi aiutato nel darvi una immagine delle operazioni di un chip ricordandovi che quelle operazioni non sono affatto così complesse come molte persone suppongono. L'obiettivo è ora di convincersi che operare con i numeri binari è un ostacolo piccolo da superare e, una volta superato,sarete sulla strada che vi porterà a programmare direttamente il chip 7501.

Quasi tutto di quello detto fino ad ora a proposito del Lawland 10000 è valido per il 7501 ad eccezione del fatto che il 7501 non può operare con i normali numeri decimali che usiamo nella vita di tutti i giorni, numeri le cui cifre sono basate sulle potenze di dieci, ma lavora con i numeri binari, le cui cifre sono basate sulle potenze di due.

Di volta in volta dovrete essere in grado di tradurre un numero decimale in binario per poter vedere il suo formato, o tradurre un numero binario in decimale ma una tale trasformazione non può essere fatta a memoria. La cosa più importante è che voi comprendiate cosa siano i numeri binari, poichè ciò vi rivelerà molti dei segreti del chip 7501 che volete programmare.

Numeri e basi

Prima che iniziate ad usare dei numeri deve essere presa una decisione a proposito della "base di quei numeri. Il nostro sistema decimale opera su base dieci, il che significa che ha 10 unità (0-9) e che una volta passato il 9, deve essere introdotta un'altra cifra. Il valore di una particolare cifra è determinato considerando la sua distanza dalla destra del numero, così che il numero 1234 può altresì essere descritto come:

(1*10^3)+(2*10^2)+(3*10^1)+(4*10^0)

È anche vero, comunque, che ogni numero (tranne 0 o 1) può essere usato come base per un sistema di numerazione. I più usati comunemente sono il binario (base 2), l'ottale (base 8), duodecimale (base 12) ed esadecimale (base 16). Tra questi il sistema binario, con una base di 2 è universalmente usato dai computers poichè si adatta perfettamente al modo con cui essi operano.

Il computer e i numeri binari

Senza entrare in tecnicismi, i chip montati sul vostro computer sono essenzialmente scatole riempite di minuscoli interruttori on/off. Diversi tipi di chip hanno modi lievemente differenti di indicare se un interruttore è on o off, ma tutti si riassumono nella stessa cosa. Qualsiasi cosa fatta, e qualsiasi cosa immagazzinata, deve essere svolta tramite il settaggio di interruttori. Gli stessi inerruttori sono disposti in gruppi di otto, ogni gruppo conosciuto come "byte . Questi "byte" corrspondono a quello che avevamo chiamato sempre locazioni di memoria , essendoun'area di immagazzinamento in memoria in grado di registrare un numero, sebbene il range di numeri che possono essere ricordati sia molto differente da quello del nostro chip immaginario.

Adesso torniamo ai numeri binari. Così come in numerazione decimale, le unità effettive che si possono usare vanno da 0 a 9 (1 meno di 10), in numerazione binaria le unità vanno da 0 a 1 (1 meno di 2). Ogni numero in numerazione binaria viene espresso in uni e zeri. E pensandoci, ciò è esattamente il modo con cui abbiamo detto che il computer lavora infatti esso consiste di centinaia di migliaia di minuscoli interruttori che possono essere sia on (1) che off (0). Unendo il computer, ed i suoi byte, e la numerazione binaria, non è difficile vedere come il computer lavora con numeri che vanno da zero a otto cifre binarie, tutte settate a "1 , ovvero il numero binario 11111111, che infatti è il decimale 255. C'è, ad ogni modo, un nome particolare dato alle singole cifre che formano un numero. Per salvarci diciamo che 255 è composto di 8 "binary digit (cifre binarie); il nome "binary digit è abbreviato in bit . Il più grosso numero che può essere registrato in un "byte composto da otto "bit" è quindi 255.

Il valore dei numeri binari

Come in precedenza abbiamo descritto 1234 in potenze di dieci, così possiamo descrivere un numero binario in termini di potenze di due. Le otto cifre binarie, o bit, che un singolo byte di memoria può immagazzinare rappresentano, potenzialmente:

$$(2\hat{}7)+(2\hat{}6)+(2\hat{}5)+(2\hat{}4)+(2\hat{}3)+(2\hat{}2)+(2\hat{}1)+(2\hat{}0)$$

cioè

$$128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$$

Notate che, in questo caso, non possiamo moltiplicare ognuna delle cifre per alcuna cifra. Una cifra binaria può essere solo 1 o 0; così una cifra o

rappresenta 1*2 a qualcosa oppure rappresenta 0.

Ora tutto questo sembra molto impegnativo quando è descritto in teoria. Nella pratica diventa più semplice vedendo che: 11 in binario rappresenta 3 (2^1 + 2^0) in decimale 10000000 in binario rappresenta 128 (2 ^ 7) in decimale.

Tradurre un numero in binario manualmente, o per mezzo di un programma di computer equivale a dividerlo per le successive potenze di due così per tradurre il numero 173 in binario:

173/(2^7) =1 resto 45
45/(2^6) =0 resto 45
45/(2^5) =1 resto 13
13/(2^4) =0 resto 13
13/(2^3) =1 resto 5
5/(2^2) =1 resto 1
1/(2^1) =0 resto 1
1/(2^0) =1 resto 0

Così 173 in binario è 10101101

Lo stesso tipo di calcolo si può eseguire su ogni numero e, se lo fate regolarmente, probabilmente vi accorgerete di essere in grado di scrivere la rappresentazione binaria di un numero con una piccola serie di calcoli mentali. Tuttavia, nelle prossime fasi troverete molto utile la seguente tabella, poichè mostra la rappresentazione binaria di tutti i numeri tra zero e 255, il range che è possibile inserire in un singolo byte. Tutti i numeri binari sono dati sotto forma di otto cifre, la forma nella quale si trovano quando formano un byte.

Tabella dei numeri binari ad otto cifre da o a 255 (decimali):

decimale	binario	decimale	binario
000	00000000	000	00000000
001	00000001	016	00010000
002	00000010	032	00100000
003	00000011	048	00110000
004	00000100	064	01000000
005	00000101	080	01010000
006	00000110	096	01100000
007	00000111	112	01110000
008	00001000	128	10000000
009	00001001	144	10010000
010	00001010	160	10100000
011	00001011	176	10110000

012	00001100	192	11000000
013	00001101	208	11010000
014	00001110	224	11100000
015	00001111	240	11110000

Il modo di usare la tabella è il seguente:

1) Prendete il numero che volete tradurre in binario e confrontatelo con la colonna sulla destra. Dovete cercare il numero decimale che più si avvicina per difetto al vostro numero.
2) Scrivete le prime quattro cifre del numero binario che, nella tabella, corrisponde al decimale che avete trovato.
3) Sottraete il numero decimale nella tabella dal vostro numero. Il risultato sarà al di sotto di 16.
4) Andate sulla colonna di sinistra nella tabella e trovate il decimale restante. Le quattro cifre sulla destra del corrispondente numero binario sono le quattro cifre rimanenti che servono.

L'addizione binaria

Alcuni dei vantaggi dei numeri binari, per una ingenua bestiola quale è un computer possono essere visti istantaneamente quando andiamo ad eseguire dei semplici calcoli matematici. L'addizione è un classico esempio, poichè essa in binaria viene ridotta ad un gruppo di semplici e chiare regole. Cerchiamo di vederle in pratica, e scriviamo i seguenti due numeri binari:

1 1 0 1 0 1 1 0 (Il decimale 214)
0 1 0 0 1 0 1 1 (Il decimale 75)

Se desiderate addizionare questi due numeri, dovete rispettare le seguenti regole:
Iniziando dalla colonna sulla destra, partite addizionando le colonne corrispondenti dei due numeri

1) Se non c'è carry dalla colonna precedente:
   a) se ci sono due zeri nelle colonne corrispondenti il risultato è uno zero.
   b) se c'è un uno ed uno zero il risultato è un uno.
   c) se ci sono due uni il risultato è uno zero, con un carry di uno nella colonna successiva.
2) Se c'è un carry dalla precedente colonna:
   a) se ci sono due zeri nelle corrispondenti colonne il risultato è un uno.
   b) se c'è un uno ed uno zero il risultato è zero con un carry di uno alla colonna successiva.

c) se ci sono due uni il risultato è uno, con un carry di uno alla colonna successiva.

Usando queste regole, l'addizione avviene nel seguente modo:

```
  1 1 0 1 0 1 1 0
+
  0 1 0 0 1 0 1 1

              1  (senza carry)
            0  (con carry)
          0  (con carry)
        0  (con carry)
      0  (con carry)
    1  (senza carry)
  0  (con carry)
0  (con carry)
1  (senza carry)
```

Il risultato dell'addizione è 100100001, cioè 289 in decimale. Notate che i nostri due numeri a otto cifre hanno prodotto un risultato a nove cifre. Come con il nostro chip immaginario, questa cifra in più, siccome non può essere registrata in un singolo byte, è indicata dal flag di carry.

Dimostrazione BASIC di addizione binaria senza segno

```
10 SCNCLR
20 INPUT "NUMBER 1 (0-1111):";N1
30 N1$ = RIGHT$("0000" + MID$(STR$(N1),2),4)
40 INPUT "[CD]NUMBER 2 (0-1111):";N2
50 N2$ = RIGHT$("0000" + MID$(STR$(N2),2),4)
60 BASE = 2
70 PRINT
80 FOR I = 1 TO 4
90 PRINT TAB(I*8-2) BASE "^" 4-I
100 PRINT "[CD][CD]" TAB(I*8+4) MID$(N1$,I,1)
110 PRINT TAB(I*8+4) MID$(N2$,I,1)
120 PRINT TAB(I*8-2) "--------"
130 PRINT TAB(I*8-2) "[CD]--------"
140 PRINT "[CU][CU][CU][CU][CU][CU][CU] CU][CU]"
150 NEXT I
160 PRINT "[CD][CD][CD][CD][CD][CD]"
170 CARRY = 0
180 FOR I = 4 TO 1 STEP -1
```

```
190 T1 = ASC(MID$(N1$,I,1))-48
200 T2 = ASC(MID$(N2$,I,1))-48
210 PRINT "[CU][CU]"
220 SUM = T1+T2+CARRY
230 CARRY = 0
240 IF SUM > =BASE THEN SUM = SUM-BASE : CARRY = 1
250 PRINT TAB(I*8+3) SUM
260 IF CARRY=1 THEN PRINT TAB(I*8-8)"[CD]CARRY[CU][CU]"
270 GETKEY T$
280 NEXT I
290PRINT"[CD][CD][CD][CD]"
300 END
```

Il programma vi permette di immettere due numeri binari a quattro cifre per visualizzare il loro processo di addizione, eseguito quando premete la barra spaziatrice.

La sottrazione binaria

La sottrazione è ugualmente semplice, e possiamo usare,per illustrarla, gli stessi due numeri impiegati in precedenza. Il chip la effettua in un modo non umano , come avrete senza dubbio supposto dalla trattazione sui numeri negativi nell'ultimo capitolo.

Per sottrarre,seguite queste regole:

Operando dalla colonna più a destra:

1) Se non c'è alcun borrow (prestito) dalla colonna precedente:
   a) Se si sottrae zero da zero, o uno da uno, il risultato è zero.
   b) se si sottrae zero da uno il risultato è uno.
   c) se si sottrae uno da zero il risultato è uno con un borrow dalla prossima colonna.
2) Se c'è un borrow dalla colonna precedente:
   a) se si sottrae zero da zero il risultato è uno con un borrow dalla prossima colonna.
   b) se si sottrae zero da uno il risultato è zero.
   c) se si sottrae uno da zero il risultato è zero con un borrow dalla prossima colonna.

È piuttosto interessante notare che, se sottraete un numero più grosso da uno più piccolo usando questo metodo, arrivate ad un numero "offset" come quelli esaminati nella rappresentazione di numeri negativi vista nell'ultimo capitolo e come quelli che troverete nella prossima sezione.

Un esempio di sotrrazione che usa i due termini precedentemente addizionati, si svolgerenbbe come segue:

```
  1  1  0  1  0  1  1  0
-
  0  1  0  0  1  0  1  1

              1   (con borrow)
            1   (con borrow)
          0   (senza borrow)
        1   (con borrow)
      0   (senza borrow)
    0   (senza borrow)
  0   (senza borrow)
1   (senza borrow)
```

Il risultato della sottrazione dei due numeri (214-75) è 10001011 (139 in decimale).

Dimostrazione BASIC di addizione e sottrazione con numeri binari con segno

```
10 SCNCLR
20 INPUT "NUMBER 1 (0-1111 TO 111):";N1
30 N1$ = RIGHT$("0000"+MID$(STR$(N1) ,2), 4)
40 INPUT "[CD]NUMBER 2 (0-1111):";N2
50 N2$ = RIGHT$("0000"+MID$(STR$(N2),2),4)
60 BASE = 2
70 PRINT
80 FOR I = 1 TO 4
90 PRINT TAB(I*8-2) BASE "^" 4-I
100 PRINT "[CD][CD]" TAB(I*8+4) MID$(N1$,I,1)
110 PRINT TAB(I*8+4) MID$(N2$,I,1)
120 PRINT TAB(I*8-2) "--------"
130 PRINT TAB(I*8-2) "[CD]--------"
140 PRINT "[CU][CU][CU][CU][CU][CU][CU] [CU][CU]"
150 NEXT I
160 PRINT "[CD][CD][CD][CD][CD][CD]"
170 CARRY = 1
180 FOR I = 4 TO 1 STEP -1
190 T1 = ASC(MID$(N1$,I,1))-48
200 T2 = ASC(MID$(N2$,I,1))-48
210 PRINT "[CU][CU]"
220 SUM = T1+(1-T2)+CARRY
```

```
230 CARRY = 0
240 IF SUM > = BASE THEN SUM = SUM-BASE : CARRY = 1
250 PRINT TAB(I*8 + 3) SUM
260 IF CARRY = 1 THEN PRINT TAB(I*8-8) "[CD]CARRY[CU][CU]"
270 GETKEY T$
280 NEXT I
290 PRINT "[CD][CD][CD][CD]"
300 END
```

Il programma vi permette di immettere due numeri binari e poi illustra il processo di sottrazione del secondo dal primo quando premete la barra spaziatrice.

Numeri negativi e binari

Nella descrizione del Lawland 10000 toccammo tutti i metodi con i quali il chip creava nummeri che erano "off-set" (oltre) 10000, mentre era incapace di sottrarre. Arrivando alla numerazione binaria l'intera questione dei numeri "offset" diventa più chiara.

Come i numeri decimali, il metodo per rappresentare un numero negativo consiste nel mettere un indicatore nella colonna più a sinistra del numero con il resto del numero tramutato nella forma offset. Come con i numeri decimali, ciò riduce il massimo numero che può essere immesso in un singolo byte, con una gamma che va da -128 a + 127.

Nel caso di numeri binari, il valore da cui bisogna sottrarre un numero, per poter creare un numero negativo, è 256, cioé uno più di quanto si possa immagazzinare in un singolo byte. Se vogliamo creare il numero -1, basta sottrarre 1 da 255, lasciando 255. La rappresentazione binaria di 255 è 11111111, e se stessimo usando numeri con segno, 255 sarebbe riconosciuto come -1. Comunque non dovrete avere a che fare con tutta questa matematica, infatti per creare un numero binario negativo, c'è un semplice modo infallibile. È illustrato dall'esempio seguente:

1) Per creare un numero negativo, prendete l'equivalente positivo del numero, cioé se intendete utilizzare -97, partite con +97.
2) Traducete il numero in un numero binario ad otto cifre: 97 si traduce in 01100001.
3) Invertite tutte le cifre e vdrete che 01100001 si trasforma in 10011110.
4) Aggiungete 1 al numero. Nel nostro caso il risultato è 10011111.
5) Il risultato è la rappresentazione negativa del vostro numero, cioè 159, che è 256 meno 97.

Anche qui si nota che il chip non riconosce 10011111 come un numero negativo, è semplicemente che il tipo di operazioni aritmetiche che il chip

esegue sono fatte per produrre risultati che sarebbero razionali se il numero fosse -97 piuttosto che il 159 con cui il chip sta realmente lavorando. Se sommate 1 a 159 il risultato è 160, che ha senso.

Se noi trattassimo numeri sia positivi che negativi quel 160 sarebbe visto come -96, che pure ha senso poiché -97+1=-96. Il chip si comporterà indifferentemente poiché si stanno trattando numeri che hanno un segno più o meno. La differenza sarà nel modo in cui trattate il risultato delle operazioni del chip. Questo sarà visto in dettaglio nel capitolo 10.

Dimostrazioni BASIC di addizione e sottrazione con numeri binari con segno

```
10 SCNCLR
20 INPUT "NUMBER 1 (-1000 TO 111):";N1
30 IF N1<0 THEN T = -N1 : GOSUB 360: N1 = T
40 N1$ = RIGHT$("0000"+MID$(STR$(N1),2),4)
50 INPUT "[CD]NUMBER 2 (-1000 TO 111):";N2
60 IF N2<0 THEN T = -N2 : GOSUB 360: N2 = T
70 N2$ = RIGHT$("0000"+MID$(STR$(N2),2),4)
80 BASE = 2
90 PRINT
100 FOR I = 1 TO 4
110 PRINT TAB(I*8-2) BASE "^" 4-I
120 PRINT "[CD][CD]" TAB(I*8+4) MID$(N1$,I,1)
130 PRINT TAB(I*8+4) MID$(N2$,I,1)
140 PRINT TAB(I*8-2) "--------"
150 PRINT TAB(I*8-2) "[CD]--------"
160 PRINT "[CU][CU][CU][CU][CU][CU][CU] [CU][CU]"
170 NEXT I
180 PRINT "[CD][CD][CD][CD][CD][CD]"
190 CARRY = 0
200 FOR I = 4 TO 1 STEP -1
210 T1 = ASC(MID$(N1$,I,1))-48
220 T2 = ASC(MID$(N2$,I,1))-48
230 PRINT "[CU][CU]"
240 SUM = T1+T2+CARRY
250 LASTCARRY = CARRY
260 CARRY = 0
270 IF SUM>=BASE THEN SUM = SUM-BASE : CARRY = 1
280 OVERFLOW = LASTCARRY<>CARRY
290 PRINT TAB(I*8+3) SUM
300 IF CARRY=1 THEN PRINT TAB(I*8-8) "[CD]CARRY[CU][CU]"
310 GETKEY T$
320 NEXT I
330 PRINT "[CD][CD][CD][CD]"
```

```
340 IF OVERFLOW THEN PRINT "SIGN DIGIT IN ERROR"
350 END
360 REM **************************
370 REM 2'S COMP.
380 REM **************************
390 T2 = 0
400 FOR I = 1 TO 4
410 T2 = T2/2+(1 AND T)*8
420 T = INT(T/10)
430 NEXT I
440 T2 = 16-T2
450 T = 0
460 FOR I = 1 TO 4
470 T = T/10-(T2 AND 1)*1000
480 T2 = INT(T2/2)
490 NEXT I
500 RETURN


10 SCNCLR
20 INPUT "NUMBER 1 (-1000 TO 111):";N1
30 IF N1<0 THEN T = -N1 : GOSUB 360: N1 = T
40 N1$ = RIGHT$("0000"+MID$(STR$(N1),2),4)
50 INPUT "[CD]NUMBER 2 (-1000 TO 111):";N2
60 IF N2<0 THEN T = -N2 : GOSUB 360: N2 = T
70 N2$ = RIGHT$("0000"+MID$(STR$(N2),2),4)
80 BASE = 2
90 PRINT
100 FOR I = 1 TO 4
110 PRINT TAB(I*8-2) BASE "^" 4-I
120 PRINT "[CD][CD]" TAB(I*8+4) MID$(N1$,I,1)
130 PRINT TAB(I*8+4) MID$(N2$,I,1)
140 PRINT TAB(I*8-2) "--------"
150 PRINT TAB(I*8-2) "[CD]--------"
160 PRINT "[CU][CU][CU][CU][CU][CU][CU] [CU][CU]"
170 NEXT I
180 PRINT "[CD][CD][CD][CD][CD][CD]"
190 CARRY = 1
200 FOR I = 4 TO 1 STEP -1
210 T1 = ASC(MID$(N1$,I,1))-48
220 T2 = ASC(MID$(N2$,I,1))-48
230 PRINT "[CU][CU]"
240 SUM = T1+(1-T2)+CARRY
250 LASTCARRY = CARRY
260 CARRY = 0
```

```
270 IF SUM>=BASE THEN SUM = SUM-BASE : CARRY = 1
280 OVERFLOW = LASTCARRY<>CARRY
290 PRINT TAB(I*8+3) SUM
300 IF CARRY=1 THEN PRINT TAB(I*8-8) "[CD]CARRY[CU][CU]"
310 GETKEY T$
320 NEXT I
330 PRINT "[CD][CD][CD][CD]"
340 IF OVERFLOW THEN PRINT "SIGN DIGIT IN ERROR"
350 END
360 REM **********************
370 REM 2'S COMP.
380 REM **********************
390 T2 = 0
400 FOR I = 1 TO 4
410 T2 = T2/2+(1 AND T)*8
420 T = INT(T/10)
430 NEXT I
440 T2 = 16-T2
450 T = 0
460 FOR I = 1 TO 4
470 T = T/10+(T2 AND 1)*1000
480 T2 = INT(T2/2)
490 NEXT I
500 RETURN
```

I due programmi sopra riportati vi permettono di effettuare sia l'addizione, che la sottrazione di numeri binari con segno (semplicemente usando un segno meno per indicare un numero negativo). Nel caso della sottrazione, il programma crea automaticamente la corretta forma offset del numero.

I flag binari

Una volta presa confidenza con la numerazione binaria, moltissime cose riguardanti l'operare del chip diventano più chiare. Non andremo nei dettagli di tutti i flag, ma vale la pena notare che i flag sono più semplici da visualizzare in binario di quello che erano in decimale. Il diagramma del chip mostra che ci sono otto flag di cui uno non usato. Otto flag, ciascuno capace di registrare una singola decisione si/no, dovrebbero ricordarvi qualcosa. Infatti gli otto flag sono semplicemente gli otto bit del registro a un byte conosciuto nel chip come "status register". Nel caso dei flag, se un bit è "settato" (cioè posto a "1"), il flag sta ad indicare una risposta "sì" ad alcune domande od altro, mentre se esso è "resettato" (posto a "0") la risposta indicata è "no". È utile fissare nella vostra mente questo uso un poco strano

dei termini "set" e "reset", poichè li troverete largamente usati nei "testi" di "codice" macchina "per riferire lo stato di singoli bit, sebbene "on" e "off" siano di più facile comprensione.

La "moltiplicazione" e la "divisione binaria"

Come un numero decimale può essere moltiplicato o diviso per la sua base semplicemente spostando le sue cifre verso sinistra o verso destra, così può essere fatto anche con un numero binario. Se, seguendo il discorso sulla moltiplicazione e divisione nel capitolo scorso, avete compreso il problema, non c'è nulla da aggiungere, se non due esempi di esecuzione dei procedimenti. Per semplificarli, useremo un numero binario con una sola cifra settata a "1". Tra tutti quei numeri che soddisfano questa condizione, noi abbiamo scelto il decimale 16, che espresso in forma binaria a otto cifre è 00010000.

Per moltiplicarlo per due, spostiamo le cifre verso sinistra, avendo come risultato 00100000, cioè 32 in decimale. Se dovessimo eseguire questo spostamento verso sinistra su di un numero contenuto nell'accumulatore, fino al punto che l'unico "1" contenuto nel byte fuoriuscisse dal byte stesso, il risultato sarebbe "zero, con l'"1" immagazzinato nel flag di carry.

La divisione è realizzata facendo scorrere le cifre nella direzione opposta, verso destra, e il risultato di un tale spostamento porta a 00001000, cioè 8 in decimale. Se il procedimento viene eseguito ripetutamente su un numero contenuto nell'accumulatore, sarà il tipo di comando usato (vedere Capitolo 13) a decidere se inserire l'"1" nel flag di carry oppure perderlo.

CAPITOLO 4

L'USO DI QUESTO TESTO

Per un momento, lasciamo da parte la teoria di un chip e, prima di entrare nei dettagli del codice macchina, diamo una occhiata ai metodi che si useranno in tutto il resto di questo testo. I due argomenti principali che vorremmo chiarirvi sono il sistema di numerazione esadecimale ed il modo in cui i piccoli programmi dimostrativi devono essere immessi nella vostra macchina.

L'uso dei numeri esadecimali

Prima di esaminare il metodo di approccio ai molti esempi di codice macchina inseriti nel libro, forse sarebbe meglio spiegare perchè tutti i valori numerici usati nei programmi e nelle spiegazioni sono riportati sottoforma di numeri esadecimali, cioè vale a dire numeri espressi in base 16.

I numeri esadecimali, similmente ai decimali, hanno una gamma di 16 possibili cifre, i numeri da 0 a 9 ed in più le lettere da A a F, per occupare le 6 rimanenti possibilità. La tabella seguente mostra alcuni numeri decimali con i corrispondenti valori esadecimali:

D	H	D	H
0	0	32	20
1	1	48	30
2	2	64	40
3	3	240	F0
4	4	255	FF
5	5	256	100
6	6	512	200
7	7	768	300
8	8	1024	400
9	9	2048	800
10	A	4095	FFF
11	B	4096	1000
12	C	8192	2000
13	D	16384	4000
14	E	32768	8000
15	F	61440	F000
16	10	65535	FFFF

Per tradurre da decimale in esadecimale, tutto quello che dovete fare è dividere per le potenze di 16, e continuare fino a quando il resto non è più piccolo di sedici. Prendiamo, ad esempio il numero 3781:

1) Iniziamo col dividerlo per 16^2 (16^3 è 4096 e di conseguenza troppo grosso) che dà il risultato di 14 col resto di 197.
2) La prima cifra del numero esadecimale è quindi E.
3) Dividendo 197 per 16^1 (cioè 16), troviamo 12 con un resto di 5.
4) La seconda cifra del numero esadecimale è C.
5) La cifra finale è divisa per 16^0 (cioè 1) ed, ovviamente, il risultato è 5.
6) La cifra finale del numero esadecimale è 5.
7) Mettendo assieme le cose, troviamo che la rappresentazione esadecimale del numero decimale 3781 è EC5.

Ma perchè creare tale complicazione? La risposta è che il sistema esadecimale è un giusto compromesso tra quello che ha senso per un computer, e quello che ha senso per un essere umano. Al livello di un singolo byte, l'intero range di valori 0-255 può essere espresso con due sole cifre esadecimali 00-FF. Soltanto che, ogni cifra si riferirà a quattro bits del byte e più precisamente: il primo carattere si riferirà ai quattro bit di valore piùalto ed il secondo carattere si riferirà ai quattro bit di valore più basso. Una volta presa l'abitudine ad operare con i numeri esadecimali e acquistata la familiarità di cosa significhino in binario le 16 cifre disponibili, comincierete ad intravedere, quando osservate un numero esadecimale, il formato del corrispondente binario di esso.

Ad un livello più alto, quello dell'intera macchina, il computer opera in unità di 256, come già abbiamo visto. La numerazione decimale si rivela veramente poca cosa dal punto di vista del computer. Il numero decimale 32768 sembra uno dei tanti numeri, ma in esadecimale è espresso come 8000 e quindi assume un valore particolare, infatti numerose separazioni tra diverse attività di un computer si collocano ad indirizzi che in decimale significherebbero molto poco.

È per questa ragione che i programmatori in codice macchina, quasi universalmente lavorano in esadecimale ed è questo che noi vorremmo raccomandarvi. Naturalmente, all'inizio vi sentirete impacciati e cercherete aiuto in tabelle o calcolatori che siano capaci di effettuare tali trasformazioni, ma una volta presa la mano vi sorprenderete dei risultati e non rimpiangerete la fatica.

Immissione delle routines dimostrative nel vostro computer

Tutti gli esempi in assembly riportati su questo libro, sono stati realizzati usando il monitor in codice macchina disponibile sul computer.
Vediamo ora come procedere nelle varie tecniche di programmazione.

1) REGISTRI:1 e REGISTRI:2

Lo scopo di questo libro è aiutarvi a comprendere gli effetti dei comandi in codice macchina, e molte istruzioni in codice macchina interessano lo stato dei registri interni del chip; una delle più importanti informazioni fornite nei listati dimostrativi è proprio lo stato dei registri prima e dopo che un codice macchina è stato caricato ed eseguito. Quando userete le dimostrazioni troverete che il monitor visualizzerà sempre il contenuto dei registri, sia prima di eseguire un programma dimostrativo sia quando esso è terminato. Per visualizzare i registri in ogni altro momento basterà semplicemente digitare R seguito da RETURN. Ricordate che se, nel corso degli esperimenti con gli esempi inseriti nel testo, volete alterare il valore al quale punta uno dei registri, tutto quello che dovete fare è portare il cursore sopra di esso, battere il valore esadecimale voluto e premere RETURN.

2) MEMORIA:1 e MEMORIA:2

Molte delle routine di dimostrazione fanno uso di valori immagazzinati in memoria, come vorrebbe un programma operativo in codice macchina. Piuttosto che usare locazioni di memoria a caso, abbiamo selezionato otto locazioni di memoria da usare. Quattro di loro cadono nei primi 256 bytes di memoria, cioè nella memoria zero page", partendo dall'indirizzo esadecimale $8B cioè 139 in decimale. Le altre quattro locazioni sono nel banco principale di memoria a $1C8B, cioè 7307 in decimale. In tutti i listati riportati nel libro, queste locazioni di memoria sono visualizzate prima e dopo che si esegua il programma dimostrativo. Quando immettete ed eseguite questi programmi di esempio, vi rammentiamo che potete visualizzare sempre le locazioni di memoria, usando i comandi >8B e >1C8B per specificare le due aree. In entrambi i casi solo i primi quattro bytes, degli otto visibili, saranno rilevanti e questi sono gli unici che sono mostrati nei listati. Prima di eseguire qualcuno degli esempi del libro controllate sempre che il contenuto delle locazioni di memoria sia uguale a quello indicato dai listati, se volete alterarli, battete semplicemente sopra ciò che c'è sullo schermo e premete RETURN.

3) LINEE DI PROGRAMMA

Le linee effettive del programma dimostrativo verranno visualizzate automaticamente sul vostro schermo. I programmi iniziano sempre all'indirizzo $1C00, che è nella memoria sia del C16 che del Plus 4, e una prima linea di programma potrebbe essere per esempio, A 1C00 LDX $8B. Quando premete RETURN dopo questa prima linea, il monitor rivisualizzerà ciò che avete immesso e poi stamperà: A 1C02 invitandovi ad immettere un'altra linea del

programma ed, automaticamente, fornendovi l'indirizzo in memoria al quale essa si porrà. Tutto quello che dovete fare è immettere l'istruzione da porre a quell'indirizzo. Se non ci sono più linee di programma premete semplicemente RETURN senza immettere alcuna istruzione.

