

Appunti sul processore Motorola MC68000

**Set di istruzioni,
modi di indirizzamento,
un linguaggio assembly**

A cura del prof. Bruno Fadini

2006

**© Università degli Studi di Napoli Federico II
Dipartimento di Informatica e Sistemistica**

Revisione 19/4/2010

Indice

Architettura, linguaggio macchina e assembler 68000	4
Premessa	4
1. Architettura della sezione registri	4
2. Rappresentazione dei dati	7
3. Struttura delle istruzioni	7
4. Modi di indirizzamento	9
5. Linguaggio assemblativo	10
5.1 Simboli ed espressioni	15
6. Istruzioni di trasferimento dati	17
6.1 Move in generale	17
6.2 Clear e Set	17
6.3 Move e Load verso registri di indirizzamento	18
6.4 Move per registri speciali	18
6.5 Operazioni su stack	20
6.6 Operazioni di scambio	20
6.7 Istruzioni di move assenti nel 68000	20
7. Istruzioni aritmetiche e logiche	21
7.1 Aritmetica binaria	21
7.1.1 Operazioni unarie	21
7.1.2 Addizione e sottrazione	21
7.1.3 Estensione del segno	25
7.1.4 Moltiplicazione e divisione	26
7.2 Aritmetica decimale	29
7.3 Istruzioni d'indirizzamento e controllo	29
7.4 Istruzioni logiche	30
7.5 Istruzioni di Shift	30
8. Posizionamento dei flag - Istruzioni di comparazione e test	31
8.1 Comparazione aritmetica	33
8.2 Test	34
8.2.1 Test su un bit	34
8.2.2 Test su byte, word o longword	34
9. Istruzioni di salto	36
9.1 Salti incondizionati	36
9.2 Salti condizionati	37
9.3 Salti a sottoprogrammi	40
9.4 Ritorno da sottoprogrammi	40
10. Altre istruzioni	42
10.1 Istruzione DBcc	42
10.2 Trasferimenti a blocchi	43
10.3 Stop e Nop	43

10.4 Istruzioni LINK e UNLK	43
10.4.1 Esempio d'uso di LINK, UNLK: chiamata di sottoprogrammi...	45
10.5 Istruzioni di interruzione software (trap).....	47
10.6 Istruzioni di input/output.....	48
Le direttive di assemblaggio di ASM 68000.....	50
Premessa.....	50
Program Location Counter e direttiva ORG	50
Dichiarazione di dati: le direttive DS e DC	51
2.1 Direttiva DS	51
2.2 Direttiva DC.....	52
La chiusura statica del programma: direttiva END	53
Uso di file esterni per l'assemblaggio: direttiva INCLUDE	53
Altre direttive	54
5.1 Direttiva EQU	54
5.2 Direttiva SET	54
5.3 Direttiva REG	54

Architettura, linguaggio macchina e assembler 68000

Premessa

Il processore MC68000 della Motorola è stato introdotto nel 1979. Si tratta di un'architettura a registri generali, molto interessante per la sua regolarità, pur non essendo completamente ortogonale. Molte soluzioni architetturali adottate da questo processore (riguardo ai modi d'indirizzamento, il tipo d'istruzioni, l'I/O) sono derivate dal PDP-11, architettura innovativa e di riferimento degli anni '70. Il processore 68000 è il capostipite di una famiglia di microprocessori prodotti dalla Motorola negli anni '80 e '90. Questi processori, inizialmente utilizzati in sistemi di calcolo di tipo workstation, come i "personal computer" della serie Apple Macintosh, trovano tuttora impiego in sistemi di controllo numerico programmabili.

1. Architettura della sezione registri

Il 68000 costituisce una architettura di transizione fra quelle ad 8 bit dei primi microprocessori e quelle a 32 bit dei successori: la memoria è indirizzata a byte, si intende per "word" una parola di 16 bit, pur essendo, in effetti, i registri a 32 bit (doppia word o *longword*).

La tabella 1.1 elenca i registri disponibili nella omonima sezione della architettura 68000. La figura 1.2 fornisce una immagine di tutti i registri del modello di programmazione.

Tabella 1.1 Registri di macchina del processore MC68000

Classi di registri	Denominazione	Parallelismo
<i>Program Counter:</i>	<i>PC</i>	<i>32 bit</i>
<i>Accumulatori:</i>	<i>D0-D7</i>	<i>32 bit</i>
<i>Indirizzamento:</i>	<i>A0-A7</i>	<i>32 bit</i>
<i>Stack pointer:</i>	<i>SP=A7</i>	<i>32 bit</i>
<i>Stato:</i>	<i>SR</i>	<i>16 bit</i>
<i>Indicatori (Flag):</i>	<i>CCR</i>	<i>8 bit</i>

Il 68000 è dunque una macchina con 16 registri generali di 32 bit suddivisi in due banchi:

- 8 accumulatori detti "Data Register", D0-D7, che operano su dati di tipo byte (bit 0-7 del registro), word (bit 0-15), longword (0-31);
- 8 registri d'indirizzamento, detti "Address Register", A0-A7 (A7 svolge anche la funzione di Stack Pointer, vedi in seguito).

Si noti che sia i registri D sia quelli A possono avere funzioni di registri-indice.

La macchina dispone di un Program Counter di 32 bit, atto ad indirizzare $2^{32}=4$ Giga locazioni di memoria. In effetti, nelle prime versioni del processore si usavano solo 24 dei 32 bit.

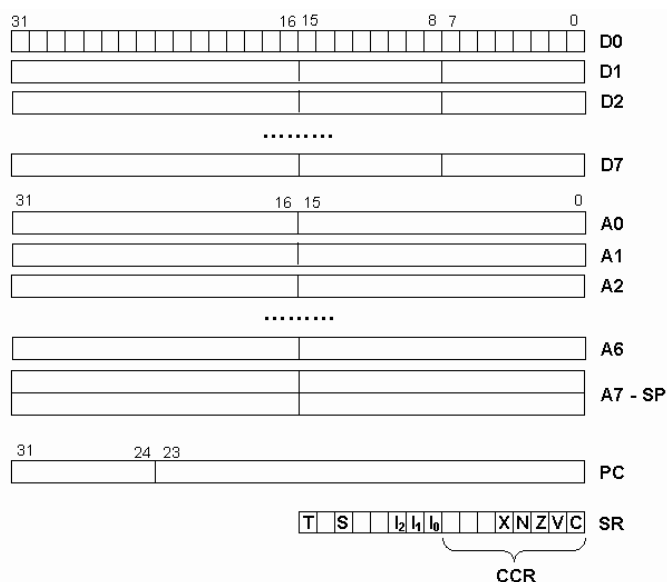


Figura 1.2 Modello di programmazione del processore MC68000

Il registro di stato della macchina è costituito da due byte (16 bit); quello di minor peso è anche detto *CCR (Condition Code Register)* e contiene gli indicatori o flag (cfr. § F2-III-I.12.6):

- **C= Carry:** segnala un riporto/prestito uscente dal bit di maggior peso in operazioni aritmetiche;
- **V= oVerlow:** overflow in rappresentazione in complementi;
- **Z= Zero:** dato nullo;
- **N= Negative:** dato negativo;
- **X= eXtend:** copia C in alcune istruzioni aritmetiche ed è usato per l'estensione ad aritmetica in precisione multipla.

Si noti che i significati attribuiti ai flag nella tabella di cui sopra sono quelli prevalenti, ma ogni istruzione che agisca su di essi ne attribuisce di propri. In particolare, il flag C rappresenta il riporto per le istruzioni aritmetiche e può rappresentare altri eventi per le altre; il flag X copia C per le istruzioni aritmetiche, mentre resta inalterato per le altre: conserva dunque memoria del riporto scaturito dall'ultima istruzione aritmetica eseguita. X rappresenta dunque la copia di C che il processore usa in alcune istruzioni aritmetiche, mentre C, così come V, Z, N sono bit da interrogare nelle istruzioni di salto condizionato.

Il byte alto di SR contiene bit che afferiscono alla gestione della macchina e precisamente:

- S= *Stato*: distingue lo stato utente da quello supervisore (cfr. § F2-III-I.12.6)
- T= *Trace*: individua uno speciale stato di “trace”, nel quale la macchina sviluppa un programma evidenziandone l'avanzamento passo per passo;
- I₂ I₁ I₀= *maschera delle interruzioni*: il sistema delle interruzioni è molto elementare e viene ampliato attraverso apposite interfacce.

Si noti infine che esistono due distinti registri A7 che hanno funzioni di stack pointer (e si chiamano dunque anche SP), uno per ciascuno stato della macchina. Nello stato utente la macchina fa automaticamente riferimento allo stack-utente USP, in quello supervisore allo stack-supervisore SSP.

I registri speciali del 68000 sono dunque CCR, SR ed SP, tutti singolarmente indirizzabili da alcune istruzioni. Nel seguito vengono usati i simboli di cui alla tabella 1.3 per indicare gli operandi delle istruzioni

Tabella 1.3 – Simboli adoperati per la semantica delle istruzioni

Simbolo	significato
D	D0-D7: un qualsiasi registro-dati
A	A0-A7: un qualsiasi registro di indirizzamento
R	un qualsiasi registro A oppure D
im	Operando immediato
M	Operando memoria
G	Operando “generale”: a seconda dei modi, è un registro, un operando-memoria o un operando-immediato
CCR	registro dei flag
SR	registro di stato
SP	stack pointer

Si useranno notazioni analoghe a quelle di cui al § F2-III-III.5, con i simboli allineati a quelli tipici dei manuali del 68000 (si usa solo R in luogo di X per motivi di chiarezza rispetto al testo).

2. Rappresentazione dei dati

I dati numerici sono rappresentati in aritmetica binaria e tipo intero, nelle due forme di numeri naturali ("unsigned", senza segno) e di numeri relativi ("signed"): in questo secondo caso si adopera la rappresentazione in complementi alla base (complementi a due).

In particolare, si hanno i seguenti tipi di dato, tutti nelle due forme di signed oppure unsigned:

- intero binario a 8 bit (byte),
- intero binario a 16 bit (word)
- intero binario a 32 bit (long)

In alcune istruzioni sono previsti valori immediati allocati in un campo dell'istruzione (ad esempio, negli ultimi 8 bit); anche in tali casi i numeri in gioco sono talora rappresentati in complementi a 2.

Oltre alla aritmetica binaria, esiste un'aritmetica decimale packed (su un singolo byte).

I caratteri sono rappresentati in ASCII su un byte.

3. Struttura delle istruzioni

Il 68000 possiede istruzioni di lunghezza variabile da uno a cinque parole: di queste, la prima fornisce codice operativo, modo d'indirizzamento ed implicitamente la lunghezza della stessa istruzione. Le parole successive contengono eventualmente, in base al modo di indirizzamento, l'operando immediato e/o gli indirizzi degli operandi. Esistono molti formati distinti, che si possono riassumere nello schema di fig. 3.1.

Le istruzioni operano sul tipo byte (8 bit), word (16) o long (32 bit), così come specificato dal campo *tipo* oppure implicitamente dal codice operativo.

È da notare che lo sforzo del progettista del 68000 è stato quello di compattare in un'unica word (16 bit), detta *opcode word*, il codice operativo e tutti gli indicatori essenziali (sono esclusi solo gli operandi immediati ed assoluti, che seguono nelle word successive). Ciò ha richiesto un intenso sfruttamento dei 16 bit disponibili, a scapito talora della linearità e regolarità dei campi.

Si ha dunque che per codici diversi, i medesimi campi hanno diverso significato (oltre quanto espresso in figura) ed inoltre, spesso i campi o parte di questi sono impiegati per esprimere informazioni specifiche della particolare istruzione. Ad esempio, gli stessi bit del codice operativo spesso invadono campi diversi da quello specificamente ad esso dedicato.

Con riferimento alla figura, si noti che in alcune istruzioni due o più campi sono accorpati in un campo unico di dimensioni maggiori:

- dalle classi base b e c sono derivate le classi b1 e c1, accorpendo *R* ed *r* in un unico campo *cc*; ciò avviene per istruzioni che non richiedono l'operando *R*, ma un codice di condizione *cc* che lo sostituisce;
- il formato della classe c2 è derivato da quello della classe c, sostituendo i campi *rr* ed *R1* con un campo *displacement*;
- le istruzioni di salto (cfr. §9) adottano entrambe le varianti c1 e c2 della classe c;
- la classe d1 è derivata dalla classe d estendendo il campo del codice operativo di due bit, recuperati dal campo *tipo* della classe base d.

