FONDAMENTI DI INFORMATICA II

ESERCITAZIONE n° 1

IL LINGUAGGIO ASSEMBLER 8086

ESERCIZIO n° 1

Indicare (in binario) lo stato dei flag C, O, S, Z, P e del registro AH dopo l'esecuzione delle due istruzioni:

```
MOV AH, 70
ADD AH, 70
```

Soluzione

La prima istruzione carica in AH la quantità 70 (46H), la seconda, sommando altrettanto, di fatto la duplica. Poiché però la ALU opera considerando i numeri espressi in notazione in complemento a due, si ha overflow, in quanto tale somma eccede il massimo intero positivo rappresentabile su 8 bit, che è 127 (7FH). Di conseguenza, lo stato finale dei flag è il seguente:

- C=0 (NC [No Carry], nessun riporto)
- O=1 (OV [OVerflow], overflow)
- S=1 (NG [NeGative], risultato negativo)
- Z=0 (NZ [Not Zero], risultato diverso da zero)
- P=0 (PO [Parity Odd], il numero totale di "1" nel risultato, 8CH, è dispari)

Il valore finale contenuto in AH è quindi 8CH = 1000 1100, interpretabile in complemento a due come -116.

NOTA: volendo provare il programma con l'utility DEBUG, tenere presente che i valori numerici immediati sono interpretati, e devono perciò sempre essere introdotti, in esadecimale.

ESERCIZIO n° 2

Indicare (in binario) lo stato dei flag C, O, S, Z, P e del registro AX dopo l'esecuzione delle due istruzioni:

```
MOV AX, 18000
ADD AL, AH
```

Soluzione

In primo luogo, occorre convertire 18000 in esadecimale: il risultato è 4650H. L'operazione successiva equivale perciò a sommare le quantità esadecimali 46H (70) e 50H (80). Poiché la destinazione della somma è AL, AH resta immutato: il suo valore finale è ancora 46H. Il valore finale di AL è invece 96H, che dà luogo a overflow per invasione del bit di segno. Di conseguenza, lo stato finale dei flag è il seguente:

- C=0 (NC, nessun riporto)
- O=1 (OV, overflow)
- S=1 (NG, risultato negativo)
- Z=0 (NZ, risultato diverso da zero)

- P=1 (PE [Parity Even], il numero totale di "1" nel risultato, 96H, è pari)

Il valore finale contenuto in AL è quindi 96H = 1001 0110, interpretabile in complemento a due come -106.

NOTA: volendo provare il programma con l'utility DEBUG, tenere presente che i valori numerici immediati sono interpretati, e devono perciò sempre essere introdotti, in esadecimale.

ESERCIZIO n° 3

Indicare (in binario) lo stato dei flag C, O, S, Z, P e del registro AX dopo l'esecuzione delle tre istruzioni:

```
STC
MOV AX, -17981
ADC AL, AH
```

Soluzione

In primo luogo, occorre convertire -17981 in esadecimale: il risultato è B9C3H. L'operazione successiva equivale perciò a sommare le quantità esadecimali B9H (-71) e C3H (-61) mettendo però in conto anche il flag di carry, come richiesto dalla istruzione ADC (in luogo della precedente ADD). Per determinare il risultato è quindi necessario conoscere il valore di tale bit. In questo caso, ciò è immediato, in quanto il flag C è stato esplicitamente settato dall'istruzione STC (SeT Carry). Di conseguenza, AH resta immutato (B9H), mentre il valore finale di AL è 7DH = 0111 1101, interpretabile in complemento a due come +125 e che dà luogo a overflow per invasione del bit di segno. Lo stato finale dei flag è il seguente:

- C=1 (CY [CarrY], riporto)
- O=1 (OV, overflow)
- S=0 (PL [Positive vaLue], risultato positivo)
- Z=0 (NZ, risultato diverso da zero)
- P=1 (PE, il numero totale di "1" nel risultato, 7DH, è pari)

NOTA: volendo provare il programma con l'utility DEBUG, tenere presente che i valori numerici immediati sono interpretati, e devono perciò sempre essere introdotti, in esadecimale.

ESERCIZIO n° 4

Indicare (in binario) lo stato dei flag C, O, S, Z, P e del registro AX dopo l'esecuzione delle tre istruzioni:

```
MOV BX, 1234H
MOV AX, 9C31H
ADD AX, BX
```

Soluzione

Si tratta di sommare le quantità esadecimali 1234H e 9C31H, interpretabili secondo la notazione in complemento a due come 4660 e -25551. Di conseguenza, il valore finale di AX è AE65H, interpretabile in complemento a due come -20891. Per determinare lo stato finale dei flag, è necessario precisare che *il flag di parità si riferisce sempre ai soli otto bit meno significativi* del risultato. Perciò, in quei casi (come quello in questione) in cui l'operazione è a 16 bit, solo metà di essi concorrono alla formazione dello stato del flag P. La situazione definitiva è perciò la seguente:

- C=0 (NC, nessun riporto)

- O=0 (NV [No oVerflow], nessun overflow)
- S=1 (NG, risultato negativo)
- Z=0 (NZ, risultato diverso da zero)
- P=1 (PE, il numero totale di "1" nel byte meno significativo del risultato, 65H, è pari)

NOTA: volendo provare il programma con l'utility DEBUG, tenere presente che i valori numerici immediati sono interpretati, e devono perciò sempre essere introdotti, in esadecimale.