L'esecuzione dei programmi

Per eseguire il programma inserito nel listato raccomandiamo di osservare la seguente procedura, che è studiata apposta per evitare l'insieme di battiture che dovreste fare:

a) Prima di inserire il comando MONITOR, battete la linea seguente: POKE 55,0 : POKE 56,28 : CLR L'effetto di questo è ridurre l'ammontare di memoria disponibile per il BASIC normale così che niente in BASIC possa interferire con il programma in codice macchina che state introducendo in memoria. Le due locazioni interessate dalle istruzioni POKE, infatti, sono conosciute con il nome di variabili di sistema chiamate RAMTOP, che dicono al vostro computer dove è situato normalmente il limite della sua memoria. Il nuovo valore dato a questa variabile con le due istruzioni POKE è tale che essa si colloca sotto l'area da voi usata per immagazzinare il vostro codice macchina. I proprietari di Plus 4 normalmente non avrebbero bisogno di abbassare la RAMTOP in un modo così drastico ma per semplicità di testo è più semplice usare un solo metodo per settare sia il Plus 4 che il C16.

b) Immettete:
KEY 1,"R"+CHR&(13)+CHR$(13)+"> 8B" +CHR$ (13)
KEY 2,"A 1C00"
KEY 3,"CHR$(13)+"G 1C00"+CHR$(13)+"> 8B"+CHR$
(13)+"1C8B"+CHR $(13)
Quanto avete fatto, è abilitare tre dei vostri tasti di funzione per rendere più facile l'ingresso e l'uso dei programmi in codice macchina. Quando lavorate col codice "monitor", premendo il tasto di funzione 1 prima si visualizzeranno i contenuti dei registri e, poi, le due aree di memoria a $8B e $1C8B usate per immagazzinare valori. Il tasto di funzione 2 farà iniziare il programma immettendo A C100 prima che battiate la prima istruzione, togliendovi la necessità di ricordare da dove deve partire il programma. Alla fine, quando tutto il programma è stato immesso, il tasto di funzione 3 lo farà svolgere e poi visualizzerà nuovamente le due aree di memoria. I registri sono visualizzati automaticamente alla fine di ogni programma in modo che non sia necessaria nessuna istruzione aggiuntiva.

c) Immettete:
MONITOR
per far partire il codice macchina monitor.

d) Premete il tasto di funzione 1 per visualizzare i contenuti dei registri. Controllate che i valori mostrati siano uguali a quelli inseriti nel listato e, se ciò non si verifica, cambiateli.
e) Premete il tasto di funzione 2 e poi immettete la prima istruzione in linguaggio assembly. Quando ciascuna istruzione è terminata, premete RETURN e poi immettete pure l'istruzione successiva. Non avete bisogno di premere il tasto RETURN due volte per indicare che il programma è terminato in quanto il tasto di funzione 3 inizia la sua opera stampando il carattere RETURN.
f) Quando avete terminato di inserire il programma, premete il tasto di funzione 3. Il programma sarà eseguito ed, al termine, sarà visualizzata la condizione dei registri e delle due aree di memoria.
g) Quando avete terminato le sperimentazioni con il codice macchina e ritornate al BASIC, premendo il tasto X, è bene anche premere il pulsante di reset per riallocare la RAMTOP.

Routines di dimostrazione con un altro assembler. Il codice macchina monitor disponibile sul vostro computer è una facilitazione potente e conveniente ma manca delle sofistificazioni dei più grossi programmi assembler commerciali. Potete, quindi, essere tentati di comprare per il vostro computer un programma assembler in grado di darvi una maggiore flessibilità nell'esecuzione e nell'uso delle dimostrazioni del codice macchina. Prima di procedere all'acquisto ricordate che, per esservi di qualche aiuto nell'esecuzione delle dimostrazioni, il vostro package assembler deve essere in grado di:

1) Accettare un programma in linguaggio assembly ed assemblarlo in memoria.
2) Guardare dentro un programma in codice macchina usando la funzione trace.
3) Settare il valore di singole locazioni di memoria tramite una facilitazione codice macchina monitor. Stabilito che queste funzioni operano correttamente, è possibile seguire l'esecuzione di un programma, istruzione per istruzione, attraverso la visione degli effetti sui registri e così via. Tuttavia, alcune routines possono causare delle difficoltà, poichè trace non sempre è efficace.

CAPITOLO 5

INTRODUZIONE ALL'INDIRIZZAMENTO

Per il resto del libro abbandoneremo la teoria e passeremo alla programmazione pratica del chip 7501, che opera sui C16 e Plus 4. Torniamo a rivedere il concetto che è stato introdotto nel primo capitolo cioè i vari metodi con cui può essere detto al chip di trovare un numero su cui effettuare una determinata azione.

Supponiamo per il momento di voler compiere qualcosa con un valore contenuto in memoria. Il byte che contiene tale valore avrà un posto qualsiasi in memoria tra il byte zero ed il byte 65535 (per un chip a otto bit come il 7501), e quel posto è conosciuto come il suo indirizzo. Fin qui tutto bene, ma come possiamo descrivere quell'indirizzo al chip?

Una semplice risposta potrebbe essere che, siccome l'indirizzo è un numero, l'istruzione in codice macchina dovrebbe semplicemente contenere il numero. Questo è perfettamente ragionevole e, infatti, è uno dei modi, non il solo, con cui un indirizzo è presentato al chip.

Se è lasciato totalmente al programma il descrivere un indirizzo, troviamo che siamo molto limitati in quello che possiamo realizzare.

Supponiamo, per esempio, di conoscere il numero su cui vogliamo operare, in realtà dobbiamo collocarlo in memoria e poi dire al chip dove trovarlo. Non avrebbe molto più senso avere la possibilità di dire al chip trascurando gli indirizzi, "prendi questo numero dal programma ed opera su di esso". Chiaramente ciò salverebbe tempo e fatica, ed aggiungeremmo un nuovo "modo di indirizzamento" che ci permette di scrivere i numeri con cui vogliamo operare direttamente dentro il nostro programma.

Sarebbe molto difficoltoso programmare senza l'aiuto di una delle tabelle di valori conosciute come "array", di solito usate in BASIC. Quando, in BASIC, ci rivolgiamo ad un array con una espressione come:

```
X = ARRAY(12)
```

noi ci affidiamo al BASIC Interpreter (il programma in codice macchina che esegue il linguaggio BASIC) per trovare la tabella di valori chiamata ARRAY e, avendo trovato l'inizio di tale tabella, guardare il tredicesimo valore (ricordate, sono numerate da zero). Se non ci fossero array, avremmo avuto tutta una serie di variabili chiamate, per esempio, A1, A2, A3,..... Programmare in questo modo sarebbe estremamente scomodo.

Chiaramente l'utilità delle tabelle di valori non è limitata al linguaggio BASIC infatti esse rappresentano una tecnica di programmazione che ogni linguaggio per computer utilizza. In codice macchina non possiamo semplicemente stabilire un array e poi aspettare di essere in grado di chiamarlo per nome, ma quello che possiamo fare è mettere da parte un'area di memoria ed immagazzinarci una sequenza di numeri. Poi, idealmente, dobbiamo essere in grado di indicare l'inizio della tabella e dire al chip "Io voglio il decimo valore della tabella che inizia qui". Questo certamente deve essere più veloce e più semplice che dover calcolare l'indirizzo di ogni posizione nella tabella; possiamo così aggiungere un terzo tipo di indirizzamento.

È tutto ora? Beh no, non ancora. Fino a questo punto possiamo specificare un valore nel programma o specificare un indirizzo al quale un valore può essere trovato, ma cosa succede se vogliamo scrivere o programmi corti che saranno eseguiti molte volte od un "loop" dentro un programma, specificando un diverso indirizzo dove ogni volta trovare un valore. Chiaramente non possiamo cambiare il programma ogni volta e abbiamo bisogno di qualcosa di simile ad una variabile BASIC, il cui valore cambi durante l'esecuzione del programma.

Abbiamo già visto che l'equivalente in codice macchina dell'immissione di un valore in una variabile BASIC è l'immagazzinamento di esso in una locazione di memoria. Da ciò si può trarre, quindi, che, se vogliamo attribuire diversi indirizzi alla stessa parte di programma, non possiamo solamente specificarli, ma dobbiamo chiedere al programma di osservare una variabile (cioè una locazione di memoria) e prendere da essa un indirizzo. Un'altra parte del programma si occuperà di cambiare di volta in volta l'indirizzo.

Alla fine, quando giungiamo all'uso di istruzioni conosciute come "jump" (salti), che sono equivalenti ai GOTO e GOSUB del BASIC, potrebbe esserci utile un'altra forma di indirizzamento. Poichè i programmi in codice macchina possono essere situati quasi dappertutto nella memoria del computer, si possono creare difficoltà usando istruzioni che dicono al chip di "saltare" alla istruzione in codice macchina al tale indirizzo. Se il programma è trasferito in un'altra parte di memoria, o se parti del programma sono spostate per fare posto ad altre istruzioni, queste istruzioni "jump" diventano significative. Un metodo per aggirare il problema è usare un altro modo di indirizzamento, conosciuto come "relative addressing" (indirizzamento relativo) dove, invece di specificare l'esatto indirizzo della parte del programma che deve essere raggiunta, al programma viene detto qualcosa come salta all'istruzione il cui indirizzo è 100 bytes avanti rispetto all'indirizzo di questa istruzione. Ora, se il programma viene spostato, a condizione che le due istruzioni distino ancora nella stessa misura, l'indirizzamento indicato avrà ancora tutta la sua validità.

Così ora noi abbiamo cinque forme differente per definire un indirizzo. Si potrebbe sottolineare però che queste sono solo descrizioni vaghe e generalizzate, ma vedremo i modi di indirizzamento anche nei dettagli. Comunque, non andate oltre a leggere fino a che non riuscite a capire perchè sono

necessari questi cinque diversi modi per descrivere un indirizzo ed il loro operato.

Indirizzi zero-page e whole memory

Fino ad ora, abbiamo supposto che i comandi possano essere usati per specificare un indirizzo qualunque compreso tra 0 e 65535, sebbene la forma con la quale un indirizzo è specificato può cambiare. La ragione dei limiti particolari nel range è dovuta al fatto che molti indirizzi sono specificati con un numero a due byte del tipo descritto alla fine del Capitolo 3.

C'è, comunque, una seconda classe di modi di indirizzamento che, sebbene molto più limitati nel loro campo d'azione, sono usati più spesso per la loro velocità. La velocità è, naturalmente , una delle mete costanti del programmatore in codice macchina e ogni operazione che il chip deve eseguire rallenta l'esecuzione del programma. Quando un indirizzo, in qualunque modo sia specificato, richiede due byte, il chip deve eseguire alcune operazioni interne per trattare il numero a due byte utilizzato. Alcuni indirizzi, comunque, eliminano questa fase del processo poichè richiedono soltanto un byte per specificarli. Siccome un byte può immagazzinare ogni valore da 0 a 255, è ovvio che i soli indirizzi che possono essere specificati con un solo byte, senza che il chip debba fare operazioni aggiuntive per arrivare all'eventuale posizione (come nell'indirizzamento relativo) sono indirizzi compresi tra 0 e 255.

Questi sono conosciuti come indirizzi zero-page perchè è convenzione trattare la memoria come un insieme di blocchi di 256 bytes di memoria chiamati pagine. Per tener conto del potenziale in velocità che procura un indirizzo ad un byte, viene fornita tutta una nuova classe di modi di indirizzamento, che permette al chip di fare completamente a meno della necessità di considerare, per un indirizzo, due byte. Senza alcuna sorpresa, questi modi di indirizzamento sono chiamati modi "zero-page". Essi sono usati spesso dai programmatori esperti, ed il sistema operativo del Commodore 64 si avvale molto di essi. L'informazione viene situata in zero-page in modo che le attività interne del computer occupino il minor tempo possibile.

Il 7501 è provvisto di un'intera gamma di istruzioni che richiedono la forma di indirizzamento zero-page; tali istruzioni sono conosciute come istruzioni zero-page. Qualche volta esse ottengono lo stesso scopo di un altro comando che può essere diretto a qualsiasi indirizzo nella memoria, indifferentemente che sia o meno nei primi 256 bytes. La convenienza del suo uso consiste solo nel fatto che la forma zero-page è più veloce.

Notate, come sempre, che quando una istruzione esiste "in due forme", quello che si vuole dire è che, malgrado ogni similitudine tra l'operato delle due istruzioni, esse sono totalmente differenti quando si conosce il comando che si deve dare al chip. Non potete mettere un indirizzo ad un byte alla fine di

una istruzione whole memory e credere che il chip riconosca che solo un byte è necessario. Se usate la forma whole memory di un comando, il chip tratterà i due bytes seguenti come indirizzo, sebbene quell'indirizzo cada in zero-page.

La forma dell'indirizzo

Tutti i modi di indirizzamento visti in precedenza, eccetto uno (modo immediato), richiedono che sia specificato un indirizzo, la posizione di un byte in memoria. Non tutte le istruzioni in codice macchina in realtà richiedono un indirizzo ottenibile dal programma, dopo alcune operazioni su locazioni fisse come l'accumulatore od uno dei flag nello status register. Quando deve essere specificato un indirizzo, sarà in una delle due forme, concordemente all'istruzione, che sarà rivolta o alla zero-page od a tutta la memoria.

1) L'indirizzo per una istruzione zero-page.
Abbiamo già notato che le istruzioni zero-page sono più veloci, per la semplice ragione che ogni indirizzo può essere composto da un solo byte. Di conseguenza, una istruzione zero-page avrà questa forma (tralasciando le aggiunte extra come quelle necessarie nelle istruzioni indicizzate):

ISTRUZIONE ZERO-PAGE <NUMERO A UN BYTE>
(specificando un indirizzo compreso tra 0 e 255)

2) L'indirizzo per una istruzione whole memory.
Una volta che ci muoviamo fuori dei primi 256 bytes di memoria, un singolo byte non è più capace di specificare un indirizzo, ma bisognerà usare due byte. Usando due byte, il numero più grosso che si può rappresentare è 255*256 (con l'"high byte") + 255 (con il "low byte"), che equivale a 65535. Ciò è pienamente sufficiente, poichè il chip 6510 è solo capace di riconoscere la presenza dei byte di memoria numerati da zero a 65535.
Se lavorate direttamente in codice macchina, piuttosto che usare l'assembler, probabilmente alterando un programma in codice macchina inserito in memoria, una importante condizione è che, nell'immettere un indirizzo nel programma, i due byte devono essere back-to-front. Cioè, se l'indirizzo da specificare è, per esempio, 1024, che equivale a 4*256 + 1, logicamente ci aspetteremmo che i due byte necessari per specificare l'indirizzo in esadecimale siano 04 seguito da 01. CIÒ NON È VERO. Per ragioni che non considereremo, l'indirizzo verrà specificato come 01 seguito da 04. La regola è, quindi:

QUANDO SPECIFICHIAMO UN INDIRIZZO A DUE BYTE, IL "LOW BYTE" VA IMMESSO PER PRIMO, L'"HIGH BYTE" PER SECONDO.

Questo strano capovolgimento è la maggior fonte di confusione per gli inesperti in codice macchina e frequentemente fa cadere in fallo persino i programmatori esperti. Gli altri valori a due byte nei vostri programmi non devono essere rovesciati, infatti rovosciandoli perderebbero il proprio senso. Quando trovate, nei vostri programmi, degli inespicabili problemi, è utile che uno dei primi controlli che eseguite sia sulla forma degli indirizzi contenuti nel programma. Se, per esempio, intendete riferirvi all'indirizzo 0401 e, nel programma in codice macchina, appare come 0401, in realtà vi siete rivolti all'indirizzo 0104. Ma ricordate che, se utilizzate l'assembler, tutto questo è fatto automaticamente infatti ogni volta che specificate un indirizzo a due byte l'assembler invertirà l'ordine dei due bytes prima di immettere l'indirizzo nel programma.

Il formato di una istruzione whole memory in codice macchina è,quindi:

ISTRUZIONE WHOLE MEMORY <LOW BYTE> <HIGH BYTE>

ed in linguaggio assembly:

ISTRUZIONE WHOLE MEMORY <HIGH BYTE> <LOW BYTE>

3) Attraversando il confine zero-page.

Chiaramente, se un indirizzo è oltre i primi 256 byte di memoria, un indirizzo ad un solo byte non può essere usato per specificarlo. Il contrario, comunque, non è vero. Se, per qualche strana ragione volete usare una istruzione whole memory per riferirvi a qualcosa inserito nella zero-page, l'indirizzo dovrà essere specificato con due bytes, con l'"high byte" settato a zero.

È essenziale che queste regole riguardanti la lunghezza degli indirizzi siano seguite fedelmente, soprattutto se operate direttamente in codice macchina, piuttosto che usare l'assembler. Se usate la forma zero-page di una istruzione e poi inserite un indirizzo a due byte dopo di essa, il 7501 considererà il primo byte del numero come indirizzo. Il secondo byte sarà trattato come l'inizio della istruzione seguente. Se, d'altra parte, usate la forma whole memory di una istruzione ed immettete un solo byte di indirizzo, il chip prenderà come indirizzo il vostro byte ed il primo byte dell'istruzione seguente. In entrambi i casi sarà irreparabilmente in discordanza col programma che avrete scritto, ed interpreterà comandi che mai intendevate creare o entrerà in aree non richieste.

CAPITOLO 6

MODI DI INDIRIZZAMENTO NEI DETTAGLI

Vista la teoria che sta dietro ai metodi di indirizzamento, torniamo nello specifico, analizzando i 14 modi con cui un indirizzo può essere specificato con il chip 7501. Nel leggere questo capitolo, comunque, si deve ricordare che non tutte le istruzioni possono essere usate con ogni forma di indirizzamento. Il comando che ha maggiore disponibilità di indirizzamento, è LDA, ne ha otto, gli altri ne hanno solo uno.

Parte 1: forme non-memory

Non tutti i numero che dobbiamo manipolare sono specificati sotto forma di un indirizzo in memoria; alcune forme conterranno l'indirizzo dentro di esse o specificheranno direttamente il numero:

Indirizzamento intrinseco ovvero l'indirizzo incorporato nel comando.

Il comando TAX, per esempio consiste nel "Trasferire il contenuto dell'accumulatore nel registro X". Poichè entrambi questi registri sono dentro il chip, essi non hanno un indirizzo come tali. Il chip quando incontra il comando TAX, sa esattamente dove prendere il numero che deve essere manipolato e dove deve metterlo, questo è "intrinseco" nel comando.

I comandi che usano il modo di indirizzamento intrinseco compariranno da soli, non avendo bisogno di indirizzi o numeri che li seguano.

Indirizzamento dell'accumulatore ovvero un richiamo incorporato all'accumulatore.

Ci sono quattro comandi, ASL, LSR, ROL e ROR (tutti questi saranno spiegati nel Capitolo 13) che usano una forma speciale di indirizzamento intrinseco chiamata "indirizzamento dell'accumulatore". Tutti questi comandi hanno come scopo l'effettuare un preset sui contenuti dell'accumulatore, senza dover specificare altro.

Anche questa volta, i comandi che impiegano il modo di indirizzamento dell'accumulatore non necessitano di indirizzi.

Indirizzamento immediato ovvero specificazione di un valore nel programma.

Quando una istruzione fa uso dell'indirizzamento immediato, comparirà nella forma:

ISTRUZIONE <NUMERO>
(compreso tra 0 e 255)

Il chip, quando incontrerà l'istruzione, del tipo LDA (LoaD Accumulator), prenderà il numero che segue il comando nel programma e lo collocherà nell'accumulatore. In un programma in linguaggio assembly, il numero sarà preceduto dal simbolo "#", che è il segno distintivo del modo immediato.

Parte 2: indirizzamento whole memory

Indirizzamento assoluto ovvero specificazione di una posizione in memoria

Qui la CPU è istruita ad agire su di un valore che è contenuto in un byte di memoria il cui indirizzo è specificato dal programma in codice macchina. Quando l'indirizzo viene dato alla CPU, questa va direttamente a quel byte per trovare il numero sul quale deve operare. La forma dell'istruzione per l'indirizzamento assoluto è:

ISTRUZIONE <NUMERO A DUE BYTE> (compreso tra 0 e 65535)

Indirizzamento assoluto indicizzato ovvero raggiungimento di parte di una tabella

Abbiamo già discusso a proposito del concetto dell'indirizzamento indicizzato che permette di ottenere la costruzione di una tabella e di specificare poi un particolare valore dentro quella tabella.

In realtà c'è da notare che ciò è una forma di indirizzamento assoluto, in quanto la posizione di partenza della tabella è data direttamente dal programma nello stesso modo in cui un indirizzo è normalmente specificato nel modo di indirizzamento assoluto. Questo, comunque, è solo metà del processo, poichè la posizione del byte desiderato dentro la tabella deve altresì essere data. Questo "indice" sarà contenuto in uno o nell'altro dei registri X e Y del 7501 (ciò dipenderà dal formato dell'istruzione). La forma dell'indirizzamento indicizzato è:

ISTRUZIONE <INDIRIZZO A DUE BYTE>,
<NOME DEL REGISTRO>

Così, se l'indirizzo a due byte fosse $0401 (1025 in decimale) ed il registro interessato dall'istruzione fosse il registro X, che in quel momento contiene il

valore decimale 16, il byte a cui il chip si rivolgerà sarà $0401 + $10, cioè $0411.

Indirizzamento indiretto ovvero ritrovamento di un indirizzo da un indirizzo

Abbiamo già notato che esistono molti casi nella programmazione dove non conosceremo necessariamente l'indirizzo che deve essere inserito, fino a quando il programma non viene eseguito. In questo caso, dobbiamo essere in grado di guardare ad uno specifico byte nella memoria e prendere da esso non il numero che vogliamo, ma l'indirizzo al quale possiamo trovare il valore desiderato. La forma di un modo di indirizzamento indiretto è:

ISTRUZIONE (< INDIRIZZO A DUE BYTE >)

In un programma in linguaggio assembly l'indirizzo potrebbe essere inserito tra le parentesi. Questo significa "prendi il contenuto di questo indirizzo e usalo per qualcosa". Se il numero tra parentesi è nella posizione di un indirizzo le parentesi significano "prendi il contenuto di questo indirizzo e usalo come indirizzo".

Una cosa da notare a proposito dell'indirizzamento indiretto è che ciò che si deve prendere dalla memoria prima di tutto sono i due byte rappresentanti un indirizzo. L'indirizzo iniziale, tra parentesi deve quindi indicare il primo dei due byte che devono essere nell'ordine solito (LOW/HIGH) di un indirizzo.

Indirizzamento relativo

Questa è la forma di indirizzamento che è usata qualche volta con le istruzioni "jump" che sono comandi usati per dire al chip di saltare a un'altra parte del programma in codice macchina. Gli indirizzi relativi sono numeri ad un solo byte e sono diversi da tutte le altre forme, poichè essi sono trattati dal chip come numeri con segno e quindi possono essere sia positivi che negativi. Nel capitolo sui numeri binari vedemmo che, invece di usare una gamma da 0 a 255, i valori che possono essere espressi vanno da -128 a +127.Come ogni numero con segno, il bit più significativo indicherà se il valore è negativo o no. Se avete seguito la descrizione dei numeri con segno nei capitoli precedenti, l'indirizzamento relativo non vi incuterà alcun timore. Se non avete ancora le idee completamente chiare sull'uso dei numeri con segno, tornate alle trattazioni precedenti.

La forma di un'istruzione in modo di indirizzamento relativo è:

ISTRUZIONE < NUMERO CON SEGNO AD UN SOLO BYTE >

Parte 3: modi zero-page

Assoluto ovvero specificazione di un indirizzo in zero-page

Nessuna differenza negli effetti dalla versione non zero-page. L'istruzione zero-page specifica un indirizzo ad un solo byte al quale è situato il numero da trovare.

La forma di una istruzione assoluta zero-page è:

ISTRUZIONE <INDIRIZZO AD UN SOLO BYTE>

Indicizzato zero-page ovvero accesso ad una tabella in zero-page

Anche questa volta, la differenza dalla forma non-zero-page è che invece di un indirizzo a due byte è necessario, per l'inizio della tabella, un indirizzo ad un solo byte. Notate comunque che la forma zero-page del modo di indirizzamento indicizzato richiede che l'indirizzo eventuale, a cui si arriva aggiungendo il contenuto del registro X o Y (ciò dipende dal comando) allo specifico indirizzo di partenza, deve cadere nella zero-page.

La forma di un comando zero-page indicizzato è:

ISTRUZIONE <INDIRIZZO AD UN SOLO BYTE>, <NOME DEL REGISTRO> (sia X che Y).

Pre-indicizzato indiretto ovvero l'uso di una tabella per ottenere un indirizzo

In sostanza l'indirizzamento pre-incidizzato indiretto è una forma del normale indirizzamento indicizzato. Ciò significa che, invece di prendere un valore da una posizione in una tabella, esso prende un indirizzo a due byte dalla tabella e poi preleva il numero a quell'indirizzo. Il registro usato per l'indirizzamento pre-indicizzato indiretto è sempre il registro X.

La forma di un comando pre-indicizzato indiretto è:

ISTRUZIONE <INDIRIZZO AD UN SOLO BYTE,X>

Post-indicizzato indiretto ovvero indicazione di un indirizzo precedentemente ottenuto dalla memoria

È l'ultima fforma, molto inusuale, di indirizzamento in zero-page. Questa volta viene prelevato un indirizo a due byte da una posizione specifica nella memoria zero-page, poi il contenuto del registro Y è sommato a questo indirizzo. Questa forma finale dell'indirizzo è quindi usata per prelevare il

valore desiderato dalla memoria.

La forma di un comando indiretto post-indicizzato è la seguente:

ISTRUZIONE <INDIRIZZO AD UN SOLO BYTE>, Y

Conclusione:

Ora conoscete gli 11 possibili modi di indirizzamento che il chip 7501 dispone anche se in realtà ce ne sono 13, poiché i comandi indicizzati possono essere usati sia col registro X che con quello Y. I formati visti mostrano il significato di ognuno dei modi di indirizzamento in un programma in lingua assembly.

Se volete operare direttamente in codice macchina dovrete scegliere i modi di indirizzamento ed i comandi di forma corretta delle istruzioni dalle tabelle inserite nel libro. Inoltre, viene data di seguito una lista dei formati in linguaggio assembly affinché nella lettura di un programma in linguaggio assembly possiate consultare velocemente tale tabella se siete insicuri di quale modo di indirizzamento usi un particolare formato di istruzione. Nel listato, è inserito un codice immaginario, chiamato MEM, sia al posto dell'istruzione reale, poiché non c'è un comando che sia in grado di usare tutti i modi di indirizzamento:

MEM # <NUMERO AD UN BYTE>:	IMMEDIATO
MEM <INDIRIZZATO A DUE BYTE>:	ASSOLUTO
MEM <INDIRIZZO AD UN BYTE>:	ZERO PAGE (ASSOLUTO)
MEM:	INTRINSECO
MEM A:	ACCUMULATORE
MEM (<INDIRIZZO AD UN BYTE>,X):	PRE-INDICIZZATO INDIRETTO
MEM (<INDIRIZZO AD UN BYTE>),Y:	POST-INDICIZZATO INDIRETTO
MEM <INDIRIZZO AD UN BYTE>,X:	ZERO PAGE INDICIZZATO X
MEM <INDIRIZZO AD UN BYTE>,Y:	ZERO PAGE INDICIZZATO Y
MEM <INDIRIZZO A DUE BYTE>,X:	ASSOLUTO INDICIZZATO, X
MEM <INDIRIZZO A DUE BYTE>, Y:	ASSOLUTO INDICIZZATO, Y
MEM <NUMERO CON SEGNO AD UN BYTE>:	RELATIVO
MEM (<INDIRIZZO A DUE BYTE>):	INDIRETTO

Però solo voi potete comprendere la differenza tra i modi di indirizzamento. Fino a che ciò non avverrà, avrete sempre difficoltà di programmazione.

Una volta imparati i comandi possibili, la programmazione non diventa altro che selezionare il giusto modo di indirizzamento per realizzare un particolare comando.

CAPITOLO 7

CARICAMENTO ED IMMAGAZZINAMENTO CON A,X, ED Y

Con tutto quanto acquisito fino ad ora, rivolgersi al codice macchina effettivo, che è il vero scopo del libro. Inizieremo con uno dei gruppi di istruzioni più facilmente comprensibili, quelle che caricano valori nei registri del chip e quelle che prendono valori da essi e li immagazzinano in qualche altro luogo.

Le istruzioni Load : LDA,LDX ed LDY

Funzione: copiare un valore da una specifica locazione in uno dei registri della CPU.

Paragonando quello che stiamo facendo al BASIC, il caricare un registro è piuttosto simile al settare il valore di una variabile con una affermazione del tipo:

LET X = 10

In BASIC, la ragione solita per cui si immette un valore in una variabile è che, in tal modo, si possono eseguire alcune operazioni con o a quella variabile. Lo stesso si verifica in codice macchina. Molte azioni che possono essere eseguite dipendono, in buona parte, dai valori che uno o più registri contengono ed in molti casi sarà lasciato in uno dei registri anche il risultato di un calcolo aritmetico, quando il calcolo stesso sia stato completato. Qualche volta il risultato sarà esso stesso la conclusione del processo, oppure si renderà importante per delle decisioni seguenti e, perciò, dovrà essere immagazzinato in altri luoghi meno utilizzati dei registri. Noi, quindi, abbiamo bisogno di conoscere come inserire un valore in uno dei registri e, susseguentemente, come estrarre quel valore ed inserirlo in un'area di memoria permanente.

I diversi metodi per caricare un registro

I registri possono essere caricati tramite tre diverse sorgenti:

1) Dallo stesso programma, cioè con il "modo di indirizzamento immediato". L'equivalente BASIC è:

   LET X = <NUMERO REALE>

2) Con il contenuto di uno specificato byte in memoria, cioè con l'indirizzamento diretto". L'equivalente BASIC è:

   LET X = <CONTENUTO DI UN'ALTRA VARIABILE>

3) Con il contenuto di un byte che è ad una posizione numerata in un blocco di memoria, cioè con l'indirizzamento indicizzato". L'equivalente BASIC è:

   LET X = <ELEMENTO DI ARRAY(10)>

Non tutti i registri possono essere caricati da ogni locazione con un singolo comando, così sarà spesso necessario usare combinazioni di comandi per fare pieno uso di tutti i registri. Il listato dei modi di indirizzamento disponibili con le differenti istruzioni Load è dato nella tabella alla fine di questa sezione, ma, per dimostrare il funzionamento di alcuni di loro, vi viene dato qui di seguito un programma dimostrativo da immettere usando le tecniche descritte nel Capitolo 4.

Esempio del caricamento di un registro direttamenete da programma.

```
  PC          SR  AC  XR  YR  SP
; 1C00        30  00  00  00  F0
> 008B        00  00  00  00
> 1C8B        00  00  00  00

A  1C00       A2  8B  LDX #$8B
A  1C02       00      BRK

  PC          SR  AC  XR  YR  SP
; 1C04        B0  00  8B  00  F0
> 008B        00  00  00  00
> 1C8B        00  00  00  00
```

Commento:
1C00: Carica il registro X (LoaD X) con il valore esadecimale 8B. Il simbolo

"#" informa la CPU che quello che deve essere caricato è il valore specificato nell'istruzione. Il segno "$" è il simbolo usato convenzionalmente per indicare che il valore che segue è espresso in esadecimale. Confrontando il formato dell'indirizzo con quelli dati nel capitolo precedente vedrete che l'istruzione usa il modo di indirizzamento "immediato".
1C02: Questa è l'istruzione che conclude tutti i programmi inseriti nel testo, restituisce il controllo al Monitor e visualizza i registri.

Quando un programma è stato svolto, esaminando la visualizzazione dei registri, vedrete che il valore esadecimale 8B è stato collocato nel registro X.

Questo semplice caricamento di un valore in un registro può essere eseguito con tutti i tre principali registri tramite le istruzioni LDA, LDX ed LDY, rappresentate in esadecimale dai codici A9, A2 (come nell'esempio in esame) ed A0. Una lista completa delle istruzioni e dei formati per il caricamento dei registri è data alla fine del capitolo.

Esempio di caricamento da memoria:

```
  PC          SR  AC  XR  YR  SP
; 1C00        30  00  00  00  F0
> 008B        AA  00  00  00
> 1C8B        00  00  00  00

A  1C00       A4  8B  LDX $8B
A  1C02       00      BRK

  PC          SR  AC  XR  YR  SP
; 1C04        B0  00  00  AA  F0
> 008B        AA  00  00  00
> 1C8B        00  00  00  00
```

Commento:
1C00: Qui il formato dell'istruzione mostra che è stato impiegato il modo di indirizzamento "diretto" per copiare un valore da una locazione in memoria (indirizzo $8B) nel registro Y. Notate il termine "copiare" poichè l'istruzione Load non ha alcun effetto sul valore contenuto nella locazione di memoria $8B. Notate pure l'assenza del simbolo "#", usato nell'esempio precedente, che dimostra come, in questo caso, $8B rappresenti un indirizzo anzichè un numero.

