Array

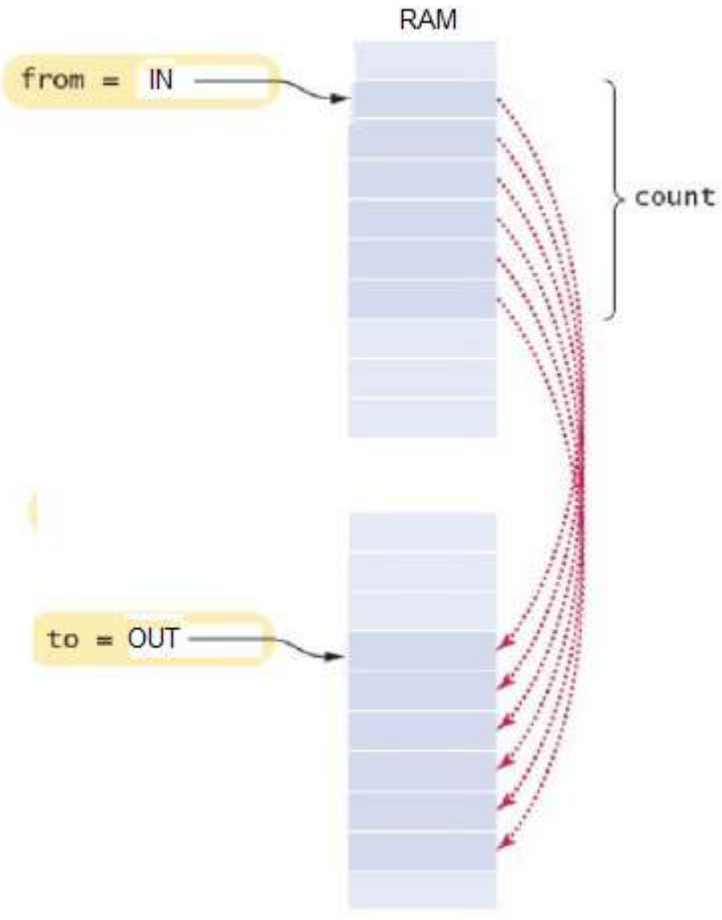
; array ad una dimensione: insieme ordinato omogeneo

IN: dd 4,6 ; **due elementi di tipo intero (ognuno occupa 4 byte)**
OUT: dd 0,0

MOV EBX, IN **; puntatore inizio array**
MOV EDX, OUT **; puntatore fine array**
sposta:
MOV EAX, [EBX] **; primo elemento**
MOV [OUT],EAX

MOV EAX,[EBX+4] **; secondo elemento**
MOV [EDX+4],EAX

Attività: realizzare un programma in ***assembly*** per **copiare** un array di 6 elementi (numeri interi)

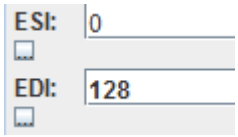


Copiare un array

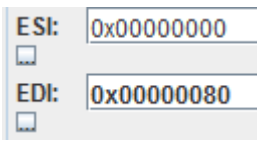
```
; commenta il procedimento e sintetizza la funzionalità del programma

JMP program
procedura:
; ESI =  array begin
; EDI =  array  end
  MOV EAX, EDI; contatore per il loop esterno
  MOV EBX, ESI; contatore per il loop interno
  outer_loop:
    inner_loop:
      ADD EBX, 4
      MOV EDX, [EBX-4]
      CMP [EBX], EDX
      JG inner_loop_end
      MOV ECX, [EBX]
      MOV [EBX], EDX
      MOV [EBX-4], ECX
      inner_loop_end:
      CMP EBX ,EAX
      JNE inner_loop
    SUB EAX, 4
    MOV EBX,ESI
    CMP EAX, ESI
    JNE outer_loop
  RET
program:
data:
dd 12, 14, 15, 1, 12, 11 ,44, 345, 35627, 125 ,34626, 435, 78, 987345 ,234    ;  32 interi che occupano 4 byte
dd 235, 151, 236, 234, 2 ,25, 2623, 6, 3, 54674, 25, 24, 37373, 23, 26, 337, 3  ; da 0x00 a 0x80 escluso
data_end:    ; in 0x80
dd 0
```

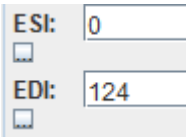

```
MOV ESI, data
MOV EDI, data_end
```

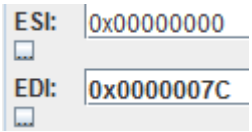


; in codice esadecimale




```
SUB EDI, 4  ; ora ESI ed EDI puntano agli indirizzi che delimitano l'array
            ; 32 elementi  ed ogni elemento occupa 4 byte → da 0 a 124
            ; espresso in codice esadecimale: da 0x0 a 0x07C
```






```
call procedura

; si veda tutorial all'URL http://wwwi10.lrr.in.tum.de/~jasmin/tutorials_applications.html
```

