În cadrul unei instrucţiuni există 3 moduri de a specifica un operand pe care aceasta îl solicită:

* *modul registru*, dacă pe post de operand se află un registru al maşinii; mov ax, bx



* *modul imediat*, atunci când în instrucţiune se află chiar valoarea operandului (nu adresa lui şi nici un registru în care să fie conţinut); mov eax,2
* *modul adresare la memorie*, dacă operandul se află efectiv undeva în memorie. În acest caz, adresa offsetului lui se calculează după următoarea formulă:

***adresa\_offset* = [ bază ] + [ index × scală ] + [ constanta]**

**(SIB) (+deplasament + imediat)**

constanta = offset constant sau/si val.imediata

mov eax, [v] – asta unde se incadreaza ????? La Mod adresare la memorie !!!

[EBX+ECX\*2 + v -7] – ok

SIB depl. const.

V db 17

add edx, [EBX+ECX\*2 + v -7] – OK !!!!

mov ebx, [EBX+ECX\*2 - v-7] – Syntax error !!!! – invalid effective address – impossible segment base multiplier

adc ecx, [EBX+ECX\*2 + a+b-7] – Syntax error din cauza “a+b” ; invalid effective address – impossible segment base multiplier

sub [EBX+ECX\*2 + a-b-7], eax – ok, pt ca a-b este o operatie corecta cu pointeri !!!

SIB const.

Deci *adresa\_offset* se obţine din următoarele (maxim) patru elemente:

* conţinutul unuia dintre regiştrii EAX, EBX, ECX, EDX, EBP, ESI, EDI sau ESP ca bază;
* conţinutul unuia dintre regiştrii EAX, EBX, ECX, EDX, EBP, ESI sau EDI drept index;
* factor numeric (scală) pentru a înmulţi valoarea registrului index cu 1, 2, 4 sau 8
* valoarea unei constante numerice, pe octet sau dublucuvânt.

De aici rezultă următoarele moduri de adresare la memorie:

* *directă*, atunci când apare numai *constanta*;
* *bazată*, dacă în calcul apare unul dintre regiştrii bază;
* *scalat-indexată*, dacă în calcul apare unul dintre regiştrii index;

Adresarea care NU este directa se numeste ***adresare indirecta*** (bazata si/sau indexata). Deci o adresare indirecta este cea pt care avem specificat cel putin un registru intre parantezele drepte.

In cadrul sistemului de adresare se efectueaza operatii cu adrese (pointeri). Care sunt operatiile ARITMETICE cu pointeri permise IN INFORMATICA ?...

**Raspuns**: Orice operatie care are sens... aceasta insemnand orice operatie ce exprima ca rezultat o localizare in memorie corecta si utila ca informatie pt programator.

Aritmetica de pointeri/adrese (pointer arithmetic – operatii cu pointeri) = utilizarea de expresii aritmetice, care au ca operanzi adrese !

* adunari si scaderi de adrese
* adunari si scaderi de constante la o/dintr-o adresa = necesare si utile pt accesarea elementelor dintr-un array
* inmultirea a doua adrese ?- nepermisa (in majoritatea cazurilor valoarea obtinuta este dincolo de limita maxima a memoriei posibile a fi accesata).
* Inmultirea cu o constanta (in majoritatea cazurilor valoarea obtinuta este dincolo de limita maxima a memoriei posibile a fi accesata). In plus, CE reprezinta valoarea obtinuta ?... Nimic util !!
* Impartire ?... No way !
* Scadere de adrese = ??? CE reprezinta ? q-p = nr octeti dintre cele doua adrese de memorie (niciodata nu depasim dim memoriei ; valoarea obtinuta este o CONSTANTA NUMERICA !!!! )
* Adunare de adrese = ??? CE reprezinta ? NIMIC !!!!!

Pointer arithmetic...? Cuprinde 3 operatii:

1). Scaderea a doua adrese

Adresa – adresa = ok (q-p = scadere de 2 pointeri = sizeof(array) sau nr de elemente (in C)/octeti (asamblare) dintre doua adrese de memorie)

Adresa – offset = adresa – adresa = ok

2) Adunarea unei constante numerice la o adresa

Adresa + constanta numerica (identificarea unui element prin indexare – a[7]) , q+9

3). Scaderea unei constante numerice dintr-o adresa

Adresa – constanta numerica - a[-4] , p-7

* scaderea a 2 pointeri – valoare SCALARA !!! (valoare numerica constanta imediata)
* adunarea unei constante la un pointer à POINTER !
* scaderea unei constante dintr-un pointer à POINTER !

(ultimele doua utile pt referirea de elemente dintr-un array/zona de memorie)

ADUNAREA A DOI POINTERI NU ESTE PERMISA !!!!!!

p+q = ???? (allowed in NASM...sometimes...) – dar nu inseamna ADUNARE DE POINTERI in cele din urma asa cum vom vedea...

* Singura exceptie de la regulile aritmeticii de pointeri o constituie formula de calcul a offsetului unui operand unde sunt permise adunari de valori de registri... In rest nu exista exceptii (de exemplu la nivel de adrese de variabile cu care se opereaza intr-un program).

