

2.0.9 Scrittura delle istruzioni con linguaggio simbolico. Progetto del software

Non si può certo pensare che il programmatore scriva un programma nel modo rappresentato nella **RAM** di **Fig.2.0.7.1**, dato che dovrebbe ricordare a memoria il significato che hanno, per la **CPU**, tutti i bit che compongono ogni singolo codice operativo. Per questo motivo i programmi vengono scritti, utilizzando particolari *linguaggi* costruiti ad hoc e costituiti da una successione di *frasi*, generalmente una per ogni istruzione, scritte con vocaboli predefiniti e nel rispetto di una sintassi più semplice possibile.

Il programma di **Fig.2.0.7.1** scritto in *linguaggio* (o *codice*) *macchina* e memorizzato nelle locazioni a 10 bit, dall'indirizzo 0 fino all'indirizzo 5, è, come già detto, riferito alla **CPU** studiata. Il programma in questione, può essere scritto usando un particolare linguaggio simbolico, detto *linguaggio* (o *codice*) *assembly*, secondo la seguente corrispondenza:

<u>linguaggio macchina</u>		<u>linguaggio assembly</u>	
0	0100001010 (8 10)	inizio: MOVE	#10, D0
1	01011----- (11 --)	SHL	D0
2	0001001010 (2 10)	ADD	10, D0
3	0010001011 (4 11)	SUB	11, D0
4	0011101100 (7 12)	MOVE	D0, 12
5	1000100101 (17 5)	fine: JMP	fine

(2.0.9.1)

Il programma diviene, in codice assembly, la successione di sei frasi, in cui si usano *parole chiave*, mediate dalla lingua inglese, come **MOVE**, che ha il significato di “copia”, come **ADD**, parola inglese che significa “Somma”, come **SUB**, contrazione della parola inglese SUBtract, che significa “sottrai”, ed, infine, come **JMP**, che è la contrazione della parola in inglese “JuMP” che significa “Salta”. Queste parole, tante quante sono le operazioni che possono essere comandate alla **CPU**, costituiscono un *vocabolario* della lingua assembly e vengono indicate come *codice mnemonico* del codice operativo. Necessariamente, per programmare in assembly dette parole devono essere conosciute a memoria dal programmatore.

In questo linguaggio, sapendo che le istruzioni sono memorizzate ad indirizzi consecutivi della memoria, non è necessario indicare esplicitamente gli indirizzi a fianco di ogni istruzione. Si è visto comunque l'esistenza di istruzioni di salto, per cui, in alcuni casi, vi può essere la necessità di “marcare” alcune istruzioni, alle quali si deve fare esplicito riferimento, o che iniziano una parte significativa di codice. Per questo motivo è consentito l'uso di parole, inventate ad hoc, seguite dal carattere “:” e chiamate *etichette* (in inglese *label*), che rappresentano, in modo simbolico, gli indirizzi delle istruzioni cui si riferiscono. Nel programma (2.0.9.1) l'etichetta “**inizio:**” corrisponde all'indirizzo 0 nel quale è memorizzata la prima istruzione, l'etichetta “**fine:**” corrisponde all'indirizzo 5, al quale è memorizzata l'ultima istruzione. Questi stesso indirizzo simbolico compare anche nel campo operando dell'istruzione di “salto sul posto”.

Le prima, la terza, la quarta e la quinta istruzione del programma coinvolgono due dati, detti **operandi**, posti, uno in memoria e l'altro nel registro **D0**, interno alla **CPU**. Nel codice assembly il riferimento ai due dati è ottenuto facendo seguire il codice mnemonico con due espressioni separate da una virgola. Il dato referenziato dall'espressione a sinistra della virgola è detto **operando sorgente**, quello referenziato dall'espressione a destra della virgola, è detto **operando destinazione**.

La prima istruzione contiene nella posizione sorgente l'espressione #10 che significa “dato di valore 10”. La posizione di destinazione (espressione dopo la virgola) della prima istruzione contiene solo il nome del registro **D0** a significare “dato contenuto nel registro **D0**”. Con queste regole, la prima istruzione può essere letta complessivamente: “copia il dato 10, che si trova nel campo operando dell'istruzione, nel registro **D0**”.

La seconda istruzione usa un solo operando, **D0**, e comanda di raddoppiare il valore che si trova in questo registro con lo scorrimento di un bit a sinistra (Shift Left).

La terza istruzione comanda “somma al dato contenuto nel registro **D0** il dato che si trova in memoria all'indirizzo 10 e pone il risultato nel registro **D0**”. Il registro **D0** usato come sorgente di uno degli addendi e destinazione della somma è detto **registro accumulatore**. Questa è una regola generale che vale per tutte le istruzioni di elaborazione logico aritmetica a due operandi.

La quarta istruzione comanda la differenza del dato in **D0** con il dato di memoria di indirizzo 11.

Infine la quinta termina l'elaborazione ponendo il risultato che è in **D0** nella locazione di memoria di indirizzo 12.

L'ultima istruzione non fa parte delle istruzioni di elaborazione ma di **controllo della sequenza** di esecuzione, infatti obbliga la **CPU** ad eseguire sempre se stessa, limitandosi a mettere, ripetutamente nel registro **PC**, sempre il suo indirizzo (salto sul posto).

Per referenziare gli operandi elaborati dall'istruzione, oltre ai modi indicati nel programma esemplificato, vi sono altre possibilità. L'insieme di queste possibilità determina i **modi di indirizzamento**. La prima istruzione utilizza il modo di **indirizzamento immediato**, la seconda il modo di **indirizzamento a registro**, la terza e quarta utilizzano il modo di **indirizzamento diretto** alla memoria, per l'operando sorgente, e indirizzamento a registro per quello destinazione, la quinta inverte questi due modi per i suoi operandi, la sesta usa ancora indirizzamento diretto.

Le capacità elaborative di una CPU sono tanto più elevate quanto maggiore è l'insieme di operazioni che è in grado di eseguire sui dati e quanto più ricco è l'insieme dei modi di indirizzamento. L'elenco dei principali modi di indirizzamento è rinviato ad un paragrafo successivo.

Queste poche premesse sulla scrittura il linguaggio (o codice) assembly, sono uno strumento utile all'apprendimento di nuove istruzioni dato che, per comprenderne la logica, sarà sufficiente la loro scrittura in linguaggio assembly, ignorando la scrittura delle stesse in **codice macchina**, utile solo alla CPU!

