

Architetture dei Sistemi a Elaborazione – a.a. 2010/11

Esercitazione di Laboratorio 2

1. Si scriva un programma in linguaggio Assembler 8086 che esegua le seguenti operazioni:
 - a. Definisca un vettore VETT di 3 elementi, positivi e negativi, ciascuno su 8 bit, inizializzato a piacimento
 - b. Si esegua l'ordinamento degli elementi di VETT

n.b. si utilizzi il meccanismo di *salto indiretto*.

suggerimento: si disegni il diagramma di flusso per definire la tabella di jump opportuna (vedi slide 10-11-12 del blocco "Istruzioni per il controllo di flusso").

Possibile soluzione:

```
JTAB    DW      LAB_CAB, LAB_ACB, LAB_ABC, LAB_CBA, LAB_BCA, LAB_BAC
        .CODE
        .STARTUP

        MOV AH, VETT
        MOV BH, VETT+1
        MOV CH, VETT+2
        MOV SI, 0

        CMP AH, BH
        JG  A_g_B
        ADD SI, 6
        CMP CH, BH
        JG  SWITCH      ; SI = 6
        ADD SI, 2
        CMP CH, AH
        JG  SWITCH      ; SI = 8
        ADD SI, 2
        JMP SWITCH      ; SI = 10

A_g_B:  CMP CH, AH
        JG  SWITCH      ; SI = 0
        ADD SI, 2
        CMP CH, BH
        JG  SWITCH      ; SI = 2
        ADD SI, 2
        JMP SWITCH      ; SI = 4

SWITCH: JMP JTAB[SI]

LAB_ABC: MOV VETT,  AH
        MOV VETT+1, BH
        MOV VETT+2, CH
        JMP FINE
LAB_ACB: MOV VETT,  AH
        MOV VETT+2, BH
        MOV VETT+1, CH
        JMP FINE
LAB_BAC: MOV VETT+1, AH
        MOV VETT,   BH
        MOV VETT+2, CH
        JMP FINE
LAB_BCA: MOV VETT+2, AH
        MOV VETT,   BH
        MOV VETT+1, CH
        JMP FINE
LAB_CAB: MOV VETT+1, AH
        MOV VETT+2, BH
        MOV VETT,   CH
        JMP FINE
LAB_CBA: MOV VETT+2, AH
        MOV VETT+1, BH
        MOV VETT,   CH
        JMP FINE
FINE:    NOP
```

```
.EXIT
END
```

2. Si scriva un programma in linguaggio Assembler 8086 che esegua le seguenti operazioni:
- Definisca un vettore VETT di 20 elementi di dimensione opportuna per contenere i primi 20 numeri della serie di Fibonacci
 - Memorizzi in VETT la serie di Fibonacci in ordine invertito (VETT[19]=0, VETT[18]=1, VETT[17]=1, etc...)

Serie di Fibonacci: $v[i] = v[i-1] + v[i-2] \Rightarrow \text{vet} = 0, 1, 1, 2, 3, 5, 8, 13, 21, \dots$

n.b. si utilizzi il modo di indirizzamento *indirect addressing* e l'istruzione **LOOP**
suggerimento: si assegni il valore all'elemento 19 e 18 di VETT prima di entrare nel ciclo.

Possibile soluzione:

```
N      EQU      20
      .MODEL    small
      .STACK
      .DATA
VETT   DW        N-2 dup (?), 1, 0
      .CODE
      .STARTUP
      MOV CX, N-2
      LEA BX, VETT
      ADD BX, N*2-6
      MOV AX, [BX]+2
NEW:   ADD AX, [BX]+4
      MOV [BX], AX
      SUB BX, 2
      LOOP NEW
      .EXIT
      END
```

3. Si scriva un programma in linguaggio Assembler 8086 che esegua le seguenti operazioni:
- Definisca una variabile VAR su 16 bit inizializzata a piacimento
 - Calcoli il numero di bit al valore 1 contenuti nella variabile e ponga il risultato in una variabile RIS di dimensione opportuna
 - Controlli se il quarto bit di VAR (a partire dal meno significativo) vale 1
 - In caso positivo, inverta il segno del valore contenuto in RIS
 - Altrimenti non esegua alcuna operazione

n.b. il valore di VAR *non deve essere distrutto* durante l'elaborazione.

Possibile soluzione:

```
      .MODEL    small
      .STACK
      .DATA
VAR    DW        1100110011001100B
RIS    DW        0
      .CODE
      .STARTUP

      MOV CX, 16
      MOV AX, VAR
NEW:   ROL AX, 1
      JNC NOT_1
      INC RIS
NOT_1: LOOP NEW
      TEST VAR, 00001000B
      JE     FINE
      NEG   RIS
FINE:  NOP
```

```
.EXIT
END
```

4. Si scriva un programma in linguaggio Assembler 8086 che esegua le seguenti operazioni:
- Definisca in memoria una matrice quadrata MATR di N elementi positivi ciascuno su 8 bit
 - Ne trasformi il contenuto moltiplicando e dividendo (nell'ordine) il valore di ogni cella per il proprio vicino destro e sinistro, rispettivamente.
 - Se l'elemento è su un lato, considerare il valore mancante uguale a 1
 - Se un elemento può provocare una operazione di divisione per 0, sostituirlo col valore 1.

n.b. Si supponga che non si ecceda mai la precisione massima raggiungibile con il tipo byte.

