di Pierluigi Panunzi

Anche i microprocessori hanno i loro «compatibili»

seconda parte

Dopo aver visto nella prima parte le istruzioni che il V20 ha in più rispetto all'8086/88 e che condivide con i modelli superiori (80186, 80286 e 80386), in questa seconda parte parleremo di istruzioni completamente nuove, che potranno essere usate solamente in PC dotati di V20: per un AT e cioè il personal dotato di 286 c'è il rischio di incappare in errori del tipo «invalid opcode», mentre nei computer dotati di 386, oltre a questo rischio ce n'è un altro dovuto al fatto che alcune istruzioni nuove del V20 condividono il codice operativo con altre istruzioni (completamente differenti) introdotte con il 386

Le novità nel set di istruzioni del V20

Abbiamo dunque detto delle istruzioni nuove: in dettaglio si tratta di:
— due nuove istruzioni di controllo per la gestione delle stringhe
— due istruzioni che permettono di gestire (inserire ed estrarre) un certo numero di bit da locazioni di memoria
— cinque istruzioni che facilitano la gestione di quantità espresse in BCD e che lavorano sui «nibble»
— quattro istruzioni che permettono la gestione dei singoli bit di byte, word, ecc. (era ora! Ma tanto ne ritroveremo di simili nel 386...)
— alcune istruzioni che consentono il passaggio dal «modo nativo» al modo «emulazione 8080» e viceversa.

Analizziamole dunque in maggior dettaglio.

Le due nuove istruzioni di controllo per la gestione delle stringhe sono

REPC
REPNC

e sono praticamente identiche alle «REPZ» e «REPNZ», con l'unica differenza che testano lo stato del flag di «carry» anziché del flag di «zero»: in particolare (e ciò valeva anche per le REPZ e REPNZ) l'operazione di stringa che deve essere ripetuta deve essere una CMPS (altre istruzioni qui non avrebbero alcun senso).

L'istruzione elementare di comparazione di stringhe, preceduta da una REPC, viene dunque eseguita un numero di volte pari al contenuto del registro CX, a meno che in una qualsiasi delle iterate il flag di «carry» non risulti resettato, nel qual caso l'iterazione termina.

Analogamente nel caso di CMPS preceduta da REPNC, il loop viene eseguito un numero di volte pari al contenuto di CX, a meno che il carry non risulti settato a seguito di una delle operazioni di comparazione.

C'è da dire inoltre che in entrambi i casi viene effettuato sempre prima il controllo del contenuto del registro CX (CW per il V20) e poi quello del flag di carry (subito dopo la prima comparazione): nel caso in cui CX sia nullo proprio prima di eseguire la prima operazione di comparazione di stringhe, allora non viene effettuata nemmeno una comparazione: viceversa non ha alcuna importanza il valore che assume il carry subito prima dell'inizio del ciclo, mentre,

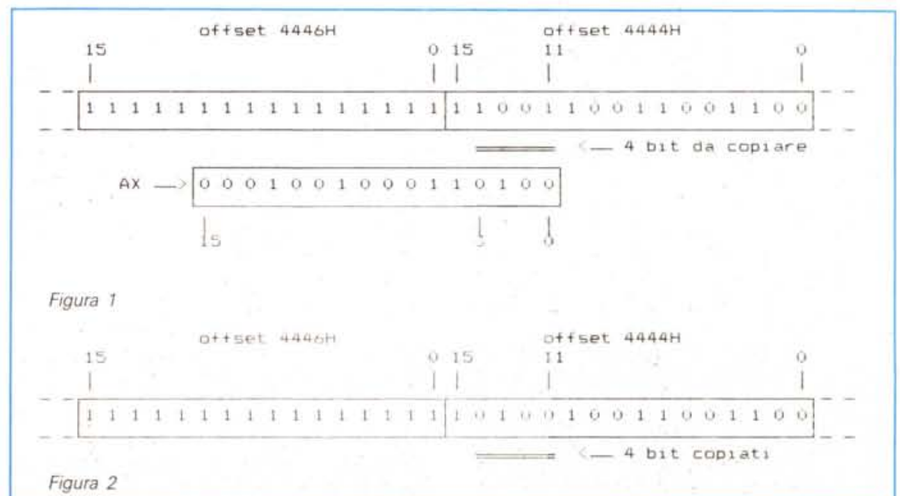

Figura 1

Figura 2

come detto, questo flag viene testato solo dopo la prima comparazione.