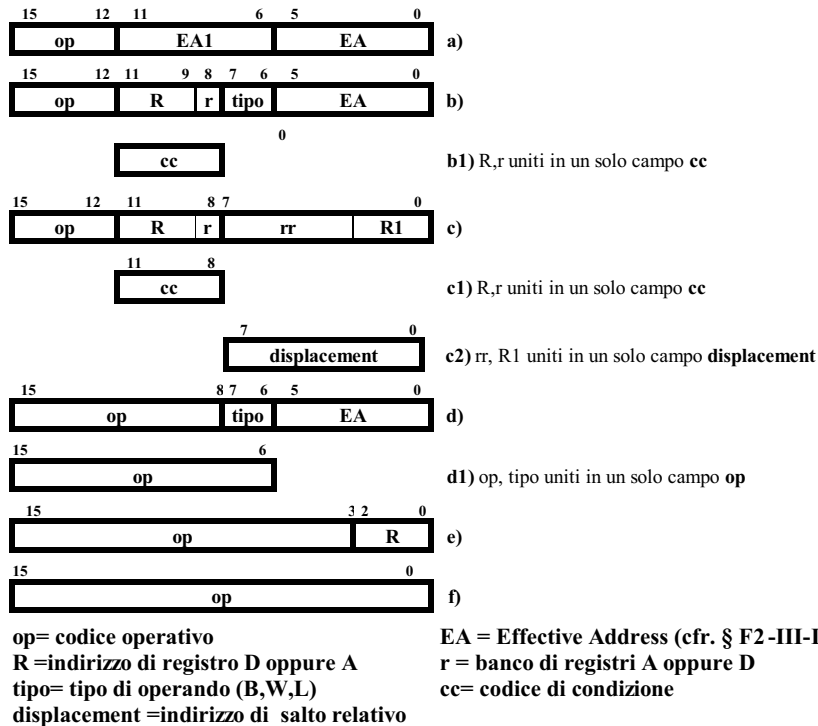


Figura 3.1 Formato delle istruzioni del 68000:

- a) Istruzioni a 2 operandi G; b) 2 operandi, uno G e l'altro R; c) a 2 operandi R;
d) ad 1 operando G; e) ad 1 operando-R; f) a 0 operandi oppure implicito

4. Modi di indirizzamento

I modi di indirizzamento del 68000 sono illustrati nella tabella 4.1, ove per ciascun modo sono riportati la tecnica usata a livello macchina per individuare il modo, la notazione assembler e il riferimento al capitolo del testo di Fondamenti II nel quale il modo stesso è illustrato in teoria.

Tabella 4.1: Modi di indirizzamento del 68000

	EA		Assembler		Rif § cap II
	m	reg	Notazione	Esempio	
Diretto di registro dati	0	num	Dn	D3	5.3.1
Diretto di registro indirizzi	1	num	An	A4	5.3.1
Indiretto attraverso reg. indirizzi	2	num	(An)	(A5)	5.4.2
Indiretto con postincremento	3	num	(An)+	(A1)+	6.3
Indiretto con predecremento	4	num	-(An)	-(A3)	6.3
Indiretto con displacement (detto anche <i>based</i>)	5	num	offset16(An)	\$54(A2)	6.1.2
Indiretto con displacement e indice	6	num	offset8(An, Rn) offset8(An, Rn.L)	\$30(A3,A6) \$50(A3,A6.L)	6.2.1
Diretto di memoria (assoluto)	7	0	addr16	\$7124	5.3.2
	7	1	addr32	\$247124	
Relativo a PC	7	2	offset16(PC)	\$8(PC)	-
Relativo a PC con displ. e indice	7	3	offset8(PC, Rn)	\$4E(PC,A3)	-
Immediato	7	4	im	\$107E	5.2

In particolare, le prime due colonne indicano la tecnica usata a livello macchina: il modo di indirizzamento è indicato esplicitamente in quelle istruzioni che prevedono il campo EA (classe a, b, d di fig.3.1) ed è esplicitato attraverso due sottocampi di 3 bit l'uno:

- m (codificato in ottale in figura) esprime un codice di modo:
- reg che individua il registro indirizzato per $m < 7$ (nei tre bit se ne indica allora l'indirizzo), specifica ulteriormente il modo per $m=7$.

Si noti l'uso del displacement che qui viene anche detto offset. Esso può essere espresso attraverso un numero relativo ad 8 bit (offset8) oppure a 16 (offset16), quantità che in ogni caso è immessa nella parola che segue l'istruzione (si tratta

infatti di un immediato). In particolare, nel modo “indiretto con displacement” l'offset è di 16 bit ed occupa per intero la parola seguente; nel caso che vi sia anche l'indice, l'offset è limitato ad 8 bit, ma nella parola che segue è anche immesso l'indice R e l'indicatore r che lo riguarda.

La tecnica dell'indirizzamento relativo ad un indirizzo contenuto in un registro An è estesa nel 68000, per i modi "relativo a PC", sostituendo PC ad An; questa sostituzione è implicita in linguaggio macchina, esplicita in assembler.

L'indirizzamento diretto a memoria avviene usando un indirizzo assoluto su 16 bit (address16) oppure su 32: i due casi sono distinti in linguaggio macchina da un bit (num = 0/1) in quanto nei due casi l'indirizzo richiederà l'accesso a 1 oppure 2 parole successive. In assembler, non è necessaria la distinzione: l'assemblatore traduce l'istruzione nella prima o nella seconda forma a seconda della lunghezza effettiva dell'indirizzo espresso.

5. Linguaggio assemblativo

La programmazione di un processore non è mai effettuata direttamente nel linguaggio macchina. Si ricorda che a ciascun linguaggio macchina è associato un linguaggio simbolico elementare, detto *linguaggio assemblativo o assembly*, che costituisce il più elementare fra i linguaggi simbolici (cfr. § F2-IV-VII.1).

In questo paragrafo si illustrano le caratteristiche essenziali del linguaggio assemblativo (per brevità, indicato come ASM68K), definito dalla Motorola per l'intera famiglia di processori MC680x0, di cui il 68000 è il capostipite. Questo linguaggio è adottato dai sistemi di sviluppo prodotti sia dalla Motorola che da terze parti.

Pur nella sua specificità, il linguaggio segue le linee generali di tutti i linguaggi assemblativi e pertanto esso va studiato a valle della acquisizione di concetti e terminologie di carattere generale. Per questi si rinvia ai libri di testo.

Solo a scopo riepilogativo, si ricorda in questa sede che:

- la traduzione da linguaggio assemblativo a linguaggio macchina è effettuata da un apposito programma traduttore, detto *assemblatore* (si rimanda a § F2-IV-VII-1 per un opportuno approfondimento sulle tecniche adoperate dagli assemblatori);
- in un linguaggio assemblativo le istruzioni del linguaggio macchina non perdono la loro individualità, ma ai codici operativi numerici sono sostituiti codici simbolici letterali, con caratteristiche mnemoniche e gli operandi sono indicati con nomi simbolici;

- alle istruzioni corrispondenti al linguaggio macchina, l'assembler aggiunge apposite *dichiarazioni* (dette anche *direttive di assemblaggio*), espresse attraverso appositi pseudocodici in luogo dei codici operativi;
- le direttive di assemblaggio consentono, tra le altre cose, di introdurre nomi simbolici, i quali vengono associati dal programma assembler ad indirizzi di macchina.

Alle nozioni qui riepilogate, ne aggiungiamo in questa sede alcune altre, valide in generale per qualsiasi assembly.

- Un programma in linguaggio assemblativo è strutturato come un file di testo, in cui ciascuna linea è una istruzione oppure una direttiva di assemblaggio (*pseudo-istruzione*). Questa soluzione è preferita in genere dagli assembler in luogo di soluzioni tipiche di linguaggi ad alto livello ove il testo è una stringa di caratteri ed una istruzione si distingue da un'altra mediante appositi separatori (ad esempio";").
- Così come nei linguaggi ad alto livello, è importante potere inserire commenti in un programma; questi in generale possono essere aggiunti in coda ad una istruzione, sulla medesima linea, oppure in apposite *linee di commento*.
- Così come nei linguaggi ad alto livello, ad una istruzione o pseudo-istruzione può essere associata una *etichetta*, che è il nome simbolico assegnatole; essa viene associata dall'assembler all'indirizzo della locazione di memoria in cui l'istruzione verrà posta. In particolare, l'etichetta serve per assegnare un nome simbolico ad una variabile introdotta mediante apposita dichiarazione oppure ad una istruzione della sequenza del programma.
- Un assembler (come del resto un qualsiasi programma applicativo) è "*case-insensitive*" se non distingue un carattere maiuscolo dal corrispondente minuscolo, *case-sensitive* altrimenti.

Nel linguaggio ASM68K istruzioni e direttive di assemblaggio sono strutturate in campi secondo il seguente formato¹:

[Etichetta] CodiceOperativo [Operando1 [[,Operando2]...[,OperandoN]]] [Comme n- to]

I campi sono separati da caratteri separatori (spazi bianchi o caratteri di tabulazione).

L'**etichetta** è una sequenza di caratteri (priva di separatori, e di solito limitata in lunghezza ad 8 caratteri) che costituisce il nome dell'istruzione o della variabile

¹ Si ricorda che i campi fra parentesi quadra sono opzionali.

introdotta mediante la pseudo-istruzione. Essa è ovviamente opzionale e, laddove manchi, è necessario iniziare il rigo con almeno un carattere separatore prima del codice operativo, affinché ASM68K possa distinguere il codice operativo come tale e non confonderlo con una etichetta.

Il **codice operativo** può essere o uno dei codici mnemonici associati alle istruzioni in linguaggio macchina del processore (elencati in Tabella 5.1), o uno pseudo-codice di una direttiva di assemblaggio (elencati in Tabella 5.2).

Il tipo di dato fondamentale che tratta ASM68K è la word (16 bit), ma può trattare anche dati di tipo byte (8 bit) oppure “longword” (32 bit). A questo scopo alcuni codici o pseudo-codici operativi consentono di specificare il tipo dell’operando mediante la giustapposizione di un suffisso. Sono usati i suffissi .B, .W, o .L, che indicano rispettivamente il tipo Byte, Word e Longword rispettivamente (la mancanza di suffisso indica il tipo Word).

Il **campo operandi** contiene gli operandi dell’istruzione, che possono essere zero, uno, due o più. Più di due operandi possono comparire solo in particolari pseudo-istruzioni contenenti, ad esempio, liste di valori da assegnare a locazioni di memoria consecutive.

Il campo **commenti** è rappresentato dalla parte dell’istruzione che segue il campo operandi. Non è necessario l’uso di un carattere speciale per delimitare l’inizio del campo commenti, in quanto ASM68K controlla la sintassi della istruzione e “comprende” che qualsiasi carattere che la segua nel rigo appartiene ad un commento.

Infine, il linguaggio consente di inserire nel testo del codice sorgente apposite linee di commento (non interpretate dall’assemblatore), mediante l’uso del carattere iniziale “*”.

Nel seguito si descriveranno le istruzioni che compongono il repertorio di codici operativi del processore Motorola 68000. In tabella 5.1 è riportato l’intero set dei codici operativi del processore, insieme all’indicazione della tabella in cui l’istruzione è descritta. In tabella 5.2, invece, sono elencate le direttive di assemblaggio riconosciute dall’assemblatore ASM68K. Si rimanda al prossimo capitolo per una loro descrizione.

Tabella 5.2: Direttive di assemblaggio dell’assemblatore ASM68K

opcode	Direttiva
ORG	<i>Set Origin</i>
EQU	<i>Equate Symbol</i>
SET	<i>Set Symbol</i>
REG	<i>Register Range</i>
DC	<i>Define Constant</i>
DS	<i>Define Storage</i>
END	<i>End assembly</i>
INCLUDE	<i>Include source</i>

Infine, in figura 5.1 si riporta come esempio il codice sorgente in assembler ASM68K di un semplice programma che effettua l'ordinamento di un vettore di interi