Per verificare questa istruzione, alterate prima il valore del byte $8B come descritto nel Capitolo 4 con un valore scelto nella gamma esadecimale 0-FF. Ora immettete e fate eseguire il programma. Quando il codice macchina è stato eseguito, dovreste trovare che il registro Y è stato caricato con qualsiasi valore che avete situato nella locazione di memoria $8B.

Esempio di caricamento da una posizione dentro un blocco di memoria:

```
  PC          SR  AC  XR  YR  SP
; 1C00         30  00  00  00  F0
> 008B         AA  00  00  00
> 1C8B         00  00  00  00

A  1C00       A2  8B     LDX #$8B
A  1C02       B5  00     LDA $00,X
A 1C04        00         BRK

  PC          SR  AC  XR  YR  SP
; 1C06         B0  AA  8B  00  F0
> 008B         AA  00  00  00
> 1C8B         00  00  00  00
```

Commento:

In BASIC, è molto spesso utile definire una tabella di valori dentro un array e poi usare singoli valori di quella tabella usando affermazioni come:

LET TT = ARRAY(5)

Nel fare questo, noi diamo per sottointeso che l'interprete BASIC conosce già dove trovare ARRAY e poi come identificare il valore nella quinta posizione. Nessuna facilitazione del genere è disponibile in codice macchina, ma è possibile simulare una tabella con ciò che è conosciuto come "indirizzamento indicizzato".

Nell'indirizzamento indicizzato, prima viene caricato, con un valore, il registro X oppure quello Y (conoscete già come fare ciò) e poi viene dato alla CPU, da programma, un valore (START) che rappresenta l'inizio di un blocco di memoria che deve essere usato come una tabella. Il valore nel registro X od Y è usato per determinare quale byte, nella tabella che comincia allo START, debba riferirsi all'indirizzo usato. Dopo aver scoperto di quale byte si tratta, il contenuto di tale byte è poi caricato in uno dei registri in funzione della forma esatta dell'istruzione.

Come altri metodi di indirizzamento, l'indicizzato si divide in due forme principali, vale a dire l'indirizzamento zero-page e la forma parallela diretta a tutta la memoria. Una sequenza tipica di comandi che realizzano l'indirizzamento indicizzato (questa volta nella versione zero page) è rappresentata dall'esempio.

1C00: Carica il registro X con il valore esadecimale $8B che è il valore indice.
1C02: Carica l'accumulatore con il contenuto del byte che è numerato zero + il contenuto del registro X. Lo zero è l'inizio della tabella. Notate che l'Assembler Mastercode, tramite il quale il listato è stato preparato, usa un punto invece di una virgola, per separare la "X" alla fine della istruzione.

Potete osservare, quando il programma è terminato, che qualsiasi valore abbiate immesso nel byte $8B è ora copiato nell'accumulatore.

Facendo esperimenti, cercate di cambiare il valore $8B che è situato nel registro X, con uno dei F9, FA, o 8C che sono i numeri degli altri tre bytes di memoria prescelti per il programma DEMO. Eseguendo nuovamente il programma, dovreste vedere che l'accumulatore viene caricato da qualsiasi byte, purchè abbia lo stesso numero di quello memorizzato nel registro X.

Esempio di caricamento dell'accumulatore con modo di indirizzamento assoluto:

```
  PC          SR  AC  XR  YR  SP
; 1C00        30  00  00  00  F0
> 008B        00  00  00  00
> 1C8B        AA  00  00  00

A  1C00                 A2  8B  1C
                                    LDX $1C8B
A  1C02     00          BRK

  PC          SR  AC  XR  YR  SP
; 1C05        B0  AA  00  00  F0
> 008B        00  00  00  00
> 1C8B        AA  00  00  00
```

Commento:
1C00: Qui una sola istruzione è tutto quello che è necessario per prendere un valore immagazzinato nella parte alta della memoria a $1C8B.

Esempio di caricamento dell'accumulatore con modo di indirizzamento assoluto indicizzato:

```
  PC          SR  AC  XR  YR  SP
; 1C00        30  00  00  00  F0
> 008B        00  00  00  00
> 1C8B        AA  00  00  00

A  1C00     A2  8B  LDX #$8B
A  1C02             BD  00  1C
                                LDA $1C00,X
A  1C05     00      BRK

  PC          SR  AC  XR  YR  SP
; 1C07        B0  AA  8B  00  F0
> 008B        00  00  00  00
> 1C8B        AA  00  00  00
```

Commento:
1C00: Il registro X, che sarà usato come registro indice, è caricato con il valore $8B.
1C02: L'accumulatore è caricato con il contenuto della locazione di memoria il cui indirizzo è $1C00 più il valore nel registro X. Nei vostri esperimenti, se volete sostituire il valore $8B contenuto nel registro X, con un valore tale che, sommato a $1C00, indichi una locazione di alta memoria, potete farlo, purchè specifichiate il valore che deve essere caricato nell'accumulatore.

Tabella delle istruzioni Load e dei modi di indirizzamento disponibili

I modi di indirizzamento disponibili all'uso con LDA sono:

```
Zero page:
LDA $          : A5        : A5 FF
Assoluto:
LDA $02FF      : AD        : AD FF 02
Immediato:
LDA #$FF       : A9        : A9 FF
Pre-indicizzato indiretto:
LDA ($FF,X)    : A1        : A1 FF
Post-indicizzato indiretto:
LDA ($FF),Y    : B1        : B1 FF
Zero page indicizzato,X:
LDA $FF,X      : B5        : B5 FF
Assoluto indicizzato,X:
LDA $02FF,X    : BD        : BD FF 02
Assoluto indicizzato,Y:
LDA $02FF,Y    : B9        : B9 FF 02
```

I modi di indirizzamento disponibili all'uso con LDX sono:

```
Zero page:
LDX $FF        :A6         : A6 FF
Assoluto:
LDX $02FF      : AE        : AE FF 02
Immediato:
LDX #$FF       : A2        : A2 FF
Zero page indicizzato,Y:
LDX $FF,Y      : B6        : B6 FF
Assoluto indicizzato,Y:
LDX $02FF,Y    : BE        : BE FF 02
```

I modi di indirizzamento disponibili all'uso con LDY sono:

```
Zero page:
LDY $FF        :A4         : A4 FF
```

Assoluto:
LDY $02FF : AC : AC FF 02
Immediato:
LDX #$FF : A0 : A0 FF
Zero page indicizzato,X:
LDY $FF,X : B4 : B4 FF
Assoluto indicizzato,X:
LDY $02FF,X : BC : BC FF 02

Trasferimento tra registri : istruzioni TXA, TAX, TYA, TAY, TXS e TSX

Funzione: copiare un valore da uno specifico registro della CPU all'altro.

Un altro metodo per depositare un valore in uno dei tre registri principali della CPU o nello stack pointer è l'uso delle istruzioni "transfer". In linguaggio assembly, queste istruzioni cominciano con una "T" seguita da due lettere che rappresentano il registro da dove il valore deve essere copiato (o trasferito) ed il registro nel quale deve essere immesso. Così:

TAX: "trasferisci il contenuto dell'accumulatore nel registro X."
TXS: "trasferisci il contenuto del registro X nello stack pointer."
TYA: "trasferisci il contenuto del registro Y nell'accumulatore."

Esempio di trasferimento del contenuto dell'accumulatore nel registro X:

```
  PC      SR  AC  XR  YR  SP
; 1C00     30  00  00  00  F0
> 008B     00  00  00  00
> 1C8B     00  00  00  00

A  1C00    A2  FF  LDX #$FF
A  1C02    A9  AA  LDA #$FF
A  1C04    AA      TAX
A  1C05    00      BRK

  PC      SR  AC  XR  YR  SP
; 1C07     B0  AA  AA  00  F0
> 008B     00  00  00  00
> 1C8B     00  00  00  00
```

Commento:
1C00-1C02: L'accumulatore viene caricato con $AA ed il registro X con $FF. Non c'è una ragione particolare nella scelta di questi valori, semplicemente aiutano a distinguere i due registri.

1C04: Un unico comando è tutto quello che è necessario per copiare il contenuto dell'accumulatore nel registro X.

Esempio dell'uso di TYA

```
  PC          SR  AC  XR  YR  SP
; 1C00         30  00  00  00  F0
> 008B         00  00  00  00
> 1C8B         00  00  00  00

A  1C00       A0 FF     LDY #$FF
A  1C02       A9  AA    LDA #$AA
A  1C04       98        TYA
A  1C05       00        BRK

  PC          SR  AC  XR  YR  SP
; 1C07         B0  FF  00  FF  F0
> 008B         00  00  00  00
> 1C8B         00  00  00  00
```

Commento:
È la procedura contraria di quella vista nell'esempio precedente. In questo caso il registro Y e l'accumulatore vengono caricati rispettivamente con i valori $AA e $FF, poi il valore nel registro Y viene trasferito nell'accumulatore con un solo comando.

Tabella dei comandi Transfer

Nota: tutti i comandi Transfer utilizzano unicamente il modo di indirizzamento "intrinseco". Le stesse istruzioni specificano tutte le locazioni interessate e non c'è bisogno di ulteriori indirizzi.

Trasferire A in X : TAX : AA : AA
Trasferire X in A : TXA : 8A : 8A
Trasferire S in X : TSX : BA : BA
Trasferire X in S : TXS : 9A : 9A
Trasferire A in Y : TAY : A8 : A8
Trasferire Y in A : TYA : 98 : 98

Caricamento di valori nello stack pointer, nello status register e nel program counter.

Sebbene i registri principalmente usati sono sempre l'accumulatore ed i registri X ed Y, è utile ricordare che ci sono altri tre registri in cui è possibile

riporre dei valori, sebbene sia necessario, prima di intraprendere ciò, considerare attentamente gli effetti di tale azione. Nel caso in cui volessimo depositare dei valori in questi registri non abbiamo a disposizione un semplice comando del genere "Load", ma dobbiamo fare solitamente uso di una combinazione di comandi.
Esempio di caricamento dello stack pointer.

```
  PC          SR  AC  XR  YR  SP
; 1C00        30  00  00  00  F0
> 008B        F0  00  00  00
> 1C8B        00  00  00  00

A  1C02       A6  8B      LDY #$8B
A  1C03       00          BRK

  PC          SR  AC  XR  YR  SP
; 1C05        B0  00  F0  00  F0
> 008B        F0  00  00  00
> 1C8B        00  00  00  00
```

Commento:
(NB: l'uso dello stack sarà trattato in un capitolo a parte).

Non c'è un comando che permetta di caricare lo stack pointer direttamente da memoria ma, come abbiamo visto, c'è un comando per caricare un valore dal registro X. Poiché non c'è nessuna difficoltà nel caricamento di un valore da memoria nel registro X, lo stack pointer può essere caricato in due fasi, prima caricando il valore desiderato nel registro X e poi, trasferendolo nello stack pointer.

1C00: Carica il registro X con il contenuto del byte $8B in zero page.
1C02: Trasferisce il contenuto del registro X allo stack pointer.

Esempio di caricamento dello status register

```
  PC          SR  AC  XR  YR  SP
; 1C00        30  00  00  00  F0
> 008B        AA  00  00  00
> 1C8B        00  00  00  00

A  1C00       A5  8B      LDA $8B
A  1C02       48          PHA
A  1C03       28          PLP
A  1C04       00          BRK
```

```
  PC       SR  AC  XR  YR  SP
; 1C06     BA  00  00  00  F0
> 008B     AA  00  00  00
> 1C8B     00  00  00  00
```

Commento:

Sebbene ci sia un comando che permetta di fare questo, ci sono pochissime occasioni in cui sia necessario caricare lo status register come tutti gli altri registri.

Come abbiamo già visto nella trattazione del nostro chip immaginario Lawland 10000, gli otto bits dello status register non sono puramente un valore ad otto bits, bensì otto indicatori separati (sebbene solo sette siano utilizzati). Il settare l'intero registro a volte avrà maggiore efficacia, ma si rivela utile nel resettare l'intero sistema da alcuni stati precedenti come infatti fà il programma MONITOR, che prima salva il contenuto dello status register e poi, dopo che è stato eseguito il programma in codice macchina, riporta il sistema allo stato originale, assicurando così che i flags, i cui valori precedenti potrebbero danneggiare il sistema, siano resettati al valore che permette al BASIC di funzionare.

1C00-1C02: Il valore contenuto nella locazione di memoria $8B è caricato nell'accumulatore e poi situato nello stack usando l'istruzione PusH Accumulator, che verrà trattata più largamente nel Capitolo 17; a questo punto tutto quello che serve è sapere che, in termini di codice macchina, "push" significa mettere un valore sullo stack.
1C03: Il valore situato nello stacck è caricato nello status register usando il PuL1 Processor dello status register. Una istruzione "pull" è il contrario di una "push", cioè porta fuori dallo stack l'ultimo valore.

Quando il codice macchina è stato eseguito dovreste vedere sul monitor che il valore $AA che avevate immesso nella locazione $8B è diventato $BA nello status register. Il motivo di ciò è che il flag di break, il bit 5 del registro, è stato settato dall'istruzione BRK che conclude il programma, trasformando il valore binario "10101010" che abbiamo immesso nello status register, in "10111010".

Caricamento del program counter

Siamo rimasti con un solo registro che non sappiamo come caricare: il program counter. Viene detto a volte che non esiste comando per caricare questo registro ma non è vero. Esiste tutta una serie di istruzioni per caricare il program counter, ma non sono chiamate istruzioni LOAD, sono chiamate istruzioni "JUMP". Poiché il pogram counter ha il compito di ricordare il punto dal quale deve continuare l'esecuzione del programma in codice mac-

china, caricare un nuovo valore dentro di esso equivale a "saltare" ad un'altra parte del programma.

Questo sarà trattato con più dettagli nel Capitolo 15.

Immagazzinamento dei contenuti dei registri

Le istruzioni STA, STX e STY

Abbiamo così raggiunto la fase in cui possiamo prendere un valore da ogni parte della memoria, o quasi, ed immetterlo in uno dei registri del 7501. Più avanti vedremo i metodi usati per fare qualcosa con quello che abbiamo trasferito. Prima di ciò, comunque, esploreremo l'aspetto contrario del caricamento dei registri, e cioè, la collocazione dei contenuti dei registri in una locazione di memoria prescelta. Come nella precedente sezione, esamineremo anche alcuni dei diversi modi di indirizzamento della memoria che sono permessi per effettuare il trasferimento, ma solo alcuni. La serie completa dei modi di indirizzamento disponibili per immagazzinare il contenuto di un registro è contenuta nella tabella alla fine della sezione.

Esempio dell'immagazzinamento del contenuto dell'accumulatore.

```
  PC          SR  AC  XR  YR  SP
; 1C00         30  00  00  00  F0
> 008B         AA  00  00  00
> 1C8B         00  00  00  00

A  1C00       A5  8B    LDA $8B
A  1C02       A2  8C    LDX  #$8C
A  1C04       95  00    STA  $00,X
A  1C06       00        BRK

  PC          SR  AC  XR  YR  SP
; 1C08         B0  AA  8C  00  F0
> 008B         AA  AA  00  00
> 1C8B         00  00  00  00
```

Commento.

Il nostro primo esempio di questa sezione mostra come si può immagazzinare il contenuto dell'accumulatore nella memoria zero-page usando il registro X per l'indirizzamento indicizzato. Le istruzioni del linguaggio assembly utilizzate sono:

1C00: Carica l'accumulatore con il contenuto del byte di memoria $8B.

1C02: Carica il registro X con il numero$8B.

1C04: Immagazzina il contenuto dell'accumulatore all'indirizzo rappresentato dalla somma di zero più il contenuto del registro X.

Quando il codice macchina è stato eseguito, dovreste trovarvi con lo stesso valore in $8B, nell'accumulatore ed in $8C. Chiaramente, questo avviene perchè il contenuto di $8B viene copiato nell'accumulatore e da qui copiato in $8C. Il contenuto del registro X sarebbe $8C, indicando così il byte in zero page nel quale il contenuto dell'accumulatore è stato immagazzinato.

Potete eseguire questo immagazzinamento indicizzato in zero-page anche con l'accumulatore ed il registro Y, usando il registro X per registrare il numero indice, o con il registro X, usando il registro Y per registrare il numero indice. Un esempio di immagazzinamento del contenuto del registro X può essere:

```
LDX $8B
LDY $8C
STX 0,Y
```

Per verificare questa sequenza, caricate quanto segue nel programma DEMO:

```
  PC          SR  AC  XR  YR  SP
; 1C00         30  00  00  00  F0
> 008B         AA  00  00  00
> 1C8B         00  00  00  00

A  1C00       A6  8B    LDX $8B
A  1C02       A0  8C    LDY  #$8C
A  1C04       96  00    STX $00,Y
A  IC06       00        BRK

  PC          SR  AC  XR  YR  SP
; 1C08         B0  00  AA  8C  F0
> 008B         00  00  00  00
```

Dopo l'esecuzione del codice macchina, $8B, il registro X e $8C dovrebbero contenere tutti il valore originario della locazione $8C. Il registro Y dovrebbe contenere il valore $8C.

Immagazzinamento del contenuto dello status register, dello stack pointer e del program counter

Mentre i tre registri principali, quando si tratta di immagazzinare il loro contenuto in memoria, sono facilmente accessibili, gli altri tre registri richiedono una certa routine con l'uso di molte fasi intermedie. Inizieremo con

l'immagazzinamento del contenuto dello status register.

Esempio dell'immagazzinamento del contenuto dello status register.

```
  PC          SR  AC  XR  YR  SP
; 1C00        30  00  00  00  F0
> 008B        00  00  00  00
> 1C8B        00  00  00  00

A  1C00       A8      PHP
A  1C01       68      PLA
A  1C02       85  8C  STA $8C
A  1C04       00

  PC          SR  AC  XR  YR  SP
; 1C06        30  00  00  00  F0
> 008B        00  00  00  00
> 1C8B        00  00  00  00
```

Commento:

1C00-1C01: Sono usate ancora una volta le istruzioni "push" e "pull", che verranno trattate nel Capitolo 17, per trasferire il contenuto dello status register nello stack e, poi, per richiamarlo dall'accumulatore.

1C02: Dopo aver trasferito il contenuto nell'accumulatore, non c'è alcun problema per il suo trasferimento in memoria usando una istruzione di immagazzinamento.

Esempio dell'immagazzinamento del contenuto dello stack pointer.

```
  PC          SR  AC  XR  YR  SP
; 1C00        30  00  00  00  F0
> 008B        00  00  00  00
> 1C8B        00  00  00  00

A  1C00       BA      TSX
A  1C01       86  8C  STX $8C
A  1C03       00      BRK

  PC          SR  AC  XR  YR  SP
; 1C05        B0  00  F0  00  F0
> 008B        00  F0  00  00
> 1C8B        00  00  00  00
```

Commento:

Abbiamo già visto che il metodo con cui muovere il contenuto dello stack pointer ad un altro registro consiste nel trasferirlo nel registro X dal quale è cosa semplice trasferirlo ad una specifica locazione di memoria.

Immagazzinamento del contenuto del program counter

La procedura per immagazzinare il contenuto del program counter è piuttosto complessa e sarà discussa nel Capitolo 16, quando le istruzioni necessarie verranno chiarite.

Tabella delle istruzioni con i loro modi di indirizzamento disponibili

I modi di indirizzamento disponibili all'uso con STA sono:

```
Zero page:
STA $FF          : 85          : 85
Assoluto:
STA $02FF        : 8D          : 8D FF 02
Pre-indicizzato indiretto:
STA ($FF,X)      : 81          : 81 FF
Post-indicizzato indiretto:
STA ($FF),Y      : 91          : 91 FF
Zero page indicizzato,X:
STA $FF,X        : 95          : 95 FF
Assoluto indicizzato,X:
STA $02FF,X      : 9D          : 9D FF 02
Assoluto indicizzato,Y:
STA $02FF,Y      : 99          : 99 FF 02
```

I modi di indirizzamento disponibili all'uso con STX sono:

```
Zero page:
STX $FF          : 86          : 86 FF
Assoluto:
STX $02FF        : 8E          : 8E FF 02
Zero page indicizzato,Y:
STX $FF,Y        : 96          : 96 FF
Assoluto indicizzato,Y:
STX $02FF,Y      : 9E          : 9E FF 02
```

I modi di indirizzamento disponibili all'uso con STY sono:

```
Zero page:
STY $FF          : 84          : 84 FF
Assoluto:
STY $02FF        : 8C          : 8C FF 02
Zero page indicizzato,X:
STY $FF,X        : 94          : 94 FF
Assoluto indicizzato:
STY $02FF,X      : 9C          : 9C FF 02
```

CAPITOLO 8

I FLAG

Dopo aver iniziato a vedere le istruzioni necessarie per scrivere un programma in codice macchina, facciamo una pausa ed introduciamo una piccola importante spiegazione veramente necessaria. Prima di prendere in considerazione tutta la gamma delle istruzioni abbiamo bisogno di chiarire la funzione dei flag, che sono fondamentali per molte delle decisioni che prende un programma in codice macchina, durante la sua esecuzione. Poichè i flag non sono così importanti di per se stessi, ma, piuttosto, per i loro effetti su altre istruzioni, nel corso del capitolo useremo alcune istruzioni a voi non ancora familiari. In ogni caso, daremo una descrizione grossolana dello scopo di un comando, ma senza entrare nei dettagli. L'unica altra alternativa sarebbe spiegare tali comandi senza aver trattato i flag, che sono essenziali per il loro uso: in altre parole un ostacolo insormontabile.

Cos'è un flag

Nei capitoli precedenti, quando studiavamo l'immaginario Lawland 10000, scoprimmo che c'era una collezione di sette locazioni speciali, capaci di registrare decisioni si/no nella forma di un "1" o di uno "0". Sebbene non li trattammo in dettaglio, notammo che, nell'esecuzione di un calcolo, l'eventuale generazione di un "carry" è registrata in un flag. Notammo pure che ogni flag è un bit singolo in uno speciale registro dentro la CPU, denominato "status register". Come tutti i registri del 7501, eccetto il program counter, lo status register è lungo otto bit ma, in realtà, solo sette sono usati. Ognuno dei sette bit è usato per registrare una decisione si/no o il risultato di una operazione fatta dalla CPU. I sette flag sono:

Bit 0: Flag di Carry
1: Flag di Zero
2: Flag di Interrupt
3: Flag di Decimal
4: Flag di Break
5: Non usato (di solito settato a 1)
6: Flag di Overflow
7: Flag di Sign

Ogni flag sarà spiegato in seguito, ma, prima di entrare in queste spiegazioni, assicuratevi che abbiate compreso che un flag è la risposta ad una sola domanda, con la risposta "sì" se il flag è a "1" e "no" se il flag è a "0". Tutto questo può sembrare una cosa banale da sottolineare, ma così tanti programmatori in codice macchina si confondono sul fatto che, per esempio, il flag di zero viene settato quando si crea uno zero piuttosto che quando esso sia a zero in accordo con lo zero che è stato creato. Più semplice, e più attendibile, è il ricordare che il flag di zero risponde alla domanda "È stato creato uno zero?". Nelle sezioni che seguono, ogni flag e le domande a cui risponde sono spiegate e verrà più che ripagato il tempo impiegato per memorizzarle.

Il Flag di Carry

Funzione: Registrare se l'ultima azione eseguita dall'accumulatore è sfociata nella creazione di un valore che è al di fuori del range degli otto bit. La domanda a cui il flag risponde è quindi: "L'ultima operazione eseguita è risultata nella creazione di un numero più grande di 255 o minore di zero".

Questo è il flag che verrà più usato di ogni altro. Infatti,c'è tutta una serie di azioni che la CPU esegue basandosi direttamente sul valore del flag di carry, senza che si debba specificare di tenerne conto. Il suo uso differisce poco a seconda che si usino numeri con o senza segno, ma il suo compito è sempre registrare se una operazione eseguita sul contenuto dell'accumulatore ha avuto, o dovrebbe avere, come risultato, la creazione di un valore sopra 255.

Nonostante tutto ciò, ci sono anche un paio di istruzioni, che operano su valori contenuti in locazioni di memoria, non nell'accumulatore, capaci di settare il flag di carry. Le istruzioni "shift" e "rotate" saranno trattate nel Capitolo 13.

Notate che il flag di carry è interessato da ogni operazione eseguita dall'accumulatore. Se una operazione genera un carry, il flag di carry sarà settato. Se un'altra operazione seguente risulta in un valore nel range 0-255 il flag di carry sarà "riazzerato". Ogni uso del flag di carry, fatto per esaminare il risultato di una particolare operazione, deve essere quindi eseguito immediatamente, prima che si sia fatto uso nuovamente dell'accumulatore.

Il flag di carry ed i numeri senza segno

Questo è l'uso più semplice dei due. Con i numeri senza segno, la sola operazione che può portare ad un valore sopra 255, usando l'accumulatore, è sommare un numero a quello che è già inserito nell'accumulatore. Il formato del comando ADD ed esempi del suo uso saranno dati in un capitolo

seguente. A questo momento semplicemente affermiamo che una serie di comandi del tipo: LDA # 200(carica l'accumulatore con 200); ADC # 200(aggiungi 200 all'accumulatore, registrando ogni carry) setteranno il flag di carry visto che il valore creato nell'accumulatore è maggiore di 255. Poichè il valore massimo che l'accumulatore può contenere è 255, cioè otto cifre binarie tutte settate a uno, il risultato generato sarà maggiore di 256 rispetto al numero corretto. Il nono bit, rappresentante 256 in binario, viene perso. Quando questo accade, il flag di carry è settato, diventando così in effetti un nono bit per l'accumulatore, permettendo di generare valori, in lunghezza, fino alla nona cifra binaria (fino a 511 in decimale).

Nel caso di numeri composti da più di un unico byte, il flag di carry è parte integrante del processo di addizione dei bytes successivi, permettendo di riportare il risultato di ogni addizione dalla cifra più significativa di un byte alla meno significativa del byte seguente, più significativo, allo stesso modo di come avviene tra le cifre di uno stesso byte.

Il flag di carry ed i numeri con segno

Con l'impiego dei numeri con segno l'uso del flag di carry non è chiaro come nel caso precedente. I numeri con segno, come abbiamo già visto, usano la cifra binaria più significativa per indicare se il numero è negativo o no. Per questa ragione, i numeri che possono essere registrati in un unico byte sono compresi tra -128 e +127. Sommando due di questi numeri, si può verificare una modifica della cifra binaria più significativa, ma non si potrà mai generare un numero così grosso da non poter immetterlo in otto cifre binarie. Poichè ogni modifica nel bit del segno è competenza di un altro flag, il flag di overflow, il flag di carry non ha alcuna voce in capitolo quando devono essere sommati nell'accumulatore numeri con segno ad un solo byte.

Nonostante ciò, il flag di carry ricopre spesso un ruolo importante nei calcoli aritmetici con segno, per la semplice ragione che i programmi in codice macchina fanno uso frequentementedi numeri con segno a due (o più) byte. Per costruire un numero con segno a due byte, che può essere compreso tra -32768 e 32767, usiamo di nuovo la cifra binaria più significativa per registrare se il numero è positivo o negativo, ma solo la cifra binaria più significativa dell'high byte. Per esempio, il numero 511 può essere registrato in due byte come segue:

HIGH BYTE	LOW BYTE
00000001	11111111

Qui potete vedere che la cifra binaria più significativa, lo zero sulla sinistra, indica che il numero è positivo. La cifra binaria più a destra nell'high byte è settata a uno; ciò indica che il numero contiene 256. Il low byte ha tutte le cifre binarie settate, registrando un valore di 255 e non -127 poichè non ha senso il bit di segno nel secondo byte.

Quando eseguiamo la somma di due valori con segno composti da più di un byte (es.511 + 511), gli high byte si comportano proprio come numeri con segno ad un solo byte e non fanno uso del flag di carry. Il flag di carry è utilizzato sommando qualsiasi byte alla destra dell'high byte, poichè ciò che si sta sommando sono due valori senza segno.

Istruzioni che si riferiscono al flag di carry

Oltre alle istruzioni che alterano il valore dell'accumulatore, e quindi, potenzialmente, settano o azzerano il flag di carry, c'è tutta una serie di istruzioni che agisce direttamente su di esso o sulle basi del suo contenuto:

SEC (SEt Carry) : pone il flag di carry ad uno, indifferentemente dal suo stato.

CLC (CLear Carry) : pone il flag di carry a zero, indifferentemente dal suo stato.

BCS (Branch Carry Set) : l'esecuzione del programma si indirizza ad un punto specificato se il flag di carry, quando si incontra tale istruzione, è settato ad uno.

BCC (Branch Carry Clear) : l'esecuzione del programma si indirizza ad un punto specificato se il flag di carry, quando si incontra tale istruzione, è a zero. Per una spiegazione più chiara e per le dimostrazioni relative a queste istruzioni "branch", potete vedere il capitolo sul controllo di programma.

Routine di dimostrazione di SEC.

```
  PC          SR  AC  XR  YR  SP
; 1C00         30  00  00  00  F0
> 008B         00  00  00  00
> 1C8B         00  00  00  00

A  1C00       A9  00    LDA #$00
A  1C02       48        PHA
A  1C03       28        PLP
A  1C04       38        SEC
A  1C04       18        BRK

  PC          SR  AC  XR  YR  SP
; 1C07         31  00  8B  00  F0
> 008B         00  00  00  00
> 1C8B         00  00  00  00
```

Commento:
1C00-1C03: Avevamo già brevemente introdotto le istruzioni "push" e "pull", nella trattazione relativa al trasferimento di un valore nello status register.

Queste linee caricano zero nell'accumulatore e poi lo trasferiscono nello status register, assicurando così che il flag di carry sia azzerato.
1C04: Il solo comando SEC è sufficiente per settare il flag di carry.

Dopo l'esecuzione del codice macchina, dovreste trovare che il flag di carry, la cifra più a destra nello status register, è settata ad uno. Ciò è dimostrato dal fatto che il valore del registro ora è $31. In ogni caso dovete ignorare l'elemento $30 nel valore dello status register, poiché esso è dato solo da bit 5 del registro, che non è usato, ma che, normalmente, si trova settato da uno. La differenza nel valore del registro ("1") indica che il bit zero è settato, poiché $31-$30=2^0.

Routine di dimostrazione di CLC.

```
  PC          SR  AC  XR  YR  SP
; 1C00        30  00  00  00  F0
> 008B        AA  00  00  00
> 1C8B        00  00  00  00

A  1C00       A9  FF  LDA #$FF
A  1C02       48      PHA
A  1C03       28      PLP
A  1C04       18      BRK

  PC          SR  AC  XR  YR  SP
; 1C07        FE  FF  00  00  F0
> 008B        AA  00  00  00
> 1C8B        00  00  00  00
```

Commento:
1C00-1C03: Come nel progrramma precedente, eccetto che, in questo caso, il valore inserito nello status register è $FF, coè 255 in decimale. In binario, $FF è 11111111; così, la sua immissione nello status register assicura che il flag di carry, assieme a tutti gli altri, sia settato.
1C04: Dopo aver settato il flag di carry, l'istruzione CLC lo azzera.

Dopo l'esecuzione del codice macchina, dovreste trovare che il flag di carry è stato settato a zero, con lo status register che contiene il valore $FE, cioè $FF-2^0.

Nei prossimi capitoli, incontrerete molti esempi dell'uso del flag di carry e noi contiamo su di essi piuttosto che darvi, qui, routines che richiederebbero istruzioni che non avete ancora esaminato.

Il Flag di Zero

Funzione: Registra se il risultato dell'ultima operazione eseguita è uno zero in un registro od in una locazione di memoria. Nel caso che si sia creato uno zero, il flag è settato ad uno, in caso contrario è posto a zero. I comandi che non hanno effetto sul flag di zero sono NOP (no operation), ogni forma dei comandi di immagazzinamento, istruzioni di azzeramento del flag, le branche del controllo di programma ed i richiami alle subroutines.

La domanda a cui il flag di zero risponde è:
"L'ultima operazione eseguita è sfociata nella creazione di uno zero?".

Il flag di zero è simile a quello di carry nel fatto che esso riflette il risultato dell'ultima operazione che viene eseguita; così deve essere esaminato subito se si vuole trarre una informazione dal suo contenuto.

