Seminar III ASC

Reprezentarea little-endian a numerelor în memorie. Salt necondiționat, condiționat. Operații pe șiruri.

3. Conținut

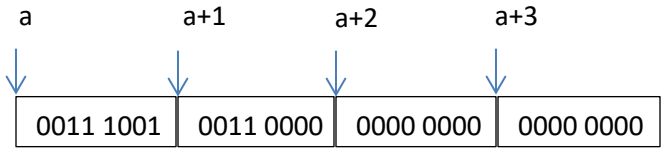
3.1. Reprezentarea little-endian a numerelor în memorie și conversii nedistructive

În memoria calculatorului (nu regiștrii procesorului) numerele reprezentate pe mai mult de un octet sunt reprezentate în formatul *little-endian* (cel mai puțin semnificativ octet din reprezentare se afla la cea mai mică adresă în memorie).
Fiecare număr este reprezentat pe 1, 2, 4 sau 8 octeți (în funcție de contextul în care este declarat; pentru următoarea declarație `a dw 2` numărul doi este reprezentat pe doi octeți). Numărul este reprezentat în binar în interpretarea cu/fără semn. Dacă numărul este reprezentat în memorie pe mai mult de un octet (nu în regiștrii procesorului) octeții reprezentării cu/fără semn sunt stocați în memorie în formatul little-endian.

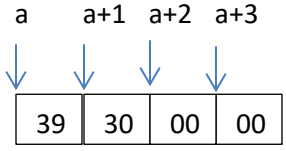
Exemplu:

```
a dd 0
...
mov dword [a], 12345
```

În exemplul de mai sus, numărul 12345 este pozitiv și va fi reprezentat fără semn pe patru octeți ca și $(00000000000000000011000000111001)_2 = (00003039)_{16}$ (conversia rapidă în baza 16 din baza 2 se face grupând patru biți în baza doi și scriind acești biți în baza 16). Numărul reprezentat pe dublucuvânt va fi reprezentat în memorie în formatul little-endian pe patru octeți astfel:



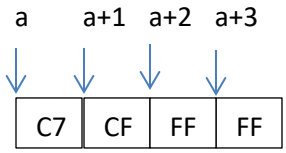
Adresele „a”, „a+1”, „a+2”, „a+3” reprezintă adresa în memorie pentru cei patru octeți ai variabilei „a”, fiecare adresa (offset) reprezintă adresa unui octet. Octetul din memorie de la adresa „a” este cel mai puțin semnificativ octet al numărului, octetul din memorie de la adresa „a+1” este următorul octet cel mai puțin semnificativ și așa mai departe. Se obișnuiește să se reprezinte în hexazecimal un număr din memorie (datorită reprezentării mai scurte – mai puține cifre), de asemenea se știe că patru biți reprezintă o cifră în baza 16. Reprezentarea little-endian de mai sus este:



Dacă ținem cont de următorul exemplu:

```
mov dword [a], -12345
```

prima dată reprezentăm numărul -12345 în reprezentare cu semn ca și 1111 1111 1111 1111 1100 1111 1100 0111, secvență în hexazecimal reprezentată pe 32 de biți este FFFFCFC7h, numărul va fi reprezentat little-endian în memorie astfel:



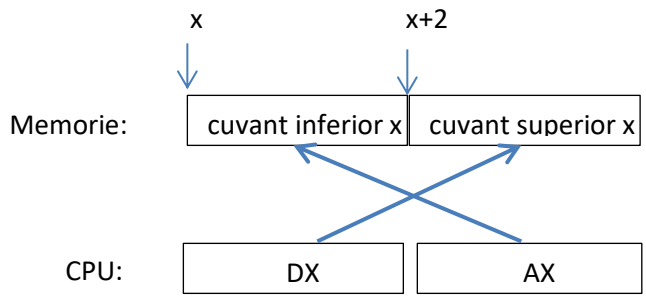
Reprezentarea little-endian este importantă când vrem să facem conversii fără pierderi (nedistructive): se referă cuvântul inferior al unui dublu cuvânt din memorie, se referă octetul superior al unui cuvânt din memorie, etc.