A questo punto è possibile delineare quali sono le fasi fondamentali che portano alla produzione del software di un calcolatore, scritto in linguaggio assembly. La prima stesura del programma viene fatta, dal programmatore, in questo linguaggio, chiamato in gergo **programma**, o **codice sorgente**. Successivamente il programma deve essere *tradotto* nel linguaggio macchina, unico “comprensibile” dal processore, e, in questa forma, posto nella memoria centrale per essere eseguito. La traduzione del codice sorgente in codice macchina è detta anche **codice oggetto**.

Questa traduzione potrebbe, volendo, essere fatta anche manualmente, per mezzo di fogli di specifica che indicano il **set** (insieme) **di istruzioni** della **CPU** utilizzata.

Nel caso della **CPU** descritta nel paragrafo precedente i fogli di specifica, da un'analisi dei microprogrammi contenuti nella **ROM** di controllo di **Fig.2.0.7.1**, potrebbero assumere il seguente formato:

<i>Codice Operativo</i>	<i>Codice mnemonico e formato istruzione</i>	<i>Operazione eseguita</i>
00010 _{bin} = 2 _{dec}	ADD <mem>, <reg>	<reg> = <reg> + <mem>
00100 _{bin} = 4 _{dec}	SUB <mem>, <reg>	<reg> = <reg> - <mem>
00110 _{bin} = 6 _{dec}	MOVE <mem>, <reg>	<reg> = <mem>
00111 _{bin} = 7 _{dec}	MOVE <reg>, <mem>	<mem> = <reg>
01000 _{bin} = 8 _{dec}	MOVE #dato, <reg>	<reg> = dato
01001 _{bin} = 9 _{dec}	SHR <reg>	<reg> = <reg> / 2
01011 _{bin} = 11 _{dec}	SHL <reg>	<reg> = 2 · <reg>
01101 _{bin} = 13 _{dec}	BEQ <mem>	PC = <mem> se Z =1
01111 _{bin} = 15 _{dec}	BPL <mem>	PC = <mem> se N =0
10001 _{bin} = 17 _{dec}	JMP <mem>	PC = <mem>

I codici mnemonici **ADD**, **SUB**, **MOVE**, **SHL** e **JMP** sono già stati spiegati in precedenza. Il codice **SHR** è un acronimo che deriva dalla frase **SH**ift **R**ight e che significa scorrimento a destra di un bit, equivalente alla divisione per la base 2. **BEQ** deriva da **B**ranch on result **E**qual cioè salta se il risultato dell'ALU è uguale a zero. **BPL** deriva da **B**ranch on result **P**lus cioè salta se il risultato dell'ALU è positivo. I (o il) campi operandi indicano, in modo formale, le informazioni che debbono completare l'istruzione in assembly. Così <reg> esprime il fatto che in quella posizione deve essere indicato il nome di un registro; nel caso esemplificato solo **D0**. <mem> sta ad indicare che deve essere riferita una locazione di memoria; nella macchina esemplificata ciò può essere fatto solo scrivendo un indirizzo di memoria (indirizzamento diretto). Nel seguito si vedranno altre modalità di indirizzamento alla memoria.

In realtà, al fine di evitare un lavoro alquanto noioso e passibile di errori, dopo la scrittura del programma in codice assembly, si ricorre ad un ***programma di traduzione***, da codice assembly a codice oggetto, detto ***Assemblatore (Assembler)***. In questa prospettiva, si comprende la necessità di scrivere il codice sorgente nel rispetto rigoroso della sua grammatica, così rapidamente sintetizzata in precedenza, se si vuole evitare che il programma di traduzione non si rifiuti di svolgere il suo lavoro a causa di errori di scrittura.

Dato che l'attività svolta da un calcolatore può essere anche notevolmente complessa, è, solitamente, richiesta, al programmatore, la capacità di scomporre il problema complessivo in un insieme di sottoproblemi più semplici (*top-down design*), per ognuno dei quali scrivere un *modulo di programma* separato. Anzi i diversi moduli possono essere scritti da programmatori diversi, purché, per ognuno, sia ben chiaro quali, e “dove”, sono i dati da *elaborare (dati di ingresso)* e quali, e “dove”, sono i risultati ottenuti (*dati di uscita*). In questa prospettiva ogni programmatore non sa come il suo modulo, dopo la traduzione in codice macchina, sarà collegato agli altri moduli nella memoria centrale, oltre a non conoscere la posizione occupata dalla prima istruzione del programma complessivo. Per questi motivi, la traduzione fatta dal programma assemblatore presume, solitamente, che la prima istruzione di ogni modulo sia memorizzata a partire dall'indirizzo (convenzionale) zero. Ad un secondo programma, detto ***Collegatore (o Linker)***, è lasciato il compito di giustapporre i singoli moduli ad indirizzi consecutivi, a partire dal primo modulo, per creare un unico programma in codice macchina, che va sotto il nome di ***programma eseguibile***. Un terzo programma, detto ***Caricatore (o Loader)*** ha il compito di porre nella memoria centrale il programma eseguibile in un'area di memoria disponibile a partire da un prefissato indirizzo iniziale.

