1 Istruzioni per l'elaborazione

In questo capitolo tratteremo delle istruzioni che servono per "calcolare", nel senso più vasto del termine. In particolare spiegheremo le varie istruzioni aritmetiche e le altre che producono il loro risultato secondo una funzione combinatoria.

1.1 Istruzioni aritmetiche

Le operazioni aritmetiche effettuate dai vari tipi di CPU sono molto varie. Le CPU più semplici dispongono solo delle operazioni di somma e sottrazione fra numeri interi, mentre quelle più sofisticate hanno istruzioni per tutte le operazioni aritmetiche, anche con i numeri a virgola mobile, e perfino istruzioni specializzate, utili nei calcoli per la grafica e per le telecomunicazioni, che eseguono contemporaneamente più di un'operazione aritmetica.
Per quanto riguarda la famiglia X86, nel "capostipite" erano presenti tutte le operazioni aritmetiche ordinarie fra numeri interi, comprese moltiplicazione e divisione. Inoltre il coprocessore dell'8086 (8087) supportava tutte le operazioni aritmetiche fra numeri a virgola mobile.
I codici mnemonici di tutte le istruzioni del "floating point coprocessor" iniziano per F. Dal 486 in poi, il coprocessore è stato incluso all'interno delle CPU, per cui le istruzioni a virgola mobile fanno parte a tutti gli effetti del set d'istruzioni X86.
Le ultime versioni di CPU X86, sia Intel (Pentium) che AMD (K6, Athlon), hanno esteso significativamente il numero delle istruzioni aritmetiche, sia nel campo dei numeri interi (MMX), che in quello dei numeri a virgola mobile (MMX II, AMD now!). In questo capitolo trattiamo solamente le istruzioni aritmetiche per i numeri interi.

1.1.1 Cambio di segno

L'istruzione NEG esegue il complemento a due del suo unico operando.

```
NEG  <OperandoDestinazione>
; Negate
; funzionamento:
; <OperandoDestinazione> <- ComplementoAdue (<OperandoDestinazione>)
```

L'effetto della NEG; come ricorda il suo mnemonico, è la commutazione del segno ad un numero con segno.
Di conseguenza il suo uso con un numero senza segno è del tutto sconsigliabile.
L'istruzione modifica tutti i flag aritmetici.
Esempio:

```
    ..
    ; non so se "numero" è positivo o no, devo fare cose diverse in base al
    ; segno ma comunque devo cambiare il segno del numero. Allora lo faccio
    ; subito, poi potrò saltare sul flag di segno:
    NEG [numero]
    JS OraEnegativo  ; se l'ho trasformato in negativo, salto là
OraEpositivo:
    ; se arrivo qui ora numero è positivo
    ..
```

Per il complemento a 2 in Assembly si usa la NEG. Si potrebbe usare anche una NOT (o anche XOR) seguita da una ADD, anche se il flag di carry sarebbe invertito rispetto alla NEG:
con NOT:

```
NOT AX
ADD AX, 1
; !! IMPORTANTE: il flag di carry dopo questa ADD funziona al contrario !!
```

Nelle CPU a partire dal 386 l'operando può essere di 32 bit.

1.1.2 Somme e sottrazioni di interi

Le istruzioni per somma e sottrazione fra numeri interi sono ADD e SUB, rispettivamente.
La loro sintassi è la seguente:

```
ADD <OperandoDestinazione>, <OperandoSorgente>
; Add (to add significa "sommare")
; funzionamento:
; <OperandoDestinazione> <- <OperandoDestinazione> + <OperandoSorgente>

SUB <OperandoDestinazione>, <OperandoSorgente>
; Subtract (sottrai)
```

```
; funzionamento:
; <OperandoDestinazione> <- <OperandoDestinazione> - <OperandoSorgente>
```

Le due istruzioni sono sottoposte ai consueti vincoli per quanto riguarda le operazioni in memoria (uso di un solo operando in memoria, registri alcuni registri fanno da puntatori ..).
Naturalmente i due operandi devono avere lo stesso numero di bit, nelle CPU a partire dal 386 possono essere di 32 bit.
Per via delle "magiche" proprietà del complemento a due ADD e SUB possono funzionare utilizzando indifferentemente numeri con segno o senza segno. Le istruzioni ADD e SUB non conoscono la differenza fra i due tipi di numero, per cui il controllo se il risultato è giusto è affidato esclusivamente al programmatore. Se il programma deve controllare la correttezza del risultato di somme e sottrazioni il programmatore deve chiedersi, ad ogni ADD o SUB, se sta trattando con numeri con o senza segno ed utilizzare la giusta jump per il controllo.
Si ricorda che se il numero è senza segno l'errore in ADD o SUB è segnalato dal flag di carry, mentre si usa il flag di overflow per i numeri con segno, in complemento a due.
Dopo un'istruzione ADD o SUB il valore di tutti i flag aritmetici è sempre determinato in base al risultato dell'operazione eseguita.

L'esecuzione di un'istruzione SUB ha una particolarità che riguarda il flag di carry.
Una SUB viene eseguita utilizzando gli stessi circuiti elettronici dedicati alla ADD. Per cui per una SUB la CPU esegue tre "passi", come indicato nella Figura 1.

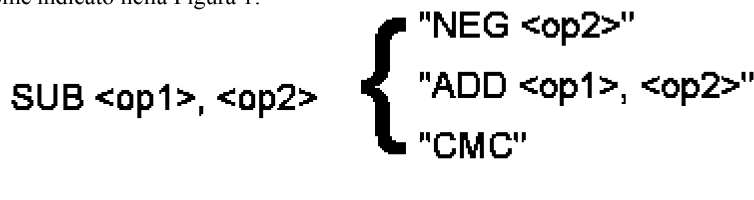


Figura 1: istruzioni equivalenti della SUB
Prima cambia di segno al secondo operando, poi gli somma il primo. Infine cambia il valore del flag di carry (CMC, per i dettagli vedi oltre in questo capitolo).
Vediamo la ragione di quest'ultima operazione, effettuando la sottrazione in colonna, così come la farebbe la CPU.
Supponiamo che siano:
<op1> = 10 e <op2> = 3

NEG <op2>

<op2>	0	0	0	0	0	0	1	1	NOT
	1	1	1	1	1	1	0	0	+
								1	
<op2>	1	1	1	1	1	1	0	1	

ADD <op1>,<op2>

Riporto	1	1	1	1	1					
<op1>		0	0	0	0	1	0	1	0	+
<op2>		1	1	1	1	1	1	0	1	
<op1>		0	0	0	0	0	1	1	1	

Si può notare come il riporto si sia propagato fino in fondo, anche se l'operazione non è errata ed il risultato in op1 è esatto (10 – 3 = 7). Questo succede sempre ed il carry dell'operazione è sempre "rovesciato" rispetto al caso della ADD.
Perché ADD e SUB funzionino nello stesso modo la CPU esegue sempre, alla fine della SUB, l'equivalente di una CMC (complement carry).
CMC

Carry flag	0

INC e DEC

Queste istruzioni effettuano un'addizione od una sottrazione del numero 1.

```
INC <OperandoDestinazione>
; Increment
; <OperandoDestinazione> <- <OperandoDestinazione> + 1

DEC <OperandoDestinazione>
```

```
; Decrement
; <OperandoDestinazione> <- <OperandoDestinazione> - 1
```

INC e DEC non hanno vantaggi in termini di velocità di esecuzione ma richiedono un operando in meno, dato che l'1 è implicito. Ciò può comportare una certa diminuzione della quantità di memoria occupata dal programma, dato che l'aggiunta di 1 ad un numero è molto frequente.

Esempio:

Istruzione	Ling. macchina	Istruzione	Ling. macchina
ADD AX, 1	05	INC AX	40h
	01		
	00		

La ADD usa tre byte, la INC ne usa uno.

