di Pierluigi Panunzi

Istruzioni di controllo

sesta parte

Siamo quasi arrivati al termine dell'analisi del set di istruzioni del microprocessore 80386, almeno per quel che riguarda il funzionamento in modo reale: terminata questa analisi, ci addentreremo infine nel mondo del Protected Mode, che nel 386 si è arricchito di nuove funzionalità tra le quali spicca il già citato «Virtual 8086 Mode».
Ma non precorriamo troppo i tempi e proseguiamo dunque nell'analisi delle istruzioni di controllo del flusso di programmazione e cioè delle istruzioni di salto, condizionato o meno

I salti condizionati

Evidentemente il 386 non ha apportato alcun cambiamento al funzionamento logico delle istruzioni di salto condizionato, che sono rimaste le stesse fin dai tempi dell'8086: abbiamo cioè anche in questo caso le 16 «jump», ben note, ma che per completezza riportiamo in figura 1.

Come altrettanto ben noto, queste istruzioni soffrono di una limitazione intrinseca in quanto consentono un salto ad una locazione posta all'interno di un range più o meno 128 (per l'esattezza +127 e −128) byte rispetto all'indirizzo in cui è posta l'istruzione stessa: questo fatto ha sinora comportato il vantaggio di una codifica estremamente breve per tali istruzioni (un byte per l'opcode ed uno per il cosiddetto **displacement** e cioè lo spostamento rispetto all'indirizzo effettivo), ma d'altro canto è stata sempre una limitazione nel caso di salti a posizioni più lontane, fatto questo che ha sempre richiesto l'uso intrecciato di due istruzioni di salto.

Se ad esempio si voleva effettuare il salto ad un'etichetta lontana più di 128 byte (nell'esempio si chiama ETICH) se il carry era settato, ecco che bisognava sempre scrivere un qualcosa del genere:

```
        ...
        JNC SOTTO
        JMP ETICH
SOTTO:
        ...
```

Ulteriore problema, a meno di non usare delle macro, era quello che ogni volta si doveva usare un'etichetta differente per «SOTTO».

Finalmente dunque con il 386 questa situazione è mutata (ed in fondo potevamo senz'altro aspettarcelo...) in quanto ora, accanto alle normali istruzioni di «salto condizionato corto», esistono le rispettive istruzioni di «salto condizionato lungo», dove dunque il **displacement** consente di raggiungere qualsiasi locazione all'interno del Code Segment: in prima analisi dunque potremo effettuare un salto nell'ambito dei canonici 64K byte di un normale Code Segment, con un displacement a 16 bit e non più ad 8.

Ma non dimentichiamoci che il 386 è un microprocessore a 32 bit e come tale può gestire informazioni di tale grandezza: abbiamo già visto, ma lo rivedremo meglio nei dettagli, che anche i segmenti di codice e di dati pos-

istruzione	Jump ...
JO	if Overflow
JNO	if Not Overflow
JB,JC,JNAE	if Below, if Carry, if Not Above nor Equal
JNB,JNC,JAE	if Not Below, if Not Carry, if Above or Equal
JE,JZ	if Equal, if Zero
JNE,JNZ	if Not Equal, if Not Zero
JBE,JNA	if Below or Equal, if Not Above
JNBE,JA	if Not Below nor Equal, if Above
JS	if Sign
JNS	if NOT Sign
JP,JPE	if Parity, if Parity Even
JNP,JPO	if Not Parity, if Parity Odd
JL,JNGE	if Less, if Not Greater nor Equal
JNL,JGE	if Not Less, if Greater or Equal
JLE,JNG	if Less or Equal, if Not Greater
JNLE,JG	if Not Less nor Equal, if Greater

Figura 1 - Ecco le sedici possibili funzioni di salto condizionato, ed i rispettivi sinonimi, con il loro significato.

sono essere formati da quantità appunto a 32 bit ed in particolare un assemblatore per il 386 (Turbo Assembler, per citarne uno...) avrà la possibilità di definire se un segmento contiene dati e/o istruzioni a 16 o 32 bit.