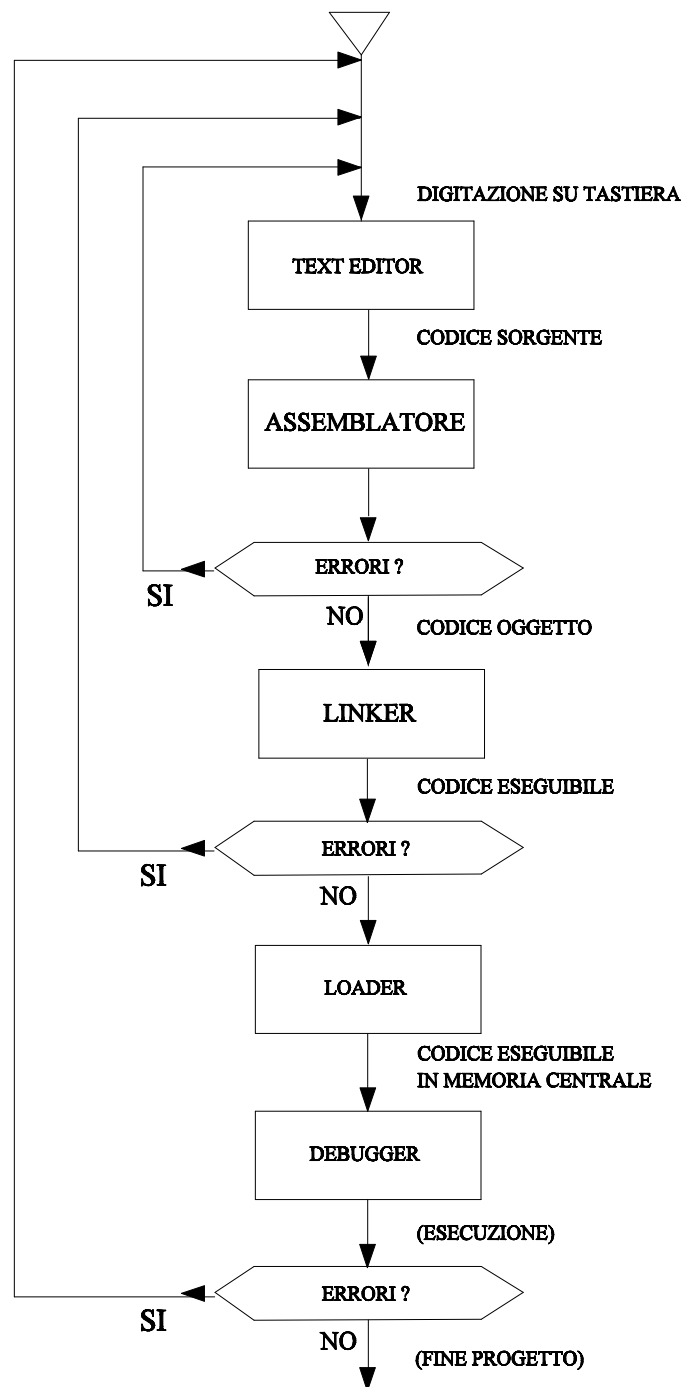


Fig. 2.1.4.1 Fasi di progetto del software.

La fase finale del progetto del software è quella di collaudo ed è coadiuvata da un programma, detto **Debugger**, che coesiste nella memoria centrale con il programma realizzato (sistema multiprogrammato). Il collaudo viene effettuato valutando se, porzioni più o meno ampie del programma in prova, rispettano la logica di funzionamento prevista, utilizzando speciali funzioni del debugger che consentono di analizzare sia risultati intermedi sia i risultati finali, delle elaborazioni effettuate.

Per ognuna delle suddette fasi di progetto si possono verificare degli errori che obbligano a ripetere le fasi precedenti, secondo la logica espressa in **Fig. 2.1.4.1**.

In questa figura sono mostrati anche gli strumenti utilizzati per la progettazione. Il primo strumento usato è un programma Editore di testi (**Text Editor**), che consente di scrivere il codice sorgente e di memorizzarlo in un file come qualunque altro testo scritto. La successiva fase di assembler porta al codice oggetto solo se nel codice sorgente non sono stati trovati errori di grammatica. Segue la fase di linker che produce il codice eseguibile e, in mancanza di errori le successive fasi di loader e di debugger. Eventuali errori trovati in fase di esecuzione obbligano a modificare il codice sorgente e quindi a tornare all'inizio della procedura.

Le operazioni di assembler, linker e loader possono essere eseguite su un calcolatore contenente la **CPU** che dovrà eseguire il codice oggetto o su qualsiasi altra macchina. In quest'ultimo caso si parla di **cross-assembler**, **cross-linker**. La grande diffusione del PC (Personal Computer, non confondere con il Program Counter), per il quale sono disponibili grandi quantità di programmi, per i più svariati tipi di impieghi, consiglia di usare questa macchina, come "ospite" (**host computer**) delle fasi di assembler e di linker, e di collegare ad essa, solitamente con una connessione seriale **RS-232**, una scheda madre, contenente la **CPU** cui è destinato il codice oggetto (**target**), detta **emulatore**. In questo caso la fase di loader è eseguita da un programma contenuto nell'*host computer*, che trasmette il codice oggetto, e da un programma, contenuto nell'emulatore, che riceve i dati inviati dall'*host*. Il programma debugger è, solitamente, eseguito dall'emulatore.

Il sistema complessivo formato dal PC e dall'emulatore, comprensivo di tutto il software necessario al progetto di un programma, viene detto **sistema di sviluppo**.

2.1.5 Tipi di istruzioni

L'uso sempre più diffuso degli elaboratori, in tutti i settori tecnologici, ha stimolato le case produttrici di microchip a realizzare un consistente numero di CPU che si differenziano: per architettura interna, per velocità di esecuzione delle istruzioni, per numero di operandi elaborati da ogni istruzione, per modalità di indirizzamento, per insieme di istruzioni eseguibili (Instruction Set).

In tutti i casi, la tendenza dei costruttori di processori è stata quella di fornire CPU capaci di eseguire programmi con velocità sempre maggiori. A questo riguardo, nell'ultimo decennio, si sono andate differenziando due filosofie. La prima che ha portato a realizzare CPU capaci di eseguire istruzioni sempre più complesse in modo che la raccolta di un solo codice operativo comandasse un'elaborazione abbastanza impegnativa, magari su un discreto numero di dati. L'aumento di velocità è così ottenuto riducendo le fasi fetch e quindi il tempo di raccolta delle istruzioni dalla memoria. Questi processori appartengono alla categoria CISC (Complex Instruction Set Computer). La seconda filosofia, che si è andata affermando più recentemente, prevede la realizzazione di CPU con un numero ridotto di istruzioni che producono elaborazioni molto semplici tanto da poter essere eseguite in pochissimi impulsi di clock. Al limite un solo ciclo di clock per ogni istruzione. Questa secondo tipo di CPU appartiene alla categoria RISC (Reduced Instruction Set Computer).

A prescindere dalla categoria nella quale si vuol far rientrare una generica CPU, si può affermare che una sua qualunque istruzione appartiene ad uno dei seguenti quattro tipi:

- **trasferimento dati**
- **elaborazione logico-aritmetica di dati**
- **confronto (o test) sui dati**
- **controllo di sequenza di esecuzione**