Una particolarità di INC e DEC degli 80X86 è che non modificano il flag di carry. Questo può essere utile nei cicli per incrementare il contatore di controllo senza rovinare il carry di un'istruzione precedente. Peraltro può comportare errori se ci si fida del carry in uscita da queste istruzioni. Se serve verificare il carry dopo aver aggiunto 1, bisogna usare la ADD.
Vediamo un esempio nel quale il fatto che la INC non modifichi il carry si rivela vantaggioso. Supponiamo di voler trovare a quale elemento di un vettore di Byte la somma dei valori degli elementi diventa maggiore di 255.
Per far ciò dobbiamo "scorrere" il vettore continuando a sommare in un registro, fino a che non scatta il flag di carry.
Vediamo una realizzazione con ADD, confrontata ad una realizzazione con INC:

```
Programma con ADD
..
MOV BX, 0
; in AL la somma degli elementi del vettore V
MOV AL, 0
GiroFinoAlCarry:
ADD AL, [V + BX]
JC Esci
ADD BX, 1
JMP GiroFinoAlCarry
Esci:
```

```
Programma con INC
..
MOV BX, 0

MOV AL, 0
GiroFinoAlCarry:
ADD AL, [V + BX]
INC BX
JNC GiroFinoAlCarry
```

Mentre nella versione con ADD siamo costretti alla JC subito dopo la ADD AL, [V + BX], perché la ADD successiva rovina il flag di carry, nella versione con INC possiamo posticipare la jump condizionata a dopo l'incremento, dato che la INC non rovina il flag di carry sistemato dalla ADD AL, [V + BX].
Questo ci permette di risparmiare una jump.

Numeri con molti bit (ADC e SBB)

Potrebbe succedere di dover eseguire calcoli aritmetici con numeri più grandi di quelli che i registri a nostra disposizione possono contenere. In questo caso sarà necessario usare più di un registro per memorizzare ogni operando ed ogni risultato. Le operazioni aritmetiche sono ancora possibili a patto di osservare alcune precauzioni[1].
Per sommare numeri più grandi dei registri si opera come se si trattasse una normale operazione in colonna, fatta sulla carta. Si deve cominciare a sommare le parti basse; il risultato che si ottiene è la parte bassa del risultato.
Se la somma comporta un riporto esso deve essere propagato alle parti alte. La somma delle parti più alte dovrà perciò tenere conto del carry della somma precedente.
Con alcune istruzioni di codice si potrebbe verificare se il carry flag è ON ed aggiungere uno alla somma della parti più alte.

Esempio "sulla carta"; nel seguente schema sommiamo due numeri a 32bit: A6F1DA01h + 30203450h:

Riporti		1		1	1				
	A	6	F	1	D	A	0	1	+
	3	0	2	0	3	6	5	0	=
	D	7	1	2	1	0	5	0	

Nello schema sono evidenziate le due word che costituiscono i due numeri da sommare ed il risultato.
Nel caso specifico la somma fra le word basse genera un carry (evidenziato nello schema), che deve essere sommato alla somma delle parti alte per ottenere il risultato corretto.
Nell'8086 esistono istruzioni specifiche, per la somma e la sottrazione, che rendono semplicissima la propagazione del riporto. Esse sono la ADC e la SBB.

1 nelle seguenti argomentazioni consideriamo l'istruzione di somma, per la sottrazione il discorso sarà del tutto analogo.

La loro sintassi è la seguente:

```
ADC <OpDestinazione>, <OpSorgente>
; Add with carry (somma con riporto)
; <OpDestinazione> <- <OpDestinazione> + <OpSorgente> + carry

SBB <OperandoDestinazione>, <OperandoSorgente>
; Subtract with borrow (sottrai con prestito)
; <OpDestinazione> <- <OpDestinazione> - <OpSorgente> + carry
```

Esse, oltre ad eseguire una "normale" ADD o SUB, sommano al risultato anche il valore corrente del flag di carry.

Esempio:

```
; la somma di due numeri a 32 bit viene memorizzata
; nel primo dei due:
PrimoNumero DD (?)
SecondoNumero DD (?)
..
; somma delle parti basse SENZA il carry:
MOV AX, WORD PTR [SecondoNumero]
ADD WORD PTR [PrimoNumero], AX
; somma delle parti alte, che in memoria sono due byte "più in là",
; somma anche l'eventuale carry che proviene dalla prima ADD
MOV AX, WORD PTR [SecondoNumero + 2]
; MOV non modifica i flag!
ADC WORD PTR [PrimoNumero + 2], AX    ; ADC!
JC NumeroPiuGrandeDi32bit
..
```

ADC funziona anche con numeri con segno

1.1.3 Moltiplicazioni e divisioni di interi

Per le operazioni di moltiplicazione e divisione l'8086 mette a disposizione istruzioni specifiche che, a differenza della ADD e della SUB, devono essere diverse se l'operazione è con o senza segno.
Perciò, a differenza della ADD, che era unica per tutti i tipi di numeri interi, nelle CPU X86, ed in tutte le altre, esistono sia l'istruzione MUL (per i numeri senza segno) che la **I**MUL (**integer** MUL per i numeri con segno).
Nelle MUL e nelle DIV il registro AX viene usato nel modo evocato dal suo nome, cioè come "accumulatore". L'accumulatore è infatti un registro che non viene indicato esplicitamente fra gli operandi delle istruzioni ma viene usato automaticamente dall'istruzione stessa.
Dato che un operando è sempre AX o una sua parte, esso non viene indicato; MUL e DIV richiedono <u>un solo operando</u>.
Il comportamento di queste istruzioni dipende dalla dimensione dell'operando, che può essere a 8, 16 o, dal 386 in poi, a 32 bit.

MUL e IMUL a un operando

Vediamo la loro sintassi:

```
MUL[BYTE PTR] <Op Sorgente Da 8 bit>
; unsigned Multiplicate (moltiplicazione a 8 bit)
; AX <- AL * <O    p Sorgente Da 8 bit>

MUL[WORD PTR] <Op Sorgente Da 16 bit>
; unsigned Multiplicate (moltiplicazione a 16 bit)
; funzionamento:
; <temp a 32 bit> <- AX * <Op Sorgente Da 16 bit>
; AX <- <temp a 32 bit parte bassa>
; DX <- <temp a 32 bit parte alta>

MUL[DWORD PTR] <Op Sorgente Da 32 bit> ; solo 386 e >
; unsigned Multiplicate (moltiplicazione a 32 bit)
; funzionamento:
; <temp a 64 bit> <- EAX * <Op Sorgente Da 16 bit>
; EAX <- <temp a 64 bit parte bassa>
; EDX <- <temp a 64 bit parte alta>
```

<u>Limitazioni</u>: MUL non accetta un operando immediato nelle CPU precedenti al 286.

Esempi:
!!!! FARE

Le MUL usano registri "piccoli" per gli operandi e memorizzano il risultato in registri più "grandi".
Esse funzionano in modo diverso in base alla dimensione dell'operando sorgente. Se l'operando è da 8 bit il risultato è messo in AX (16 bit); se l'operando è da 16 bit si memorizza a 32 bit (DX contiene la parte alta, AX la parte bassa); se l'operando è a 32 bit il risultato è di 64 bit (EDX parte alta, EAX parte bassa).
E' interessante notare che il fatto che nella moltiplicazione raddoppi il numero dei bit del risultato non rende possibile l'errore aritmetico in moltiplicazione.
Per capirlo supponiamo di effettuare una moltiplicazione fra i due numeri più grandi di 8 bit, senza segno. Il risultato 255 * 255 = 65025 è minore di 65535, che è il massimo intero senza segno rappresentabile in AX, di 16 bit. Siccome questo vale anche per le IMUL con segno, se si considera il risultato "grande" dell'istruzione, non c'è mai errore aritmetico nelle moltiplicazioni.
Se invece si considera la sola "metà bassa" del risultato, cioè un risultato che ha lo stesso numero di bit dell'operando esplicito, allora errore ci può essere, eccome. I flag accesi dalle MUL considerano proprio questa possibilità di errore aritmentico.
Infatti il funzionamento dei **flag** nelle istruzioni MUL e IMUL è particolare.
La CPU mette a zero sia il flag di carry che quello di overflow se il risultato è ancora "piccolo". Altrimenti li mette a uno. I flag diversi da CF e OF assumono valori indeterminati. Gli altri flag sono indefiniti.
In particolare la MUL mette a zero CF e OF se la parte alta del risultato (AH, DX (o anche EDX dal 386)) è nulla.
IMUL mette a zero CF e OF se la parte alta è composta da tutti zero, in caso di risultato positivo, o tutti uno, se il risultato è negativo (si ricorda che un numero negativo "piccolo" è composto da tutti uno nella sua parte alta (es. –2 a 16 bit = 1111111111111110).