Tabella 5.1: Set di istruzioni completo del processore 68000

opcode	Istruzione	Rif. Tab.	opcode	Istruzione	Rif. Tab.
ABCD	<i>Add BCD with extend</i>	7.1	MOVE	<i>To SR</i>	6.1
ADD	<i>ADD binary</i>	7.1	MOVE	<i>From SR</i>	6.1
ADDA	<i>ADD binary to An</i>	7.1	MOVE	<i>USP to/from Address Register</i>	6.1
ADDI	<i>ADD Immediate</i>	7.1	MOVEA	<i>MOVE Address</i>	6.1
ADDQ	<i>ADD 3-bit immediate</i>	7.1	MOVEM	<i>MOVE Multiple</i>	10.2
ADDX	<i>ADD eXtended</i>	7.1	MOVEP	<i>MOVE Peripheral</i>	10.6
AND	<i>Bit-wise AND</i>	7.2	MOVEQ	<i>MOVE 8-bit immediate</i>	6.1
ANDI	<i>Bit-wise AND with Immediate</i>	7.2	MULS	<i>MULTiply Signed</i>	7.1
ASL	<i>Arithmetic Shift Left</i>	7.2	MULU	<i>MULTiply Unsigned</i>	7.1
ASR	<i>Arithmetic Shift Right</i>	7.2	NBCD	<i>Negate BCD</i>	7.1
BCHG	<i>Test a Bit and ChanGe</i>	8.1	NEG	<i>NEGate</i>	7.1
BCLR	<i>Test a Bit and CleaR</i>	8.1	NEGX	<i>NEGate with eXtend</i>	7.1
Bcc	<i>Conditional Branch</i>	9.2	NOP	<i>No Operation</i>	10.3
BRA	<i>BRanch Always</i>	9.1	NOT	<i>Form one's complement</i>	7.2
BSET	<i>Test a Bit and SET</i>	8.1	OR	<i>Bit-wise OR</i>	7.2
BSR	<i>Branch to SubRoutine</i>	9.3	ORI	<i>Bit-wise OR with Immediate</i>	7.2
BTST	<i>Bit TeST</i>	8.1	PEA	<i>Push Effective Address</i>	6.1
CHK	<i>CHeCK Dn Against Bounds</i>	10.5	RESET	<i>RESET all external devices</i>	10.6
CLR	<i>CLeaR</i>	6.1	ROL	<i>ROtate Left</i>	7.2
CMP	<i>CoMPare</i>	8.1	ROR	<i>ROtate Right</i>	7.2
CMPA	<i>CoMPare Address</i>	8.1	ROXL	<i>ROtate Left with eXtend</i>	7.2
CMPI	<i>CoMPare Immediate</i>	8.1	ROXR	<i>ROtate Right with eXtend</i>	7.2
CMPI	<i>CoMPare Memory</i>	8.1	RTE	<i>ReTurn from Exception</i>	10.5
DBcc	<i>Looping Instruction</i>	10.1	RTR	<i>ReTurn and Restore</i>	9.4
DIVS	<i>DIVide Signed</i>	7.1	RTS	<i>ReTurn from Subroutine</i>	9.4
DIVU	<i>DIVide Unsigned</i>	7.1	SBCD	<i>Subtract BCD with eXtend</i>	7.1
EOR	<i>Exclusive OR</i>	7.2	Scc	<i>Set to -1 if True, 0 if False</i>	6.1
EORI	<i>Exclusive OR Immediate</i>	7.2	STOP	<i>Enable & wait for interrupts</i>	10.3
EXG	<i>Exchange any two registers</i>	6.1	SUB	<i>SUBtract binary</i>	7.1
EXT	<i>Sign EXTend</i>	7.1	SUBA	<i>SUBtract binary from An</i>	7.1
ILLEGAL	<i>Illegal instruction</i>	10.5	SUBI	<i>SUBtract Immediate</i>	7.1
JMP	<i>JuMP to Effective Address</i>	9.1	SUBQ	<i>SUBtract 3-bit immediate</i>	7.1
JSR	<i>Jump to SubRoutine</i>	9.3	SUBX	<i>SUBtract eXtended</i>	7.1
LEA	<i>Load Effective Address</i>	6.1	SWAP	<i>SWAP words of Dn</i>	6.1
LINK	<i>Allocate Stack Frame</i>	10.4	TAS	<i>Test & Set MSB & Set N/Z-bits</i>	8.1
LSL	<i>Logical Shift Left</i>	7.2	TRAP	<i>Execute TRAP Exception</i>	10.5
LSR	<i>Logical Shift Right</i>	7.2	TRAPV	<i>TRAPV Exception if V-bit Set</i>	10.5
MOVE	<i>Move</i>	6.1	TST	<i>TeST for negative or zero</i>	8.1
MOVE	<i>To CCR</i>	6.1	UNLK	<i>Deallocate Stack Frame</i>	10.4

5.1 Simboli ed espressioni

Un simbolo è una sequenza di caratteri che inizia con una lettera od il punto ('.'). I rimanenti caratteri possono essere lettere o i caratteri speciali '\$', '.', ' ' e '_'. Gli assembler di solito considerano come significativi solo i primi 8 caratteri di un simbolo. In un programma in linguaggio assembler, i simboli possono apparire nel campo etichetta e nel campo operandi, e possono rappresentare o valori costanti (ad essi associati mediante un'apposita direttiva) oppure operandi di tipo memoria (cioè indirizzi di memoria).

Un'espressione può comparire in un programma assembler ovunque sia richiesto un numero. Essa consiste di uno o più operandi (numeri o simboli) combinati mediante operatori unari o binari. Il valore di un'espressione è valutato dall'assemblatore al tempo dell'assemblaggio come numero a 32 bit e sostituito all'espressione nel codice macchina generato. Le costanti numeriche sono, per default, interpretate come numeri decimali. È comunque possibile utilizzare la rappresentazione esadecimale (anteponendo al numero il simbolo '\$'), quella ottale (anteponendo al numero il simbolo '@'), e quella binaria (anteponendo al numero il simbolo '%').

16 Architettura, linguaggio macchina e assembler 68000

```

* BSORT.ASM - Un semplice programma di ordinamento bubblesort
* Ordina in senso crescente 5 numeri alle locazioni $1000-
$1004
* Registri usati:
* A0: indirizzo del primo elemento dell'array
* D0: variabile di appoggio
* D1: idem
* D2 flag di scambio, (1 = almeno uno scambio effettuato)
* D3: contatore di iterazioni

N      EQU      5              array size
      ORG      $400
* Qui comincia il programma
PROG   LEA      ARRAY,A0      carica in A0 indirizzo array
      CLR      D2              azzera il flag di scambio
      MOVEQ    #N-1,D3        inizializza contatore iterazioni
LOOP1  MOVE.B   (A0),D0        carica in D0 il 1° el. della coppia
      MOVE.B   1(A0),D1        carica in D1 il 2° el. della coppia
      CMP.B    D0,D1           confronto
      BGE.S    NOSWAP         se il 2° è minore del 1°, scambiali
SWAP   MOVE.B   D0,1(A0)       sostituisci il 2° con il 1°
      MOVE.B   D1,(A0)         sostituisci il 1° con il 2°
      MOVEQ    #1,D2           metti ad 1 il flag di scambio
NOSWAP ADDQ.L   #1,A0          fa puntare A0 alla coppia seguente
      SUBQ     #1,D3           decrementa contatore di iterazioni
      BNE      LOOP1          ripeti se ci sono ancora coppie
      TST      D2              test del flag di scambio
      BNE      PROG           ripeti se fatto almeno uno scambio
      STOP     #$2000         arresta processore

      ORG      $1000
ARRAY  DC.B     5,4,3,2,1     array, dopo ordinamento: 1,2,3,4,5
      END      PROG

```

Figura 5.1 Programma per l'ordinamento di un vettore di interi in ASM68K

6. Istruzioni di trasferimento dati

Si ricorda che:

- le istruzioni di trasferimento dati si rifanno concettualmente alla (F2-II.1.2):
a:= b
- le istruzioni sono genericamente dette di **move** (come nel 68000), talora di load se verso registri (solo una istruzione del 68000) e di store se verso la memoria (non per il 68000).

In tabella 6.1 sono esemplificate le istruzioni in esame, suddivise nelle classi che seguono.

6.1 Move in generale

L'istruzione MOVE del 68000 nella sua forma fondamentale si presenta come:

MOVE G1,G2

nel significato G2:= G1, coerentemente con il fatto che in ASM68K il secondo operando è sempre quello destinazione (contrariamente ad altri assembler).

L'istruzione è praticamente ortogonale: G1 e G2 sono trasferiti a livello di linguaggio macchina rispettivamente in EA, EA1 con il modo di EA praticamente qualsiasi (qui e nel seguito non si pongono in evidenza alcuni limiti di dettaglio, per i quali si rinvia ai manuali) e quello di EA1 che esclude soltanto quello immediato (non potrebbe essere altrimenti!), quello relativo a PC ed il diretto a registro A (esiste in proposito altra istruzione, MOVEA).

Esiste inoltre un'altra istruzione di MOVE, detta **Move Quickly** in quanto più veloce della MOVE:

MOVEQ im, D

per la quale l'immediato è posto negli ultimi 8 bit dell'istruzione, l'istruzione può quindi utilmente sostituire una MOVE per la quale l'operando-origine sia un immediato, nel caso che tale operando sia esprimibile in 8 bit.

6.2 Clear e Set

L'istruzione di Clear (codice operativo CLR) è per il 68000 un'istruzione di azzeramento aritmetico di un operando generale di tipo B,W oppure L::

CLR G

nel significato G:= 0.

Una istruzione particolare di set condizionato

Scc G

pone $G=0$ oppure $G=<\text{tutti } 1>$ a seconda del valore dei flag: essa sarà più chiara in seguito, quando sarà spiegata l'istruzione di salto condizionato *Bcc* (cfr. § 9.2).

Non esistono codici semplici per set/clear di bit; queste ultime operazioni sono peraltro possibili attraverso istruzioni (*BCHG*, *BCLR*, *BSET*) che azzerano, pongono ad 1 o complementano (*BCHG*) un bit opportunamente specificato, dopo che su tale bit sia stato fatto un test: tali istruzioni saranno illustrate in seguito (cfr. § 8.1.2).

6.3 Move e Load verso registri di indirizzamento

Una particolare classe di istruzioni è dedicata agli operandi di tipo "indirizzo", tipicamente trattati nei registri A. In particolare la move è:

MOVEA G,A

che differisce da *MOVE* solo per il registri-destinazione (nel move non poteva essere di tipo A, qui lo è necessariamente) ed inoltre perché non altera i flag.

Come per il *MOVE*, esistono altre istruzioni che si specializzano in istruzioni diverse se gli operandi sono indirizzi (cfr. *ADDA*, *SUBA*, *CMPA*).

Una logica diversa sottende l'istruzione

LEA M,A

che sposta in A l'indirizzo di M costruito nella fase di preparazione dell'operando e non il contenuto di detta locazione di memoria. M è espresso come un generico operando di tipo G, ma ha il vincolo che deve essere di tipo memoria (sono esclusi i modi diretto a registri, immediato e relativo al PC).

6.4 Move per registri speciali

Per quanto attiene al caricamento di registri speciali, esistono appositi codici di macchina che gestiscono il transito verso/da CCR, SR, SP. A livello assembler si usa sempre il codice *MOVE*, specificando il nome del registro speciale che ne è un operando:

- *MOVE G,SR*
- *MOVE G,CCR*
- *MOVE A,SP*
- *MOVE SR,G*
- *MOVE SP,A*

Tabella 6.1 Istruzioni di trasferimento dati

OP	Formato	Struttura	Semantica	flag XNZVC	Priv.	Note	Esempio	Commento
TRASFERIMENTO DATI								
Move in generale								
MOVE	a	MOVE Ga,Gb	Gb:=Ga	-xx00			MOVE D2,(A4)	
MOVEQ	c2*	MOVEQ im,D	D:=im	-xx00			MOVEQ #-1,D0	
Clear								
CLR	d	CLR G	G:=0	-0100			CLR (A0)	
Set/Clear condizionato								
Scc	b*	Scc G	G=tutti 1 se cc, altrimenti G=0	-----		G è byte cfr. tabella 9.2		
Move verso registri A								
MOVEA	a	MOVE G,A	A:=G e non modifica i flag	-----			MOVEA.L #\$1234,A0	
LEA	b	LEA M,A	A:=indirizzo di M	-----			LEA (A0),A1	
Move verso registri speciali								
MOVE registri speciali	d1	MOVE G,SR	SR:=G	xxxxxx	SI		MOVE (A0),SR	
	d1	MOVE SR,G	G:=SR	-----			MOVE SR,(A0)	
	d1	MOVE G,CCR	CCR:=G	xxxxxx			MOVE (A0),CCR	
	e	MOVE SP,A	A:=SP	-----				
	e	MOVE A,SP	SP:=A	-----				
Operazioni su stack								
PEA	d1	PEA M	pushm(SP,ind. di M)	-----			PEA (A0)	
Scambi								
EXG	c	EXG Ra,Rb	Rb :=: Ra	-----			EXG A0,D1	
SWAP	e	SWAP D	scambia word in D	-xx00				

6.5 Operazioni su stack

Nel 68000 esiste una particolare istruzione di push, per un operando di tipo indirizzo:

PEA M

nel significato: pushm(SP, indirizzo di M), cioè push nello stack di memoria puntato da SP l'indirizzo di M. In altri termini, l'istruzione è come la LEA, soltanto che l'indirizzo costruito viene messo nello stack piuttosto che in A.

Altre istruzioni operano su stack; in particolare:

- le istruzioni di MOVE che impiegano il modo di indirizzamento indiretto con pre-decremento attraverso il registro SP per l'operando destinazione sono equivalenti ad una push (es. MOVE D0,-(SP) equivale a pushm(SP,D0));
- le istruzioni di MOVE che impiegano il modo di indirizzamento indiretto con post-incremento attraverso il registro SP per l'operando sorgente sono equivalenti ad una pop (es. MOVE (SP)+,D1 equivale a popm(SP,D1));
- le istruzioni di salto a subroutine (BSR, JSR) effettuano pushm(SP, PC) (cfr. § 9.3); analogamente, le istruzioni di interruzione software (TRAP, TRAPV e CHK) effettuano pushm(SSP,PC), seguito da pushm(SSP,SR);
- le istruzioni di ritorno da subroutine (RTS, RTR) effettuano un'operazione di popm(SP,PC) (cfr. § 9.4); analogamente, l'istruzione di ritorno da interruzione (RTE) effettua un popm(SSP,SR) seguito da popm(SSP,PC).

Pertanto, non esistono, oltre a PEA, istruzioni che, come in altri processori, abbiano come obiettivo esclusivamente la realizzazione di un puro push o pop.

6.6 Operazioni di scambio

A questa classe appartengono le seguenti istruzioni:

- EXG Ra,Rb scambia i contenuti dei due registri Ra e Rb;
- SWAP D scambia la parte alta con la parte bassa del registro D

6.7 Istruzioni di move assenti nel 68000

Rispetto ad altri processori si notino le seguenti assenze nel processore 68000,

- Istruzioni di elaborazione e move (p.es. b:=-a)
- Operazioni di load e store (la ortogonalità del processore rende con il move queste operazioni)
- Operazioni di push e pop, fatta eccezione per quanto illustrato nel § 6.5.