Le istruzioni di trasferimento dati (in realtà di copiatura di dati), pur nella loro semplicità, sono le istruzioni più largamente utilizzate nei programmi. Appartengono a questa categoria sia le istruzioni che scambiano dati tra CPU e memoria, sia quelle che scambiano dati tra i componenti precedenti e le periferiche (istruzioni di I/O). Le istruzioni di elaborazione aritmetica consentono di effettuare semplici operazioni aritmetiche su uno o due operandi (alle volte solo quelle di somma e differenza), nei processori a basso numero di istruzioni, o operazioni matematiche più complesse, come il calcolo del logaritmo o di un esponenziale, nei processori di tipo CISC, all'interno dei quali è presente una unità aritmetica che elabora numeri in virgola mobile (**FPU**: **F**loating **P**oint **U**nit). Le istruzioni di elaborazione logica agiscono ancora su uno o due operandi secondo le regole dell'algebra di Boole per effettuare le operazioni NOT, AND, OR, EOR, oltre ad operazioni di scorrimento logico. Le istruzioni di confronto o di test di dati segnalano l'esito di un confronto aritmetico tra due dati (senza alterare gli operandi), attivando in modo adeguato i segnalatori aritmetici (flags), dai valori dei quali si può dedurre se i due dati sono uguali oppure se il primo è maggiore o minore del secondo, sia nel caso che gli operandi siano numeri senza segno, sia che siano numeri con segno. Le istruzioni di controllo di sequenza di esecuzione cambiano il contenuto del program counter (PC), consentendo di deviare dal normale ordine sequenziale di esecuzione delle istruzioni e producendo quello che, in senso figurato, è chiamato "salto". In questo modo l'istruzione successiva, a quella di controllo di sequenza, non è più necessariamente quella memorizzata nella locazione consecutiva, ma può essere una qualsiasi istruzione del programma in esecuzione, memorizzata sia in una locazione successiva a quella attuale (salto in avanti), sia in una locazione precedente a quella attuale (salto indietro). Alcune di queste istruzioni, di notevole importanza, effettuano il salto previsto solo se un segnalatore aritmetico, o una particolare combinazione di essi, assume valori predeterminati. In questo modo le porzioni di programma eseguite sono *condizionate* dai risultati ottenuti.

Senza entrare in ulteriori dettagli, a titolo esemplificativo si descriveranno successivamente alcuni esemplari dei suddetti quattro tipi di istruzioni riferiti alla CPU MC68000. La descrizione sarà fatta sinteticamente mostrando il codice sorgente, la traduzione in codice oggetto in uno spaccato della memoria centrale ed una mappa che "visualizza" in modo grafico l'esecuzione. A meno che non sia rilevante per l'esecuzione della istruzione, non sono stati indicati gli indirizzi di memoria ai quali sono memorizzate le singole istruzioni.

• Trasferimento dati

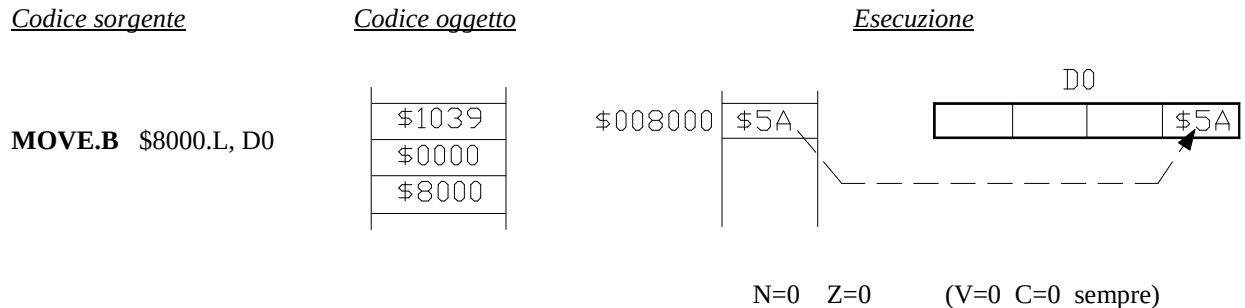


Fig. 2.1.4.1a

Quest'istruzione comanda la copia di un dato, di dimensione byte (**.B**), che si trova in memoria all'indirizzo \$008000 (in questa CPU l'A.B. ha dimensione 24 bit), indicato direttamente in posizione sorgente (**indirizzamento assoluto**), nel byte meno significativo del registro D0. Va ricordato che la CPU MC68000 ha tutti i registri interni di dimensione 32 bit, capaci cioè di contenere un intero long word. La copia del dato, *nel registro dati* D0, viene effettuata ponendolo nel byte meno significativo e lasciando i tre byte rimanenti inalterati. Il simbolo ".L" nel codice sorgente, scritto sul lato destro dell'indirizzo \$8000, è un comando, rivolto direttamente al programma l'Assemblatore, affinché *traduca* l'indirizzo come numero a 32 bit ("L" significa Long word). Questo indirizzo, quindi, compare direttamente nel codice macchina, che occupa tre word consecutivi di memoria. Il primo word è, come di consueto, il codice operativo, il secondo word sono i 16 bit più significativi dell'indirizzo sorgente, il terzo word sono i 16 bit meno significativi dello stesso indirizzo. Le istruzioni della CPU MC68000, quindi, possono essere di lunghezza variabile, ed occupare un solo word (vedi ad esempio programma (2.1.4.1)) o un numero multiplo di word (da due a cinque word).

In fase di esecuzione, il codice operativo (\$1039) posto nel registro istruzioni, comanda alla CPU di raccogliere i due word consecutivi per completare la fase di fetch. Per ogni word raccolto il program counter è incrementato per puntare al word successivo. Ne deriva che l'istruzione successiva a quella attuale inizia dalla locazione consecutiva a quella contenente la parte bassa dell'indirizzo (\$8000). L'esecuzione dell'istruzione, oltre che copiare il byte di memoria nel registro D0, sensibilizza i due segnalatori N-flag e Z-flag e annulla, in ogni caso, V-flag e C-flag, che, per questa operazione non rivestono alcun significato. L'N-flag ricopia il bit più significativo del dato trasferito e lo Z-flag segnala se il dato trasferito è nullo o meno. In questo caso il byte \$5A ha il bit più significativo nullo (\$5A=%01011010), per cui N=0, inoltre il dato è diverso da zero, per cui Z=0.

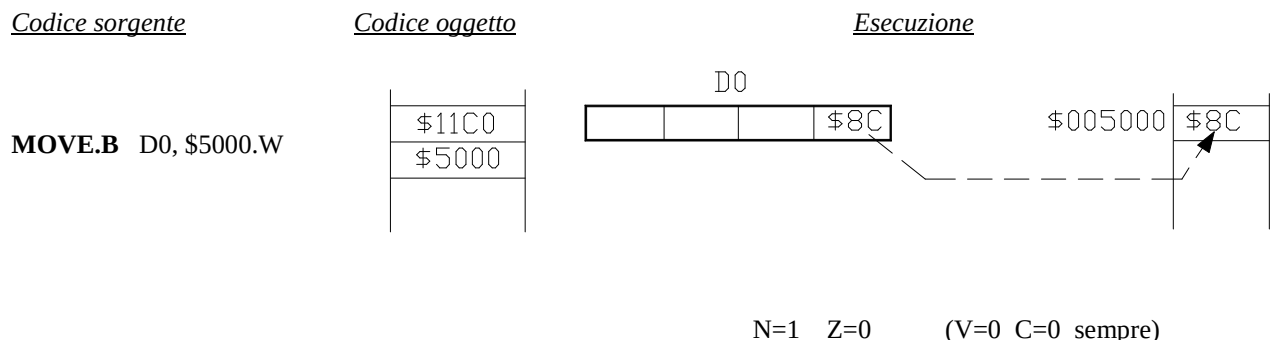


Fig. 2.1.4.1b

Quest'istruzione esegue il trasferimento opposto al precedente in quanto comanda di copiare il byte meno significativo di D0 in memoria. L'indirizzo di destinazione è precisato con il valore numerico \$5000 seguito dal simbolo ".W" (leggi Word). Con questo simbolo il programmatore comanda all'assemblatore di tradurre questo valore