Ex. 1: Scrieți un program care determină valoarea expresiei: $x := a \cdot b + c \cdot d$ unde toate numerele sunt reprezentate fără semn pe un cuvânt.

```
bits 32
global start
extern exit
import exit msvcrt.dll

segment data use32 class=data
    a dw 2
    b dw 5
    c dw 10
    d dw 40
    x dd 0


segment code use32 class=code
start:
    mov ax, [a]
    mul word [b]     ; DX:AX := AX * b = 2*5 = 10
    ; salvam dublucuvantul DX:AX in x
```



```
mov word [x], ax     ; se salveaza AX in cuvantul inferior "x"
                     ; (cuvantul de la adresa "a")
mov word [a+2], dx   ; se salveaza DX in cuvantul superior "x"
                     ;(cuvantul de la adresa "a+2")

mov ax, [c]
mul word [d]         ; DX:AX := AX*d = c*d = 400

; adauga DX:AX to x
add word [x], ax     ; se adauga AX la cuvantul inferior "x"
                     ; in caz de overflow, flagul CF (Carry Flag)
                     ; va avea valoarea 1
                     ; altfel CF = 0
adc word [x+2], dx   ; adauga DX la cuvantul superior "x", se
                     ; adauga si posibilul transportdin  CF

push dword 0
call [exit]
```

Instrucțiuni noi:
ADC – adaugă cu transport
adc d, s => d :=d +s + CF

SBB – scade cu împrumut
sbb d, s => d :=d +s + CF

3.2 Instrucțiuni condiționale de salt

Instrucțiunile condiționale de salt sunt similare cu instrucțiunea „IF” dintr-un limbaj de programare de nivel înalt. În limbajul de asamblare, o instrucțiune „IF” este compusă din două instrucțiuni: o instrucțiune de comparație (pentru a determina relația de ordine între operanzi) și o instrucțiune de salt.

Instrucțiunea de comparație:

```
cmp a, b
```

realizează o scădere nedistructivă `sub a, b` (nu modifică valoarea lui a) și modifică registrul EFLAGS corespunzător rezultatului.
Instrucțiunile condiționale de salt verifică valoarea unor flag-uri și în funcție de valoarea acestora efectuează un „salt în program” la un offset definit de o etichetă (modifică valoarea registrului EIP, EIP nu primește adresa următoarei instrucțiuni imediat după instrucțiunea de salt ci adresa etichetei definită în cod).
Instrucțiuni condiționale de salt care interpretează numerele fără semn:

jb label : (Jump if below) salt la etichetă dacă a<b
jbe label : (Jump if below or equal) salt la etichetă dacă a<=b
jnb label : (Jump if not below) salt la etichetă dacă a>=b
jnbe label : (Jump if not below or equal) salt la etichetă dacă a>b
ja label : (Jump if above) salt la etichetă dacă a>b

jae label : (Jump if above or equal) salt la etichetă dacă a>=b
jna label : (Jump if not above) salt la etichetă dacă a<=b
jnae label : (Jump if not above or equal) salt la etichetă dacă a<b

Valorile *a* și *b* de mai sus reprezintă operanzii instrucțiunii **cmp** care a fost executată înainte de instrucțiunea de salt condiționat.

Instrucțiuni condiționale de salt care interpretează numerele cu semn:

jl label : (Jump if less) salt la etichetă dacă a<b
jle label : (Jump if less or equal) salt la etichetă dacă a<=b
jnl label : (Jump if not less) salt la etichetă dacă a>=b
jnle label : (Jump if not less or equal) salt la etichetă dacă a>b
jg label : (Jump if greater) salt la etichetă dacă a>b
jge label : (Jump if greater or equal) salt la etichetă dacă a>=b
jng label : (Jump if not greater) salt la etichetă dacă a<=b
jnge label : (Jump if not greater or equal) salt la etichetă dacă a<b

Valorile *a* și *b* de mai sus reprezintă operanzii instrucțiunii **cmp** care a fost executată înainte de instrucțiunea de salt condiționat.

