1 Altre istruzioni 8086

NOP

```
NOP     ; Not Operate
```

Sembra inutile, dato che non fa nulla. Può servire per "riempire del posto" da destinare successivamente ad altre istruzioni. Ne hanno bisogno i compilatori non ottimizzati di linguaggi ad alto livello, che, per funzionare, devono chiudere i "blocchi" di compilazione con delle NOP. Inoltre può essere aggiunta dai compilatori che ottimizzano il codice per le nuove CPU che eseguono più istruzioni contemporaneamente. La presenza di NOP piazzate al momento giusto può far in modo che non si interrompa il flusso di istruzioni che possono essere eseguite in parallelo.

Anche l'Assembler deve aggiungere delle NOP in qualche rara occasione.
Vediamo un esempio in sorgente, in linguaggio macchina e in disassemblato:

Sorgente	Indirizzo	L.M.	Disassemblato
QuiSopra:			
JMP QuiSopra	CS:0000	EB FE	JMP 0000
JMP QuiSotto	CS:0002	EB 01	JMP 0005
	CS:0004	90	NOP
QuiSotto:	CS:0005		

Quando il compilatore arriva a JMP QuiSopra sa già dov'è QuiSopra (ha uno spostamento di 0), per cui fa una jump short all'indietro (EB FE) che prende due Byte di codice.
Quando invece arriva a JMP QuiSotto non sa ancora quanto lontana sarà l'etichetta QuiSotto, perché non è l'ha ancora "vista", dato che è dopo. Quindi il compilatore non sa se la jump dovrà essere short o long. Per continuare a compilare la considera long ed occupa per essa tre Byte. In seguito essa si rivela short, così il compilatore inserisce una jump short in avanti (EB 01), ma rimane un Byte vuoto, che "riempie" con una NOP.

XLAT ("translate", per l'uso di tabelle di "transcodifica")

```
XLAT    ; Translate Byte to AL
```

Sintassi:
Sostituisce un Byte in AL con un Byte preso da una tabella in memoria. Inizialmente AL contiene un numero senza segno, che costituisce l'indice nella tabella. La Tabella ha inizio all'indirizzo il cui offset è in BX. BX non viene modificato. È utile per convertire un codice in un altro (p.es. ASCII in BCD), e, più in generale, per accedere ad una "lookup table" (vedi oltre).
L'istruzione fa ciè che farebbe l'istruzione "MOV AL, [BX + AL]", che in Assembly 8086 non si può fare (qualcosa del genere si può invece fare con un 386). Inoltre l'istruzione non ha operandi espliciti, cosa che la rende facilmente "iterabile" con un prefisso "REP" (vedi poco oltre).

```
XLATB   ; Translate Byte in Lookup Table
```

Istruzioni di stringa
Sono istruzioni che, facendo uso implicito di registri della CPU, puntano alla memoria per effettuare in modo efficiente operazioni ripetitive come copia di stringhe o vettori, scansione, ricerca di un numero particolare, inizializzazione o azzeramento di aree di memoria ..
Esse gestiscono automaticamente i puntatori alla memoria, in modo che il programmatore non li debba modificare esplicitamente. Infatti alla fine di queste istruzioni i relativi puntatori vengono aumentati o calati automaticamente di uno o di due Byte, in base al parallelismo degli operandi dell'istruzione.
La direzione di scansione viene stabilita dallo stato corrente del flag di direzione (DF **D**irection **F**lag). Se esso è zero, la scansione avviene per indici che aumentano, SI e/o DI vengono incrementati. Se DF = 1 si procede a ritroso, decrementando SI e DI.
Nelle istruzioni di stringa ES si usa sempre insieme a DI (destination index) e DS insieme a SI (source index).

Trasferimento di stringhe: MOVS
Esistono istruzioni di tipo "String MOV" (MOVS) sia per operazioni a Byte che a Word.

Ogni MOVS copia in ES:[DI] quello che trova in DS:[SI]; la MOVSB copia un singolo Byte, la MOVSW una Word. Dopo aver effettuato il trasferimento i valori di DS e SI vengono modificati automaticamente durante l'istruzione, in modo da puntare al prossimo elemento da copiare. Le MOVS, come le vere MOV, non hanno nessun effetto sui flag.