Le due istruzioni successive (che permettono di inserire ed estrarre un certo numero di bit da locazioni di memoria) sono veramente nuove: in particolare si tratta delle istruzioni

```
INS reg1,reg2
INS reg1,imm4
EXT reg1,reg2
EXT reg1,imm4
```

Iniziamo dalle «INS» (da non confondere con le omonime dell'Intel, che viceversa si riferiscono all'input multiplo di dati): queste istruzioni servono ad inserire un certo numero di bit (specificato dal contenuto di «reg2» oppure dal valore intero «imm4» entrambi incrementati di 1) del registro AX all'interno della memoria (considerata come una successione di celle, una di seguito all'altra).

Tale cella di memoria è indirizzata per default dalla coppia ES:DI (come per le normali istruzioni di stringa) però è in più possibile specificare (con il contenuto di «reg1») a partire da quale bit (non necessariamente l'LSB, cioè) vogliamo andare a porre i bit di AX.

Al termine dell'istruzione tanto DI quanto «reg1» vengono aggiornati per essere pronti alla successiva INS.

Ma vediamo un esempio dettagliato, corredato di disegno: vogliamo dunque inserire i primi 4 bit di AX (cioè dal bit 0 al bit 3) in una cella di memoria il cui indirizzo è posto in ES:DI, a partire dal bit 11 della cella considerata come word.

Allora (ES e DI già supponiamo contenere valori validi) possiamo caricare ad esempio BL con il valore 3 (il numero di bit di AX da gestire, meno 1) e DH con il valore 11, in quanto è proprio dal dodicesimo bit che vogliamo porre i 4 bit di AX: ciò si ottiene con:

```
MOV AX,qualcosa    ;solo 4 bit ci interessano...
MOV DH,11          ;a partire dal dodicesimo bit...
MOV BL,3           ;solo 4 bit...
INS DH,BL          ;inserisce i bit e aggiorna
```

Supponiamo che:
— AX contenga il valore 1234H, e cioè 0001 0010 0011 0100 in binario: i quattro bit specificati nel registro BL sono i quattro bit meno significativi di AX e cioè 0100;
— la cella di memoria indirizzata da ES:DI (ad esempio posta all'indirizzo B800H:4444H) valga CCCCH e la word all'indirizzo successivo (B800H:4446H) valga FFFFH: graficamente possiamo rappresentare le due celle una a fianco all'altra, esplicitandone il contenuto a bit (vedi figura 1).

Subito dopo l'esecuzione dell'istruzione, dunque, il contenuto delle celle di memoria consecutive è rappresentato in figura 2.

Inoltre si ha l'aggiornamento automatico del registro «reg1», che nel nostro caso è DH: prima valeva 11, mentre ora, spostatici di 4 bit, varrà dunque 15, pronto a puntare il bit corretto in caso di nuova istruzione di INS.

Viceversa il registro BL, che conteneva 3 (il numero di bit di AX da inserire in memoria più 1) non viene aggiornato, tanto è vero che nell'istruzione INS potevamo tranquillamente sostituirlo con il valore immediato «3»: capito dunque il significato di «reg2» o di «imm4» vediamo che dunque vengono considerati solo 4 bit di entrambi in quanto al massimo si può «puntare» al bit 15.

Analogamente di «reg1» vengono considerati solamente i 4 bit meno significativi in quanto possiamo al massimo arrivare al bit 15 per avere il numero di bit di AX da estrarre ed inserire in memoria.

In questo esempio i 4 bit del registro AX sono entrati tutti all'interno della locazione il cui offset è posto in DI: per tale motivo ancora adesso (al termine della INS) DI: punta a quella cella.

Supponiamo a questo punto di cambiare il contenuto di AX in 8888H e di voler inserire 9 bit di tale registro in memoria proprio subito dove avevamo lasciato il tutto: eseguiamo dunque le istruzioni

```
MOV AX,8888H
INS DH,8 ;sono 9 i bit di AX da inserire...
```

Abbiamo dunque una situazione come in figura 3 e dopo l'esecuzione delle istruzioni la situazione della memoria diventa come rappresentato in figura 4, mentre il registro DH ora varrà 8 (il valore precedente, 15, più il numero di bit spostati, 9, il tutto modulo 16), mentre ora DI punterà (correttamente!) alla cella di offset 4446H.

Tirando le somme si tratta di un'istruzione alquanto complicata: la sua utilizzabilità pratica sfugge anche ad un esame poco più che superficiale.

Tuttavia potrebbe utilmente servire ad esempio nella gestione della memoria video, per andare a settare solo alcuni bit corrispondenti a pixel che devono essere accesi oppure colorati in un certo modo: visto che già tale operazione (senza le INS) risulta abbastanza complessa dato il numero di operazioni da eseguire (calcolo di indirizzi, shift di locazioni di memoria con mascheramento opportuno di bit), ecco che invece con la INS si otterrebbe forse qualche miglioramento, anche se non in termini di tempi di esecuzione, ma senz'altro in termini di occupazione di memoria.