come numero a 16 bit (dimensione word). Il codice oggetto di questa istruzione occupa due word, il primo, come sempre, il codice operativo, il secondo l'indirizzo di destinazione in dimensione 16 bit.

Come è noto gli indirizzi hanno dimensione 24 bit, per cui, in fase di esecuzione la CPU MC68000 prende l'iniziativa di aggiungere gli 8 bit mancanti, ricopiando a sinistra otto volte il bit più significativo. Quindi in fase di esecuzione il valore \$5000, raccolto durante la fase di fetch, porta a generare l'indirizzo \$005000. Quest'operazione non spreca tempo aggiuntivo da parte della CPU, per cui si può affermare che l'uso di questa modalità di indirizzamento (indirizzamento assoluto corto), quando ciò è possibile, risulta conveniente rispetto alla precedente (indirizzamento assoluto lungo), in quanto si hanno istruzioni che occupano meno memoria (due word invece di tre) e sono eseguite più velocemente (la fase di fetch prevede due cicli di memoria invece di tre). Si noti che, in questo caso i segnalatori sono attivati con N=1 e Z=0 dato che il dato copiato è diverso da zero e ha il bit più significativo che è uno (\$8C=%10001100).

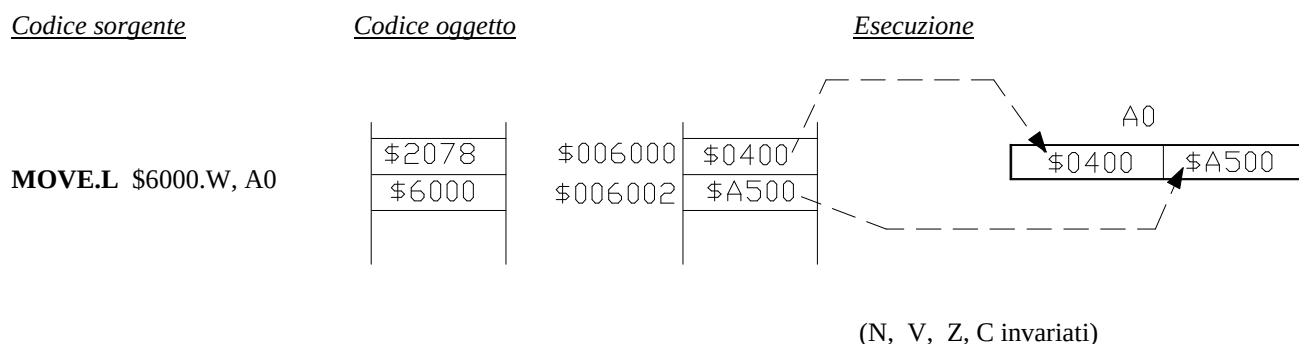


Fig. 2.1.4.1c

In quest'istruzione il trasferimento di un dato, di dimensione long word (.L), avviene nel registro indirizzi A0. Una prima differenza rispetto al registro dati è che non vengono alterati i segnalatori aritmetici. Il dato sorgente è formato da due word consecutivi con il più significativo all'indirizzo \$006000, precisato nell'istruzione, ed il meno significativo all'indirizzo \$006002.

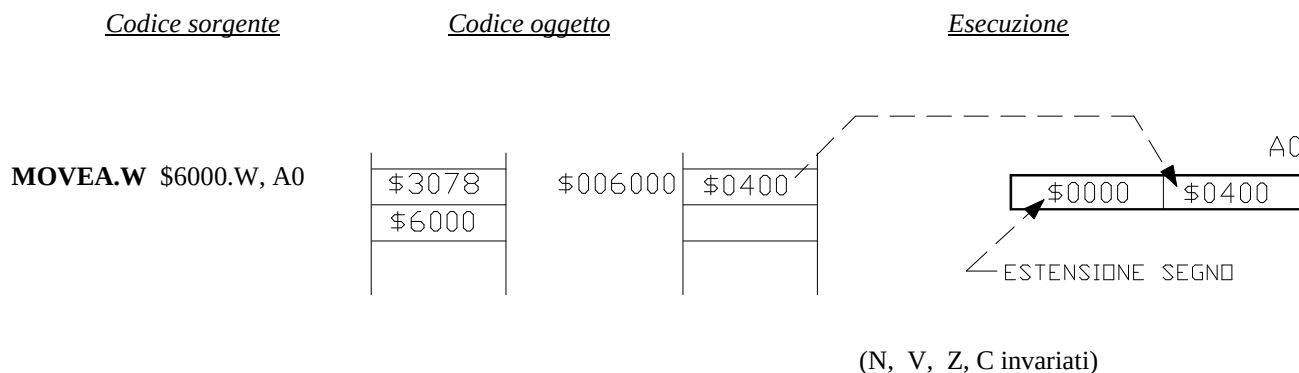


Fig. 2.1.4.1d

Il trasferimento di un dato di dimensione word in un registro indirizzi, a differenza dei registri dati, occupa tutti i 32 bit del registro. Il dato viene posto nel word meno significativo (\$0400) ed il bit più significativo di questo dato, detto bit del segno, viene riprodotto 16 volte sul word più significativo (*estensione del segno*).