Il funzionamento di questa istruzione dà ragione dei nomi assegnati ad SI e DI. Infatti SI viene detto "source index" perché punta all'offset di ciò che viene letto dalla memoria (sorgente), mentre DI l'indice di destinazione perché all'indirizzo in DI si fa una scrittura (destinazione).

```
    MOVSB   ; Move String of Bytes
```

Il suo funzionamento:

```
MOV Byte ptr ES:[DI], DS:[SI]
Se DF =0
        INC DI
        INC SI;
Altrimenti se DF = 1
        DEC DI
        DEC SI;
```

```
    MOSVW   ; Move String of Words
```

Il suo funzionamento:

```
MOV Word ptr ES:[DI], DS:[SI]
Se DF =0
        ADD DI, 2
        ADD SI, 2;
Altrimenti se DF = 1
        SUB DI, 2
        SUB SI, 2;
```

Confronto fra "stringhe": CMPS
Le "String Compare" (CMPS) sono istruzioni per il confronto di due stringhe, ad indirizzi diversi. Il risultato, come in una vera CMP, si riflette solo sui flag.

```
    CMPSB   ; Compare String of Bytes
```

Il suo funzionamento:

```
CMP Byte ptr ES:[DI], DS:[SI]
Se DF =0
        INC DI
        INC SI;
Altrimenti se DF =1
        DEC DI
        DEC SI;
```

CMPSB effettua lo spostamento di SI e DI di un Byte, nella direzione indicata dal flag D, come visto in precedenza.

```
    CMPSW   ; Compare String of Words
```

Il suo funzionamento:

```
CMP Word ptr ES:[DI], DS:[SI]
Se DF =0
        ADD DI, 2
        ADD SI, 2;
Altrimenti se DF =1
        SUB DI, 2
        SUB SI, 2;
```

CMPSB effettua lo spostamento di SI e DI di due Byte, nella direzione indicata dal flag D.

Ricerca in una stringa: SCAS
Le "**Sc**an **S**tring" sono istruzioni per la ricerca di un numero in un'area di memoria.

```
    SCASB   ; Scan String of Bytes
```

Istruzione per la ricerca di un carattere in una stringa, confronta l'elemento di destinazione con il contenuto corrente di AL.
Il suo funzionamento:

```
CMP Byte ptr AL, ES:[SI] ; come una vera CMP, ha effetto solo sui flag
Se DF =0
            INC SI;
Altrimenti se DF =1
            DEC SI;
```

```
    SCASW   ; Scan String of Words
```

Il suo funzionamento:
CMP Word ptr AX, ES:[SI]
```
Se DF =0      ADD SI, 2;
Altrimenti se DF =1
        SUB SI, 2;
```

Lettura di un carattere di una stringa: LODS
```
    LODSB   ; Load String Byte

    LODSW   ; Load String Word
```

Le LODS, a Byte o a Word, caricano l'accumulatore (registro "A") con l'elemento corrente della stringa sorgente.
LODSB ; fa questo: "MOV AL, DS:[SI]", poi INC SI o DEC SI in base al valore di DF
LODSW ; fa questo: "MOV AX, DS:[SI]", poi ADD SI o SUB SI, 2 in base al valore di DF
Questa istruzione non ha effetto sui flag.

Scrittura in un carattere di una stringa: STOS
```
    STOSB   ; Store String Byte

    STOSW   ; Store String Word
```

Le STOS, a Byte o a Word, memorizzano nell'elemento corrente della stringa destinazione il valore dell'accumulatore.
STOSB ; fa questo: "MOV ES:[DI], AL" , poi INC DI o DEC DI in base al valore di DF
STOSW ; fa questo: "MOV ES:[DI], AX" , poi ADD DI, 2 o SUB DI, 2 in base al valore di DF
Questa istruzione non ha effetto sui flag.

Prefissi per la ripetizione
Dato che le istruzioni di stringa servono per lavorare su 'stringhe di memoria', normalmente esse andrebbero ripetute più volte in un ciclo, ma in tal modo si perderebbe parte del vantaggio per cui sono state create. Per questo motivo è stata introdotta una speciale istruzione che, usata come prefisso per le istruzioni di stringa, serve per eseguire tante volte l'istruzione di stringa specificata.
I prefissi per la ripetizione permettono di ripetere un certo numero di volte, e a certe condizioni, l'istruzione di stringa che viene subito dopo.

```
    REP     ; Repeat string instruction
```

Il suo funzionamento:
```
Ripeti:
    Esegui l'istruzione di stringa scritta dopo REP nel sorgente
DEC CX
    JNZ Ripeti
```

Mettendo in CX, prima della REP, il numero di volte che si vuole che la istruzione di stringa sia ripetuta, essa viene ripetuta CX volte.