7. Istruzioni aritmetiche e logiche

In tab.7.1 sono riportate le istruzioni aritmetiche del processore 68000, che rispecchiano lo schema della F2-III-II.1.4' (operazione binaria a 2 operandi), e della F2-III-II.1.5' (operazione unaria ad un operando). Si noti che il 68000 esclude istruzioni a 3 operandi (come in F2-III-II.1.4) ed istruzioni di elaborazione e move (come in F2-III-II.1.5).

7.1 Aritmetica binaria

7.1.1 Operazioni unarie

Si è già detto che il 68000 supporta l'aritmetica binaria a livello byte, word e long, ed a queste aritmetiche si riferisce l'operazione unaria

- NEG G equivalente a $G := -G$

Analogo è il comportamento dell'istruzione NEGX, salvo che quest'ultima considera che l'operando da negare sia $G+1$ se il flag X è uguale ad 1, G altrimenti (in sintesi, $G+X$):

- NEGX G equivalente a $G := -(G+X)$

e considera quindi che il dato G sia affetto da un riporto X^2 provenientegli da un'operazione analoga effettuata su una parte meno significativa di un dato complessivo cui G appartiene (ad esempio, G è l'ennesimo byte o l'ennesima longword di una rappresentazione del dato complessivo su k byte o long). Questa istruzione è necessaria quando si voglia costruire in software una aritmetica in precisione multipla (ad esempio, una aritmetica su 64 bit nella quale un dato sia rappresentato da 2 longword da 32 bit l'una).

7.1.2 Addizione e sottrazione

L'istruzione fondamentale di addizione in aritmetica binaria (ADD) e l'analoga per la sottrazione (SUB) assume le forme:

- ADD G,D equivalente a $D := D+G$
- ADD D,G equivalente a $G := G+D$

ed è "quasi" ortogonale in quanto esclude il caso $G1 := G1+G2$ ma per il resto consente in pratica qualsiasi modo di indirizzamento a G.

Altre operazioni di addizioni e sottrazione binarie sono:

- ADDI im,G equivalente a $G := G+im$
- ADDQ im,G come la precedente, con im di tipo byte (cfr. MOVEQ)

² Si ricorda che il flag X rappresenta il riporto aritmetico (carry) generatosi da una qualsiasi operazione aritmetica.

- ADDX Da,Db aggiunge anche riporto X, operandi-registro
- ADDX Ma,Mb come la precedente, con operandi-memoria.

Si guardino con attenzione le istruzioni con un operando immediato: con ADD non è possibile l'operazione $G:=G+im$, che invece è consentita dai due codici ADDQ (ove l'immediato è nel corpo dell'istruzione) ed ADDI, ove esso è posto nella parola seguente l'istruzione ed assume le dimensioni che gli competono (B, W, L).

Qualche importante considerazione va sviluppata sull'aritmetica delle somme algebriche. Si ricorda che:

- il 68000 adotta per i numeri negativi la rappresentazione in complementi secondo la quale un numero è rappresentato dal suo resto-modulo-M:

$$\rho(x) = |x|_M$$

- $M=2^k$ ove k è la lunghezza in bit dei registri nei quali i numeri sono rappresentati; l'intervallo di numeri rappresentabili su k bit è:

$$-2^{k-1} \leq x < 2^{k-1}$$

- le rappresentazioni dei numeri $\rho(x) = |x|_M$ sono sempre numeri positivi ed iniziano con il bit 1 (bit-segno) se sono rappresentazioni di numeri negativi; con il bit 0 se positivi;
- per la rappresentazione in complementi vale la relazione (cfr. F1, § III-IV.6):

$$|x \pm y|_M = |x|_M \pm |y|_M \quad (7.1)$$

Dalla summenzionata relazione deriva che la rappresentazione di una somma $|x \pm y|_M$ si ottiene dalla somma modulo-M delle rappresentazioni di x, $|x|_M$, e di y, $|y|_M$.

Ebbene, l'operazione che esegue l'addizionatore del processore è esattamente la somma modulo-8 (byte), modulo-16 (word) oppure modulo-32 (long)³ e da questa operazione si ottiene un numero S che se letto come numero naturale (unsigned) è appunto la somma-modulo-M degli addendi, ma che è anche, letto come signed, la rappresentazione in complementi della somma: un unico addizionatore (ed un'unica classe di codici operativi) funge da adder sia per numeri signed sia per numeri unsigned.

³ Si veda in seguito l'operazione di estensione del segno.

7.1 Istruzioni aritmetiche

OP	Formato	Struttura	Semantica	flag XNZVC	Priv.	Note	Esempio	Note
ARITMETICHE BINARIE								
Unarie								
NEG	d	NEG G	G:= -G	xxxxxx			NEG D4	Complementi a 2
NEGX	d	NEGX G	G:= -G - FlagX	xxxxxx			NEGX 16(A5)	Complementi a 2
Addizione e sottrazione								
ADD	b	ADD G,D	D:=D+G	xxxxxx			ADD \$15000,D1	D1è la destinazione
		ADD D,G	G:=G+D	xxxxxx			ADD D1, \$15000	L'indirizzo \$15000 è la destinaz.
ADDA	b	ADDA G,A	A:=A+G	-----			ADDA.L D1,A1	solo su longword
ADDI	d	ADDI im,G	G:=G+im	xxxxxx		im è in i+2,	ADDI \$A3BO,\$1500	A3B0+H26 è in i+2, 1500 è l'indirizzo di G, memorizzato in i+4
ADDQ	b	ADDQ im,G	G:=G+im	xxxxxx			ADDQ 2,\$1500	im (2) è nella word-istruzione al posto di R, in esempio 1500 è l'indirizzo di G, memorizzato in i+2
ADDX	b	ADDX Da,Db	Db:=Db+Da+FlagX	xxxxxx			ADDX D2,D3	Serve per precisione multipla
		ADDX Ma,Mb	Mb:=Mb+Ma+FlagX				ADDX -(A3),-(A1)	
SUB, SUBA, SUBI, SUBQ, SUBX		vedi ADD, ADDA, ADDI, ADDQ, ADDX						
Moltiplicazione e divisione								
MULS	b*	MULS G,D	D:=D*G	-xx00		Muliply. Signed= complementi	MULS (A0),D1	
MULU	b*	MULU G,D	D:=D*G	-xx00		Mult. Unsigned= Numeri naturali	MULU (A0),D1	
DIVS	b*	come MULS	D:=D/G	-xxx0		Div Signred		
DIVU	b*	come MULU	D:=D/G	-xxx0		Div Unsigned		
Estensione del segno								
EXT	d*	EXT D	estensione bit segno	-xx00			EXT.L D5	
ARITMETICA DECIMALE								
NBCD	d1	NBCD G	G:= -G - FlagX	x?x?x			NBCD.B (A1)	Rapresentazione BCD
ABCD	b*	ABCD Ma,Mb	Mb:=Mb+Ma+FlagX	x?x?x		solo con predecr.	ABCD -(A1), -(A2)	
SBCD	b*	vedi ABCD	Mb:=Mb-Ma-FlagX	x?x?x		solo con predecr.		

Un problema nasce però per l'overflow. L'overflow per i numeri naturali coincide con il traboccamento finale dell'adder, che il processore segnala nel flag C: se ad esempio si effettua la seguente somma di byte:

$$\begin{array}{r} 10000000 \quad (=128)+ \\ 10000010 \quad (=130) \\ \hline 1 \ 00000010 \quad (= 2) \end{array}$$

si ottiene come somma 2 (che è il resto modulo-256 della somma 258) ed un Carry=1, che annuncia appunto che c'è stato overflow di unsigned nella somma in quanto il numero 258 non è rappresentabile nel byte.

Nel caso di aritmetica per numeri signed, la condizione di overflow è diversa, in quanto non più rappresentata dal bit Carry. Ad esempio, la medesima addizione di cui sopra assume il significato di $-128-126=-2$ (ancora resto della somma -254), con overflow e $C=1$, mentre l'addizione $127-126=1$:

$$\begin{array}{r} 01111111 \quad (= 127)+ \\ 10000010 \quad (= -126) \\ \hline 1 \ 00000001 \quad (= 1) \end{array}$$

produce anch'essa $C=1$, ma non overflow.

Analogamente, l'addizione:

$$\begin{array}{r} 00000010 \quad (= 2)+ \\ 10000010 \quad (= -126) \\ \hline 0 \ 10000100 \quad (= -124) \end{array}$$

produce $C=0$ e non overflow mentre

$$\begin{array}{r} 01111111 \quad (=127)+ \\ 01111111 \quad (=127) \\ \hline 0 \ 11111110 \quad (= -2) \end{array}$$

produce $C=0$ ed overflow (-2 è il resto mod-256 della somma 254).

Dunque, il Carry C non è significativo per la valutazione dell'overflow in aritmetica signed. Si osservi ora in particolare l'ultimo esempio e si noti che entrambi gli addendi sono positivi mentre la somma è un numero negativo, Questo non è un caso, ma è viceversa la regola:

In aritmetica dei complementi si ha overflow se dall'addizione di due numeri positivi deriva una somma negativa o viceversa dall'addizione di due numeri negativi deriva una somma positiva.

Quanto sopra, ulteriormente esemplificato dall'addizione:

$$\begin{array}{rcl}
 10000001 & (= -127) + \\
 10000001 & (= -127) \\
 \hline
 1\ 00000010 & (= 2)
 \end{array}$$

può essere facilmente dimostrato graficamente (cfr. fig. 7.2): essendo la somma compresa fra $-M$ ed M , l'overflow positivo è un numero compreso fra $M/2$ ed M , il cui resto mod- M è quello dei numeri negativi dell'intervallo rappresentabile e viceversa per l'overflow negativo.

Per segnalare al programmatore il verificarsi della condizione di overflow in aritmetica signed, il processore utilizza il flag V: *V=1 indica overflow per addizione signed* ed è costruito confrontando i segni degli addendi con quello della somma (o con tecnica equivalente).

In conclusione, deve essere cura del programmatore verificare, successivamente ad un'operazione di addizione, il flag V oppure il flag C, a seconda che si stia operando in aritmetica signed (in complementi) o in aritmetica unsigned.

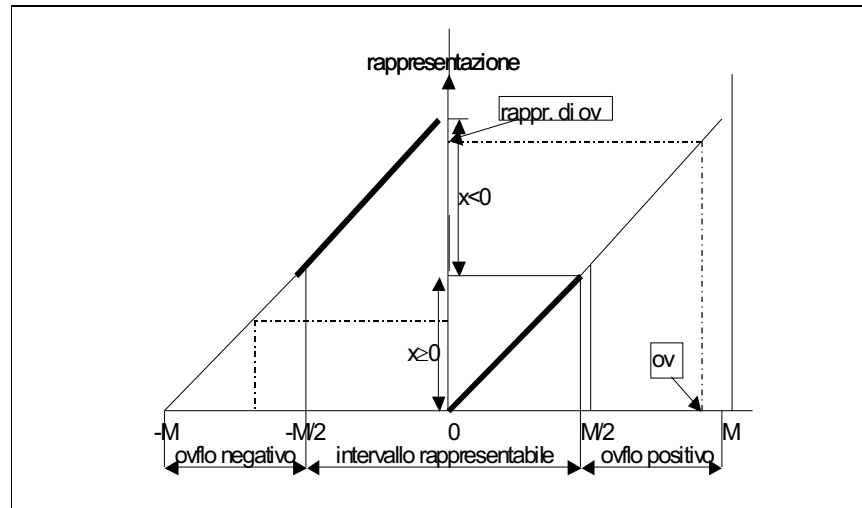


Figura 7.2 Rappresentazione per complementi e overflow

7.1.3 Estensione del segno

Una importante operazione aritmetica è quella dell'estensione del segno:

EXT D

che può essere applicata ad un operando di tipo byte (oppure word) e lo trasforma in uno di tipo word (oppure long) avente nel byte (o nella word) più significativo

tutti bit 1 se l'operando originario era negativo, tutti 0 se positivo, come ad esempio:

10110101 → 11111111 10110101
01100111 → 00000000 01100111

L'estensione del segno ha il notevole significato aritmetico di mutare un dato espresso in complementi in modulo- $M=2^k$ ($k=8$ per il byte) in complementi modulo $M^2=2^{2k}$, passando dunque dall'aritmetica di tipo byte a quella di tipo word o da questa a quella longword.

7.1.4 Moltiplicazione e divisione

Si premette che:

- 1) Il prodotto di due numeri compresi in un intervallo $[0, M)$ è compreso nell'intervallo $[0, M^2)$ e quindi se i fattori di un prodotto sono numeri su k bit il prodotto necessita di $2k$ bit per essere compiutamente espresso.
- 2) Lo stesso vale per le rappresentazioni in complemento: i fattori su k bit sono nell'intervallo $[-M/2, M/2)$ e il prodotto in $[-M^2/4, M^2/4]$.
- 3) Vale anche per il prodotto la regola dei complementi espressa in (7.1) sulle operazioni \pm .
- 4) Non vale per la moltiplicazione la regola dell'overflow espressa per la somma algebrica in quanto, essendo l'intervallo del prodotto molto ampio, un eventuale overflow positivo può essere rappresentato sia da prodotti positivi che negativi.

Tanto premesso, il 68000 prevede due distinte moltiplicazioni:

- MULU G, D Multiply Unsigned (numeri naturali)
- MULS G, D Multiply Signed (rappresentazione in complementi)

Entrambe operano su operandi a 16 bit (word)⁴ fornendo un risultato a 32 bit (long).