Nell'uso pratico il flag di zero è usato molto spesso per compiti semplici, come determinare se una azione è stata eseguita un numero determinato di volte, immettendo un valore in uno dei registri e sottraendo uno da essi ogni volta che essa è eseguita. Se il flag di zero è esaminato ogni volta che si sottrae uno, esso indicherà quando il numero di sottrazioni equivale al numero che era originariamente nel registro. Il contenuto del flag di zero è spesso usato, nel controllo di programma, per saltare ad un'altra parte del programma, quando un certo valore raggiunge zero.

Anche questa volta, troverete molti esempi dell'uso del flag di zero più avanti nel libro, quando avremo il tempo di saminare una selezione più larga di comandi. Non viene data alcuna routine di dimostrazione, poiché tutto quello che è richiesto per settare il flag di zero è caricare zero nell'acccumulatore (LDA #$0) e tutto quello che è richiesto per inizializzare il flag è caricare nell'accumulatore un numero diverso da zero (LDA #$1).

Istruzioni che si riferiscono al flag di zero

BNE (Branch Non-Equal): l'esecuzione del programma si rivolge ad uno specifico punto se, quando incontra tale istruzione, il flag di zero è a zero.

BEQ (Branch Equal): l'esecuzione del programma si rivolge ad uno specifico punto se, quando incontra tale istruzione, il flag di zero è a uno.

Più ampi dettagli nell'uso di queste due istruzioni saranno dati nel capitolo sul controllo di programma.

Il Flag di Interrupt

Funzione: Disabilita o abilita gli interrupt della CPU che permettono di eseguire i compiti essenziali del sistema, durante le interruzioni dell'esecuzio-

ne del programma inserito. La domanda a cui il flag risponde è:

"Gli interrupt sono disabilitati?"

Questo è un flag che userete raramente, a meno che intendiate lavorare su applicazioni estremamente tecniche. Per dirla in breve, la CPU ha un collegamento che è usato per informarla di bloccare le operazioni a cui essa sta dedicandosi. La ragione di ciò è che la CPU ha molti compiti da eseguire durante il funzionamento della macchina, molti dei quali devono essere eseguiti ad intervalli regolari. Mentre, per esempio, esegue un vostro programma BASIC, la CPU deve anche assicurare che la tastiera sia controllata, per una eventuale immissione esterna di dati. Quando il sistema necessita dell'esecuzione di questi compiti "fissi", esso invia un segnale alla CPU, per fermare, per il momento, ciò che sta facendo e fare qualche altra cosa. La CPU terrà una registrazione del punto che ha raggiunto nella realizzazione del suo compito, si rivolgerà alle nuove richieste e le eseguirà, riprendendo poi il compito originale là dove l'aveva lasciato. Questo processo è, naturalmente, così veloce che chi utilizza il computer è completamente ignaro del fatto che il programma BASIC venga continuamente interrotto.

Alcune funzioni, comunque, sono studiate per non essere interrotte; ad esempio i "loop" temporali, che verrebbero resi insignificanti se la CPU fosse interrotta nella loro esecuzione. In questi casi, il 7501 provvede a disabilitare gli interrupt. Le richiste di interruzione sono lo stesso rivolte alla CPU, ma essa le ignora e prosegue con il compito che sta svolgendo in quel momento. La disabilitazione è effettuata settando il flag di interrupt a uno. Riportando il flag di interrupt a zero si permette agli interrupt di funzionare normalmente.

Come abbiamo detto, il numero di occasioni in cui si rende necessario disabilitare gli interrupt è estremamente limitato. Se desiderate disabilitarli, assicuratevi di sapere cosa state facendo perchè una volta che gli interrupt sono disinseriti, la CPU èa tutti gli effetti isolata dall'esterno e il vostro programma può bloccarsi, se non è in grado di ripristinare in modo corretto gli interrupt.

Comandi che si riferiscono al flag di interrupt

SEI (SEt Interrupt) : setta il flag di interrupt ad uno e disabilita gli interrupt.

CLI (Clear Interrupt) : pone a zero il flag di interrupt abilitando gli interrupt.

Esempio dell'uso di SEI.

```
  PC          SR  AC  XR  YR  SP
; 1C00         30  00  00  00  F0
> 1C8B         00  00  00  00
> 1C8B         00  00  00  00

A  1C00       A9  00    LDA #$00
A  1C02       48        PHA
A  1C03       28        PLP
A  1C04       78        SEI
A  1C05       00        BRK

  PC          SR  AC  XR  YR  SP
; 1C07         34  00  00  00  F0
> 008B         00  00  00  00
> 1C8B         00  00  00  00
```

Commento:

Lo status register è caricato con zero, poi la sola istruzione SEI è sufficiente per settare il flag di interrupt, che si trova nel bit due nel registro. Questo è indicato dal fatto che il valore dello status register è ora $34 : $34-$30 = 2^2.

Esempio dell'uso di CLI.

```
  PC          SR  AC  XR  YR  SP
; 1C00         30  00  00  00  F0
> 1C8B         00  00  00  00
> 1C8B         00  00  00  00

A  1C00       A9  FF    LDA #$FF
A  1C02       48        PHA
A  1C03       28        PLP
A  1C04       78        SEI
A  1C05       00        BRK

  PC          SR  AC  XR  YR  SP
; 1C07         FB  FF  00  00  F0
> 008B         00  00  00  00
> 1C8B         00  00  00  00
```

Commento:

Ancora una volta lo status register è caricato con $FF ed il valore risultante $FB (= $FF-2???) mostra che il bit 2 è stato azzerato.

Il Flag di Decimal

Funzione: Predispone il chip nel modo "decimale", dove ogni byte registra due cifre decimali ed i calcoli aritmetici sono eseguiti in accordo con le normali regole del sistema di numerazione decimale. La domanda a cui il flag di decimal risponde è: "Il chip è in modo decimale?"

La famiglia di chip di cui il 7501 è un membro è il solo grosso gruppo di chip con CPU a 8 bit che hanno la facilitazione di eseguire direttamente calcoli aritmetici decimali, sebbene in un modo piuttosto limitato. Questa capacità sarà discussa in più ampi dettagli nel capitolo dedicato alle istruzioni aritmetiche.

L'unico scopo del flag di decimal è settare il chip in questo insolito modo.

Istruzioni che si riferiscono al flag di decimal

SED (SEt Decimal) : il flag di decimal è settato ad uno quando viene incontrata questa istruzione, immettendo il chip nel modo decimale.

CLD (CLear Decimal) : il flag di decimal è settato a zero quando viene incontrata questa istruzione, riportando il chip al consueto modo binario.

Il normale funzionamento del computer non può essere in modo "decimale" il quale torna utile solo per l'esecuzione di specifici scopi di programmazione ed è sempre disattivato prima che la CPU ritorni ad uno stato di "attesa".

Dimostrazione di SED.

```
  PC          SR  AC  XR  YR  SP
; 1C00        30  00  00  00  F0
> 1C8B        00  00  00  00
> 1C8B        00  00  00  00

A  1C00       A9  00  LDA #$00
A  1C02       48      PHA
A  1C03       28      PLP
A  1C04       78      SEI
A  1C05       00      BRK

  PC          SR  AC  XR  YR  SP
; 1C07        38  00  00  00  F0
> 008B        00  00  00  00
> 1C8B        00  00  00  00
```

Commento.

Lo status register è caricato con zero per pulire tutti i flags, poi la sola istruzione SED è sufficiente per settare il flag; questo è indicato dal fatto che il contenuto dello status register è $38, cioè $30 + 2^3.

Dimostrazione di CLD.

```
  PC          SR  AC  XR  YR  SP
; 1C00        30  00  00  00  F0
> 1C8B        00  00  00  00
> 1C8B        00  00  00  00

A  1C00       A9  00      LDA #$00
A  1C02       48          PHA
A  1C03       28          PLP
A  1C04       D8          SEI
A  1C05       00          BRK

  PC          SR  AC  XR  YR  SP
; 1C07        F7  FF  00  00  F0
> 008B        00  00  00  00
> 1C8B        00  00  00  00
```

Commento:

L'opposto della precedente routine; infatti, i flag sono settati tutti a uno e poi, il flag di decimal è azzerato con la sola istruzione CLD, come indicato dal valore $F7, cioè $FF-2^3, nello status register.

Il Flag di Break

Funzione: Indica se una richiesta di interruzione ricevuta dalla CPU è generata dall'hardware del sistema (valore del flag zero) o dall'istruzione BRK di un programma in codice macchina in esecuzione (valore del flag uno). La domanda a cui il flag risponde è: "L'ultimo interrupt che è avvenuto è stato dato da una istruzione 'break' inserita in un programma in codice macchina".

Dopo aver brevemente discusso a proposito degli interrupt in relazione con il flag di interrupt, dobbiamo fare una distinzione tra gli interrupt che sono generati dal sistema che circonda la CPU e quelli che sono generati con un programma in codice macchina. C'è una grossa differenza tra i due, poichè gli interrupt hardware, il tipo che abbiamo trattato in precedenza, sono manifestati alla CPU tramite i collegamenti che la uniscono al resto del sistema, mentre gli interrupt software giungono ad essa nella forma di specifica istruzioni in codice macchina (BRK) contenuta in un programma. L'uso di BRK sarà discusso in un prossimo capitolo.

Il problema per cui il flag di break è stato ideato a rispondere è che la CPU non è capace di distinguere immediatamente tra un interrupt software ed uno hardware; essa non può quindi sapere se le venga chiesto di andare a controllare la tastiera oppure eseguire una subroutine dettata dal programma in codice macchina. Il problema è risolto dal flag di break poichè è possibile scrivere, per il programmatore in codice macchina, una corta routine che, quando si registra un interrupt, esamini il flag di break per vedere se l'interrupt è creato dal sistema o se c'è una routine, specificata dal programmatore, che deve essere eseguita.

Oltre all'istruzione BRK, che non trattiamo a questo punto nei dettagli, non ci sono altre istruzioni che si riferiscono al flag di break.

Il Flag di Overflow

Funzione: Rileva, quando si usano numeri con segno, se c'è stato un carry dalla parte principale del numero, ovvero dalle prime sette cifre binarie verso l'ottava cifra che è usata per registrare il segno del numero, modificando così il segno del numero. Il problema che si verifica quando il segno di un numero viene cambiato erroneamente da un carry è assai complicato e sarà trattato nel Capitolo 10, ma la domanda a cui il flag di overflow risponde è: "L'ultima operazione eseguita è sfociata in un numero, con segno, avente segno non valido".

Abbiamo già discusso l'uso del flag di carry nella registrazione di un risultato superiore a 255, che è più di quanto possa essere contenuto in otto cifre binarie. Abbiamo notato anche che, lavorando con numeri con segno ad un solo byte, dove l'ottavo bit è usato per registrare il segno del numero, il flag di carry non ci serve poichè la parte del byte che forma il valore effettivo comprende le prime sette cifre binarie.

Facciamo l'esempio dell'addizione di due numeri +127 e +127, per i quali la rappresentazione binaria con segno sarebbe:

```
01111111
+
01111111
```

con lo zero, nella posizione più a sinistra, indicante che entrambi i numeri sono positivi. La CPU tratterà i due numeri come farebbe con qualsiasi altri; essa non ha alcun modo operativo speciale per i numeri con segno, essi sono una creazione del programmatore. Così i due 127 saranno sommati per produrre 254, cioè 11111110 in binario. Questo è giusto fino a quando si trattano numeri senza segno, ma non ha senso se operiamo con numeri con segno, poichè l'"1" nella posizione più a sinistra significa che il numero è negativo. Infatti, con il metodo convenzionale di lettura dei numeri con segno, il valore prodotto è -126. Chiaramente, c'èqualcosa di sbagliato ed è altrettanto chiaro che c'è stato un carry dalla settima posizione all'ottava, che contiene il bit del segno.

Quando capita ciò, il flag di overflow è settato ad uno per indicare sia che il bit del segno è stato invertito, sebbene non doveva esserlo, sia che si deve sommare 128 al numero depositato nell'accumulatore. Il flag di overflow agisce quindi come una cifra binaria in più per l'accumulatore, ma, invece di fornire una nona cifra per un numero senza segno, come nel caso del flag di carry, si comporta come come un ottavo bit per i numeri con segno, permettendo così di trattare valori da -256 a 255, a condizione, naturalmente, che il programmatore abbia ideato il programma in modo tale da tenere in conto il bit di overflow.

Istruzioni che si riferiscono al flag di overflow

BVC (Branch if Overflow Clear) : l'esecuzione del programma salta ad un punto specifico se il flag di overflow è settato a zero quando si incontra questa istruzione.

BVS (Branch if Overflow Set) : l'esecuzione del programma salta ad un punto specificato se il flag di overflow è settato ad uno quando si incontra questa istruzione.

CLV (CLear Overflow) : setta il flag di overflow a zero.

Questa istruzione è usata quasi esclusivamente in routines che sono destinate a trattare numeri con segno e non ha alcun uso specifico nella programmazione. Potrete trovare la sua trattazione nei capitoli sui calcoli aritmetici con segno.

Dimostrazone di CLV.

```
  PC          SR  AC  XR  YR  SP
; 1C00        30  00  00  00  F0
> 1C8B        00  00  00  00
> 1C8B        00  00  00  00

A  1C00       A9  00  LDA #$FF
A  1C02       48      PHA
A  1C03       28      PLP
A  1C04       B8      SEI
A  1C05       00      BRK

  PC          SR  AC  XR  YR  SP
; 1C07        BF  FF  00  00  F0
> 008B        00  00  00  00
> 1C8B        00  00  00  00
```

Commento:
1C00-1C03: Tutto questo vi dovrebbe essere familiare dall'ultimo capitolo,

poichè è stato usato per dimostrare il metodo per caricare un valore nello status register. In questo caso, come nell'esempio precedente, si pone $FF, cioè 255 in decimale, nel registro, settando così ogni bit.
1C04: La sola istruzione CLV è sufficiente per azzerare il flag di overflow al bit 6, come indicato dal valore di $BF, cioè $FF-2^6, contenuto nello status register.

Il Flag di Sign

Funzione: Riflette lo stato della cifra binaria più a sinistra del valore creato dall'ultima operazione del chip. Esso è interessato dalla stessa serie di operazioni che è stata data in precedenza per il flag di zero. La domanda a cui il flag di sign risponde è: "L'ultima operazione ha creato un valore che potrebbe essere trattato come negativo, cioè ha il valore del bit sette settato".

Il flag di sign è un altro aiuto nell'uso dei calcoli aritmetici con segno, sebbene con applicazioni molto più vaste nelle tecniche generali di programmazione. Il suo scopo è semplicemente il riflettere cosa sia successo alla cifra binaria più a sinistra di un valore che è stato appena creato. Se il numero generato è negativo, il flag di sign saràsettato ad uno, proprio come è posto ad uno il bit più a sinistra del numero.

Sebbene paragonato al flag di overflow, il flag di sign ha, comunque, una ampia gamma di usi. Per esempio, un semplice metodo per il conteggio del numero di ripetizioni di una azione è inserire nel registro X un valore che è minore di uno del numero delle ripetizioni, poi decrementare (sottrarre uno) dal totale ogni volta che l'azione desiderata viene eseguita. Quando il registro X passerà da zero a 255, il flag di sign sarà settato e potrà essere usato per controllare lo scorrimento del programma.

Istruzioni che si riferiscono al flag di sign

BPL (Branch on PLus) : l'esecuzione del programma è indirizzata ad un punto specificato se, quando si incontra questa istruzione, il flag di sign è settato a zero.

BMI (Branch on MInus) : l'esecuzione del programma è indirizzata ad un punto specificato se, quando si incontra questa istruzione, il flag di sign è settato ad uno.

Dimostrazioni dell'uso di queste istruzioni saranno date nel capitolo del controllo di programma.

Dimostrazione di come settare il flag di sign.

```
  PC        SR  AC  XR  YR  SP
; 1C00       30  00  00  00  F0
```

```
> 1C8B     00  00  00  00
> 1C8B     00  00  00  00

A  1C00    A9  00     LDA #$00
A  1C02    48         PHA
A  1C03    28         PLP
A  1C04    A9  88     LDA #$88

  PC       SR  AC  XR  YR  SP
; 1C08     B0  88  00  00  F0
> 008B     00  00  00  00
> 1C8B     00  00  00  00
```

Commento:
1C00-1C03: Ancora una volta una routine che vi dovrebbe essere familiare dall'ultimo capitolo; il suo effetto è di azzerare tutti i bits nello status register.
1C04: L'accumulatore è caricato con il valore $8B, cioè 136 in decimale. Nella forma binaria questo numero è 10001000 e, poichè il bit più a sinistra, il bit sette, è settato, il bit di sign al bit sette nello status register deve corrispondere a questo valore inserito nell'accumulatore, risultando, nello status register, $B0 cioè $30+2^7.

Conclusione

Non siate preoccupati se, a questo punto, vi accorgete che questo capitolo non era il più semplice tra quelli trattati. Ogni introduzione al codice macchina cade in queste situazioni, dove alcune informazioni devono essere date prima che si sia in grado di utilizzarle. Nei prossimi capitoli troverete una miriade di comandi e di tecniche che dipendono interamente dai flag che abbiamo appena descritto e vi sarà comodo tornare indietro, più di una volta, per rinfrescarvi le idee a riguardo di cosa registrino i flag. Non è certo molto semplice, ma è l'unica via per proseguire.

CAPITOLO 9

ARITMETICA SEMPLICE

L'aritmetica è sempre basilare in ogni libro sul codice macchina, non solo a causa della sua importanza, ma anche perchè i metodi relativamente gravosi che il codice macchina impone ne richiedono sostanziali spiegazioni. In questo capitolo inizieremo col dare un'occhiata ad alcuni semplici calcoli aritmetici come l'addizione o la sottrazione di piccoli numeri ed ai comandi che realizzano ciò, lasciando ai capitoli successivi l'introdurci in campi più complessi.

I comandi compresi in questo capitolo includono:

INC	DEC	INX
DEX	INY	DEY
ADC	SBC	

INCrement e DECrement: addizione e sottrazione di uno

La forma più semplice di calcolo aritmetico che il chip 7501 è in grado di compiere è addizionare o sottrarre uno ad un valore in un registro. Questo, a prima vista, può sembrare elementare, ma è fondamentale per molte tecniche di programmazione in codice macchina, come la creazione di loop paragonabili ai BASIC FOR.....NEXT usati per ripetere compiti un certo numero di volte. La possibilità di addizionare o sottrarre uno è così importante ed è usata così frequentemente, che il chip 7501 ha comandi speciali incorporati proprio per realizzare questi semplici calcoli, evitando così al programmatore in codice macchina la necessità di specificare se il numero deve essere sommato o sottratto.

Ci sono tre forme di comando per addizionare o sottrarre uno. I comandi per addizionare uno ad un valore sono conosciuti come istruzioni "increment" e sono:

INC : addiziona uno al valore presente in uno specificato indirizzo di memoria.
INX : addiziona uno al valore presente nel registro X.
INY : addiziona uno al valore presente nel registro Y.

La forma contraria di queste sono le istruzioni "decrement":

DEC : sottrae uno dal valore presente in uno specificato indirizzo di memoria.
DEX : sottrae uno dal valore presente nel registro X.
DEY : sottrae uno dal valore presente nel registro Y.

Quando usiamo le istruzioni che si riferiscono ai registri X ed Y, non c'è altro da aggiungere, nessun indirizzo o valore ulteriore; esse sono conosciute come istruzioni ad un solo byte, avendo tutte le informazioni che servono alla CPU (come il nome del registro) inserite dentro esse. Notate che, purtroppo, non c'è alcuna istruzione singola in grado di incrementare o decrementare un valore nell'accumulatore. Questo può essere ottenuto solo con l'uso delle istruzioni di addizione e sottrazione, che saranno discusse in seguito. Quando incrementiamo o decremetiamo il valore contenuto in una locazione di memoria specificata (INC e DEC), si dovrà specificare un indirizzo e per questo sono disponibili parecchi modi di indirizzamento; si veda la sezione successiva in questo capitolo.

I flag e le istruzioni INC o DEC

Quando usiamo le istruzioni di incremento o decremento, bisogna tener presente che esse sono diverse dalle forme più complesse di addizione e sottrazione, infatti le istruzioni di addizione o sottrazione hanno, inserita in esse, la capacità di settare e azzerare il flag di carry. L'incremento od il decremento non hanno effetto sul flag di carry, sebbene essi interessino i flag di sign e di zero in accordo alle regole descritte nel capitolo sui flag.

Per questa ragione, i flag di zero e di sign sono usati frequentemente per controllare i risultati dell'uso delle istruzioni di incremento e decremento. Se una istruzione di incremento è applicata ad un registro contenente il valore 255, il registro passerà attraverso lo zero. Questo cambiamento non sarà reso noto, come già detto, dal flag di carry, che rimarrà in qualunque stato fosse prima che l'incremento avvenisse, ma sarà annotato dal flag di zero, poichè si è generato, nel registro che è stato incrementato, uno zero.

Quando si effettua un decremento su un registro che, correntemente, si trova a zero, il valore nel registro diventerà 255. Il cambiamento, nella direzione opposta, può essere annotato con l'uso del flag di sign, che riprende lo stato dell'ottavo bit, essendo questo passato da zero, quando l'intero byte era a zero, ad uno, quando il byte diventa 255.

Modi di indirizzamento per l'uso con INC e DEC

Mentre INX, INY, DEX e DEY non hanno bisogno di un indirizzo che li accompagni, poichè specificano il nome del registro al quale si applicano, non

potete incrementare o decrementare un valore contenuto in un particolare indirizzo di memoria, senza, prima di tutto, specificare su quale indirizzo di memoria intendiate operare. Per ottenere questo, dovete unire all'istruzione INC o DEC un indirizzo, e questo può prendere forme differenti. Non possiamo, per ragioni di spazio, discutere nei dettagli le forme dei modi di indirizzamento, poichè essi sono già stati trattati nel capitolo sull'indirizzamento. Se non avete ben chiaro cosa significhino i diversi modi di indirizzamento, fate uno sforzo e tornate indietro a dare un'altra occhiata al capitolo precedente.

Generalmente, la forma di INC e DEC è l'istruzione seguita da un indirizzo contenente un valore sul quale si deve eseguire l'incremento od il decremento.

I modi di indirizzamento disponibili per l'uso con INC sono:

```
Zero Page:
INC $FF          : E6            : E6 FF
Assoluto:
INC $02FF        : EE            : EE FF 02
Indicizzato Zero Page:
INC $FF,X        : F6            : F6 FF
  Indicizzato Assoluto:
INC $02FF,X      : FE            : FE FF 02
```

Esempio dell'uso di INC.

```
  PC          SR   AC   XR   YR   SP
; 1C00         30   00   00   00   F0
> 008B         AA   00   00   00
> 1C8B         00   00   00   00
A  1C00      : E6   8B    INC $8B
A  1C02         00        BRK

  PC          SR   AC   XR   YR   SP
; 1C04         B0   00   00   00   F0
> 008B         AB   00   00   00
> 1C8B         00   00   00   00
```

Commento:

1C00: Una sola istruzione mostra l'effetto di INC. Settate il valore della locazione di memoria $8B con qualsiasi valore desiderato e poi fate eseguire il codice macchina. Esperimenti con differenti valori in $8B vi daranno un'idea dell'effetto di INC sui flag.

I modi di indirizzamento disponibili per l'uso con DEC sono:

```
Zero Page
DEC $FF          C6              C6 FF
```

```
Assoluto
DEC $02FF       CE              CE FF 02
Indicizzato Zero Page
DEC $FF,X       D6              D6 FF
Indicizzato Assoluto
DEC $02FF,X     DE              DE FF 02
```

Esempio dell'uso di DEC.

```
  PC            SR  AC  XR  YR  SP
; 008B           AA  00  00  00  00
> 1C8B           00  00  00  00

A  1C00         C6  8B      DEC $8B
A  1C02         00          BRK

  PC            SR  AC  XR  YR  SP
; 1C04           B0  00  00  00  F0
> 008B           00  00  00  00
> 1C8B           00  00  00  00
```

Commento
1C00: Ancora una volta, un'unica istruzione. La sperimentazione con il valore in $8B vi dimostrerà l'effetto di DEC sui flag.

Addizione con ADC

Passiamo ora dalla forma più semplice di addizione , che fa uso delle istruzioni incremento, al metodo più generale, che fa uso dell'istruzione ADC, cioè "ADd with Carry" (Addizionare con carry). Nell'uso di ADC saremo, in qualche modo, più limitati che nell'uso delle istruzioni incremento dal momento che il calcolo effettivo può essere eseguito solamente su un valore contenuto nell'accumulatore. Per questa ragione, ADC è usata raramente da sola, ma quasi sempre assieme ad altre istruzioni che trasferiscono valori da altre aree in memoria all'accumulatore, in modo che possano essere elaborati, e che, poi, riportino di nuovo i valori in memoria. Abbiamo già visto i modi in cui i valori possono essere trasferiti, così gli esempi dati in seguito comprenderanno trasferimenti da e verso l'accumulatore.

L'istruzione ADC può essere usata con molti modi di indirizzamento, ma il suo effetto comune è sempre la specificazione di un valore o di un indirizzo al quale si può trovare un valore, e poi il prendere quel valore ed addizionarlo al contenuto di quel momento dell'accumulatore. Inoltre, il termine "C" dell'istruzione sta a ricordare che il flag di carry è parte integrante del

processo. Se il flag di carry è settato, quando l'addizione inizia, sarà aggiunto un "1" extra alla somma nell'accumulatore. Più tardi vedremo che questo è essenziale per i calcoli aritmetici che coinvolgono numeri più lunghi di un unico byte. Quando si tratta di addizionare numeri ad un solo byte, se il flag di carry è settato per qualche motivo irrilevante prima che si inizi l'addizione, si ha come conseguenza un risultato errato. Per questa ragione troverete sempre che, nelle applicazioni di addizioni normali ad un solo byte, l'istruzione CLC si usa di preferenza per assicurare che il flag di carry sia azzerato prima dell'esecuzione dell'addizione.

I flag interessati dall'uso di ADC sono quelli di Carry, di Zero, di Overflow e di Sign.

I modi di indirizzamento disponibili per l'uso con ADC sono:

```
Zero page:
ADC $FF          : 65          : 65 FF
Assoluto:
ADC $02FF        : 6D          : 6D FF 02
Immediato:
ADC $FF          : 69          : 69 FF
Pre-indicizzato indiretto:
ADC ($FF,X)      : 61          : 61 FF
Post-indicizzato indiretto:
ADC ($FF),Y      : 71          : 71 FF
Indicizzato Zero page,X
ADC $FF,X        : 75          : 75 FF
Indicizzato Assoluto,X:
ADC $02FF,X      : 7D          : 7D FF 02
Indicizzato Assoluto,Y:
ADC $02FF,Y      : 79          : 79 FF 02
```

Esempio dell'uso di ADC.

```
  PC          SR  AC  XR  YR  SP
; 1C00        30  00  00  00  F0
> 008B        40  00  30  00
> 1C8B        00  00  00  00

A  1C00       18      CLC
A  1C01       A5  8B  LDA $8B
A  1C03       65  8D  ADC $8D
A  1C05       00      BRK

  PC          SR  AC  XR  YR  SP
; 1C07        30  70  00  00  F0
> 008B        40  00  30  00
> 1C8B        00  00  00  00
```

Commento:
1C00: Il flag di carry è dapprima azzerato, per la ragione spiegata in precedenza.

1C01-1C03: Il contenuto della locazione di memoria $8B è caricato nell'accumulatore e poi è addizionato al contenuto della locazione di memoria $8D. Il risultato dipende dai valori che scegliete per le due locazioni.

Sottrazione con SBC

Il processo contrario ad ADC è SBC "SuBtract with Carry" (Sottrarre con carry), che sottrae un valore specificato dal contenuto dell'accumulatore. Proprio come ADC, SBC è normalmente usata con istruzioni che trasferiscono un valore nell'accumulatore per l'elaborazione e poi lo ritrasferiscono nuovamente.

I modi di indirizzamento con i quali SBC funzione sono gli stessi di ADC e sono riportati in seguito. L'elemento "carry" dell'istruzione si riferisce al fatto che, se il flag di carry, quando inizia l'istruzione, è a zero, sarà sottratto un "1" extra dal reale risultato. Notate che questo è il contrario di ciò che è stato mostrato, con molta cura, per ADC. Ancora una volta, per questa ragione, il flag di carry è settato con SEC prima che si esegua una sottrazione ad un solo byte.

I flag intressati da SBC sono gli stessi di ADC

I modi di indirizzamento disponibili per l'uso con SBC sono:

Zero page		
SBC $FF	E5	E5 FF
Assoluto		
SBC $02FF	ED	ED FF 02
Immediato		
SBC $FF	E9	E9 FF
Pre-indicizzato indiretto		
SBC ($FF,X)	E1	E1 FF
Post-indicizzato indiretto		
SBC ($FF),Y	F1	F1 FF
Indicizzato Zero page,X		
SBC $FF,X	F5	F5 FF
Indicizzato Assoluto,X		
SBC $02FF,X	FD	FD FF 02
Indicizzato Assoluto,Y		
SBC $02FF,Y	F9	F9 FF 02

Esempio dell'uso di SBC.

```
  PC        SR  AC  XR  YR  SP
; 1C00       30  00  00  00  F0
> 1C8B       40  00  30  00
> 1C8B       00  00  00  00

A 1C00       38      SEC
A 1C01       A5  8B  LDA $8B
A 1C03       E5  8D  SBC  $8D
A 1C05       00      BRK

  PC        SR  AC  XR  YR  SP
; 1C07       31  10  00  00  F0
> 008B       40  00  30  00
> 1C8B       00  00  00  00
```

Commento

1C00 SEC è usata per settare il flag di carry ed assicurare un risultato esatto. 1C01-1C03: Il contenuto della locazione di memoria $8B è caricato nell'accumulatore, poi il contenuto della locazione di memoria $8D è sottratto. Sperimentate con valori diversi nelle due locazioni ed osservate l'effetto sul risultato e sui flag.

Conclusione

A questo punto vi sarete di certo accorti che il testo è diventato chiaramente meno "amichevole". Sono lontane le semplici pagine sul comportamento, interno ed esterno, del chip. Improvvisamente vi si chiede di capire il concetto di addizione e di sottrazione in strani modi chiamati preindicizzato indiretto,X o qualcosa di ugualmente anomalo.

In effetti, questo capitolo è ciò che, nel libro, determinerà se siete sulla strada giusta verso la comprensione del codice macchina. Una cosa è spiegare ogni concetto, passare attraverso liste di modi di indirizzamento mettendo i puntini sulle i ed i trattini alle t, discutere l'idea dell'addizione e di come il metodo di addizione della CPU possa essere simulato su un pezzo di carta o trattare nei dettagli come debba essere interpretato ogni flag. Tutto ciò è stato fatto nella prima parte del libro.

Altra cosa è spiccare un balzo dal conoscere che, per esempio:

a) indirizzamento indiretto significa ottenere un indirizzo da un'area in memoria, per poi ottenere un valore dal byte in memoria a quell'indirizzo,
b) LDA significa inserire un valore nell'accumulatore,

c) ADC significa addizionare un valore a quello che è già nell'accumulatore
e
d) il flag di carry influenza ed è influenzato dalla forma del risultato dell'addizione per vedere le istruzioni del linguaggio assembly:

```
LDA #$F0
ADC $01F0
```

sapendo che il numero $F0 deve essere caricato nell'accumulatore e che, poi, deve essere addizionato il contenuto di un byte di memoria il cui indirizzo è indicato dal contenuto dei due byte che sono inseriti alla locazione di memoria $01F0.

Inoltre, questo capitolo introduce un argomento che confonde spesso i principianti del codice macchina: vale a dire che, mentre un'istruzione in linguaggio assembly mostra chiaramente con il suo formato l'istruzione che si sta per usare ed il modo di indirizzamento entrambi sotto forma di termini separati, quando si lavora direttamente in codice macchina, un singolo codice rappresenta sia l'istruzione che il modo di indirizzamento.