Se solo fosse già esistito il V20 quando hanno scritto le routine di gestione del video dei PC...

Le altre due istruzioni, opposte alle corrispondenti INS, sono le «EXT» (che sta per «EXTract») e che servono invece ad estrarre dalla memoria (dalla locazione il cui indirizzo è posto in DS:SI) un certo numero di bit (specificati nel secondo operando, «reg2» oppure «imm4») a partire dal bit di posizione indicata come contenuto di «reg1»: questi bit andranno posti in AX a partire dall'LSB.

Anche in questo caso al termine dell'esecuzione viene correttamente aggiornato il registro «reg1» (in modo da poter restare in passo alla successiva

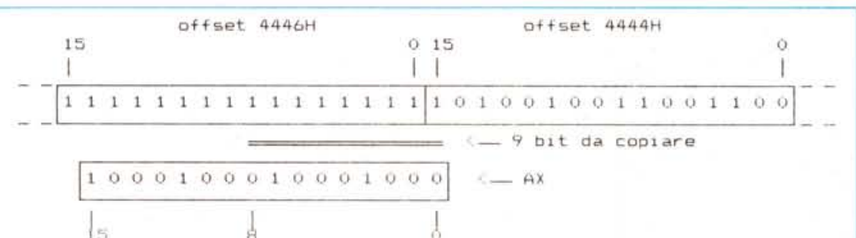

Figura 3

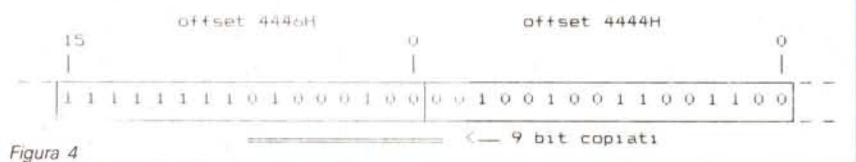

Figura 4

EXT, oppure, perché no, ad una successiva INS...), ed inoltre, se si è superata la «barriera» di divisione tra le word in memoria, allora viene anche aggiornato il puntatore SI.
non sia utile a questo punto appesantire il discorso con un altro esempio, che viceversa ricalcherebbe praticamente le orme lasciate dall'esempio delle INS.

Cinque nuove istruzioni di gestione di quantità BCD

Prima di vedere queste istruzioni ricordiamo come vengono gestite quantità codificate in BCD («Binary Coded Decimal», quantità decimali codificate in binario): il tutto è molto semplice.

Ogni cifra decimale, da 0 a 9, viene codificata con 4 bit («nibble») e perciò un byte serve a rappresentare un numero decimale compreso tra 0 a 99 (di due cifre, perciò): il numero 123456 è così espresso con tre byte, che dal meno significativo al più significativo valgono 56H, 34H e 12H.

Attenzione al fatto che queste cifre sono esadecimali: infatti il numero 123456 (decimale!) è rappresentato come

0001 0010	0011 0100	0101 0110

Detto questo, le nuove istruzioni gestiscono appunto stringhe di valori espressi in BCD; le cinque istruzioni si possono in realtà suddividere in due sottogruppi:
— il primo prevede l'addizione, la sottrazione e la comparazione tra quantità espresse in BCD e poste in memoria;
— il secondo permette di effettuare rotazioni a destra e a sinistra di un nibble alla volta.

Iniziamo dal primo sottogruppo che comprende le istruzioni «ADD4S», «SUB4S» e «CMP4S», che possono avere due forme, una in cui vengono esplicitati gli indirizzi delle due zone di memoria dove sono allocate le stringhe di caratteri BCD da addizionare, sottrarre e confrontare e l'altra forma in cui invece non compare alcun operando ed in questo caso gli indirizzi sono impliciti:

```
ADD4S dst,src
ADD4S
SUB4S dst,src
SUB4S
CMP4S dst,src
CMP4S
```

Tenendo bene a mente l'«operazione» che svolgono le tre istruzioni, le possiamo analizzare contemporaneamente: in particolare ed analogamente alle altre istruzioni di stringa, le forme contenenti gli operandi («dst» la stringa destinazione e «src» la stringa sorgente) sono facoltative, nel senso che nulla cambia a non mettere gli operandi, in quanto la codifica è sempre la stessa.

In ogni caso il registro CL (e questa è una novità) contiene la lunghezza in byte delle due stringhe che devono essere di lunghezza uguale, variabile tra 1 e 254 byte.