Atribuire: i:=i+1 Adresa lui I ß valoarea lui I + 1

LHS (valoare stanga a unei atribuiri L-value = adresa) := RHS (valoarea dreapta a unei atribuiri = R-Value = CONTINUT !!)

Symbol := expression\_value (identificator := expresie)



Expresie\_calcul\_de\_adresa := expresie (forma cea mai generala)

Dereferentierea (extragerea valorii de la o adresa) este implicita in 99% din limbaje. Exemplu exceptie – limbajul BLISS – unde dereferentierea trebuie mentionata explicit intotdeauna; iß\*i+1

(+ unele situatii in Algol 68)

Symbol := expression\_value (99% of the cases…)

*Address\_computation\_Expression* := expression\_value (forma cea mai generala)

In C++ f(a+3, b-2, 2) = x+y+z

Int& f(x,i) {….return v[i];} – Functia f returneaza o LHS (valoare stanga)

F(a,7) = 79; inseamna ca v[7]=79 !!!

Int& j = i; // j devine ALIAS pt i

(a+2?b:c) = x+y+z ; - correct

(a+2?1:c) = x+y+z; - syntax error !!! 1:=n !!!!

Exemple/studii de caz

Mov ax, ebx - reg16 ß reg32 – Sintax error ! invalid combination of opcode and operands

Mov ebx, ch - Sintax error ! invalid combination of opcode and operands

Mov op\_size\_dest, op\_SAME\_size b,b w,w dw,dw

Mov eax, ebx - eaxßEBX

Daca avem incepand de la adresa [EBX] valorile 01h 02h 03h 04h

Mov eax, [ebx] - eax ß [val DIN MEMORIE preluata de la adresa indicata de continutul reg. EBX]; ; eax ß 4 octeti de la adresa DS:[EBX]; EAX:= 04030201h

Mov ax, [ebx] = mov ax, [ds:ebx] ; ax = valoarea cuvantului din memorie ce incepe la adresa DS:EBX; ax ß 2 octeti de la adresa DS:[EBX]; AX:= 0201h

Mov edx, [eax+ebx] edx ß [valoarea DIN MEMORIE preluata de la adresa indicata de rezultatul adunarii continuturilor registrilor EAX si EBX]

Mov edx, eax+ebx ; syntax error ! A se intelege diferenta intre OPERATORUL + si INSTRUCTIUNEA ADD ! Operatorii pot efectua doar calcule cu valori constante determinabile la momentul asamblarii !!

Add ebx, ecx ; ok

Mov eax, ebx+ecx - syntax error !

Mov edx, eax+v !!!! – syntax error !!

Mov edx, [ebx+eax] – corecta , face acelasi lucru ca mai sus; NU STIM CARE va fi pt processor BAZA si care INDEXUL !!

Mov edx, [esp+ecx] ; ok – aici STIM ca ESP e baza , deoarece ESP poate fi DOAR BAZA !!!!

Mov edx, [ecx+esp]; ok – si aici STIM ca ESP e baza , deoarece ESP poate fi DOAR BAZA !!!!

Mov edx, [esp+2\*ecx] – correct…ESP – baza; ECX – index ; 2 = scala

Mov edx, [ecx+2\*esp] – syntax error !!!! pt ca ESP NU poate fi index

mov dh, [edx + ecx \* 4 + 3] ; CORECTA ; DHßPRIMUL OCTET de la offsetul desemnat de formula de calcul edx + ecx \* 4 + 3

mov dx, [edx + ecx \* 4 + 3] ; CORECTA ; DXßPRIMUL CUVANT (WORD) de la offsetul desemnat de formula de calcul edx + ecx \* 4 + 3

mov eax, [eax\*3] = mov eax, [eax+2\*eax] – CORECT !!!!!!!

mov eax, [ebx\*9 + 12] - EBX + 8\*EBX + 12 – CORECT !!!!!

mov eax, [esp\*5] – syntax error – ESP NU POATE FI INDEX !!!!

Mov reg, [var] – in care dintre modurile de specificare a operanzilor se incadreaza ?

Mov eax, [a] - corecta!!! Pt ca OFFSETUL ORICAREI VARIABILE DEFINITE IN PROGRAM ESTE O VALOARE NUMERICA CONSTANTA DETERMINABILA LA MOMENTUL ASAMBLARII !!!!

Mov eax, [a] - ??? ce valoare are a vs. [a] ??

a = adresa sa ! Notatia [a] inseamna CONTINUTUL de la adresa a

[] = operatorul de dereferentiere (in limbaj de asamblare, dar si in C !!)

Memoria se poate accesa in 2 moduri:

* utilizand nume de variabile (mov eax, [a]) – adresare directa ! (displacement – only !)
* prin calcul de adresa utilizand formula de calcul al offset-ului unui operand (mov eax, [ebx + 2 \* ecx-7] in asamblare sau var1=\*(p-8) in C)

Mov eax, dword ptr [DS:2] – vezi OllYDBG !!!!

Var d? ….

