TEORIE

1. adresă de memorie = identificator unic al poziției unei locații de memorie pe care procesorul o poate accesa pentru citire sau scriere
   ex.: pt. memoria flat primul elem. din memorie va avea adresa 32 de 0 (pe 32 de biți)
2. segment de memorie = diviziune logică a memoriei, o succesiune de locații de memorie menite să deservească scopuri similare
   ex.: code segment conține instrucțiuni mașină (de la 1 la 15 octeți)
3. offset = nr. de octeți adăugați la o adresă de bază (nr. de octeți de la adresa de început de segment până la locația căreia îi vrem offset-ul)
   ex.: în data segment a db 1 → a are offset 0 (raportat la $$)
   b db 2 → b are offset 1 (raportat la $$)
4. specificare de adresă = (adresă FAR) formată din segment și offset
   ex.: mov EAX, [DS:a] → în EAX pune adresa FAR a lui a
5. mecanism de segmentare = proces de împărțire a memoriei în diviziuni logice, menite să deservească același scop
   ex: data segment, code segment, stack segment, extra segment
6. adresă liniară = (adresă de segmentare) formată din bază + offset
   $a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0 = b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 + 0_7 0_6 0_5 0_4 0_3 0_2 0_1 0_0$
   ex: avem baza = 2000h și offset = 1000h ⇒ adresa liniară = 3000h
7. model de memorie flat = toate segmentele încep de la 0 (bloc continuu)
   ex.: modul protejat x86 folosește modelul de memorie flat
8. adresă fizică efectivă = rezultatul final al segmentării (+ eventual paginării) în memoria fizică (hardware)
9. adresare directă: implică doar operanzi direcți și imediați
   ex.: mov eax, [a+4]
10. adresare bazată: intervin regiștrii de bază
    ex.: mov eax, [ebx]

11. adresare indexată : intervin registrii index (și implicit scala)

ex.: mov eax, [2·eax]

12. adresare indirectă: care nu e directă

ex: mov ax, [ebx+V+4]

13. adresă NEAR: formată doar din offset, segmentul se adaugă implicit în loading time

ex: mov eax, [v]

14. reguli implicite de asociere între offset și registru segment:

CS : pt. jmp, call, ret — ex.: jmp acolo

DS : restul — ex.: mov ax, [EBP+ECX+4]

SS: dacă avem ESP sau EBP bază — ex.: mov ax, [ESP]

31 ian. 2020

2. x dw -256, 256h

* -256

|-256| = 256 = 1 0000 0000

C2(256) = 1111 1111 0000 0000 = FF00 ⇒ în mem. va fi 00/FF

* 256h în mem. va fi 56/02

y dw 256|-256, 256h & 256

* 256|-256

256 = 0000 0001 0000 0000
-256 = 1111 1111 0000 0000 } ⇒ 256|-256 = 1111 1111 0000 0000 = FF00
⇒ în mem. va fi 00/FF

* 256h & 256

256h = 0000 0010 0101 0110
256 = 0000 0001 0000 0000 } ⇒ 256h & 256 = 0000 0000 0000 0000
⇒ în mem. va fi 00/00

z db $-z, y-x
db 'y'-'x', 'y-x'

* $-z = 0 ⇒ în mem. va fi 00
* y-x = 4 (4 bytes între x și y) ⇒ în mem va fi 04
* 'y'-'x' face scădere de coduri ASCII ⇒ 'y'-'x' = 1 ⇒ în mem. va fi 01
* 'y-x' le va lua ca și caractere separate ⇒ în mem va fi 'y''-''x'

a db 512 >> 2, -512 << 2

* >> înseamnă împărțire cu 2^ nr. cu care se shiftează
⇒ 512 >> 2 = 512 : 2^2 = 512 : 4 = 128 = 2^7 = 1000 0000b = 80h ⇒ în mem. 80
* -512 << 2
<< înseamnă înmulțire cu 2^ nr. cu care se shiftează
⇒ -512 << 2 = $-2^9 \cdot 2^4 = -2^{13}$