Ecco che dunque anche il segmento di codice potrà essere a 32 bit e perciò in questo caso il **displacement** sarà a 32 bit: con tali valori il range di indirizzi in cui è possibile effettuare il salto andrà tra −2^31 a +2^31−1 con il che possiamo stare più che tranquilli, in quanto tali valori sono nientemeno che −2G +2G-1 dove la «G» sta per «Giga»...

Il tutto ancora una volta avverrà sempre e comunque all'interno del Code Segment e perciò un salto condizionato verso una locazione posta in un altro Code Segment dovrà per forza di cose avvenire ancora nel modo che ben conosciamo, per mezzo della JMP di tipo «inter-segment».

Nella figura 2 vediamo un esempio, redatto per il Turbo Assembler, in cui si vedono le tre possibili situazioni:
— nella prima, in un segmento «normale», la codifica avviene secondo il metodo ben noto e cioè con due byte;
— nella seconda, che prevede un segmento normale, ma in ambiente 386 (la direttiva **.386** serve per far codificare le istruzioni secondo le regole e le caratteristiche di tale microprocessore, mentre la direttiva **USE16** indica quantità a 16 bit), vediamo che il salto avviene tranquillamente ad una locazione posta ad un offset pari a 1238H;
— nella terza situazione, in cui il segmento il codice è a 32 bit (grazie alla direttiva **USE32**), il salto condizionato avviene alla locazione posta all'indirizzo 0123456DH (a 32 bit!), ben al di là di quanto possiamo immaginare ed il tutto senza alcun errore di sorta.

In figura 3 vediamo invece lo stesso esempio in cui il salto però è effettuato ad un'etichetta posta rispettivamente all'indirizzo 13H, 15H e 16H, ottenute sommando 10H al program counter dell'istruzione successiva alla **jnz**: in

```
 1     0000                    code1   segment
 2                                     assume cs:code1
 3     0000  75 11                     jnz sotto1
 4                                     org $ + 10h
 5     0012  90                        nop
 6     0013                    sotto1:
 7     0013                    code1   ends
 8
 9                             .386
10     0000                    code2   segment use16
11                                     assume cs:code2
12     0000  0F 85 1235                jnz sotto2
13                                     org $ + 1234h
14     1238  90                        nop
15     1239                    sotto2:
16     1239                    code2   ends
17
18 00000000                    code3   segment use32
19                                     assume cs:code3
20 00000000  0F 85 01234568            jnz sotto3
21                                     org $ + 1234567h
22 0123456D  90                        nop
23 0123456E                    sotto3:
24 0123456E                    code3   ends
25                                     end
```

Figura 2 - Esempio di frammento di programma redatto in Turbo Assembler, per mostrare le possibilità di jump incondizionato in Code Segment «normali», ed in ambiente puramente 386, rispettivamente a 16 ed a 32 bit: si vede dunque come la stessa istruzione prevede codifiche differenti.

```
 1     0000                    code1   segment
 2                                     assume cs:code1
 3     0000  75 11                     jnz sotto1
 4                                     org $ + 10h
 5     0012  90                        nop
 6     0013                    sotto1:
 7     0013                    code1   ends
 8
 9                             .386
10     0000                    code2   segment use16
11                                     assume cs:code2
12     0000  75 13 90 90               jnz sotto2
13                                     org $ + 10h
14     0014  90                        nop
15     0015                    sotto2:
16     0015                    code2   ends
17
18 00000000                    code3   segment use32
19                                     assume cs:code3
20 00000000  75 15 90 90 90 90         jnz sotto3
21                                     org $ + 10h
22 00000016  90                        nop
23 00000017                    sotto3:
24 00000017                    code3   ends
25                                     end
```