ESERCIZIO n° 5

Indicare (in binario) lo stato dei flag C, O, S, Z, P e del registro AX dopo l'esecuzione delle tre istruzioni:

```
MOV AX, 6838H ; interpretati come due valori BCD, 68 e 38
ADD AL, AH
DAA
```

Soluzione

L'istruzione ADD somma le due quantità contenute in AL e AH come fossero quantità binarie (o, se si preferisce, esadecimali). Di conseguenza, il valore di AL successivo alla somma è A0H, interpretabile in un'ottica "mista" binario/BCD come 100_d (A*10=10*10=100). Sebbene tutto appaia normale, la "spia" che è accaduto qualcosa di potenzialmente errato (qualora si stia operando in BCD, come nel caso in questione) è costituita dalla generazione, nel corso della somma, di un riporto intermedio fra il primo e il secondo nibble, indicata dal flag di *carry ausiliario*, che risulta perciò settato. La successiva istruzione DAA[1], intercettando questa condizione, può quindi provvedere a correggere il risultato come richiesto, sommando 6 a entrambi i nibble. Poiché il risultato complessivo, 106H, non è rappresentabile su un byte, si genera overflow. La situazione successiva alla esecuzione dell'istruzione ADD è perciò la seguente:

- C=0 (NC, nessun riporto)
- O=1 (OV, overflow)
- S=1 (NG, risultato negativo)
- Z=0 (NZ, risultato diverso da zero)
- P=1 (PE, il numero totale di "1" del risultato, A0H, è pari)
- A=1 (AC [Auxiliary Carry], riporto ausiliario)

Al contrario, la situazione finale, dopo DAA, risulta:

- C=1 (CY, riporto)
- O=0 (NV, nessun overflow)
- S=0 (PL, risultato positivo)
- Z=0 (NZ, risultato diverso da zero)
- P=1 (PE, parità pari)
- A=1 (AC, riporto ausiliario [settato da prima])

NOTA: volendo provare il programma con l'utility DEBUG, tenere presente che i valori numerici immediati sono interpretati, e devono perciò sempre essere introdotti, in esadecimale.

[1] Le istruzione di aggiustamento BCD, DAA e DAS, operano su addizioni e sottrazioni ad ampiezza di byte, ovvero col risultato in AL (e non in AX!).

ESERCIZIO n° 6

Scrivere un programma Assembler che sommi i primi N numeri naturali, in modo equivalente alla funzione C:

```
int somma(int N)
{
  int i, s=0;
  for ( i=0; i<=N; i++ )
        s += i;
  return s;
}
```

Soluzione

Per progettare il codice in assembler, occorre evidentemente prendere alcune decisioni. In particolare, in questo caso si assume che:

- il numero N si trovi in memoria, alla locazione indicata simbolicamente dalla label N;
- il risultato vada posto, alla fine, alla locazione di memoria indicata dalla label RES;
- si usi il registro AX per sommare i valori;
- si usi il registro BX per contare da 0 a N.

Tralasciando in questa sede le necessarie direttive all'assemblatore, ci concentriamo sulla parte attiva del codice, che risulta:

```
        MOV BX, [N]     ; pone in BX il contenuto della cella etichettata "N"
        XOR AX,AX       ; azzera AX facendo lo XOR del suo contenuto con se stesso
ciclo:  ADD AX, BX      ; aggiungi l'addendo corrente
        DEC BX          ; decrementa il contatore
        JNE ciclo       ; se non è zero, salta a "ciclo"
        MOV [RES], AX   ; salva il risultato nella cella etichettata RES
```

Si consiglia di verificare, a mezzo del programma DEBUG, l'effettivo funzionamento della procedura, usando per N e RES due aree di memoria libere a propria scelta, tenendo presente che i valori numerici immediati sono interpretati, e devono perciò sempre essere introdotti, in esadecimale.

ESERCIZIO n° 7

Scrivere un programma Assembler che effettui la conversione da stringa a numero, in modo equivalente all'algoritmo C:

```
int s=0;
for ( i=0; st [i] != '\0';  i++ )
      { int t = st [i] - '0';  s = s*10 + t; }
```

Soluzione

Assumiamo che:

- la stringa sia in formato C (terminata da '\0'), etichettata dalla label STRINGA;
- il risultato vada posto nella locazione di memoria indicata dalla label RES;
- si usi il registro SI per contenere il moltiplicatore, 10;
- si usi il registro DI per scorrere i caratteri della stringa (analogo alla variabile "i" sopra);

- si usi il registro AX per sommare via via i valori (analogo alla variabile "s" sopra);
- si usi il registro BX per contenere l'indirizzo iniziale della stringa.