Sta a voi quindi fare lo sforzo di assimilare i modi di indirizzamento e le poche istruzioni del codice macchina del 7501, cercando di fare in modo che la conoscenza dei due campi cominci a fondersi. A questo punto nessun libro può fare il lavoro per voi poichè spiegare completamente i modi di indirizzamento a fianco di ogni istruzione che ne fa uso, richiederebbe centinaia di pagine in più col risultato di ottenere un volume simile all'enciclopedia EI del Gruppo Editoriale Jackson e non un manuale per neofiti.

Così, prima di continuare, accertatevi di aver capito i differenti modi di indirizzamento ed il modo in cui essi si riferiscono a singole istruzioni come ADC od SBC. Se così non fosse, tornate a dare un'occhiata ai capitoli precedenti.

CAPITOLO 10

ARITMETICA A DUE BYTE

Imparato come addizionare numeri che possono essere contenuti in un solo byte o come sottrarre un tale numero da un altro, passiamo all'argomento, leggermente più complesso, dei numeri che richiedono due o più byte. Notate che abbiamo detto leggermente più complesso, e quel "leggermente" è esattamente la parola giusta poichè, anche se non ve ne rendete bene conto, conoscete già tutte le tecniche necessarie per addizionare o sottrarre numeri a due byte.

Addizione di numeri a due byte

Addizionare numeri a due byte è proprio come addizionare numeri a due cifre decimali. Quando si addizionano due numeri ad una sola cifra, l'unica complicazione è che, se c'è un carry, esso deve essere scritto nella colonna più a sinistra del risultato. Con numeri a due cifre il carry deve essere sommato nella seconda colonna della somma. Per chiarire il punto, ecco due esempi:

```
   6          16
 +7         +17
----       ----
  13          3
  ^           1      <-----CARRY
CARRY        2
            ----
            33
```

Il principio è esattamente lo stesso quando si tratta di addizionare in codice macchina, e conosciamo già come addizionare ogni carry, poichè viene fatto automaticamente quando si usa il comando ADC. Per addizionare valori a due byte, la procedura è:

1) Caricare l'accumulatore con il low byte di uno dei due valori.
2) Azzerare il flag di carry, come all'inizio di ogni addizione.
3) Usare ADC per sommare il low byte del secondo valore.
4) Immagazzinare il risultato in un'area.
5) Caricare l'accumulatore con il successivo byte di uno dei due valori.
6) Usare ADC per sommare il successivo byte del secondo valore.

Alla fine di questa procedura, il risultato è che voi avete un low byte immagazzinato da qualche parte, un high byte nell'accumulatore, e, forse, un "1" nel flag di carry che rappresenta non 256 ma 256^2, cioè 65536, cioè 10000 in esadecimale.

Se volete andare oltre i numeri a due byte, dovete semplicemente ripetere le fasi 4-6 fino ad esaurimento dei byte nei due numeri. Alla fine del processo, se c'è un numero nel flag di carry, esso rappresenta 256 alla potenza del numero di byte in ognuno dei valori che si sono addizionati.

Esempio dell'addizione di numeri a due byte.

```
  PC          SR  AC  XR  YR  SP
; 1C00         30  00  00  00  F0
> 1C8B         40  99  30  55
> 1C8B         00  00  00  00

A  1C00        18      CLC
A  1C01        A5  8B  LDA $8B
A  1C03        65  8D  ADC $8D
A  1C05        AA      TAX
A  1C06        A5  8C  LDA $8C
A  1C08        65  8E  ADC $8E
A  1C0A        00      BRK

  PC          SR  AC  XR  YR  SP
; 1C0C         B0  EE  70  00  F0
> 008B         40  99  30  55
> 1C8B         00  00  00  00
```

Commento:
1C00-1C03: I due low bite dei numeri, che sono contenuti rispettivamente nelle locazioni di memoria $8B e $8D, vengono addizionati.
1C05: Poichè deve essere fatta una ulteriore addizione, il risultato è trasferito, per tenerlo al sicuro, nel registro X.
1C06-1C08: I due high byte dei numeri, che sono contenuti rispettivamente nelle locazioni di memoria $8C e $8E, vengono ora addizionati. Notate che il flag di carry non è azzerato per questa seconda addizione poichè esso contiene l'unico indizio che ci sia un carry dall'addizione dei low byte, e questo è essenziale per un risultato corretto. Il risultato dell'high byte è contenuto nell'accumulatore ed il low byte nel registro X.

Il risultato finale dipende dai valori che scegliete di immettere nelle locazioni di memoria $8B-$8E. Ricordate che le locazioni di memoria sono usate dal programma come segue:

$8B: low byte del valore numero 1
$8C: high byte del valore numero 1

$8D: low byte del valore numero 2
$8E: high byte del valore numero 2

Se, per esempio, ponete 1, 2, 3 e 4 nelle locazioni, dovrete addizionare (1 +256*2) e (3 + 256*4).

Sottrazione di numeri a due byte

Quando si tratta di sottrarre, le cose sono altrettanto semplici a patto che vi ricordiate che il flag di carry opera in modo contrario, cioè che quando il flag di carry è a zero, deve essere chiesto un "borrow" dal successivo byte alla sinistra.

La procedura per la sottrazione a due byte è la seguente:

1) Settare il flag di carry, come all'inizio di ogni sottrazione.
2) Caricare l'accumulatore con il low byte del numero dal quale si deve sottrarre.
3) Usare SBC per sottrarre il low byte dell'altro valore.
4) Immagazzinare il risultato in un'area.
5) Caricare l'accumulatore con il byte successivo del numero dal quale si deve sottrarre.
6) Usare SBC per sottrarre il byte successivo dell'altro valore.

Notate che, mentre nel caso di ADC non ha importanza da quale dei due numeri siano presi i byte da immettere nell'accumulatore (potete alternarli di byte in byte), nel caso di SBC ciò ha la sua importanza. Qualsiasi valore che inserite nell'accumulatore sarà quello da cui si sottrarrà. Avendo così iniziato con il low byte di uno dei due numeri nell'accumulatore, ogni byte susseguente inserito in esso dovrà essere dello stesso numero.

Alla fine del processo vi ritroverete con il low byte del risultato immagazzinato in un'area di memoria e l'high byte nell'accumulatore. Se il flag di carry è a zero significa che c'è un borrow, ed il risultato effettivo è minore di 65536 del risultato registrato dai byte low ed high.

Anche questa volta,la sottrazione di numeri con più di due byte, richiede semplicemente la ripetizione delle fasi 4-6 fino a che tutti i byte sono esauriti. Se il flag di carry, alla fine, è zero, il risultato richiede la sottrazione di 256 elevato alla potenza uguale al numero di byte di ognuno dei due valori.

Esempio di sottrazione a due byte.

```
  PC       SR  AC  XR  YR  SP
; 1C00      30  00  00  00  F0
> 008B      40  99  30  55
> 1C8B      00  00  00  00
```

```
A  1C00    38       SEC
A  1C01    A5  8B   LDA  $8B
A  1C03    E5  8D   ABC $8D
A  1C05    AA       TAX
A  1C06    A5  8C   LDA  $8C
A  1C08    E5  8E   SBC $8E
A  1C0A    00       BRK

   PC      SR  AC   XR  YR  SP
;  1C0C    71  44   10  00  F0
>  008B    40  99   30  55
>  1C08    00  00   00  00
```

Commento:
1C00-1C03: Sottrazione del low byte di un numero dal low byte dell'altro. Il risultato è immagazzinato nel registro X.

1C06-1C08: Sottrazione dei due high byte. Notate ancora una volta come non si sia toccato il flag di carry quando viene inserito l'high byte, poichè il suo valore indica che c'è un borrow dalla sottrazione precedente. Il risultato prevede l'high byte conservato nell'accumulatore ed un low byte nel registro X.

CAPITOLO 11

ARITMETICA CON SEGNO

Dopo aver affrontato le basi dell'addizione e della sottrazione, dedichiamoci ad un argomento un pò più complicato cioè alle operazioni con i numeri con segno, cioè numeri che possono essere sia positivi che negativi. Abbiamo già toccato un paio di volte i concetti relativi ai numeri con segno nei capitoli di apertura e nel discorso sui flag, quindi sarete a conoscenza che sono meno semplici da trattare dei numeri senza segno visti negli ultimi due capitoli.

La ragione per cui le cose non sono così semplici è che il chip 7501 non è progettato per riconoscere o trattare numeri con segno. Tali numeri sono una "convenzione", una decisione del programmatore, per cui certi numeri devono essere trattati come se fossero potenzialmente positivi o negativi. Quando il chip accetta un numero come 255, 11111111 in binario, esso è trattato proprio come 255, è solo il programmatore che è in grado di intervenire per dire: "questo numero non è realmente 255 ma -1, poichè l'ottava cifra binaria viene usata per indicare che il numero è negativo e quindi opereremo nella notazione "complemento a due". In altre parole, lavorare con i numeri con segno richiede appunto questa operazione.

A questo punto, se siete interessati al modo con cui sono trattati i numeri con segno, ripassatevi l'ultimo paragrafo. Abbiamo visto precedentemente l'uso dell'ottava cifra binaria per sostituire il segno più o meno, ed anche il calcolo aritmetico del complemento a due. Se qualcosa non vi è chiaro, tornate indietro e rinfrescatevi la memoria, altrimenti non c'è dubbio che vi ritroverete impantanati in quello che segue.

Selezione della posizione per il bit di segno

La prima cosa che il programmatore deve fare, quando opera con numeri con segno, è costruire il formato dei numeri. Fino ad ora, nel trattare i numeri con segno abbiamo assunto come cifra del segno l'ottava cifra del byte. Questa è soltanto una convenzione che il programmatore sceglie di osservare, sostenuto dal fatto che il flag di overflow può essere usato per registrare ogni variazione dell'ottava cifra, come abbiamo visto nel capitolo sui flag. Il programmatore potrebbe decidere, in ugual modo, di usare la prima cifra come bit di segno sebbene, nella realtà, questo presenti problemi pratici.

Ciò che vogliamo sottolineare è che sta a voi e non al 7501 "formare" i numeri con segno. Questo implica una decisione da parte vostra circa la

posizione del bit di segno, così che voi possiate programmare tenendo conto di essa. Di solito si pone il bit di segno in ottava posizione. Passiamo ora alla parte difficile. Bene, cercate di comporre il numero meno 255 (un numero binario ad otto cifre), se avete messo da parte una cifra del byte ad otto cifre per rappresentare il segno del numero.

La risposta è, naturalmente, che non potete farlo e da tale risposta deve essere tratta una lezione importante. Prima di intraprendere l'uso dei numeri con segno e di scrivere tutte le routine basate sull'immissione del bit di segno in una certa posizione, è necessario stabilire la gamma da usare. Nei capitoli precedenti, abbiamo osservato che un numero con segno a un solo byte può cadere solo tra -128 e +127. Questo range è piuttosto limitato per molti degli scopi dei programmi in codice macchina seri, così numeri con segno ad un solo byte sono usati raramente. Numeri con segno a due byte permettono un range da -32768 a +32767, che è sicuramente più adeguato per molte applicazioni. Quando passate ad applicazioni più complesse, come moltiplicare numeri a molti byte (un compito orrendo, che noi non descriveremo), vi potrà capitare di lavorare con numeri con segno che sono lunghi quattro o più byte.

Per il momento, tuttavia, ci limitiamo a numeri di due byte, ponendo la cifra di segno nella posizione disponibile più a sinistra, l'ottava cifra dell'high byte del numero. Ogni valore creato sarà costituito come segue:

HIGH BYTE	LOW BYTE
[(CIFRA DI SEGNO) (7 CIFRE)]	[8 CIFRE]

Una volta deciso, questo formato deve considerarsi come fisso, altrimenti dovete scrivere ulteriori routine in codice macchina per maneggiare numeri aventi altri formati. Questo significa che dovranno essere evitati numeri fuori dal range da -32768 a +32767 ed i numeri compresi nel range da -128 a +127 dovranno essere scritti a due byte, anche se essi potrebbero essere inseriti in un solo byte. Inoltre, ogni routine che scriverete per trattare numeri con segno, dovrà essere divisa in due parti, una per elaborare il low byte, che non ha cifre di segno, ed un'altra sezione per operare sull'high byte, che l'avrà.

Fin qui, tutto bene, ma avrete notato che, finora, abbiamo completamente tralasciato il problema di come creare un numero con segno, a seconda che il bit di segno sia nell'ottava posizione o nella sesta.

Creazione di un numero con segno

La complessità della creazione di un numero con segno varia a seconda che il numero sia positivo o negativo:

1) Numeri positivi: abbiamo già visto che possiamo operare solo con numeri nel range da -32768 a +32767, numeri che si inseriscono in due

byte con la cifra più significativa riservata al segno del numero. La rappresentazione di un numero positivo è uno zero nella cifra binaria più alta e un numero compreso entro 32767 (il più grande numero positivo). Il programma deve trattare il numero come se fosse dotato di segno.

2) Per creare un numero con segno negativo, è sufficiente una semplice regola. Prima prendete il numero che volete rappresentare come negativo. In questo caso prenderemo come esempio 4232, che in rappresentazione binaria senza segno sarebbe: 00010000 10001000 La prima azione consiste nell'invertire tutti i bit, col seguente risultato: 11101111 01110111 e poi addizionare uno: 11101111 01111000

C'è un secondo metodo, che è leggermente più rapido, sebbene si presti di più ad un "user error" (errore dell'utente). La prima fase è prendere la rappresentazione binaria del numero, nel nostro caso: 00010000 10001000

Partendo dalla destra del numero, scorrete le cifre del numero fino a che non ne incontrate una che contenga qualcosa che non sia "0". Nel nostro caso, la quarta cifra da destra è "1". Trascurate quella cifra, ma dalla cifra successiva in avanti, muovendovi verso sinistra, invertite ogni cifra, inclusa quella più a sinistra. Il risultato di questo, nel nostro caso è: 11101111 01111000 esattamente lo stesso del metodo precedente.

Ottenuto il vostro numero con segno in binario, è vostro compito tradurlo in decimale od in esadecimale, in accordo col metodo di immissione del programma in codice macchina nella memoria, che voi state usando. La forma esadecimale è la più facile poichè tutto quello che dovete fare è prendere ogni gruppo di quattro cifre binarie, cominciando dalla sinistra, e poi tradurli in una cifra esadecimale nel range 0-F (0-15 in decimale).

Dopo aver letto tutto questo avrete capito la fondatezza di quello che abbiamo detto in precedenza a proposito dei vantaggi di un assembler che sia in grado di maneggiare direttamente numeri con segno.

Addizione di numeri con segno

L'addizione di numeri con segno a due byte è simile all'addizione dei numeri senza segno. Il problema sorge se generate un numero che non può essere immagazzinato nelle 15 cifre binarie disponibili, una volta che si è riservato un bit al segno. Quando questo accade la CPU, non sapendo che state operando con numeri dotati di segno, opera un carry di uno e lo immette nella sedicesima posizione, all'estrema sinistra del numero. Qualsiasi cosa il vostro bit di segno doveva essere, è ora l'opposto. Problema numero uno .

Il problema numero due è un pò più semplice: cercate di addizionare i due numeri con segno seguenti, rappresentanti entrambi -127: 10000001 e 10000001. Il risultato dovrebbe essere: 00000010, con il flag di carry settato, ad indicare che bisogna considerare un extra di 256. Ma noi non stiamo trattando numeri senza segno; la cosa che è accaduta in realtà è che le due

cifre di segno sono state addizionate e si sono cancellate una con l'altra. Il numero risultante è +2, che è sicuramente corretto.

Entrambi questi problemi sono il risultato di bit portati dentro o fuori della cifra di segno, ed è qui che il flag di overflow, che abbiamo già esaminato, entra in gioco. Esaminando il flag di overflow, quando due numeri con segno vengono addizionati, possiamo vedere immediatamente se la cifra di segno è corretta o meno. Qui la regola è semplice.

Iniziamo con zero:

se qualcosa è caricato nel bit di segno, addizionare 1. Se qualcosa è scaricato dal bit di segno, addizionare 1.

Se il risultato è zero o due, il bit di segno è corretto; se il risultato è uno, il bit di segno è inesatto.

Inoltre, il modo con cui i numeri complemento a due sono costruiti fa sì che il flag di overflow sia un segno infallibile per vedere se un numero è troppo grande per le sette o le 15 cifre binarie disponibili; cioè se il flag di overflow è settato, il numero è fuori dal range e c'è stato un carry fuori del numero. A prima vista, sembra inesatto, ma se addizioniamo due numeri positivi con segno, 127 e 127, il risultato risulterà troppo grande per il range da -128 a +127:

```
0 1 1 1 1 1 1 1
0 1 1 1 1 1 1 1
---------------
1 1 1 1 1 1 1 0
```

Ecco che il flag di overflow è settato ad uno, indicando che il segno è sbagliato e che il numero è fuori del range. Se prendiamo le stesse cifre decimali e cambiamo lo zero all'inizio con un uno:

```
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
---------------
1 1 1 1 1 1 1 0 (ed 1 nel flag di carry)
```

In questo caso c'è stato un carry verso il bit di segno ma anche un carry nel flag di carry. In accordo alle regole sopra riportate, il flag di overflow risulta azzerato ed il numero non sarà fuori del range. Infatti, 11111111 in complemento a due significa 1 in decimale, quindi non si genera alcun overflow. Mettiamo il caso di introdurre valori che potrebbero portare ad un overflow, diciamo -65 più -65, le cui notazioni in complemento a due danno un 1 nella posizione della settima cifra per avere un overflow nell'ottava:

```
1 0 1 1 1 1 1 1
1 0 1 1 1 1 1 1
---------------
0 1 1 1 1 1 1 0
```

Qui, l'overflow nel bit di carry non è bilanciato da un carry nel bit di segno, quindi il flag di overflow sarà settato, indicando correttamente che il risultato non è "-2" ma "-2 più -128" cioè -130. Non è necessario che voi ricordiate tutto questo come cosa primaria, ma solo che comprendiate che il flag di overflow significa due cose:

1) Una indicazione che il bit di segno di un numero è inesatto.
2) Che il risultato è troppo grande per il numero di cifre binarie con cui avete scelto di operare.

Cosa fare? Bene, nel secondo caso, la risposta è niente. Abbiamo già spiegato che quando si usano numeri con segno, il formato dei numeri è fisso. Siano essi a due byte o ad uno solo (o comunque di quanti avrete scelto), il programma deve sapere dove trovare la cifra di segno. Un numero fuori del range non può essere utilizzato a meno che, prima, non cambiate il vostro formato.

Anche se si tollera la perdita della cifra binaria più significativa, così non è per il cambiamento di segno, infatti non possiamo avere numeri positivi,mascherati come negativi, e viceversa. Il problema è risolto invertendo la cifra inesatta usando l'istruzione EOR, spiegata nel Capitolo 13.

La procedura di addizione di numeri con segno a due byte è, quindi, la seguente:

1) Azzerare il flag di carry, come in tutte le addizioni.
2) Caricare il low byte di uno dei valori nell'accumulatore.
3) Usare ADC per addizionare il low byte dell'altro valore.
4 Immagazzinare il risultato in una area.
5) Immettere il byte successivo di uno dei valori nell'accumulatore.
6) Usare ADC per addizionare il successivo byte dell'altro numero.
7) Esaminare il flag di overflow.
8) Se il flag di overflow è settato ad uno, invertire la cifra di segno del byte nell'accumulatore usando EOR (vedere Capitolo 13).

Il risultato sarà un valore il cui low byte sarà immagazzinato in una area in memoria, ed il cui high byte si trova nell'accumulatore, con una cifra di segno esatta. Se il flag di overflow era settato, il risultato è inesatto di 32768.

Esempio dell'addizione di due numeri con segno ad un solo byte.

```
  PC      SR  AC  XR  YR  SP
; 1C00    30  00  00  00  F0
> 008B    40  00  30  00
> 1C8B    00  00  00  00

A 1C00    18      CLC
A 1C01    A5  8B  ADC $8D
```

```
A  1C03    65  8D    ADC $1C09
A  1C05    50  02    BVC $1C09
A  1C07    49  80    EOR #$80
A  1C09    00        BRK

  PC       SR  AC  XR  YR  SP
; 1C0B     30  70  00  00  F0
> 008B     40  00  30  00
> 1C8B     00  00  00  00
```

Commento:
1C00-1C03: i due numeri vengono addizionati.
1C05-1C07: Queste linee impiegano due istruzioni non spiegate fino ai capitoli seguenti. Il comando BVC è piuttosto simile a GOTO in Basic, ma in questo caso è eseguito solo se il flag di overflow si trova a zero.

L'istruzione EOR inverte il bit di segno del risultato. Se il bit di overflow è settato, l'istruzione EOR inverte il bit di segno per correggerlo. Se il bit di overflow è azzerato l'istruzione BVC fa saltare il programma dopo l'istruzione EOR poiché non c'è niente da correggere.

Esempio dell'addizione di numeri con segno a due bytes.

```
  PC       SR  AC  XR  YR  SP
; 1C00     30  00  00  00  F0
> 008B     40  99  30  55
> 1C8B     00  00  00  00

A  1C00    18        CLC
A  1C01    A5  8B    LDA  $8B
A  1C03    65  8D    ADC $8D
A  1C05    AA        TAX
A  1C06    A5  8C    LDA $8C
A  1C08    65  8E    ADC $8E
A  1C0A    50  02    BVC $1C0E
A  1C0C    49  80    EOR #$80
A  1C0E    00        BRK

  PC       SR  AC  XR  YR  SP
; 1C10     B0  EE  70  00  F0
> 008B     40  99  30  55
> 1C8B     00  00  00  00
```

Commento:
1C00-1C05: Niente di nuovo qui, i low bytes dei due numeri vengono addizionati ed il risultato è trasferito al sicuro nel registro X.

1C06-1C08: Vengono addizionati i due high bytes.
1C0A-1C0C: Si esegue l'eventuale correzione del bit di segno all'high byte del risultato.

Sottrazione di numeri con segno

Come con ADC e SBC usati con i numeri senza segno, l'unica vera differenza nell'uso delle due istruzioni con numeri con segno è che c'è una limitazione riguardante quello dei due che ha i suoi byte caricati nell'accumulatore. Qualsiasi valore sia immesso nell'accumulatore sarà quello su cui si effettua la sottrazione.

La procedura per sottrarre un numero con segno da un altro è:

1) Settare il flag di carry, come in tutte le sottrazioni.
2) Caricare il low byte del valore da cui si sottrae nell'accumulatore.
3) Usare SBC per sottrarre il low byte dell'altro valore.
4) Immagazzinare il risultato in una area.
5) Immettere il byte successivo del valore da cui si sottrae nell'accumulatore.
6) Usare SBC per sottrarre il byte successivo dell'altro numero.
7) Esaminare il flag di overflow.
8) Se il flag di overflow è settato ad uno, invertire la cifra di segno del byte nell'accumulatore usando EOR (vedere Capitolo 13).

Come con ADC, il risultato sarà un valore il cui low byte sarà posto in una area in memoria, ed il cui high byte è nell'accumulatore, con cifra di segno corretta. Se il flag di overflow era settato, il risultato è inesatto per 32768. Se il "carry" è positivo o negativo dipenderà dal segno corretto del risultato.

Esempio di sottrazione di due numeri con segno ad un solo byte.

```
  PC          SR  AC  XR  YR  SP
; 1C00         30  00  00  00  F0
> 008B         40  00  30  00
> 1C8B         00  00  00  00

A  1C00        18      CLC
A  1C01        A5  8B  LDA $8B
A  1C03        65  8D  ADC  $8D
A  1C05        50  02  BVC  $1C09
A  1C07        49  80  EOR # $80
A  1C09        00      BRK

  PC          SR  AC  XR  YR  SP
; 1C0B         30  70  00  00  F0
> 008B         40  00  30  00
> 1C8B         00  00  00  00
```

Commento:
1C00-1C03: Normale sottrazione.
1C05-1C07: Come nell'addizione di prima, il comando di overflow è usato per determinare se deve essere fatta una correzione al bit di segno del risultato.

Esempio di sottrazione di numeri con segno a due byte.

```
  PC          SR  AC  XR  YR  SP
; 1C00        30  00  00  00  F0
> 008B        40  99  30  55
> 1C8B        00  00  00  00

A  1C00       18      CLC
A  1C01       A5  8B  LDA $8B
A  1C03       65  8D  ADC $8D
A  1C05       AA      TAX
A  1C06       A5  8C  LDA  $8C
A  1C0A       50  02  BVC  $1C0E
A  1C0C       49  80  EOR #$80

A  1C06       00      BRK

  PC          SR  AC  XR  YR  SP
; 1C10        B0  EE  70  00  F0
> 008B        40  99  30  55
> 1C8B        00  00  00  00
```

Commento:
La routine è molto simile in struttura all'addizione a due byte mostrata prima.

CAPITOLO 12

ARITMETICA DECIMALE

È necessario esaminare un'ultima forma di aritmetica prima di passare oltre. Si tratta della forma decimale codificato binario, o BCD (binary coded decimal), che il chip usa quando è predisposto nel "modo decimal". Il BCD fa sì che lo status register contenga un flag che non abbiamo ancora spiegato. L'obiettivo del BCD è permettere al chip di operare direttamente con numeri decimali. Questo non è sempre necessario, poichè se si devono usare numeri decimali, i calcoli possono essere fatti in binario e, quando è stato ottenuto un risultato,

fare una conversione in decimale. In alcuni casi, quando un programma richiede un input od un output in numeri decimali sullo schermo, si può risparmiare tempo e spazio in programma operando direttamente in decimale. Fortunatamente il 7501 ci permette di fare questo. L'uso del BCD è solo un fatto di convenienza in quanto il sistema di numerazione decimale non ha alcuna importanza per il chip stesso e non è mai usato per scopi correnti. Ma siccome gli esseri umani insistono a lavorare in decimale, i progettisti del 7501 hanno previsto un mezzo attraverso il quale effettuare in decimale le operazioni più semplici come addizione e sottrazione.

Dopo aver detto che è possibile operare in decimale, vi starete meravigliando proprio di come possa essere, poichè tutta la struttura del chip,con le sue locazioni di memoria ad otto bit, sembra non permetterlo. Il BCD sfrutta il chip in maniera completamente artificiale, occupando memoria e richiedendo potenza di processo. Se si accorciano alcuni programmi, tuttavia, può valere la pena di usarlo.

L'effetto del modo BCD sul chip è che, quando il flag di decimal è settato, ogni valore nell'accumulatore è considerato costituito da due numeri a quattro bit. Questa non è la normale convenzione, infatti il numero ad otto bit viene considerato formato da due metà, ciascuna delle quali trattata come una singola cifra decimale. Invece di operare sulla base che quattro bit possono registrare un numero nel range 0-15, il chip riduce il range a 0-9. Se viene creato un numero più grande di nove nei quattro bit più bassi dell'accumulatore, viene effettuato un carry ai quattro bit superiori. Se viene a crearsi, nei quattro bit superiori, un numero maggiore di nove, viene settato il flag di carry. In altre parole, se addizionate un numero al contenuto dell'accumulatore, ogni risultato ed i carry si comportano esattamente come se aggiungete due numeri decimali.

Il modo decimale non è affatto semplice. Se, per esempio, immettete il valore $FF nell'accumulatore (decimale 255), in modo che ogni metà del byte risulti 15 se vista separatamente, ed avete settato il flag di decimal per procedere con l'addizione o la sottrazione, il risultato ottenuto è senza senso. Pertanto il sistema BCD garantisce la validità dei valori che non eccedono 9 in ognuna delle due metà del byte, ed essi saranno trattati come valori decimali per semplici addizioni e sottrazioni.

È difficile darvi un quadro reale dell'uso di BCD, dato che è normalmente usato per comunicare con l'utente, essendoci pochissime ragioni per altri usi. Non è comunque tra gli scopi di questo libro analizzare il tipo di tecniche necessarie per tali comunicazioni. Ecco in ogni caso un esempio.

Supponete di voler scrivere qualcosa come un programma per calcolatore, che permetta all'utente di immettere un numero come 12, e poi immettere un altro numero da addizionare ad esso, come 89. Il metodo basilare sarebbe quello di porre prima il chip in modo decimal settando il flag di decimal, poi leggere l'input da tastiera e poi porre l'1 ed il 2 nelle due metà di un solo byte. Ora 8 e 9 sarebbero letti dalla tastiera ed immessi nelle due metà dell'accumulatore. Verrebbe poi usata una istruzione ADC per addizionare il byte contenente 12 ed il contenuto dell'accumulatore. Il risultato sarebbe un accumulatore che conterrebbe il numero binario 00000001, con il flag di carry settato. In modo decimal questo significa un "1" (il flag di carry) all'inizio del numero, zero come carattere seguente (i primi quattro zero) ed un uno per concluderlo, dando un risultato decimale di 101. La cifra decimale corrisponente sarebbe visualizzata sullo schermo.

Se avessimo continuato ad operare in binario, il numero dodici avrebbe dovuto essere tradotto in binario ed immagazzinato. Poi si sarebbe tradotto 89. Si sarebbe usata l'istruzione ADC per addizionarli, poi il risultato 01100101 avrebbe dovuto essere tradotto di nuovo in formato decimale per essere stampato sullo schermo.

Il vantaggio del modo decimale si fa sentire per usi particolari, ma, allo stesso tempo, vanno ricordati tutti i suoi limiti. In particolare, va ricordato che il modo decimal non può essere usato durante un intero programma. Nel momento in cui non avete più il bisogno di operare in decimale, il flag di decimal deve essere azzerato in modo da permettere al chip di funzionare normalmente.

Esempio dell'addizione di numeri ad un solo byte in BCD.

```
  PC        SR  AC  XR  YR  SP
; 1C00      30  00  00  00  F0
> 008B      40  00  30  00
> 1C8B      00  00  00  00

A 1C00      18      CLC
A 1C01      F8      SED
A 1C02      A5  8B  LDA $8B
```

```
A  1C04     65  8D    ADC $8D
A  1C06     00        BRK

  PC        SR  AC  XR  YR  SP
; 1C08       38  70  00  00  F0
> 008B       40  00  30  00
> 1C8B       00  00  00  00
```

Commento:
1C01: L'unica linea insolita. Una volta settato il modo decimal, siete liberi di addizionare come siete soliti fare, col chip che farà tutte le traduzioni necessarie.

Per verificare il modo decimal, provate quanto segue: immettete "9" nella locazione di memoria $8B ed "1" nella locazione di memoria $8D. Il risultato nell'accumulatore sarà 16. Questo non è un bug in quanto ecco cosa accade. Il "9" in $8B è stato immagazzinato come (0000)(1001) cioè (0)(9). L' "1" in $8D è stato immagazzinato come (0000)(0001) cioè (0)(1). Il risultato, in modo BCD è (0001)(0000) cioè (1)(0).

Il Monitor, che non opera nel modo BCD, riproduce questo come un normale numero binario con il valore decimale di 16.
Esempio di addizione di numeri a due byte in modo BCD.

```
  PC        SR  AC  XR  YR  SP
; 1C00       30  00  00  00  F0
> 008B       40  99  30  55
> 1C8B       00  00  00  00

A  1C00     18        CLC
A  1C01     F8        SED
A  1C02     A5  8B    LDA $8B
A  1C04     65  8D    ADC  $8D
A  1C06     AA        TAX
A  1C07     A5  8C    LDA  $8C
A  1C09     65  8E    ADC $8E
A  1C0B     99        BRK

  PC        SR  AC  XR  YR  SP
; 1C0D       B9  54  70  00  F0
> 008B       40  99  30  55
> 1C8B       00  00  00  00
```

Commento:

Una semplice addizione di numeri a due byte, con il low byte del risultato nel registro X e l'high byte nell'accumulatore. La sola differenza è che il flag di decimal è settato, con i risultati che potete osservare da soli.