Approfondiamo dapprima l'istruzione MULU e facciamo riferimento ad un esempio:

1111111111111110	(65534) x	
0000000000000011	(3) =	
0000000000000010	1111111111111010	(196602)

Per leggere bene l'esempio e per familiarizzare con i numeri binari, si noti che:

⁴ D è in effetti a 32 bit, ma come operando l'istruzione ne considera solo gli ultimi 16, come destinazione tutti.

- $2^{16}=65536$, una stringa di tutti 1 è il massimo numero esprimibile su 16 bit, pari a 65535.
- il primo fattore è di 1 unità inferiore a 65535, dunque è 65534
- il prodotto è esteso su 32 bit e si può leggere come:
 $2 \times 65536 + 65530 = 196602$
- si può verificare la stringa di bit effettuando per esercizio a mano la moltiplicazione binaria con il medesimo algoritmo usato manualmente per i numeri decimali.

La domanda da porsi a valle dell'esempio è come adoperare il risultato della moltiplicazione. È vero che si è ottenuto il prodotto esatto sui 32 bit, ma se questo risultato dovesse di nuovo essere moltiplicato per un altro valore? e poi ancora?...In realtà, in un calcolatore si fissa una aritmetica nella quale gli operandi hanno tutti una medesima lunghezza in bit e quindi, se i fattori erano a 16 bit, anche il prodotto deve esserlo ed occorre completare il moltiplicatore con un indicatore di overflow, come avviene del resto anche per l'addizionatore.

Il controllo di overflow per la moltiplicazione è lasciato integralmente al programmatore con l'uso delle istruzioni di moltiplicazione. In effetti, sui 16 bit l'intervallo dei numeri trattabili è $[0, 65536)$ e quindi la moltiplicazione dell'esempio dava overflow.

Nella teoria della moltiplicazione (che qui si accenna soltanto) si divide in generale il risultato di un prodotto $XY=Z$ in due componenti, P e Q, ciascuna delle dimensioni di X e Y, avendosi

$$Z = QM + P$$

(nell'esempio Q è la parte di sinistra, P la parte meno significativa). Da ovvie considerazioni si deriva che:

*Per l'aritmetica dei numeri naturali si ha overflow sempre che sia $Q \neq 0$.
 Per $Q=0$ il prodotto è P.*

In qualche caso (come spesso accade per la mantissa di un numero in virgola mobile) si usa trattare non numeri interi, ma numeri frazionari (minori di 1) con la cosiddetta *aritmetica dei frazionari*.

Nell'aritmetica dei frazionari un numero intero x in un registro è la rappresentazione di un numero frazionario x/M : il numero è come se avesse il punto (virgola) frazionario in prima posizione. Ad esempio, l'intero

0000000000000011

rappresenta il frazionario $3/2^{16}$.

Per la somma algebrica nulla cambia: l'addizionatore di interi e quello dei frazionari coincidono. Diverso è il caso della moltiplicazione: il prodotto $Z=QM+P$ è frazionario e la sua parte più significativa Q si può assumere rappresentativa del prodotto espresso con il medesimo numero di cifre di X e Y :

Per l'aritmetica dei frazionari non si ha mai overflow. Il risultato è Q e P è l'approssimazione del risultato.

Il prodotto frazionario dell'esempio di cui sopra è dunque

$$0000000000000010 * 2^{16} = 0, 0000000000000010$$

mentre

$$111111111111010 * 2^{32} \text{ ne è l'approssimazione.}$$

Nella aritmetica in virgola mobile, poi il prodotto può essere normalizzato, guadagnando in precisione. Sempre con riferimento all'esempio, il prodotto sarebbe:

$$101111111111110 * 2^{30}$$

mentre $(10)_2 * 2^{-62}$ ne sarebbe l'approssimazione.

Ai fini ora dell'analisi dell'istruzione MULS, si riprenda il medesimo esempio analizzandolo nel contesto dei numeri signed:

$$\begin{array}{r} 111111111111110 \quad (-2) \times \\ 000000000000011 \quad (3) = \\ \hline (000000000000010) \quad 111111111111010 \quad (-6) \end{array}$$

e quindi la componente P del prodotto intero sarebbe ancora valida nell'aritmetica dei complementi, coerentemente con l'osservazione 3) fatta all'inizio del paragrafo. Ma, coerentemente con l'osservazione 4), nessun controllo vi sarebbe per l'overflow. Ad esempio, la moltiplicazione

$$\begin{array}{r} 011111111111111 \quad (32767) \times \\ 0000000000000110 \quad (6) = \\ \hline 000000000000010 \quad 111111111111010 \quad (196602) \end{array}$$

fornisce il medesimo risultato, ma è un overflow.

Occorre dunque assumere un altro algoritmo per il calcolo del prodotto ed ecco perché esiste l'istruzione

MULS G,D

Essa fornisce il prodotto espresso in modulo M^2 e quindi su tutti i 32 bit, ad esempio⁵:

$$\begin{array}{r} 111111111111110 \quad (-2) \times \\ 000000000000011 \quad (3) = \\ \hline (111111111111111) \quad 111111111111010 \quad (-6) \end{array}$$

Poiché il risultato desiderato è quello modulo-M, è allora sufficiente che la prima word sia la pura estensione in segno della seconda affinché non vi sia overflow.

*Per l'aritmetica di interi in complementi si ha overflow sempre che Q non sia la pura estensione in segno di P .
In assenza di overflow, il prodotto è P .*

Con l'algoritmo dei complementi ed aritmetica dei frazionari, essendo tutto il risultato espresso in complementi, vale ancora la regola:

Per l'aritmetica dei frazionari in complementi non si ha mai overflow. Il risultato è Q , e P è l'approssimazione del risultato.

7.2 Aritmetica decimale

Il 68000 gestisce anche l'aritmetica decimale, mediante le istruzioni:

- ABCD Ma, Mb equivalente a Mb := Mb+Ma+X
- ABCD Ma, Mb equivalente a Mb := Mb+Ma+X
- NBCD G equivalente a G := -G-X

In tale aritmetica i numeri sono rappresentati in forma BCD (*Binary Coded Decimal*), con due cifre impaccate in ogni byte e posti in memoria ad indirizzi crescenti (la cifra più significativa in testa). Le istruzioni, dunque, operano sui due addendi-cifre portando in conto il riporto dell'operazione precedente che si suppone essere stata effettuata su cifre meno significative. Per facilitare l'operazione, le istruzioni operano un predecremento, in modo da spostarsi automaticamente verso le cifre più significative.

7.3 Istruzioni d'indirizzamento e controllo

L'addizione (ADDA) e la sottrazione (SUBA) sono istruzioni nate per gestire gli indirizzi:

- ADDA G,A equivalente ad A := A+G

Similmente alla MOVEA, ADDA e SUBA non alterano i flag.

⁵ L'algoritmo consiste correttamente nell'estendere il segno dei fattori, esprimendoli quindi in modulo M^2 , e procedendo poi come nel caso dei numeri naturali.

7.4 Istruzioni logiche

Come operazione unaria, esiste il NOT (cfr. tabella 7.3), strutturalmente simile alla NEG, che effettua il not bit a bit.

Come operazioni binarie fondamentali esistono la AND, la OR e la EOR (or esclusivo) strutturalmente simili alla ADD:

- AND G, D equivalente a $D := D \wedge G$
- AND D, G equivalente a $G := G \wedge D$

che effettuano l'operazione logica bit a bit.

Esistono altresì le istruzioni ANDI, ORI, EORI che nella versione fondamentale sono strutturalmente simili alla ADDI:

- ANDI im, G equivalente a $G := G \wedge im$

Per quest'ultima classe di istruzioni esiste inoltre la possibilità di usare come destinazione i registri speciali CCR oppure SR (cfr. tab.7.3).

7.5 Istruzioni di Shift

È previsto uno shift logico di un solo bit a sinistra (LSL) o a destra (LSR) con immissione del bit 0 a destra:

- LSL M equivalente a $shl(M, 0)$

e traboccamento verso i flag C ed X del bit uscente.

Se l'operazione avviene su un registro dati invece che in memoria, è allora possibile effettuare uno shift multiplo (sia a sinistra che a destra):

- LSL Da, Db equivalente a $mshl(Db, 0, Da)$
- LSL im, D equivalente a $mshl(D, 0, im)$

Analoghe sono le operazioni di shift aritmetico (ASL, ASR): la ASL è identica alla LSL (cambia solo l'effetto sul flag V) mentre la ASR conserva il bit segno (il bit più significativo) invece che immettervi lo 0. Infatti, si dimostra che in tal modo lo shift a destra equivale ad una divisione per 2.

Ancora analoghe sono le istruzioni di shift circolare:

- ROL M equivalente a $shl(M)$
- ROL Da, Db equivalente a $mshl(Db, Da)$
- ROL im, D equivalente a $mshl(D, im)$

(ROR se a destra), per le quali il bit circolante posiziona anche il flag C, e le

- ROXL M equivalente a $shl(M)$
- ROXL Da, Db equivalente a $mshl(Db, Da)$
- ROXL im, D equivalente a $mshl(D, im)$

per le quali il bit circolante posiziona anche i flag C e X. Queste ultime sono usate per l'aritmetica in precisione multipla.

8. Posizionamento dei flag - Istruzioni di comparazione e test

Nel 68000, così come nella maggioranza dei processori, i bit-flag sono posizionati in conseguenza dell'esecuzione di quasi tutte le istruzioni di trasferimento e di calcolo (vedi tutte le tabelle dei codici). Le regole per il posizionamento dei flag sono spesso ovvie; in particolare, per le operazioni aritmetiche si ha:

- $Z=1$ se il risultato è nullo,
- $N=1$ se il risultato è negativo,
- $C=1$ se c'è un riporto (o prestito) uscente,
- $V=1$ se c'è un overflow in aritmetica dei complementi,
- $X=C$

Così come in tutti i processori che fanno ricorso sistematicamente ai flag, il 68000 possiede altresì apposite istruzioni di *comparazione*, che comparano due operandi aritmetici, e di *test* che posiziona i flag in funzione di un unico operando.

Tabella 7.2 Istruzioni logiche e di shift

OP	Formato	Struttura	Semantica	flag XNZVC	Priv.	Note	Esempio	Commento
LOGICHE								
unarie								
NOT	d	NOT G	G:= not G	-xx00			NOT \$1000	
binarie								
AND	b	AND G,D	D:=D and G	-xx00			AND \$15000,D1	D1è la destinazione
		AND D,G	G:=G and D	-xx00			AND D1, \$15000	L'indirizzo \$15000 è la destinazione
ANDI	d	ANDI im,G	G:= G and im	-xx00	SI	il modo distingue i formati	ANDI #\$0F,D0	
		ANDI im,CCR	CCR:= CCR and im				ANDI #\$0F,CCR	
		ANDI im,SR	SR:=SR and im				ANDI #\$8000,SR	
OR	d-b	vedi AND	D:= D or G	-xx00			OR (A2)+,D0	
ORI	d	vedi ANDI	G:= G or im	-xx00			ORI #\$0F,D0	
EOR	b	come ADD		-xx00				
EORI	d	come ADDI	G:= G xor im	-xx00				
DI SHIFT LOGICO								
LSL	b*	LSL Da,Db	mshl (Db,0, Da)	xxx0x		un bit distingue i formati	LSL D3,D0	
		LSL im,D	mshl (D, 0, im)	xxx0x			LSL #2,D0	
	d1	LSL M	shl (M, 0)	xxx0x			LSL (A4)	
LSR	b*	come LSL	shift a destra	xxx0x		shift 1 bit		
ROL	b	ROL Da,Db	mcil(Db,Da)	-xx0x			ROL D3,D0	
	b	ROL im,D	mcil(D,im)	-xx0x			ROL #2,D0	
	d1	ROL M	cil(M)	-xx0x			ROL (A4)	
ROR	b, d1	come ROL	shift a destra	-xx0x				
ROXL	b, d1	come ROL	shift anche FlagX	xxx0x				
ROXR	b, d1	come ROXL	shift a destra	xxx0x				
DI SHIFT ARITMETICO								
ASL		come LSL	posiziona anche V	xxxxx				V=1 se il bit segno cambia
ASR		come LSR	mantiene bit segno	xxxxx				

8.1 Comparazione aritmetica

L'istruzione di comparazione fondamentale:

- CMP G,D

equivale a SUB G,D, con l'unica differenza che la quantità D-G non è memorizzata in D. I flag sono posizionati coerentemente con il valore della differenza D-G, e quindi:

- Z=1 se D = G;
- N=1 se il bit più significativo (bit segno) del risultato della sottrazione D-G è 1;
- C=1 per numeri unsigned indica che D < G;
- V=1 se D e G hanno segno opposto (cioè D e -G hanno lo stesso segno) ed il risultato della sottrazione D-G ha segno opposto a D;
- X viene lasciato inalterato.

Analoghe sono le istruzioni:

- CMPA G, A (vedi SUBA)
- CMPI im,G (vedi SUBI)
- CMPM Ma,Mb (vedi SBCD)

La Tabella 8.1 riporta, a seconda della relazione esistente tra i valori di G e D, sia per il caso signed che per il caso unsigned, le combinazioni di valori dei flag che si ottengono in seguito all'esecuzione dell'istruzione CMP G,D.