Esempio, confronto di due stringhe della stessa lunghezza:
```
    MOV     CX,[Len]        ; lunghezza in CX
    JCXZ    zero            ; esce se CX è zero
    LDS     SI,[Source]     ; carica il "source pointer" in DS:SI
    LES     DI,[Dest]       ; carica il "destination pointer" in ES:DI
    CLD                     ; Manda all'avanti le istruzioni di stringa
    REPZ    CMPSB           ; Confronta le due aree di memoria in DS:SI e ES:DI
    JZ      zero            ; se le stringhe sono uguali, lascia CL = 0
    MOV     CL, 1           ; se sono diverse CL = 1
zero:
```

```
    REPE|REPZ  Repeat string instruction, with Equal | Zero
```
Oltre a funzionare come una REP, REPE e REPZ, che sono identiche (hanno lo stesso codice operativo), rimangono nel ciclo di ripetizioni se il risultato dell'operazione di stringa dà valore di flag = 0 (uguale o zero).
Ciò significa che l'istruzione non viene più ripetuta appena il dato trattato non è uguale o zero, oppure quando si è finito di contare (CX=0).

Funzionamento:
```
Ripeti:
```

```
    Esegui l'istruzione di stringa scritta dopo REPE nel sorgente
    JNE EsciDalREPE
    DEC CX
    JNZ Ripeti
EsciDalREPE:
```

`REPNE|REPNZ  Repeat string instruction, with Not Equal | Not Zero`

Analogo a REPE, ma con logica invertita. Si ripete se il risultato della istruzione di stringa NON è zero.

I prefissi di tipo REP non hanno un loro codice operativo. Infatti la presenza di una REP implica la modifica del codice operativo della istruzione di stringa che la segue. La modifica, che è un'aggiunta di un numero fisso, viene applicata a tempo di compilazione.

Esempio, ricerca della lettera "A" in una stringa di 10 elementi:

```
    MOV CX, 10
    MOV AX, SEG Stringa
    MOV     ES, AX
    MOV DI, OFFSET Stringa
    MOV AL, "A"
    REPNE   ; ripetizione: esce dalla REPNE alla prima "A" che trova,
            ; oppure se abbiamo fatto 10 SCASB (che segue)
            ; REPNE, essendo un prefisso, va messo prima
            ; della istruzione di stringa
    SCASB   ; scansione della stringa alla ricerca di "A"
```

Tabelle di salti (jump table)

!!!! fare

Tabelle di lookup (lookup table)

Invece di calcolare funzioni complesse, può essere molto più veloce disporre di tabelle precalcolate in diversi valori, mantenute in memoria e lette rapidamente dalla tabella quando necessita il valore della funzione in un determinato punto.

Esempio, convertire una variabile "carattere" da maiuscolo a minuscolo:

```
    ; il metodo "classico"
    MOV AL, character
    CMP AL, 'a'
    JB  NonMaggiore
    CMP AL, 'z'
    JA  NonMaggiore
    AND AL, 05Fh ; come SUB AL,32, più veloce
NonMaggiore:
    MOV character, AL

    ; con una lookup table servono meno istruzioni e più veloci:
    MOV AL, character
    ; carica il puntatore all'inizio della tabella
    LEA BX, InizioTabellaConversione
    XLAT    ; equivale a "MOV AL, [BX + AL]",
            ; prende il codice del maiuscolo corrispondente
    MOV character, AL
```

Per usare una tabella di lookup di questo genere essa deve essere fatta di Byte.

Se non si usa la XLAT si può sempre arrivarci con un indice:
Esempio:

```
    MOV SI, NumeroDellElementoDellaTabella
    SHL SI, 1 ; moltiplica per due perché la tabella è fatta di Word
    MOV AX, [Tabella + SI]   ; legge l'elemento voluto della tabella

oppure, con indirizzamento indicizzato(per 386>, funzionante in modo protetto)
    MOV SI, NumeroDellElementoDellaTabella
    MOV AX, [Tabella + ESI * 2]  ; legge l'elemento voluto della tabella
```

Spesso si usano tabelle di lookup per evitare calcoli complessi, che richiedono sottoprogrammi lenti e codice complicato da programmare, come per esempio il calcolo del valore del seno di una funzione. Il valore del seno a certi angoli viene memorizzato in una tabella di lookup, poi, dato un angolo specifico, di cui calcolare il seno, si fa un'interpolazione linerare fra i due valori più vicini che si trovano nella tabella. Questa tecnica è senz'altro meno accurata, ma estremamente più veloce degli algortmi "completi" usati comunemente.

Istruzioni solo per CPU 386 e successive
BT Bit Test
!!!! Fare
BTC Bit Test and Complement
!!!! Fare
BTR Bit Test and Reset
!!!! Fare
BTS Bit Test and Set
!!!! Fare

Caricamento dei flag
!!!! Fare