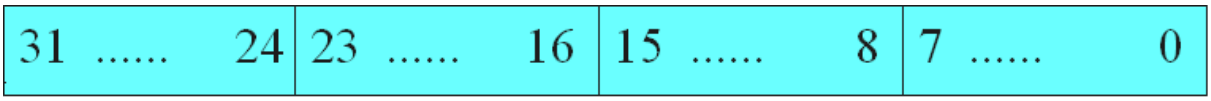
registri (inclusiv indicatorii de conditii)
* introducere în Visual Studio
* instrucțiuni simple (mov, add, sub, inc, dec)
* instrucțiuni logice: not, or, and, xor, test, shl, shr, sar ror, rol, înmulțire, împărțire

Octetul (**BYTE**): bitul cu indexul 7 este cel mai semnificativ, cel cu indexul 0 este cel mai putin semnificativ. Folosit pentru reprezentarea numerelor intregi cu sau fara semn (char/unsigned char)

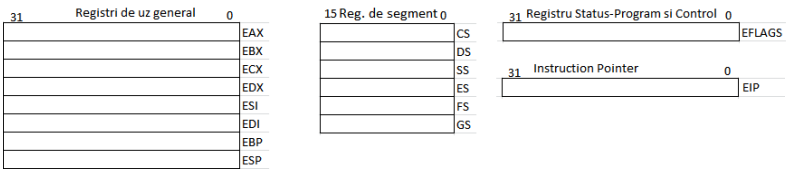
Cuvantul de dimensiune 2 (**WORD**): octetul 15-8 este cel superior; octetul 7-0 este cel inferior. Folosit pentru reprezentarea numerelor intregi cu sau fara semn (short/unsigned short)

Cuvantul de dimensiune 4: (**DWORD**): octetul 31-24 este cel superior; octetul 7-0 este cel inferior. Folosit pentru reprezentarea numerelor intregi cu sau fara semn (int/unsigned int), a adreselor de memorie (pointeri)



Bit and Byte Order. Procesoarele IA-32 sunt masini de tipul little-endian: octetii dintr-un WORD/DWORD sunt pusi dinspre adresa mai mare de memorie catre adresa mai mica de memorie.

Tipuri de registri:
Registri de uz general. Acesti 8 registri sunt folositi pentru stocarea operanzilor si a pointerilor.
Registri de segment. Acesti registri stocheaza 6 selectori de segment.
EFLAGS(Status-Program si Control). Acest registru retine statusul programului in curs de executie si permite controlul limitat al procesorului.
EIP(intruction Pointer). Contine un pointer pe 32 de biti catre urmatoarea instructiune de executat.



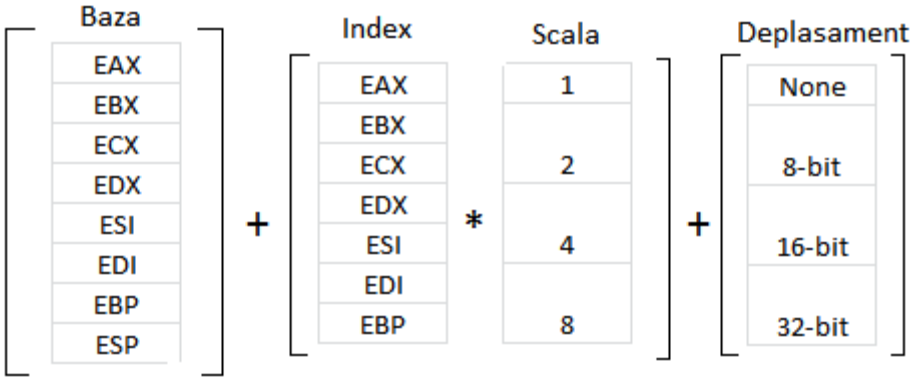
Cei mai putin semnificativi 16 biti din cadrul registrilor de uz general sunt asociati direct cu registrii generali din cadrul procesoarelor 8086 si Intel 286, si pot fi referentiati prin registrii: AX, BX, CX, DX, BP, SP, SI, DI, SP. Fiecare din cei mai putin semnificativi doi octeti din cadrul registrilor EAX, EBX,ECX,EDX pot fi accesati prin numele: AH, BH, CH, DH (cei mai semnificativi) si respectiv AL, BL, CL, DL (cei mai putin semnificativi)