Prima dell'ordinamento in RAM

e ***dopo***

data

address	signed int
0x0	12
0x4	14
0x8	15
0xC	1
0x10	12
0x14	11
0x18	44
0x1C	345
0x20	35627
0x24	125
0x28	34626
0x2C	435
0x30	78
0x34	987345
0x38	234
0x3C	235
0x40	151
0x44	236
0x48	234
0x4C	2
0x50	25
0x54	2623
0x58	6
0x5C	3
0x60	54674
0x64	25
0x68	24
0x6C	37373
0x70	23
0x74	26
0x78	337
0x7C	3
0x80	0

fine

32 elementi non ordinati

address	signed int
0x0	**1**
0x4	**2**
0x8	**3**
0xC	**3**
0x10	**6**
0x14	**11**
0x18	**12**
0x1C	**12**
0x20	**14**
0x24	**15**
0x28	**23**
0x2C	**24**
0x30	**25**
0x34	**25**
0x38	**26**
0x3C	**44**
0x40	**78**
0x44	**125**
0x48	**151**
0x4C	**234**
0x50	**234**
0x54	**235**
0x58	**236**
0x5C	**337**
0x60	**345**
0x64	**435**
0x68	**2623**
0x6C	**34626**
0x70	**35627**
0x74	**37373**
0x78	**54674**
0x7C	**987345**
0x80	**0**

32 elementi ordinati

Bubble Sort - Spiegazione del codice

Struttura

C'è una procedura di bubble-sort che ordina i dati in memoria dall'indirizzo indicato nel registro ESI all'indirizzo indicato nel registro EDI. Il risultato sarà nella stessa posizione nella memoria.

Il resto del programma è per testare la procedura.

Programma principale

Il primo comando è un salto di programma in modo che i dati vengano scritti nella memoria e gli indirizzi di inizio e di fine di questi dati vengono salvato in ESI ed EDI.
L'istruzione Sub è necessaria perché non vogliamo avere lo 0 tra i dati da ordinare.
L'espressione chiamata esegue l'ordinamento bubble-sort.

Bubble-sort

In questa implementazione abbiamo un ciclo infinito che conta quante volte si percorre la lista ed un ciclo interno che conta gli elementi della lista quando scorrono attraverso di essa.

Per prima cosa dobbiamo copiare gli indirizzi che delimitano l'array per ripristinarne più tardi il valore originario.

Cominciamo con il ciclo interno: dobbiamo confrontare tutte le coppie di valori a 32 bit consecutivi dall'inizio alla fine.

L'indirizzo effettivo è salvato in EAX così dobbiamo aggiungere 4 per puntare alla cella successiva. Copiamo il primo dei due valori dalla memoria in EDX e confrontiamo con il secondo.

Sono necessarie due istruzioni perché per un ogni comando è possibile solo un accesso di memoria.

Se il secondo elemento è più piccolo si scambiano i due valori con uso di ECX per salvare temporaneamente uno di loro.
Se il secondo è più grande saltiamo alla fine del ciclo interno.
Se è stata raggiunta la fine dei dati l'ordinamento è terminato altrimenti si continua con il passo successivo.

Il ciclo esterno è semplice da spiegare: dopo ogni ciclo interno sappiamo che l'ultimo valore è il più grande.

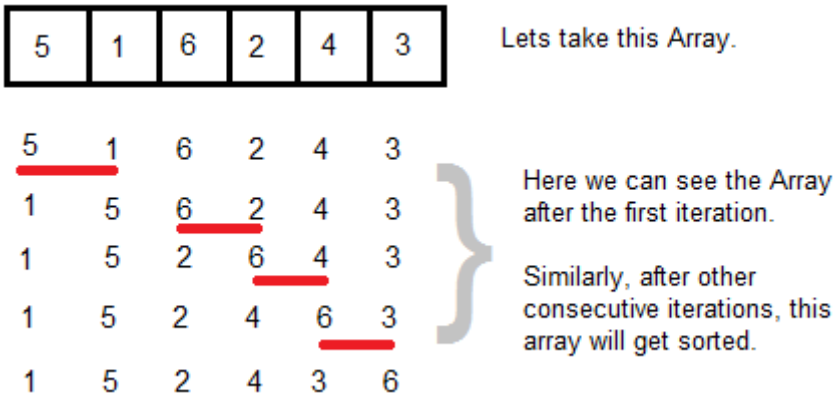
Quindi possiamo escludere l'ultimo valore dai nostri dati sottraendo 4 da EAX (indirizzo di fine array).

Inoltre dobbiamo resettare EBX, il contatore del ciclo interno, e verificare se tutti i dati sono stati ordinati.

Abbiamo terminato quando entrambi gli indirizzi che delimitano l'array puntano alla stessa destinazione in memoria.

Allora possiamo tornare indietro al programma per eseguire l'istruzione successiva alla chiamata.

Attività: realizzare un programma in ***assembly*** per **ordinare** un array di 6 elementi (numeri interi) come in figura:



Attività: realizzare un programma in ***assembly*** per **ordinare** l' array di figura:

6 5 3 1 8 7 2 4

Si usi come algoritmo di ordinamento il *bubble sort* volendo ordinare gli elementi in senso crescente (*non decrescente*): partendo dall'inizio della lista e scorrendola progressivamente, si confrontino due elementi adiacenti e si scambino le loro posizioni se non sono nel giusto ordine (ossia se il secondo elemento è minore del primo).

Dopo la prima iterazione il **massimo** si troverà in **ultima posizione**, quella sua definitiva; dopo la seconda iterazione il secondo massimo si troverà in penultima posizione, anch'essa la sua definitiva; dopo la terza iterazione il terzo massimo si troverà in terz'ultima posizione, anch'essa la sua definitiva e così via.

Ad ogni iterazione almeno un elemento si va a collocare nella sua giusta posizione: la procedura termina quando non ci sono più elementi da scambiare.

Si veda anche l'*animazione* all'URL http://it.wikipedia.org/wiki/Bubble_sort