Il fatto che la MUL sia un'istruzione un po' atipica può essere pericoloso nella programmazione Assembly.
Vediamo un esempio nel quale MUL viene usata impropriamente:

```
                 AH   AL
MOV AX, 2001h  | 20 | 01 | AX
MOV BL, 1      |  ? | 01 | BX
MUL BL           ; moltiplica AL = 1 per BL = 1; il risultato è 1 :-), che finisce in AX
                 ; anche se è piccolo!
               | 00 | 01 | AX
                 AH   AL
```

Se non si fa attenzione la MUL può avere effetti collaterali indesiderati. Nell'esempio il valore 20h che era stato scritto in AH è stato sovrascritto dalla MUL. Se quel 20h era utile al programma, esso malfunziona.
Ancora più facile sbagliare con la MUL a 16 bit, che va ad "invadere" un altro registro (DX). Bisogna tenere presente che, indipendentemente dal risultato dell'operazione, la MUL a 16 bit scrive qualcosa in DX. Prima della MUL è necessario salvarne il contenuto, se non si può perderlo.

IMUL 8086

L'istruzione **IMUL** dell'8086 (**I**nteger **Mul**tiplication) ha la stessa sintassi della MUL; la differenza è che opera con numeri con segno (i numeri interi con segno vengono detti "Integer", questa è quindi la Integer MUL).

```
    IMUL <Op Sorgente Da 8, 16 o 32 bit>
    ; Integer Multiplicate (come la MUL, ma con segno)
    ; funzionamento:
    ; il funzionamento della IMUL ad un solo operando è del tutto
    ; analogo alla MUL a 8, 16 e 32 bit, ma viene effettuata una
    ; moltiplicazione fra numeri con segno
```

IMUL 386

Dal 386 la IMUL può funzionare con due o addirittura tre operandi.

```
    IMUL <Registro destinazione da 16 o da 32 bit>, <Sorgente 8/16/32>

    ; solo 386 e >

    ; Integer Multiplicate (moltiplicazione con segno a due operandi)
    ; gli operandi devono essere della stessa dimensione
    ; funzionamento:
    ; <Registro Destinazione> <- <Registro Destinazione> * <Sorgente>

    IMUL <Registro destinazione 16/32>, <Operando memoria 16/32>, <Operando im-
mediato 8/16/32>
    ; solo 386 e >
    ; Integer Multiplicate (moltiplicazione con segno a tre operandi)
    ; funzionamento:
```

```
; <Registro Destinazione> <- <Memoria> * <Immediato>
```

Le istruzioni a due operandi hanno come destinazione un registro qualsiasi e non "invadono" altri registri. Esse moltiplicano i due operandi mettendo il risultato nel primo dei due.
L'operando sorgente può essere ottenuto con un accesso in memoria o può essere un numero in immediato. Il numero in immediato può essere di un numero di bit minore della destinazione. In tal caso la CPU effettuerà automaticamente un'estensione del segno.

Esempi:

```
IMUL DX, SI          ; fra registri di 16 bit, risultato SOLO in DX
IMUL AX, Byte PTR 10 ; moltiplicazione con sorgente da 8 bit in immediato
                     ; la CPU procede all'estensione del segno di 10
                     ; il risultato finisce SOLO in AX
IMUL ECX, Dword PTR [31] ; risultato in ECX, operazione senza estensione del
                         ; segno di [31] (è già a 32 bit)
IMUL EBP, Byte PTR 28    ; estensione del segno, 28 viene trasformato da 8 a 32 bit
```

Le IMUL contr operandi eseguono la moltiplicazione fra i due operandi, memorizzando il risultato in un altro registro, diverso dai due operandi.
Gli operandi possono essere un numero prelevato dalla memoria ed un altro in immediato.
Esempi:

```
IMUL DX, [Var16], Byte PTR 10     ;
IMUL DX, [Var16], Word PTR 10     ;
IMUL EDX, [Var32], Byte PTR 7     ;
IMUL EDX, [Var32], Dword PTR 7    ;
; Var16 è a 16 bit, Var32 è a 32
```

Non sono possibili tutte le combinazioni nelle dimensioni degli operandi e dei registri di destinazione. Per il dettaglio di ciò che è possibile si veda l'istruzione IMUL in appendice XXXX (set d'istruzioni X86) .

DIV e IDIV

Le istruzioni di divisione fanno il contrario delle moltiplicazioni. Partendo da numeri "grandi" producono numero "più piccoli".
Anche le DIV hanno un funzionamento differenziato secondo la dimensione dell'operando, come si può apprezzare analizzandone la sintassi:

```
DIV[BYTE PTR] <Op Sorgente Da 8 bit>
; unsigned Division (divisione a 8 bit)
; AL <- AX \ <Op Sorgente Da 8 bit>)
; !! con \ si intende la divisione intera
; !! in AL va il quoziente della divisione intera
; AH <- AX mod <Op Sorgente Da 8 bit>
; !! con mod si intende il resto divisione intera !!
; !! in AH va il resto della divisione intera

DIV [WORD PTR] <Op Sorgente Da 16 bit>
; unsigned Division (divisione a 16 bit)
; AX <- (DX,AX \ <Op Sorgente Da 16 bit>)
; DX <- (DX,AX) mod (<Op Sorgente Da 8 bit>)
; !! con DX,AX si intende un numero di 32 bit che ha
; !! DX come parte alta ed AX come parte bassa

DIV [DWORD PTR] <Op Sorgente Da 32 bit> ; solo 386 e >
; unsigned Division (divisione a 32 bit)
; EAX <- (EDX,EAX \ <Op Sorgente Da 16 bit>)
; EDX <- (EDX,EAX) mod (<Op Sorgente Da 8 bit>)
; !! con EDX,EAX si intende un numero di 64 bit che ha
; !! EDX come parte alta ed EAX come parte bassa
```

Limitazioni: DIV non ammette operando immediato in nessuno dei processori della famiglia X86. Unici tipi di operandi possono essere registri e posizioni di memoria.
~~DIV 20~~ : vietata: operando in immediato

Il dividendo dell'operazione DIV è più grande del risultato. Il risultato comprende la parte intera (quoziente), che viene messa nella parte bassa dell'accumulatore-risultato, ed il resto della divisione intera (modulo), che viene messo nella parte alta.

Se il divisore è da 8 bit il dividendo è da 16 bit e sta in AX, la parte intera del risultato va in AL, mentre il resto (modulo) va in AH.
Con divisore da 16 bit il quoziente è contenuto nei due registri DX ed AX; in DX la parte più significativa, in AX la parte meno significativa. I risultati vanno: in AX la parte intera, in DX il resto.