31 16	15 8	7 0	
	AH	AL	EAX
	BH	BL	EBX
	CH	CL	ECX
	DH	DL	EDX
	SI		ESI
	DI		EDI
	BP		EBP
	SP		ESP

Registrul pe 32 de biti EFLAGS contine mai multi marcatori de tip "Status Flag":
* **CF** (bit 0): Carry Flag este setat doar atunci cand o operatie aritmetica genereaza un transport sau se foloseste de un imprumut pe cel mai semnificativ bit. Se poate folosi la testarea depasirii in cadrul aritmeticii cu numere intregi fara semn.
* **PF** (bitul 2): Parity Flag este setat doar atunci cand cel mai putin semnificativ octet din cadrul rezultatului are un numar par de biti de 1.
* **ZF** (bitul 6): Zero flag este setat doar atunci cand rezultatul este zero.
* **SF** (bitul 7): Sign flag are valoarea celui mai semnificativ bit din cadrul rezultatului, adica este semnul al unui numar intreg cu semn (0=pozitiv,1=negativ).
* **OF** (bitul 11): Overflow flag este setat doar atunci cand apare o eroare de depasire in cadrul aritmeticii cu numere intregi cu semn.

Modul de adresare a operanzilor.
Instructiunile masina IA-32 se bazeaza pe zero sau mai multi operanzi. Unii operanzi sunt specificati in mod explicit, altii in mod implicit.
Datele pentru un operand de tip sursa pot fi localizate intr-un registru, intr-o locatie de memorie sau in cadrul instructiunii respective (un operand direct ~ immediate operand).
Datele pentru un operand de tip destinatie pot fi localizate intr-un registru sau intr-o locatie de memorie.

Adresarea memoriei.
Offsetul din cadrul unei adrese de memorie poate fi specificat direct ca o valoare statica (deplasament) sau prin intermediul unui calcul ce poate contine cel putin una din urmatoarele componente: **desplasament** (8/16/32bit)**, baza** (valoarea din cadrului unui registru general), **index** (valoarea din cadrului unui registru general), **factor de scalare** (valoarea 2, 4, sau 8).
Offsetul care rezulta din aceste elemente se numeste adresa efectiva. Offset = Baza+(Index*Factor) + Deplasament



Instructiuni:
1. Moving data: **MOV** dest, source // dest <- source.
2. Exchange: **XCHG** dest, source // temp <- dest; dest <- source; source <- temp.

3. Integer add: **ADD** dest, source// dest <- (dest+source).
4. Substract: **SUB** dest, source// dest <- (dest-source).
5. Increment: **INC** dest //dest <- (dest + 1).
6. Decrement: **DEC** dest //dest <- (dest - 1).
7. Negate: **NEG** dest //dest <- (-dest).
8. Bitwise logical not: **NOT** dest //dest <- C1(dest).
9. Bitwise logical and: **AND** dest, source //dest <- (dest & source).
10. Bitwise logical or: **OR** dest, source //dest <- (dest | source).
11. Bitwise logical xor: **XOR** dest, source //dest <- (dest ^ source).
12. Logical compare: **TEST** operand1, operand2 //Computes the Bitwise logical AND between the two operands and sets the SF, ZF and PF. The result is then discarded.

Shift Instructions: The last bit shifted beyond the destination boundary are shifted into the cary flag, then discarded.
The *count* operand can be the **CL** register or an immediate value.
There is no difference between SAL and SHL

13. Shift arithm. right: **SAR** signed_dest, count // signed_dest <- (signed_dest >> (count%32)).
14. Shift logic right: **SHR** unsigned_dest, count // unsigned_dest <- (unsigned_dest >> (count%32)).
15. Shift arithm. left: **SAL** dest, count // dest <- (dest << (count%32)).
16. Shift logic. left: **SHL** dest, count // dest <- (dest << (count%32)).
17. Rotate right: **ROR** dest, count// Rotate *dest* bits *count%32* times to the right.
17. Rotate left: **ROL** dest, count// Rotate *dest* bits *count%32* times to the left.
18. Rotate right including CF: **RCR** dest, count // The RCR instruction shifts the CF flag into the most-significant bit and shifts the least-significant bit into the CF flag
19. Rotate left including CF: **RCL** dest, count // The RCL instruction shifts the CF flag into the least-significant bit and shifts the most-significant bit into the CF flag