Tralasciando in questa sede le necessarie direttive all'assemblatore, ci concentriamo sulla parte attiva del codice, che risulta:

```
        MOV SI, 10              ; SI ← 10 (base)
        XOR DI, DI              ; azzera DI
        XOR AX, AX              ; azzera AX
        MOV BX, offset STRINGA  ; BX ← indirizzo iniziale STRINGA
        XOR CH, CH              ; azzera CH
ciclo:  MOV CL, [BX+DI]         ; CL ← carattere corrente della stringa
        CMP CL, 0               ; confronta col terminatore 0
        JE fine                 ; se è uguale, salta a "fine"
        SUB CX, '0'             ; se no, sottrai il codice del carattere '0'
        MUL SI                  ; moltiplica AX (implicito) per SI
        ADD AX, CX              ; aggiungi ad AX il nuovo contributo
        INC DI                  ; incrementa il puntatore alla stringa
        JMP ciclo               ; ripeti il ciclo
fine:   MOV [RES], AX           ; salva in RES il risultato
```

Si consiglia di verificare, a mezzo del programma DEBUG, l'effettivo funzionamento della procedura, usando per STRINGA e RES due aree di memoria libere a propria scelta, tenendo presente che i valori numerici immediati sono interpretati, e devono perciò sempre essere introdotti, in esadecimale. Ciò vale anche per i valori ASCII: '0' dovrà quindi essere introdotto come 30 (hex).

ESERCIZIO n° 8

Scrivere un programma Assembler che effettui la conversione duale della precedente, da numero a stringa (in formato C, ovvero terminata da '\0').

Soluzione

In primo luogo, osserviamo che il noto algoritmo per divisioni successive produce le cifre in ordine dalla meno significativa alla più significativa, cioè da destra a sinistra: la stringa dovrebbe quindi essere riempita a partire dal fondo. Poiché questo è scomodo, perché non sappiamo a priori quanto essa sarà lunga, decidiamo di sfruttare la caratteristica LIFO dello stack per invertire "gratis" l'ordine dele cifre: in pratica, accumuleremo le cifre via via prodotte nello stack, prelevandole poi per metterle nella stringa al termine della fase di generazione. Così facendo, esse si renderanno disponibili a partire dall'ultima prodotta, che è appunto quella da porre in testa alla stringa. A sua volta, questo richiede di contare le cifre prodotte nella prima fase (equivalenti al numero delle PUSH) per effettuare, nella seconda fase, il corretto numero di POP.

Assumiamo che:
- la stringa da produrre vada posta in memoria alla locazione etichettata STRINGA;
- il numero da convertire si trovi inizialmente nella locazione di memoria etichettata NUM;
- si usi il registro SI per contenere il divisore, 10;
- si usi il registro BX per contenere l'indirizzo iniziale della stringa e quindi per scorrerla;
- si usi il registro CX per contare i caratteri prodotti (ossia la lunghezza della stringa generata);
- si usi il registro AX per ospitare via via i quozienti del calcolo;

- si usi il registro DX per ospitare via via i resti delle divisioni.

Queste due ultime scelte sono in effetti obbligate, in quanto l'istruzione DIV dell'Assembler 8086 presume di avere il dividendo in AX, e pone il AX il quoziente e il DX il resto: l'unico grado di libertà è costituito dal registro in cui porre il divisore, che nel nostro caso si è assunto essere SI.

La parte attiva del codice risulta:

```
        MOV SI, 10                ; SI ← 10 (base)
        MOV BX, offset STRINGA    ; BX ← indirizzo iniziale STRINGA
        MOV AX, [NUM]             ; AX ← numero da convertire
        XOR CX,CX                 ; azzera CX
                                  ;
ciclo:  SUB DX, DX                ; azzera DX
        DIV SI                    ; calcola AX = AX / SI; resto in DX
        ADD DX, '0'               ; converti la cifra nel carattere ASCII
        PUSH DX                   ; salva il carattere nello stack
        INC CX                    ; incrementa il contatore (lungh. stringa)
        OR AX,AX                  ; test se AX==0 (senza alterare AX)
        JNZ ciclo                 ; se AX!=0, ripeti il ciclo
                                  ;
  rep:  POP DX                    ; preleva la cifra dallo stack
        MOV [BX], DL              ; poni in memoria la cifra corrente
        INC BX                    ; aggiorna puntatore alla stringa
        DEC CX                    ; decrementa il contatore delle cifre
        JNZ rep                   ; ripeti se ce ne sono altre
                                  ;
        MOV [BX], AL              ; aggiungi in fondo il terminatore '\0'
```

Da notare l'ultima istruzione, che sostituisce la forse più ovvia (MOV [BX], 0), che non esiste in questo Assembler (l'indirizzamento indiretto con operando immediato non è consentito con il registro BX) .Si consiglia di verificare, a mezzo del programma DEBUG, l'effettivo funzionamento della procedura, usando per NUM e STRINGA due aree di memoria libere a propria scelta, tenendo presente che i valori numerici immediati sono interpretati, e devono perciò sempre essere introdotti, in esadecimale. Ciò vale anche per i valori ASCII: '0' dovrà quindi essere introdotto come 30 (hex).