$|-2^{13}| = 2^{13}$. 1000000000000

$C2(2^{13})$ = 1110 0000 0000 0000 => pe un byte va fi 00

b dw 2-a, !0(2-a)

* 2-a = -6 (6 bytes între 2 și a)

=> |-6| = 6 = 0110

C2(6) = 1111 1010 = FA => în mem. va fi FA/FF (suntem pe word, nu pe byte)

* !0(2-a) = !0(-6) = 0 => în mem. va fi 00/00

c dd ($-b) + (d-$), $-2*y+3

* ($-b) + (d-$) = 4 + 4 (pt. că $-2*y+3 e syntax error și nu îl luăm în calcul)

= 8 => în mem va fi 08/00/00/00

* înmulțirea de pointeri nu e validă! Syntax error

d: db -128, 128 | (~128)

* -128 = -128+256 = 128 = 2^7 = 1000 0000 = 80 în mem. va fi 80

* 128 ^ (~128) = 1111 1111 = FF în mem. va fi FF

↓

nr ⊕ ~nr

e times 2 resw 6

* rezervă de 2 ori câte 6 words

=> în memorie va fi 00/00/00/00/00/00/00/00/00/00/00/00/00/00/00/00/00/00/00/00/00/00/00/00

* times 2 dd 1234h, 5678h

pune de 2 ori în mem. cele 2 numere, respectând little endian => în mem. va fi

34/12/00/00/78/56/00/00/34/12/00/00/78/56/00/00

a)

1) lea eax, [6+ESP]

mută în eax valoarea ESP+6 (lea scapă de parantezele pătrate)

2) mov eax, 6+ESP

syntax error → esp nu e o valoare determinabilă la momentul asamblării

3) movsx ax, [6+esp]

pune în ax cu sign extended un byte de la adresa [6+esp]

merge și fără specificare dimensiune pt. că doar byte poate fi pus acolo

4) mov ebp, [6+ebp*2]

pune în ebp valoarea de la adresa [6+ebp*2] (un dword de acolo)

5) mov [6+ebp*2], 12

! syntax error → trebuia specificată dimensiunea măcar la unul dintre operanzi

6) mov ebp, [ebx+esp]

esp nu poate fi index => ia esp ca bază și ebx ca index

pune în ebp un dword de la adresa [ebx+esp]

7) movsx [6+esp], eax

pune la adresa [6+esp] pe eax (valoarea din eax)

8) mov [6+esp*2], eax

! esp nu poate fi index => syntax error (invalid effective address)

9) mov [6+ebp*2], [6+esp]

! syntax error → nu pot fi ambii operanzi expliciți din memorie

10) movzx eax, [6+ebp*2]

! syntax error → trebuia specificată dimensiunea operandului din dreapta pt. că acolo poate fi byte sau word

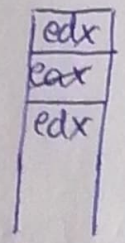
b) push ed ⇔ pune edx pe stivă

push eax ⇔ pune eax pe stivă

pop edx ⇔ pune în edx pe eax

Scoate eax de pe stivă

edx = eax



xor dh, dh ⇔ dh = 0

shl edx, 16 ⇔ edx era |_|_|0|dl| apoi 16 la stânga ⇒ edx = |0|dl|0|0|

clc ⇔ pune în carry flag 0 ⇒ CF = 0

rcr edx, 16 ⇔ rotește cu carry 16 la dreapta ⇒ edx = |0|0|0|dl|

add edx, ebx → edx va fi practic ebx+dl

push edx ⇒ pune edx pe stivă

pop esi ⇒ în esi va fi practic ebx+al

lodsb ⇒ pune în al un byte de la [ESI] adică de la [EBX+al] ⇒ (XLAT)

pop edx ⇒ restaurează valoarea din edx

⇒ echivalent XLAT