20. Unsigned multiply: **MUL** source2// destination <- source1 * source2;

Operand Size	Source 1	Source 2	Destination
Byte	AL	r/m8	AX
Word	AX	r/m16	DX:AX
Doubleword	EAX	r/m32	EDX:EAX

21. Unsigned divide **DIV** divizor// divident = quotient * divizor + remainder

Operand Size	Dividend	Divisor	Quotient	Remainder	Maximum Quotient
Word/byte	AX	r/m8	AL	AH	255
Doubleword/word	DX:AX	r/m16	AX	DX	65,535
Quadword/doubleword	EDX:EAX	r/m32	EAX	EDX	$2^{32} - 1$

22. Signed multiply:
* **IMUL** r/m (The same as **MUL** instruction)
* **IMUL** r, r/m (First operand = first operand*second operand)
* **IMUL** r, r/m, imm

* Compare: **CMP** *source1, source2*

Face diferenta intre *source1* si *source2.* Se seteaza flagurile de status corespunzator. (CF, OF, SF, ZF, AF, PF)

Instructiunea **CMP** este folosita, de obicei, impreuna cu o instructiune de salt conditionat (Jcc)

* Salturi conditionate: **Jcc** *label*

- testarea egalitatii:
  - Jump if equal/zero **JE/JZ** *label*
  - Jump of not equal/zero: **JNE/JNZ** *label*
- -fara semn
  - Jump if above/not below or equal: **JA/JNBE** *label*
  - Jump if above or equal/not below: **JAE/JNB** *label*
  - Jump if below/not above or equal: **JB/JNAE** *label*
  - Jump if below or qual/not above: **JBE/JNA** *label*
- cu semn
  - Jump if greater/not lower or equal: **JG/JNLE** *label*
  - Jump if greater or equal/not lower: **JGE/JNL** *label*
  - Jump if lower/not greater or equal: **JL/JNGE** *label*
  - Jump if lower or qual/not greater: **JLE/JNG** *label*
- testarea flagurilor
  - Jump if carry: **JC** *label*
  - Jump if not carry: **JNC** *label*
  - Jump if overflow : **JO** *label*
  - Jump if not overflow: **JNO** *label*
  - Jump if sign(negative): **JS** *label*
  - Jump if not sign(non-neg):**JNS** *label*
  - Jump if parity odd: **JPO/JNP** *label*
  - Jump if parity even: **JPE/JP** *label*

- urmatoarele instructiuni nu intra in programa si sunt optionale:
  - Jump when register *ECX* is zero: **JECXZ** *label*
  - Jump when register *CX* is zero: **JCXZ**
  - * Perform a loop operation using *ECX* or *CX* as a counter. Each time the *LOOP* instruction is executed, the count register is decremented and then checked for 0. If the count is 0, the loop terminates:
  - Decrement count and jump if count <> 0: **LOOP** *label*
  - Decrement count and jump if count <> 0 and *ZF=1*: **LOOPE** *label*
  - Decrement count and jump if count <> 0 and *ZF=1*: **LOOPZ** *label*
  - Decrement count and jump if count <> 0 and *ZF=0*: **LOOPNE** *label*
  - Decrement count and jump if count <> 0 and *ZF=0*: **LOOPNZ** *label*

RDTSC - Read TimeStamp Counter (EDX:EAX)

CLD - Clear Direction Flag (operatiile cu stringuri incrementeaza registrii index ESI/EDI)

STD - Set Direction Flag (operatiile cu stringuri decrementeaza registrii index ESI/EDI)

LODS/B/W/D = Load String
MOV AL/AX/EAX, [ESI]
ADD/SUB ESI,1/2/4
REP LODS/B/W/D = SE REPETA INSTRUCTIUNEA LODS/B/W/D DE ECX ORI

STOS/B/W/D = Store string
MOV [EDI], AL/AX/EAX
ADD/SUB ESI,1/2/4
REP STOS/B/W/D = SE REPETA INSTRUCTIUNEA LODS/B/W/D DE ECX ORI

MOVS/B/W/D = Move data from String to String
MOV TEMP, BYTE/WORD/DWORD PTR [ESI]
ADD/SUB ESI, 1/2/4
MOV BYTE/WORD/DWORD PTR [EDI], TEMP
ADD/SUB EDI, 1/2/4
REP MOVS/B/W/D = SE REPETA INSTRUCTIUNEA MOVS/B/W/D DE ECX ORI