Esempio:

| | | |
|---------|---|-----------|
| 1 2 4 5 | ➔ | 2 8 10 1 |
| 2 3 5 7 | ➔ | 6 7 11 1 |
| 3 4 6 8 | ➔ | 12 8 12 1 |
| 0 3 5 1 | ➔ | 0 15 1 0 |

Possibile soluzione:

```
lato EQU 4
.MODEL small
.STACK
.DATA
A DB 1, 2, 4, 5
  DB 2, 3, 5, 7
  DB 3, 4, 6, 8
  DB 0, 3, 5, 1
B DB 4 dup (?)
  DB 4 dup (?)
  DB 4 dup (?)
  DB 4 dup (?)
.CODE
.STARTUP
MOV SI, 0
MOV BX, 0
avanti: MOV AL, A[BX][SI]
        CMP SI, 0
        JE no_div
        CMP SI, lato - 1
        JE no_mul
        MUL byte PTR A[BX][SI+1]
        CMP A[BX][SI-1], 0
        JE fatto
        DIV byte PTR A[BX][SI-1]
fatto:  MOV B[BX][SI], AL
        INC SI
        JMP avanti
no_div: MUL byte PTR A[BX][SI+1]
        JMP fatto
no_mul: MOV AH, 0
        DIV byte PTR A[BX][SI-1]
        MOV B[BX][SI], AL
        ADD BX, lato
        MOV SI, 0
        CMP BX, lato*lato
        JNE avanti
        LEA DI, A
.EXIT
END
```

```
; Si scriva un programma che esegua le seguenti operazioni:
; a. Definisca un vettore VETT di 3 elementi, positivi e negativi,
;    ciascuno su 8 bit, inizializzato a piacimento
; b. Si esegua l'ordinamento degli elementi di VETT
; n.b. si utilizzi il meccanismo di salto indiretto.
```

```
.model small
.stack
.data
```

```
VETT db 3, 4, 5
TAB dw label1, label2
    dw label3, label4
    dw label5, label6
MIN db 0
```

```
.code
.startup
```

```
    lea si, VETT
    mov al, VETT[0]
    cmp al, VETT[1]
    jl jump1
```

```
jump4:    ; VETT[0] > VETT[1]
    mov al, VETT[1]
    cmp al, VETT[2]
    jl jump3    ; VETT[1] < VETT[2]
    mov bx, 10    ; label6
    jmp ordina
```

```
jump3:    mov al, VETT[0]
    mov bx, 6
    cmp al, VETT[2]
    jl ordina    ; label4
    mov bx, 8    ; label5
    jmp ordina
```

```
jump1:    mov al, VETT[1]
    cmp al, VETT[2]
    jg jump2
    mov bx, 0
    jmp ordina    ;label1
```

```
jump2:    mov al, VETT[0]
    mov bx, 2
    cmp al, VETT[2]
    jl ordina    ;label2
    mov bx, 4
    jmp ordina    ;label3
```

```
ordina:    jmp TAB[bx]
```

```
    ; gia' ordinati
label1:    jmp fine
```

```
    ; a, c, b
label2:    mov al, VETT[1]
    mov bl, VETT[2]
    mov byte ptr VETT[2], al    ; cast non necessario, al e' a 8 bit
    mov byte ptr VETT[1], bl
    jmp fine
```

```
    ; c, a, b
label3:    mov al, VETT
    mov bl, VETT[1]
    mov cl, VETT[2]
    mov byte ptr VETT, cl
    mov byte ptr VETT[1], al
    mov byte ptr VETT[2], bl
    jmp fine
```

```

; b, a, c
label4:  mov al, VETT
        mov bl, VETT[1]
        mov byte ptr VETT, bl
        mov byte ptr VETT[1], al
        jmp fine

; b, c, a
label5:  mov al, VETT
        mov bl, VETT[1]
        mov cl, VETT[2]
        mov byte ptr VETT, bl
        mov byte ptr VETT[1], cl
        mov byte ptr VETT[2], al
        jmp fine

; c, b, a
label6:  mov al, VETT
        mov bl, VETT[2]
        mov byte ptr VETT, bl
        mov byte ptr VETT[2], al

fine:    nop

.exit
end
```

```

; Si scriva un programma che esegua le seguenti operazioni:
; a. Definisca un vettore VETT di 20 elementi di dimensione opportuna
; per contenere i primi 20 numeri della serie di Fibonacci
; b. Memorizzi in VETT la serie di Fibonacci in ordine invertito
; (VETT[19] = 0, VETT[18] = 1, VETT[17] = 1, etc...)
;
; n.b. Si utilizzi il modo di indirizzamento indirect addressing e l'istruzione LOOP

