ASSEMBLER 8086 8088

di Pierluigi Panunzi

Il set di istruzioni
Istruzioni di trasferimento dati

A questo gruppo appartengono le istruzioni che consentono di effettuare un generico trasferimento di dati da un registro ad un altro, dalla memoria ad un registro, da un registro alla memoria, dalla memoria alla memoria (in due soli casi!) ed infine da e verso un dispositivo periferico di I/O.
In particolare analizzeremo 14 istruzioni decisamente utili ed utilizzate effettivamente parecchio all'interno di un qualsiasi programma.
La parte del leone la fa l'istruzione «MOV», che abbiamo già incontrato per forza di cose parecchie volte nelle scorse puntate: questa volta formalizzeremo tutte le varie possibilità di utilizzo di tale istruzione, come pure (ovviamente) delle altre.

L'istruzione MOV

Si tratta, come è facilmente intuibile, dell'istruzione che consente il passaggio di un dato ad 8 o a 16 bit da una sorgente verso una destinazione, secondo la sintassi:

```
MOV dest, source
```

dove tanto «dest» quanto «source» possono essere abbinati in undici differenti modi.

In particolare abbiamo riportato in tabella le possibili combinazioni.

Come abbiamo già visto la scorsa puntata, i termini indicati nella tabella qui a fianco hanno il seguente significato:

«memoria»: indica una qualsiasi locazione di memoria indirizzata direttamente o indirettamente o tramite un registro base o tramite un registro indice, secondo le ben note possibilità;

«accumulatore»: è il registro AX per trasferimento dati a 16 bit oppure AL o AH per dati ad 8 bit;

«segment reg.»: altro non è che uno dei registri di segmento (CS, DS, ES e SS) che in questo caso vengono trattati con un tipo di codifica differente che non i registri normali;

Istruzione MOV

	Destinazione	Sorgente
1)	memoria	accumulatore
2)	accumulatore	memoria
3)	segment reg.	memoria
4)	segment reg.	registro
5)	memoria	segment reg.
6)	registro	segment reg.
7)	registro	registro
8)	registro	memoria
9)	memoria	registro
10)	registro	immediato
11)	memoria	immediato

«registro»: è appunto uno qualsiasi dei rimanenti registri, sia i «general purpose» (BX, CX, CH, DL e DH se i dati sono ad 8 bit), che i registri indice (DI e SI), nonché i registri BP ed SP.

«immediato»: è un qualsiasi valore numerico ad 8 o a 16 bit a seconda di quanto richieda l'istruzione.

In particolare per ognuno di questi undici possibili tipi di combinazione sorgente-destinazione riportiamo un esempio:

```
1)   MOV WORD1,AX
     MOV BYTE1,AL
     MOV DELTA[BX][SI+4],AH

2)   MOV AL,ALFA
     MOV AH,ALFA[DI]
     MOV AX,PAROLA+22

3)   MOV DS,DATASEG
     MOV ES,ALFA[BP]

4)   MOV SS,CX
     MOV DS,DI

5)   MOV EXTRASEG,ES
     MOV ZETA[BX][SI+9],CS

6)   MOV DX,DS
     MOV AX,SS
     MOV SI,CS

7)   MOV CL,BH
     MOV SI,BX

8)   MOV DI,ALFA[BX][DI]
     MOV SI,[SI]

9)   MOV BETA+3,DH
     MOV [DI],BX

10)  MOV CX,2000H
     MOV SI,PUNTATORE

11)  MOV CELLA[BX][DI+9],0
     MOV [BX+2],1234H
```

Dal momento che l'istruzione MOV non fa altro che spostare un dato da una sorgente ad una destinazione ecco che tutto sommato non c'è molto altro da dire, se non che in qualunque caso i flag non vengono alterati (anche se ad esempio si carica 0 in un registro il flag di Zero non viene minimamente toccato).

C'è da aggiungere invece che nel modo 3 di combinazione non è ammesso come destinazione il registro di segmento CS in quanto a pensarci bene non è chiaro a cosa possa servire, se non a complicarsi la vita, dal momento che (lo ricordiamo) il registro CS può essere alterato, ma in maniera completamente automatica e trasparente (invisibile, cioè) ad esempio con un salto inter-segment. Viceversa si può (modi 5 e 6) scrivere il contenuto di CS in una locazione di memoria o in un registro e ciò è molto comodo per caricare ad esempio il DS per poter indirizzare variabili poste nel Code Segment: questo accade parecchie volte scrivendo programmi che girano in ambiente MS-DOS, laddove è comodo porre variabili nel Code Segment invece di allocarle in un Data Segment, il tutto per problemi di rilocabilità per cui non si sa dove viene allocato poi fisicamente il Data Segment all'atto del caricamento in memoria del file.