Figura 3 - Altro esempio di frammento di programma, analogo al precedente di figura 2, nel quale i salti vengono fatti ad una locazione «vicina» in tutti e tre i casi: come spiegato nel testo, la codifica nei tre casi è sempre la stessa e si può notare l'uso di istruzioni NOP aggiunte dall'assemblatore stesso.

condizione	operazione logica effettuata
O	OF = 1
NO	OF = 0
B,C,NAE	CF = 1
NB,NC,AE	CF = 0
E,Z	ZF = 1
NE,NZ	ZF = 0
BE,NA	(CF = 1) OR (ZF = 1)
NBE,A	(CF = 0) AND (ZF = 0)
S	SF = 1
NS	SF = 0
P,PE	PF = 1
NP,PO	PF = 0
L,NGE	SF <> OF
NL,GE	SF = OF
LE,NG	(ZF = 1) OR (SF <> OF)
NLE,G	(ZF = 0) AND (SF = OF)

Tabella 1 - In questa tabella abbiamo riportato tutte le condizioni possibili sia per le istruzioni di «jmp condizionato» che per le nuove istruzioni di SET, per avere sempre sott'occhio quali sono le effettive operazioni logiche sui flag che determinano le condizioni.

questo caso vediamo che l'Assembler prende alcune decisioni importanti:
— nel primo caso il tutto va come nell'esempio di figura 2;
— nel secondo e nel terzo caso, relativi al 386 «puro», rispettivamente in segmenti a 16 ed a 32 bit, la codifica avviene sempre con lo stesso opcode e con un displacement ad un byte come nel primo caso, mentre sono state aggiunte delle istruzioni NOP per pareggiare i conti.

A questo punto c'è da fare una precisazione, a proposito di queste NOP: in particolare tali byte 90H vengono usati dall'Assembler (nella seconda passata) per riempire lo spazio che era stato riservato all'istruzione di salto nella prima passata.

Comunque mettendo al posto di

```
jnz sotto2
```

e simili, l'istruzione

```
jnz short sotto2
```

si può fare in modo che l'Assembler riservi solo i due byte canonici dell'istruzione nella forma più corta, anche in ambiente 386 con segmenti a 16 o 32 bit.

Chiudiamo con queste prime istruzioni di salto condizionato dando un'occhiata ai tempi di esecuzione, che in tutti i casi visti richiedono 3 oppure 8 cicli di clock a seconda se, rispettivamente, si passa all'istruzione successiva oppure se viene effettuato il salto (indipendentemente dal tipo di **displacement** utilizzato).

Altre istruzioni di salto condizionato

A questa categoria appartengono altre istruzioni già ben note: in particolare si tratta delle **JCXZ, LOOP, LOOPZ** e **LOOPNZ** che sono registrate al solito arricchite del funzionamento con i registri estesi.

In dettaglio, l'istruzione.

```
JCXZ LABEL
```

ora assume anche la forma

```
JECXZ LABEL
```

che non fa altro che controllare il valore contenuto nel registro esteso **ECX** e saltare all'etichetta «**LABEL**» indicata allorché tale contenuto sia nullo.

Per quel che riguarda le istruzioni **LOOP, LOOPZ** e **LOOPNZ** c'è da dire che non esistono versioni esplicite a 32 bit per tali istruzioni, ma viceversa le stesse istruzioni agiscono in modi differenti a seconda che ci si trovi in un Code Segment a 16 oppure a 32 bit: il primo tipo a 16 bit è quello per il quale le istruzioni **LOOP** agiscono nel modo che ben conosciamo, mentre nel caso in cui tali istruzioni si trovino in segmenti da 32 bit, continuano ancora a funzionare nello stesso modo, con l'unica differenza che il registro utilizzato non è più **CX** ma **ECX**.

In modo del tutto analogo il **displacement** in questo caso può essere espresso solamente con un byte il che significa che ancora una volta i loop possono essere lunghi al massimo −128 o +127 byte.