Tabella 8.1 Relazione tra i valori di D e G e i flag dopo CMP G,D⁶

	caso UNSIGNED	caso SIGNED
D=G	Z=1	Z=1
D<G	C=1	$N \oplus V=1$
D>G	C=0 and Z=0	Z=0 and $N \equiv V=1$
D<=G	C=1 or Z=1	Z=1 or $N \oplus V=1$
D>=G	C=0 or Z=1	$N \equiv V=1$

⁶ Si ricorda che il simbolo \oplus indica la relazione binaria "or esclusivo":

$$x \oplus y = (x \wedge \bar{y}) \vee (\bar{x} \wedge y)$$

Il simbolo \equiv indica invece la relazione binaria "equivalenza":

$$x \equiv y = (x \wedge y) \vee (\bar{x} \wedge \bar{y})$$

8.2 Test

Esistono nel 68000 due classi di istruzioni di test, l'una che opera su un bit, e l'altra su un dato di 8, 16 o 32 bit.

8.2.1 Test su un bit

Il test su un bit avviene con l'istruzione:

- BTST *im*,G

che posiziona il flag Z in funzione del valore del bit *im*-esimo di G. Le istruzioni BCLR, BSET, BCHG operano come BTST ma inoltre azzerano, pongono ad 1 o complementano rispettivamente il bit sul quale è stato fatto il test (cfr. § 6.2).

8.2.2 Test su byte, word o longword

Il test su un byte, su una word o una longword avviene con l'istruzione:

- TST G

che posiziona i flag N e Z in funzione del valore di G. L'istruzione:

- TAS G

opera come TST ma solo su un dato di tipo byte, ed inoltre pone ad 1 il bit più significativo (*msb, most significative bit*) di G.

Tabella 8.2 Istruzioni di comparazione e di test

OP	Formato	Struttura	Semantica	flag XNZVC	Priv.	Note	Esempio	Commento
Compare								
CMP	b	CMP G,D	D-G e posiziona flag	-xxxx		come ADD	CMP \$001234,D1	
CMPA	b	CMPA G,A	A-G e posiziona flag	-xxxx		come ADDA		
CMPI	d	CMPI im,G	G-im e posiz. flag	-xxxx		come ADDI		
CMPM	b*	CMPM Ma,Mb	Mb-Ma e posiz. flag	-xxxx		solo con postincr. (cfr. ABCD)	CMPM (A0)+,(A1)+	
Test								
TST	d1	TST G	posiziona flag in funzione di G	-xx00				
TAS	d1	come TST,poi	pone ad 1 bit msb	-xx00			TAS (A0)	
BTST	b*	BTST D,G	posiziona flag in funz. di bit D di G	--x--		Z=1 se bit=0	BTST D3, D1	
	d1	BTST im,G	posiziona flag in funz. di bit im di G				BTST #4, D1	
BSET	b	come BTST, poi	pone bit ad 1	--x--				
BCHG			complementa bit	--x--				
BCLR			azzerà bit	--x--				

9. Istruzioni di salto

Si ricorda che:

- Per salto si intende un riposizionamento del contatore di programma ad un determinato valore a : $PC := a$
- Il salto è *assoluto* se l'operando O dell'istruzione esprime in modo direttamente il valore dell'indirizzo cui saltare: $PC := O$
- Il salto è *relativo* se l'operando O lo esprime come incremento da dare a PC : $PC := PC + O$
- Il salto è *condizionato*, se avviene solo se una assegnata condizione logica è vera: *if ... then* $PC := \dots$
- Il salto è *incondizionato* se avviene comunque.

9.1 Salti incondizionati

Le istruzioni di salto incondizionato sono sia assolute (*jump*) che relative (*branch*). Il salto assoluto:

- **JMP M**

provoca il salto all'indirizzo di memoria M specificato nell'operando G con le classiche tecniche del 68000 (sono ovviamente esclusi i modi di indirizzamento diretti a registro, né hanno senso il post- o il pre-decremento). Ad esempio:

- **JMP \$7800:** salto diretto all'indirizzo 7800_{16}
- **JMP ciclo:** salto diretto all'indirizzo associato all'etichetta "ciclo"
- **JMP (A3)** salto indiretto all'indirizzo contenuto in $A3$

Il salto relativo:

- **BRA *dest***

provoca il salto all'indirizzo di memoria $PC + disp$, ove PC è il valore del Program Counter dopo che è stato incrementato di 2 nella fase di fetch, e $disp$ è un displacement che viene calcolato dall'assemblatore come differenza tra l'indirizzo *dest* (operando dell'istruzione assembler) e $(PC+2)$. Il displacement *disp* può essere un dato ad 8 oppure a 16 bit, espresso in aritmetica dei complementi, che, prima di essere addizionato a $(PC+2)$, viene automaticamente esteso nel segno (cfr. § 7.1). Nel primo caso, l'istruzione occupa 2 byte (il displacement è inserito nella stessa word che contiene il codice operativo), mentre nel secondo caso l'istruzione occupa 4 byte (il displacement è codificato in una word aggiuntiva). Dato un indirizzo di destinazione *dest*, l'assemblatore automaticamente sceglie una forma di codifica tra le due possibili, con il seguente criterio:

- se $dest-(PC+2)$ è compreso in $[-128,126]$ viene scelta la forma di salto “corto”, cioè con displacement ad 8 bit;
- altrimenti, viene scelta la forma di salto “lungo”, cioè con displacement a 16 bit.

Si osservi, che, quando l’indirizzo di destinazione è specificato mediante una etichetta definita più avanti nel codice sorgente (*riferimento in avanti*), l’assemblatore sceglie sempre la forma di salto lungo. In questo caso, alcuni assemblatori consentono al programmatore di forzare la scelta del formato corto, aggiungendo il suffisso .S (*short*) all’etichetta usata come operando.

Infine, si fa notare che, per specificare l’indirizzo di destinazione di un salto, è possibile utilizzare il simbolo ‘*’ (che indica l’indirizzo a cui è memorizzata l’istruzione corrente, cfr. §4.4).

9.2 Salti condizionati

Diverse sono in generale le tecniche per i salti condizionati (cfr. F2, § III-III.9). Nel 68000, essi si basano esclusivamente sull’analisi dei flag di condizione contenuti nel registro CCR. Il salto è inoltre solo di tipo relativo (branch) ed assume la forma:

- Bcc *dest*

ove cc specifica in qualche modo il valore di CCR:

if cc then PC:= PC+disp

In particolare cc viene espresso in assembler attraverso una sigla che esprime mnemonicamente il significato della condizione da testare (esempio, GE= maggiore o eguale, EQ= Eguale e così via) mentre a livello di linguaggio macchina cc viene espresso in un codice a 4 bit nel campo cc dell’istruzione. In tabella 9.2 sono listate le condizioni cc nella simbologia di ASM68K, in linguaggio macchina e nella semantica.

Si ricorda che cc è posizionato dalle istruzioni precedenti quella di salto, tipicamente da una istruzione CMP O_1, O_2 che calcola $R=O_2-O_1$ e posiziona i flag di conseguenza. Pertanto le sigle GT, LT, etc. vanno lette come $O_2 > O_1$, $O_2 < O_1$, etc.

Il significato delle espressioni logiche in ultima colonna è immediato per le condizioni CC, CS, NE, EQ, VC, VS, VC, PL, MI. Per quanto attiene GE, si ricorda che nell’aritmetica dei complementi, in caso di overflow positivo il bit-segno del risultato è 1 e quindi $N=1$ e viceversa, in caso di overflow negativo; il fatto dunque di $O_2 \geq O_1$ va individuato attraverso i flag con due casi: quello in cui non vi sia stato overflow (\bar{V}) e sia $R \geq 0$ (\bar{N}) oppure quello in cui, essendovi stato l’overflow (V) risulti $R < 0$ (N). All’opposto LT: $O_2 < O_1$ se il risultato è negativo (N) senza overflow (\bar{V}) oppure positivo ma con overflow. GE è come GT, ma de-

ve escludere il caso che sia $R=0$ (\bar{Z}), LE ne è il negato. Infine, HI e LS sono utilizzate per il confronto di numeri unsigned: la prima è vera se e solo se non è 1 né C né Z (cioè $O_2 > O_1$); la seconda è vera se e solo se o C o Z sono 1, (cioè $O_2 \leq O_1$).

Tabella 9.2. Codici cc

ASM	mnemonico	semantica	macchina	condizione
HI	Higher	maggiore	0010	$\bar{C} \wedge \bar{Z}$
LS	Lower or the Same	minore o uguale	0011	$C \vee Z$
CC	Carry Clear	$C=0$	0100	\bar{C}
CS	Carry Set	$C=1$	0101	C
NE	Not Equal (to zero)	$Z=0$	0110	\bar{Z}
EQ	Equal (to zero)	$Z=1$	0111	Z
VC	oVerflow Clear	$V=0$	1000	\bar{V}
VS	oVerflow Set	$V=1$	1001	V
PL	Plus	positivo	1010	\bar{N}
MI	Minus	negativo	1011	N
GE	Greater than or Equal	\geq	1100	$(N \wedge V) \vee (\bar{N} \wedge \bar{V})$ ovvero $N \equiv V$
LT	Less than	$<$	1101	$(N \wedge \bar{V}) \vee (\bar{N} \wedge V)$ ovvero $N \oplus V$
GT	Greater than	$>$	1110	$(N \wedge V \wedge \bar{Z}) \vee (\bar{N} \wedge \bar{V} \wedge \bar{Z})$ ovvero $\bar{Z} \wedge (N \equiv V)$
LE	Less than or Equal	\leq	1111	$Z \vee (N \wedge \bar{V}) \vee (\bar{N} \wedge V)$ ovvero $Z \vee (N \oplus V)$

Tabella 9.1 Istruzioni di salto

OP	Formato	Struttura	Semantica	flag XNZVC	Priv.	Note	Esempio	Commento
Salti incondizionati								
JMP	d1	JMP M	PC:=M	----			JMP ciclo	salta all'indirizzo il cui label è "ciclo" in ASM
BRA	c1+c2	BRA M	PC=PC+disp	----		= Bcc, con cc=0000 disp = M-(PC+2)	BRA \$8000	
Salti condizionati								
Bcc	c1+c2	Bcc im	if cc PC=PC+im	----		vedi tabella 9.2	BGE *+\$4	se >= salta 1 istruzione lunga 2 byte dopo questa (lunga anche essa 2 byte)
Salti a subroutine								
JSR	d1	JSR M	pushm(SP, PC); PC=M	----				
BSR	c1+c2	BSR M	pushm(SP, PC); PC=PC+disp	----		= Bcc con cc=0001 disp = M-(PC+2)	BSR DEST	
Ritorno da subroutine								
RTS	f	RTS	pop(SP,PC)	----			RTS	
RTR	f	RTR	popm(SP,CCR); popm(SP,PC)	xxxxxx		ritorno e ripristino	RTR	

La tabella 9.2 vale anche per la condizione cc espressa nell'istruzione "set con condizione" Scc (cfr. § 6.2). Inoltre, per Scc si ha:

- cc=0000 significa "vero" e quindi determina il set del byte a "tutti 1" (similmente, in Bcc il valore cc=0000 significa "salto incondizionato, cfr. tabella 10.1)
- cc=0001 significa "falso" e quindi determina il "clear" del byte (questo valore logico non avrebbe senso per il Bcc ed è sfruttato dal processore per un altro codice).

9.3 Salti a sottoprogrammi

Come è noto (cfr § F2-III-III.10), il salto ad una subroutine che inizi all'indirizzo SUB avviene con le due microoperazioni:

SAVE:=PC; PC:= SUB

ed esistono diverse tecniche di salvataggio del PC corrente. Nel 68000 il salvataggio è nello stack e il salto può essere assoluto o relativo:

- JSR M
- BSR im

che operano come JMP, BRA dopo aver salvato il PC nello stack:

pushm (SP, PC);

9.4 Ritorno da sottoprogrammi

Il ritorno si realizza con l'istruzione RTS, simmetrica a quella di salto a subroutine, che ripristina il PC quale era al tempo della chiamata:

popm (SP, PC);

Il processore 68000 dispone anche dell'istruzione RTR, che effettua le operazioni:

popm (SP, CCR); popm (SP, PC);

Si ricorda che la gestione attraverso lo stack dei salti a sottoprogrammi e dei relativi ritorni consente una agevole gestione del nesting delle chiamate.