Instrucțiuni de salt necondiționate:

jmp label : salt la etichetă

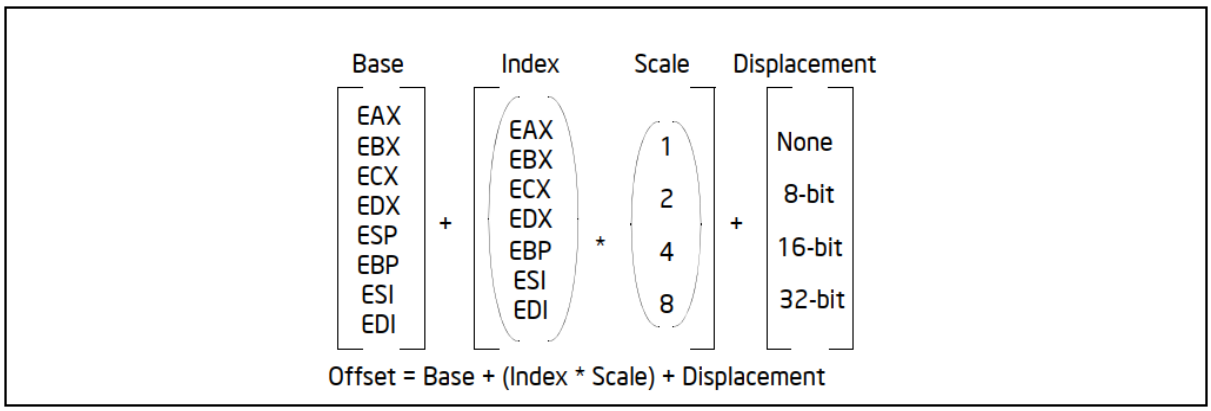
3.3 Șiruri de octeți

Adrese în memorie și offset-uri

Pentru a lucra cu șiruri de octeți/cuvinte/dublucuvânt sunt necesare mai multe cunoștințe legat de modul de adresare în memorie. Până acum am utilizat expresii

```
mov ax, [a]
```

unde *a* este o variabilă și instrucțiunea de mai sus copiază un cuvânt din memorie începând de la adresa „*a*”. Numele unei variabile este doar o adresă constantă – adresa variabilei în memorie. Mai precis, numele unei variabile este un *offset* constant. O specificare completă de adresă se face folosind două numere: ***selector segment*** și ***offset***.
Selectorul de segment este un număr pe 16 biți care specifică un pointer în tabela descriptor de segment (GDT – global descriptor table) care definește o zonă de memorie. Pentru seminarul 3 nu sunt necesare cunoștințe legate de un segment de memorie (mai multe detalii la curs), este suficient să se știe că un segment de memorie este o zonă continua de memorie și adresa acestuia este deja stocată într-un registru segment (CS, DS, ES sau SS) de sistemul de operare înainte de începerea programului scris de voi, valoarea din acești regiștrii nu poate fi modificată. Un ***offset*** este un număr pe 32 de biți care specifică un pointer în interiorul segmentului (segment specificat de selectorul de segment). Numele unei variabile reprezintă offset-ul variabilei (segmentul la care acest offset pointează este segmentul de date, adresa de început a segmentului de date este în DS). Definirea unui offset se face folosind patru valori: bază, index, scala și o constantă (a se vedea figura de mai jos).



În specificarea unui offset se poate utiliza orice combinație din valorile de mai sus („[ ]” valoarea este opțională). Mai jos se dau câteva exemple de specificare de offset (ca și operand sursă pentru instrucțiunea *mov*):

```
mov ax, [a]              ; doar constanta
mov ax, [eax]            ; doar baza sau index
mov ax, [a+eax+ebx]      ; baza, index si constanta
mov ax, [eax+eax+a+2]    ; baza, index si constanta
mov ax, [a+4+ebx*2]      ; index, scala (2) si constanta
mov ax, [eax + ebx*4 + 20]; baza, index, scala si constanta
```

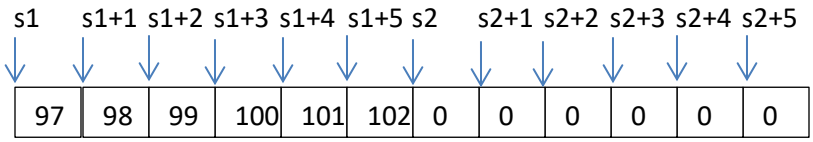
Ex. 2. Se dă un șir de octeți care conține litere mici, să se construiască un șir nou de octeți care să conțină literele din șirul inițial transformate în majuscule.