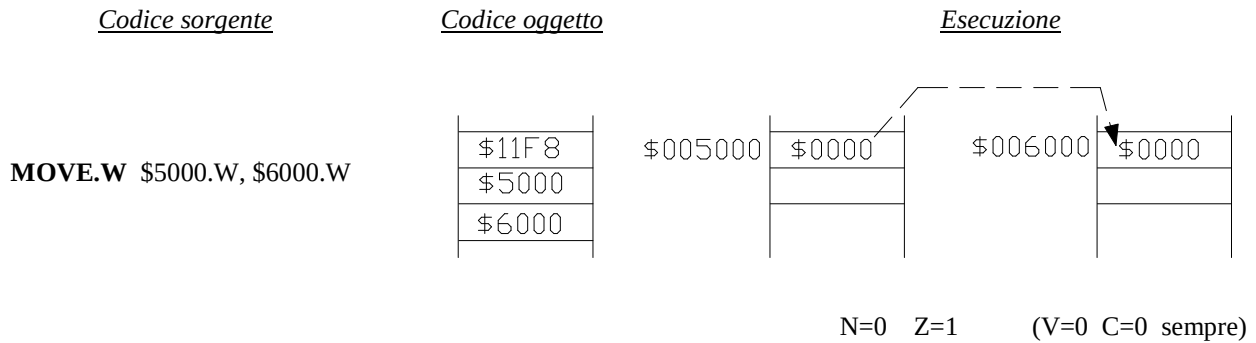


Fig. 2.1.4.1e

La CPU MC68000 consente di trasferire direttamente un dato da memoria a memoria, senza alterare il contenuto di alcun registro interno, sia di tipo dati che di tipo indirizzi. Vengono comunque attivati i due segnalatori N e Z e azzerati i segnalatori V e C, come per quei trasferimenti che coinvolgono un registro dati. Si nota che Z-flag viene posto a uno dato che il dato copiato è zero.

• **Elaborazione dati (aritmetica e logica)**

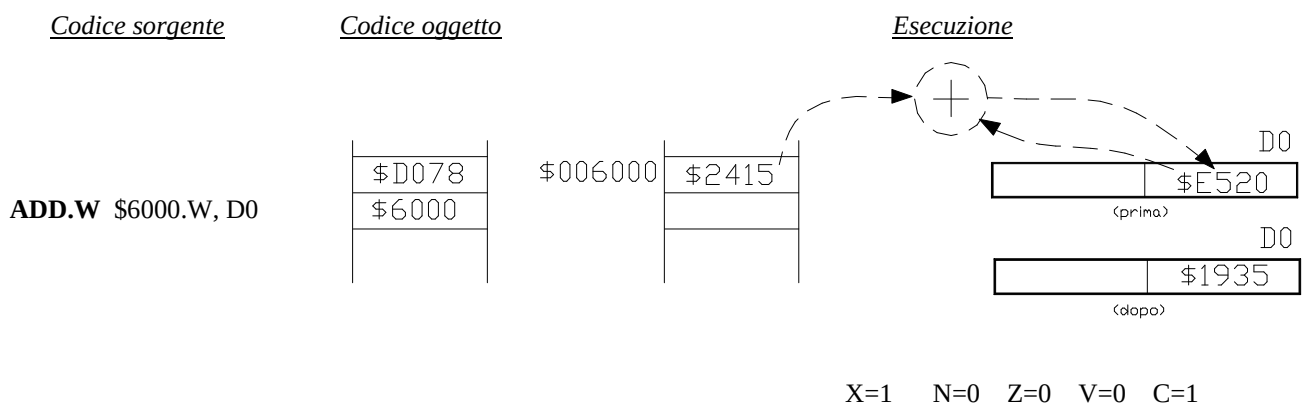


Fig. 2.1.4.2a

L'istruzione di somma agisce su due operandi, uno in memoria, all'indirizzo \$006000 di valore \$2415, e l'altro nel registro D0, di valore \$E520. Il risultato, \$1935, posto nel registro D0 (destinazione), attiva tutti i segnalatori aritmetici. I progettisti di questa CPU hanno ritenuto utile prevedere una "copia" del C-flag, chiamata X-flag, che, in questo caso, ne riproduce esattamente il valore (1). C-flag=1 significa che il risultato, letto come numero senza segno è errato. V-flag=0 significa che lo stesso risultato letto come numero con segno (codificato in complemento a due) è corretto.

<u>Codice sorgente</u>	<u>Codice oggetto</u>	<u>Esecuzione</u>
------------------------	-----------------------	-------------------

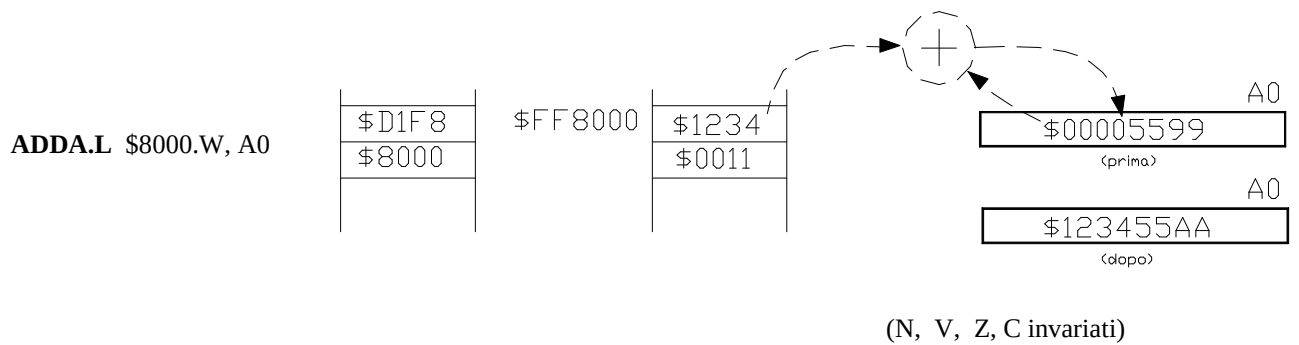


Fig. 2.1.4.2b

La somma in un registro di tipo indirizzo non altera i segnalatori. Si noti la “A” aggiuntiva nel codice mnemonico che indica una somma in un registro indirizzi. La posizione del dato sorgente in memoria è stata specificata con indirizzamento assoluto corto (\$8000.W). In fase di esecuzione, dato che il bit più significativo di questo valore è 1 (\$8000=%10000000000000000), la CPU estende questo bit per otto volte in posizione più significativa generando l’indirizzo \$FF8000.