Esempio di sottrazione di numeri ad un solo byte in BCD.

```
  PC          SR  AC  XR  YR  SP
; 1C00         30  00  00  00  F0
> 008B         40  00  30  00
> 1C8B         00  00  00  00

A  1C00       38          SEC
A  1C01       F8          SED
A  1C02       A5  8B      LDA $8B
A  1C04       E5  8D      SBC $8D
A  1C06       00          BRK

  PC          SR  AC  XR  YR  SP
; 1C08         39  10  00  00  F0
> 008B         40  00  30  00
> 1C8B         00  00  00  00
```

Commento:

Ancora una volta una semplice sottrazione. Il flag di decimal è settato prima che essa sia eseguita.

Esempio di sottrazione di numeri a due byte in BCD.

```
  PC          SR  AC  XR  YR  SP
; 1C00         30  00  00  00  F0
> 008B         46  99  27  55
> 1C8B         00  00  00  00

A  1C00       38          SEC
A  1C01       F8          SED
A  1C02       A5  8B      LDA $8B
A  1C04       E5  8D      SBC $8D
A  1C06       AA          TAX
A  1C07       A5  8C      LDA $8C
A  1C09       E5  8E      SBC $8E
A  1C0B       00          BRK

  PC          SR  AC  XR  YR  SP
; 1C0D         79  44  19  00  F0
> 008B         46  99  27  55
> 1C8B         00  00  00  00
```

Commento:

Identico al precedente eccetto che la sottrazione è eseguita con numeri a due byte.

CAPITOLO 13

OPERATORI LOGICI ED ISTRUZIONI DEI BIT

Fino ad ora abbiamo sempre dato per scontato che un byte di memoria è una unità che può essere caricata, immagazzinata, trasferita e che se un byte deve essere modificato, si mette in opera la CPU tramite istruzioni come ADC. In questo capitolo "scenderemo di un passo sulla scala" e studieremo come è possibile manipolare i singoli bit che vanno a formare un byte, ed alcuni dei motivi per cui è necessario fare ciò.

Nel corso del capitolo tratteremo le operazioni logiche, le istruzioni relative ai bit e le istruzioni shift/rotate.

Operazioni Logiche

Una operazione logica è un modo di paragonare due byte come se essi fossero semplicemente insiemi di bit, e di produrre un terzo byte sulla base di questo paragone. Ci sono tre tipi di paragone:

1) I bit nella stessa posizione in entrambi i byte, sono entrambi a "1?" Se è così, il bit equivalente nel byte risultante sarà a "1". Questa forma di operazione logica è eseguita dall'istruzione AND.
2) Uno dei due, od entrambi i due bit, sono a "1" Se è così, il bit equivalente nel byte risultante sarà pure a "1". Questa forma di operazione logica è eseguita dall'istruzione ORA.
3) L'uno o l'altro dei due bit, ma non entrambi, è a "1" Se è così, il bit equivalente nel byte risultante sarà pure a "1". Questa forma di operazione logica è eseguita dall'istruzione EOR.

Nel caso di queste tre istruzioni, i due byte su cui si effettua il confronto sono il contenuto dell'accumulatore ed un byte di memoria specificato nel programma in codice macchina. Il byte risultante dal confronto sarà posto nell'accumulatore.

L'istruzione AND

Funzione: creare un byte che abbia i bit ad "1" in ogni posizione dove i due byte originali avevano entrambi un bit settato.

Come tutte le operazioni logiche, il funzionamento di AND è meglio compreso osservando gli esempi che seguono, relativi a coppie di byte e del byte prodotto quando si applica AND ad essi:

11110000 AND 00001111 = 00000000
10101010 AND 01010101 = 00000000
11110000 AND 11111111 = 11110000
10101010 AND 00001111 = 00001010

AND è usata generalmente per resettare i bit. Per esempio, supponiamo che per qualche ragione un programma in codice macchina desideri assicurare che nessuno dei valori di una serie abbia uno dei bit 4-7 settato. Questo potrebbe essere ottenuto facilmente operando il confronto AND del valore con 15, cioè 00001111 in binario. Poichè nessuno dei bits 4-7 potrebbe essere settato in entrambi i byte, essi sarebbero tutti resettati. Nell'altra metà del byte, il fatto che ci sono bit settati in ogni posizione significa che niente sarà cambiato infatti dove c'è uno zero i due byte non si uguaglieranno e dove c'è un uno l'istruzione AND assicurerà che esso rimanga.

Un secondo uso di AND consiste nell'esaminare singoli bit in un byte specificato. Voi sapete già, per esempio, il risultato di una operazione AND su un byte con un termine che viene posto nell'accumulatore. Se il termine nell'accumulatore consiste in un valore dove solo uno degli otto bit è settato, il risultato sarà o lo stesso valore o zero, a seconda se il bit equivalente nel byte di memoria specificati è settato. Conoscete pure che il flag di zero indica se l'ultima operazione sull'accumulatore è sfociata in un valore zero. Mettete assieme tutte queste cose e vedete che usando "00001000" con AND, ed esaminando il flag di zero, si può capire se il quarto bit nella locazione di memoria specificata è settato.

La procedura per l'uso di AND è quindi molto semplice:

1) Caricare il byte di controllo, quello che usate per controllare il formato eventuale del byte, nell'accumulatore. Questo valore è conosciuto solitamente come "maschera" (nel senso di uno "stampino" che si usa in disegno), poichè è, in un certo modo, posto "sopra" l'altro byte così che solo alcuni bit possono essere visti,mentre gli altri sono coperti e resettati. La ragione per cui la maschera è normalmente posta nell'accumulatore è che essa sarà di solito specificata nel programma e può essere caricata nell'accumulatore in modo immediato.

2) Applicare la maschera ad una locazione di memoria specificata usando AND.

3) Fare qualcosa con il risultato, che si trova nell'accumulatore. Può darsi che, per esempio, vogliate sostituire il valore originale in memoria con

quello creato con AND, in tal caso dovete immagazzinare il contenuto dell'accumulatore. D'altra parte potete semplicemente voler prendere una decisione sulla base del risultato, senza effettuare alcun cambiamento al valore originale.

Modi di indirizzamento disponibili con AND:

```
Zero page:
AND $FF          : 25          : 25 FF
Immediato:
AND #$FF         : 29          : 29 FF
Assoluto:
AND $02FF        : 2D          : 2D FF 02
Pre-indicizzato, X:
AND ($FF,X)      : 21          : 21 FF
Post-indicizzato, Y:
AND ($FF),Y      : 31          : 31 FF
Zero page, X:
AND $FF,X        : 35          : 35 FF
Assoluto, X:
AND $02FF,X      : 3D          : 3D FF 02
Assoluto, Y:
AND $02FF,Y      : 39          : 39 FF 02
```

Esempio d'uso di AND:

```
  PC        SR  AC  XR  YR  SP
; 1C00       30  00  00  00  F0
> 008B       9A  99  00  00
> 1C8B       00  00  00  00

A  1C00      A5  8B     LDA #$8B
A  1C02      425 8C     AND $8C
A  1C04      00         BRK

  PC        SR  AC  XR  YR  SP
; 1C06       B0  98  00  00  F0
> 008B       9A  99  00  00
> 1C8B       00  00  00  00
```

Commento:
I due numeri su cui eseguire AND devono essere immessi nelle locazioni di memoria $8B e $8C. Il contenuto di $8B è copiato nell'accumulatore e poi confrontato tramite AND con il contenuto di $8C.

L'istruzione ORA

Funzione: creare un byte che abbia i bits settati in ogni posizione dove l'uno o l'altro, od entrambi i bytes originali hanno un bit settato.

Alcuni esempi dell'effetto di OR:

```
11110000 OR 00001111 = 11111111
10101010 OR 01010101 = 11111111
11110000 OR 11111111 = 11111111
10101010 OR 00001111 = 10101111
```

OR è generalmente usato per settare bits. Quando un byte è sottoposto ad OR con una maschera, ogni bit settato nella maschera sarà automaticamente settato nel byte risultante. Le cifre che, nel byte originale, non sono appaiate ad un bit settato nella maschera, saranno lasciate immutate siano settate o resettate.

La procedura per l'uso di ORA è la seguente:

1) Caricare il valore da usare come maschera nell'accumulatore.
2) Applicare la maschera ad un byte specificato in memoria usando OR.
3) Il risultato si troverà ora nell'accumulatore e può essere usato sia per sostituire il byte originale sia come base per qualche altra operazione del programma.

Modi di indirizzamento disponibili con ORA.

```
Zero page:
ORA $FF          : 05          : 05 FF
Immediato:
ORA #$FF         : 09          : 09 FF
Assoluto:
ORA $02FF        : 0D          : 0D FF 02
Pre-indicizzato, X:
ORA ($FF,X)      : 01          : 01 FF
Post-indicizzato, Y:
ORA ($FF,Y)      : 11          : 11 FF
Zero pag, X:
ORA $FF,X        : 15          : 15 FF
Assoluto, X:
ORA $02FF,X      : 1D          : 1D FF 02
Assoluto, Y:
ORA $02FF,Y      : 19          : 19 FF 02
```

Esempio dell'uso di ORA:

```
  PC        SR  AC  XR  YR  SP
; 1C00       30  00  00  00  F0
> 008B       9A  99  00  00
> 1C8B       00  00  00  00

A  1C00     A5  8B    LDA #$8B
A  1C02     05  8C    ORA $8C
A  1C04     00        BRK

  PC        SR  AC  XR  YR  SP
; 1C06       B0  9B  00  00  F0
> 008B       9A  99  00  00
> 1C8B       00  00  00  00
```

Commento:

L'esempio è esattamente uguale per struttura a quello dato per AND, eccetto che il contenuto delle due locazioni di memoria sono confrontate con ORA.

L'istruzione EOR

Funzione: creare un byte che abbia i bits settati in ogni posizione dove o l'uno o l'altro dei due bytes originali ha un bit settato.

È importante ricordare la distinzione tra OR ed OER (che vuol dire Exclusive OR). Siccome ORA setta un bit se c'è un bit settato in uno o nell'altro dei bytes originali a quella posizione, indifferentemente dalla combinazione, EOR setterà un bit se solo uno dei bytes originali aveva un bit settato in quella posizione. Se entrambi i bytes originali hanno un uno nella stessa posizione, il byte risultante avrà in quella posizione un bit resettato.

Qui di seguito sono dati alcuni esempi dell'uso di EOR:

```
11110000 EOR 00001111 = 11111111
10101010 EOR 01010101 = 11111111
11110000 EOR 11111111 = 00001111
10101010 EOR 00001111 = 10100101
```

EOR è generalmente usato per "colpire" il valore di uno o più bits, cambiandoli con il valore opposto senza loro stato originale. Nel capitolo sull'aritmetica con segno, abbiamo notato che se il flag di overflow era settato, indicava

che il bit di segno di un valore era inesatto ed aveva bisogno di essere cambiato con il suo opposto. Questo può essere fatto applicando la maschera 10000000 (128 in decimale) al numero. Se il bit sette è un uno, EOR lo resetterà, poiché entambi i bits nei due bytes sono settati. Se il bit sette è a zero, EOR lo setterà.

La procedura per l'uso di EOR è la seguente:

1) Caricare il valore da usare come maschera nell'accumulatore.
2) Applicare la maschera ad un byte specificato in memoria usando EOR.
3) Il risultato si trova ora nell'acccumulatore e può essere usato sia per sostituire il byte originale in memoria sia come base per qualche altra operazione del programma.

Modi di indirizzamento disponibili con EOR:

```
Zero page:
EOR $FF          : 45          : 45 FF
Immediato:
EOR #$FF         : 49          : 49 FF
Assoluto:
EOR $02FF        : 4D          : 4D FF 02
Pre-indicizzato, X:
EOR ($FF,X)      : 41          : 41 FF
Post-indicizzato, Y:
EOR ($FF,Y)      : 51          : 51 FF
Zero pag, X:
EOR $FF,X        : 55          : 55 FF
Assoluto, X:
EOR $02FF,X      : 5D          : 5D FF 02
Assoluto, Y:
EOR $02FF,Y      : 59          : 59 FF 02
```

Esempio dell'uso di EOR.

```
  PC          SR  AC  XR  YR  SP
; 1C00         30  00  00  00  F0
> 008B         9A  99  00  00
> 1C8B         00  00  00  00

A  1C00        A5  8B      LDA $8C
A  1C02        45  8C      EOR $8C
A  1C04        00          BRK
```

```
  PC      SR  AC  XR  YR  SP
; 1C06    30  03  00  00  F0
> 008B    9A  99  00  00
> 1C8B    00  00  00  00
```

Commento:

L'esempio ha la stessa struttura dei due precedenti ma applica EOR al contenuto delle due locazioni di memoria.

Le istruzioni BIT

Nella sezione relativa ad AND abbiamo notato come quella istruzione possa essere usata per esaminare il contenuto di un singolo bit in una specifica locazione. C'è, tuttavia, un'altra istruzione che può essere usata per questi scopi, sebbene le sue applicazioni siano molto più limitate.

BIT è in effetti una forma AND con una piccola aggiunta, e la ragione per cui non è usata più intensamente è che può essere usata solo con due modi di indirizzamento.

La funzione di BIT è fornire:

1) Un esame molto chiaro del contenuto dei bit 6 e 7 di una locazione di memoria specificata.
2) Un metodo un pò meno chiaro per esaminare lo stato di un singolo bit.

Esame dei bits 6 e 7 con BIT: Qualunque sia il valore nell'accumulatore, se applicate ad una locazione di memoria BIT, il contenuto dei bit 6 e 7 della locazione di memoria saranno trasferiti nei bit 6 e 7 dello status register, che sono i flag di sign e di overflow. Il contenuto dell'accumulatore resta inalterato, poichè, al contrario di AND, BIT non immette il risultato nell'accumulatore, ma viene scartato. Questa è una tecnica utile per determinare il segno di un numero,dato che applicando semplicemente un'istruzione BIT all'indirizzo al quale un numero è inserito, essa setterà il flag di sign con il segno del numero, senza interessare il contenuto originale dell'accumulatore.

Esaminare un bit con BIT: Mettendo da parte la sua funzione automatica, BIT può essere usato per esaminare un bit in uno specificato byte in memoria immettendo la maschera appropriata nell'accumulatore. Per esempio inserendo 00001000 nell'accumulatore ed applicandolo ad uno specificato byte in memoria si esamina la condizione del bit 3. Il flag di zero registra se il bit corrispondente nel byte di memoria specificato era settato, e pur senza modificare l'accumulatore. A prima vista BIT appare come un comando molto utile, poichè la maschera situata nell'accumulatore non viene alterata quando la si applica, come succede con AND. Sfortunatamente, la prospettiva di una applicazione di una maschera inalterante ad una serie di byte (che avrebbe molte applicazioni), è rovinata dal fatto che BIT può essere usato

solo in due modi di indirizzamento. Per questa ragione, BIT è raramente usato per qualcosa di diverso dall'esame del bit 7, dove i suoi vantaggi rispetto ad AND sono ovvi.

Procedura per usare BIT:

1) A meno che si stia semplicemente esaminando i bit 6 e 7, caricare la maschera desiderata nell'accumulatore.
2) Applicarla al byte di memoria specificato usando BIT.
3) Esaminare i flag di sign e di overflow per il contenuto del sesto e del settimo bit del byte di memoria specificato.
4) Esaminare il flag di zero per vedere se il byte specificato in memoria ha qualche bit settato nella stessa posizione della maschera.

Modi di indirizzamento disponibili con BIT:

```
Zero page
: BIT $FF        : 24          : 24 FF
Assoluto
: BIT $02FF      : 2C          : 2C FF 02
```

Esempio dell'uso di BIT.

```
  PC          SR  AC  XR  YR  SP
; 1C00         30  00  00  00  F0
> 008B         9A  00  00  00
> 1C8B         00  00  00  00

A  1C00        A9  10   LDA #$10
A  1C02        24  8B   BIT  $8B
A  1C04        00       BRK

  PC          SR  AC  XR  YR  SP
; 1C06         B0  10  00  00  F0
> 008B         9A  00  00  00
> 1C8B         00  00  00  00
```

Commento:

La routine dimostra l'effetto del confronto di $10 (nell'accumulatore) e del contenuto di $8B usando BIT. Notate che il valore $10 caricato nell'accumulatore è ancora lì alla fine poichè il risultato di BIT è scartato ed ininfluente. Il flag di zero, tuttavia, mostra correttamente che è stato creato uno zero se non ci sono bit appaiati. Calcolate qualsiasi numero volete in forma binaria ed immettetelo in $8B per vedere l'effetto.

Rotating e Shifting

Dopo aver visto alcuni dei modi con cui si possono modificare le singole cifre in un byte, passiamo ad un'altra serie di istruzioni che ci permettono di muoverle dentro il byte. Tali istruzioni sono conosciute come "shift" e "rotate", per ragioni che diverranno ben presto ovvie, poichè il loro effetto è muovere i bit dentro il byte da un posto all'altro, in certi casi trasferendo un bit che "cade fuori" da una parte del byte.

Le istruzioni Shift

La forma più semplice di questo gruppo di istruzioni è lo shift. Le due istruzioni shift sono:
ASL - Arithmetic Shift Left
LSR - Logical Shift Right

L'istruzione Arithmetic Shift Left

Funzione: Sposta i bit in una locazione specificata di un posto verso sinistra. Il contenuto del bit 7 sarà posto nel flag di carry.

Il mezzo più semplice per spiegare l'istruzione ASL è un esempio. Prendete il byte seguente, contenuto nell'accumulatore:

0 1 0 1 0 1 0 1
FLAG DI CARRY: ?

Il ? nel flag di carry indica che il contenuto del flag di carry prima che l'istruzione ASL sia eseguita è irrilevante. Applicando ASL, il risultato è il seguente:

1 0 1 0 1 0 1 0
FLAG DI CARRY: 1

Notate che, se voi state operando con numeri con segno, il bit di segno si trova ora nel flag di carry, così che possiate regolarvi di conseguenza.

La procedura per applicare ASL è la seguente:

1) Identificare il byte su cui si deve effettuare lo shift. Questo può essere in memoria (ci sono quattro modi di indirizzamento disponibili) o nello

stesso accumulatore. Le istruzioni shift/rotate sono le sole che, sul chip 7501, abbiano un modo di indirizzamento (indirizzamento dell'accumulatore) che permette loro di essere applicate direttamente al contenuto dell'accumulatore.

2) Applicare ASL al byte usando il modo di indirizzamento appropriato.

3) Se necessario, esaminare i flag significativi. Il flag di zero indicherà se l'ultimo bit settato è stato spostato fuori del byte. Il flag di carry indicherà il contenuto del bit di segno del byte prima dell'inizio del procedimento.

I principali usi di ASL sono:

1) Moltiplicazione per una potenza di due. Un semplice spostamento a sinistra, in un byte, effettivamente moltiplica il valore contenuto dentro di esso per 2. ASL, di per sè, è di uso limitato sotto questo aspetto, poichè i bit che fuoriescono alla sinistra del byte sono persi se essa è applicata parecchie volte in successione e cioè quando il risultato oltrepassa 255. In questi casi, ASL è usata congiunta con l'istruzione "rotate left" che vedremo tra un momento.
2) Usata in un loop, ASL è uno strumento effettivo per esaminare il contenuto di ogni bit di un byte. Otto spostamenti trasferiranno ogni bit a turno nel flag di carry, che può essere facilmente esaminato, senza l'uso di una maschera.

I modi di indirizzamento disponibili con ASL sono:

```
Accumulatore
: ASL A          : 0A          : 0A
Zero page
: ASL $FF        : 06          : 06 FF
Assoluto
: ASL $02FF      : 0E          : 0E FF 02
Zero page, X
: ASL $FF,X      : 16          : 16 FF
Assoluto, X
: ASL $02FF,X    : 1E          : 1E FF 02
```

Esempio dell'uso di ASL.

```
  PC        SR  AC  XR  YR  SP
; 1C00       30  00  00  00  F0
> 1C8B       9A  00  00  00
> 1C8B       00  00  00  00
```

```
A  1C00     06  8B    ASL  $8B
A  1C02     00        BRK

  PC        SR  AC  XR  YR  SP
; 1C04       31  00  00  00  F0
> 008B       34  00  00  00
> 1C8B       00  00  00  00
```

Commento:

Una sola istruzione è tutto quanto richiesto per usare il Monitor al fine di esplorare l'effetto di ASL sui valori a vostra scelta immessi nella locazione di memoria $8B.

L'istruzione Logical Shift Right

Parallela all'istruzione ASL c'è la LSR, sebbene non sia una immagine speculare della precedente. Arithmetic Shift Left è così chiamata perchè conserva il bit di segno (nel flag di carry) e vi permette di eseguire operazioni aritmetiche in modo corretto, cosa impossibile se il bit di segno fosse perso. Molti processori hanno sistemi di spostamento aritmetici sia a sinistra che a destra che permettono di ricuperare il bit di segno. Inoltre, essi ne hanno degli altri logici, che semplicemente spostano i bit senza preoccuparsi del flag di segno. Questo importa realmente solo con gli spostamenti a sinistra. Gli spostamenti a sinistra quasi sempre immettono il bit più significativo nel flag di carry, così che ogni spostamento a sinistra può essere chiamato aritmetico.

Se state effettuando uno spostamento dei bit verso destra, è necessario prendere alcune precauzioni per preservare il bit di segno, altrimenti il bit di segno si sposterà nel bit 6 e sarà rilevabile solo con qualcosa come AND o BIT. L'istruzione del 7501 relativa allo spostamento verso destra non prevede niente per conservare il bit di segno in qualche luogo accessibile, così è che non lo si può chiamare spostamento aritmetico a destra ma spostamento logico a destra.

Sotto tutti gli altri aspetti, LSR è equivalente a ASL. Il bit che cade fuori del byte è ricevuto dal flag di carry, ed il flag di zero indica se lo spostamento ha portato fuori del byte tutti i bit settati. Qui di seguito viene dato un esempio dell'uso di LSR su un byte particolare:

```
1 0 1 0 1 0 1 0
FLAG DI CARRY: ?
diventa
0 1 0 1 0 1 0 1
FLAG DI CARRY: 0
```

La procedura per l'uso di LSR è la seguente:

1) Scegliere il byte su cui operare, che sia, come con ASL, nell'accumulatore.
2) Applicare LSR usando l'appropriato modo di indirizzamento.
3) In accordo con le vostre necessità, esaminare il flag di carry per il contenuto precedente del bit zero oppure il flag di zero per controllare se il risultato di LSR è zero. Il segno del numero esistente prima che LSR fosse applicata può essere controllato solamente sondando il bit 6 del risultato.

LSR è generalmente usata per:

1) Semplici divisioni per due di un singolo byte. In combinazione con altre istruzioni è parte integrante di forme più complesse di divisione. In combinazione con le istruzioni "rotate", può essere usata per eseguire divisioni per due su numeri che sono formati da più di un byte.
2) Può essere fatto un esame del contenuto del bit zero spostandolo nel flag di carry, eliminando così la necessità di una maschera.
3) Come con ASL, può essere usata per esaminare ogni bit di un byte immettendoli a turno nel flag di carry, di nuovo senza l'uso di una maschera.

I modi di indirizzamento disponibili con LSR sono:

```
Accumulatore:
LSR A          : 4A          : 4A
Zero page:
LSR $FF        : 46          : 46 FF
Assoluto:
LSR $02FF      : 4E          : 4E FF 02
Zero page, X:
LSR $FF,X      : 56          : 56 FF
Assoluto,X:
LSR $02FF,X    : 5E          : 5E FF 02
```

Esempio dell'uso di LSR.

```
  PC          SR  AC  XR  YR  SP
; 1C00         30  00  00  00  F0
> 008B         34  00  00  00
> 1C8B         00  00  00  00

A  1C00        46  8B   LSR $8B
A  1C02        00       BRK
```

```
  PC       SR  AC  XR  YR  SP
; 1C04      30  00  00  00  F0
> 008B      1A  00  00  00
> 1C8B      00  00  00  00
```

Commento:

Ancora una volta una sola istruzione vi permette di esaminare gli effetti di LSR sui valori immessi nella locazione di memoria $8B. Dovete porre particolare attenzione all'effetto dell'istruzione sui numeri che sono dotati di un segno positivo o negativo.

Le istruzioni "rotate"

Se avete compreso perchè le istruzioni shift sono chiamate così, cioè perchè esse spostano i singoli bit di un posto verso sinistra o verso destra, non ci dovrebbero essere difficoltà nel capire le due istruzioni rotate. ROR (rotate right) e ROL (rotate left). Poichè ROR e ROL sono letteralmente l'una l'immagine speculare dell'altra, esse saranno descritte assieme e, ancora una volta, il modo migliore per comprendere questi comandi particolari è quello di vederli in azione in un esempio:

```
1  0  1  0  1  0  1  0
FLAG DI CARRY: 1
applicando ROR noi abbiamo:
1  1  0  1  0  1  0  1
FLAG DI CARRY: 0
mentre con ROL:
0  1  0  1  0  1  0  1
FLAG DI CARRY: 1
```

Si usa il termine rotate perchè i bit che cadono fuori da una parte del byte sono posti nel flag di carry ed il contenuto del flag di carry è fornito al byte dalla parte opposta (il contenuto del byte scorre attraverso un circolo composto dagli otto bit del byte e da un bit del flag di carry) e nulla si perde. Se una istruzione rotate è eseguita nove volte, il contenuto del flag di carry ed il contenuto del byte saranno allo stato in cui erano all'inizio il processo.

Le istruzioni rotate settano i flag nello stesso modo delle istruzioni shift. Se c'è un solo bit settato, ruotandolo fino al flag di carry, sarà settato il flag di zero. Ruotando il byte in modo tale che il bit di segno diventi settato, si setterà pure il flag di sign.

La procedura per usare ROR e ROL è la seguente:

1) Selezionare il byte su cui operare, che può essere un byte in memoria od il contenuto dell'accumulatore.
2) Assicurare che il flag di carry sia al valore desiderato, poichè entrerà a far parte del byte.
3) Applicare ROR e ROL al byte usando il modo di indirizzamento appropriato.
4) Esaminare ogni flag come richiesto dal programma.

Rotate è usata molto raramente da sola, di solito è accompagnata dalle istruzioni shift:

1) Per eseguire una semplice moltiplicazione per 2 su un numero di più di un byte. La procedura è la seguente: a) azzerare il flag di carry. b) spostare il low byte del numero a sinistra con ASL. c) ruotare il byte successivo del numero a sinistra usando ROL. Questo aggiunge il contenuto del flag di carry alla parte finale destra del byte, assicurando che ogni overflow dal low byte sia riportato al secondo. d) ripetere la fase c) per ogni byte seguente se il numero è composto da più di due byte.
2) Per eseguire una semplice divisione per due su un numero inserito in più di un byte. La procedura è la seguente: a) azzerare il flag di carry. b) spostare verso destra l'high byte con LSR. c) ruotare il byte successivo del numero verso destra usando ROR. Questo aggiunge il contenuto del flag di carry alla parte finale sinistra del byte, assicurando che ogni rimanenza dall'high byte sia riportata al successivo byte inferiore. d) ripetere la fase c) per ogni byte seguente se il numero è composto da più di due byte.
3) Sia ROR che ROL possono essere usati per invertire l'ordine dei bit in un byte. Un test classico per gli studenti che hanno appena appreso le istruzioni shift e rotate è quello di chiedere loro di escogitare un metodo per invertire l'ordine dei bit in un byte usando una serie di istruzioni shift e rotate e due byte di memoria, uno dei quali contenente il byte da invertire. Se volete cercatelo da soli, viceversa la risposta è qui di seguito:

Spostate il byte originario in una direzione e per ogni spostamento, ruotate il byte ottenuto nella direzione opposta. Fatelo per otto volte. Ogni spostamento prenderà un bit da una parte del byte e lo immetterà nel flag di carry. La rotazione fornirà il contenuto del flag di carry nella parte opposta del byte.

I modi di indirizzamento disponibili con ROL sono:

Accumulatore:		
ROL A	: 2A	: 2A
Zero page:		
ROL $FF	: 26	: 26 FF

Assoluto:
ROL $02FF : 2E : 2E FF 02
Zero page, X:
ROL $FF,X : 36 : 36 FF
Assoluto, X:
ROL $02FF,X : 3E : 3E FF 02

Esempio dell'uso di ROL nella rotazione di un valore ad un solo byte.

```
  PC        SR  AC  XR  YR  SP
; 1C00       30  00  00  00  F0
> 008B       1A  00  00  00
> 1C8B       00  00  00  00

A  1C00      38      SEC
A  1C01      26  8B  ROL $8B
A  1C03      00      BRK

  PC        SR  AC  XR  YR  SP
; 1C05       30  00  00  00  F0
> 008B       35  00  00  00
> 1C8B       00  00  00  00
```

Esempio dell'uso di ROL nello spostamento di un valore a due bytes.

```
  PC        SR  AC  XR  YR  SP
; 1C00       30  00  00  00  F0
> 008B       AA  AA  00  00
> 1C8B       00  00  00  00

A  1C00      06  8B  ASL $8B
A  1C02      26  8C  ROL $8C
A  1C04      00      BRK

  PC        SR  AC  XR  YR  SP
; 1C06       31  00  00  00  F0
> 008B       54  55  00  00
> 1C8B       00  00  00  00
```

Commento:
Queste routines seguono il metodo indicato sopra per la rotazione verso sinistra di un valore a due bytes.
1C00: Il low byte del numero è fatto scorrere verso sinistra ed il suo bit alto viene posto nel flag di carry.

1C02: Ruotando l'high byte ci si assicura che il contenuto del flag di carry venga considerato il bit meno significativo del risultato. In pratica, se ponete dei valori a due byte in $8B-$8C troverete cche la routine li moltiplica per due sempre che il valore a due byte originario non superi 32767 (cioè 127 nella locazione $8C e 255 nella locazione $8B: 32767 = 255+256*127).

I modi di indirizzamento disponibili con ROR sono:

```
Accumulatore:
ROR A          : 6A          : 6A
Zero page:
ROR $FF        : 66          : 66 FF
Assoluto:
ROR $02FF      : 6E          : 6E FF 02
Zero page, X:
ROR $FF,X      : 76          : 76 FF
Assoluto, X:
ROR $02FF,X    : 7E          : 7E FF 02
```

Esempio dell'uso di ROR per far scorrere valori a due bytes.

```
  PC          SR  AC  XR  YR  SP
; 1C00         30  00  00  00  F0
> 008B         54  55  00  00
> 1C8B         00  00  00  00

A  1C00        46  8C  LSR #$8C
A  1C02        66  8B  ROR $8B
A  1C04        00      BRK

  PC          SR  AC  XR  YR  SP
; 1C06         B0  00  00  00  F0
> 008B         AA  2A  00  00
> 1C8B         00  00  00  00
```

Commento:
L'opposto della procedura nell'ultima routine di esempio.

Esempio dell'uso di ROR su numeri con segno ad un solo byte.

```
  PC          SR  AC  XR  YR  SP
; 1C00         30  00  00  00  F0
> 008B         AA  00  00  00
> 1C8B         00  00  00  00
```

```
A  1C00    38       SEC
A  1C01    24  8B   BIT $8B
A  1C03    30  01   BMI $1C06
A  1C05    18       CLC
A  1C06    66  8B   ROR $8B
A  1C08    00       BRK

  PC       SR  AC  XR  YR  SP
; 1C0A      B0  00  00  00  F0
> 008B      D5  00  00  00
> 1C8B      00  00  00  00
```

Commento:

Il problema introdotto da questa piccola routine è già stato menzionato, ed è quello di conservare il bit di segno di un valore quando la parte principale del valore sta ruotando. Il modo con cui fare questo è usare un'altra istruzione "branch", equivalente al GOTO del BASIC, ma questa volta il salto avviene se il numero saminato è negativo.

1C00-1C01: Il flag di carry è dapprima settato, poi viene applicato BIT alla locazione di memoria scelta. Questo, come abbiamo visto, trasferisce il contenuto dei bits sei e sette ai flags di overflow e di sign.

1C03-1C05: Il flag di carry è azzerato di nuovo se l'istruzione BIT non ha settato il flag di sign, indicando che il valore nella locazione di memoria $8B è negativo. Il risultato del processo, se il valore in $8B è negativo, porta al settaggio del flag di carry, mentre se il valore è positivo il flag di carry verrà azzerato.