```

```

LEN equ 20

```

```

.model small
.stack
.data

```

```

VETT dw 0, 1, 1, 2, 3, 5, 8, 13, 21, 34
dw 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181

```

```

.code
.startup

```

```

mov cx, LEN/2          ; numero di iterazioni
lea si, VETT            ; puntatore al primo elemento
lea bx, VETT+(LEN*2)-2  ; puntatore all'ultimo elemento

```

```

ciclo: mov ax, [bx]
xchg ax, [si]
mov [bx], ax
add si, 2
sub bx, 2
LOOP ciclo

```

```

.exit
End

```

;Si scriva un programma che esegua le seguenti operazioni:
; **a. Definisca una variabile VAR su 16 bit inizializzata a piacimento**
; **b. Calcoli il numero di bit al valore 1 contenuti nella variabile e**
; **ponga il risultato in una variabile RIS di dimensione opportuna**
; **c. Controlli se il quarto bit di VAR (a partire dal meno significativo) vale 1**
; **i. In caso positivo, inverta il segno del valore contenuto in RIS**
; **ii. Altrimenti non esegua alcuna operazione**
;n.b. Il valore di VAR non deve essere distrutto durante l'elaborazione

.model small
.stack
.data

VAR dw 12345
RIS db 0

.code
.startup

leax si, VAR
mov cx, 16
mov bl, 0
mov ax, VAR
ciclo: ror ax, 1
adc bl, 0
loop ciclo
mov byte ptr RIS, bl
test VAR, 0008h
jz fine
not bl
inc bl
mov byte ptr RIS, bl

fine: nop

.exit
end

```
;Si scriva un programma che esegua le seguenti operazioni:
;   a. Definisca in memoria una matrice quadrata MATR di N elementi
;       positivi ciascuno su 8 bit
;   b. Ne trasformi il contenuto moltiplicando e dividendo (nell'ordine)
;       il valore di ogni cella per il proprio vicino destro e sinistro,
;       rispettivamente:
;       i. Se l'elemento è su un lato, considerare il valore mancante
;           uguale a 1
;       ii. Se un elemento può provocare una operazione di divisione per 0,
;           sostituirlo col valore 1
;n.b. Si supponga che non si ecceda mai la precisione massima raggiungibile
;con il tipo byte
```

```
N      equ 4
CONF_S equ 01h
CONF_D equ 02h
```

```
.model small
.stack
.data
```

```
MATR  db 1, 2, 4, 5
      db 2, 3, 5, 7
      db 3, 4, 6, 8
      db 0, 3, 5, 1
```

```
TEMP  db N*N dup (0)
```

```
.code
.startup
```

```
lea dx, MATR
mov bx, 0
mov si, 0
mov cx, N*N
```

```
ciclo1:      jmp check_confine
continue:    test ax, CONF_S
             jnz confine_sx
             test ax, CONF_D
             jnz confine_dx
```

```
;non sono a confine
```

```
no_confine:  mov al, MATR[bx][si]
             mul MATR[bx][si+1]
             ; verifico che l'elemento a sinistra non sia zero
             test byte ptr MATR[bx][si-1], 0FFh
             jz store_al ; e' 0, divido al per 1, memorizzo quindi al
             div MATR[bx][si-1]
             mov byte ptr TEMP[bx][si], al
```

```
next:       inc si
             cmp si, N
             jz new_row
             dec cx
             jnz ciclo1
             jmp store_matrix
```

```
new_row:    mov si, 0
             add bx, N
             dec cx
             jnz ciclo1
             jmp store_matrix
```

```
store_al:   mov byte ptr TEMP[bx][si], al
             jmp next
```

```
confine_sx: mov al, MATR[bx][si]
             mul MATR[bx][si+1]
             ; sono a confine sinistro, dovrei dividere per 1, non lo faccio
```



```

        mov byte ptr TEMP[bx][si], al
        jmp next

confine_dx:    ; sono a confine destro, dovrei moltiplicare per 1, non lo faccio
                ; verifico che l'elemento di sinistra non sia zero
        mov al, MATR[bx][si-1]
        test al, 0FFh
        ; l'elemento e' 0, moltiplico e divido per uno, l'elemento non varia
        jz next
        mov al, MATR[bx][si]
        div MATR[bx][si-1]
        mov byte ptr TEMP[bx][si], al
        jmp next

check_confine: mov ax, si
                test ax, 0FFFFh
                jz set_conf_s
                cmp ax, N-1
                jz set_conf_d
                jmp no_confine

set_conf_s:    mov ax, CONF_S
                jmp continue
set_conf_d:    mov ax, CONF_D
                jmp continue

store_matrix:  mov cx, N*N
                mov si, 0
ciclo2:        mov al, byte ptr TEMP[si]
                mov byte ptr MATR[si], al
                inc si
                dec cx
                jnz ciclo2

fine:         nop
.EXIT
end

```