Tabella 10.1 Altre istruzioni

OP	Forma- to	Struttura	Semantica	flag XNZVC	Priv.	Note	Esempio	Commen- to
Trasferimento a blocchi								
MOVEM	d	MOVEM reg.,M MOVEM M, reg.	Trasferimento a blocchi	-----			MOVEM D1/D3/A1- A3,(A5) MOVEM (A5),D1/D3/A1- A3	
Stop – NOP								
STOP	f	STOP im	SR:=im; attende interrupt	xxxxx	SI			
NOP	f	NOP	No Operation	-----			NOP	
Gestione record di attivazione di sottoprogrammi sullo stack								
LINK	e	LINK A,im	pushm(SP,A); A:=SP; SP:=SP+im	-----			LINK A6,#-8	
UNLK	e	UNLK A	SP:= A popm (SP, A)	-----			UNLK A2	
Decremento e salto								
DBcc	c1	DBcc D,im	if (not cc) {D:=D-1; if (D != -1) then PC=PC+im}	-----			DBNE D0,*-14	
Interruzioni								
TRAP	e	TRAP im	S=1; pushm(SSP,PC); pushm(SSP,SR); PC:=VETT(im)	-----		genera interruzione software VETT(im) = M[128+4*im]	TRAP #15	
TRAPV	f	TRAPV im	if (V) { TRAP im}	-----		genera interruzione se V=1	TRAPV	
CHK	b	CHK G,D	if (D ∉ [0,G]) { S=1; pushm(SSP,PC); pushm(SSP,SR); PC:=M[24]; }	-x???		genera interruzione se D è fuori dai limiti 0,G	CHK (A1),D4	
ILLEGAL	f	ILLEGAL	S=1; pushm(SSP,PC); pushm(SSP,SR); PC:=M[16];	-----		genera TRAP di tipo Illegal instruction		
Ritorni da interrupt								
RTE	f	RTE	popm(SP,SR); popm(SP,PC)	xxxxx	SI	ritorno da exception	RTE	
Input/output								
RESET	f	RESET	Pone ad 1 la linea di reset verso periferiche	-----	SI		RESET	
MOVEP	b	MOVEP M, D	Trasferimento per periferiche	-----			MOVEP 0(A0),D1	
		MOVEP D,M		-----			MOVEP D0,0(A0)	

10. Altre istruzioni

Per motivi di completezza si illustrano in questo paragrafo le altre istruzioni del 68000, pur non rientrando esse negli obiettivi di questo capitolo, che vuole presentare soltanto le istruzioni fondamentali del processore. Quelle qui illustrate sono in genere o istruzioni che ottimizzano l'uso del processore oppure istruzioni dedicate alla sua gestione a livello del software di base.

10.1 Istruzione DBcc

Allo scopo di rendere più efficiente la programmazione di cicli, il processore 68000 è stato dotato dell'istruzione:

- DBcc D,dest

la quale corrisponde alla seguente sequenza di micro-operazioni:

```
if (not cc) then
begin
  D := D - 1;
  if (D = -1) then
    PC := PC + 2;          { esegui l'istruzione seguente }
  else
    PC := PC + disp;      { esegui l'istruzione all'indirizzo dest }
end
else
  PC := PC + 2;          { esegui l'istruzione seguente }
```

L'istruzione DBcc è impiegata per creare un ciclo che termina quando sia verificata la condizione cc oppure il contatore di iterazioni D abbia raggiunto il valore -1. In aggiunta ai codici di condizione cc della tabella 9.2, per l'istruzione DBcc è consentita la variante con cc=false, che in assembler è associata ai due codici operativi equivalenti DBF (*Decrement And BRanch on False*) e DBRA (*Decrement And BRanch Always*). Il segmento di programma che segue illustra l'uso dell'istruzione DBEQ per realizzare la ricerca della prima occorrenza di un carattere (memorizzato in D0) all'interno di un'area di memoria.

SEARCH	LEA.L	BUFFER,A0	A0 punta all'inizio dell'area
	MOVE.W	#BUFSIZE,D1	D1 := dimensione di BUFFER (>0)
LOOP	CMP.B	(A0)+,D0	confronta un byte
	DBEQ	D1,LOOP	esci se (M[(A0)]==D0 or D1==1)

10.2 Trasferimenti a blocchi

Una particolare istruzione, MOVEM, consente di trasferire un blocco di locazioni di memoria in un insieme specificato di registri o viceversa: per i dettagli si rinvia ai manuali.

Utilizzando il modo di indirizzamento indiretto con pre-decremento su SP, l'istruzione MOVEM salva sulla cima dello stack il valore di un insieme di registri D ed A specificato come operando sorgente:

- MOVEM.L D0-D4/A0/A2-A4,-(SP)

L'operazione duale si realizza impiegando il modo di indirizzamento indiretto con post-incremento su SP:

- MOVEM.L (SP)+,D0-D4/A0/A2-A4

L'istruzione MOVEM è tipicamente impiegata all'entrata in una subroutine, per "salvare" il contenuto dei registri del processore, in modo che lo si possa ripristinare al momento di ritornare al programma chiamante.

10.3 Stop e Nop

L'istruzione:

- STOP *im*

carica il valore *im* nel registro di stato SR, incrementa PC in modo che esso punti alla istruzione successiva, e pone il processore in uno stato di attesa di un interrupt. Quando un device esterno produce una richiesta di interruzione, il processore serve l'interruzione e, successivamente, prosegue eseguendo l'istruzione seguente la STOP.

L'istruzione No-Operation:

- NOP

non compie alcuna operazione in fase execute. L'esecuzione dell'istruzione NOP richiede comunque che il processore carichi l'istruzione dalla memoria in fase fetch. NOP è talvolta usata quando si effettua il debugging di un codice oggetto e si vuole eliminare un'istruzione senza dover ricompilare l'intero codice: un'istruzione lunga *n* word è sostituita con *n* istruzioni NOP.

10.4 Istruzioni LINK e UNLK

Perché un sottoprogramma possa invocare ricorsivamente se stesso, occorre che sia i parametri effettivi, sia le variabili locali siano allocate in un'area di memoria (detta *record di attivazione*, cfr. § F2-IV-VI.1) definita al momento

moria (detta *record di attivazione*, cfr. § F2-IV-VI.1) definita al momento della chiamata al sottoprogramma. Per facilitare la allocazione sullo stack del record di attivazione di un sottoprogramma⁷, i progettisti del 68000 hanno dotato il processore delle istruzioni:

- LINK A,*im*
- UNLK A

L'istruzione LINK A,*im* (dove *im* è un valore immediato negativo o nullo) effettua le seguenti micro-operazioni:

pushm(SP,A);
A := SP;
SP := SP + *im*;

cioè salva il contenuto del registro A sullo stack, carica in A il valore aggiornato dello Stack Pointer, ed infine incrementa SP dell'offset *im*. Sommando il valore negativo *im* al contenuto di SP, l'istruzione LINK riserva un'area di memoria di *im* byte sulla cima dello stack. Quest'area è utilizzata per l'allocazione delle variabili locali del sottoprogramma. La figura 10.1 presenta lo stato dello stack prima e dopo l'esecuzione dell'istruzione LINK A1,#-8.

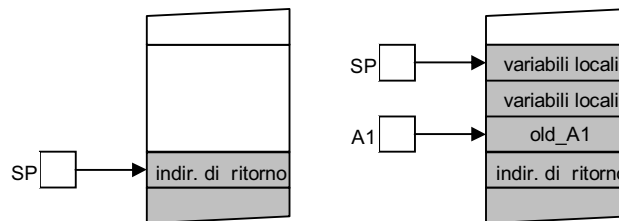


Figura 10.1 Stato dello stack prima e dopo l'esecuzione di LINK A1,#-8

L'istruzione UNLK A ripristina il contenuto del registro A e lo stato dello stack, mediante la sequenza di micro-operazioni:

SP := A;
popm(SP,A);

L'area di memoria allocata mediante l'istruzione LINK è acceduta all'interno del sottoprogramma mediante indirizzamento indiretto con displacement attraverso il registro A, che svolge la funzione di puntatore al record di attivazione (de-

⁷ Quando si adotta uno schema di allocazione sullo stack, il record di attivazione è di solito indicato nella letteratura tecnica in lingua inglese con il termine *stack frame*.

nominato BP in § F2-IV-VI.3, ed in altri testi indicato con il termine *frame pointer*).

10.4.1 Esempio d'uso di LINK, UNLK: chiamata di sottoprogrammi

L'utilizzo dello stack per il passaggio dei parametri effettivi ad un sottoprogramma è una forma di allocazione dinamica usata sia da programmatori assembler che dai compilatori di linguaggi di alto livello come il Pascal ed il C. In questo paragrafo si illustra, mediante un esempio, una possibile convenzione per il passaggio di parametri. Tale convenzione assegna al chiamante la responsabilità di allocare sullo stack lo spazio richiesto dai parametri di output e nel caricare successivamente sullo stack i valori dei parametri di input (Fig. 10.2a). Dopo l'esecuzione dell'istruzione di salto a sottoprogramma (JSR o BSR), il valore dell'indirizzo di ritorno è in cima allo stack (Fig. 10.2b). Il sottoprogramma crea uno *stack frame* mediante l'istruzione LINK, che inizializza un *frame pointer* (il registro A1 nell'esempio in figura) ed alloca sullo stack spazio per le variabili locali (Fig. 10.2c). Il sottoprogramma può accedere ai parametri di scambio ed alle variabili locali mediante indirizzamento indiretto attraverso il *frame pointer*.

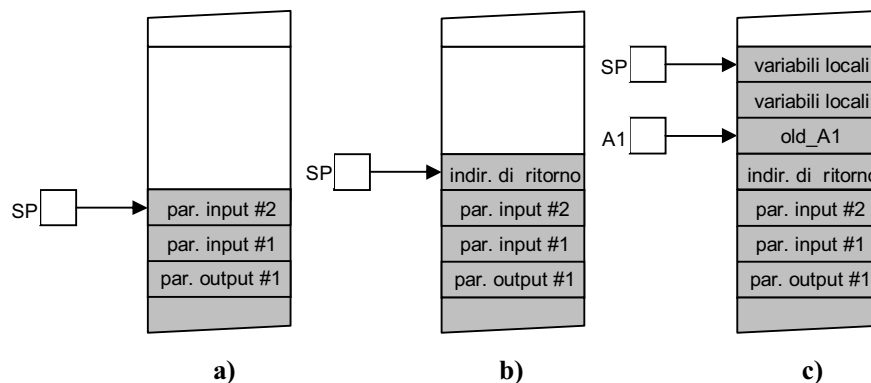


Figura 10.2 Una convenzione per il passaggio di parametri sullo stack

Il programma in figura 10.3 illustra l'utilizzo delle istruzioni LINK e UNLK per la gestione del record di attivazione. Si noti l'utilizzo di valori di spiazzamento positivi per l'accesso ai parametri di scambio di ingresso/uscita. Eventuali variabili locali del sottoprogramma (non presenti nell'esempio in figura) dovrebbero essere allocate nell'area al di sopra del *frame pointer*, e quindi sarebbero accedute mediante displacement a valori negativi.

46 Architettura, linguaggio macchina e assembler 68000

```

* Calcola Z = X^Y mediante sottoprogramma POWR
*****
*
* Programma principale
MAIN      ORG          $1000
          ADDA.L      #-2,SP          riserva 2 byte per Z
          MOVE        Y,-(SP)        push esponente Y (word)
          MOVE        X,-(SP)        push base X (word)
          JSR         POWR           chiama subroutine POWR
          ADDQ        #4,SP          rimuovi X e Y dallo stack
          MOVE        (SP)+,Z        copia parametro di output
          JMP         $8008          ritorna al sist.operativo

* Allocazione variabili X, Y e Z
X          DC.W        3
Y          DC.W        4
Z          DS.W        1
*****
*
* Subroutine POWR: calcola A^B
* Definizione di costanti di spiazzamento per l'accesso
* ai parametri di scambio e alle variabili locali
OLD_FP    EQU          0
RET_ADDR  EQU          4
A          EQU          8          spiazzamento base
B          EQU          10         spiazzamento esponente
C          EQU          12         spiazzamento risultato
POWR      LINK         A6,#0        usa A6 come frame pointer
          MOVEM.L      D0-D2,-(SP)  salva registri su stack
          MOVE        A(A6),D0      copia A in D0
          MOVE        B(A6),D1      copia B in D1
          MOVE.L       #1,D2        usa D2 come accumulatore
LOOP      SUBQ         #1,D1        decrementa B
          BMI.S        EXIT         if D1-1<0 then EXIT
          MULS         D0,D2        moltiplica D2 per A
          BRA         LOOP          e ripeti se necessario
EXIT      MOVE        D2,C(A6)      C:=(D2)
          MOVEM.L      (SP)+,D0-D2  ripristina registri
          UNLK         A6
          RTS
          END          MAIN

```

Figura 10.3 Esempio di passaggio di parametri sullo stack

10.5 Istruzioni di interruzione software (trap)

L'istruzione TRAP è utilizzata per effettuare chiamate ai servizi del sistema operativo, o, in generale, quando si vuole realizzare un passaggio dal modo utente al modo supervisore (non è infatti possibile settare direttamente $S=1$).

In particolare, l'istruzione:

- TRAP *im*

è simile ad una chiamata a subroutine, con le seguenti differenze:

- il processore è posto in stato supervisore ($S=1$);
- il valore corrente del PC è salvato sullo stack di sistema (puntato da SSP) invece che sullo stack di utente;
- il valore corrente di SR è anch'esso salvato in cima allo stack di sistema;
- l'indirizzo della subroutine a cui saltare è contenuto in memoria in una *tabella dei vettori delle eccezioni*, allocata in memoria agli indirizzi $[128_{10}, 191_{10}]$; il processore usa l'operando immediato *im* (che è un numero compreso tra 0 e 15) come un indice in tale tabella:

$$PC \leftarrow M[128 + 4 * im]$$

L'istruzione:

- TRAPV *im*

è simile a TRAP, con la differenza che l'interruzione è abilitata se e solo se è: $V=1$.

L'istruzione:

- ILLEGAL

è simile a TRAP, ma produce il caricamento in PC dell'indirizzo a 32 bit contenuto in memoria nella locazione 16_{10} (*Illegal instruction trap*).

L'istruzione:

- CHK G,D

lavora su operandi di tipo word: essa genera un'eccezione se il contenuto del registro D è fuori dell'intervallo $[0,G]$. L'indirizzo della subroutine di gestione dell'eccezione generata da CHK è prelevato in memoria all'indirizzo 24_{10} .