L'istruzione XCHG

L'istruzione XCHG (che trae il nome da «eXCHanGe», scambia) ha il compito appunto di scambiare tra loro i contenuti della destinazione e della sorgente, che a rigor di logica ora hanno perso la loro identità di «sorgente» e di «destinazione» dal momento che si tratta di un semplice scambio («swap»).

Comunque, mantenendo la notazione solita, la sintassi dell'istruzione è la seguente:

```
XCHG dest, source
```

dove ora le combinazioni possibili si sono enormemente ridotte a quelle riportate in tabella

Istruzione XCHG

Destinazione	Sorgente
1) registro	accumulatore
accumulatore	registro
2) registro	memoria
memoria	registro

Riportiamo come al solito alcuni esempi molto semplici:

```
1) XCHG BX, AX
   XCHG AX, SI
2) XCHG ALFA[BX] [SI], SI
   XCHH AL, [DI]
   XCHG [DI], DI
```

In particolare c'è da aggiungere che nel modo 1 l'accumulatore può essere soltanto AX e cioè l'istruzione di XCHG (che in questo caso è ad un solo byte) lavora solo sui registri a 16 bit: comunque nel modo 2 si possono viceversa lavorare anche con registri ad 8 bit ed in particolare AL ed AH. Classica a questo punto è l'istruzione

```
XCHG AL, AH
```

che però, come le altre possibili istruzioni del modo 2, ha un opcode a due byte.

Anche in questo caso, l'istruzione XCHG non altera in alcun modo i flag.

Le istruzioni XLAT e XLATB

Si tratta in pratica della stessa istruzione (che ha comunque lo stesso opcode, per la cronaca D7H) e trae il suo nome dal verbo «translate» in quanto consente di trasformare il valore contenuto in AL nel valore proveniente da una tabella di 256 valori.

In particolare il valore originario di AL serve come indice all'interno della tabella di 256 byte ed il valore estratto viene posto in AL: l'indirizzo di memoria della tabella deve essere preventivamente caricato nel registro BX.

La sintassi delle due istruzioni è la seguente:

```
XLAT table
XLATB
```

ma come detto si tratta di un'unica istruzione: in particolare la dualità nasce dal fatto che la XLATB è stata aggiunta in un secondo tempo, mentre dunque all'inizio era necessario indicare (ma solo per l'assemblatore) l'indirizzo della tabella («table»).

Ora dato che nella codifica dell'istruzione non compare l'indirizzo della tabella, allora basta usare l'istruzione XLATB per semplicità in quanto in entrambi i casi è comunque BX che deve contenere l'indirizzo della tabella.

Con questa istruzione si può realizzare dunque la gestione di una cosiddetta «lookup table», in parole povere un vettore di 256 byte al quale si accede con indice dato da AL.

Un esempio di uso di tale istruzione può essere il seguente:

```
MOV BX, OFFSET TABELLA
XLATB
```

e se ad esempio TABELLA contiene i valori

```
indice → 0 1 3 4 5 6 7 8 9
TABELLA → 5 7 22 33 120 45 1 1 4 ecc.
```

con AL inizialmente caricato a 5, si otterrà in AL, per effetto della XLATB, il valore 45.

Anche in questo caso l'istruzione XLATB non altera i flag.

L'istruzione LEA

Questa istruzione trae il nome da «Load Effective Address» e serve a caricare nel registro indicato come destinazione dell'istruzione stessa, l'indirizzo effettivo della locazione di memoria posta come sorgente: la sintassi dell'istruzione è la seguente

```
LEA dest, source
```

dove si ha una sola possibilità data da

Istruzione LEA

Destinazione	Sorgente
registro	offset di cella di memoria

In particolare è utile quando la memoria è indirizzata tramite registri indice e/o base

```
LEA DX, ALFA [BX] [SI+3]
```

oppure nei casi in cui non compare l'etichetta di una variabile, come nel caso

```
LEA SI, [BP] [SI]
```

che permetterà di caricare in SI l'indirizzo ottenuto dalla somma dei valori attuali di BP e di SI, il tutto al momento dell'esecuzione (si badi bene!).

Anche in questo caso i flag non vengono alterati.