Come per le altre istruzioni di stringa, la stringa sorgente è individuata in memoria dalla coppia DS:SI, mentre quella destinazione è posta all'indirizzo contenuto nella coppia ES:DI: mentre la stringa di destinazione deve essere sempre posta nell'«Extra Segment» (ES); viceversa (in modo del tutto analogo alle altre istruzioni di stringa) la stringa sorgente può essere posta in un qualunque segmento ed a tale scopo è utile la forma delle istruzioni dotata di operandi, nella quale si può senz'altro porre un simbolo di «override».

Per chi non avesse ben chiaro il concetto, facciamo un esempio: supponiamo che la stringa sorgente ALFA sia nel Data Segment e la stringa destinazione sia nell'Extra Segment all'indirizzo definito da BETA.

Volendo sommare i due valori ALFA e BETA e volendo porre il risultato in BETA bisogna scrivere il seguente frammento di programma:

```
LEA SI, ALFA
LEA DI, BETA
MOV AX, EXTRASEGM
MOV ES, AX
ADD4S
```

Supponendo che invece la stringa ALFA sia posta nel Code Segment, allora bisogna usare la forma dotata di operandi e cioè:

```
LEA SI, ALFA
LEA DI, BETA
MOV AX, EXTRASEGM
MOV ES, AX
ADD4S BETA,CS:ALFA
```

In questo caso la «ADD4S» saprà che il valore contenuto in SI (l'offset della stringa ALFA) è riferito al CS e non al DS: questo avviene perché il codice operativo della ADD4S (per la cronaca 0FH 20H) è preceduto dal «prefisso di override» 2EH.

Aggiungiamo inoltre che conviene sempre lavorare con stringhe di lunghezza pari, eventualmente forzando a zero il nibble più significativo del byte più significativo: in tal modo i flag vengono correttamente settati al termine delle operazioni.

Ciò è importante per la «CMP4S» in quanto il carry e gli altri flag potrebbero non essere corretti se CL è dispari: in tal caso infatti il nibble più significativo contiene un valore indefinito che perciò può inficiare i risultati.

Il secondo sottogruppo invece è formato dalle istruzioni:

```
ROL4 reg
ROL4 mem
ROR4 reg
ROR4 mem
```

che permettono di ruotare di un nibble verso sinistra o verso destra (rispettivamente) il contenuto del registro «reg» o della locazione «mem» (entrambi solo ad 8 bit), utilizzando nella rotazione il nibble inferiore di AL.

Vediamo dunque cosa succede con le istruzioni:

```
MOV AL,12H
MOV DH,37H
ROR4 DH
```

in modo grafico si ha la situazione «prima» dell'esecuzione come è rappresentato in figura 5; dopo l'esecuzione della ROR4: AL ora vale 13H mentre DH ora vale 72H.

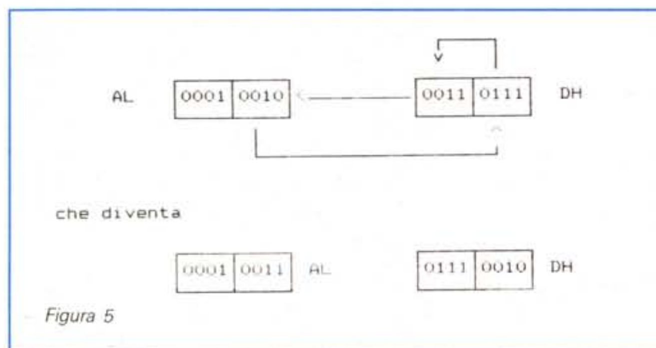

Figura 5

Lasciamo ai lettori la facile analisi di

cosa accade invece con le istruzioni ROL4...

Istruzioni di controllo di singoli bit

Il V20 introduce quattro nuove istruzioni che permettono la gestione dei singoli bit di byte, word: si tratta delle istruzioni «TEST1», «NOT1», «CLR1» e «SET1», le quali rispettivamente testano, complementano, resettano e settano un certo bit (specificato dal secondo operando) di un registro o di una locazione di memoria, sia ad 8 che a 16 bit.

Indicando con «XXX» una qualunque delle quattro istruzioni nuove, esistono (per ognuna di esse) otto possibilità date da:

```
XXXX reg8,imm3
XXXX reg8,CL
XXXX mem8,imm3
XXXX mem8,CL
XXXX reg16,imm4
XXXX reg16,CL
XXXX mem16,imm4
XXXX mem16,CL
```

In particolare, ad esempio, l'istruzione

```
CLR1 ALFA,CL
```

ka.25

resetta il bit (il cui numero d'ordine, tra 0 e 15, è posto in CL) della locazione di memoria ALFA, mentre

```
NOT1 AX,4
```

complementa il bit 4 del registro AX.