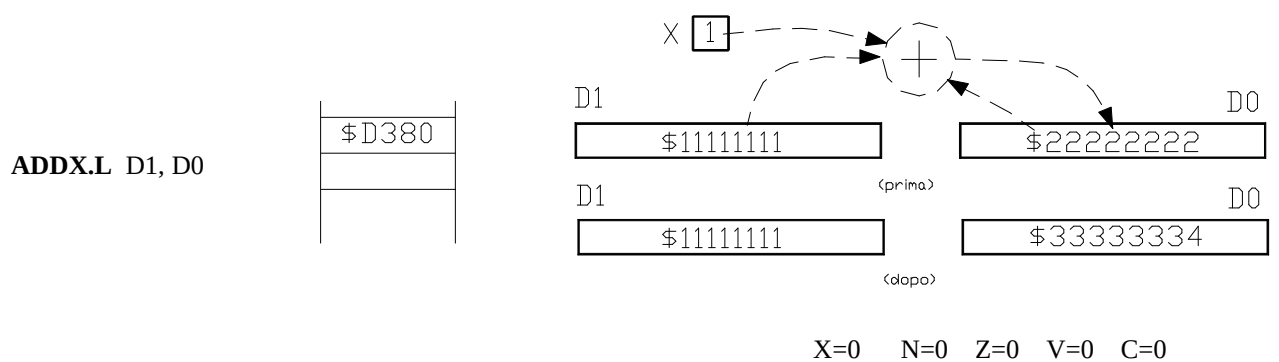
Codice sorgenteCodice oggettoEsecuzione

Fig. 2.1.4.2c

È possibile comandare una somma di tre dati. Due operandi, ad esempio in due registri interni, di dimensione byte, word o long word, con il bit X-flag (copia dello C-flag), riporto di somme precedenti. Ripetendo questa istruzione più volte si possono fare somme a 64, 128 bit ecc. (precisione multipla).

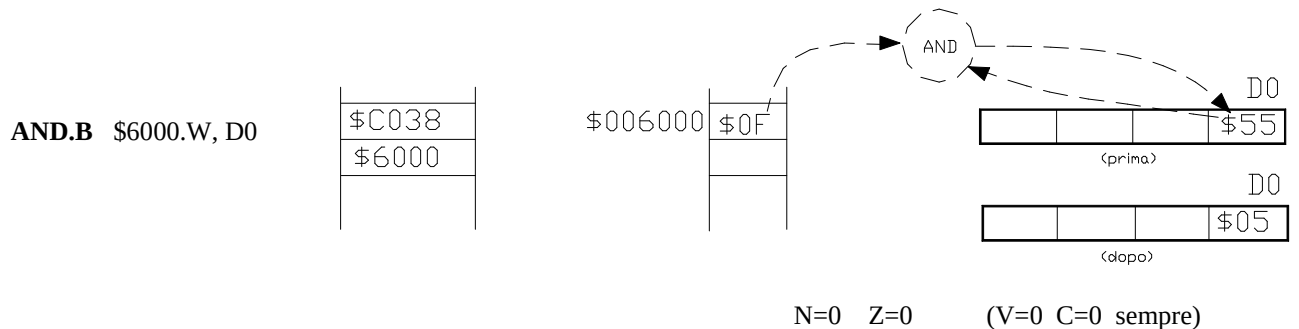
Codice sorgenteCodice oggettoEsecuzione

Fig. 2.1.4.2d

Quest'istruzione comanda l'operazione logica AND, tra bit corrispondenti, dei due byte (**.B**) contenuti, rispettivamente, in memoria (sorgente) e nel registro D0 (destinazione), con risultato in D0. I segnalatori attivati sono N e Z mentre V e C vengono posti a zero.

• Confronto di dati

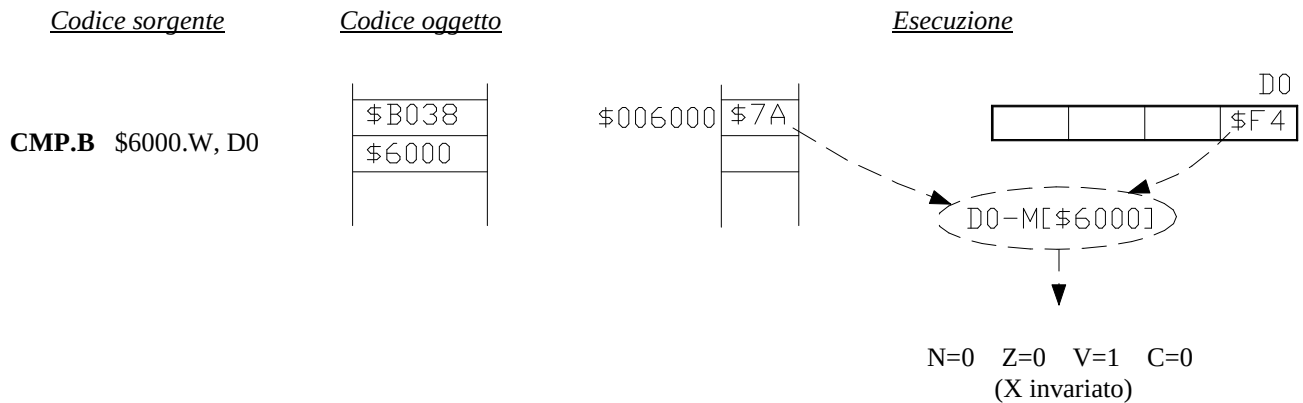


Fig. 2.1.4.3a

L'istruzione di confronto aritmetico (CoMPare: CMP), esemplificata, sottrae il dato di destinazione a quello sorgente e usa il risultato per attivare i quattro segnalatori aritmetici (il segnalatore X rimane invariato). Con i dati rappresentati viene effettuata la sottrazione \$F4-\$7A che dà come risultato \$7A. Questo valore non viene memorizzato in alcun registro, ma utilizzato solo per porre: 1) N=0, dato che il bit più significativo è zero, 2) Z=0, dato che il risultato è diverso da zero, 3) V=1, dato che il segno del risultato è uguale a quello di sorgente (\$7A), 4) C=0, dato che la sottrazione non richiede alcun prestito.