1C06: Il valore nella locazione di memoria $8B è ora ruotato verso destra. Il contenuto del flag di carry, (il bit sette originale) shifta all'indietro nella sua posizione corretta ed assicura che il segno del numero non cambi.

Qualcuno si chiederà perché non abbiamo menzionato il fatto che il bit di segno originale, che non era parte del valore, ma semplicemente un indicatore, è stato ruotato nella parte principale del numero. Niente paura, a ciò provvede l'aritmetica del complemento a due, che usiamo per i numeri con segno.

Supponiamo di prendere il numero negativo 10000001, cioè -127 in decimale e di applicargli rotate in accordo con il metodo usato prima. Il risultato è 11000000, con il bit di segno originario che si è spostato al bit 6. Ma quando traduciamo qusto numero in decimale troviamo che effettivamente è 64. Le regole dell'aritmetica di complemento a due assicurano un risultato sensato entro i limiti di un numero ad otto byte.

Esempio dell'uso di ROR su numeri con segno a Doppio Byte.

```
  PC          SR  AC  XR  YR  SP
; 1C00        30  00  00  00  F0
> 008B        AA  AA  00  00
> 1C8B        00  00  00  00

A  1C00       38      SEC
A  1C01       24  8C  BIT $8C
A  1C03       30  01  BMI $1C06
A  1C05       18      CLC
A  1C06       66  8C  ROR $8C
A  1C08       66  8B  ROR $8B
A  1C0A       00      BRK

  PC          SR  AC  XR  YR  SP
; 1C0C        30  00  00  00  F0
> 008B        55  D5  00  00
> 1C8B        00  00  00  00
```

Commento:

Questa seconda routine esegue uno spostamento verso destra su un numero con segno a due bytes. Il bit di segno è conservato grazie al sistema descritto per la routine precedente.

CAPITOLO 14

PARAGONI

Ora che siamo in grado di addizionare e sottrarre, possiamo proseguire col vedere una serie di istruzioni essenziali per le operazioni di un normale programma in codice macchina. Queste sono conosciute come istruzioni "compare" (paragone) e la loro funzione è esaminare se due valori scelti sono uguali o se uno è più grande dell'altro. Le tre istruzioni compare sono tra le più usate nella programmazione, proprio come gli operatori "<" e ">" sono largamente usati in BASIC come basi per prendere delle decisioni. Inoltre, poichè in codice macchina non esiste l'istruzione FOR...NEXT che provvede a ripetizioni automatiche di una serie di istruzioni dentro un loop, le istruzioni "compare" sono usate generalmente per creare loop che funzionano come una semplice routine BASIC del tipo:

```
10 I=0
20 PRINT "QUESTA È LA RIPETIZIONE NUMERO ";I
30 I=I+1
40 IF I<10 THEN GOTO 20
```

Le istruzioni "compare" possono essere confuse per qualcuno che ha appena iniziato a studiare il codice macchina poichè esse operano con gli stessi principi di SBC, sottraendo un valore dal contenuto dell'accumulatore e settando i flag appropriatamente, sebbene il risultato della sottrazione è scartato ed il valore nell'accumulatore non è alterato. Nella lettura dei risultati di un paragone il flag più importante è il flag di carry, e voi ricorderete dal precedente capitolo che in sottrazione, se un valore è sottratto da un valore più alto posto nell'accumulatore, il flag di carry è settato, poichè la sottrazione opera sul tipo di logica usata dall'aritmetica del complemento a due, dove i ruoli di "1" e "0" sono invertiti.

L'istruzione CMP

Funzione: Sottrae il contenuto di uno specificato byte di memoria dal contenuto dell'accumulatore e setta i flag in accordo con il risultato. Il risultato in se stesso non è trattenuto e nessuno dei valori usati nel paragone è alterato in qualche modo.

La procedura per l'uso di CMP è la seguente:

1) Assicurarsi che uno dei due valori che devono essere paragonati sia nell'accumulatore.
2) Scegliere la locazione in memoria del byte che contiene il valore o specificare quel valore nel programma stesso.
3) Applicare CMP ai due valori usando un modo di indirizzamento appropriato, come specificato nella tabella alla fine di questa sezione.
4) Esaminare i flag per accertare il risultato del paragone:
   a) Flag di zero: se è settato, i due valori sono uguali, poichè il risultato della sottrazione è zero.
   b) Flag di carry: se il flag di carry è settato, il valore nell'accumulatore è più grande od uguale al valoreconcui è stato paragonato (di quale delle due possibilità si tratti può essere determinato esaminando il flag di zero). Se il flag di carry è a zero, il valore nell'accumulatore è minore del valore con cui è stato paragonato.

In molti casi, i paragoni devono essere eseguiti su valori a due byte. Questa è una procedura più complessa che richiede l'uso di istruzioni che non abbiamo ancora esaminato. La procedura è la seguente:

1) Porre l'high byte del primo numero che deve essere paragonato nell'accumulatore.
2) Usare CMP per paragonarlo con l'high byte del secondo numero.
3) Esaminare il flag di zero. Se è resettato, il flag di carry indicherà quale dei due numeri è ilpiùgrande, senza dover fare ulteriori test. Se il flag di zero è settato, si dovranno svolgere le seguenti fasi:
4) Caricare il low byte del primo valore da paragonare nell'accumulatore.
5) Usare CMP per paragonarlo conillow byte del secondo numero.
6) I flag di carry e di zero daranno ora il risultato del paragone dell'intero numero a due byte.

Nota: Se i numeri sono compostidapiù di due byte, omettete il passo sei e ripetete fino a quando sia necessaria la fase 3.

Modi di indirizzamento disponibili con CMP:

```
Zero page
: CMP $FF        : C5          : C5 FF
Assoluto
: CMP $02FF      : CD          : CD FF 02
Immediato
: CMP #$FF       : C9          : C9 FF
Pre-indicizzato,X
: CMP ($FF,X)    : C1          : C1 FF
Post-indicizzato,Y
```

```
: CMP ($FF),Y   : D1          : D1 FF
Zero page, X
: CMP $FF,X     : D5          : D5 FF
Assoluto, X
: CMP $02FF, X  : DD          : DD FF 02
Assoluto,Y
: CMP $02FF,Y   : D9          : D9 FF 02
```

Esempio dell'uso di CMP.

```
  PC          SR  AC  XR  YR  SP
; 1C00         30  00  00  00  F0
> 008B         9A  00  00  00
> 1C8B         00  00  00  00

A  1C00        A9  10  LDA #$10
A  1C02        A9  55  LDA  #$55
A  1C02        C9  66  CMP  #$66
A  1C04        00      BRK

  PC          SR  AC  XR  YR  SP
; 1C06         B0  55  00  00  F0
> 008B         9A  00  00  00
> 1C8B         00  00  00  00
```

Commento:
La routine usa il modo di indirizzamento immediato per paragonare i valori $55 e $66 ed è così più limitata nei suoi scopi di quelle date in precedenza. Un utile esercizio consta nel modificarla in modo che carichi il contenuto, per dire, della locazione di memoria $8B nell'accumulatore e poi paragoni questo al contenuto della locazione di memoria $8C.

Le istruzioni CPX e CPY

Le istruzioni Compare X e Compare Y sono parallele all'istruzione CMP nella funzione, differiscono solo nel fatto che, invece di sottrarre il contenuto di un byte di memoria specificato dal contenuto dell'accumulatore, questo viene sottratto dal registro X o da quello Y, a seconda dell'istruzione usata.

Le istruzioni CPX e CPY sono più limitate di CMP in quanto solotremodi di indirizzamento sono disponibili per loro, come mostrato dalla tabella data di seguito.

La procedura per l'uso di CPX e CPY è esattamente la stessa di quella per CMP, eccetto che per le limitazioni imposte dai pochi modi di indirizzamento

e per il fatto che sia il registro X che quello Y prendono il posto dei riferimenti all'accumulatore.

CPX e CPY sono quasi sempre usati per esaminare le variabili di un loop dentro i programmi poichè i loop spesso formano la base di una serie di operazioni, su byte di memoria, che usano l'indirizzamento indicizzato. Usare uno dei registri indice sia per definire il byte di memoria su cui si deve operare e sia per determinare la posizione dentro un loop, è un metodo di programmazione assai economico.

Modi di indirizzamento disponibili con CPX:

```
Immediato
: CPX #$FF      : E0          : E0 FF
Zero page
: CPX $FF       : E4          : E4 FF
Assoluto
: CPX $02FF     : EC          : EC FF 02
```

Modi di indirizzamento disponibili con CPY

```
Immediato
: CPY #$FF      : C0          : C0 FF
Zero page
: CPY $FF       : C4          : C4 FF
Assoluto
: CPY $02FF     : CC          : CC FF 02
```

Esempio dell'uso di CPX.

```
  PC          SR  AC  XR  YR  SP
; 1C00         30  00  00  00  F0
> 008B         9A  00  00  00
> 1C8B         00  00  00  00

A  1C00        A2  55      LDX #$55
A  1C02        E0  55      CPX  $55
A  1C04        00          BRK

  PC          SR  AC  XR  YR  SP
; 1C06         33  00  55  00  F0
> 008B         00  00  00  00
> 1C8B         00  00  00  00
```

Esempio dell'uso di CPY:

```
  PC          SR  AC  XR  YR  SP
; 1C00         30  00  00  00  F0
```

```
> 008B      00   00   00   00
> 1C8B      00   00   00   00

A  1C00     A0   55    LDY #$55
A  1C02     C0   44    CPY  $44
A  1C04     00         BRK

  PC        SR   AC   XR   YR   SP
; 1C06      31   00   00   55   F0
> 008B      00   00   00   00
> 1C8B      00   00   00   00
```

CAPITOLO 15

CONTROLLO DI PROGRAMMA

Se immaginate cosa sarebbe la programmazione in BASIC senza le istruzioni GOTO, GOSUB, IF...THEN o FOR...NEXT (in altre parole nessun modo di eseguire ogni parte del programma all'infuori di quello nello stretto ordine nel quale il programma è scritto), capirete che questo capitolo è molto importante nella comprensione della programmazione in codice macchina. In questo capitolo vedremo il modo in cui il programma in codice macchina può essere fatto per eseguire tutti i diversi tipi di decisione e quali parti del programma devono essere usate in determinate circostanze. Possono essere scritti programmi che non richiedono decisioni od alcun cambiamento nell'ordine con cui i compiti sono eseguiti, ma tali programmi sono raramente utili. Una delle caratteristiche essenziali di una proficua programmazione è quella per cui il programma osservi quello che sta accadendo durante la sua esecuzione e poi prenda decisioni su come procedere. Tali decisioni richiedono la scelta di determinati compiti piuttosto che di altri, e ciò può essere fatto solo saltando da una parte del programma all'altra.

Potete usare tre metodi per muovervi attraverso un programma in codice macchina: salto (jump) assoluto, diramazione (branch) relativa e chiamata (call) alle subroutine (inclusi i ritorni dalle stesse).

Il metodo più comune di eseguire un programma in codice macchina è quello di scegliere un punto di partenza nella memoria dal quale iniziare esaminando ed eseguendo le istruzioni che formano il programma. Consideriamo i due byte seguenti, che supponiamo che si trovino alle locazioni $1C00 e $1C01:

24 18

Questa è la forma in codice macchina di BIT $18, una istruzione che abbiamo visto nel Capitolo 13, e sarebbe portata a termine se dovessimo eseguire il programma partendo da $1C00. Se iniziamo l'esecuzione del programma non da $1C00 ma da $1C01, l'istruzione sarebbe letta come "18" semplicemente, che è il codice operativo di una istruzione totalmente differente, "clear carry".

Il dove si salti in un programma in codice macchina può fare una differenza enorme su cosa accade quando iniziate ad eseguirlo. Se non iniziate nel punto corretto, in molte circostanze danneggiate il vostro sistema (come minimo il programma non realizzerà quello per cui era ideato). Questo è un punto importante quando si giunge al controllo di programma, e non solo

quando lo iniziamo. Più avanti vedremo che un problema più grande può essere quello di assicurare che, quando l'esecuzione del programma salta da un posto all'altro, essa salti nel luogo giusto, piuttosto che all'interno di una istruzione.

Dopo questa digressione, ritorniamo ad esaminare come viene eseguito un programma in codice macchina. È stato detto che per iniziare l'esecuzione del programma ad un certo punto, la CPU esamina il byte in quel punto e lo traduce in una istruzione. Qualche volta, l'istruzione richiede qualche ulteriore informazione; ad esempio una istruzione con incorporato il modo di indirizzamento immediato richiederebbe alla CPU di prendere un altro byte dal programma come "operatore" per l'istruzione. Tralasciando il fatto che deve eseguire l'istruzione, la CPU ora deve anche muoversi oltre l'istruzione che ha appena letto (incluso il suo operatore) ed eseguito, portandosi all'inizio della istruzione successiva. Questo è assicurato da un registro speciale conosciuto sotto il nome di program counter" (contatore di programma), usato appunto per prendere nota di dove la CPU è arrivata nell'esecuzione di un programma in codice macchina inserito in memoria. La CPU possiede la capacità di conoscere quanto sia lunga ogni istruzione, inclusi i suoi operatori, e così non ha difficoltà alcuna nello spostarsi all'inizio della istruzione successiva. L'esecuzione del programma procede in questo modo con fasi irregolari che comprendono uno, due o tre byte, a seconda di come la CPU interpreta le istruzioni e si porta avanti.

Tutto questo prosegue fino a che il programmatore non includa una istruzione nel programma che indichi un ordine differente circa il modo in cui deve avvenire il processo. Tale istruzione consisterà nel dire alla CPU di non eseguire l'istruzione successiva ma di saltare ad un punto preciso in memoria ed iniziare là l'esecuzione del programma. Il punto in memoria può essere espresso nella forma di un indirizzo a due byte, nel qual caso il programma fà quello che è conosciuto come un salto assoluto . Alternativamente, il punto al quale si può effettuare il salto può essere specificato come "diramazione relativa", vale a dire un numerodibyteavantiod indietro rispetto alla posizione attuale del program counter. Le istruzioni relative "branch" si presentano in una varietà di forme parallele al formato IF...THEN...GOTO in BASIC, permettendo di esaminare qualcosa (un flag) e di fare un salto (o meno) sulla base del risultato. Finalmente il comando BASIC GOSUB ha il suo equivalente in codice macchina sotto forma di una istruzione per saltare ad un'altra parte del programma, ricordando però dove l'esecuzione del programma era arrivata quando è stato fatto il salto. Una volta eseguito il salto, una istruzione di ritorno simile a RETURN in BASIC riporterà l'esecuzione al punto dal quale era stato eseguito il salto.

Salti assoluti con l'uso di JMP

Funzione: Permettere al programmatore di specificare una posizione in

memoria alla quale il programma salterà quando sarà incontrata l'istruzione JMP.

Ci sono ragioni buone e cattive per usare le istruzioni JMP, proprio come esistono ragioni buone e cattive per usare GOTO nel BASIC. Le cattive ragioni solitamente si rifanno al fatto che il programma è scritto alla rinfusa e le istruzioni JMP (o GOTO) sono necessarie per eseguire le varie parti nell'ordine giusto.

Le buone ragioni non sono così frequenti come la maggior parte dei programmatori sembra asserire, e molte istruzioni JMP sono usate semplicemente per correggere cattive programmazioni. I pregi di JMP sono:

1) La possibilità di saltare ad una routine nella ROM del computer.Ci sono, a volte, buone ragioni per non usare JSR, l'equivalente del GO-SUB BASIC, per fare questo ma esse saranno esaminate nella sezione relativa a tale istruzione.
2) La creazione di loop dove non venga presa la decisione riguardante la continuazione del loop. Un esempio in BASIC potrebbe essere:

```
10 INPUT A$
20 IF A$= "STOP" THEN STOP
30 PRINT A$
40 GOTO 10
```

Qui il GOTO nella linea 40 è un comando incondizionato, senza alcun IF relativo ad esso, poichè la decisione è presa prima nel loop.

3) Concludendo un programma in cui varie sezioni differenti (le quali non tutte devono essere usate) vanno scritte in sequenza. Osservate il semplice programma BASIC che segue:

```
10 INPUT "DAMMI UN NUMERO TRA 1 E 10" ;NUMBER
20 IF NUMBER > =1 AND NUMBER < =10 THEN50
30 PRINT "HO DETTO TRA 1 E 10.!!"
40 GOTO 60
50 PRINT "IL TUO NUMERO ERA MINORE DI 10 DI";10-
NUMBER
60 INPUT "ANCORA (Y/N):";Q$
70 IF Q$= "Y" THEN GOTO 10
```

In questo caso c'è un salto incondizionato eseguito verso la linea 60 ed è una parte di programmazione perfettamente legittima, poichè avendo raggiunto la linea sessanta non c'è dubbio che si intenda saltare la linea 50. La decisione è stata già presa alla linea 20.

4) JMP con il modo di indirizzamento indiretto (la sola istruzione ad usare questo modo) è un mezzo per eseguire una parte di un programma ad indicazione dell'utente. Così l'utente potrebbe specificare un numero da

1 a 4 in base ad una lista sullo schermo. In base all'input, uno dei quattro indirizzi viene trasferito da una tabella ad una locazione conosciuta. Alla fine, JMP in modo indiretto indirizzerà l'esecuzione del programma a quell'indirizzo.

Questi sono alcuni esempi tratti da parallelismi con il BASIC. Essi non si propongono di essere esaurienti ma intendono ricordarvi che JMP va usato nel momento giusto oppure i vostri programmi potrebbero diventare un ammasso di salti.

La procedura per l'uso di JMP è semplice:

1) Selezionare la locazione di memoria alla quale desiderate che il programma salti.
2) Includere una istruzione JMP nel programma al punto in cui volete che il salto sia fatto.
3) Quando l'esecuzione del programma raggiunge l'istruzione JMP, si verifica il salto.

Modi di indirizzamento disponibili con JMP:

```
Assoluto:
JMP $FFF0      : 4C          : 4C F0 FF
Indiretto:
JMP ($FFF0)    : 6C          : 6C F0 FF
```

Nota importante.

Dovuta ad una imperfezione del chip 7501, c'è una importante limitazione dell'uso di JMP in modo indiretto. Se l'indirizzo dal quale l'indirizzo finale deve essere preso finisce in FF, per esempio 01FF, invece di prendere il byte successivo (0200) come secondo byte dell'indirizzo, il processore si sposta indietro all'inizio della stessa pagina di memoria (0100). Questo è un difetto nel processore e non c'è alcuna soluzione per ovviare a ciò, così che dovete porre attenzione a non creare indirizzi senza senso.

Esempio dell'uso di un salto indiretto.

```
  PC          SR  AC  XR  YR  SP
; 1C00         30  00  00  00  F0
> 008B         AA  00  00  00
> 1C8B         00  00  00  00

A  1C00        A9  0B  LDA #$0B
A  1C02        A2  1C  LDX #$1C
A  1C04        85  8B  STA $8B
```

```
A  1C06     86  8C    STX $8C
A  1C08                6C  8B  00
                                    JMP ($008B)
A  1C0B     00         BRK

  PC        SR  AC  XR  YR  SP
; 1C0D       30  0B  1C  00  00
> 008B       0B  1C  00  00
> 1C8B       00  00  00  00
```

Commento:
1C00-1C06: Carica l'indirizzo della istruzione BRK in $8B-$8C.
1C08: Il salto indiretto avviene sulla base dell'indirizzo in $8B-$8C, e BRK è eseguita.

L'uso di subroutine con JSR ed RTS

Funzione: Permette l'esecuzione di una sezione del programma tramite una chiamata JSR da un'altra parte del programma. L'esecuzione di questa sezione secondaria continuerà fino a che non si raggiungerà una istruzione RTS. A questo punto, l'esecuzione del programma ritorna dal punto in cui è avvenuto il salto continuando dall'istruzione seguente a JSR originale.

Le subroutines sono un aspetto essenziale di una buona programmazione. Esse permettono al flusso principale del programma di continuare mentre i compiti minori sono trattati in unità del programma chiaramente identificabili.

Questo punta a rendere programmi più semplici da capire e scrivere e, non ultimo per importanza, li rende più facili da leggere nel caso in cui desideraste apportare dei cambiamenti nello svolgersi del programma. Le subroutines permettono anche di abbreviare i programmi, poiché la stessa subroutine che esegue un compito comune può essere usata da più parti del programma.

Le istruzioni di subroutine disponibili sul chip 7501 non sono dissimili nella loro essenza dai GOSUB e RETURN del BASIC. Per usare una subroutine in docide macchina dovete prima scrivere la vostra subroutine, poi richiamarla usando l'istruzione JSR. La subroutine si concluderà non appena il program counter incontrerà l'istruzione RTS ed il programma principale continuerà dalla istruzione seguente la JSR originale.

Un punto da ricordare, sebbene sia già stato menzionato prima, è che tutto questo funzion solo se non viene alterato lo tack aggiungendo o sottraendo valori da esso durante lo svolgimento della subroutine. Se avete bisogno di usare lo stack durante lo svolgimento della subroutine, naturalmente siete liberi di farlo, ma assicuratevi di togliere tutti i valori che avete immesso e solo quei valori, altrimenti l'istruzione RTS riporterà l'esecuzione in una posizione che per il programma non avrà senso.

Inoltre, è necessario ricordare che ogni istruzione **JSR** memorizza il suo indirizzo di ritorno sullo tack.

L'indirizzo effettivo registrato è di due bytes sopra il byte contenente il codice operativo JSR. Poiché JSR ha solo un modo di indirizzamento, quello assoluto, che richiede due bytes per specificare un indirizzo a cui saltare, l'aggiunta di due è sempre sufficiente per creare un indirizzo che punti all'ultimo byte dell'istruzione JSR. Se chiamate una subroutine e poi ne chiamate un'altra mentre siete ancora nella prima, avvrete posto due indirizzi di ritorno sullo stack. Un ritorno dalla seconda subroutine vi riporterà non nella posizione originale, ma alla prima subroutine. Questa operazione è conosciuta come "nidificazione" di subroutines. In qualche occasione i programmatori la evitano a favore del semplice salto, JMP. La procedura consiste nel chiamare la subroutine 1 che a turno chiama la subroutine 2; la differenza è che mentre la subroutine 1 è chiamata con una istruzione JSR, la subroutine 2 è chiamata con JMP. Il risultato è che quando si incontra un RTS, il ritorno non è fatto alla subroutine 1 ma all'originale JSR. Un esempio di questo potrebbero essere due subroutines che usano la stessa parte finale. Piuttosto che scrivere la parte finale due volte, ssa è scritta una volta alla fine di una delle subroutines, terminando con RTS. L'altra subroutine, quando raggiunge il punto dove si dovrebbe trovare il finale, salta con JMP all'ultima parte della prima subroutine ed usa quel finale, incluso il suo RTS.

Modi di indirizzamento disponibili con JSR:

```
Assoluto:
JSR $FFF0       : 20            : 20 F0 FF
```

Modi di indirizzamento disponibili con RTS:

```
Intrinseco:
RTS             : 60            : 60
```

Esempio dell'uso di RTS.

```
  PC          SR  AC  XR  YR  SP
; 1C00         30  00  00  00  F0
> 008B         AA  00  00  00
> 1C8B         00  00  00  00

A  1C00       A9  1C   LDA #$1C
A  1C02       48       PHA
A  1C03       A9  06   LDA #$06
A  1C05       48       PHA
A  1C06       60       RTS
A  1C07       00       BRK
```

```
  PC      SR  AC  XR  YR  SP
; 1C09     30  06  00  00  F0
> 008B     AA  00  00  00
> 1C8B     00  00  00  00
```

Commento:
1C00-1C05: L'indirizzo di chiusura è posto sullo stack.
1C06: L'esecuzione di RTS guida il chip a prendere i primi due bytes dallo stack, trattandoli come un indirizzo e saltando al byte che segue quell'indirizzo.

Le istruzioni relative "branch"

Funzione: Eseguono un salto dentro il range -128/+127 dei bytes di memoria, contando dal byte seguente l'istruzione branch, sulla base della condizione di uno dei flags.

Oltre alle due forme che abbiamo già esaminato, il chip 7501 fornisce una serie di altre istruzioni di controllo del programma che sono più vicine nella forma al formato BASIC IF...THEN...GOTO. Queste sono conosciute come diramazioni relative, con il termine relativo che indica che invece di dover specificare la totalità di un indirizzo in memoria, il programmatore deve solo dare una posizione relativa rispetto alla posizione del program counter in quel momento.

I branch relativi usano solo un byte per specificare il salto che deve essere fatto e questo salto può essere fatto sia avanti che indietro nella memoria. Questo significa che torniamo indietro all'argomento familiare dei numeri con segno ad un solo byte, e non vi sorprenderete nell'apprendere che un salto può essere fatto nel range da -128 a +127 byte. La posizione da cui il salto è effettuato è il byte dopo l'istruzione di branch. Alcuni esempi generali di come questo funzioni, usando una istruzione immaginaria, BRANCH, può essere:

```
BRANCH 0
LDA #$FF
```

L'effetto dell'istruzione branch qui è nullo. Poichè l'indirizzo base per l'istruzione branch è in ogni caso il primo byte dopo l'istruzione, cioè il primo byte di LDA $FF, il saltare zero byte da quel punto non avrà alcun effetto sullo scorrere del programma, qualunque esso sia.

```
BRANCH -2
LDA #$FF
```

Questo è un loop infinito. Come tutte le istruzioni branch, il nostro BRANCH immaginario è lungo due byte. Se noi saltiamo indietro di due byte dall'inizio dell'istruzione LDA ci troveremo di nuovo all'inizio di BRANCH, e così via all'infinito.

Alla fine:

```
BRANCH 2
LDA #$FF CLC
```

Qui l'istruzione BRANCH porta ad ignorare totalmente i due byte dell'istruzione LDA, così l'esecuzione salta al byte che contiene il codice operativo CLear Carry.

Molti dei problemi che si incontrano usando le istruzioni branch sorgono da errori di conteggio, contando dal posto sbagliato come dall'inizio dell'istruzione branch o dimenticando che l'indirizzo è un numero dotato di segno. Se contate con cura e dal punto corretto, non c'è alcuna ragione per incontrare delle difficoltà.

Le condizioni per un branch

Abbiamo già detto che le istruzioni branch sono simili alle costruzioni IF...THEN GOTO del BASIC, ma come? In realtà, il 7501 vi dà la possibilità di esaminare otto differenti condizioni e fare un branch se una di queste condizioni è eseguita. Le otto diverse condizioni sono basate su quattro flag:

```
CARRY
ZERO
SIGN
OVERFLOW
```

Per ognuno di questi flag c'è una forma di istruzione branch che esegue un branch se il flag è settato ed una che eseguirà un branch se il flag è resettato. Sotto è fornita una tabella che mostra le otto istruzioni branch in relazione ai rispettivi flag:

```
Flag di Carry:
SE SETTATO      : Branch Carry Set (BCS)
SE RESETTATO    : Branch Carry Clear (BCC)
Flag di Zero:
SE SETTATO      : Branch Equal (BEQ)
SE RESETTATO    : Branch Not Equal (BNE)
Flag di Sign:
SE SETTATO      : Branch Minus (BMI)
```

SE RESETTATO : Branch Plus (BPL)
Flag di overflow:
SE SETTATO : Branch O/flow Set (BVS)
SE RESETTATO : Branch O/flow Clear (BVC)

BCS e BCC, i branch relativi al flag di carry, sono essenziali nelle routine per le varie forme di aritmetica e per i paragoni. Si devono prendere spesso delle decisioni in base al fatto che il risultato di una operazione aritmetica sia fuoriuscito dal range di valori che può essere registrato da un singolo byte, e questi due branch permettono di prendere tali decisioni senza fatica. Nel Capitolo 14 abbiamo visto il confronto di numeri ed abbiamo notato che il flag di carry registra se un numero è più grande dell'altro. BCS e BCC sono frequentemente usati per inviare l'esecuzione a routine differenti a seconda del risultato di tale paragone. Nel Capitolo 13 abbiamo visto le istruzioni shift ed abbiamo notato come esse possono essere usate per spostare i contenuti di un byte, un bit per volta, nel flag di carry per vedere quali bit sono settati e quali no. Ancora una volta, ogni azione basata sul risultato di ciò è come se fosse basata su BCS e BCC.

Le istruzioni BNE e BEQ sono usate molto spesso per eseguire paragoni. Vi ricorderete che nel Capitolo 13 abbiamo dato lo schema di un procedimento per paragonare due numeri a due byte. La prima azione consisteva nel confrontare gli high byte e, se erano gli stessi, andare a confrontare i low byte. I dettagli erano stati deliberatamente lasciati vaghi perchè, per realizzare la routine, avevate bisogno di usare l'istruzione BNE per saltare il confronto dei low byte se ogni cosa era stata decisa dal confronto degli high byte. BNE è usato frequentemente alla fine di loop, specialmente quelli che devono essere eseguiti più di 127 volte (vedere i branch relativi al flag di sign). Qui sarà decrementato un registro ogni volta che il programma passa attraverso il loop e quando il registro raggiunge zero, BNE non rinvierà più l'esecuzione all'inizio del loop.

BMI e BPL si riferiscono al flag di sign e sono ovviamente utili quando si passa all'aritmetica con segno. BPL è usata frequentemente anche per loop che devono essere eseguiti meno di 128 volte, poichè permette al loop di girare fino a che il registro usato per contare le ripetizioni raggiunge zero. Solo nel caso di ulteriori ripetizioni il registro BPL andrebbe sotto zero e rinvierebbe l'esecuzione all'inizio del loop.

Le istruzioni BVC e BVS, relative al flag di overflow, sono usate quasi esclusivamente nell'aritmetica con segno.

Branch e loop

In quanto è stato detto sopra, avete trovato riferimento costante ai loop ed all'uso delle istruzioni branch per crearli. I loop sono una parte vitale della

programmazione in codice macchina, persino più che nel BASIC, poichè anche funzioni abbastanza banali richiedono spesso di eseguire più volte una semplice operazione (come la pulizia dello schermo in codice macchina, che richiede che ogni byte della memoria di schermo sia azzerato a turno).

L'uso dei loop per creare un ritardo in un programma, è descritto nel Capitolo 18.

Tabella delle istruzioni branch:

Flag di Carry settato:
: BCS $FE : B0 FE
Flag di Carry resettato:
: BCC $FE : 90 FE
Flag di Zero settato (Equal):
: BEQ $FE : F0 FE
Flag di Zero resettato (Non-Equal):
: BNE $FE : D0 FE
Flag di Sign settato (Minus):
: BMI $FE : 30 FE
Flag di Sign resettato (Plus):
: BPL $FE : 10 FE
Flag di Overflow settato:
: BVS $FE : 70 FE
Flag di Overflow resettato:
: BVC $FE : 50 FE

Esempi dell'uso dei branch

In ognuno dei due esempi, ci sono due istruzioni branch per eseguire l'istruzione BRK, entrambe basate sulla condizione del flag di carry. Eseguendole entrambe ed osservando il contenuto dell'accumulatore ci si accorge che nel primo esempio il primo branch è eseguito ed il secondo ignorato, mentre nella seconda versione il secondo branch è eseguito ed il primo ignorato.

```
  PC          SR  AC  XR  YR  SP
; 1C00        30  00  00  00  F0
> 008B        9A  00  00  00
> 1C8B        00  00  00  00

A  1C00       A9  AA  LDA #$AA
A  1C02       18      CLC
A  1C03       A9  04  BCC $1C09
A  1C05       A9  FF  LDA  #FF
A  1C07       B0  00  BCS $1C09
A  1C09       00      BRK
```

```
  PC      SR  AC  XR  YR  SP
; 1C08     B1  AA  00  00  F0
> 008B     00  00  00  00
> 1C8B     00  00  00  00

  PC      SR  AC  XR  YR  SP
; 1C00     30  00  00  00  F0
> 008B     9A  00  00  00
> 1C8B     00  00  00  00

A  1C00     A9  AA    LDA #$AA
A  1C02     38        SEC
A  1C03     90  04    BCC $1C09
A  1C05     A9  FF    LDA # $FF
A  1C07     B0  00    BCS $1C09
A  1C09     00        BRK

  PC      SR  AC  XR  YR  SP
; 1C0B     B0  FF  00  00  F0
> 008B     00  00  00  00
> 1C8B     00  00  00  00
```

CAPITOLO 16

INTERRUPT

In questo capitolo trattiamo una forma più specializzata del controllo di programma; è quella relativa al comando BRK, ed al suo compagno RTI.