Aggiungiamo che, lavorando su quantità ad 8 bit, il valore immediato o il contenuto di CL è significativo solo nei primi tre bit (meno significativi), mentre con quantità a 16 bit i bit significativi salgono a 4.

Terminiamo l'analisi di queste istruzioni lasciando ai lettori l'analisi dell'istruzione.

```
SET1 CL,CL
```

apparentemente inutile...

Il modo «emulazione 8080»

Abbiamo già parlato la scorsa puntata del fatto che il V20 può emulare a tutti gli effetti un 8080: ciò si ottiene allorché il V20 incontra l'istruzione

```
BRKEM imm8
```

di codice operativo 0FH FFH «imm8» e che serve a saltare all'ambiente 8085 come se su di esso fosse «piovuto» l'interrupt «imm8».

In particolare l'istruzione in esame compie le seguenti operazioni:
— salva nello stack i flag e la coppia CS:IP;
— resetta il flag MD;
— calcola l'indirizzo relativo all'interrupt «imm8» dell'8080, all'interno dell'«interrupt vector table»;
— salta all'indirizzo indicato nella «table» e prosegue l'esecuzione come se fosse un 8080.

Il tutto prosegue fino a che, in modo 8080, incontra l'istruzione

```
RETEM
```

di codice operativo EDH FDH con il che viene disabilitato il bit MD e viene ripristinato dallo stack lo stato del V20 e cioè i flag e la coppia CS:IP: dopodiché l'istruzione successiva viene interpretata come un'istruzione del V20.

Altra possibilità del V20 (a partire dal modo «emulazione 8080») è di chiamare una subroutine scritta in V20, per mezzo dell'istruzione

```
CALLN imm8
```

di codice operativo EDH EDH «imm8», a seguito della quale vengono effettuate le seguenti operazioni:
— viene salvato nello stack lo stato dell'8080 e cioè la PSW, il Code Segment (non dimentichiamo che l'8080 è «immerso» in un ambiente V20, all'interno di un certo segmento) e il «Program Counter»;
— viene posto MD ad 1;
— viene calcolato l'indirizzo relativo all'interrupt «imm8» del V20, all'interno dell'«interrupt vector table»;
— salta all'indirizzo indicato nella «table» e prosegue l'esecuzione come se fosse un V20, il tutto fino a che viene incontrata un'istruzione IRET (la normalissima IRET di fine routine di interrupt).

Quattro chiacchiere sui tempi di esecuzione

Terminiamo dunque l'analisi del V20 mostrando una tabella comparativa tra i cicli di clock necessari all'esecuzione di istruzioni da parte di un V20 e di un 8086: le istruzioni vogliono essere un campione non del tutto casuale, mentre lasciamo ai lettori ogni ulteriore commento.

Con questo diamo l'appuntamento alla prossima puntata, la prima di una serie riguardante l'80386 (finalmente!!!) e che si chiamerà appunto «Assembler 80386».

istruzioni	clock V20	clock 8086
MOV ES,AX	2	2
MOV ALFA[BX],DL	13	9 + EA
LAHF	2	4
LEA SI,[BX][DI+12]	4	2 + EA
LODSB	7	12
REP LODSB (n cicli)	7 + 9 * n	9 + 13 * n
IN AL,DX	8	8
INC DX	2	2
DEC ALFA[SI]	24	15 + EA
MUL CL	22	77
MUL ALFA[DI]	40	124 + EA
DIV SI	25	144
CALL ROUTINE	20	19
CALL BX	18	16
CALL ALFA[BX][DI]	31	21 + EA
CALL FAR PTR QQ	29	28
CALL FAR PTR MEM	47	37 + EA
RET	19	8
RET 0006	24	12
RET (FAR)	29	18
PUSH AX	12	11
POP ALFA[SI]	25	17 + EA
JMP ROUTINE	12	15
JMP AX	11	11
JMP ALFA[BX][DI]	24	18 + EA
JMP FAR PTR PIPPO	15	15
JMP FAR PTR MEM[BX]	35	24 + EA
JLE ETICHETTA	14 o 4	16 o 4
INT 10H	50	52
IRET	39	24
HLT	2	2
STI	2	2
NOP	3	3
nuove istruzioni		
INS	35..113	-
EXT	34..59	-
ADD4S (CL=n)	7 + 19 * N / 2	-
ROR4 AL,5	29	-
BRKEM n	50	-