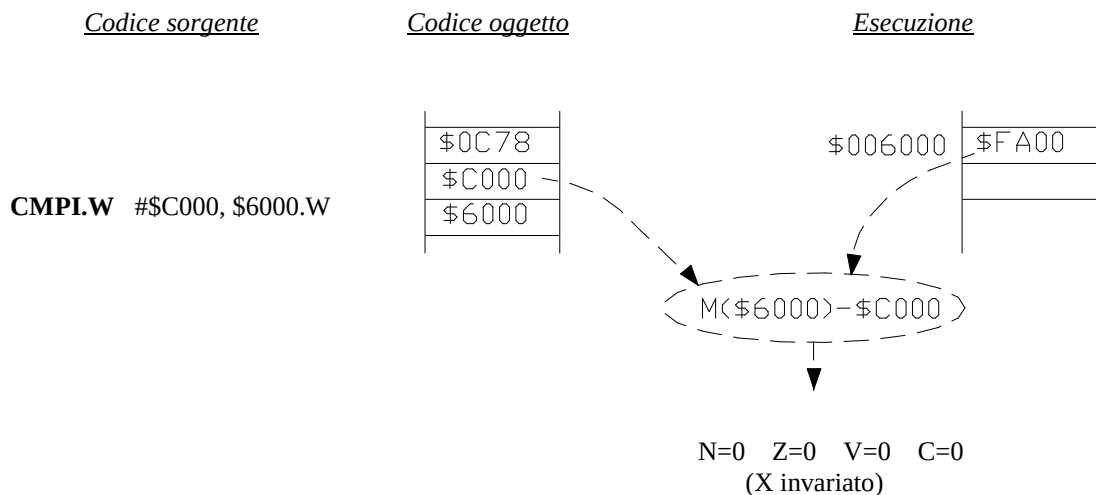


Fig. 2.1.43b

Istruzione che confronta un dato in memoria con un dato che è indicato direttamente nel campo sorgente (**indirizzamento immediato**). Per segnalare che il campo sorgente contiene un dato e *non* un indirizzo, è utilizzato il simbolo “#” (leggi “numero”), come prefisso al valore numerico.

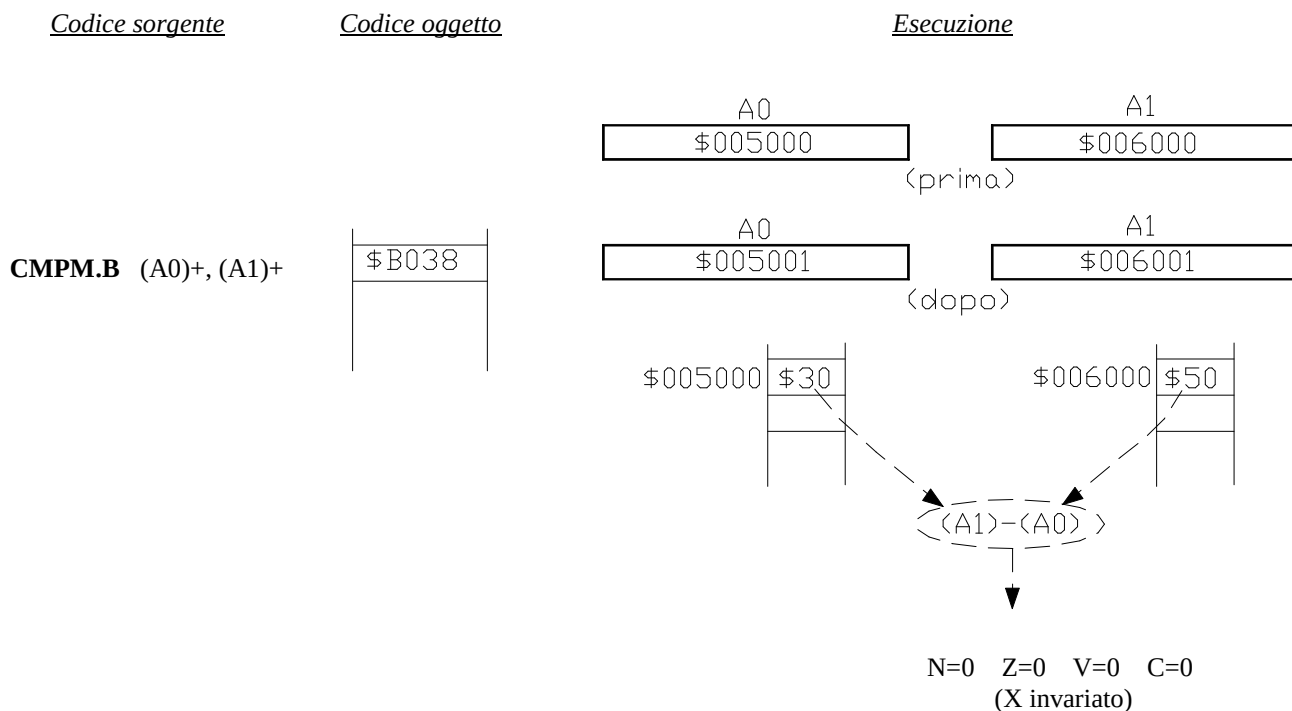


Fig. 2.1.4.3c

È possibile effettuare confronti di due dati posti in memoria utilizzando il contenuto di due registri indirizzi per indirizzare la memoria (**indirizzamento indiretto**). I due dati in memoria, agli indirizzi contenuti nei due registri A1 e A0, vengono sottratti ed con il risultato sono attivati i segnalatori aritmetici. Successivamente i due indirizzi vengono incrementati, in modo da puntare ai due dati contenuti nelle locazioni consecutive (**indirizzamento indiretto con postincremento**) in previsione di un loro confronto. Visto che i dati hanno dimensione byte, l'incremento degli indirizzi è di un'unità. Con dati di dimensione word (**CMPM.W**) o long word (**CMPM.L**) gli incrementi sarebbero stati, rispettivamente, di due e di quattro unità, in quanto i relativi dati occuperebbero due e quattro byte consecutivi di memoria.

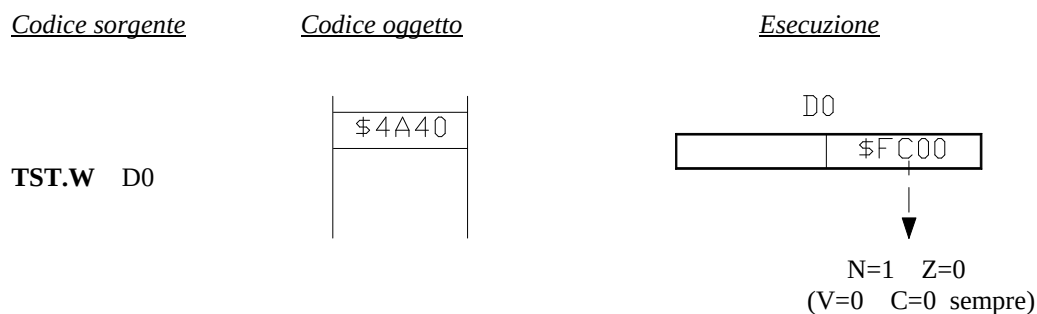


Fig. 2.1.4.3d

L'istruzione di "Test" su un solo operando, contenuto nel registro D0 in questo esempio, attiva i due segnalatori N e Z (V e C posti a zero in ogni caso), per poter stabilire se il dato è zero o diverso da zero e, in questo caso, stabilire se è positivo (N=0) o negativo (N=1).