Come IMUL, **IDIV** (**I**nteger **Div**ision) funziona con **i**nteger (numeri con segno). IDIV ha la stessa sintassi e funzionamento di DIV.
Anche l'istruzione DIV cambia la dimensione dei suoi risultati, per cui anche con questa istruzione il programmatore dovrà essere attento ..
Vediamo un esempio dei pericoli che si corrono:

```
; questo brano di programma dividerebbe AX = 48 per BX = 6
; (se funzionasse!)

MOV DX, 1      | 00   01 | DX

MOV AX, 48     | 00  2Ch | AX

MOV BX, 6      | 00   06 | BX

DIV BX         | 2A   B2 | AX  ; NON è quello che volevo! ; (mi aspettavo 8!)
               | 00   00 | DX  ; è il resto della divisione che non volevo fare!
```

Specie quando i numeri sono piccoli, come in questo caso, ci si può dimenticare che la DIV a 16 bit non lavora solo con AX, ma anche con DX, che è la parte alta del dividendo. In questo brano di programma in DX c'è 1, per cui la divisione eseguita non è 44 : 6 = 7 col resto di 2, ma 01002Ch : 6 = 2AB2h con il resto di 0.
A differenza della MUL, che non dà mai errore aritmetico se si considera tutto il risultato, l'istruzione DIV può dare errore in due casi: la divisione per zero e l'overflow.
Se si prova a dividere per zero la CPU deve comunicare un errore di divisione per zero.
L'altro errore occorre quando un numero troppo grande viene diviso per uno troppo piccolo. In questo caso il risultato può essere più grande del registro destinato a contenerlo, la qual cosa genera un errore.
Se si osserva la tabella delle istruzioni in Appendice XXXX si può notare che DIV e IDIV lasciano tutti i flag aritmetici in condizioni non determinate. Dunque non è possibile usare i flag per sapere se una divisione è andata a buon fine.
Il meccanismo con il quale la CPU comunica un errore di divisione viene detto "**eccezione**" e verrà sviluppato nel seguito. Per ora basti sapere che nel caso scattino questi errori la CPU è in grado di far partire automaticamente un particolare sottoprogramma, detto "eccezione", che gestisce l'errore, solitamente chiudendo in anticipo il programma. Dunque, fra le tante di un X86, esistono l'"eccezione di overflow in divisione" e quella di "divisione per zero".

Casting ed estensione del segno

Nell'effettuare le divisioni può essere indispensabile far diventare "più grandi" i numeri con cui si sta lavorando. Infatti la DIV utilizza come dividendo numeri che hanno il doppio di bit rispetto al divisore, per cui è probabile che sia necessario sistemare la parte alta del numero "grande" in modo che sia "zero".
Se il numero che si deve dividere è senza segno la promozione non è un problema: la parte alta del numero "più grosso" deve essere semplicemente messa a zero.
Vediamo un esempio:

```
; divisione di A per B, numeri da 32 bit senza segno, con un 386:
A DD (?)
B DD (?)
..
MOV EAX, [A] ; il numero da dividere deve essere da 64 bit!
MOV EDX, 0   ; allora la parte alta deve essere zero
; ora posso dividere il numero da 64 bit EDX, EAX
; per B:
DIV [B]
; ora salvo il risultato in A (la parte alta non conta perché
; era zero ed ora A è più piccolo o uguale a prima)
MOV [A], AX
```

Se i numeri hanno il segno il discorso cambia, perché per promuovere il numero bisognerebbe fare operazioni diverse in base al segno.
Il set d'istruzioni X86 comprende istruzioni che agevolano questa operazione. Per esempio nell'8086 l'istruzione CBW (**C**onvert **B**yte to **W**ord) converte un numero con segno di 8 bit contenuto nel registro AL (solo AL, accumulatore) in uno di 16, contenuto nel registro AX, estendendone il segno.
Le tre istruzioni di promozione del segno sono le seguenti:

```
CBW
; Convert Byte to Word
```

```
; AX <- EstensioneDelSegno(AL)
; conversione di numero con segno da 8 a 16 bit

CWD
; Convert Word to DoubleWord
; DX, AX <- EstensioneDelSegno(AX)
; conversione di numero con segno da 16 a 32 bit

CDQ
; Convert DoubleWord to QuadWord
; EDX, EAX <- EstensioneDelSegno(EAX)
; conversione di numero con segno da 32 a 64 bit
; 386 >
```

Queste operazioni sono senza operandi perché lavorano implicitamente sui registri che fanno da accumulatore.
Le istruzioni copiano il bit più significativo dell'operando "piccolo" in tutti i bit della parte alta.

Un esempio analogo al precedente, ma con numeri con segno:

```
; divisione di C per D, numeri da 32 bit con segno, con un 386:
C DD (?)
D DD (?)
..
MOV EAX, [C] ; il numero da dividere deve essere da 64 bit!
CDQ          ; Convert Double to Quad
; ora posso dividere il numero con segno da 64 bit
; contenuto in EDX, EAX per D:
DIV [D]
; ora salvo il risultato in C
MOV [C], AX
```

Per agevolare le operazioni di promozione di variabili ed estensione del segno esistono, dal 386 in poi, due istruzioni che caricano un operando sorgente di una certa dimensione in un operando destinazione di dimensione diversa.

MOVZX <op1>, <op2> (**Mov**e with **z**ero e**x**tend), carica l'operando "piccolo" in quello "grande" come numero senza segno (zero extend), aggiungendo tutti zeri nella parte alta.

MOVSX <op1>, <op2> (**Mov**e with **s**ign e**x**tend), carica l'operando "piccolo" in quello "grande" come numero con segno (sign extended), aggiungendo il bit del segno in tutta la parte alta.

<op2> può essere a 8 o 16 bit, <op1> da 16 o 32 bit.

Per concludere le istruzioni aritmetiche ricordiamo che esistono specifiche istruzioni per l'aritmetica decimale in BCD; segue un elenco senza ulteriori commenti, stante la rarità dell'uso di queste istruzioni.
DAA Decimal Adjust for Add
DAS Decimal Adjust for Subtract
AAA ASCII Adjust for Add
AAS ASCII Adjust for Subtract
AAM ASCII Adjust for Multiply
AAD ASCII Adjust for Divide
Queste istruzioni non hanno operandi perché lavorano tutte sull'accumulatore (AL, sono tutte istruzioni a 8 bit).

Moltiplicazioni e divisioni fra numeri "grandi"
La moltiplicazione si può eseguire così come la si farebbe sulla carta, mentre la divisione è ancora più complicata. Il codice per queste operazioni è piuttosto complicato e non lo tratteremo qui. Il dettaglio si può trovare in testi specializzati (vedere p.es. "The Art of Assembly", libro gratuito reperibile in Rete, capitolo 9)

1.1.4 Manipolazione dei flag

Nella programmazione può essere necessario cambiare il contenuto di alcuni flag. Le istruzioni che esistono per questo scopo sono SET flag, che li mette ON, CLEAR flag, che li spegne e COMPLEMENT flag che ne commuta il contenuto

Impostazione di flag (set)
Questa istruzione mette a uno il flag; la sua sintassi:

```
ST<flag>
; Set <flag>
; <flag> <- 1
<flag> := C | D | I
```

Mette a 1 il flag specificato dalla lettera che segue ST. Si può usare solo su tre flag: **C**arry, **D**irection e **I**nterrrupt Flag.
Di conseguenza sono permesse solo le seguenti: STC, STD, STI
Esempio:

STC ; mette a 1 il bit di carry

Azzeramento di flag (clear)

E' il contrario della ST<flag> e mette a zero:

```
CL<flag>
; Set <flag>
; <flag> <- 1
<flag> := C | D | I
```

Funziona solo sugli stessi flag della ST<flag>: **C**arry, **D**irection e **I**nterrrupt Flag
Instruzioni permesse: CLC, CLD, CLI (**Cl**ear **C**arry, **D**irection, **I**nterrrupt flag)
Esempio:

CLI ; mette a 0 il flag di interrupt (vedi prossimo volume)

Commutazione di flag (CMC)

```
CMC
; Complement Carry
; <carry> <- NOT <carry>
```

Questa istruzione, che esiste solo per il flag di carry, lo fa commutare; se è 0 diventa 1, se 1è 1 diventa 0.

SET

Disponibile solo dal 386 in poi, permette di mettere 1 oppure 0 in un qualsiasi registro da un byte o locazione di memoria, in base del valore corrente del flag indicato con la sua lettera.
Si usa per implementare il tipo boolean nei linguaggi ad alto livello.

Sintassi:

```
SET<cc>  <Destinazione 8 bit>
; Set destination on flag
; <Destinazione 8 bit> <- <cc>
<cc> := O |    S | A | P | C | T | I | D
```

<cc> è la lettera del flag, una di quelle indicate sopra. <Destinazione 8 bit> può essere anche in memoria.
Esempi:

```
SETC AL  ; mette 1 in AL se il flag di carry è 1
         ; (in AL, da 8 bit, va il numero 1 cioè 00000001b)
         ; oppure mette 0 se il flag di carry è 1
SETO [nConSegno] ; può funzionare anche in memoria (in questo caso con
                 ; il flag di overflow)
```

Lettura e modifica di tutto il registro dei flag

Per modificare i flag che non prevedono una specifica istruzione del tipo ST<flag> o CL<flag> esistono, fin dall'8086, due istruzioni che caricano <u>tutti</u> i flag nel registro AH o scrivono nei flag cioè che è attualmente in AH.

```
LAHF
; Load AH with flags
; AH <- SF, ZF, xx, AF, xx, PF, xx, CF
; non modifica i flag
```

LAHF carica nel registro AH (**L**oad in **AH**) gli 8 bit del registro di stato 8086 (**F**LAGS). La posizione dei singoli flag è indicata nel seguente schema:

Bit	7	6	5	4	3	2	1	0
flag	Sign	Zero	XX	Aux. carry	XX	Parity	XX	Carry

```
SAHF
; Store AH into flags
; SF, ZF, xx, AF, xx, PF, xx, CF <- AH
```

SAHF fa il contrario di LAHF: scrive nel registro di stato FLAGS il contenuto di AH. In questo modo modifica, tutti insieme, tutti i flag 8086.

Si noti che non esistono le istruzioni corrispondenti per caricare i flag del registro di stato esteso del 386 (EFLAGS).

Come esempio cancelliamo il flag di zero, che non può essere cambiato con la CL<flag>:

```
LAHF        ; caricamento di tutti i flag
; segue un'istruzione di mascheramento (vedi il paragrafo successivo)
; che lascia tutti i bit di AH come sono, tranne il bit 6, che
; corrisponde al flag di Zero e che viene messo a zero:
AND AH, 10111111b ; azzera il bit del posto di peso 6
SAHF        ; salva tutti i flag, Zero Flag è stato azzerato
```

Altre istruzioni che salvano o modificano i flag sono PUSHF, POPF, PUSHFD e POPFD, che vedremo in seguito.

1.2 Istruzioni per la manipolazione dei bit (bitwise operations)

Nell'ultimo esempio del paragrafo precedente c'era la necessità di azzerare un singolo bit in un byte, senza che gli altri bit fossero influenzati in alcun modo.
Questa è un'esigenza che capita spesso nella programmazione Assembly, specie nei programmi che utilizzano dispositivo di I/O (per questo le istruzioni di questo paragrafo saranno utilizzate molto di frequente nel prossimo volume).
Per trattare i singoli bit di un numero binario abbiamo a disposizione molte istruzioni e possiamo fare molti trucchi, che descriviamo in questo capitolo.

1.2.1 NOT: complemento a uno

Se è necessario commutare tutti i bit di un numero, si usa l'istruzione NOT, che ha la seguente sintassi:

```
NOT <Destinazione>
; Negazione in complemento a uno
; <Destinazione> <- NOT <Destinazione>
; !! non modifica i flag !!
```

Ogni bit a 1 di <Destinazione> diviene uno 0, e viceversa.

Nota importante: come si vede anche nella tabella in appendice, l'istruzione NOT non modifica i flag. Per cui bisogna evitare di mettere una jump condizionata dopo una NOT. Essa funzionerebbe con i flag precedenti!
L'esempio fatto per la NEG, in 1.1.1 a pagina 1, non funzionerebbe con una NOT.

1.2.2 Mascheramenti

Queste istruzioni servono per modificare uno qualsiasi dei bit di una locazione senza cambiare gli altri.
Per fare i mascheramenti si usano istruzioni che eseguono bit per bit le funzioni logiche elementari, AND, OR e XOR.
La sintassi generale delle istruzioni di mascheramento è la seguente:

```
<Mnemonico istruzione di mascheramento> <destinazione>, <maschera>

<Mnemonico istruzione di mascheramento> := AND | OR | XOR
```

Le istruzioni di mascheramento hanno due operandi. Esse sono tre: AND, OR e XOR.
L'operando di destinazione è il valore che si deve modificare, cambiando qualcuno dei suoi bit, mentre il secondo operando, che viene comunemente detto "**maschera**", è un numero che permette di selezionare quali sono i bit da modificare.
Le istruzioni di mascheramento eseguono sugli operandi una funzione logica. La funzione viene valutata bit per bit, secondo la tabella di verità delle funzioni AND, OR o XOR.
Per esempio, se l'istruzione è AND, il bit meno significativo del risultato sarà l'AND fra il bit meno significativo della destinazione ed il bit meno significativo della maschera, e così via per tutti gli altri bit.
Nella spiegazione in dettaglio delle istruzioni di mascheramento indicheremo sempre uno schema mnemonico, concepito per ricordare come funzionano queste istruzioni. In quest'esempio supponiamo per un momento di avere un microprocessore da 4 bit, che esegua le operazioni di mascheramento sui due operandi Op1 e Op2, e che il risultato finisca in Op1.

AND

La funzione logica AND ha la tabella di verità indicata nello Schema 1. L'istruzione Assembly AND esegue la funzione logica AND per ciascuno dei bit dei due operandi coinvolti. La sintassi è la seguente:

```
AND <destinazione>, <maschera>
; reset dei bit che sono a zero nella maschera
<destinazione> <- <destinazione> AND <maschera>
```

Destinazione (prima)	0	1	0	1	**AND**
Maschera	1	1	0	0	=
Risultato (Destinazione dopo)	0	1	0	0	

Tabella di verità di AND:

	0	1
0	0	0
1	0	1

Schema 1: AND schema mnemonico e tabella di verità

Per fare lo schema mnemonico fingiamo di avere un registro da 4 bit e di dover modificare con una maschera il numero 0101b. Se usiamo la maschera 1100b proviamo tutte le combinazioni e possiamo vedere cosa succede al primo operando.
Se analizziamo l'operazione AND vediamo che nei posti della maschera dove c'è zero (a destra nello schema mnemonico) il risultato è comunque zero, mentre dove c'è 1 il risultato è identico a prima.

Perciò AND serve per mettere a zero i bit che nella maschera hanno 0. I bit con 1 nella maschera non cambiano.

AND nell'8086 ammette i consueti operandi; in particolare <maschera> può essere un operando in immediato, una locazione di memoria o un registro.

Esempi:

```
AND DX, 11111101b    ; cancella il bit di peso 2^2

AND DL, 00001111b    ; cancella il nibble più significativo e
                     ; mantiene intatto quello meno significativo
AND ECX, 0FFFFFFFF0h ; cancella il nibble meno significativo di ECX

AND BYTE PTR [SI], 10000b        ; cancella tutti i bit tranne il bit di peso 2^4
```

AND si usa molto spesso per verificare quanto vale un singolo bit di un numero. Se, per esempio, si vuole verificare se il secondo bit più significativo di SI è 1, si può fare così:

```
AND DI, 40h          ; cancella tutti i bit tranne il bit di peso 2^6
JNZ QuelBitEraUno    ; salta se il bit quattro è uno
```

Questo modo di verificare il valore di un bit è interessante perché dopo la AND si può usare subito una jump, senza far prima confronti. Infatti il risultato della AND può essere solamente 0 oppure 40h (01000000h), per cui si può saltare subito se si usa una JZ od una JNZ.

Il controllo senza compare può essere fatto anche su più di un bit alla volta; vediamo come facciamo a controllare se i bit che ci interessano sono <u>tutti</u> a zero:

```
AND AL, 1010000b ; mi interessano il bit 2^4e 2^6
; dopo la AND AL è 0 oppure 1010000b, salto sul flag zero:
JZ  SonoTuttiAzero
```

E' importante tenere bene a mente che l'istruzione AND "distrugge" l'operando destinazione, che non è più uguale a prima. Per questo, se il valore dell'operando destinazione è necessario anche nel prosieguo del programma, sarà necessario salvarlo prima (oppure usare TEST, come si vedrà presto ..).
Si ricorda che l'istruzione AND bit per bit è quella che in C viene espressa con &, mentre && rappresenta l'AND "logico" da usarsi nelle espressioni logiche, come quelle delle istruzioni "if".

TEST

Questa istruzione è come una AND, ma non memorizza il risultato. Modifica solo i flag, che possono essere usati per una jump subito dopo. Visto che TEST non modifica nessuna "destinazione" è facile farne una sequenza, per realizzare strutture di controllo simili alle "Case" o alle "switch" dei linguaggi ad alto livello.

```
TEST <operando>, <maschera>
; Test bit
<operando> AND <maschera>
```

Esempio:

```
TEST AL, 1001101b
JZ SonoTuttiAzero
; se arrivo qui almeno un bit è a uno. Ora guardo qual è:
```

```
    TEST AL, 0000100b
    JNZ Bit2vale1    ; il bit di peso 2 vale 1
    TEST AL, 0001000b
    JNZ Bit3vale1    ; il bit di peso 3 vale 1
    TEST AL, 1000000b
    JNZ Bit7vale1    ; il bit di peso 1 vale 1
    ; se arrivo qui è 1 il bit di peso 0
```

TEST è come AND ma non modifica registri o memoria.

OR

Anche l'istruzione OR esegue la funzione OR bit per bit. La tabella di verità di OR è indicata nello Schema 2.

```
    OR <destinazione>, <maschera>
    ; set dei bit che sono a uno nella maschera
    <destinazione> <- <destinazione> OR <maschera>
```

Destinazione (prima)	0	1	0	1	**OR**
Maschera	1	1	0	0	=
Risultato (Destinazione)	1	1	0	1	

Tabella di verità di OR:	0	1
0	0	1
1	1	1

Schema 2: OR schema mnemonico e tabella di verità

Dallo schema mnemonico si può capire che, quando nella maschera c'è 0 (a destra nello schema mnemonico) i bit del primo operando rimangono uguali, mentre quando c'è 1 divengono 1.

Quindi OR mette a 1 i bit selezionati con un 1 nella maschera, lascia come sono i bit con lo 0.

Esempi:

```
    OR DX, 11111101b     ; mette tutti a uno, tranne il bit di peso 2^2
                         ; che rimane com'era
    OR DL, 00001111b ; mette tutti 1 nel nibble meno significativo e
                         ; mantiene intatto quello più significativo
    OR ECX, 0Fh  ; mette nel nibble meno significativo di ECX

    OR BYTE PTR [SI], 10000b     ; mette 1 nel il bit di peso 2^4
```

Si può notare che le maschere che si usano nelle OR tendono ad essere il contrario di quelle che si userebbero in una AND. Infatti AND non ha nessun effetto sui bit a 1 della maschera, mentre per OR ciò accade sui bit a 0.
Esempio:

```
ResetBit3 EQU    11110111b
SetBit3 EQU          1000b
    ..
    ; qui devo cancellare il bit 3 di AL, lasciando gli altri come sono:
    AND AL, ResetBit3
    ..
    ; qui devo impostare il bit 3 di AL, lasciando gli altri come sono:
    OR AL, SetBit3
    ..
```

La OR non può essere utilizzata per fare semplici test sui singoli bit. Se si vuole fare di simile a quanto illustrato per la AND, si deve usare una CMP in più:

```
    ; voglio vedere se il bit 2^2 è 1, con OR
    OR AL, 11101111b
    ; quo AL vale 11101111b oppure 11111111b
    ; il flag di zero non aiuta, come nella AND!
    ; devo fare una CMP:
    CMP AL, 0FFh
    JE C_era_Uno
C_era_Zero:

    .. parte del programma che tratta il "caso zero"

    JMP Continua
C_era_Uno:

    .. parte del programma che tratta il "caso uno"
```

```
Continua:
```

Un esempio interessante di uso di AND e OR è la "fusione" di due nibble da registri diversi:

```
    AND AL, 11110000b    ; azzera il nibble basso di AL
    AND BL, 00001111b    ; azzera  il nibble alto di BL
    OR AL, BL            ; "fonde" in AL i due nibble
    ; ora nel nibble alto di AL c'è il nibble alto di BL, mentre nel nibble
    ; basso c'è ancora lo stesso nibble basso di prima.
```

XOR

La funzione di OR esclusivo ha una tabella di verità simile alla OR, solo che quando entrambi i bit sono a 1 il risultato è 0.

```
    XOR <destinazione>, <maschera>
    ; toggle dei bit che sono a uno nella maschera
    <destinazione> <- <destinazione> XOR <maschera>
```

Destinazione (prima)	0	1	0	1	**XOR**
Maschera	1	1	0	0	=
Risultato (Destinazione)	1	0	0	1	

Tabella di verità di XOR:	0	1
0	0	1
1	1	0

XOR lascia i bit come sono quando la maschera è zero, mentre cambia ("toggle") il valore dei bit che hanno 1 nella maschera.

Esempio:

```
    ..
    ; cambio il bit di peso 2 di AX
    ; se non ci fosse la XOR dovrei fare così:
    TEST AL, 100B
    JZ EraZero
EraUno: ; lo faccio diventare zero:
    OR AL, 100b
    JMP Continua
EraZero:
    AND AL, 11111011b
Continua:
    ..
```

Un uso particolare della XOR, molto diffuso, è il seguente:

```
    MOV AX, 0        ; azzeramento di un registro

    XOR AX, AX       ; è un modo veloce per azzerare un registro
```

Il risultato che delle due istruzioni precedenti è esattamente lo stesso.
La differenza è che l'operazione con XOR avviene tutta fra registri e non coinvolge operandi immediati da caricare in fase di fetch. Per questo l'istruzione occupa meno memoria ed è più veloce da caricare in fase di fetch, come si può vedere dal confronto dei codici di macchina del seguente esempio:

Istruzione	Ling. Macchina	Istruzione	Ling. Macchina
MOV AX, 0	B8h	XOR AX, AX	33h
	00		C0h
	00		

Un altro modo di azzerare un registro potrebbe essere il seguente:

```
    AND AL, 0
```

ma questo non dà nessun vantaggio rispetto alla MOV AX, 0 ed è molto meno comprensibile di quest'ultima.

Come conseguenza di quanto detto precedentemente queste strane istruzioni non fanno "nulla":

```
    AND AL, 0FFh ; questa istruzione cambia solo i flag
    OR AL, 0     ; questa istruzione cambia solo i flag
    XOR AL, 0    ; questa istruzione cambia solo i flag
```

In verità, dato che cambiano i flag, esse possono essere usate per verificare se un registro è zero o se è negativo, anche se i vantaggi non sono molto grandi ed il codice diventa molto più oscuro.

Vediamo ora un esempio di uso un po' più complicato delle istruzioni di manipolazione dei singoli bit. Il seguente brano di programma, che è utile nella visualizzazione dei numeri in esadecimale, mette nel registro AL il codice ASCII della cifra esadecimale meno significativa del numero che era contenuto inizialmente in AL.

```
    ..
    ; in AL c'è il numero da convertire
    MOV BL, AL      ; copia AL in BL, perché ora devo
                    ; distruggere AL
    AND AL, 1111B   ; cancella il nibble alto
    ; ora in AL c'è il numero binario della cifra esadecimale meno significativa
    ; se la cifra è fra 0 e 9 devo mettere in AL un codice ASCII fra "0" e "9";
    ; se la cifra è fra 10 e 15 devo mettere in AL un codice ASCII fra "A" e "F"
    CMP AL, 9
    ; se è maggiore di 9 => Lettera:
    JA ASCIIlettera
ASCIIcifra:
    ; se arrivo qui devo mettere in AL il codice ASCII della cifra
    ADD AL, "0"
    JMP Avanti
ASCIIlettera:
    ADD AL, "A" - 10 ; in AL va "A" se prima era 10,
                    ; "B" se era 11 .. "F" se era 15
Avanti:
    ..
```

1.2.3 Spostamenti e rotazioni

L'Assembly 8086 mette a disposizione alcune istruzioni che permettono di spostare i bit di un numero in vari modi.

Shift

L'istruzione di shift sposta a destra, verso le cifre meno significative, o a sinistra, verso le più significative, tutti i bit del suo operando. Naturalmente questo spostamento crea un bit libero da una parte e ne espelle un alto dall'alta parte.
Il bit che vene "espulso" finisce nel flag di carry, mentre nel bit libero entra uno zero.

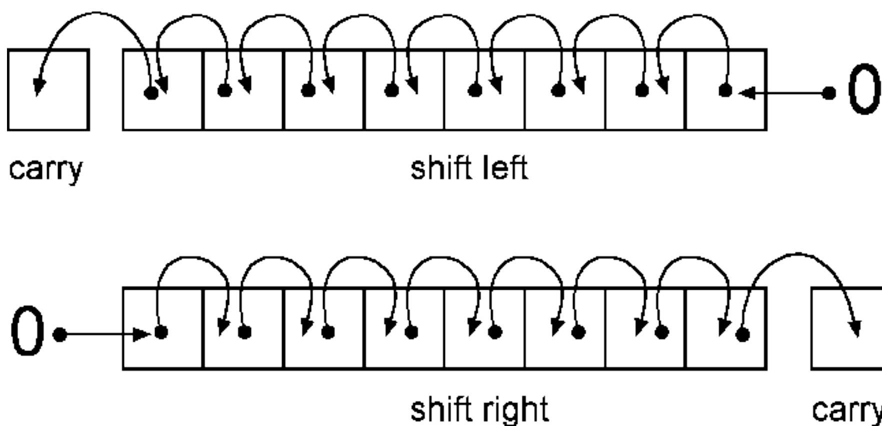


Figura 2: shift a sinistra e a destra

La Figura 2 illustra il funzionamento delle istruzioni di shift, che hanno la seguente sintassi:

```
SHR <destinazione>, <numero di spostamenti>
; Shift Right
; shift a destra di tutti i bit di <destinazione> tante volte
; quanto <numero di spostamenti>
; <destinazione> <- <destinazione> >>

SHL <destinazione>, <numero di spostamenti>
; Shift Left
; shift a sinistra di tutti i bit di <destinazione> tante volte
; quanto <numero di spostamenti>
; <destinazione> <- <destinazione> <<

<numero di spostamenti> := <numero in immediato> | CL
```

Una singola istruzione X86 può fare più di uno spostamento o rotazione, anche se ciò è sottoposto al limitazioni nell'8086. Il numero di spostamenti si può mettere nel registro CL o indicare in immediato nell'istruzione
Nell'8086 l'unico numero che si può usare in immediato è 1, per cui se si vuole fare più di uno spostamento bisogna usare CL.
Una shift da 4 spostamenti è mostrata in Figura 3.

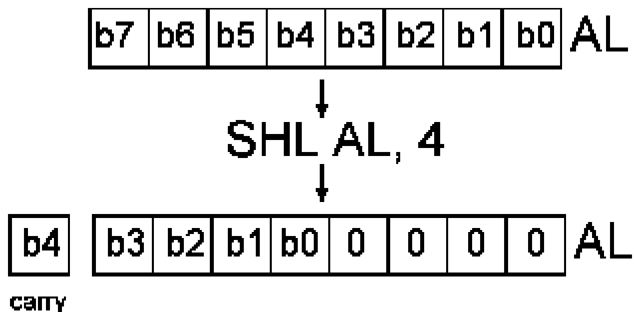


Figura 3: shift "multipla" a sinistra (286>)

Esempi:

```
SHR AL, 1    ; shift a destra di 1 posto in AL
SHL [A], CL  ; shift a sinistra in memoria del numero di posti che c'è in CL
SHR EDX, 12  ; shift a destra di 12 posti
; !! ATTENZIONE: LA PROSSIMA FUNZIONA SOLO CON CPU DALL'80286 IN POI!!
SHL CX, 4    ; shift a sinistra di 4 !! SOLO 286 > !!
SHL [21], CX ; VIETATO, non CX!
SHL EAX, AL  ; VIETATO, solo CL!
```

Una bella proprietà aritmetica dei numeri binari è quella di rendere molto facili le moltiplicazioni e divisioni per 2, 4, 8 e per tutte le potenze di due.
Come nei numeri in base 10 la moltiplicazione per 10 si otteneva aggiungendo alla destra del numero una nuova cifra di valore zero, così accade nel sistema binario per moltiplicare per due.

L'aggiunta di un bit a zero alla "destra" di un numero binario è una moltiplicazione per due.

Esempi:

110b = 6 ; aggiungo uno zero a destra: 1100b = 6 * 2 = 12
1001b = 9 ; aggiungo uno zero a destra: 10010b = 9 * 2 = 18
110b = 6 ; aggiungo due zeri a destra: 11000b = 6 * 4 = 24

Un discorso analogo alla moltiplicazione si può fare per la divisione per potenze di due.

Togliere un bit alla "destra" di un numero binario equivale a dividerlo per due.

Esempi:

1010b = 10 ; tolgo uno zero a destra: 101b = 10 / 2 = 5
1001b = 9 ; tolgo un UNO a destra: 100b = 4 con resto di 1 (il bit "tolto" a destra del numero).
1100b = 12 ; tolgo due cifre a destra: 11b = 12 / 4 = 3
1010b = 10 ; tolgo due cifre a destra: 10b = 10 / 4 = 2 con resto di 2 (10b "tolto" a destra)

Si può concludere che, considerando normali numeri binari, cioè numeri senza segno, l'operazione di shift a sinistra di n posti equivale alla moltiplicazione per 2^n, mentre lo shift a destra è una divisione per 2^n.
Se un numero è con segno ed ha segno negativo, le cose cambiano, perché entra in gioco il complemento a due. L'operazione può comunque essere risolta con uno spostamento di bit, per cui sono state introdotte due istruzioni specifiche che dividono e moltiplicano per potenze di due i numeri con segno.
Le due istruzioni in questione sono SAR e SAL, le operazioni di "shift aritmetico".

```
SAR <destinazione>, <numero di spostamenti>
; Shift Arithmetically Right
; divide <destinazione> per 2^<numero di spostamenti>
; <destinazione> <- <destinazione> / 2^<numero di spostamenti>

SAL <destinazione>, <numero di spostamenti>
; Shift Arithmetically Left
; moltiplica <destinazione> per 2^<numero di spostamenti>
; <destinazione> <- <destinazione> * 2^<numero di spostamenti>
```

```
<numero di spostamenti> := <numero in immediato> | CL
```

SAR e SAL spostano i bit ed aggiungono uno o zero in modo che il risultato sia, aritmeticamente, una divisione od una moltiplicazione con segno per una potenza di due.

Rotazioni

L'operazione di rotazione non ha significato aritmetico ma può essere utile quando è necessario spostare i bit ma non si vuole perdere quelli che verrebbero "espulsi" da una shift.
Rotate funziona come una shift, con la differenza che non "entra" sempre un bit di valore 0, ma "ricircolato" quello che esce dall'altra parte e che viene anche copiato nel carry.

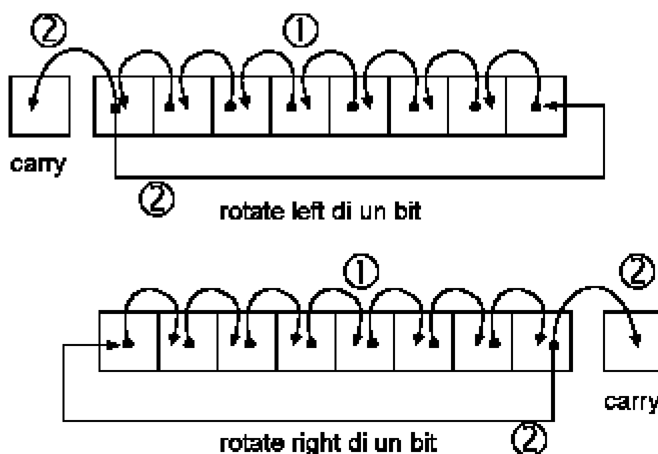
```
ROR <destinazione>, <numero di spostamenti>
; Rotate Right
; ruota i bit a destra

ROL <destinazione>, <numero di spostamenti>
; Rotate Left
; ruota i bit a sinistra

<numero di spostamenti> := <numero in immediato> | CL
```

La Figura 4 illustra il funzionamento dell'istruzione rotate, i numeri indicano che "prima" si fa lo spostamento di tutti i bit, poi si copia il valore che esce sia nel carry che nel primo bit.

Figura 4: rotazioni semplici



Un'istruzione simile alla rotate ma che la "estende" anche al carry è la rotate con carry, che fa "entrare" il valore che è nel flag di carry prima dell'istruzione. Nella Figura 5 è illustrato il funzionamento. I numeri significano che il valore del carry che viene copiato nel bit che entra è quello che vi era prima dell'esecuzione dell'istruzione. Dopo l'esecuzione dell'istruzione il flag di carry assume il valore del bit uscito per lo spostamento.

```
RCR <destinazione>, <numero di spostamenti>
; Rotate with Carry Right
; ruota i bit a destra col carry

RCL <destinazione>, <numero di spostamenti>
; Rotate with Carry Left
; ruota i bit a destra col carry

<numero di spostamenti> := <numero in immediato> | CL
```

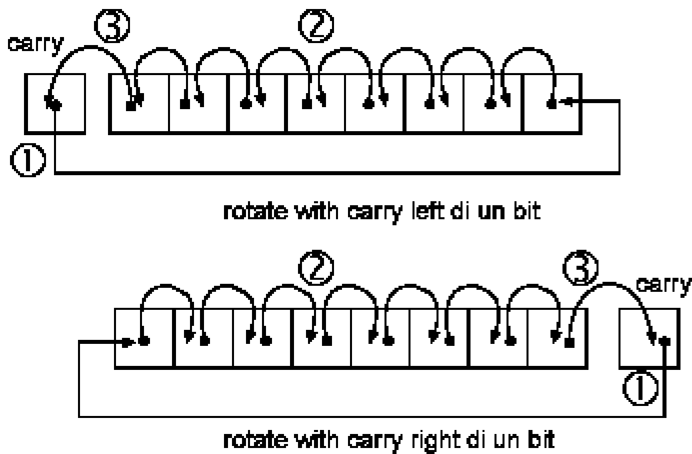


Figura 5: rotazioni con carry

Un esempio di uso di RCL è indicato nel seguente codice:

```
    ..
    ; una "shift" di 4 bit in AX in cui "entrano" 1 invece di zero:
    MOV CX, 4    ; quattro "shift"
QuattroBit:
    STC          ; Set Carry, mette CF = 1
    RCL AX, 1    ; "shift" in cui entra un 1 (CF = 1)
    LOOP QuattroBit
    ..
```

Ora un trucco per cambiare l'ordine dei bit di una word:

```
    ..
    ; cambio l'ordine dei bit di AX
    MOV CX, 16   ; AX ha 16 bit 8-|
SediciBit:
    SHL AX, 1    ; sbatto fuori i bit più significativi, che
                 ; vanno, uno alla volta, nel carry
    RCR BX, 1    ; dal carry vengono messi, uno alla volta,
                 ; in BX, spingendo dall'altra parte
    LOOP SediciBit
    MOV AX, BX
    ..
```

Vediamo ora un esempio che completa uno dei precedenti e visualizza tutte le cifre esadecimali di un numero di 16 bit. In questo esempio si può vedere anche un uso dell'istruzione ROL.

```
    .286 ; utilizzerò una ROL da 4!
    ..
    ; in AX un numero da visualizzare in esadecimale
    ; si lavora su quattro nibble, userò la ROL 4 volte:
    MOV CX, 4
QuattroNibble:
    ROL AX, 4
    ; copio il numero "girato" in DX così lo posso distruggere con AND
    ; (uso DX perché mi serve la sua parte bassa DL)
    MOV DX, AX
    ; lascio solo il nibble meno significativo:
    AND DX, 1111b
    CMP DL, 9
    JA ASCIIlettera
    ; è una cifra: gli aggiungo il codice ASCII della prima cifra:
    ADD DL, "0"
    JMP ScrivoSuMonitor
ASCIIlettera:
    ; è una lettera: gli aggiungo il codice ASCII della prima lettera,
    ; ma devo togliergli 10 perché quando il numero è 10 devo stampare
    ; la lettera A:
    ADD DL, "A" - 10
ScrivoSuMonitor:
    ; salvo AX perché viene rovinato dal servizio DOS che visualizza
    ; il carattere:
```

```
    PUSH AX  ; salva AX nello stack (vedi relativo capitolo)
    MOV AH, 02   ; servizio di visualizzazione del carattere
                 ; il cui codice ASCII è in DL
    INT 21h ; chiamata del servizio DOS
    POP AX   ; ripristina AX dallo stack (vedi relativo capitolo)
    LOOP QuattroNibble
    ; all'uscita, a forza di "girare" AX, è tornato uguale all'inizio
    ..
```

Scansione dei bit di un numero

Le istruzioni di scansione di bit sono presenti dal 386 e permettono di ricercare in un registro se c'è un uno od uno zero. Sintassi:

```
BSF <RegistroDestinazione>, <Sorgente>
```
; **B**it **S**can **F**orward (386>)
; scansione dei bit di <sorgente> da destra a sinistra

```
BSR <RegistroDestinazione>, <Sorgente>
```
; **B**it **S**can **R**everse (386>)
; scansione dei bit di <sorgente> da sinistra a destra

<RegistroDestinazione> può essere un registro a 16 o a 32 bit, <Sorgente> può essere un registro od una locazione di memoria, a 16 o a 32 bit.
L'istruzione BSF scandisce, dal bit 0 "in avanti" (forward), l'operando sorgente e azzera il bit di carry se tutto il sorgente è zero. Se invece c'è almeno un bit a uno il bit di carry viene messo a uno e nell'operando di destinazione finisce il peso del primo bit che è stato trovato a uno.
BSR (**B**it **S**can **R**everse) fa la stessa scansione di BSF, solo che l'ordine di scansione è dal bit più significativo al meno significativo (reverse).

Per realizzare le operazioni compiute da BSF o BSR con un 8086 sarebbe stato necessario scrivere codice piuttosto complesso, come nell'esempio seguente (l'esempio è comunque istruttivo sulle tecniche di programmazione con le istruzioni di spostamento e rotazione):

```
    ; codice 8086 equivalente a BSR AX, BX (386>)
    MOV CX, 16   ; contatore dei bit del ciclo
PerTuttiIbit:
    MOV AX, 1    ; maschera iniziale
    TEST BX, AX  ; guarda se è a uno il bit di BX che corrisponde a quello a uno
                 ; della maschera
    JNZ Trovato
    SHL AX       ; sposta a sinistra l'uno della maschera
    LOOP PerTuttiIbit
NonTrovato:      ; se giungo qui nessun bit di BX è ON
    OR CX, 0FFFFh; solo perché il flag Z sia zero (il risultato è diverso da zero)
                 ; (si modifica anche CX, ma il suo valore non ci interessa più)
    JMP Prosegui
Trovato:
    ; se salto qui vuol dire che in CX c'è il numero del bit trovato + 1
    DEC CX       ; CX <- numero del bit trovato
    MOV AX, CX   ; alla fine quel numero deve andare in AX
    XOR CX, CX   ; istruzione per mettere ON il flag di zero
                 ; (cancella anche CX, ma non importa)
    OR CX, 1         ; solo perché il flag Z sia uno si cancella CX, ma
                 ; non importa)
Prosegui:
    ; il risultato è identico a quello dell'istruzione BSR AX, BX,
    ; l'unica differenza è che questo codice mette -1 nel registro CX
    ; se BX era a zero e vi mette 0 altrimenti
```