Un'occhiata ai tempi di esecuzione ci mostra che le istruzioni **JCXZ** e **JECXZ** vengono eseguite in 5 o 10 cicli di clock a seconda che, rispettivamente, si passi semplicemente all'istruzione successiva oppure si effettui il salto all'etichetta indicata.

Per quel che riguarda le istruzioni **LOOP**, c'è da dire solo che ora i cicli di clock sono 12, in tutti i casi visti ed indipendentemente dal fatto che ci si trovi in segmenti a 16 o 32 bit.

Nuove istruzioni legate allo stato dei flag

Ecco finalmente 16 nuove istruzioni (più i loro sinonimi) che consentono di

istruzione	Set byte ...
SETO	if Overflow
SETNO	if Not Overflow
SETB,SETC,SETNAE	if Below, if Carry, if Not Above nor Equal
SETNB,SETNC,SETAE	if Not Below, if Not Carry, if Above or Equal
SETE,SETZ	if Equal, if Zero
SETNE,SETNZ	if Not Equal, if Not Zero
SETBE,SETNA	if Below or Equal, if Not Above
SETNBE,SETA	if Not Below nor Equal, if Above
SETS	if Sign
SETNS	if NOT Sign
SETP,SETPE	if Parity, if Parity Even
SETNP,SETPO	if Not Parity, if Parity Odd
SETL,SETNGE	if Less, if Not Greater nor Equal
SETNL,SETGE	if Not Less, if Greater or Equal
SETLE,SETNG	if Less or Equal, if Not Greater
SETNLE,SETG	if Not Less nor Equal, if Greater

Figura 4 - Elenco delle nuove istruzioni di «set su condizione» (con i loro sinonimi), introdotte con il 386 e particolarmente utili nell'implementazione di linguaggi ad alto livello.

allargare lo spettro di possibilità del 386 rispetto ai suoi predecessori: si tratta di istruzioni che consentono di porre ad 1 oppure a 0 l'operando (rigorosamente ad 8 bit) a seconda se sia verificata o meno la condizione indicata dall'istruzione stessa.

In figura 4 vediamo l'elenco di tali nuove istruzioni accompagnate dal loro significato: ad esempio l'istruzione

```
setz al
```

che si legge «set AL if Zero», pone nel registro **AL** il valore 1 se si è in condizione di «zero» (cioè se il flag di Zero è stato settato dalle istruzioni precedenti) oppure il valore 0 nel caso contrario.

Se indichiamo con «x» una generica condizione espressa dallo stato di un flag oppure dall'OR, o dal NOR di due flag, genericamente si può dire che l'istruzione **SETx** pone nel bit meno significativo dell'operando proprio il valore del flag o il risultato dell'OR o del NOR di cui sopra: ad esempio l'istruzione

```
setng bh
```

controlla la condizione «NG» («Not Greater»), verificata se il flag di Zero è settato oppure se il flag di Segno è diverso dal flag di Overflow, e pone tale valore nel bit meno significativo di BH.

In formula si potrebbe dire così:

```
LSB(BH) <— ((ZF = 1) or (SF <> OF))
```

In generale dunque si può sintetizzare il funzionamento della generica istruzione «**SETx op**» con la formula

```
LSB(op) <— bit_calcolato
```

dove:
— «op» è l'operando ad 8 bit;
— «x» è la condizione da testare;
— «bit_calcolato» è il valore che assume il bit a seguito di una certa operazione logica, riportata in tabella 1, che può essere utile anche per le istruzioni del tipo «**Jx**» viste in precedenza.

