TEORIE - OVERFLOW CONCEPT

Conceptul de overflow este folosit pentru a semnala faptul că rezultatul unei anumite operații nu a încăput în spațiul destinat acesteia. În funcție de ce operație este vorba, setarea CF și OF se va face după anumite reguli, rezultând diferite concluzii legate de UOE.

I Adunarea:

Prima dată vom analiza adunarea în interpretarea fără semn. Aici va fi considerată depășire dacă rezultatul nu e în intervalul absolut (ex: [0,255] pe byte), iar CF va fi setat la 1. În caz contrar, CF va fi 0.

ex: 1. mov al, 100 — CF=1 pt. că 100+200=300>255
mov ah, 200
add al, ah

2. mov al, 100 — CF=1 pt. că -1 trebuie transformat în nr. fără semn = 256-1=
mov ah, -1 = 255
add al, ah
=> 100+255 = 355>255

A doua interpretare de analizat este cea cu semn. În acest caz numerele vor fi transformate în numere cu semn (dacă este cazul), iar o situație de depășire (ex: rez ∉ [-128, 127] pe byte) va avea ca efect setarea OF cu 1, în caz contrar aceasta fiind 0.

ex: 1. mov al, 100 — OF=1 pt. că 100+100=200>127
mov ah, 100
add al, ah

2. mov al, 100 — OF=1 pt. că 156 trebuie transformat în nr. cu semn = 156-256=-100
mov ah, 156
add al, ah
=> 100+(-100) = -200 ∉ [-128, 127]

II Scăderea

În interpretarea fără semn, CF va fi setat dacă există un împrumut de la o poziție care nu există, altfel spus dacă rezultatul nu aparține intervalului de reprezentare. În caz contrar CF=0

ex: 1. mov al,100 CF=1 pt. că 100-101=-1∉[0,255]
mov ah,101
sub al,ah

2. mov al,100 CF=1 pt.că -1 trebuie transformat în nr. fără semn = (-1+256)
mov ah,-1 =255 ⇒ 100-255=-155 ∉ [0,255]
sub al,ah

În interpretarea cu semn, OF va fi setat în unul dintre următoarele 2 cazuri: pozitiv - negativ = negativ sau negativ - pozitiv = pozitiv.

ex.: 1. mov al,100 OF=1 pt. că 100-(-100) = 100+100 = 200
mov ah,-100 aducem în interpretarea cu semn ⇒ 200=200-256=-56
sub al,ah ⇒ poz - neg = neg ⇒ OF=1

2. mov al,100 OF=1 pt.că trebuie să aducem 156 în interpretarea cu semn
mov ah,156 ⇒ 156=156-256=-100
sub al,ah ⇒ 100-(-100)=200
transf 200 în interpretarea cu semn ⇒ 200=200-256=-56
⇒ poz - neg = neg ⇒ OF=1

III Înmulțirea

La înmulțire OF și CF semnalează depășire în mod diferit. S-a hotărât ca depășirea să însemne faptul că rezultatul nu a încăput în același interval de reprezentare cu operanzii (ex byte * byte chiar a generat un rez. care începe doar pe word)

ex: 1. mov al,20 CF=OF=1 pt. că 20*20=400 care nu începe pe byte
mov ah,20
mul ah

2. mov al, 3 CF=OF=0 pt. că 3*6=18 încape pe un byte, nu are nevoie
mov ah, 6 de spațiu adițional
imul ah

IV Împărțirea

Depășirea la împărțire constituie un caz special deoarece, dacă se produce, programul se oprește (se division overflow → runtime error). În acest caz, valorile din CF și OF sunt irelevante. Depășirea înseamnă că rezultatul nu a încăput în spațiul destinat pt. acesta.

ex: mov ax, 4096 division overflow pt. că în dl ar trebui să fie
mov bl, 10 4096 : 10 = 409 rest 6, iar 409 nu încape pe un
div bl ↓ al ↓ ah
byte → programul se oprește

Există mai multe metode prin care se poate ține cont de aceste depășiri. Asamblorul ne oferă 2 instrucțiuni specifice pt. adunare și scădere: ADC (add with carry) respectiv SBB (scădere cu carry) în care ține cont de transportul existent în flag-uri.

De obicei nu se ține cont de carry, dar atunci când avem un nr. salvat de exemplu în DX:AX și altul în CX:BX, dacă vrem să le adunăm vom proceda astfel: add ax, bx pentru a obține un rezultat corect (altfel spus, atunci când
adc dx, cx

noi producem transportul, ținem cont de el)

9 feb. 2018

2. a1 db '256, -256'

* le consideră caractere pe fiecare, un caracter ocupă un byte

=> în memorie va fi '2'|'5'|'6'|','|'-'|'2'|'5'|'6'

a2 dw 256, 256h

* 256 trebuie transformat în baza 16

$256 = 2^8 = 0000\ 0001\ 0000\ 0000 = 0100h$ => în mem. va fi 00/01

* 256h în mem. va fi 56/02

! Se respectă little endian

a3 dw $-a2 dar dacă ar fi fost a3 dw 20h, $-a2 ? => ar fi fost la fel, $ e începutul liniei aparent

* $-a2 = 4 (au fost declarate 2 words = 4 bytes între locația curentă și a2)