Le istruzioni LDS e LES

Si tratta in questo caso di due particolari istruzioni che gestiscono quantità a 32 bit ed in particolare un puntatore posto in memoria sotto forma di OFFSET (nella word meno significativa) e di SEGMENT (nella word più significativa): le due istruzioni in esame consentono di trasferire tale puntatore nella coppia di registri «DS: registro qualunque» (per la LDS) oppure «ES: registro qualunque» (per la LES).

Per effetto dell'esecuzione di una delle due istruzioni, l'offset contenuto nella word più bassa nella memoria andrà nel registro posto come destinazione, mentre il valore del segmento posto nella word più alta nella memoria andrà a finire in DS con la LDS ed in ES con la LES.

Detto che la sintassi delle due istruzioni è la seguente

```
LDS dest, source
LES dest, source
```

i parametri «dest» e «source» possono essere solo del tipo indicato in tabella

Istruzione LDS LES	
Destinazione	**Sorgente**
registro	offset di una double-word

Ad esempio se

```
ALFA DW 1000H
     DW 2222H
```

con l'istruzione

```
LDS BX, ALFA
```

si caricherà automaticamente in BX il valore 1000H ed in DS il valore 2222H.

Ovviamente nei programmi in cui viene usata la LES e/o la LDS, in ALFA non ci saranno dei valori fissi, ma calcolati, ad esempio gli entry point di una serie di routine a cui bisognerà poi saltare.

Inutile dire che anche nel caso delle due istruzioni or ora esaminate i flag non vengono assolutamente toccati.

Le istruzioni di gestione dello stack: PUSH e POP

Riteniamo di nessuna utilità parlare ancora una volta di che cosa è lo stack e di come si gestisce: è oramai un concetto che i programmatori devono avere ben presente in mente.

L'unica cosa che interessa sapere a questo livello è che lo stack pointer (SP) in ogni istante punta sempre alla word «affiorante» dallo stack e cioè all'ultima word inserita, e non come nel caso di altri microprocessori alla prima cella «vuota» sulla quale depositare un valore.

Tenendo dunque ciò in mente, l'esecuzione di una PUSH decrementerà innanzitutto di 2 lo Stack Pointer (per poter puntare ad una word vuota) e poi successivamente salverà in tale cella l'operando posto nell'istruzione.

Viceversa l'istruzione POP estrarrà innanzitutto la word puntata da SP e la deporrà nell'operando destinazione o poi incrementerà di 2 lo Stack Pointer.

Non dimentichiamoci che lo stack «cresce», «evolve» verso indirizzi decrescenti di memoria e si «svuota» andando verso indirizzi sempre più alti!!!

Veniamo dunque alla sintassi delle due istruzioni

```
PUSH source
POP dest
```

dove in questo caso possiamo (data la similitudine «speculare» delle due istruzioni) accorpare «source» e «dest» in un unico «operando» per vedere nella tabella le varie possibilità

Istruzioni PUSH e POP
Operando
1) registro 2) segment reg. 3) memoria

Ecco dunque alcuni esempi, se vogliamo banali, di istruzioni contenenti PUSH e POP:

```
1) PUSH AX
   POP DI
2) PUSH SS
   PUSH CS
   POP BX
3) PUSH BETA [SI]
   PUSH [BX]
   POP GAMMA
   POP [SI+7]
```

C'è da notare il fatto, riportato negli esempi del modo 2, che il registro CS può essere salvato nello stack, ma non può essere ripristinato dallo stack.

Altro fatto notevole è che il modo 3 (e ciò si vede dagli esempi) consente il trasferimento di dati da memoria a memoria in quanto a pensarci bene lo stack (sia in «estrazione» che in «inserimento») altro non è che la memoria e non un registro interno: è questo un caso alquanto raro di istruzioni che trasferiscono dati da memoria a memoria.

Comunque per l'8086/8088 esiste tutta una serie di istruzioni di gestione di stringhe che effettuano trasferimenti da memoria a memoria e sulle quali ritorneremo, ma altri microprocessori non hanno questa possibilità: tornando «indietro» allo Z80, ad esempio, il codice operativo che avrebbe potuto effettuare un trasferimento dalla memoria alla memoria è addirittura stato usato per implementare l'istruzione di HALT.

Infine ancora una volta le istruzioni di PUSH e di POP non alterano i flag.

Le istruzioni di Input/Output: IN e OUT

Come ogni bravo microprocessore, anche l'86/88 possiede una coppia di istruzioni che consentono l'I/O da e verso porte. In particolare il trasferimento può avvenire per quantità ad 8 o a 16 bit ed in entrambi i casi coinvolge l'accumulatore (AL per dati ad 8 bit ed AX per dati a 16 bit).