Varianta 1:

```
bits 32
global start
extern exit
import exit msvcrt.dll

segment data use32 class=data
    s1 db 'abcdef'
    lenS1 equ $-s1
    s2 times lenS1 db 0

; in memorie segmentul de date arata in felul urmator
```




```
; randul de sus din figura reprezinta valori offset. De exemplu, in
; OllyDebugger primul octet din segmentul de date (offset-ul lui s1
; in cazul problemei) este tot timpul 0x00401000.
; Offset-ul variabilei s2 este egal cu offset-ul variabilei s1 plus
; 6 (0x00401006). Randul de jos reprezinta valorile stocate in
; memorie in baza 10 (97 este codul ASCII pentru „a”, ...). Octetii
; din s2 sunt initializati cu 0.
```

```
; lenS1 equ $-s1   defineste o constanta (equ = constanta, nu se
; rezerva memorie)
; $ este contorul de locatii, offset-ul curent in segmentul de date
; (cati octeti sunt de la inceputul segmentului de date pana la linia
; in care apare). Inainte de linia de cod „lenS1 equ $-s1” au fost
; rezervati 6 octeti pentru s1 deci $=s1+6 (s1 este offset-ul
; variabilei). len1 = $ - s1 = s1+6 - s1 = 6 octeti (lungimea in
; octeti a sirului s1).

; s2 times lenS1 db 0   defineste variabila s2 care contine lenS1=6
; octeti, toti initializati cu 0.


segment code use32 class=code
start:
    ; in ESI se tine indexul curent in siruruile s1 si s2
    ; se creaza o bucla cu lenS1=6 iteratii si in fiecare iteratie se
    ; muta octetul s1[ESI] in s2[ESI] dupa ce se modifica in litera
    ; mare.
    ; In aceasta bucla ESI va primi valorile: 0, 1, 2, 3, 4, 5.

    mov esi, 0

    repeat:
        mov al, [s1+esi]       ; AL <- octetul de la offset-ul s1+esi
        sub al, 'a' -'A'       ; se obtine litera majuscula in AL
        mov [s2+esi], al       ; AL -> octetul de la offset s2+esi

        inc esi                ; esi:=esi+1; se trece la urmatorul
                               ; index in sirurile s1 si s2
        cmp esi, lenS1
        jb repeat              ; IF (esi < lenS1) salt la repeat,
                              ; altfel se continua executia


    push dword 0
    call [exit]
```

Varianta 2:

```
bits 32
global start
extern exit
import exit msvcrt.dll

segment data use32 class=data
    s1 db 'abcdef'
    lenS1 equ $-s1
    s2 times lenS1 db 0
```

```
segment code use32 class=code
start:
    ; in aceasta varianta ESI reprezinta offset-ul octetului curent
    ; din sirul s1 si EDI offset-ul curent al octetului din sirul s2.
    ; Bucla va itera de lenS1=6 ori si in fiecare iteratie se muta un
    ; octet de la offset-ul EDI la offset-ul ESI dupa ce se modifica
    ; litera in majuscula. In aceasta bucla ESI va avea valorile s1+0,
    ; s1+1, s1+2, s1+3, s1+4, s1+5 iar EDI va avea valorile s1+6,
    ; s1+7, s1+8, s1+9, s1+10, s1+11.

    mov esi, s1         ; initializare esi
    mov edi, s2         ; initializare edi
    mov ecx, lenS1      ; ecx tine numarul iteratiilor din bucla

    repeat:
        mov al, [esi]     ; AL <- octetul de la offset s1+esi
        sub al, 'a' -'A' ; se obtine litera majuscula in AL
        mov [edi], al     ; AL -> octetul de la offset s2+edi

        inc esi           ; esi:=esi+1; se trece la urmatorul octet
                          ; din sirul s1
        inc edi           ; edi:=edi+1; se trece la urmatorul octet
                          ; din sirul s2
        dec ecx           ; exc:=ecx-1
        cmp ecx, 0        ; IF (ecx > 0) salt la repeat
        jb repeat         ;  altfel se continua executia la push..


    push dword 0
    call [exit]
```