=> în mem. va fi 04/00

a4 equ -256/4

* ! nu ocupă memorie

a5 db 128>>1, -128<<1

* >> este de fapt o împărțire cu $2^{nr.}$ care se shiftează

=> $128>>1 = 128 : 2^1 = 64 = 2^6 = 0100\ 0000 = 40h$ => în mem va fi 40

* << este o înmulțire cu $2^{nr.}$ cu care se shiftează

=> $-128<<1 = -128 * 2^1 = -256$ nu încape pe byte => $-256 = -256 + 256 = 0$

=> în mem. va fi 00

a6 dw a2-a5, ~(a2-a5)

* a2-a5 = -6 (între a2 și a5 sunt 6 bytes)

în mem va fi FA/FF (pt. că avem word)

$|-6| = 6 = 0110$ => C2(6) = 1111 1010 = FA => pe word va fi FA FF

* ~(a2-a5) = ~ 1111 1111 1111 1010 = 0000 0000 0000 0101 = 0005 *

-> în memorie va fi 05|00

a7 dd [a2], ba2

! a2 nu e o valoare determinabilă la momentul asamblării
nu se pot efectua operații pe biți decât cu valori scalare

a8 dd 256h ^ 256, 256256h

* 256h ^ 256

256h = 0010 0101 0110
256 = 0001 0000 0000
=> 256h ^ 256 = 0011 0101 0110 = 356h => în mem. pe dword va fi 56|03|00|00

* 256256h în mem va fi 56|62|25|00

a9 dd ($-a8) + (a10-$)

* $-a8 = 8 (8 bytes între locația curentă și a8)
a10-$ = 4 (4 bytes între locația curentă și a10)
=> ($-a8) + a10-$ = 12 = 0Ch => în mem. pe dword va fi 0c|00|00|00

a10 dw -255, 256

* -255

|-255| = 255 = 1111 1111 => C2(255) = 1111 1111 0000 0001 pe word = FF01h

255 |2
1 127 |2
1 63 |2
1 31 |2
1 15 |2
1 7 |2
1 3 |2
1 1 |2
1 0

=> 255 = 11111111b

=> în mem. va fi 01|FF

* $256 = 2^8 = 1\ 0000\ 0000 = 0100h$ → în mem. va fi 00|01

a11 resb 6

* 00|00|00|00|00|00 → rezervă 6 bytes goi

a12 times 4 dw 256

* 256 = 00|01 în memorie și îl pune de 4 ori (ca și word)
  → 00|01|00|01|00|01|00|01

a13 dw times 4 -128

! syntax error: trebuia să fie a13 times 4 dw -128

times 2 resw 2

* rezervă de 2 ori câte 2 words
  → în mem va fi 00|00|00|00|00|00|00|00

times 2 dd 12345678h

* pune în memorie 78|56|34|12|78|56|34|12 (de 2 ori nr. cu little endian)

3. a) 1. lea ebx, [ebx+6] și mută rez în ebx
   adună la ebx 6 → categoria I.1

   2. lea ebx, [bx+6] → categoria I.3 ← ca și I.1, dar adună 6 la bx
   3. lea bx, [bx+6] → categoria I.2
   4. lea bx, [ebx+6] → categoria I.2
   → adună la bx 6 și mută în bx → categorie I.2

   5. mov ebx, ebx+6
   syntax error (acest tip de adunare este valid doar în calcul de offset → categoria II

   6. mov ebx, [ebx+6] → categoria III
   încearcă să pună în ebx val. de la adresa [ebx+6], dacă e validă

   7. mov2x ebx, [ebx+6]
   → syntax error, trebuia precizată dimensiunea pt [ebx+6] → categoria II

   8. mov2x ebx, [bx+6]
   → syntax error, trebuia precizată dimensiunea pt [bx+6] → categoria II

   9. add bx, 6
   adaugă 6 la bx și îl mută în bx rezultatul → categoria I.2 apoi avem; cerva
   ⇒ nu face nimic

10. mov [ebx], dword [bx+6]

Syntax error, nu pot fi ambii operanzi din memorie → categoria II

11. add ebx, 6

adună 6 la ebx și pune rez în ebx → categoria I.1

12. add bx, 6

adună 6 la bx și pune rez în bx → categoria I.2

13. push [ebx+6]

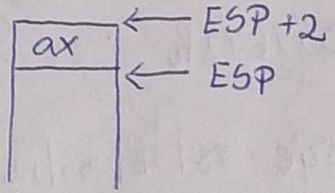
syntax error → trebuia specificată dimensiunea (word sau dword)

→ categoria II

14. xchg ebx, [ebx+6]

pune în ebx valoarea de la [ebx+6] dacă e validă → categoria III

b) cld → direcția de la st. la dr.



push ax <=> la esp+2 va fi ax

mov ax, esp+2 <=> mov ax, ax

stosw ; la de pe dină un word și pune la adresa EDI

→ [EDI] = ax

EDI ← EDI+2

lea EDI, [EDI-2] ; scade 2 din EDI

add ESP, 2 <=> restaurează val din ESP

→ echivalent cu mov [EDI], ax (doar asta s-a schimbat, toate celelalte valori au fost restaurate