Per quel che riguarda l'operando, oltre ad un registro ad 8 bit, può essere una qualsiasi locazione di memoria, il che comporta che può essere indicata nell'istruzione con uno qualsiasi dei modi previsti dal 386, che usano anche i registri estesi: ad esempio in figura 5 vediamo come ultima istruzione la

```
1     0000                    .model tpascal
2     0000                    .data
3     0000  ??                dato    db ?
4     0001  ??                tabella db ?
5
6     0002                    .code
7                             .386
8     0000  0F 93 C0             setae al
9     0003  0F 92 C3             setc bl
10    0006  0F 95 C1             setnz cl
11    0009  0F 92 C4             setb ah
12    000C  0F 9F C6             setnle dh
13    000F  0F 98 06 0000r       sets dato
14    0014  0F 90 06 1234r       seto dato + 1234h
15    0019  0F 94 06 FFEEr       sete dato - 12h
16    001E  0F 96 00             setbe [bx][si]
17    0021  67! 0F 92 84 F8      setb tabella[eax+edi*8+4]
18          00000005r
19                               end
```

Figura 5 - Esempio di applicazione di istruzioni del tipo «set byte su condizione».

setb tabella[eax+edi*8+4]

il cui operando è posto ad un indirizzo ottenuto a partire dall'offset di «**tabella**», aggiungendo ad esso il contenuto di **EAX**, il valore 4 ed il contenuto di **EDI** moltiplicato per 8...

In questo caso vediamo pure come ne risente la codifica, che prevede l'uso dei modi di indirizzamento propri del 386 e che fatti i conti portano alla codifica su 10 byte dell'istruzione in esame: questo è appunto il prezzo (in termini di occupazione di memoria) che si deve pagare per avere la potenza piena del 386.

Lo stesso discorso però non si applica minimamente ai tempi di esecuzione delle istruzioni stesse, che si mantengono sempre e comunque a livelli incredibilmente bassi: 4 cicli di clock per operandi di tipo registro e 5 per operandi in memoria, comunque calcolati!

Le ultime istruzioni di controllo del flusso

A questo gruppo appartengono istruzioni molto ben note e sulle quali ci soffermeremo solo per indicarne il nome e le funzioni svolte: si tratta di istruzioni alle quali il 386 ha aggiunto di suo non tanto qualcosa collegato ai 32 bit, ma ben altro, relativo ai suoi modi di programmazione in modo protetto.

Solo dopo aver analizzato in dettaglio e con calma tutte le sfaccettature delle possibilità di programmazione di tale microprocessore, potremo cercare di capire in particolare perché ad esempio una **CALL** verso un «Virtual 8086 task», a partire da un «286 task», richiede la bellezza di ben 218 cicli di clock, mentre se si parte da un «386 task» i cicli diventano 228.

A questo punto ricordiamo solo che le istruzioni di questo gruppo sono le ben note:
— **CALL**, per chiamare una subroutine;
— **JMP**, per saltare ad un certo indirizzo;
— **RET**, per ritornare da una subroutine al programma chiamante;
— **INT**, **INTO** e **BOUND**, rispettivamente l'attivazione software di un certo interrupt, l'attivazione dell'interrupt 3 in caso di Overflow e attivazione dell'interrupt 5 se l'indice di un vettore esce dai limiti imposti;
— **IRET**, per ritornare, al termine dell'esecuzione di una routine di servizio di un interrupt, al programma che era stato interrotto.

In tutti questi casi non si parlerà di «programmi» generici, ma bensì di **task** e sarà determinante sapere se tale task è in ambiente 80386 «puro», in «Virtual 8086 Mode» oppure ancora in ambiente 286: ricordiamo di sfuggita poi l'esistenza di quei «filtri logici» chiamati «Call Gate», «Trap gate», ecc, nonché dei differenti livelli di privilegio...

Comunque fin dalla prossima puntata parleremo di queste caratteristiche del 386, molte delle quali prendono le mosse da analoghe strutture presenti nel 286 e che avevamo già analizzato nell'ambito della rubrica apposita: i lettori ricorderanno bene che non si tratta di argomenti facili da trattare e da comprendere e per tale motivo cercheremo di renderli accessibili, magari tenendo sotto mano gli articoli relativi al 286.

MC