La sintassi delle due istruzioni è la seguente

```
IN accumulator, port
IN accumulator, DX
OUT port, accumulator
OUT DX, accumulator
```

ed in questi casi «accumulator» è come detto AL o AX a seconda se si desidera trasferire byte o word (e ciò è legato strettamente all'hardware del sistema) e «port» è il numero di una porta di I/O, numero il cui valore deve essere compreso tra 00H ed FFH (da 0 a 255).

Dal momento che però lo spazio di I/O del microprocessore è esteso da 0000H a FFFFH (da 0 a 65535) ecco che, dovendo indirizzare porte di indirizzo maggiore di 255, si deve usare il registro DX come supporto dell'indirizzo della porta e perciò usare le istruzioni che coinvolgono DX.

A parte questioni hardware, l'assembler si aspetta, nell'istruzione coinvolgente l'accumulatore, l'indicazione di AL o AX in modo da generare il corretto opcode.

Vediamo alcuni esempi, dove PORTA vale 23H e PORTAW vale 3F8H, in entrambi i casi grazie ad una EQU:

```
IN AL, PORTA
IN AX, PORTA+1
OUT 128, AL
OUT PORTA+7, AX
MOV DX, PORTAW
IN AL, DX
MOV DX, 0F000H
OUT DX, AX
```

In questi esempi si vede come il numero della porta può essere inserito nell'istruzione sia come valore immediato, sia come «literal» e cioè con un simbolo (ad esempio PORTA) che è stato inizializzato con una EQU.

La forma più generica dell'istruzione di I/O e cioè quella che coinvolge il registro DX, può essere usata anche per valori della porta di I/O minori di 256: ad esempio si può scrivere sia

```
PORT EQU 10H
...
IN AL, PORT
```

sia

```
MOV DX, 1OH
IN AL, DX
```

nel quale ultimo caso si hanno parec-

chi byte in più nella codifica, mentre però si ha un guadagno se si deve accedere a porte di I/O di valori ad esempio consecutivi, alle quali si può accedere dall'interno di un loop.

E con questa coppia di istruzioni terminiamo lo studio delle istruzioni che non alterano i flag.

Le istruzioni coinvolgenti i flag: LAHF, SAHF, PUSHF e POPF

A questo ultimo gruppo di istruzioni fanno parte due coppie di istruzioni «speculari» che hanno a che vedere con i flag. Rispettivamente si tratta della «Load AH with Flags», «Save AH to Flags», «PUSH Flags» e «POP Flags».

La prima carica il contenuto del registro dei flag nel registro AH, avendosi la seguente disposizione all'interno del byte:

bit	7	6	5	4	3	2	1	0
	S	Z	X	A	X	P	X	C

dove «S» è il flag di Segno, «Z» è il flag di Zero, «A» è l'Auxiliary flag, «P» è il flag di parità, «C» è il carry e «X» indica un bit che può essere indifferentemente 0 o 1. Evidentemente anche questa istruzione non altera i flag, anche perché se lo facesse non si avrebbe più in AH lo stato corrente dei flag!

Invece l'istruzione SAHF carica il registro dei flag con il contenuto del registro AH, secondo la codifica dei bit valida per l'istruzione precedente.

In questo caso i flag vengono (finalmente!) alterati (e vorrei vedere se non fosse così!!!) ed in particolare i bit indicati con «X» possono essere in origine posti indifferentemente a 0 o ad 1, tanto vengono ignorati.

Come si vede in entrambi i casi vengono gestiti solo i cinque bit che l'8086/88 aveva ricevuto in eredità dall'8080 e dall'8085.

Le ultime due istruzioni viceversa consentono la gestione di tutti i bit, che come sappiamo, sono gestiti all'interno del Flag Register a 16 bit.

L'istruzione PUSHF agisce come una PUSH, solo che salva nello stack il contenuto del Flag Register e viceversa la POPF ripristina il Flag Register con il contenuto della parola affiorante dallo stack.

In entrambi i casi la codifica dei bit all'interno della word dello stack coincide con quella del Flag Register ed è data da:

▼

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

dove, oltre ai flag già visti, «O» è il flag di Overflow, «D» è il Direction Flag, «I» è l'Interrupt Flag e «T» è il Trap Flag.

Anche in questo caso i bit segnati con «X» sono «don't care» e perciò possono essere indifferentemente 0 o 1 ed ignorati.

Con questo abbiamo terminato l'analisi delle istruzioni di trasferimento dati: la prossima puntata ci occuperemo delle istruzioni logiche (shift, rotate, and, or, ecc).

MC