BReaK svolge un compito piuttosto insolito: il suo scopo originale è infatti quello di permettere ai programmatori professionisti di porre rimedio agli errori che hanno fatto. Saprete che le basi del vostro computer, lasciando da parte la CPU del 7501, sono formate da una serie di complessi programmi in codice macchima inseriti in quella che è conosciuta come "read only memory" (memoria di sola lettura), ovvero chip che hanno la particolarità di mantenere nella memoria i dati anche quando il computer viene spento.

Questi programmi ROM sono inseriti di solito in chip più flessibili, le PROM, cioè "ROM Programmabile". L'inconveniente di tali chip è che se, dopo aver sviluppato l'intero sistema operativo ci si accorge che esiste un problema nella programmazione (e questo avviene inevitabilmente), è oltremodo difficoltosa, costosa e dispendiosa in tempo l'alterazione dell'intera struttura del programma inserito nel chip.

La soluzione ovvia è fornirsi di quello che è conosciuto come un "patch", una subroutine posta in una memoria di scorta nel chip, in modo che il programma principale non debba essere spostato. Il "patch" sarà attivato da una chiamata posta nel programma principale. Si pongono ora due problemi. Prima che un chip acquisisca un programma, si trova normalmente con tutti i bit settati (ogni byte contiene il valore $FF). La ragione di questo è che con i chip ROM in produzione attualmente è relativamente facile azzerare permenentemente un bit che è settato, ma è difficile l'operazione contraria. In altre parole, una volta che il programma è stato installato, ed i relativi bit azzerati come parte di un processo, diventa molto difficile rimediare portando a "1" i bit resettati. Per chiamare il "patch" avremmo bisogno di porre una istruzione JSR nel programma contenuto in ROM, cosa del tutto impossibile. Il modo per aggirare questo era azzerare l'istruzione errata dandole un codice operativo pari a zero.

L'istruzione che salva tempo e denaro e che leva le castagne dal fuoco è BRK. La sua funzione è quella di avvisare il chip di cercare un indirizzo ad un punto di memoria specifico, e poi eseguire, partendo da quel punto, una routine in codice macchina conosciuta come routine di interrupt. Il punto al quale deve essere cercato l'indirizzo è inserito nel 7501 stesso, per cui BRK non ha bisogno di essere seguito da un indirizzo.

L'istruzione BRK così come esiste sul C16/Plus 4 è estremamente utile (cortesia della Commodore). Quando il 7501 incontra una istruzione BRK in un programma in codice macchina del C16/Plus 4 guarda, come è solito fare, al luogo dove si aspetta di trovare un indirizzo e si dirige verso una routine inserita in ROM, da là, per poi dirigersi ai due byte a $316-317 (790-1 in decimale) che sono in RAM. Da questi due byte esso prende un indirizzo e passa ad eseguire ogni istruzione in codice macchina trovata al punto indicato a quell'indirizzo.

Il vantaggio di questo, dal punto di vista del programmatore, è che, siccome $316-7 sono nella RAM, l'utente può immagazzinarci l'indirizzo di una routine in codice macchina da eseguire e poi richiamare quella routine usando un solo byte per mezzo BRK.

Esempio dell'uso di BRK.

```
  PC          SR  AC  XR  YR  SP
; 1C00         30  00  00  00  F0
> 008B         9A  00  00  00
> 1C8B         00  00  00  00


A  1C00       A9  0B        LDA #$0B
A  1C02       A2  1C        LDX #$1C
A  1C04       8D  16  03    STA $0316
A  1C07       8E  17  03    STX $0317
A  1C0A       00            BRK
A  1C0B       A0  AA        LDY #$AA
A  1C0D       A9  4C        LDA #$4C
A  1C0F       A2  F4        LDX  #$F4
A  1C11       8D  16  03    STA $0316
A  1C14       8E  17  03    STX $0317
A  1C17       00            BRK


  PC          SR  AC  XR  YR  SP
; 1C19         B4  4C  F4  AA  EA
> 008B         9A  00  00  00
> 1C8B         00  00  00  00
```

Commento:
1C00-1C02: 50-60: Questi due byte caricati nell'accumulatore e nel registro X rappresentano l'indirizzo dell'istruzione LDY $AA più avanti nel programma.

1C04-1C07: L'indirizzo dell'istruzione è immesso nel registro a due byte a $316-7.
1C0A: Esecuzione dell'istruzione BRK. Poichè l'indirizzo dell'istruzione LDY è stato immesso in $316-7, il risultato di BRK è il salto a quell'istruzione.
1C0D-1C14: L'indirizzo esatto dei registri di BRK viene riportato in $316-317 e si esegue un secondo BRK per tornare al Monitor.

Eseguito il programma, avrete notato un punto importante. Il list mostra che prima di eseguire il programma, lo stack pointer è settato ad F0. Quando il programma è terminato il valore è EA, per cui sono stati occupati sullo stack 6 bytes. La ragione di ciò è che parte della routine di BRK richiama automaticamente dallo stack i valori dei registri, immagazzinati all'accettazione del BRK e quindi, cambiando l'indirizzo al quale BRK salta, abbiamo cortocircuitato questo processo. Quando il BRK è accettato, i registri sono salvati ma la routine che li richiama non è eseguita per il primo BRK nel programma, così ci sono sei byte di differenza sullo stack. Il morale di ciò è che, se avete intenzione di alterare il registro di BRK a $316-7, dovete anche ricordare di rimuovere 6 bytes dallo stack prima che il programma ci ritorni. Le tecniche reali per rimuovere valori dallo stack le vedremo nel prossimo capitolo.

Il ritorno con RTI

Il comando corretto per terminare l'esecuzione di una subroutine chiamata per mezzo di BRK è RTI, che significa ReTurn from Interrupt (ritorno da interrupt). Trovando una istruzione RTI, a patto che la subroutine sia stata chiamata con BRK, l'esecuzione del programma ritorna al punto seguente il BRK.

Spesso si fà confusione tra RTS, ritorno da subroutine, ed RTI ma è molto importante che voi le distinguiate subito tra di loro.

Quando una subroutine è chiamata con JSR, un indirizzo a due byte viene posto sullo stack, e quell'indirizzo è l'ultimo byte dell'istruzione JSR. Quando un programma vi fa ritorno con RTS, tale indirizzo è ripreso dallo stack, il program counter è incrementato di uno e viene eseguita l'istruzione che segue JSR.

BRK opera molto diversamente, immagazzinando prima un indirizzo a due byte ed un altro byte che rappresenta il contenuto dello status register. L'indirizzo posto sullo stack, non è solamente un byte extra immagazzinato, ma l'indirizzo dell'istruzione successiva a BRK. Quando si incontra l'istruzione RTI, il program counter e lo status register sono ripristinati ma il program counter non viene incrementato e sta già indicando l'istruzione successiva.

È chiaro, quindi, che voi non potete terminare una subroutine chiamata con BRK con una istruzione RTS in quanto questa prende come indirizzo di ritorno dallo stack, i due byte che costituiscono il contenuto dello status register e l'indirizzo originale del program counter. Nei programmi per il chip 6502, che è molto simile al 7501, voi troverete a volte delle routine di interrupt terminanti con PLP (prendere lo status register dallo stack pointer), seguita da RTS. Questo fà ritornare l'indirizzo corretto al program counter ma sul 7501 non è possibile perchè RTS incrementa il program counter ed il chip salta il byte successivo del programma. Se nei programma per 6502, molti dei quali funzioneranno sul 7501, vi imbattete in PLP,RTS dovete assicurarvi che siano usati per chiudere una routine di interrupt, e quindi sostituire i due comandi con un unico RTI.

Esempio dell'uso di RTI.

```
  PC          SR  AC  XR  YR  SP
; 1C00         30  00  00  00  F0
> 008B         9A  00  00  00
> 1C8B         00  00  00  00

A  1C00        A9  1C  LDA #$1C
A  1C02        48      PHA
A  1C03        A9  08  LDA  #$08
A  1C05        48      PHA
A  1C06        08      PHP
A  1C07        40      RTY
A  1C08        00      BRK

  PC          SR  AC  XR  YR  SP
; 1C0A         30  08  00  00  F0
> 008B         00  00  00  00
> 1C8B         00  00  00  00
```

Commento:

1C00-1C05: L'indirizzo della istruzione BRK è posto direttamente sullo stack.

1C06: Il contenuto dello status register è pure posto sullo stack. Non c'è una relazione particolare con il contenuto attuale del registro, semplicemente è che RTI necessita di un valore in più sullo stack in modo che esso prenda correttamente l'indirizzo a cui vogliamo che salti. Ogni valore andrebbe bene in questo caso particolare, ma in altri potrebbe essere necessario che il valore immagazzinato sia il contenuto reale dello status register.

1C07: Un solo RTI è sufficiente per chiudere l'istruzione BRK,e riportare il programma al Monitor.

Modi di indirizzamento disponibili con BRK ed RTI

L'unico modo che si adatta ad entrambe le istruzioni è il modo intrinseco, poichè le istruzioni indicano esplicitamente i registri cui puntare.

Intrinseco:		
BRK	: 00	: 00
Intrinseco:		
RTI	: 40	: 40

CAPITOLO 17

LO STACK

Nei primi capitoli del libro, studiando in generale i metodi operativi di una CPU, ci siamo imbattuti nel concetto di "stack" come una serie di valori alla quale aggiungere altri valori e dalla quale prendete dei valori.

Sul C16/Plus 4 l'area di memoria che si estende da 256 ($0100) a 511 ($01FF) è riservata allo stack, fatto da tener ben presente per non andare ad immagazzinare dati proprio su di essa.

I valori sono immagazzinati nello stack al contrario di quanto potete verificare poiché il primo valore sullo stack è situato a 511 ed i valori susseguenti sono memorizzati in indirizzi più bassi.

Niente di questo vi deve preoccupare, poiché l'accesso diretto alla memoria stack è molto improbabile. Per tutte le applicazioni pratiche è più facile esprimersi come se i valori fossero posti "sullo" stack e rappresentarlo come un susseguersi di valori a cui se ne possono aggiungere altri e da cui si può togliere solo l'ultimo.

Programma BASIC che illustra uno stack first-in/first-out (FIFO)

Il programma è ideato per visualizzare sullo schermo lo stack semplificato illustrato nel Capitolo 1.

```
10 REM PROGRAMMA PER DIMOSTRARE LO STACK FIFO
20 SPTR = 0
30 FOR I = 0 TO 7
40 GOSUB 170
50 INPUT "IMMETTI IL NUMERO DA PORRE NELLO
STACK:";T(SPTR)
60 SPTR = (SPTR+1) AND 7
70 NEXT I
80 GOSUB 170
90 FOR I = 0 TO 7
100 INPUT "RETURN PER TOGLIERE DALLO STACK";T$
110 SPTR = (SPTR-1) AND 7
120 GOSUB 170
130 PRINT "L'ITEM TOLTO È:" T(SPTR)
140 PRINT
```

```
150 NEXT I
160 END
170 REM*****************************
180 REM PRINT STACK
190 REM*****************************
200 SCNCLR
210 PRINT " [C=A] [^C] [^C] [^C] [^C] [^C] [^C] [C=S] "
220 FOR J = 0 TO 7
230 T$ = RIGHT$("      "+STR$(T(J)),6)
240 PRINT "      [^B] " T$ "[^B]   ;
250 IF J=SPTR THEN PRINT " -STACK POINTER" : ELSE PRINT
270 IF J<7 THEN PRINT "      [C=Q] [^C] [^C] [^C] [^C] [^C] [^C]
[C=W]"
280 NEXT J
290 PRINT "      [C=Z] [^C] [^C] [^C] [^C] [^C] [^C] [C=X]"
300 PRINT
310 RETURN
```

Lo stack è parte integrante del sistema operativo, poiché è usato ampiamente come un blocco note sul quale annotare i valori importanti che temporaneamente non sono richiesti.

Quando un codice macchina, oppure un programma BASIC salta ad una subroutine, la posizione attuale dentro il programma è immagazzinata nello stack così che quando si verifica il ritorno dalla subroutine, il sistema conosce esattamente da dove riprendere nuovamente il programma.

L'uso dello stack non è però limitato al sistema, infatti è usato anche nella programmazione.

Lo stack è una delle parti più semplici e grezze del sistema e va trattato semplicemente come un notes sul quale riportare non più di 256 valori.

La principale limitazione di questo notes è che l'ultimo valore informativo posto in esso è sempre il primo ad essere richiamato, poi il prossimo e così via. La dimenticanza di questo principio "last in... first out" è una delle cause più frequenti del malfunzionamento dei programmi in codice macchina. Se immagazzinate i valori X, Y e Z, non c'è l'immediata possibilità di riaverli indietro nello stesso ordine per cui dovete programmare in modo tale che i valori siano richiamati nell'ordine contrario: Z, Y, X.

L'unica complicazione reale nell'uso dello stack è che esso è condiviso con il sistema, che pure lo usa come un blocco note. Se cadete in disattenzione, è possibile che la CPU, svolgendo i suoi compiti, prende il valore sbagliato dallo stack bloccando il sistema.

Per queste ragioni, è essenziale che, usando lo stack abbiate la precauzione di contare quanti valori immettere e vi assicuriate che essi siano rimossi al termine delle operazioni.

Per esempio, abbiamo già menzionato che se chiamate una subroutine nel corso del programma in codice macchina, il punto nel quale il programma deve continuare, quando la subroutine è terminata, è registrato sullo stack. Se, durante lo svolgersi della subroutine, voi aggiungete valori sullo stack che poi dimenticate di rimuovere, o se rimuovete più valori di quelli che avete immesso, quando la subroutine finisce, il sistema prende i due valori dal vertice dello stack e li considera come indirizzo valido per ritornare al programma principale.

Questo non vuole scoraggiare l'uso dello stack, che rappresenta una parte importante del bagaglio di un programmatore in codice macchina, ma è semplicemente un avviso per assicurarvi di ver fatto i conti correttamente.

Underflow ed Overflow

Un'altra limitazione importa nell'uso dello stack è quella della sua forma. Abbiamo già notato che lo stack è lungo 256 bytes, almeno potenzialmente, ma è importante capire le implicazioni di questa limitazione.

Poiché il sistema conosce dove inizia lo stack, esso usa un registro ad un solo byte (lo stack pointer) per registrare la posizione del "vertice" rispetto all'inizio, dato che un singolo byte può solo registrare valori nel range 0-255.

Se ponete nello stack un numero tale da fargli superare 255, si verifica l'azzeramento con la susseguente sistemazione dei valori eccedenti. L'ovvio inconveniente di questo "overflow" è che i valori precedentemente immagazzinati andrebbero persi, con conseguenze probabilmente fatali per il programma.

Parallelamente a questo problema di "overflow" c'è l'underflow. Se così tanti valori sono estratti dallo stack che lo stack pointer scende sotto zero, esso si resetta a 255 indicando l'esaurimento della memoria di stack. Anche in questo caso, saranno presi dei valori a capo, ancora una volta con serie conseguenze.

Lo scopo di questo avvertimento, lo ripetiamo, non è quello di scoraggiarvi dall'immagazzinare grandi quantità di dati sullo stack, poiché ciò è molto improbabile, ma di rendervi consapevoli che possono sorgere inaspettatamente dei problemi di difficole soluzione. Alcuni programmi abbastanza semplici, per esempio, usano una tecnica chiamata "recursion" (ricorrenza), dove è chiamata una subroutine che poi continua a chiamare essa stessa. La tecnica può essere estremamente utile purché si ricordi che ogni chiamata di subroutine aggiunge allo stack ed ogni ritorno da una subroutine sottrae dallo stack.

È sorprendentemente facile cadere fuori dei limiti da una parte o dall'altra.

Due programmi BASIC per illustrare l'overflow e l'underflow

Il primo di questi due programmi dimostra l'overflow richiedendo di immettere nove valori sullo stack. Il secondo pone otto valori sullo stack e cerca di rimuovere il nono, dimostrando così l'underflow.

```
10 REM PROGRAMMA DI DIMOSTRAZIONE DELL'OVERFLOW
DELLO STACK
20 SPTR = 0
30 FOR I = 0 TO 8
40 GOSUB 170
50 INPUT "NUMERO DA PORRE NELLO STACK";T(SPTR)
60 SPTR = (SPTR+1) AND 7
70 NEXT I
80 GOSUB 170
90 FOR I = 0 TO 8
100 INPUT "RETURN PER TOGLIERE DALLO STACK";T$
110 SPTR = (SPTR-1) AND 7
120 GOSUB 170
130 PRINT "L'ITEM TOLTO È:" T(SPTR)
140 PRINT
150 NEXT I
160 END
170 REM*****************************
180 REM PRINT STACK
190 REM*****************************
200 SCNCLR
210 PRINT " [C= A][^C][^C][^C][^C][^C][C= S]"
220 FOR J = 0 TO 7
230 T$ = RIGHT$(""+STR$(T(J)),6)
240 PRINT "[^B]" T$ "[^B]";
250 IF J=SPTR THEN PRINT "- STACK POINTER" : ELSE PRINT
270 IF J<7 THEN PRINT " [C= Q][^C][^C][^C][^C][^C][C= W]"
280 NEXT J
290 PRINT " [C= Z][^C][^C][^C][^C][^C][C= X]"
300 PRINT
310 RETURN
```

```
10 REM PROGRAMMA DI DIMOSTRAZIONE DELL'UNDERFLOW
DELLO STACK
20 SPTR = 0
30 FOR I = 0 TO 6
40 GOSUB 170
50 INPUT "NUMERO DA PORRE NELLO STACK:"; T (SPTR)
60 SPTR = (SPTR+1) AND 7
70 NEXT I
80 GOSUB 170
90 FOR I = 0 TO 7
100 INPUT "RETURN PER TOGLIERE DALLO STACK"; T$
110 SPTR = (SPTR-1) AND 7
120 GOSUB 170
130 PRINT "ITEM TOLTO DALLO STACK:" T(SPTR)
140 PRINT
150 NEXT I
160 END
170 REM************************
180 REM PRINT STACK
190 REM************************
200 SCNCLR
210 PRINT "[C= A] [^C] [^C] [^C] [^C] [^C] [C= S]"
220 FOR J = 0 TO 7
230 T$ = RIGHT$ (""+STR$(T(J)),6)
240 PRINT "[^B]" T$ "[^B]";
250 IF J=SPTR THEN PRINT "- STACK POINTER": ELSE PRINT
270 IF J<7 THEN PRINT "[C= Q] [^C] [^C] [^C] [^C] [^C] [^C] [C=W]"
280 NEXT J
290 PRINT "[C= Z] [^C] [^C] [^C] [^C] [^C] [^C] [C= X]"
300 PRINT
310 RETURN
```

Lo Stack e l'Accumulatore : le istruzioni PHA e PLA

La prima delle due serie di istruzioni che il chip 7501 fornisce per accedere allo stack è quella relativa all'accumulatore. L'istruzione PHA è l'abbreviazione di PusH Accumulator. Il termine "push è usato con la maggior parte dei chip per indicare che sullo stack deve essere posto un valore. PLA sta per PulL Accumulator. Anche "pull è usato quasi universalmente, indifferentemente dal tipo di chip, per indicare che un valore deve essere estratto dallo stack.

I due comandi si spiegano, quindi, da soli:
L'istruzione PHA pone il contenuto corrente dell'accumulatore sul vertice dello stack, e lo stack pointer è incrementato di 1 in modo tale che indichi il nuovo valore. Il contenuto dell'accumulatore rimane invariato e non c'è alcun effetto sui vari flag.

L'istruzione PLA prende il valore dal vertice dello stack (quello indicato realmente dallo stack pointer) e lo immette nell'accumulatore. Lo stack pointer è decrementato di 1 per indicare che il vertice dello stack è ora un valore più basso. Notate che il contenuto effettivo dello stack è irrilevante per la posizione del vertice dello stack. I valori prelevati non sono realmente rimossi, semplicemente è che lo stack pointer si abbassa ed essi vengono ignorati. Gli inserimenti successivi si sovrapporranno ad essi. In contrasto con PHA, l'istruzione PLA interessa i flag nello stesso modo in cui lo farebbe ogni altro metodo di caricamento dell'accumulatore.

PHA e PLA sono usate per procurare un immagazzinamento temporaneo ai dati che, viceversa, potrebbero essere danneggiati dallo svolgimento del programma. Accade spesso il caso che, per esempio, i valori inseriti nell'accumulatore e negli altri registri siano significativi per la parte principale del programma, ma che una chiamata ad una subroutine cambia questi valori in un modo tale che non abbiano più senso quando, completata la routine, il programma principale continuerà. In tali circostanze, PHA sarebbe usata per porre il contenuto dell'accumulatore sullo stack. I contenuti dei registri X ed Y, o qualsiasi altro valore che necessita di essere preservato, sarebbe poi a turno trasferito all'accumulatore ed immagazzinato sullo stack allo stesso modo. Quando tutti i valori sono immagazzinati, può essere chiamata la subroutine ed essere lasciata libera di fare uso di tutti i registri con la sicurezza che qualsiasi cambiamento sia fatto, non avrà effetto sui valori immagazzinati sullo stack. Quando la subroutine è terminata, i valori possono essere tolti dallo stack, ricordando che essi sono nell'ordine contrario a quello con cui sono stati immagazzinati, e ricollocati dall'accumulatore alle loro aree originali. In questo modo, a patto che voi contiate correttamente i valori che si pongono e che si levano dallo stack, il numero limitato di registri disponibili dentro la CPU può essere usato per una vasta varietà di compiti, quasi contemporaneamente, senza pericolo di confusione.

Esempio che illustra l'immagazzinamento del contenuto dell'accumulatore sullo stack.

```
  PC          SR  AC  XR  YR  SP
; 1C00        30  00  00  00  F0
> 008B        9A  00  00  00
> 1C8B        00  00  00  00

A  1C00       68      PLA
A  1C01       00      BRK
```

```
  PC        SR  AC  XR  YR  SP
; 1C03       ??  ??  00  00  F1
> 008B       00  00  00  00
> 1C8B       00  00  00  00
```

Esempio che illustra il ricupero del contenuto dell'accumulatore dallo stack.

```
  PC        SR  AC  XR  YR  SP
; 1C00       30  AA  00  00  F0
> 008B       9A  00  00  00
> 1C8B       00  00  00  00

A  1C00      48      PHA
A  1C01      00      BRK
  PC        SR  AC  XR  YR  SP
; 1C03       30  AA  00  00  EF
> 008B       00  00  00  00
> 1C8B       00  00  00  00
```

Commento:
In entrambi i casi le routine operano con qualsiasi valore ci sia nell'accumulatore o sullo stack. Potreste cercare di modificare la prima routine in modo che essa prenda un valore dalla locazione di memoria $FB, immetta quel valore nell'accumulatore e poi lo ponga sullo stack.

Immagazzinamento del contenuto dello status register con le istruzioni PHP e PLP

L'unico problema che rimane quando si tratta di immagazzinare valori sullo stack è quello relativo allo status register del processore. Pensando all'esempio della chiamata di una subroutine, dato in precedenza, l'intera procedura non è granchè utile se i flag non possono essere immagazzinati assieme ai registri. Come gli altri registri o locazioni di memoria, essi possono essere trasferiti nell'accumulatore e poi sullo stack. Lo stautus register, tuttavia, non può essere trasferito in nessuno altro registro o locazione di memoria. Di conseguenza esso è fornito di proprie speciali istruzioni di stack, PHP (PusH Processor status register) e PLP (PulL Processor status register). Il loro effetto è di immettere il contenuto corrente dello status register sullo stack, senza cambiare tali contenuti, o prendere il valore attualmente al vertice dello stack ed inserirlo nello status register.

PHP e PLP sono spesso usate assieme a PHA e PLA per salvare il contenuto dello status register assieme agli altri registri. In altre occasioni sono usate da sole. È comune, per esempio, voler fare una serie di test o di

manipolazioni del risultato di un calcolo. Il problema è che queste manipolazioni normalmente cambiano i flag così che quando il secondo test inizia i flag non si riferiranno al risultato originale ma a quello che è stato fatto in seguito. In tali casi si può immettere sullo stack il valore dello status register, eseguire un esame sui flag, e poi riportare il valore originale dentro lo status register affinchè possa essere esaminato un altro flag.

Esempio dell'uso di PHP.

```
  PC          SR  AC  XR  YR  SP
; 1C00        30  00  00  00  F0
> 008B        00  00  00  00
> 1C8B        00  00  00  00

A  1C00       28      PLP
A  1C01       28      BRK
  PC          SR  AC  XR  YR  SP
; 1C03        ??  00  00  00  F1
> 008B        00  00  00  00
> 1C8B        00  00  00  00
```

Esempio dell'uso di PLP.

```
  PC          SR  AC  XR  YR  SP
; 1C00        30  00  00  00  F0
> 008B        00  00  00  00
> 1C8B        00  00  00  00

A  1C00       08      PHP
A  1C01       00      BRK

  PC          SR  AC  XR  YR  SP
; 1C03        30  10  00  00  EF
> 008B        00  00  00  00
> 1C8B        00  00  00  00
```

Commento:
Queste dimostrazioni rivelano come avviene un processo di push" o di pull.

Modi di indirizzamento disponibili con le istruzioni push e pull

Non ci sono modi di indirizzamento, come noi normalmente li interpretiamo, disponibili con le istruzioni push e pull. Le due locazioni a cui si riferiscono, sia l'accumulatore e lo stack come lo status register e lo stack,

sono inserite nella istruzione stessa, come con le istruzioni transfer che abbiamo visto nel Capitolo 7. La forma delle istruzioni è la seguente:

Push accumulatore:	PHA	: 48	: 48
Pull accumulatore	PLA	: 68	: 68
Push status register	PHP	: 08	: 08
Pull status register	PLP	: 28	: 28

CAPITOLO 18

OCCUPATO FACENDO NIENTE

Una istruzione che necessita di un breve commento, è NOP, che significa No OPeration (nessuna operazione). Il potere di NOP è quello di costringere la CPU ad attendere per due microsecondi, senza influenzare il contenuto di alcun registro o locazione di memoria.

NOP può essere una istruzione molto utile perchè può essere usata per semplificare alcune lunghe routine in codice macchina, sia come utile aiuto nel debugging dei programmi.

NOP e la temporizzazione

Migliorando la vostra programmazione in codice macchina, vi scoprirete capaci di usare i loop sia per temporizzare le azioni sia semplicemente per rallentare un programma in certi punti. Nei giochi in codice macchina standard, per esempio, non c'è possibilità di muovere un oggetto attraverso lo schermo alla massima velocità disponibile poichè esso risulterebbe praticamente invisibile. Molto spesso in tali giochi, i comandi che creano movimento sono separati da loop abbastanza lunghi che non fanno altro che rallentare il programma.

NOP può essere utile in queste circostanze perchè è di gran lunga piiù facile porre due o tre istruzioni NOP dentro un loop piuttosto che usare due o più loop per superare la limitazione che un solo loop non può essere eseguito più di 256 volte. I due esempi seguenti illustrano il problema:

Esempio di un loop temporale senza istruzioni NOP.

```
  PC        SR  AC  XR  YR  SP
; 1C00       30  FF  FF  FF  FF
> 008B       FF  FF  FF  FF
> 1C8B       FF  FF  FF  FF

A  1C00      A2  00   LDX #$00
A  1C02      86  8B   STX  $8B
A  1C04      A0  00   LDY  #$00
```

```
A  1C06    88       DEY
A  1C07    D0  FD   BNE $1C06
A  1C09    CA       DEX
A  1C0A    D0  FA   BNE $1C06
A  1C0C    C6  8B   DEC $8B
A  1C0E    D0  F6   BNE $1C06
A  1C10    00       BRK

  PC       SR  AC   XR  YR  SP
; 1C12     32  FF   00  00  F0
> 008B     00  FF   FF  FF
> 1C8B     FF  FF   FF  FF
```

Commento:
1C00-1C04: I registri X ed Y e la locazione di memoria $8B sono caricati tutti con zero.
1C07-1C0E: Tre loop "nidificati" da zero a 255 per un ritardo totale equivalente ad un loop eseguito 4278190080 volte. Il tempo impiegato nell'esecuzione è di circa un minuto, prima che il controllo ritorni al Monitor.

Esempio di un loop temporale che usa NOP.

```
  PC       SR  AC   XR  YR  SP
; 1C00     30  FF   FF  FF  F0
> 008B     FF  FF   FF  FF
> 1C8B     FF  FF   FF  FF

A  1C00    A2  00   LDX #$00
A  1C02    86  8B   STX $8B
A  1C04    A0  00   LDY #$00
A  1C06    EA       NOP
A  1C07    EA       NOP
A  1C08    EA       NOP
A  1C09    EA       NOP
A  1C0A    EA       NOP
A  1C0B    88       DEY
A  1C0C    D0  F8   BNE $1C06
A  1C0E    CA       DEX
A  1C0F    D0  F5   BNE  $1C06
A  1C11    C6  8B   DEC $8B
A  1C13    D0  F1   BNE  $1C06
A  1C15    00       BRK
```

```
  PC      SR  AC  XR  YR  SP
; 1C17     32  FF  00  00  F0
> 008B     00  FF  FF  FF
> 1C8B     FF  FF  FF  FF
```

Commento:
Esattamente la stessa routine eccetto che nel loop interno. Sono state aggiunte cinque istruzioni NOP. L'effetto ricavato è l'incremento del tempo per cui lo schermo rimane spento a circa 4 minuti. Per ottenere ciò senza le NOP ci sarebbe voluto l'intervento di un altro loop, con tutte le sue complicazioni.

NOP ed il debugging

L'altro uso principale di NOP, quello più importante, è nel debugging. Esso permette di dare ad un programma una struttura più flessibile, isolando le singole istruzioni malfunzionanti.

a) NOP e la struttura di un programma: quando si sviluppano dei programmi è spesso utile separare singole routine con blocchi di istruzioni NOP. Il vantaggio di questo è che se una routine ha bisogno di essere allungata, si perde tuttalpiù qualche NOP senza dover spostare tutto il resto del programma in memoria.
b) NOP ed il debugging: Una tecnica utile quando un programma gira male è quella di sostituire i comandi sospettati, con istruzioni NOP. Questo equivale ad immettere una REM all'inizio di una linea in BASIC per vedere se togliendo la linea si risolve il problema di programmazione. Quando si usa NOP in questo modo è importante ricordare che è il complesso dell'istruzione che deve essere sostituito, e non semplicemente il codice operativo.

Epilogo

Sebbene abbiate ancora molto da imparare pensiamo che a questo punto possiate ritenervi soddisfatti dei primi passi sulla strada che vi porterà ad essere un programmatore in codice macchina che sa il fatto proprio.

Con l'esperienza acquisita dalle routine contenute in questo libro potete iniziare a sperimentare a vostro piacere. Il Monitor vi tornerà utile, permettendovi di costruire semplici routine per vostro conto, senza paura che queste vi distruggano il sistema. Il vostro prossimo passo potrebbe essere quello di prendere un assembler più sofisticato ed alcuni libri pratici di programmazione in codice macchina che vi mostrino come applicare quanto imparato.

La cosa più importante è che ora siate consapevoli che non c'è nulla da temere dal linguaggio macchina, ma che, anzi, esso vi fà sentire il vostro computer ancora più vicino.

Ecco finalmente il volume che affronta la programmazione in linguaggio macchina del C16 con un linguaggio non specialistico. Non vi sono né formule da ricordare, né hardware da studiare, ma solamente concetti da capire ed esempi per assicurarsi di aver capito.
Lo scopo di questo volume non è quello di insegnare a programmare in codice macchina, bensì quello di preparare chi legge a libri più impegnativi e dettagliati che, molto spesso, danno per scontato la conoscenza delle nozioni base.

L. 16.000 Cod. CC329 ISBN 88-7056-435-5