Mov eax, var ; ok – eax = OFFSET-ul variabilei var (32 biti)

Mov eax, [var] ; ok – eax = CONTINUTUL de la adresa var – 4 octeti

In TASM si MASM – la definirea unei variabile I se asociaza un TD ferm (db, dw, dd) si atunci:

Mov eax, var ; e syntax error daca var e diferit de dublucuvant in MASM si TASM unde NU exista operator de dereferentiere [] ; functioneaza doar daca var DD … si se transfera continutul de la adresa var;

Daca vrem in TASM /MASM sa transferam adresa trebuie folosit operatorul OFFSET:

Mov eax, OFFSET var – transfera offset-ul in EAX in TASM/MASM

Mov eax, var – transfera CONTINUTUL in EAX in TASM/MASM

Mov eax, var ; EAX ß offsetul (var) = pe 32 biti

Mov eax, [var] ; EAX ß se iau 4 octeti de la adresa var (indiferent de cum am definit var) si se transfera in EAX !

Mov ax, var ; va merge cu Warning (16 bits reloc of 32 bits value)

* se vor prelua cei doi octeti INFERIORI din reprezentarea IN MEMORIE a offset-ului

Mov ax, [var] ; AX ß se iau 2 octeti de la adresa var (indiferent de cum am definit var) si se transfera in AX !

Mov ah, var ; Syntax error ! OBJ file can only handle 16 – or 32 bits addresses

Mov ah, [var] ; AH ß se ia 1 octet de la adresa var (indiferent de cum am definit var) si se transfera in AH !

Var db 17, 18, 19, 29, 2ah, -3

Mov [var], eax; mov dword ptr [var], eax - - continutul lui eax va suprascrie primii 4 octeti de la var

A db 17

B db 19

C db 21

D db 23

Mov eax, [A]; ia 4 octeti de la adresa A si ii transfera in EAX !!!

Mov eax, [A]; 4 octeti luati in ordinea alocarii (17, 19, 21, 23) si transferarea lor in EAX

Mov eax, [B-1] = mov eax, [C-2] = mov eax, [D-3] = mov eax, [A]

Mov ah, ebx ; syntax error – invalid combination of opcode and operands

Mov ah, [ebx] ; se ia 1 octet de la [ebx] si se transfera in ah

Mov ax, [ebx]; 2 octeti

Mov eax, [ebx] ; 4 octeti

Offset\_spec16 = [BX|BP] + [SI|DI] + [constanta] (formula de calcul al offset-ului unui operand sub 16 biti)

Mov ah, [bx] ; 1 octet in AH luat pe baza formulei de calcul pe 16 biti

Mov ax, [bx] ; ia 2 octeti de la DS:[BX] si ii transfera in AX

Mov eax,[bx] ; 4 octeti in EAX

Mov ah, [bh] ; syntax error !!! – invalid effective address (syntax error deoarece BH nu poate fi registru de baza !!!)

Au variabilele NASM tip de data asociat ? NU !

a db 17,19

b dw 1234h ; 34h 12h

c dd….

Rolul directivelor de definire a datelor NU este in NASM de a preciza tipul de data al variabilelor definite, ci DOAR de a genera octetii corespunzatori acelor zone de memorie pe care le ocupa in conformitate cu directiva specifica aleasa si respectand ordinea de plasare de tip little-endian !!!

Deci a NU este de tip byte – ci doar un offset/deplasament si atat… un simbol desemnand inceputul unei zone de memorie FARA VREUN TIP ASOCIAT !

Deci b NU este de tip word – ci doar un offset/deplasament si atat… un simbol desemnand inceputul unei zone de memorie FARA VREUN TIP ASOCIAT !

Deci c NU este de tip dword – ci doar un offset/deplasament si atat… un simbol desemnand inceputul unei zone de memorie FARA VREUN TIP ASOCIAT !

Atunci DE CE mai asociem in definitie o directiva de tip de data ???? Pt a sti CUM sa populam cu date/initializam zona de memorie respectiva !!!

Spre deosebire de majoritatea limbajelor de nivel înalt care accesează memoria exclusiv pe baza numelui de variabile , în limbaj de asamblare memoria poate fi accesată **DOAR prin intermediul formulei de calcul a offset-ului unui operand** unde este utilizata aritmetica de pointeri. Aritmetica de pointeri este un mecanism ce poate fi utilizat si in C !

Numele unei variabile este OFFSET-ul său raportat la segmentul în care apare definiția sa! **Offset-urile variabilelor dintr-un program sunt întotdeauna valori constante determinabile la momentul asamblării /compilării**.

mov ax, [ebx] - Având în vedere că operandul sursă nu are un tip asociat, operandul destinație este cel care hotărăște tipul de dată al transferului efectuat, iar transferul se va face ținând cont de reprezentarea Little Endian din memorie.

Operatorii pot efectua doar calcule cu valori constante determinabile la momentul asamblării!! Singura excepție o constituie formula de la 2 noaptea, a calculului OFFSET-ului unui operand! (acolo apare operatorul + care opereaza cu continuturi de registri !!)

***Assembly language*** and ***C*** are ***value oriented languages***, meaning that everything is reduced in the end to a numeric value (vezi tipul Boolean de exemplu in C – similar in Assembly). This is a low level feature.