SCAS/B/W/D = Scan String
CMP AL/AX/EAX, [EDI]
PUSHFD
ADD/SUB EDI, 1/2/4
POPFD
REPE/REPNE SCAS/B/W/D = SE REPETA INSTRUCTIUNEA SCAS/B/W/D DE MAXIM ECX ORI, CAT TIMP ZF=1 (REPE)/ZF=0(REPNE)

CMPS/B/W/D = Compare String Operands
MOV TEMP, BYTE/WORD/DWORD PTR [ESI]
CMP TEMP, BYTE/WORD/DWORD PTR [EDI]
PUSHFD
ADD/SUB ESI, 1/2/4
ADD/SUB EDI, 1/2/4
POPFD
REPE/REPNE CMPS/B/W/D = SE REPETA INSTRUCTIUNEA CMPS/B/W/D DE MAXIM ECX ORI, CAT TIMP ZF=1 (REPE)/ZF=0(REPNE)

* Salturi neconditionate:
Jump: **JMP** label

* Implementarea intructiunii *If-else*:

```
int a,b;
if( a > b){
   //instructiuni-1
}else{
   //instructiuni-2
}
```

```
-------------------------------------------------------------------------
;a si b sunt signed deci folosim instructiuni de salt contitionat - signed
MOV eax,a
MOV ebx,b
CMP eax,ebx
JLE _else
   //instructiuni-1
   JMP _end_if
_else:
   //instructiuni-2
_end_if
```

* Implementarea intructiunii *While:*

```
unsigned a,b;
while(a<b){
//instructiuni
```

```
}
```

```
--------------------------------------------------------------------------------
;a si b sunt unsigned deci folosim instructiuni de salt contitionat - unsigned
MOV eax,a
MOV ebx,b
_while:
CMP eax,ebx
JAE _end_while

//instructiuni

JMP _while

_end_while:
```

* Implementarea intructiunii *do-while:*

```
unsigned a,b;
do{
//instructiuni
}while(a<b);
```

```
--------------------------------------------------------------------------------
;a si b sunt unsigned deci folosim instructiuni de salt contitionat - unsigned
MOV eax,a
MOV ebx,b
_do_while:
    //instructiuni
    CMP eax,ebx
    JB _do_while
```

* Implementarea intructiunii *for*

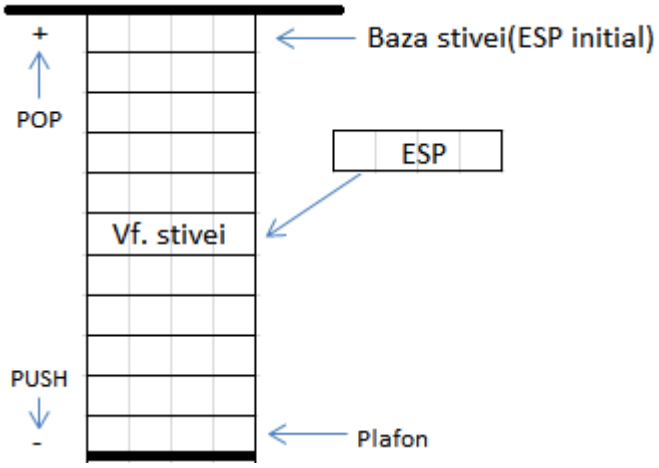
```
short limit;
for(short i=0;i<limit;i++){
    //instructiuni
}

--------------------------------------------------------------------------------
;i si limit sunt signed deci folosim instructiuni de salt contitionat - signed

MOV dx ,limit
XOR cx, cx
_for:
    CMP cx,dx
    JGE _end_for
    //instructiuni
    INC cx
    JMP _for
_end_for:
```

* stiva (push, pop, pusha, popa, pushf, popf)
* Functii, apel - CALL, RET
* descrierea unui apel de funcție (parametri, variabile locale)

Stiva este o zona continua de memorie si poate avea maxim 4GB. Elementele se pot adauga si elimina de pe stiva folosind intructiunile **PUSH** si **POP**.
La adaugarea unui element pe stiva, procesorul decrementeaza valoarea registrului *ESP,* dupa care scrie valoarea (la noul varf al stivei).
La eliminarea unui element de pe stiva, procesorul citeste valoarea ce se afla in varful stivei, dupa care incrementeaza valoarea lui *ESP.*
Astfel putem spune ca *Stiva* creste "in jos" (adica elementele noi sunt adaugate la adrese mai mici de memorie), si se micsoreaza "in sus" (la eliminarea elementelor de pe stiva, aceasta se micsoreaza inspre adrese mai mari de memorie)