Infine, l'istruzione:

- RTE

è utilizzata per terminare l'esecuzione di una subroutine di gestione di un'interruzione. Se il processore si trova in stato supervisore, ripristina i valori

dei registri SR e PC, prelevandoli dalla cima dello stack, altrimenti genera un'eccezione.

10.6 Istruzioni di input/output

Il processore 68000 è dotato di una linea fisica di uscita, detta reset, la cui funzione è quella di inizializzare opportunamente tutti i dispositivi esterni connessi al processore. Tale linea può essere attivata dal processore mediante l'istruzione RESET.

Il processore è inoltre dotato di un'istruzione per il trasferimento dati da e verso i registri dell'interfaccia di una periferica:

- MOVEP M,D per operazioni di trasferimento dall'interfaccia
- MOVEP D,M per operazioni di trasferimento verso l'interfaccia

Questa istruzione è stata specificamente progettata per facilitare l'accesso a periferiche i cui registri di interfaccia sono memorizzati in byte alternati allocati in word consecutive di memoria⁸, secondo lo schema esemplificato in fig. 10.4.

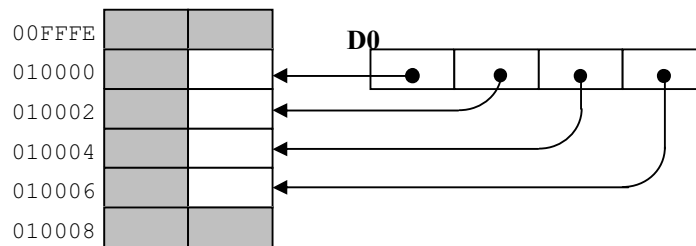


Figura 10.4 Effetto dell'istruzione MOVEP.L D0,0(A0)

L'operando memoria dell'istruzione MOVEP può essere indirizzato esclusivamente mediante indirizzamento indiretto con displacement. Nel caso di figura 10.4, il registro indirizzo A0 è stato preventivamente caricato con l'indirizzo base dell'interfaccia, cioè con il valore \$10001. L'istruzione MOVEP.L D0,0(A0) carica nei quattro registri dell'interfaccia (posti agli indirizzi \$010001, \$010003, \$010005 e \$010007) i valori contenuti nei quattro byte in cui si suddivide il registro D0 (a partire dal byte più significativo). Un comportamento analogo ha l'istruzione MOVEP quando è usata per trasferire dati da un'interfaccia verso un registro D.

⁸ Un tale schema di allocazione dei registri di interfaccia è effettivamente adottato da alcuni dispositivi di controllo di periferiche prodotti dalla Motorola.

L'istruzione MOVEP è stata concepita per dispositivi di input/output ad 8 bit, collegati al bus dei dati del processore 68000. Tali dispositivi possono essere collegati secondo uno dei due schemi di Figura 10.5 a) e b). Nel caso in cui il dispositivo sia collegato alle otto linee meno significative ($d_0 \dots d_7$) del bus dati del processore, come in Fig. 10.5 a), i registri di interfaccia sono associati ad indirizzi dispari (come nel caso illustrato in Fig. 10.4). Viceversa, nel caso in cui il dispositivo sia collegato alle otto linee più significative ($d_8 \dots d_{15}$) del bus dati, come in Fig. 10.5 b), i registri di interfaccia sono associati ad indirizzi pari.

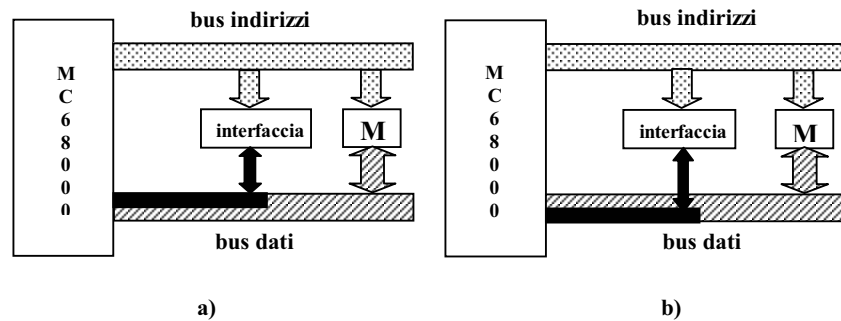


Figura 10.5 Connessione di un dispositivo di I/O ad 8 bit al processore 68000

Le direttive di assemblaggio di ASM 68000

Premessa

In questo capitolo si descrivono le direttive riconosciute dal linguaggio assembler che la Motorola ha definito per il proprio processore MC68000.

Program Location Counter e direttiva ORG

Per definire gli indirizzi (relativi) di memoria nei quali saranno allocati programma e dati, il programma assembler gestisce una variabile interna a 32 bit, detta *Program Location Counter* (PLC), il cui valore è incrementato ogni volta che un'istruzione è tradotta, di un numero pari alla dimensione in byte del codice macchina dell'intera istruzione.

Una apposita pseudo-istruzione, direttiva ORG (Set Origin), è utilizzata per inizializzare o per assegnare un valore a PLC. Il valore è espresso come operando della direttiva, secondo la seguente sintassi:

ORG *indirizzo*

ove indirizzo è un numero (costante o espressione).

Ad esempio:

ORG \$3000

nel significato: la prima istruzione che segue avrà l'indirizzo \$3000.

ORG INDIRIZZO_BASE+200

nel significato: la prima istruzione che segue avrà l'indirizzo della variabile INDIRIZZO_BASE incrementato di 200.

Il simbolo speciale "*" sta ad indicare il valore del Program Location Counter e può essere utilizzato nelle espressioni. Ad esempio:

ORG *+\$1000

nel significato: la prima istruzione che segue avrà un indirizzo pari a quello corrente di PLC incrementato di \$1000 o, in altri termini, lascia \$1000 locazioni di memoria libere.

La direttiva ORG è tipicamente presente all'inizio del codice sorgente, per definire l'indirizzo base a partire dal quale il codice macchina verrà caricato in memoria. Altre direttive ORG possono essere utilizzate all'interno del programma (ad esempio per allocare i dati in una zona della memoria separata dall'area in cui è allocato il codice).

Dichiarazione di dati: le direttive DS e DC

Per introdurre in un programma una (o più) variabili, dando ad essa un nome ed eventualmente assegnando loro un valore iniziale definito già al tempo dell'assemblaggio si usano le direttive DS (*Define Storage*) e DC e DC (*Define Constant*).

Preliminarmente si ricorda che il 68000 gestisce dati di tipo Byte (.B), Word (.W, tipo di default) e Longword (.L) e si pone in evidenza un vincolo della sua architettura: gli indirizzi dei dati di tipo W e L debbono essere pari (l'indirizzo è calcolato in byte). In altri termini, un dato che sia più lungo di un byte deve iniziare ad un indirizzo pari. Le direttive in esame realizzano automaticamente tale vincolo "allineando" l'indirizzo mediante eventuale incremento di 1 unità a PLC.

2.1 Direttiva DS

La direttiva DS (*Define Storage*) definisce il nome di una variabile, le assegna una locazione ed uno spazio di memoria. Ad esempio, la dichiarazione

ALFA DS.B 1

introduce una variabile di nome ALFA, di tipo byte, posta in memoria all'indirizzo PLC. A tale scopo ASM compie le seguenti azioni:

- 1) immettere in tabella simboli la coppia (ALFA, valore di PLC);
- 2) incrementa PLC di 1 (dimensione in byte del dato immesso)

Più in generale, la dichiarazione si presenta nella forma:

[etichetta]	DS[[.L] [.W] [.B]]	espressione

ed ASM preliminarmente allinea PLC ad un indirizzo pari (in caso di W o L) e quindi incrementa PLC di una quantità pari al valore specificato nel campo operando (come costante o espressione numerica) moltiplicata per 1, 2 o 4, a seconda che si tratti di dato di tipo B, W o L.

Si noti che di norma l'etichetta dell'istruzione è non-vuota, anche se sintatticamente potrebbe esserlo. In realtà, un'etichetta vuota viene usata se si vuole usare la dichiarazione soltanto per allineare PLC ad indirizzo pari:

DS.W 0

2.2 Direttiva DC

La direttiva DC ha un effetto analogo a DS, ma, contrariamente a quest'ultima, inizializza l'area di memoria riservata con la dichiarazione. In altri termini, essa è usata per allocare in memoria sequenze di valori costanti o stringhe di caratteri. La sintassi della direttiva è:

[*etichetta*] DC[*[.L]* | *[.W]* | *[.B]*] <elemento>
[[,<elemento>]...,<elemento>]]

ove il singolo <elemento> può essere un'espressione o una stringa di caratteri. Si noti che mentre con la direttiva DS la dimensione dell'area riservata è definita dal numero presente nel campo-operandi, con la DC è il numero degli elementi della lista che lo definisce.

Come esempio, si consideri il seguente frammento di codice assembly:

```
1:          ORG      $1000
2:          DC.W     $1, $2, $3
3:          DC.L     $FF
4:      ST1      DC.B   'Hello'
5:      ST2      DC.B   'world', 0
6:          DC.B     $4
7:          DC.B     $5
8:          DC.W     $6
```

ASM produce un codice oggetto che, quando sarà caricato in memoria, darà luogo alla situazione illustrata nella figura seguente:

0000 1000	00	01	00	02
0000 1004	00	03	00	00
0000 1008	00	FF	48	65
0000 100C	6C	6C	6F	77
0000 1010	6F	72	6C	64
0000 1014	00	04	05	00
0000 1018	00	06	00	00

Osservando la figura precedente, è possibile fare le seguenti considerazioni:

- le costanti numeriche sono rappresentate su 1, 2 o 4 byte, a seconda del suffisso usato nella direttiva DC;
- le stringhe di caratteri sono rappresentate come sequenze di byte; ciascun byte ha il valore del codice ASCII corrispondente ad un carattere (linee 4 e 5 del programma);
- per ottenere stringhe terminate dal carattere con codice ASCII 0, come quelle utilizzate dalle librerie C standard, occorre allocare esplicitamente la costante 0 nel byte successivo a quello in cui è memorizzato l'ultimo carattere della stringa (linea 5);
- se si usa una direttiva DC.W o DC.L per allocare una stringa, verranno eventualmente allocati degli zeri in coda alla stringa, in modo da rendere la dimensione allocata un multiplo di 2 o di 4 byte rispettivamente;
- la direttiva DC.B alloca un valore (o una sequenza di valori) a partire dall'indirizzo corrente del Program Location Counter, anche se quest'ultimo ha un valore dispari (linee 6 e 7). Di contro, le direttive DC.W e DC.L, nel caso in cui il PLC abbia un valore dispari, ne incrementano il valore di 1 (linea 8)

La chiusura statica del programma: direttiva END

La direttiva END è utilizzata per contrassegnare la fine del sorgente assembly. ASM ignora tutto ciò che si trova nel file sorgente dopo la linea contenente questa direttiva. In ASM68K, alla direttiva END deve essere associato come operando l'*entry point* del programma (ovvero l'indirizzo della prima istruzione da eseguire).

Uso di file esterni per l'assemblaggio: direttiva INCLUDE

Così come avviene in linguaggi ad alto livello, una buona organizzazione del lavoro di programmazione ed assemblaggio consiste nel riusare in programmi diversi i medesimi file già usati altrove. A tale scopo si usa la direttiva

INCLUDE File_Esterno

Essa fa sì che ASM prosegua il processo di assemblaggio a partire dalla prima linea del codice sorgente di File_Esterno (espresso mediante un path). Quando l'assemblatore raggiunge la fine del file esterno, continua il processo di assem-

blaggio del file sorgente originario, a partire dalla linea successiva alla direttiva INCLUDE.

Altre direttive

5.1 Direttiva EQU

La direttiva EQU (*Equate Symbol*) è utilizzata per definire un simbolo che sostituisca per ASM un valore.

etichetta EQU *espressione*

La stringa utilizzata come etichetta è il nome a cui l'assemblatore associa il valore specificato come operando della direttiva. Ad esempio:

```
                ORG    $1000
ENTRY1 EQU     *
```

La direttiva EQU assegna al simbolo ENTRY1 il valore esadecimale \$1000.

Si noti che:

- il valore del simbolo introdotto con la direttiva EQU non può essere alterato durante l'assemblaggio;
- l'espressione operando di EQU non può contenere riferimenti in avanti (cioè simboli definiti in linee di programma seguenti).

5.2 Direttiva SET

La direttiva SET (*Set Symbol*) è simile ad EQU, ma consente di modificare il valore del simbolo mediante una ulteriore direttiva SET:

etichetta SET *espressione*

5.3 Direttiva REG

La direttiva REG (Register Range) associa ad un simbolo una lista di registri. Tale simbolo potrà essere utilizzato nel resto del programma sorgente come operando per l'istruzione MOVEM, per operazioni di load e store multiple. L'operando di questa direttiva consiste in una lista di nomi di registri (Registri Dato o Registri Indirizzo) separati dal carattere '/'. Ciascun elemento della lista può essere o un singolo registro, o una sequenza di registri individuata mediante il primo e l'ultimo registro separati dal carattere '-'. Registri e sequenze di registri possono essere specificati secondo un ordine qualunque. Un esempio d'uso della direttiva è il seguente:

