Assembly base con MASM e TASM

< Sommario >

Capitolo 19 Istruzioni logiche

Nei precedenti capitoli, abbiamo lasciato in sospeso alcune questioni che non potevano essere affrontate con l'ausilio delle sole istruzioni che già conosciamo (istruzioni aritmetiche e di trasferimento dati); tali questioni riguardavano, principalmente, la necessità di poter operare sui singoli bit di un numero binario.

Abbiamo visto, ad esempio, che moltiplicando tra loro due numeri interi di ampiezza arbitraria, si ottengono una serie di prodotti parziali, che devono essere poi sommati tra loro in modo da ottenere il prodotto finale; prima di poter effettuare tale somma, sappiamo però che è necessario sottoporre a scorrimento verso sinistra, le singole cifre dei prodotti parziali stessi. Come si effettua una tale operazione?

Nel precedente capitolo, invece, abbiamo visto che per convertire in binario un numero **Unpacked BCD**, possiamo servirci del metodo delle divisioni successive, con divisore **2**; in questo modo, otteniamo una sequenza di resti, ciascuno dei quali rappresenta una delle cifre del numero appena convertito in binario. Come facciamo a memorizzare tali cifre, nei singoli bit di un numero binario?

Per poter rispondere a queste domande, dobbiamo studiare in dettaglio una importante categoria di istruzioni della **CPU**, che vengono definite: **istruzioni logiche**; si tratta di istruzioni estremamente potenti, che permettono di effettuare, ad esempio, scorrimenti, rotazioni, test, inversioni, e numerose altre operazioni, sui singoli bit di un numero binario.
Grazie a queste particolari caratteristiche, le istruzioni logiche vengono largamente impiegate dai programmatori **Assembly**, per realizzare svariati trucchi di programmazione; a tale proposito, in questo capitolo vedremo numerosi esempi pratici.

Appare intuitivo il fatto che il principio di funzionamento delle istruzioni logiche, è strettamente legato ai concetti esposti nei capitoli 3, 4, 5 e 6; eventualmente, una rapida rilettura di tali capitoli, può aiutare a capire meglio ciò che verrà esposto nel seguito.

Come accade per le istruzioni aritmetiche, anche per le istruzioni logiche è proibito l'uso di operandi di tipo **SegReg**; nel seguito del capitolo quindi, quando si parla di generici registri, ci si riferisce ai registri generali della **CPU**.

19.1 L'istruzione AND

Con il mnemonico **AND** si indica l'istruzione **logical AND** (**AND** logico); lo scopo di questa istruzione è quello di effettuare un **AND** logico tra i contenuti dei due operandi, **SRC** e **DEST**, specificati esplicitamente dal programmatore.
Sono permesse tutte le combinazioni tra **SRC** e **DEST**, ad eccezione di quelle illegali o prive di senso; possiamo avere quindi i seguenti casi:

```
AND   Reg, Reg
AND   Reg, Mem
AND   Mem, Reg
AND   Reg, Imm
AND   Mem, Imm
```

L'istruzione **AND** opera bit per bit (**bitwise**); ciò significa che ciascun bit di **DEST**, viene sottoposto ad un **AND** con il bit corrispondente (stessa posizione) di **SRC**. Affinché sia possibile un confronto di tipo **bitwise**, i due operandi devono avere la stessa ampiezza in bit; il risultato prodotto da **AND**, viene memorizzato nell'operando **DEST**.

Riprendendo ciò che è stato esposto nel Capitolo 5, consideriamo due **proposizioni logiche**, indicate simbolicamente con **A** e **B**; associamo, inoltre, il simbolo **1** alla proposizione logica vera (**TRUE**), e il simbolo **0** alla proposizione logica falsa (**FALSE**).
La composizione delle due proposizioni logiche attraverso l'operatore **AND** si scrive:

```
A AND B
```

e **risulta vera se, e solo se, entrambe le proposizioni sono vere**.

Poniamo, ad esempio:

```
A = La capitale della Francia è Parigi
```

e:

```
B = La capitale dell'Italia è Berlino
```

In questo caso otteniamo:

```
A AND B = 0
```

Infatti, la proposizione **A** è vera, mentre la proposizione **B** è falsa; in sostanza, alla precedente relazione dobbiamo attribuire il significato di:

```
A e B
```

In base alle considerazioni appena esposte, possiamo ricavare la tabella della verità (**truth table**) relativa all'operatore **AND** logico; nella parte sinistra della Figura 1 viene mostrato anche il simbolo logico di questo operatore.

Figura 1 - AND logico

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

Come esempio pratico, poniamo **BX=1000111010100110b**, **CX=0110100111110010b**, ed eseguiamo la seguente istruzione:

```
and bx, cx
```

In presenza di questa istruzione, la **CPU** ottiene il seguente risultato:

```
            BX = 1000111010100110b
            CX = 0110100111110010b
 ---------------------------------
BX = BX AND CX = 0000100010100010b
```

Per pervenire a questo risultato, la **CPU** esegue le seguenti elaborazioni (a partire dai bit meno significativi dei due operandi):

0 AND **0** = **0** (**AND** tra i bit in posizione **0**)

1 AND **1** = **1** (**AND** tra i bit in posizione **1**)

1 AND **0** = **0** (**AND** tra i bit in posizione **2**)

e così via.

19.1.1 Istruzione AND con maschere di bit

L'istruzione **AND** può essere utilizzata per portare a livello logico **0**, determinati bit di un numero binario; osserviamo, infatti, che indipendentemente dal valore (**0** o **1**) di una proposizione logica **A**, risulta:

A AND **0** = **0** e A AND **1** = A

In sostanza, un **AND** tra **A** e **0** produce, in ogni caso, **A=0**; invece, un **AND** tra **A** e **1** lascia inalterato il contenuto di **A**.
In gergo tecnico, l'azione che consiste nel portare a livello logico **0** un bit di un numero binario, si definisce **clearing** (lucidare, rendere trasparente).

Come esempio pratico poniamo **AX=1001110010000011b**, e supponiamo di voler portare a livello logico **0**, i bit di **AX** in posizione **1**, **7**, **11** e **15**; prima di tutto, definiamo la seguente costante simbolica:

```
BITMASK_AND = 0111011101111101b
```

Come si può notare, gli unici bit di **BITMASK_AND** che valgono **0**, sono proprio quelli in posizione **1**, **7**, **11** e **15**; a questo punto, possiamo scrivere l'istruzione:

```
and ax, BITMASK_AND
```

In presenza di questa istruzione, la **CPU** ottiene il seguente risultato:

```
                    AX = 1001110010000011b
           BITMASK_AND = 0111011101111101b
 ----------------------------------------
 AX = AX AND BITMASK_AND = 0001010000000001b
```

Come possiamo notare, i bit di **AX** in posizione **1**, **7**, **11** e **15** sono stati portati a **0**, mentre tutti gli altri sono rimasti inalterati; una costante simbolica come **BITMASK_AND**, viene definita **bit mask** (maschera di bit).

19.1.2 AND logico tra operandi di ampiezza arbitraria

L'istruzione **AND** opera bit per bit, per cui può essere applicata con estrema semplicità, anche ad operandi di ampiezza arbitraria; in sostanza, non dobbiamo fare altro che suddividere ciascun operando, in tante parti direttamente gestibili dalla **CPU**.
Se, ad esempio, abbiamo a che fare con operandi a **128** bit, possiamo suddividere ciascuno di essi in **4 DWORD**; a questo punto, dobbiamo effettuare un **AND** tra ciascuna coppia di **DWORD** corrispondenti.

Consideriamo le seguenti definizioni:

```
Num1     dd    3CF218A6h, 2B96CDEAh, 6F9C482Dh, 881E63CAh
Num2     dd    48FB1DC9h, 66D41695h, 3C9DF1AFh, 7BB3F28Dh
Num3     dd    4 dup (0)
```

Se vogliamo ottenere:

```
Num3 = Num1 AND Num2
```

non dobbiamo fare altro che tradurre in **Assembly** le seguenti pseudo-istruzioni:

```
Num3[0] = Num1[0] AND Num2[0] = 3CF218A6h AND 48FB1DC9h = 08F21880h

Num3[4] = Num1[4] AND Num2[4] = 2B96CDEAh AND 66D41695h = 22940480h

Num3[8] = Num1[8] AND Num2[8] = 6F9C482Dh AND 3C9DF1AFh = 2C9C402Dh

Num3[12] = Num1[12] AND Num2[12] = 881E63CAh AND 7BB3F28Dh = 08126288h
```

Ovviamente, questo procedimento non è applicabile ai numeri **BCD** (e ciò vale per tutte le altre istruzioni logiche illustrate in questo capitolo); in una eventualità del genere, dobbiamo prima convertire i numeri **BCD** in binario!

19.1.3 Effetti provocati da AND sugli operandi e sui flags

L'esecuzione dell'istruzione **AND** tra **SRC** e **DEST**, modifica il contenuto del solo operando **DEST** che viene sovrascritto dal risultato prodotto dall'istruzione stessa; il contenuto dell'operando **SRC** rimane inalterato.
Bisogna però prestare particolare attenzione ad istruzioni del tipo:

```
and si, [si]
```

Dopo l'esecuzione di questa istruzione, la componente **Offset** contenuta in **SI**, viene sovrascritta dal risultato prodotto da **AND**!

La possibilità di effettuare un **AND** tra **Reg** e **Reg**, ci permette di scrivere anche istruzioni del tipo:

```
and dx, dx
```

Come si può facilmente constatare, l'esecuzione di una tale istruzione non provoca nessuna modifica del contenuto di **DX**; infatti:

```
0 AND 0 = 0 e 1 AND 1 = 1
```

Sfruttando questo aspetto, possiamo utilizzare **AND** per sapere se il contenuto di un registro vale zero; supponiamo, ad esempio, di avere **DX=0111001101001101b**. Eseguendo ora l'istruzione:

```
and dx, dx
```

otteniamo:

```
            DX = 0111001101001101b
            DX = 0111001101001101b
 ---------------------------------
DX = DX AND DX = 0111001101001101b
```

In sostanza, il risultato della precedente istruzione è zero se, e solo se, tutti i bit di **DX** valgono **0**; se almeno un bit di **DX** vale **1**, il risultato della precedente istruzione è diverso da zero. Come viene spiegato più avanti, per sapere se abbiamo ottenuto un risultato nullo, ci basta consultare **ZF**; è importante anche ribadire che la precedente istruzione, non altera il contenuto di **DX**!

L'esecuzione dell'istruzione **AND**, modifica i campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; il solo campo **AF** assume un valore indeterminato, e non ha quindi nessun significato.
La **CPU** pone, in ogni caso, **CF=0** e **OF=0**, mentre i campi **PF**, **ZF** e **SF** forniscono informazioni ordinarie relative al risultato prodotto da **AND**; in sostanza, **PF** indica se i primi **8** bit del risultato contengono un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il risultato vale zero, **SF** indica se il bit più significativo del risultato vale **1**.

19.2 L'istruzione TEST

Con il mnemonico **TEST** si indica l'istruzione **logical compare** (comparazione logica mediante **AND**); lo scopo di questa istruzione è quello di effettuare una comparazione di tipo **AND** logico, tra i contenuti dei due operandi, **SRC** e **DEST**, specificati esplicitamente dal programmatore.
Sono permesse tutte le combinazioni tra **SRC** e **DEST**, ad eccezione di quelle illegali o prive di senso; possiamo avere quindi i seguenti casi:

```
TEST  Reg, Reg
TEST  Reg, Mem
TEST  Mem, Reg
TEST  Reg, Imm
TEST  Mem, Imm
```

L'istruzione **TEST** opera bit per bit (**bitwise**); ciò significa che ciascun bit di **DEST**, viene sottoposto ad un **AND** con il bit corrispondente (stessa posizione) di **SRC**. Affinché sia possibile un confronto di tipo **bitwise**, i due operandi devono avere la stessa ampiezza in bit; il risultato prodotto da **TEST**, viene scartato.

In sostanza, l'istruzione **TEST** è del tutto simile a **AND**; la differenza fondamentale sta nel fatto che il risultato prodotto da **TEST**, viene scartato. Il contenuto dell'operando **DEST** non subisce quindi nessuna modifica; in effetti, in analogia a **CMP** (comparazione aritmetica), l'istruzione **TEST** (comparazione logica) è stata concepita proprio per permettere di "testare" lo stato di determinati bit dell'operando **DEST**, senza alterarne il contenuto.
Ad eccezione di questo aspetto, tutto ciò che è stato esposto in relazione all'istruzione **AND** si applica quindi anche all'istruzione **TEST**.

19.2.1 Istruzione TEST con maschere di bit

Come è stato appena spiegato, l'istruzione **TEST** è stata concepita per permettere di "testare" lo stato di determinati bit dell'operando **DEST**, senza alterarne il contenuto; vediamo alcuni esempi pratici.

Poniamo **BH=01101101b**, e supponiamo di voler sapere se il bit in posizione **3** di **BH** vale **1**; a tale proposito, possiamo servirci di una maschera di bit attraverso l'istruzione:

```
test bh, 00001000b
```

Come si può notare, solo il bit in posizione **3** della maschera di bit, vale **1**; in base a quanto è stato esposto per l'istruzione **AND**, possiamo affermare che la precedente istruzione produce un risultato diverso da zero se, e solo se, il bit in posizione **3** di **BH** vale **1**. Se, e solo se, il bit in posizione **3** di **BH** vale **0**, otteniamo un risultato nullo; come viene chiarito più avanti, per conoscere il risultato prodotto dalla precedente istruzione, possiamo servirci del flag **ZF**.

Dalle considerazioni appena esposte, si intuisce che possiamo utilizzare l'istruzione **TEST**, anche per sapere se un numero intero con segno in complemento a **2**, è positivo o negativo; a tale proposito, dobbiamo osservare innanzi tutto che il bit di segno è quello più significativo.
Per sapere allora se il registro **AL** contiene un numero positivo o negativo, ci basta scrivere l'istruzione:

```
test al, 10000000b
```

o, in modo equivalente:

```
test al, 80h
```

Questa istruzione fornisce un risultato nullo se, e solo se, il bit più significativo di **AL** vale **0** (numero positivo); se, e solo se, il bit più significativo di **AL** vale **1** (numero negativo), otteniamo un risultato diverso da zero.
Come viene chiarito più avanti, per conoscere il risultato prodotto dalla precedente istruzione, possiamo servirci del flag **ZF** o di **SF**.

Osserviamo che, se nei precedenti due esempi avessimo utilizzato l'istruzione **AND**, avremmo alterato il contenuto di **DEST**; utilizzando, invece, **TEST**, possiamo svolgere questo tipo di verifiche in modo assolutamente sicuro.

19.2.2 TEST tra operandi di ampiezza arbitraria

L'istruzione **TEST** opera bit per bit, per cui può essere applicata con estrema semplicità, anche ad operandi di ampiezza arbitraria; in sostanza, non dobbiamo fare altro che suddividere ciascun operando, in tante parti direttamente gestibili dalla **CPU**.

Supponiamo, ad esempio, di voler sapere se il contenuto a **96** bit di un dato chiamato **BigNum1**, rappresenta un numero positivo o negativo (in complemento a **2**); come al solito, dobbiamo partire dal presupposto che il bit di segno è sempre quello più significativo.
In base al fatto che **BigNum1** è formato da **96** bit, cioè da **12** byte, possiamo scrivere la semplice istruzione:

```
test byte ptr BigNum1[11], 80h
```

19.2.3 Effetti provocati da TEST sugli operandi e sui flags

L'esecuzione dell'istruzione **TEST** tra **SRC** e **DEST**, preserva il contenuto di entrambi gli operandi; infatti, il risultato del test viene scartato.

La possibilità di effettuare un **TEST** tra **Reg** e **Reg**, ci permette di scrivere anche istruzioni del tipo:

```
test dx, dx
```

In questo caso, valgono tutte le considerazioni già esposte per l'istruzione **AND**.

L'esecuzione dell'istruzione **TEST**, modifica i campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; il solo campo **AF** assume un valore indeterminato, e non ha quindi nessun significato.
La **CPU** pone, in ogni caso, **CF=0** e **OF=0**, mentre i campi **PF**, **ZF** e **SF** forniscono informazioni ordinarie relative al risultato prodotto da **TEST**; in sostanza, **PF** indica se i primi **8** bit del risultato contengono un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il risultato vale zero, **SF** indica se il bit più significativo del risultato vale **1**.

19.3 L'istruzione NOT

Con il mnemonico **NOT** si indica l'istruzione **one's complement negation** (negazione in complemento a uno); lo scopo di questa istruzione è quello di invertire lo stato di tutti i bit dell'unico operando **DEST**, specificato esplicitamente dal programmatore.
Non essendo possibile modificare il contenuto di un **Imm**, le uniche forme lecite per l'istruzione **NOT** sono le seguenti:

```
NOT   Reg
NOT   Mem
```

L'istruzione **NOT** opera bit per bit (**bitwise**); ciò significa che ciascun bit di **DEST**, viene sottoposto ad una inversione (se vale **0** diventa **1** e viceversa). Il risultato prodotto da **NOT**, viene memorizzato nello stesso operando **DEST**.

Il complemento a uno di una proposizione logica **A** si scrive:

NOT A

e **risulta vera se, e solo se, la proposizione è falsa**.

Poniamo, ad esempio:

A = La capitale della Francia è Parigi

In questo caso otteniamo:

NOT A = 0

Infatti, stiamo negando una proposizione vera; stiamo affermando cioè, che la capitale della Francia **NON** è Parigi.

Poniamo, ad esempio:

A = La capitale dell'Italia è Berlino

In questo caso otteniamo:

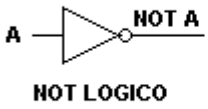
NOT A = 1

Infatti, stiamo negando una proposizione falsa; stiamo affermando cioè, che la capitale dell'Italia **NON** è Berlino.
In sostanza, alle precedenti relazioni dobbiamo assegnare il significato di:

negazione di A

In base alle considerazioni appena esposte, possiamo ricavare la tabella della verità (**truth table**) relativa all'operatore **NOT** logico; nella parte sinistra della Figura 2 viene mostrato anche il simbolo logico di questo operatore.

Figura 2 - NOT logico



A	NOT A
0	1
1	0

Come esempio pratico, poniamo **CX=1000111010100110b**, ed eseguiamo la seguente istruzione:

```
not cx
```

In presenza di questa istruzione, la **CPU** ottiene il seguente risultato:

```
         CX = 1000111010100110b
 ---------------------------------
 CX = NOT CX = 0111000101011001b
```

partire dal bit meno significativo dell'operando):

NOT 0 = 1 (**NOT** del bit in posizione **0**)

NOT 1 = 0 (**NOT** del bit in posizione **1**)

NOT 1 = 0 (**NOT** del bit in posizione **2**)

e così via.

19.3.1 Complemento a 1 di un operando di ampiezza arbitraria

L'istruzione **NOT** opera bit per bit, per cui può essere applicata con estrema semplicità, anche ad operandi di ampiezza arbitraria; in sostanza, non dobbiamo fare altro che suddividere l'operando, in tante parti direttamente gestibili dalla **CPU**.
Consideriamo, ad esempio, la seguente definizione:

```
BigNum1 dd 3CF28615h, 8FB489DDh, 1C8DE4FFh
```

Se vogliamo ottenere il complemento a uno di **BigNum1**, non dobbiamo fare altro che tradurre in **Assembly** le seguenti pseudo-istruzioni:

BigNum1[0] = NOT BigNum1[0] = NOT 3CF28615h = C30D79EAh

BigNum1[4] = NOT BigNum1[4] = NOT 8FB489DDh = 704B7622h

BigNum1[8] = NOT BigNum1[8] = NOT 1C8DE4FFh = E3721B00h

19.3.2 Complemento a 2 di un operando di ampiezza arbitraria

Supponiamo di aver definito un dato **Num1** il cui contenuto rappresenta un numero intero con segno in complemento a **2**; a questo punto, vogliamo cambiare di segno lo stesso contenuto di **Num1**.
Come sappiamo, se l'ampiezza in bit di **Num1** è direttamente gestibile dalla **CPU**, possiamo servirci semplicemente della sola istruzione **NEG**; se, invece, l'ampiezza in bit di **Num1** eccede la capacità massima dei registri della **CPU**, possiamo sfruttare il fatto che, come è stato già dimostrato nei precedenti capitoli:

NEG(Num1) = NOT(Num1) + 1

Come esempio pratico, supponiamo di operare sui numeri interi con segno a **64** bit; in tal caso, possiamo rappresentare tutti i numeri interi con segno compresi tra il limite minimo:

$-(2^{64} / 2)$

e il limite massimo:

$+((2^{64} / 2) - 1) = +(2^{63} - 1)$

Come al solito, il bit di segno è quello più significativo (che in questo caso è il bit in posizione **63**); tutti i numeri il cui bit più significativo vale **0** sono positivi, mentre tutti i numeri il cui bit più significativo vale **1** sono negativi.

Consideriamo ora il numero **-3**, che a **64** bit in complemento a **2** si scrive:

2^{64} - 3 = 18446744073709551616 - 3 = 18446744073709551613 = FFFFFFFFFFFFFFFDh

Negando ora questo numero, dovremmo ottenere **+3**, cioè **0000000000000003h**; in base a quanto è stato detto in precedenza, per negare un numero intero con segno a **64** bit, dobbiamo prima sottoporlo ad un **NOT**, sommando poi **1** al risultato ottenuto.

Partiamo quindi con la seguente definizione:

```
Num1 dq -3 ; FFFFFFFFFFFFFFFDh
```

Il seguente codice effettua il cambiamento di segno di **-3**:

```
; complemento a 1 di Num1

not     dword ptr Num1[0]    ; Num1[0] = NOT FFFFFFFDh = 00000002h
not     dword ptr Num1[4]    ; Num1[4] = NOT FFFFFFFFh = 00000000h

; incremento di 1 del risultato (complemento a 2 di Num1)

add     dword ptr Num1[0], 1 ; Num1[0] = 00000002h + 1h = 00000003h
adc     dword ptr Num1[4], 0 ; Num1[4] = 00000000h + 0h = 00000000h
```

È importante ricordare che in questo caso, per incrementare di **1** il contenuto di **Num1[0]**, non possiamo utilizzare **INC**; infatti, l'istruzione **INC** non modifica il flag **CF** di cui ha bisogno la successiva istruzione **ADC**!

Se vogliamo visualizzare il contenuto esadecimale di **Num1**, possiamo procedere nel solito modo con l'ausilio di **writeHex32**; in questo caso, l'output deve iniziare da **Num1[4]**.

19.3.3 Effetti provocati da NOT sugli operandi e sui flags

L'esecuzione dell'istruzione **NOT** modifica il contenuto del suo unico operando **DEST**; tale operando viene sovrascritto dal risultato prodotto dall'istruzione stessa.

L'esecuzione dell'istruzione **NOT**, non modifica nessun campo del **Flags Register**.

19.4 L'istruzione OR

Con il mnemonico **OR** si indica l'istruzione **logical inclusive OR** (**OR** logico inclusivo); lo scopo di questa istruzione è quello di effettuare un **OR** logico inclusivo tra i contenuti dei due operandi, **SRC** e **DEST**, specificati esplicitamente dal programmatore.
Sono permesse tutte le combinazioni tra **SRC** e **DEST**, ad eccezione di quelle illegali o prive di senso; possiamo avere quindi i seguenti casi:

```
OR    Reg, Reg
OR    Reg, Mem
OR    Mem, Reg
OR    Reg, Imm
OR    Mem, Imm
```

L'istruzione **OR** opera bit per bit (**bitwise**); ciò significa che ciascun bit di **DEST**, viene sottoposto ad un **OR** con il bit corrispondente (stessa posizione) di **SRC**. Affinché sia possibile un confronto di tipo **bitwise**, i due operandi devono avere la stessa ampiezza in bit; il risultato prodotto da **OR**, viene memorizzato nell'operando **DEST**.

La composizione di due proposizioni logiche **A** e **B** attraverso l'operatore **OR** si scrive:

A OR B

e **risulta vera se, e solo se, almeno una di esse è vera**.

Poniamo, ad esempio:

A = La capitale della Francia è Parigi

e:

B = La capitale dell'Italia è Berlino

In questo caso otteniamo:

A OR B = 1

Infatti, almeno una delle due proposizioni (la **A**) è vera; in sostanza, all'operatore **OR** dobbiamo attribuire il significato di:

A o B o entrambe

In base alle considerazioni appena esposte, possiamo ricavare la tabella della verità (**truth table**) relativa all'operatore **OR** logico inclusivo; nella parte sinistra della Figura 3 viene mostrato anche il simbolo logico di questo operatore.

Figura 3 - OR logico inclusivo

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

Come esempio pratico, poniamo **BX=1000111010100110b**, **CX=0110100111110010b**, ed eseguiamo la seguente istruzione:

```
or bx, cx
```

In presenza di questa istruzione, la **CPU** ottiene il seguente risultato:

```
BX = 1000111010100110b
CX = 0110100111110010b
```

```
---------------------------------
BX = BX OR CX = 1110111111110110b
```

Per pervenire a questo risultato, la **CPU** esegue le seguenti elaborazioni (a partire dai bit meno significativi dei due operandi):

`0 OR 0 = 0` (**OR** tra i bit in posizione **0**)

`1 OR 1 = 1` (**OR** tra i bit in posizione **1**)

`1 OR 0 = 1` (**OR** tra i bit in posizione **2**)

e così via.

19.4.1 Istruzione OR con maschere di bit

L'istruzione **OR** può essere utilizzata per portare a livello logico **1**, determinati bit di un numero binario; osserviamo, infatti, che indipendentemente dal valore (**0** o **1**) di una proposizione logica **A**, risulta:

`A OR 0 = A` e `A OR 1 = 1`

In sostanza, un **OR** tra **A** e **0** lascia inalterato il contenuto di **A**; invece, un **OR** tra **A** e **1** produce, in ogni caso, **A=1**.
In gergo tecnico, l'azione che consiste nel portare a livello logico **1** un bit di un numero binario, si definisce **setting** (settaggio).

Come esempio pratico poniamo **AX=1001110010000011b**, e supponiamo di voler portare a livello logico **1**, i bit di **AX** in posizione **3**, **4**, **5** e **8**; prima di tutto, definiamo la seguente costante simbolica:

```
BITMASK_OR = 0000000100111000b
```

Come si può notare, gli unici bit di **BITMASK_OR** che valgono **1**, sono proprio quelli in posizione **3**, **4**, **5** e **8**; a questo punto, possiamo scrivere l'istruzione:

```
or ax, BITMASK_OR
```

In presenza di questa istruzione, la **CPU** ottiene il seguente risultato:

```
                  AX = 1001110010000011b
          BITMASK_OR = 0000000100111000b
---------------------------------------
AX = AX OR BITMASK_OR = 1001110110111011b
```

Come possiamo notare, i bit di **AX** in posizione **3**, **4**, **5** e **8** sono stati portati a **1**, mentre tutti gli altri sono rimasti inalterati.

19.4.2 OR logico inclusivo tra operandi di ampiezza arbitraria

L'istruzione **OR** opera bit per bit, per cui può essere applicata con estrema semplicità, anche ad operandi di ampiezza arbitraria; in sostanza, non dobbiamo fare altro che suddividere ciascun operando, in tante parti direttamente gestibili dalla **CPU**.
Se, ad esempio, abbiamo a che fare con operandi a **128** bit, possiamo suddividere ciascuno di essi in **4 DWORD**; a questo punto, dobbiamo effettuare un **OR** tra

ciascuna coppia di **DWORD** corrispondenti.

Consideriamo le seguenti definizioni:

```
Num1     dd    3CF218A6h, 2B96CDEAh, 6F9C482Dh, 881E63CAh
Num2     dd    48FB1DC9h, 66D41695h, 3C9DF1AFh, 7BB3F28Dh
Num3     dd    4 dup (0)
```

Se vogliamo ottenere:

```
Num3 = Num1 OR Num2
```

non dobbiamo fare altro che tradurre in **Assembly** le seguenti pseudo-istruzioni:

```
Num3[0] = Num1[0] OR Num2[0] = 3CF218A6h OR 48FB1DC9h = 7CFB1DEFh

Num3[4] = Num1[4] OR Num2[4] = 2B96CDEAh OR 66D41695h = 6FD6DFFFh

Num3[8] = Num1[8] OR Num2[8] = 6F9C482Dh OR 3C9DF1AFh = 7F9DF9AFh

Num3[12] = Num1[12] OR Num2[12] = 881E63CAh OR 7BB3F28Dh = FBBFF3CFh
```

19.4.3 Effetti provocati da OR sugli operandi e sui flags

L'esecuzione dell'istruzione **OR** tra **SRC** e **DEST**, modifica il contenuto del solo operando **DEST** che viene sovrascritto dal risultato prodotto dall'istruzione stessa; il contenuto dell'operando **SRC** rimane inalterato.
Bisogna però prestare particolare attenzione ad istruzioni del tipo:

```
or bx, cs:[bx+si+03h]
```

Dopo l'esecuzione di questa istruzione, la componente **Offset** contenuta in **BX**, viene sovrascritta dal risultato prodotto da **OR**!

La possibilità di effettuare un **OR** tra **Reg** e **Reg**, ci permette di scrivere anche istruzioni del tipo:

```
or dx, dx
```

Come si può facilmente constatare, l'esecuzione di una tale istruzione non provoca nessuna modifica del contenuto di **DX**; infatti:

```
0 OR 0 = 0 e 1 OR 1 = 1
```

Sfruttando questo aspetto, possiamo utilizzare **OR** per sapere se il contenuto di un registro vale zero; in tal caso, si applicano tutte le considerazioni già svolte per l'istruzione **AND**.

L'esecuzione dell'istruzione **OR**, modifica i campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; il solo campo **AF** assume un valore indeterminato, e non ha quindi nessun significato.
La **CPU** pone, in ogni caso, **CF=0** e **OF=0**, mentre i campi **PF**, **ZF** e **SF** forniscono informazioni ordinarie relative al risultato prodotto da **OR**; in sostanza, **PF** indica se i primi **8** bit del risultato contengono un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il risultato vale zero, **SF** indica se il bit più

significativo del risultato vale **1**.

19.5 L'istruzione XOR

Con il mnemonico **XOR** si indica l'istruzione **logical eXclusive OR** (**OR** logico esclusivo); lo scopo di questa istruzione è quello di effettuare un **OR** logico esclusivo tra i contenuti dei due operandi, **SRC** e **DEST**, specificati esplicitamente dal programmatore.
Sono permesse tutte le combinazioni tra **SRC** e **DEST**, ad eccezione di quelle illegali o prive di senso; possiamo avere quindi i seguenti casi:

```
XOR   Reg, Reg
XOR   Reg, Mem
XOR   Mem, Reg
XOR   Reg, Imm
XOR   Mem, Imm
```

L'istruzione **XOR** opera bit per bit (**bitwise**); ciò significa che ciascun bit di **DEST**, viene sottoposto ad uno **XOR** con il bit corrispondente (stessa posizione) di **SRC**. Affinché sia possibile un confronto di tipo **bitwise**, i due operandi devono avere la stessa ampiezza in bit; il risultato prodotto da **XOR**, viene memorizzato nell'operando **DEST**.

La composizione di due proposizioni logiche **A** e **B** attraverso l'operatore **XOR** si scrive:

A XOR B

e **risulta vera se, e solo se, una di esse è vera e l'altra è falsa**.

Poniamo, ad esempio:

A = La capitale della Francia è Parigi

e:

B = La capitale dell'Italia è Roma

In questo caso otteniamo:

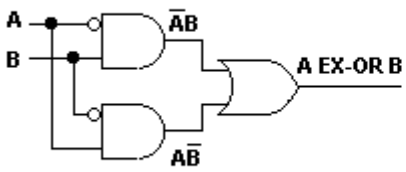
A XOR B = 0

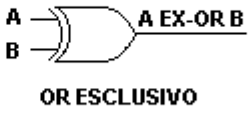
Infatti, entrambe le proposizioni sono vere; in sostanza, all'operatore **XOR** dobbiamo attribuire il significato di:

(A e NOT(B)) o (NOT(A) e B)

In base alle considerazioni appena esposte, possiamo ricavare la tabella della verità (**truth table**) relativa all'operatore **OR** logico esclusivo; nella parte centrale della Figura 4 viene mostrato anche il simbolo logico di questo operatore, mentre nella parte sinistra vediamo una possibile realizzazione dello **XOR** attraverso porte **AND**, **OR** e **NOT** (la porta **NOT** viene rappresentata mediante un cerchietto).

Figura 4 - OR logico esclusivo





A	B	A EX-OR B
0	0	0
0	1	1
1	0	1
1	1	0

Come esempio pratico, poniamo **BX=1000111010100110b**, **CX=0110100111110010b**, ed eseguiamo la seguente istruzione:

```
xor bx, cx
```

In presenza di questa istruzione, la **CPU** ottiene il seguente risultato:

```
            BX = 1000111010100110b
            CX = 0110100111110010b
 --------------------------------
BX = BX XOR CX = 1110011101010100b
```

Per pervenire a questo risultato, la **CPU** esegue le seguenti elaborazioni (a partire dai bit meno significativi dei due operandi):

`0 XOR 0 = 0` (**XOR** tra i bit in posizione **0**)

`1 XOR 1 = 0` (**XOR** tra i bit in posizione **1**)

`1 XOR 0 = 1` (**XOR** tra i bit in posizione **2**)

e così via.

19.5.1 OR logico esclusivo tra operandi di ampiezza arbitraria

L'istruzione **XOR** opera bit per bit, per cui può essere applicata con estrema semplicità, anche ad operandi di ampiezza arbitraria; in sostanza, non dobbiamo fare altro che suddividere ciascun operando, in tante parti direttamente gestibili dalla **CPU**.
Se, ad esempio, abbiamo a che fare con operandi a **96** bit, possiamo suddividere ciascuno di essi in **3 DWORD**; a questo punto, dobbiamo effettuare uno **XOR** tra ciascuna coppia di **DWORD** corrispondenti.

Consideriamo le seguenti definizioni:

```
Num1     dd    2B96CDEAh, 6F9C482Dh, 881E63CAh
Num2     dd    66D41695h, 3C9DF1AFh, 7BB3F28Dh
Num3     dd    3 dup (0)
```

Se vogliamo ottenere:

`Num3 = Num1 XOR Num2`

non dobbiamo fare altro che tradurre in **Assembly** le seguenti pseudo-istruzioni:

`Num3[0] = Num1[0] XOR Num2[0] = 2B96CDEAh XOR 66D41695h = 4D42DB7Fh`

Num3[4] = Num1[4] XOR Num2[4] = 6F9C482Dh XOR 3C9DF1AFh = 5301B982h

Num3[8] = Num1[8] XOR Num2[8] = 881E63CAh XOR 7BB3F28Dh = F3AD9147h

19.5.2 Effetti provocati da XOR sugli operandi e sui flags

L'esecuzione dell'istruzione **XOR** tra **SRC** e **DEST**, modifica il contenuto del solo operando **DEST** che viene sovrascritto dal risultato prodotto dall'istruzione stessa; il contenuto dell'operando **SRC** rimane inalterato.
Bisogna però prestare particolare attenzione ad istruzioni del tipo:

```
xor di, ss:[di+05h]
```

Dopo l'esecuzione di questa istruzione, la componente **Offset** contenuta in **DI**, viene sovrascritta dal risultato prodotto da **XOR**!

La possibilità di effettuare uno **XOR** tra **Reg** e **Reg**, ci permette di scrivere anche istruzioni del tipo:

```
xor dx, dx
```

Come si può facilmente constatare, l'esecuzione di una tale istruzione provoca l'azzeramento del contenuto di **DX**; infatti:

0 XOR 0 = 0 e 1 XOR 1 = 0

Poniamo, ad esempio, **DX=1001110010000011b**, ed eseguiamo la seguente istruzione:

```
xor dx, dx
```

In presenza di questa istruzione, la **CPU** ottiene il seguente risultato:

```
           DX = 1001110010000011b
           DX = 1001110010000011b
 ---------------------------------
DX = DX XOR DX = 0000000000000000b
```

Il ricorso a **XOR** per l'azzeramento di un registro, è una pratica molto diffusa tra i programmatori **Assembly**; si tenga presente, comunque, che non si ottiene nessun guadagno in velocità rispetto alle analoghe istruzioni:

```
mov dx, 0
```

e:

```
sub dx, dx
```

L'esecuzione dell'istruzione **XOR**, modifica i campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; il solo campo **AF** assume un valore indeterminato, e non ha quindi nessun significato.
La **CPU** pone, in ogni caso, **CF=0** e **OF=0**, mentre i campi **PF**, **ZF** e **SF** forniscono informazioni ordinarie relative al risultato prodotto da **XOR**; in sostanza, **PF** indica se i primi **8** bit del risultato contengono un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il risultato vale zero, **SF** indica se il bit più significativo del risultato vale **1**.

19.6 L'istruzione SHL

Con il mnemonico **SHL** si indica l'istruzione **SHift logical Left** (scorrimento logico verso sinistra); lo scopo di questa istruzione è quello di effettuare uno scorrimento verso sinistra, dei bit dell'operando **DEST**, il cui contenuto viene trattato come numero intero senza segno. Il numero di scorrimenti da effettuare, viene indicato dall'operando **SRC**; i posti rimasti liberi a destra nell'operando **DEST**, vengono riempiti con degli zeri.
Per ogni singolo scorrimento, il bit più significativo dell'operando **DEST** trabocca da sinistra; il valore di tale bit, viene salvato nel flag **CF**.
Entrambi gli operandi devono essere specificati, esplicitamente, dal programmatore; il risultato prodotto da **SHL**, viene memorizzato nell'operando **DEST**.
Le uniche forme lecite per l'istruzione **SHL** sono le seguenti:

```
SHL   Reg, 1
SHL   Reg, CL
SHL   Reg, Imm8
SHL   Mem, 1
SHL   Mem, CL
SHL   Mem, Imm8
```

Il codice macchina generale per l'istruzione **SHL**, è formato dal campo **Opcode 110100vw** e dal campo **mod_100_r/m**; come si può notare, nel campo **Opcode** è presente un bit indicato con **v**. Se **v=0**, allora l'operando **SRC** è rappresentato dal valore immediato **1** (un solo scorrimento); se **v=1**, allora l'operando **SRC** è rappresentato dall'half register **CL** (che contiene il numero di scorrimenti da effettuare). Con le **CPU 80286** e superiori, il numero di scorrimenti può essere indicato anche attraverso un **Imm8**; in tal caso, il codice macchina è formato dal campo **Opcode 1100000w** e dal campo **mod_100_r/m**.

Come esempio pratico poniamo **AX=0111001011011100b**, **CL=5**, ed eseguiamo la seguente istruzione:

```
shl ax, cl
```

In presenza di questa istruzione, la **CPU** fa scorrere di **5** posti verso sinistra, tutti i bit di **AX**; i **5** posti che si liberano a destra, vengono riempiti con degli zeri. La successione dei singoli scorrimenti produce quindi i seguenti risultati:

```
AX = 1110010110111000b, CF = 0

AX = 1100101101110000b, CF = 1

AX = 1001011011100000b, CF = 1

AX = 0010110111000000b, CF = 1

AX = 0101101110000000b, CF = 0
```

I singoli bit che traboccano da sinistra, vengono salvati nel flag **CF**; nel nostro caso, dopo **5** scorrimenti, l'ultimo bit traboccato da sinistra è uno **0**, per cui alla fine otteniamo **AX=0101101110000000b** e **CF=0**.

Con le **CPU 8086**, il registro **CL** può contenere un qualsiasi valore compreso tra **0**

e **255**; si tratta chiaramente di una situazione priva di senso, che in certi casi può comportare, per la **CPU**, tempi di elaborazione elevatissimi. Per rendercene conto, poniamo, ad esempio, **AX=0111010100101111b**, **CL=255**, ed eseguiamo la seguente istruzione:

```
shl ax, cl
```

Come si può facilmente constatare, dopo i primi **16** scorrimenti, tutti i **16** bit di **AX** sono traboccati da sinistra e sono stati sostituiti con **16** zeri; in sostanza, dopo i primi **16** scorrimenti si ottiene **AX=0**, per cui è perfettamente inutile continuare con gli altri **239** scorrimenti.
L'aspetto più grave è legato al fatto che le **CPU 8086**, implementano istruzioni logiche scarsamente ottimizzate; tali istruzioni, vengono quindi eseguite con una certa lentezza. Come si nota dalle tabelle, l'istruzione **SHL** con contatore **CL** viene eseguita dall'**8086** in **8** cicli di clock, ai quali bisogna aggiungere altri **4** cicli di clock per ogni scorrimento da **1** bit; se **CL** contiene il valore **255**, l'**8086** esegue l'istruzione in ben:

```
8 + (4 x 255) = 1028 cicli di clock
```

La frequenza di clock dell'**8086** è di circa **5** milioni di cicli al secondo (**5 MHz**), per cui l'esecuzione dell'istruzione richiede un tempo abissale di:

```
1028 / 5000000 = 0.0002056 secondi = 0.2056 millisecondi
```

Questo problema è stato eliminato a partire dalle **CPU 80286**; tali **CPU**, infatti, sottopongono il contenuto del contatore (**CL** o **Imm8**), ad un **AND** con il valore **00011111b**. Come sappiamo, una istruzione del tipo:

```
and cl, 00011111b
```

azzera i tre bit più significativi di **CL**, e lascia inalterati i restanti **5** bit. In sostanza, le **CPU 80286** e superiori, prendono in considerazione solamente i primi **5** bit del contatore; con **5** bit possiamo rappresentare un qualunque valore compreso tra **0** e **31**, più che sufficiente per operare con registri a **8**, **16** o **32** bit.
A tutto ciò bisogna aggiungere che le **CPU 80286** e superiori, implementano istruzioni logiche altamente ottimizzate; quando si utilizza **CL** o un **Imm8** come contatore, la **CPU 80286** richiede un solo ciclo di clock aggiuntivo per ogni bit di scorrimento. Nel caso delle **CPU 80386** e superiori, il numero di cicli di clock diventa addirittura indipendente dal numero di scorrimenti da effettuare; come si nota dalle tabelle, una **CPU 80486** a **33** MHz, esegue l'istruzione **SHL** in appena **2** o **3** cicli di clock, indipendentemente dal numero di scorrimenti specificati da **CL** o da un **Imm8**.

Nota importante.
Le considerazioni appena esposte sulla gestione del contatore **CL** o **Imm8** da parte delle **CPU 80286** e superiori, si applicano anche alle altre istruzioni di scorrimento e di rotazione dei bit, illustrate nel seguito del capitolo; tali istruzioni sono rappresentate dai mnemonici **SAL**, **SHR**, **SAR**, **ROL**, **RCR**, **ROR**, **RCR**.

19.6.1 Significato matematico dell'istruzione SHL

Come abbiamo visto nel capitolo dedicato alla matematica del computer, moltiplicare un numero intero senza segno **m**, espresso in base **b**, per un fattore b^n (**n** intero positivo), equivale ad aggiungere **n** zeri alla destra dello stesso

m; in questo modo, tutte le cifre di **m** vengono fatte scorrere di **n** posti verso sinistra.
Possiamo dire quindi che una istruzione del tipo:

```
shl ax, n
```

equivale a moltiplicare il contenuto binario di **AX** per 2^n.

Come verifica poniamo, **AX=0000101011110010b=2802**, ed eseguiamo l'istruzione:

```
shl ax, 4
```

In presenza di questa istruzione, la **CPU** sottopone il contenuto di **AX** ai seguenti **4** singoli scorrimenti:

```
AX = 0001010111100100b = 5604 = 2802 * 2, CF = 0

AX = 0010101111001000b = 11208 = 2802 * 4, CF = 0

AX = 0101011110010000b = 22416 = 2802 * 8, CF = 0

AX = 1010111100100000b = 44832 = 2802 * 16, CF = 0
```

Come si può notare, dopo **4** scorrimenti verso sinistra, il contenuto iniziale **2802** di **AX** risulta moltiplicato per 2^4**=16**!

Una **CPU 80486 DX** esegue la precedente istruzione in appena **2** cicli di clock; per ottenere lo stesso risultato, possiamo porre **AX=0000101011110010b=2802**, **BX=16**, ed eseguire l'istruzione:

```
mul bx
```

La stessa **CPU 80486 DX** esegue questa istruzione in **13** cicli di clock, cioè, oltre **6** volte più lentamente rispetto all'uso di **SHL**!
Proprio per questo motivo, ogni volta che si deve eseguire una moltiplicazione tra numeri interi senza segno, con uno dei fattori esprimibile nella forma 2^n (**n** intero positivo), è vivamente consigliabile l'utilizzo di **SHL** con contatore **n**.
Il programmatore che intende utilizzare **SHL** al posto di **MUL**, deve prestare però particolare attenzione al fatto che, i registri della **CPU** (e le locazioni di memoria), hanno una ampiezza in bit limitata; un numero eccessivo di scorrimenti verso sinistra, può portare quindi al trabocco di cifre significative dell'operando **DEST**, con un conseguente risultato privo di senso!
Per verificare questo aspetto, riprendiamo il precedente esempio nel quale, dopo **4** scorrimenti verso sinistra, avevamo ottenuto:

```
AX = 1010111100100000b = 44832 = 2802 * 16, CF = 0
```

Sottoponendo ora **AX** ad un ulteriore scorrimento di **1** bit verso sinistra, otteniamo:

```
AX = 0101111001000000b = 24128, CF = 1
```

Questo risultato è sbagliato in quanto si è verificato il trabocco da sinistra, di un bit di valore **1**; questa situazione viene segnalata da **CF=1**. In sostanza, il risultato corretto è rappresentato dal contenuto di **AX** e da un ulteriore bit (quello che è traboccato da sinistra), per un totale di **17** bit; infatti

aggiungendo **1** alla sinistra del precedente risultato, si ottiene:

10101111001000000b = 89664 = 2802 * 2^5

In definitiva, una (pseudo) istruzione del tipo:

mul DEST, 2^n

è sostituibile con una (pseudo) istruzione del tipo:

shl DEST, n

solo se siamo sicuri che il risultato finale è interamente rappresentabile con l'ampiezza in bit di **DEST**.

19.6.2 Shift logical left su operandi di ampiezza arbitraria

Supponiamo di voler far scorrere verso sinistra, i bit di un numero intero senza segno di ampiezza arbitraria; in un caso del genere, la strada migliore da seguire consiste nel definire un procedimento generale, indipendente dall'ampiezza in bit dell'operando, e dal numero di scorrimenti che vogliamo effettuare.
Un tale procedimento ha bisogno di conoscere, l'indirizzo iniziale del dato da sottoporre a shifting, l'ampiezza in byte del dato stesso, e il numero di scorrimenti da effettuare; nel caso di una richiesta di scorrimento di **n** posti, possiamo pensare di ripetere per **n** volte, un procedimento che effettua un solo scorrimento alla volta.

Il punto fondamentale consiste allora nel trovare un metodo per effettuare uno scorrimento unitario verso sinistra; analizziamo una soluzione molto elementare, che ci permette anche di illustrare la potenza delle istruzioni logiche. Quando avremo a disposizione le procedure e le istruzioni per i loop, saremo in grado di ottimizzare tutti gli algoritmi presentati nel seguito del capitolo e nei capitoli precedenti.

Vogliamo far scorrere di un posto verso sinistra, tutti i bit del numero:

Num1 = 0011110011110010b

Poniamo **AH=0**, e carichiamo in **AL** il byte meno significativo di **Num1** (**AL=11110010b**); in seguito ad una istruzione:

shl ax, 1

otteniamo:

AX = 0000000111100100b

In sostanza, il bit più significativo di **AL** trabocca da sinistra, e finisce nel bit meno significativo di **AH**; si ha quindi **AH=00000001b** e **AL=11100100b**.
Salviamo il contenuto di **AL** in **Num1[0]**, e il contenuto di **AH**, ad esempio, in **BL**; ripetiamo ora il procedimento sul secondo byte di **Num1**. Poniamo quindi **AH=0** e **AL=00111100b**; in seguito ad una istruzione:

shl ax, 1

otteniamo:

AX = 0000000001111000b

In sostanza, il bit più significativo di **AL** trabocca da sinistra, e finisce nel bit meno significativo di **AH**; si ha quindi **AH=00000000b** e **AL=01111000b**.
Il bit che avevamo salvato in **BL** deve essere ora posizionato nel bit meno significativo di **AL**; a tale proposito, ci basta eseguire la seguente istruzione:

AL = AL OR BL = 01111000b OR 00000001b = 01111001b

Salviamo il contenuto di **AL** in **Num1[1]**; a questo punto possiamo notare che:

Num1 = 0111100111100100b

Il registro **AH** contiene il bit più significativo di **Num1**, traboccato da sinistra (nel nostro caso, **AH=00000000b**); appare anche evidente il fatto che, il procedimento appena descritto, può essere applicato a numeri interi senza segno di qualunque ampiezza.

Come esempio pratico, consideriamo la seguente definizione:

```
bigNum db 10010010b, 00111100b, 00011111b ; 000111110011111010010010b
```

Il codice che esegue lo scorrimento unitario verso sinistra, assume il seguente aspetto:

```
; shifting unitario del BYTE n. 0 di bigNum

xor     ah, ah              ; ah = 0
mov     al, BigNum[0]       ; al = 10010010b
shl     ax, 1               ; ah = 00000001b, al = 00100100b
mov     BigNum[0], al       ; BigNum[0] = 00100100b
mov     bl, ah              ; bl = ah = 00000001b

; shifting unitario del BYTE n. 1 di bigNum

xor     ah, ah              ; ah = 0
mov     al, BigNum[1]       ; al = 00111100b
shl     ax, 1               ; ah = 00000000b, al = 01111000b
or      al, bl              ; al = 01111001b
mov     BigNum[1], al       ; BigNum[1] = 01111001b
mov     bl, ah              ; bl = ah = 00000000b

; shifting unitario del BYTE n. 2 di bigNum

xor     ah, ah              ; ah = 0
mov     al, BigNum[2]       ; al = 00011111b
shl     ax, 1               ; ah = 00000000b, al = 00111110b
or      al, bl              ; al = 00111110b
mov     BigNum[2], al       ; BigNum[2] = 00111110b
```

Unendo i **3** risultati precedenti, otteniamo:

BigNum = 001111100111100100100100b

nel bit meno significativo di **AH**; naturalmente, è fondamentale ricordare che la locazione di memoria riservata a **BigNum**, deve avere spazio sufficiente per contenere il risultato dello shifting.

Se vogliamo visualizzare il contenuto binario di **BigNum**, possiamo procedere nel solito modo con l'ausilio di **writeBin8**; in questo caso, l'output deve iniziare da **BigNum[2]**.

19.6.3 Effetti provocati da SHL sugli operandi e sui flags

L'esecuzione dell'istruzione **SHL**, modifica il contenuto del solo operando **DEST** che viene sovrascritto dal risultato prodotto dall'istruzione stessa; il contenuto dell'operando **SRC** rimane inalterato.
Bisogna però prestare particolare attenzione ad un caso delicato, rappresentato da istruzioni del tipo:

```
shl cx, cl
```

(ovviamente, sarebbe opportuno evitare l'inserimento nei propri programmi, di questo genere di istruzioni).
Per capire bene il funzionamento della precedente istruzione, dobbiamo ricordare che le **CPU 80286** e superiori, effettuano un **AND** tra il contenuto di **CL** e la maschera di bit **00011111b**; vediamo allora quello che succede ponendo **CX=0011101011100110b** ed eseguendo l'istruzione:

```
shl cx, cl
```

Prima di tutto, la **CPU** calcola:

```
CL = CL AND 00011111b = 11100110b AND 00011111b = 00000110b = 6
```

Di conseguenza, il contenuto di **CX** viene alterato e diventa **0011101000000110b**; i bit di **CX** vengono fatti scorrere di **6** posti verso sinistra, con il risultato finale **CX=1000000110000000b** (**CF=0**).

L'esecuzione dell'istruzione **SHL** con **CL=0** o **Imm8=0**, non modifica nessun campo del **Flags Register**; ovviamente, in un caso del genere, il contenuto di **DEST** rimane inalterato.
L'esecuzione dell'istruzione **SHL** con contatore non nullo, modifica i campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; il solo campo **AF** assume un valore indeterminato, e non ha quindi nessun significato.
Il campo **OF** assume un valore sensato solo quando il contatore vale **1** (e quindi anche con **CL=1** o **Imm8=1**); se il contatore è maggiore di **1**, il flag **OF** assume un valore indeterminato, e non ha quindi nessun significato. Come si può facilmente immaginare, la **CPU** pone **OF=1** quando l'istruzione **SHL** provoca la modifica del bit più significativo (bit di segno) di **DEST**; in caso contrario, la **CPU** pone **OF=0**.
Il campo **CF** (nel caso in cui il contatore sia diverso da zero), contiene l'ultimo bit traboccato dalla sinistra di **DEST**.
I flags **PF**, **ZF** e **SF** (nel caso in cui il contatore sia diverso da zero) forniscono informazioni ordinarie relative al risultato prodotto da **SHL**; in sostanza, **PF** indica se i primi **8** bit del risultato contengono un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il risultato vale zero, **SF** indica se il bit più significativo del risultato vale **1**.

19.7 L'istruzione SAL

Con il mnemonico **SAL** si indica l'istruzione **Shift Arithmetic Left** (scorrimento aritmetico verso sinistra); lo scopo di questa istruzione è quello di effettuare uno scorrimento verso sinistra, dei bit dell'operando **DEST**, il cui contenuto viene trattato come numero intero con segno in complemento a **2**. Il numero di scorrimenti da effettuare, viene indicato dall'operando **SRC**; i posti rimasti liberi a destra nell'operando **DEST**, vengono riempiti con degli zeri.
Per ogni singolo scorrimento, il bit più significativo dell'operando **DEST** trabocca da sinistra; il valore di tale bit, viene salvato nel flag **CF**.
Entrambi gli operandi devono essere specificati, esplicitamente, dal programmatore; il risultato prodotto da **SAL**, viene memorizzato nell'operando **DEST**.
Le uniche forme lecite per l'istruzione **SAL** sono le seguenti:

```
SAL   Reg, 1
SAL   Reg, CL
SAL   Reg, Imm8
SAL   Mem, 1
SAL   Mem, CL
SAL   Mem, Imm8
```

A differenza di **SHL** che tratta il contenuto di **DEST** come un numero intero senza segno, l'istruzione **SAL** effettua il proprio lavoro trattando il contenuto di **DEST** come un numero intero con segno in complemento a **2**; siccome però il bit di segno di **DEST** è quello più a sinistra, e gli scorrimenti si svolgono, appunto, verso sinistra, gli effetti prodotti da **SAL** sono assolutamente identici a quelli prodotti da **SHL**!
Le due istruzioni **SHL** e **SAL** sono del tutto equivalenti, e risultano quindi interscambiabili; proprio per questo motivo, il codice macchina di **SAL** è identico a quello di **SHL**. L'istruzione **SAL** viene resa disponibile solo per motivi di simmetria; al ruolo svolto dalla coppia **SHL**, **SAL**, si contrappone, infatti, quello della coppia **SHR**, **SAR**, illustrata più avanti.

19.7.1 Significato matematico dell'istruzione SAL

Il significato matematico dell'istruzione **SAL** è del tutto simile a quello già illustrato per l'istruzione **SHL**; più precisamente, possiamo dire che una (pseudo) istruzione del tipo:

```
sal DEST, n
```

è equivalente ad una (pseudo) istruzione del tipo:

```
imul DEST, 2^n
```

Non dobbiamo dimenticare però che questa volta, l'operando **DEST** rappresenta un numero intero con segno in complemento a **2**; come si può facilmente intuire, la situazione appare più delicata, a causa del fatto che il bit di segno dell'operando **DEST** è quello più a sinistra.

Per analizzare questo aspetto poniamo, **AX=1101110011110001b=-8975**, ed eseguiamo l'istruzione:

```
sal ax, 1
```

Dopo l'esecuzione di questa istruzione, otteniamo:

AX = 1011100111100010b = -17950 = (-8975) * 2^1, CF = 1, OF = 0

Il risultato è esatto in quanto **SAL** non ha provocato la perdita del bit di segno di **AX**; infatti, il bit più significativo di **AX** continua a valere **1**. La **CPU** segnala questa situazione ponendo **OF=0**, in quanto, moltiplicando un numero negativo per un numero positivo, abbiamo ottenuto, correttamente, un numero negativo.

Vediamo però quello che succede con una ulteriore istruzione:

```
sal ax, 1
```

Dopo l'esecuzione di questa istruzione, otteniamo:

AX = 0111001111000100b = +29636, CF = 1, OF = 1

Il risultato è sbagliato in quanto **SAL** ha provocato la perdita del bit di segno di **AX**; infatti, il bit più significativo di **AX** è passato da **1** a **0**. La **CPU** segnala questa situazione ponendo **OF=1**, in quanto, moltiplicando un numero negativo per un numero positivo, abbiamo ottenuto, erroneamente, un numero positivo.

In definitiva, una (pseudo) istruzione del tipo:

```
imul DEST, 2^n
```

è sostituibile con una (pseudo) istruzione del tipo:

```
sal DEST, n
```

solo se siamo sicuri che il risultato finale è interamente rappresentabile con l'ampiezza in bit di **DEST**.

19.7.2 Shift arithmetic left su operandi di ampiezza arbitraria

Come è stato già sottolineato, gli effetti prodotti da **SAL** sull'operando **DEST**, sono assolutamente identici a quelli prodotti da **SHL**; di conseguenza, se vogliamo applicare **SAL** ad operandi di ampiezza arbitraria, dobbiamo procedere esattamente come nell'esempio mostrato per **SHL**.

19.7.3 Effetti provocati da SAL sugli operandi e sui flags

Anche in relazione agli effetti provocati da **SAL** sugli operandi e sui flags, valgono tutte le considerazioni già esposte per **SHL**.

19.8 L'istruzione SHR

Con il mnemonico **SHR** si indica l'istruzione **SHift logical Right** (scorrimento logico verso destra); lo scopo di questa istruzione è quello di effettuare uno scorrimento verso destra, dei bit dell'operando **DEST**, il cui contenuto viene trattato come numero intero senza segno. Il numero di scorrimenti da effettuare, viene indicato dall'operando **SRC**; i posti rimasti liberi a sinistra nell'operando **DEST**, vengono riempiti con degli zeri.
Per ogni singolo scorrimento, il bit meno significativo dell'operando **DEST** trabocca da destra; il valore di tale bit, viene salvato nel flag **CF**.

Entrambi gli operandi devono essere specificati, esplicitamente, dal programmatore; il risultato prodotto da **SHR**, viene memorizzato nell'operando **DEST**.
Le uniche forme lecite per l'istruzione **SHR** sono le seguenti:

```
SHR   Reg, 1
SHR   Reg, CL
SHR   Reg, Imm8
SHR   Mem, 1
SHR   Mem, CL
SHR   Mem, Imm8
```

Il codice macchina generale per l'istruzione **SHR**, è formato dal campo **Opcode 110100vw** e dal campo **mod_101_r/m**; se **v=0**, allora l'operando **SRC** è rappresentato dal valore immediato **1** (un solo scorrimento), mentre se **v=1**, allora l'operando **SRC** è rappresentato dall'half register **CL** (che contiene il numero di scorrimenti da effettuare). Con le **CPU 80286** e superiori, il numero di scorrimenti può essere indicato anche attraverso un **Imm8**; in tal caso, il codice macchina è formato dal campo **Opcode 1100000w** e dal campo **mod_101_r/m**.

Come esempio pratico poniamo **AX=0111001011011100b**, **CL=5**, ed eseguiamo la seguente istruzione:

```
shr ax, cl
```

In presenza di questa istruzione, la **CPU** fa scorrere di **5** posti verso destra, tutti i bit di **AX**; i **5** posti che si liberano a sinistra, vengono riempiti con degli zeri. La successione dei singoli scorrimenti produce quindi i seguenti risultati:

```
AX = 0011100101101110b, CF = 0

AX = 0001110010110111b, CF = 0

AX = 0000111001011011b, CF = 1

AX = 0000011100101101b, CF = 1

AX = 0000001110010110b, CF = 1
```

I singoli bit che traboccano da destra, vengono salvati nel flag **CF**; nel nostro caso, dopo **5** scorrimenti, l'ultimo bit traboccato da destra è un **1**, per cui alla fine otteniamo **AX=0000001110010110b** e **CF=1**.

19.8.1 Significato matematico dell'istruzione SHR

Come abbiamo visto nel capitolo dedicato alla matematica del computer, dividere un numero intero senza segno **m**, espresso in base **b**, per un divisore $\mathbf{b^n}$ (**n** intero positivo), equivale a troncare **n** cifre dalla destra dello stesso **m**; in questo modo, tutte le cifre di **m** vengono fatte scorrere di **n** posti verso destra. Risulta, inoltre, che le **n** cifre troncate dalla destra di **m**, rappresentano il resto della divisione intera.
Possiamo dire quindi che una istruzione del tipo:

```
shr ax, n
```

equivale a dividere il contenuto binario di **AX** per 2^n.

Come verifica poniamo, **AX=0110101011110010b=27378**, ed eseguiamo l'istruzione:

```
shr ax, 4
```

In presenza di questa istruzione, la **CPU** sottopone il contenuto di **AX** ai seguenti **4** singoli scorrimenti:

```
AX = 0011010101111001b = 13689 = 27378 / 2 (R = 0b), CF = 0

AX = 0001101010111100b = 6844 = 27378 / 4 (R = 10b), CF = 1

AX = 0000110101011110b = 3422 = 27378 / 8 (R = 010b), CF = 0

AX = 0000011010101111b = 1711 = 27378 / 16 (R = 0010b), CF = 0
```

Come si può notare, dopo **4** scorrimenti verso destra, il contenuto iniziale **27378** di **AX** risulta diviso per 2^4**=16**; inoltre, i **4** bit traboccati da destra, e cioè, **0010b=2**, rappresentano il resto della divisione intera!

Una **CPU 80486 DX** esegue la precedente istruzione in appena **2** cicli di clock; per ottenere lo stesso risultato, possiamo porre **DX=0**, **AX=0110101011110010b=27378**, **BX=16**, ed eseguire l'istruzione:

```
div bx
```

La stessa **CPU 80486 DX** esegue questa istruzione in **24** cicli di clock, cioè, **12** volte più lentamente rispetto all'uso di **SHR**!
Proprio per questo motivo, ogni volta che si deve eseguire una divisione tra numeri interi senza segno, con il divisore esprimibile nella forma 2^n (**n** intero positivo), è vivamente consigliabile l'utilizzo di **SHR** con contatore **n**.
L'istruzione **SHR** fornisce sempre un quoziente e un resto esatti, indipendentemente dal numero di scorrimenti verso destra; ovviamente, ad un certo punto il dividendo diventa **0**, per cui ogni ulteriore scorrimento di **1** bit verso destra, ci fornirà un quoziente **0** e un resto **0**.

In definitiva, una (pseudo) istruzione del tipo:

```
div DEST, 2^n
```

è sempre sostituibile con una (pseudo) istruzione del tipo:

```
shr DEST, n
```

19.8.2 Shift logical right su operandi di ampiezza arbitraria

Supponiamo di voler far scorrere verso destra, i bit di un numero intero senza segno di ampiezza arbitraria; in tal caso, possiamo facilmente adattare il procedimento già illustrato per l'istruzione **SHL**.

Vogliamo far scorrere di un posto verso destra, tutti i bit del numero:

```
Num1 = 1011110111110011b
```

(**AH=10111101b**); in seguito ad una istruzione:

```
shr ax, 1
```

otteniamo:

```
AX = 0101111010000000b
```

In sostanza, il bit meno significativo di **AH** trabocca da destra, e finisce nel bit più significativo di **AL**; si ha quindi **AH=01011110b** e **AL=10000000b**. Salviamo il contenuto di **AH** in **Num1[1]**, e il contenuto di **AL**, ad esempio, in **BL**; ripetiamo ora il procedimento sul secondo byte (da sinistra) di **Num1**. Poniamo quindi **AL=0** e **AH=11110011b**; in seguito ad una istruzione:

```
shr ax, 1
```

otteniamo:

```
AX = 0111100110000000b
```

In sostanza, il bit meno significativo di **AH** trabocca da destra, e finisce nel bit più significativo di **AL**; si ha quindi **AH=01111001b** e **AL=10000000b**. Il bit che avevamo salvato in **BL** deve essere ora posizionato nel bit più significativo di **AH**; a tale proposito, ci basta eseguire l'istruzione:

```
AH = AH OR BL = 01111001b OR 10000000b = 11111001b
```

Salviamo il contenuto di **AH** in **Num1[0]**; a questo punto possiamo notare che:

```
Num1 = 0101111011111001b
```

Il registro **AL** contiene il bit meno significativo di **Num1**, traboccato da destra (nel nostro caso, **AL=10000000b**); appare anche evidente il fatto che, il procedimento appena descritto, può essere applicato a numeri interi senza segno di qualunque ampiezza.

Come esempio pratico, consideriamo la seguente definizione:

```
bigNum db 10010010b, 00111100b, 00011111b ; 000111110011110010010010b
```

Il codice che esegue lo scorrimento unitario verso destra, assume il seguente aspetto:

```
; shifting unitario del BYTE n. 2 di bigNum

xor     al, al              ; al = 0
mov     ah, BigNum[2]       ; ah = 00011111b
shr     ax, 1               ; ah = 00001111b, al = 10000000b
mov     BigNum[2], ah       ; BigNum[2] = 00001111b
mov     bl, al              ; bl = al = 10000000b

; shifting unitario del BYTE n. 1 di bigNum

xor     al, al              ; al = 0
mov     ah, BigNum[1]       ; ah = 00111100b
shr     ax, 1               ; ah = 00011110b, al = 00000000b
```

```
mov     BigNum[1], ah       ; BigNum[1] = 10011110b
mov     bl, al              ; bl = al = 00000000b

; shifting unitario del BYTE n. 0 di bigNum

xor     al, al              ; al = 0
mov     ah, BigNum[0]       ; ah = 10010010b
shr     ax, 1               ; ah = 01001001b, al = 00000000b
or      ah, bl              ; ah = 01001001b
mov     BigNum[0], ah       ; BigNum[0] = 01001001b
```

Unendo i **3** risultati precedenti, otteniamo:

BigNum = 000011111001111001001001b

Il bit meno significativo di **BigNum**, traboccato da destra, si trova memorizzato nel bit più significativo di **AL**.

Se vogliamo visualizzare il contenuto binario di **BigNum**, possiamo procedere nel solito modo con l'ausilio di **writeBin8**; in questo caso, l'output deve iniziare da **BigNum[2]**.

19.8.3 Effetti provocati da SHR sugli operandi e sui flags

L'esecuzione dell'istruzione **SHR**, modifica il contenuto del solo operando **DEST** che viene sovrascritto dal risultato prodotto dall'istruzione stessa; il contenuto dell'operando **SRC** rimane inalterato.
Bisogna però prestare particolare attenzione al caso che già conosciamo, rappresentato da istruzioni del tipo:

```
shr cx, cl
```

L'esecuzione dell'istruzione **SHR** con **CL=0** o **Imm8=0**, non modifica nessun campo del **Flags Register**; ovviamente, in un caso del genere, il contenuto di **DEST** rimane inalterato.
L'esecuzione dell'istruzione **SHR** con contatore non nullo, modifica i campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; il solo campo **AF** assume un valore indeterminato, e non ha quindi nessun significato.
Il campo **OF** assume un valore sensato solo quando il contatore vale **1** (e quindi anche con **CL=1** o **Imm8=1**); se il contatore è maggiore di **1**, il flag **OF** assume un valore indeterminato, e non ha quindi nessun significato. Come si può facilmente immaginare, la **CPU** pone **OF=1** quando l'istruzione **SHR** provoca la modifica del bit più significativo (bit di segno) di **DEST**; in caso contrario, la **CPU** pone **OF=0**.
Il campo **CF** (nel caso in cui il contatore sia diverso da zero), contiene l'ultimo bit traboccato dalla destra di **DEST**.
I flags **PF**, **ZF** e **SF** (nel caso in cui il contatore sia diverso da zero) forniscono informazioni ordinarie relative al risultato prodotto da **SHR**; in sostanza, **PF** indica se i primi **8** bit del risultato contengono un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il risultato vale zero, **SF** indica se il bit più significativo del risultato vale **1**.

19.9 L'istruzione SAR

Con il mnemonico **SAR** si indica l'istruzione **Shift Arithmetic Right** (scorrimento aritmetico verso destra); lo scopo di questa istruzione è quello di effettuare

trattato come numero intero con segno in complemento a **2**. Il numero di scorrimenti da effettuare, viene indicato dall'operando **SRC**; i posti rimasti liberi a sinistra nell'operando **DEST**, vengono riempiti con il bit di segno dello stesso **DEST**.
Per ogni singolo scorrimento, il bit meno significativo dell'operando **DEST** trabocca da destra; il valore di tale bit, viene salvato nel flag **CF**.
Entrambi gli operandi devono essere specificati, esplicitamente, dal programmatore; il risultato prodotto da **SAR**, viene memorizzato nell'operando **DEST**.
Le uniche forme lecite per l'istruzione **SAR** sono le seguenti:

```
SAR   Reg, 1
SAR   Reg, CL
SAR   Reg, Imm8
SAR   Mem, 1
SAR   Mem, CL
SAR   Mem, Imm8
```

Il codice macchina generale per l'istruzione **SAR**, è formato dal campo **Opcode 110100vw** e dal campo **mod_111_r/m**; se **v=0**, allora l'operando **SRC** è rappresentato dal valore immediato **1** (un solo scorrimento), mentre se **v=1**, allora l'operando **SRC** è rappresentato dall'half register **CL** (che contiene il numero di scorrimenti da effettuare). Con le **CPU 80286** e superiori, il numero di scorrimenti può essere indicato anche attraverso un **Imm8**; in tal caso, il codice macchina è formato dal campo **Opcode 1100000w** e dal campo **mod_111_r/m**.

Come esempio pratico poniamo **AX=1100011100101011b**, **CL=5**, ed eseguiamo la seguente istruzione:

```
sar ax, cl
```

In presenza di questa istruzione, la **CPU** fa scorrere di **5** posti verso destra, tutti i bit di **AX**; i **5** posti che si liberano a sinistra, vengono riempiti con il bit di segno (**1**) dello stesso **AX**. La successione dei singoli scorrimenti produce quindi i seguenti risultati:

```
AX = 1110001110010101b, CF = 1

AX = 1111000111001010b, CF = 1

AX = 1111100011100101b, CF = 0

AX = 1111110001110010b, CF = 1

AX = 1111111000111001b, CF = 0
```

I singoli bit che traboccano da destra, vengono salvati nel flag **CF**; nel nostro caso, dopo **5** scorrimenti, l'ultimo bit traboccato da destra è uno **0**, per cui alla fine otteniamo **AX=1111111000111001b** e **CF=0**.

19.9.1 Significato matematico dell'istruzione SAR

Come è stato dettagliatamente spiegato nel Capitolo 6, **solamente in determinati casi**, dividere un numero intero con segno **m**, espresso in base **b**, per un divisore b^n (**n** intero positivo), equivale a troncare **n** cifre dalla destra dello stesso **m**; tale equivalenza sussiste solo se **m** è un numero intero positivo qualsiasi.

oppure se **m** è un numero intero negativo, multiplo intero del divisore!
Per verificare in pratica questo aspetto, supponiamo di voler lavorare con i numeri interi con segno a **16** bit in complemento a **2**; tali numeri sono quindi compresi tra **-32768** e **+32767**.

Poniamo **AX=0110011001110011b=+26227** (numero intero positivo), ed eseguiamo l'istruzione:

```
sar ax, 4
```

Il risultato prodotto da questa istruzione è:

```
AX = 0000011001100111b = +1639
```

Inoltre, i **4** bit traboccati da destra, rappresentano il valore **0011b=+3**.
Come si può notare, i due valori **+1639** e **+3** rappresentano, rispettivamente, il quoziente e il resto della divisione intera (**IDIV**) tra **+26227** e 2^4; come sappiamo, in questo caso i risultati coincidono in quanto, **IDIV** produce un quoziente arrotondato verso lo zero, e anche **SAR** produce un valore arrotondato verso lo zero (in sostanza, il valore **+1639.1875** diventa **+1639**, sia per **IDIV**, sia per **SAR**).

Poniamo **AX=1110011001110000b=-6544** (numero intero negativo, multiplo intero di **16**), ed eseguiamo l'istruzione:

```
sar ax, 4
```

Il risultato prodotto da questa istruzione è:

```
AX = 1111111001100111b = -409
```

Inoltre, i **4** bit traboccati da destra, rappresentano il valore **0000b=0**.
Come si può notare, i due valori **-409** e **0** rappresentano, rispettivamente, il quoziente e il resto della divisione intera (**IDIV**) tra **-6544** e 2^4; come sappiamo, in questo caso i risultati coincidono in quanto il resto della divisione è zero (quoziente esatto).

Poniamo **AX=1000011001110001b=-31119** (numero intero negativo, non divisibile per **16**), ed eseguiamo l'istruzione:

```
sar ax, 4
```

Il risultato prodotto da questa istruzione è:

```
AX = 1111100001100111b = -1945
```

Inoltre, i **4** bit traboccati da destra, rappresentano il valore **0001b=+1**.
Come si può notare, il valore **-1945** non coincide con il quoziente **-1944** della divisione intera (**IDIV**) tra **-31119** e 2^4; inoltre, il valore **+1** non coincide con il resto **-15** della divisione stessa. Come sappiamo, in questo caso i risultati non coincidono in quanto, **IDIV** produce un quoziente arrotondato verso lo zero, mentre **SAR** produce un valore arrotondato verso l'infinito negativo; in sostanza, il valore **-1944.9375** diventa **-1944** per **IDIV**, e **-1945** per **SAR**!

In base alle considerazioni appena esposte, si sconsiglia vivamente l'utilizzo di **SAR** per eseguire divisioni rapide tra un numero intero con segno e un

divisore del tipo 2^n; infatti, il rischio di ottenere risultati sbagliati è troppo elevato.
In questi casi, conviene utilizzare **IDIV** che fornisce sempre il risultato corretto; il prezzo da pagare consiste, naturalmente, in una minore velocità di esecuzione.

19.9.2 Shift arithmetic right su operandi di ampiezza arbitraria

Supponiamo di voler far scorrere verso destra, i bit di un numero intero con segno di ampiezza arbitraria; in tal caso, possiamo facilmente adattare il procedimento già illustrato per l'istruzione **SHR**.
Rispetto all'esempio illustrato per **SHR**, l'unica novità è data dal fatto che nello shift aritmetico verso destra è necessario preservare il bit di segno dell'operando; a tale proposito, non dobbiamo fare altro che utilizzare **SAR** per lo shifting del byte più significativo dell'operando, e **SHR** per lo shifting degli altri byte!

Come esempio pratico, consideriamo la seguente definizione:

```
bigNum db 10010010b, 00111100b, 11011111b ; 110111110011110010010010b
```

Il codice che esegue lo scorrimento unitario verso destra, assume il seguente aspetto:

```
; shifting unitario aritmetico del BYTE n. 2 di bigNum

xor     al, al              ; al = 0
mov     ah, BigNum[2]       ; ah = 11011111b
sar     ax, 1               ; ah = 11101111b, al = 10000000b
mov     BigNum[2], ah       ; BigNum[2] = 11101111b
mov     bl, al              ; bl = al = 10000000b

; shifting unitario logico del BYTE n. 1 di bigNum

xor     al, al              ; al = 0
mov     ah, BigNum[1]       ; ah = 00111100b
shr     ax, 1               ; ah = 00011110b, al = 00000000b
or      ah, bl              ; ah = 10011110b
mov     BigNum[1], ah       ; BigNum[1] = 10011110b
mov     bl, al              ; bl = al = 00000000b

; shifting unitario logico del BYTE n. 0 di bigNum

xor     al, al              ; al = 0
mov     ah, BigNum[0]       ; ah = 10010010b
shr     ax, 1               ; ah = 01001001b, al = 00000000b
or      ah, bl              ; ah = 01001001b
mov     BigNum[0], ah       ; BigNum[0] = 01001001b
```

Unendo i **3** risultati precedenti, otteniamo:

```
BigNum = 111011111001111001001001b
```

Il bit meno significativo di **BigNum**, traboccato da destra, si trova memorizzato nel bit più significativo di **AL**.

Se vogliamo visualizzare il contenuto binario di **BigNum**, possiamo procedere nel solito modo con l'ausilio di **writeBin8**; in questo caso, l'output deve iniziare da **BigNum[2]**.

19.9.3 Effetti provocati da SAR sugli operandi e sui flags

L'esecuzione dell'istruzione **SAR**, modifica il contenuto del solo operando **DEST** che viene sovrascritto dal risultato prodotto dall'istruzione stessa; il contenuto dell'operando **SRC** rimane inalterato.
Bisogna però prestare particolare attenzione al caso che già conosciamo, rappresentato da istruzioni del tipo:

```
sar cx, cl
```

L'esecuzione dell'istruzione **SAR** con **CL=0** o **Imm8=0**, non modifica nessun campo del **Flags Register**; ovviamente, in un caso del genere, il contenuto di **DEST** rimane inalterato.
L'esecuzione dell'istruzione **SAR** con contatore non nullo, modifica i campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; il solo campo **AF** assume un valore indeterminato, e non ha quindi nessun significato.
Il campo **OF** assume un valore sensato solo quando il contatore vale **1** (e quindi anche con **CL=1** o **Imm8=1**); se il contatore è maggiore di **1**, il flag **OF** assume un valore indeterminato, e non ha quindi nessun significato. Come si può facilmente immaginare, nel caso di uno scorrimento unitario aritmetico verso destra, la **CPU** pone sempre **OF=0**; infatti, l'istruzione **SAR** preserva sempre il bit di segno dell'operando **DEST**.
Il campo **CF** (nel caso in cui il contatore sia diverso da zero), contiene l'ultimo bit traboccato dalla destra di **DEST**.
I flags **PF**, **ZF** e **SF** (nel caso in cui il contatore sia diverso da zero) forniscono informazioni ordinarie relative al risultato prodotto da **SAR**; in sostanza, **PF** indica se i primi **8** bit del risultato contengono un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il risultato vale zero, **SF** indica se il bit più significativo del risultato vale **1**.

19.10 Le istruzioni SHLD e SHRD

Con le **CPU 80386** e superiori, vengono rese disponibili due ulteriori istruzioni di shifting, rappresentate dai mnemonici **SHLD** e **SHRD**; tali istruzioni sono molto simili a **SHL** e **SHR**, ma rispetto a queste ultime, svolgono un lavoro molto più complesso.
Entrambe le istruzioni richiedono tre operandi; le cifre del primo operando (**DEST**) vengono sottoposte ad un numero di scorrimenti indicato dal terzo operando (contatore). I posti che si liberano nell'operando **DEST**, vengono riempiti con altrettanti bit prelevati dal secondo operando (**SRC**); il contenuto del secondo operando rimane inalterato.
Il termine "doppia precisione" si riferisce al fatto che con queste istruzioni, possiamo effettuare operazioni di shifting su un operando **SRC** a **64** bit (come, ad esempio, una coppia di registri a **32** bit del tipo **EDX:EAX**).

Come al solito, vengono presi in considerazione solamente i **5** bit meno significativi del contatore; questa volta, però, il contenuto del contatore non viene modificato (in sostanza, la **CPU** si serve di un registro temporaneo, nel quale viene memorizzato il contenuto dei primi **5** bit del contatore).

19.10.1 L'istruzione SHLD

Con il mnemonico **SHLD** si indica l'istruzione **SHift logical Left, Double precision** (scorrimento logico verso sinistra, in doppia precisione); lo scopo di questa istruzione è quello di effettuare uno scorrimento logico verso sinistra, dei bit dell'operando **DEST** (primo operando). Il numero di scorrimenti da effettuare, viene indicato dal contenuto del terzo operando (contatore); i posti rimasti liberi a destra nell'operando **DEST**, vengono riempiti con altrettanti bit fatti traboccare dalla sinistra dell'operando **SRC** (secondo operando). Il contenuto dell'operando **SRC** non subisce nessuna modifica.
Per ogni singolo scorrimento, il bit più significativo dell'operando **DEST** trabocca da sinistra; il valore di tale bit, viene salvato nel flag **CF**.
I tre operandi devono essere specificati, esplicitamente, dal programmatore; il risultato prodotto da **SHLD**, viene memorizzato nell'operando **DEST**.
Le uniche forme lecite per l'istruzione **SHLD** sono le seguenti:

```
SHLD  Reg16, Reg16, Imm8
SHLD  Reg16, Reg16, CL
SHLD  Mem16, Reg16, Imm8
SHLD  Mem16, Reg16, CL
SHLD  Reg32, Reg32, Imm8
SHLD  Reg32, Reg32, CL
SHLD  Mem32, Reg32, Imm8
SHLD  Mem32, Reg32, CL
```

Come esempio pratico poniamo **EAX=11010111001010110111000111001010b**, **EBX=00111011000111111100100111110000010b**, **CL=4**, ed eseguiamo la seguente istruzione:

```
shld eax, ebx, cl
```

In presenza di questa istruzione, la **CPU** fa scorrere di **4** posti verso sinistra, tutti i bit di **EAX**; i **4** posti che si liberano a destra, vengono riempiti con altrettanti bit fatti traboccare dalla sinistra del registro **EBX**. Alla fine otteniamo il seguente risultato:

```
EAX = 01110010101101110001110010100011b, EBX=00111011000111111100100111110000010b, CF = 1
```

Come si può notare, il contenuto di **EBX** rimane inalterato; inoltre, l'ultimo bit traboccato dalla sinistra di **EAX** è un **1**, per cui alla fine otteniamo **CF=1**.

Nei precedenti capitoli abbiamo visto che determinate istruzioni della **CPU**, richiedono che uno dei loro operandi sia rappresentato, obbligatoriamente, dalla coppia **DX:AX**; si presenta quindi spesso la necessità di salvare in **DX:AX**, un valore a **32** bit (con **DX** che deve contenere la **WORD** più significativa).
Supponendo che il valore a **32** bit sia memorizzato in **EAX**, possiamo allora procedere in questo modo:

```
push    eax                 ; salva eax nello stack
pop     ax                  ; ax = word meno significativa
pop     dx                  ; dx = word più significativa
```

Una soluzione molto più semplice consiste, invece, nello scrivere l'istruzione:

```
shld edx, eax, 16
```

19.10.2 L'istruzione SHRD

Con il mnemonico **SHRD** si indica l'istruzione **SHift logical Right, Double precision** (scorrimento logico verso destra, in doppia precisione); lo scopo di questa istruzione è quello di effettuare uno scorrimento logico verso destra, dei bit dell'operando **DEST** (primo operando). Il numero di scorrimenti da effettuare, viene indicato dal contenuto del terzo operando (contatore); i posti rimasti liberi a sinistra nell'operando **DEST**, vengono riempiti con altrettanti bit fatti traboccare dalla destra dell'operando **SRC** (secondo operando). Il contenuto dell'operando **SRC** non subisce nessuna modifica.
Per ogni singolo scorrimento, il bit meno significativo dell'operando **DEST** trabocca da destra; il valore di tale bit, viene salvato nel flag **CF**.
I tre operandi devono essere specificati, esplicitamente, dal programmatore; il risultato prodotto da **SHRD**, viene memorizzato nell'operando **DEST**.
Le uniche forme lecite per l'istruzione **SHRD** sono le seguenti:

```
SHRD  Reg16, Reg16, Imm8
SHRD  Reg16, Reg16, CL
SHRD  Mem16, Reg16, Imm8
SHRD  Mem16, Reg16, CL
SHRD  Reg32, Reg32, Imm8
SHRD  Reg32, Reg32, CL
SHRD  Mem32, Reg32, Imm8
SHRD  Mem32, Reg32, CL
```

Come esempio pratico poniamo **EAX=00000111001010110111000111001010b**, **EBX=00111011000111111100100111101111b**, **CL=4**, ed eseguiamo la seguente istruzione:

```
shrd eax, ebx, cl
```

In presenza di questa istruzione, la **CPU** fa scorrere di **4** posti verso destra, tutti i bit di **EAX**; i **4** posti che si liberano a sinistra, vengono riempiti con altrettanti bit fatti traboccare dalla destra del registro **EBX**. Alla fine otteniamo il seguente risultato:

```
EAX=11110000011100101011011100011100b, EBX=00111011000111111100100111101111b, CF = 1
```

Come si può notare, il contenuto di **EBX** rimane inalterato; inoltre, l'ultimo bit traboccato dalla destra di **EAX** è un **1**, per cui alla fine otteniamo **CF=1**.

19.10.3 Effetti provocati da SHLD e SHRD, sugli operandi e sui flags

L'esecuzione delle istruzioni **SHLD** e **SHRD**, modifica il contenuto del solo operando **DEST** che viene sovrascritto dal risultato prodotto dalle istruzioni stesse; il contenuto dell'operando **SRC** rimane inalterato.
Bisogna però prestare particolare attenzione ai casi rappresentati da istruzioni del tipo:

```
shld ecx, ecx, cl
```

Si ricordi che in presenza delle istruzioni **SHLD** e **SHRD**, la **CPU** utilizza come contatore il contenuto dei primi **5** bit di **CL**, senza apportare nessuna modifica allo stesso **CL**; ne consegue che la precedente istruzione, non provoca nessuna modifica sul contenuto del registro destinazione **ECX**!

L'esecuzione delle istruzioni **SHLD** e **SHRD** con **CL=0** o **Imm8=0**, non modifica nessun campo del **Flags Register**; ovviamente, in un caso del genere, il contenuto di

L'esecuzione delle istruzioni **SHLD** e **SHRD** con contatore non nullo, modifica i campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; il solo campo **AF** assume un valore indeterminato, e non ha quindi nessun significato.
Il campo **OF** assume un valore sensato solo quando il contatore vale **1**; se il contatore è maggiore di **1**, il flag **OF** assume un valore indeterminato, e non ha quindi nessun significato. Come si può facilmente immaginare, la **CPU** pone **OF=1** quando le istruzioni **SHLD** e **SHRD** provocano la modifica del bit più significativo (bit di segno) di **DEST**; in caso contrario, la **CPU** pone **OF=0**.
Il campo **CF** (nel caso in cui il contatore sia diverso da zero), contiene l'ultimo bit traboccato da **DEST**.
I flags **PF**, **ZF** e **SF** (nel caso in cui il contatore sia diverso da zero) forniscono informazioni ordinarie relative al risultato prodotto da **SHLD** e **SHRD**; in sostanza, **PF** indica se i primi **8** bit del risultato contengono un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il risultato vale zero, **SF** indica se il bit più significativo del risultato vale **1**.

A differenza di quanto accade con **SHL**, **SAL**, **SHR** e **SAR**, se il contatore specifica un numero eccessivo di scorrimenti, le istruzioni **SHLD** e **SHRD** forniscono risultati privi di senso; ciò accade, ad esempio, con una istruzione del tipo:

```
shrd ax, bx, 30
```

In un caso del genere, anche i flags assumono valori privi di significato!

19.11 L'istruzione ROL

Con il mnemonico **ROL** si indica l'istruzione **ROtate Left** (rotazione verso sinistra); lo scopo di questa istruzione è quello di effettuare una rotazione verso sinistra, dei bit dell'operando **DEST**. Il numero di rotazioni da effettuare, viene indicato dall'operando **SRC**.
Per ogni singola rotazione, il bit più significativo dell'operando **DEST** trabocca da sinistra; tale bit viene salvato nel flag **CF**, e contemporaneamente, viene fatto rientrare dalla destra dello stesso operando **DEST**.
Entrambi gli operandi devono essere specificati, esplicitamente, dal programmatore; il risultato prodotto da **ROL**, viene memorizzato nell'operando **DEST**.
Le uniche forme lecite per l'istruzione **ROL** sono le seguenti:

```
ROL   Reg, 1
ROL   Reg, CL
ROL   Reg, Imm8
ROL   Mem, 1
ROL   Mem, CL
ROL   Mem, Imm8
```

Il codice macchina generale per l'istruzione **ROL**, è formato dal campo **Opcode 110100vw** e dal campo **mod_000_r/m**; se **v=0**, allora l'operando **SRC** è rappresentato dal valore immediato **1** (una sola rotazione), mentre se **v=1**, allora l'operando **SRC** è rappresentato dall'half register **CL** (che contiene il numero di rotazioni da effettuare). Con le **CPU 80286** e superiori, il numero di rotazioni può essere indicato anche attraverso un **Imm8**; in tal caso, il codice macchina è formato dal campo **Opcode 1100000w** e dal campo **mod_000_r/m**.

Come esempio pratico poniamo **AX=0111001011011100b**, **CL=5**, ed eseguiamo la seguente istruzione:

```
rol ax, cl
```

In presenza di questa istruzione, la **CPU** fa ruotare di **5** posti verso sinistra, tutti i bit di **AX**; ogni singolo bit che trabocca dalla sinistra di **AX**, viene memorizzato in **CF**, e contemporaneamente, viene fatto rientrare dalla destra dello stesso **AX**. La successione delle singole rotazioni produce quindi i seguenti risultati:

```
AX = 1110010110111000b, CF = 0

AX = 1100101101110001b, CF = 1

AX = 1001011011100011b, CF = 1

AX = 0010110111000111b, CF = 1

AX = 0101101110001110b, CF = 0
```

L'ultimo bit sottoposto a rotazione è uno **0**; di conseguenza, alla fine otteniamo **AX=0101101110001110b** e **CF=0**.

19.11.1 ROL su operandi di ampiezza arbitraria

Supponiamo di voler far ruotare verso sinistra, i bit di un numero binario di ampiezza arbitraria; in tal caso, possiamo facilmente adattare il procedimento già illustrato per l'istruzione **SHL**.

Come esempio pratico, consideriamo la seguente definizione:

```
bigNum db 10010010b, 00111100b, 10011111b ; 100111110011110010010010b
```

Il codice che esegue la rotazione unitaria verso sinistra, assume il seguente aspetto:

```
; shifting unitario del BYTE n. 0 di bigNum

xor     ah, ah              ; ah = 0
mov     al, BigNum[0]       ; al = 10010010b
shl     ax, 1               ; ah = 00000001b, al = 00100100b
mov     BigNum[0], al       ; BigNum[0] = 00100100b
mov     bl, ah              ; bl = ah = 00000001b

; shifting unitario del BYTE n. 1 di bigNum

xor     ah, ah              ; ah = 0
mov     al, BigNum[1]       ; al = 00111100b
shl     ax, 1               ; ah = 00000000b, al = 01111000b
or      al, bl              ; al = 01111001b
mov     BigNum[1], al       ; BigNum[1] = 01111001b
mov     bl, ah              ; bl = ah = 00000000b

; shifting unitario del BYTE n. 2 di bigNum

xor     ah, ah              ; ah = 0
mov     al, BigNum[2]       ; al = 10011111b
```

```
or      al, bl              ; al = 00111110b
mov     BigNum[2], al       ; BigNum[2] = 00111110b

; rotazione del bit più significativo

or      BigNum[0], ah       ; BigNum[0] = 00100101b
```

Unendo i **3** risultati precedenti, otteniamo:

BigNum = 001111100111100100100101b

Il bit più significativo di **BigNum**, traboccato da sinistra, oltre ad essere rientrato da destra, si trova anche memorizzato nel bit meno significativo di **AH**.

Se vogliamo visualizzare il contenuto binario di **BigNum**, possiamo procedere nel solito modo con l'ausilio di **writeBin8**; in questo caso, l'output deve iniziare da **BigNum[2]**.

19.11.2 Effetti provocati da ROL sugli operandi e sui flags

L'esecuzione dell'istruzione **ROL**, modifica il contenuto del solo operando **DEST** che viene sovrascritto dal risultato prodotto dall'istruzione stessa; il contenuto dell'operando **SRC** rimane inalterato.
Bisogna però prestare particolare attenzione al caso che già conosciamo, rappresentato da istruzioni del tipo:

```
rol cx, cl
```

L'esecuzione dell'istruzione **ROL** con **CL=0** o **Imm8=0**, non modifica nessun campo del **Flags Register**; ovviamente, in un caso del genere, il contenuto di **DEST** rimane inalterato.
L'esecuzione dell'istruzione **ROL** con contatore non nullo, modifica i campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; il solo campo **AF** assume un valore indeterminato, e non ha quindi nessun significato.
Il campo **OF** assume un valore sensato solo quando il contatore vale **1** (e quindi anche con **CL=1** o **Imm8=1**); se il contatore è maggiore di **1**, il flag **OF** assume un valore indeterminato, e non ha quindi nessun significato. Come si può facilmente immaginare, la **CPU** pone **OF=1** quando l'istruzione **ROL** provoca la modifica del bit più significativo (bit di segno) di **DEST**; in caso contrario, la **CPU** pone **OF=0**.
Il campo **CF** (nel caso in cui il contatore sia diverso da zero), contiene l'ultimo bit traboccato dalla sinistra di **DEST**.
I flags **PF**, **ZF** e **SF** (nel caso in cui il contatore sia diverso da zero) forniscono informazioni ordinarie relative al risultato prodotto da **ROL**; in sostanza, **PF** indica se i primi **8** bit del risultato contengono un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il risultato vale zero, **SF** indica se il bit più significativo del risultato vale **1**.

19.12 L'istruzione RCL

Con il mnemonico **RCL** si indica l'istruzione **Rotate through Carry Left** (rotazione verso sinistra attraverso **CF**); lo scopo di questa istruzione è quello di effettuare una rotazione verso sinistra, dei bit dell'operando **DEST+CF**, con **CF** che ricopre il ruolo di bit meno significativo. Il numero di rotazioni da effettuare, viene indicato dall'operando **SRC**.

dalla destra di **DEST**, provocando lo scorrimento di una posizione verso sinistra, di tutti i bit dello stesso **DEST**; il bit più significativo di **DEST** trabocca da sinistra, e viene salvato nel flag **CF**.
Entrambi gli operandi devono essere specificati, esplicitamente, dal programmatore; il risultato prodotto da **RCL**, viene memorizzato nell'operando **DEST**.
Le uniche forme lecite per l'istruzione **RCL** sono le seguenti:

```
RCL   Reg, 1
RCL   Reg, CL
RCL   Reg, Imm8
RCL   Mem, 1
RCL   Mem, CL
RCL   Mem, Imm8
```

Il codice macchina generale per l'istruzione **RCL**, è formato dal campo **Opcode 110100vw** e dal campo **mod_010_r/m**; se **v=0**, allora l'operando **SRC** è rappresentato dal valore immediato **1** (una sola rotazione), mentre se **v=1**, allora l'operando **SRC** è rappresentato dall'half register **CL** (che contiene il numero di rotazioni da effettuare). Con le **CPU 80286** e superiori, il numero di rotazioni può essere indicato anche attraverso un **Imm8**; in tal caso, il codice macchina è formato dal campo **Opcode 1100000w** e dal campo **mod_010_r/m**.

Come esempio pratico poniamo **AX=0111001011011100b**, **CL=5**, **CF=1**, ed eseguiamo la seguente istruzione:

```
rcl ax, cl
```

In presenza di questa istruzione, la **CPU** fa ruotare di **5** posti verso sinistra, tutti i bit di **AX+CF**; in sostanza, stiamo sottoponendo a rotazione verso sinistra, un operando a **17** bit, con **CF** che ricopre il ruolo di bit meno significativo. La successione delle singole rotazioni produce quindi i seguenti risultati:

```
AX = 1110010110111001b, CF = 0

AX = 1100101101110010b, CF = 1

AX = 1001011011100101b, CF = 1

AX = 0010110111001011b, CF = 1

AX = 0101101110010111b, CF = 0
```

L'ultimo bit traboccato da sinistra è uno **0**; di conseguenza, alla fine otteniamo **AX=0101101110010111b** e **CF=0**.

19.12.1 RCL su operandi di ampiezza arbitraria

Supponiamo di voler far ruotare verso sinistra, con l'ausilio di **CF**, i bit di un numero binario di ampiezza arbitraria; in tal caso, possiamo facilmente adattare il procedimento già illustrato per l'istruzione **SHL**.

Come esempio pratico, consideriamo la seguente definizione:

```
bigNum db 10010010b, 00111100b, 10011111b ; 100111110011110010010010b
```

Assumendo di avere, inizialmente, **CF=0**, possiamo scrivere il seguente codice:

```
; shifting unitario del BYTE n. 0 di bigNum

pushf                       ; preserva i flags (CF = 0)
xor     ah, ah              ; ah = 0
mov     al, BigNum[0]       ; al = 10010010b
popf                        ; ripristina i flags (CF = 0)
rcl     ax, 1               ; ah = 00000001b, al = 00100100b
mov     BigNum[0], al       ; BigNum[0] = 00100100b
mov     bl, ah              ; bl = ah = 00000001b

; shifting unitario del BYTE n. 1 di bigNum

xor     ah, ah              ; ah = 0
mov     al, BigNum[1]       ; al = 00111100b
shl     ax, 1               ; ah = 00000000b, al = 01111000b
or      al, bl              ; al = 01111001b
mov     BigNum[1], al       ; BigNum[1] = 01111001b
mov     bl, ah              ; bl = ah = 00000000b

; shifting unitario del BYTE n. 2 di bigNum

xor     ah, ah              ; ah = 0
mov     al, BigNum[2]       ; al = 10011111b
shl     ax, 1               ; ah = 00000001b, al = 00111110b
or      al, bl              ; al = 00111110b
mov     BigNum[2], al       ; BigNum[2] = 00111110b
sahf                        ; SF,ZF,-,AF,-,PF,-,CF = 00000001b
```

Unendo i **3** risultati precedenti, otteniamo:

```
BigNum = 001111100111100100100100b, CF = 1
```

Il vecchio contenuto di **CF**, e cioè **0**, viene fatto entrare dalla destra di **BigNum**; tutti i bit di **BigNum** scorrono di una posizione verso sinistra. Il bit più significativo di **BigNum**, trabocca da sinistra e finisce nel bit meno significativo di **AH**; a questo punto, l'istruzione **SAHF** pone **CF=1** (infatti, **CF** occupa la posizione n. **0** nel registro **FLAGS**).

Se vogliamo visualizzare il contenuto binario di **BigNum**, possiamo procedere nel solito modo con l'ausilio di **writeBin8**; in questo caso, l'output deve iniziare da **BigNum[2]**.

19.12.2 Effetti provocati da RCL sugli operandi e sui flags

L'esecuzione dell'istruzione **RCL**, modifica il contenuto del solo operando **DEST** che viene sovrascritto dal risultato prodotto dall'istruzione stessa; il contenuto dell'operando **SRC** rimane inalterato.
Bisogna però prestare particolare attenzione al caso che già conosciamo, rappresentato da istruzioni del tipo:

```
rcl cx, cl
```

L'esecuzione dell'istruzione **RCL** con **CL=0** o **Imm8=0**, non modifica nessun campo

rimane inalterato.
L'esecuzione dell'istruzione **RCL** con contatore non nullo, modifica i campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; il solo campo **AF** assume un valore indeterminato, e non ha quindi nessun significato.
Il campo **OF** assume un valore sensato solo quando il contatore vale **1** (e quindi anche con **CL=1** o **Imm8=1**); se il contatore è maggiore di **1**, il flag **OF** assume un valore indeterminato, e non ha quindi nessun significato. Come si può facilmente immaginare, la **CPU** pone **OF=1** quando l'istruzione **RCL** provoca la modifica del bit più significativo (bit di segno) di **DEST**; in caso contrario, la **CPU** pone **OF=0**.
Il campo **CF** (nel caso in cui il contatore sia diverso da zero), contiene l'ultimo bit traboccato dalla sinistra di **DEST**.
I flags **PF**, **ZF** e **SF** (nel caso in cui il contatore sia diverso da zero) forniscono informazioni ordinarie relative al risultato prodotto da **RCL**; in sostanza, **PF** indica se i primi **8** bit del risultato contengono un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il risultato vale zero, **SF** indica se il bit più significativo del risultato vale **1**.

19.13 L'istruzione ROR

Con il mnemonico **ROR** si indica l'istruzione **ROtate Right** (rotazione verso destra); lo scopo di questa istruzione è quello di effettuare una rotazione verso destra, dei bit dell'operando **DEST**. Il numero di rotazioni da effettuare, viene indicato dall'operando **SRC**.
Per ogni singola rotazione, il bit meno significativo dell'operando **DEST** trabocca da destra; tale bit viene salvato nel flag **CF**, e contemporaneamente, viene fatto rientrare dalla sinistra dello stesso operando **DEST**.
Entrambi gli operandi devono essere specificati, esplicitamente, dal programmatore; il risultato prodotto da **ROR**, viene memorizzato nell'operando **DEST**.
Le uniche forme lecite per l'istruzione **ROR** sono le seguenti:

```
ROR   Reg, 1
ROR   Reg, CL
ROR   Reg, Imm8
ROR   Mem, 1
ROR   Mem, CL
ROR   Mem, Imm8
```

Il codice macchina generale per l'istruzione **ROR**, è formato dal campo **Opcode 110100vw** e dal campo **mod_001_r/m**; se **v=0**, allora l'operando **SRC** è rappresentato dal valore immediato **1** (una sola rotazione), mentre se **v=1**, allora l'operando **SRC** è rappresentato dall'half register **CL** (che contiene il numero di rotazioni da effettuare). Con le **CPU 80286** e superiori, il numero di rotazioni può essere indicato anche attraverso un **Imm8**; in tal caso, il codice macchina è formato dal campo **Opcode 1100000w** e dal campo **mod_001_r/m**.

Come esempio pratico poniamo **AX=0111001011011100b**, **CL=5**, ed eseguiamo la seguente istruzione:

```
ror ax, cl
```

In presenza di questa istruzione, la **CPU** fa ruotare di **5** posti verso destra, tutti i bit di **AX**; ogni singolo bit che trabocca dalla destra di **AX**, viene memorizzato in **CF**, e contemporaneamente, viene fatto rientrare dalla sinistra dello stesso **AX**. La successione delle singole rotazioni produce quindi i

seguenti risultati:

AX = 0011100101101110b, CF = 0

AX = 0001110010110111b, CF = 0

AX = 1000111001011011b, CF = 1

AX = 1100011100101101b, CF = 1

AX = 1110001110010110b, CF = 1

L'ultimo bit sottoposto a rotazione è un **1**; di conseguenza, alla fine otteniamo **AX=1110001110010110b** e **CF=1**.

19.13.1 ROR su operandi di ampiezza arbitraria

Supponiamo di voler far ruotare verso destra, i bit di un numero binario di ampiezza arbitraria; in tal caso, possiamo facilmente adattare il procedimento già illustrato per l'istruzione **SHR**.

Come esempio pratico, consideriamo la seguente definizione:

```
bigNum db 10010011b, 00111100b, 00011111b ; 000111110011110010010011b
```

Il codice che esegue la rotazione unitaria verso destra, assume il seguente aspetto:

```
; shifting unitario del BYTE n. 2 di bigNum

xor     al, al              ; al = 0
mov     ah, BigNum[2]       ; ah = 00011111b
shr     ax, 1               ; ah = 00001111b, al = 10000000b
mov     BigNum[2], ah       ; BigNum[2] = 00001111b
mov     bl, al              ; bl = al = 10000000b

; shifting unitario del BYTE n. 1 di bigNum

xor     al, al              ; al = 0
mov     ah, BigNum[1]       ; ah = 00111100b
shr     ax, 1               ; ah = 00011110b, al = 00000000b
or      ah, bl              ; ah = 10011110b
mov     BigNum[1], ah       ; BigNum[1] = 10011110b
mov     bl, al              ; bl = al = 00000000b

; shifting unitario del BYTE n. 0 di bigNum

xor     al, al              ; al = 0
mov     ah, BigNum[0]       ; ah = 10010011b
shr     ax, 1               ; ah = 01001001b, al = 10000000b
or      ah, bl              ; ah = 01001001b
mov     BigNum[0], ah       ; BigNum[0] = 01001001b

; rotazione del bit meno significativo

or      BigNum[2], al       ; BigNum[2] = 10001111b
```

Unendo i **3** risultati precedenti, otteniamo:

```
BigNum = 10001111100111100100100 1b
```

Il bit meno significativo di **BigNum**, traboccato da destra, oltre ad essere rientrato da sinistra, si trova anche memorizzato nel bit più significativo di **AL**.

Se vogliamo visualizzare il contenuto binario di **BigNum**, possiamo procedere nel solito modo con l'ausilio di **writeBin8**; in questo caso, l'output deve iniziare da **BigNum[2]**.

19.13.2 Effetti provocati da ROR sugli operandi e sui flags

L'esecuzione dell'istruzione **ROR**, modifica il contenuto del solo operando **DEST** che viene sovrascritto dal risultato prodotto dall'istruzione stessa; il contenuto dell'operando **SRC** rimane inalterato.
Bisogna però prestare particolare attenzione al caso che già conosciamo, rappresentato da istruzioni del tipo:

```
ror cx, cl
```

L'esecuzione dell'istruzione **ROR** con **CL=0** o **Imm8=0**, non modifica nessun campo del **Flags Register**; ovviamente, in un caso del genere, il contenuto di **DEST** rimane inalterato.
L'esecuzione dell'istruzione **ROR** con contatore non nullo, modifica i campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; il solo campo **AF** assume un valore indeterminato, e non ha quindi nessun significato.
Il campo **OF** assume un valore sensato solo quando il contatore vale **1** (e quindi anche con **CL=1** o **Imm8=1**); se il contatore è maggiore di **1**, il flag **OF** assume un valore indeterminato, e non ha quindi nessun significato. Come si può facilmente immaginare, la **CPU** pone **OF=1** quando l'istruzione **ROR** provoca la modifica del bit più significativo (bit di segno) di **DEST**; in caso contrario, la **CPU** pone **OF=0**.
Il campo **CF** (nel caso in cui il contatore sia diverso da zero), contiene l'ultimo bit traboccato dalla destra di **DEST**.
I flags **PF**, **ZF** e **SF** (nel caso in cui il contatore sia diverso da zero) forniscono informazioni ordinarie relative al risultato prodotto da **ROR**; in sostanza, **PF** indica se i primi **8** bit del risultato contengono un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il risultato vale zero, **SF** indica se il bit più significativo del risultato vale **1**.

19.14 L'istruzione RCR

Con il mnemonico **RCR** si indica l'istruzione **Rotate through Carry Right** (rotazione verso destra attraverso **CF**); lo scopo di questa istruzione è quello di effettuare una rotazione verso destra, dei bit dell'operando **CF+DEST**, con **CF** che ricopre il ruolo di bit più significativo. Il numero di rotazioni da effettuare, viene indicato dall'operando **SRC**.
Per ogni singola rotazione, il contenuto corrente di **CF** viene fatto entrare dalla sinistra di **DEST**, provocando lo scorrimento di una posizione verso destra, di tutti i bit dello stesso **DEST**; il bit meno significativo di **DEST** trabocca da destra, e viene salvato nel flag **CF**.
Entrambi gli operandi devono essere specificati, esplicitamente, dal programmatore; il risultato prodotto da **RCR**, viene memorizzato nell'operando

Le uniche forme lecite per l'istruzione **RCR** sono le seguenti:

```
RCR   Reg, 1
RCR   Reg, CL
RCR   Reg, Imm8
RCR   Mem, 1
RCR   Mem, CL
RCR   Mem, Imm8
```

Il codice macchina generale per l'istruzione **RCR**, è formato dal campo **Opcode 110100vw** e dal campo **mod_011_r/m**; se **v=0**, allora l'operando **SRC** è rappresentato dal valore immediato **1** (una sola rotazione), mentre se **v=1**, allora l'operando **SRC** è rappresentato dall'half register **CL** (che contiene il numero di rotazioni da effettuare). Con le **CPU 80286** e superiori, il numero di rotazioni può essere indicato anche attraverso un **Imm8**; in tal caso, il codice macchina è formato dal campo **Opcode 1100000w** e dal campo **mod_011_r/m**.

Come esempio pratico poniamo **AX=0111001011011100b**, **CL=5**, **CF=1**, ed eseguiamo la seguente istruzione:

```
rcr ax, cl
```

In presenza di questa istruzione, la **CPU** fa ruotare di **5** posti verso destra, tutti i bit di **CF+AX**; in sostanza, stiamo sottoponendo a rotazione verso destra, un operando a **17** bit, con **CF** che ricopre il ruolo di bit più significativo. La successione delle singole rotazioni produce quindi i seguenti risultati:

```
AX = 1011100101101110b, CF = 0

AX = 0101110010110111b, CF = 0

AX = 0010111001011011b, CF = 1

AX = 1001011100101101b, CF = 1

AX = 1100101110010110b, CF = 1
```

L'ultimo bit traboccato da destra è un **1**; di conseguenza, alla fine otteniamo **AX=1100101110010110b** e **CF=1**.

19.14.1 RCR su operandi di ampiezza arbitraria

Supponiamo di voler far ruotare verso destra, con l'ausilio di **CF**, i bit di un numero binario di ampiezza arbitraria; in tal caso, possiamo facilmente adattare il procedimento già illustrato per l'istruzione **SHR**.

Come esempio pratico, consideriamo la seguente definizione:

```
bigNum db 10010011b, 00111100b, 10011111b ; 100111110011110010010011b
```

Assumendo di avere, inizialmente, **CF=0**, possiamo scrivere il seguente codice:

```
; shifting unitario del BYTE n. 2 di bigNum

pushf                       ; preserva i flags (CF = 0)
xor     al, al              ; al = 0
```

```
mov     ah, BigNum[2]       ; ah = 10011111b
popf                        ; ripristina i flags (CF = 0)
rcr     ax, 1               ; ah = 01001111b, al = 10000000b
mov     BigNum[2], ah       ; BigNum[2] = 01001111b
mov     bl, al              ; bl = al = 10000000b

; shifting unitario del BYTE n. 1 di bigNum

xor     al, al              ; al = 0
mov     ah, BigNum[1]       ; ah = 00111100b
shr     ax, 1               ; ah = 00011110b, al = 00000000b
or      ah, bl              ; ah = 10011110b
mov     BigNum[1], ah       ; BigNum[1] = 10011110b
mov     bl, al              ; bl = al = 00000000b

; shifting unitario del BYTE n. 0 di bigNum

xor     al, al              ; al = 0
mov     ah, BigNum[0]       ; ah = 10010011b
shr     ax, 1               ; ah = 01001001b, al = 10000000b
or      ah, bl              ; ah = 01001001b
mov     BigNum[0], ah       ; BigNum[0] = 01001001b

mov     ah, al              ; ah = 10000000b
shr     ah, 7               ; ah = 00000001b
sahf                        ; SF,ZF,-,AF,-,PF,-,CF = 00000001b
```

Unendo i **3** risultati precedenti, otteniamo:

BigNum = 010011111001111001001001b, CF = 1

Il vecchio contenuto di **CF**, e cioè **0**, viene fatto entrare dalla sinistra di **BigNum**; tutti i bit di **BigNum** scorrono di una posizione verso destra. Il bit meno significativo di **BigNum**, trabocca da destra e finisce nel bit più significativo di **AL**; a questo punto, spostiamo **AL** in **AH**, facciamo scorrere di **7** posti verso destra il contenuto di **AH**, ed eseguiamo una istruzione **SAHF** che pone **CF=1** (infatti, **CF** occupa la posizione n. **0** nel registro **FLAGS**).

Se vogliamo visualizzare il contenuto binario di **BigNum**, possiamo procedere nel solito modo con l'ausilio di **writeBin8**; in questo caso, l'output deve iniziare da **BigNum[2]**.

19.14.2 Effetti provocati da RCR sugli operandi e sui flags

L'esecuzione dell'istruzione **RCR**, modifica il contenuto del solo operando **DEST** che viene sovrascritto dal risultato prodotto dall'istruzione stessa; il contenuto dell'operando **SRC** rimane inalterato.
Bisogna però prestare particolare attenzione al caso che già conosciamo, rappresentato da istruzioni del tipo:

```
rcr cx, cl
```

L'esecuzione dell'istruzione **RCR** con **CL=0** o **Imm8=0**, non modifica nessun campo del **Flags Register**; ovviamente, in un caso del genere, il contenuto di **DEST** rimane inalterato.
L'esecuzione dell'istruzione **RCR** con contatore non nullo, modifica i campi **CF**,

indeterminato, e non ha quindi nessun significato.
Il campo **OF** assume un valore sensato solo quando il contatore vale **1** (e quindi anche con **CL=1** o **Imm8=1**); se il contatore è maggiore di **1**, il flag **OF** assume un valore indeterminato, e non ha quindi nessun significato. Come si può facilmente immaginare, la **CPU** pone **OF=1** quando l'istruzione **RCR** provoca la modifica del bit più significativo (bit di segno) di **DEST**; in caso contrario, la **CPU** pone **OF=0**.
Il campo **CF** (nel caso in cui il contatore sia diverso da zero), contiene l'ultimo bit traboccato dalla destra di **DEST**.
I flags **PF**, **ZF** e **SF** (nel caso in cui il contatore sia diverso da zero) forniscono informazioni ordinarie relative al risultato prodotto da **RCR**; in sostanza, **PF** indica se i primi **8** bit del risultato contengono un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il risultato vale zero, **SF** indica se il bit più significativo del risultato vale **1**.

19.15 Istruzioni logiche nei linguaggi di alto livello

Molti linguaggi di alto livello, forniscono due categorie di istruzioni logiche chiamate, **operatori logici** e **operatori booleani**; gli operatori logici agiscono sui singoli bit dei loro operandi (proprio come succede con le istruzioni della **CPU** esaminate in questo capitolo), mentre gli operatori booleani agiscono sul cosiddetto **valore booleano** dei loro operandi.
Il valore booleano indica se il contenuto di un operando è uguale o diverso da zero; un operando è **TRUE** (o **1**) se il suo contenuto è diverso da zero, mentre è **FALSE** (o **0**) se il suo contenuto è uguale a zero.

Se **A** e **B** sono due operandi di tipo numerico intero, allora una espressione del tipo:

```
C = A AND B
```

viene definita **espressione logica**; il risultato che viene memorizzato in **C** è un valore numerico intero, conseguenza di un **AND logico** (bit per bit) tra i due operandi **A** e **B**.

Se **A** e **B** sono due operandi di tipo booleano (**TRUE** o **FALSE**), allora una espressione del tipo:

```
C = A AND B
```

viene definita **espressione booleana**; il risultato che viene memorizzato in **C** è un valore booleano, conseguenza di un **AND booleano** tra i due operandi **A** e **B**. Il risultato è **TRUE** se, e solo se, entrambi gli operandi sono **TRUE**.

Osserviamo allora che se **A=10101010b** e **B=01010101b**, un **AND** logico produce:

```
A AND B = 10101010b AND 01010101b = 00000000b = 0
```

Un **AND** booleano produce, invece:

```
A AND B = TRUE AND TRUE = TRUE = 1
```

(sia **A** che **B** hanno, infatti, un valore non nullo).

Nel linguaggio **C/C++**, le due categorie di istruzioni logiche utilizzano nomi differenti; tali nomi vengono illustrati in Figura 5.

Figura 5 - Operatori logici e booleani del C/C++

Operatori logici		Operatori booleani	
Nome	**Significato**	**Nome**	**Significato**
&	AND logico	&&	AND booleano
~	NOT logico		
|	OR logico inclusivo	||	OR booleano inclusivo
^	OR logico esclusivo		
<<	SHL logico		
>>	SHR logico		

In **C/C++** la condizione **TRUE** viene indicata dal valore numerico **1** (o da un qualsiasi valore diverso da zero); invece, la condizione **FALSE** viene indicata dal valore numerico **0**.
Gli operatori logici possono essere applicati solamente a variabili di tipo intero; gli operatori booleani possono essere applicati a qualsiasi variabile numerica.
Per capire la differenza di comportamento tra gli operatori logici e booleani del **C/C++**, possiamo eseguire il seguente programma **C**:

```
/* file bool.c */

#include <stdio.h>

unsigned short int   v1 = 0x0F0F;   /* 0000111100001111b */
unsigned short int   v2 = 0xF0F0;   /* 1111000011110000b */

int main(void)
{
   printf("Operatori logici:\n");
   printf("v1 or v2  = %X\n", v1 | v2);
   printf("v1 and v2 = %X\n", v1 & v2);
   printf("Operatori booleani:\n");
   printf("v1 or v2  = %u\n", v1 || v2);
   printf("v1 and v2 = %u\n", v1 && v2);

   return 0;
}
```

Nel linguaggio **Pascal/Delphi**, le due categorie di istruzioni logiche utilizzano gli stessi nomi, e vengono distinte dal compilatore in base al contesto in cui compaiono; tali nomi vengono illustrati in Figura 6.

Figura 6 - Operatori logici e booleani del Pascal/Delphi

Operatori logici		Operatori booleani	
Nome	**Significato**	**Nome**	**Significato**
and	AND logico	and	AND booleano
not	NOT logico	not	NOT booleano
or	OR logico inclusivo	or	OR booleano inclusivo
xor	OR logico esclusivo	xor	OR booleano esclusivo

shl	SHL logico		
shr	SHR logico		

In **Pascal/Delphi** la condizione **TRUE** viene indicata dal nome riservato **true**; invece, la condizione **FALSE** viene indicata dal nome riservato **false**.
Gli operatori logici possono essere applicati solamente a variabili di tipo intero; gli operatori booleani possono essere applicati solamente a variabili di tipo booleano.
Per capire la differenza di comportamento tra gli operatori logici e booleani del **Pascal/Delphi**, possiamo eseguire il seguente programma **Pascal**:

```
{ file bool.pas }

program Bool;

var
   v1, v2:      Word;
   b1, b2:      Boolean;

begin
   WriteLn('Operatori logici:');
   v1 := $0F0F;                         { 0000111100001111b }
   v2 := $F0F0;                         { 1111000011110000b }
   WriteLn('v1 or v2  = ', v1 or v2);
   WriteLn('v1 and v2 = ', v1 and v2);
   WriteLn('Operatori booleani:');
   b1 := true;
   b2 := false;
   WriteLn('b1 or b2  = ', b1 or b2);
   WriteLn('b1 and b2 = ', b1 and b2);
end.
```

Come vedremo in un apposito capitolo, anche gli assembler come **MASM** e **TASM** forniscono una serie di operatori logici e booleani; tali operatori, non hanno niente a che vedere con le analoghe istruzioni della **CPU**.

19.16 Applicazioni pratiche delle istruzioni logiche

Come applicazione pratica delle istruzioni logiche della **CPU**, possiamo riprendere l'esempio del precedente capitolo, relativo alla conversione in binario di un numero **Unpacked BCD**; abbiamo visto che applicando il metodo delle divisioni successive (con divisore **2**) al numero **07050301h** (in formato **Unpacked BCD**), otteniamo la seguente sequenza di resti:

R0=01h, R1=01h, R2=00h, R3=01h, R4=00h, R5=01h, R6=01h

R7=00h, R8=01h, R9=00h, R10=01h, R11=01h, R12=01h

I vari resti rappresentano le cifre binarie del numero decimale **7531**; come facciamo a memorizzare tali resti nei singoli bit di un numero binario?

Adesso che abbiamo a disposizione le istruzioni logiche, possiamo rispondere a questa domanda in modo molto semplice; come si può facilmente intuire, possiamo risolvere il problema con l'ausilio delle sole istruzioni **OR** e **SHL**!

Prima di tutto, predisponiamo le seguenti definizioni:

```
resto       db    01h, 01h, 00h, 01h, 00h, 01h, 01h
                  00h, 01h, 00h, 01h, 01h, 01h
bcd2binary  dw    0
```

A questo punto, possiamo scrivere il seguente codice:

```
mov     al, resto[0]               ; al = 00000001b
or      byte ptr bcd2binary[0], al ; bcd2binary = 0000000000000001b

mov     al, resto[1]               ; al = 00000001b
shl     al, 1                      ; al = 00000010b
or      byte ptr bcd2binary[0], al ; bcd2binary = 0000000000000011b

mov     al, resto[2]               ; al = 00000000b
shl     al, 2                      ; al = 00000000b
or      byte ptr bcd2binary[0], al ; bcd2binary = 0000000000000011b

mov     al, resto[3]               ; al = 00000001b
shl     al, 3                      ; al = 00001000b
or      byte ptr bcd2binary[0], al ; bcd2binary = 0000000000001011b

; ... e così via
```

Dopo aver riempito gli **8** bit meno significativi di **bcd2binary**, dobbiamo procedere allo stesso modo sulla variabile **byte ptr bcd2binary[1].**

< Sommario >