Adaugarea unui element pe stiva: **PUSH** *operand*
Eliminarea unui element de pe stiva: **POP** *operand*
Operand poate fi registru (16bit/32bit), zona de memorie sau valoare (immediate byte/word/dword value)

Instructiunile *PUSH* si *POP* decrementeaza si respectiv incrementeaza in mod automat valoarea lui *ESP* cu dimensiunea operandului. Astfel, instructiunea *PUSH AX* va decrementa *ESP* cu 2 unitati.
Daca instructiunea *PUSH* este folosita impreuna cu o valoare directa atunci, indiferent de dimensiunea sa, valoarea este adaugata pe stiva intr-o reprezentare pe 32 de biti, printr-o extindere fara semn.

Ex.
PUSH 033h
MOV EAX, dword ptr [ESP]
ADD ESP, 4
//pune in registrul EAX valoarea 0x00000033 ca un DWORD: reprezentare C2(32,0).
//Codul este echivalent cu *MOV EAX,033h*

Instructiunile *PUSH* si *POP* nu pot folosi registri sau zone de memorie de dimensiune 1 octet (Ex: *PUSH AL*).

Alinierea Stivei. Se recomanda ca stiva sa fie aliniata la 4 octeti (DWORD). De exemplu, daca doriti sa puneti pe stiva continutul registrului *AX*, folositi:
PUSH EAX
sau
SUB ESP,4
MOV word ptr [ESP], AX

Bineinteles, daca doriti sa extrageti de pe stiva valoarea pe care tocmai ati adaugat-o, trebuie sa aveti grija ca valoarea lui ESP sa fie incrementata cu aceeasi valoarea cu care a fost decrementata. Astfel puteti folosi:

```
POP EAX // valoarea cautata se afla acum in AX
```

sau

```
MOV AX, word ptr [ESP]
ADD ESP,4
```

Pentru a pune pe stiva continutul tuturor registrilor generali se pot folosi instructiunile ***PUSHA/PUSHD.***
Instructiunea ***PUSHA*** pune pe stiva cei 8 registri generali pe 16 biti in aceasta ordine: AX, CX, DX, BX, SP, BP, SI, DI. Valoarea lui SP este cea initiala (inainte de executarea instructiunii). Dupa executarea acestei instructiuni, registrul ESP este mai mic cu 16 unitati.
Instructiunea ***PUSHAD*** pune pe stiva cei 8 registri generali pe 32 biti in aceasta ordine: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI. Valoarea lui ESP este cea initiala (inainte de executarea instructiunii). Dupa executarea acestei instructiuni, registrul ESP este mai mic cu 32 unitati.

Pentru a recupera de pe stiva registrii generali - in ordine inversa - se pot folosi instructiunile ***POPA/POPAD.*** Singurul registru care nu este recuperat este *ESP.* Acesta va fi incrementat cu 16, respectiv 32 unitati dupa executarea instructiunii.

Pentru a salva pe stiva/recupera continutul registrului EFLAGS se poate folosi instructiunea **PUSHFD/POPFD**

Apelul functiilor se realizeaza prin intermediul instructiunii **CALL**. Iesirea din functie se poate realiza prin intermediul instructiunii **RET.** In continuare sunt prezentate *CALL near* si *RET near* (procedura apelata/apelanta se afla in acelasi segment de cod cu procedura curenta)

La executia instructiunii **CALL**, procesorul se comporta in felul urmator:
1. Adauga pe stiva valoarea registrului *EIP.*
2. Incarca offsetul procedurii in registrul *EIP.*
3. Se executa prima instructiune din cadrul procedurii.

La executia instructiunii **RET**, procesorul se comporta in felul urmator:
1. Se face *POP* adresei de revenire de pe stiva in registrul *EIP.*
2. Daca instructiunea *RET* are un parametru (optional) specificat prin operandul *n*, se incrementeaza ESP cu valoarea *n*, pentru a elimina parametrii de pe stiva.
3. Se reia executia procedurii apelante.

Standardul MS VS C/C++ pentru apelul procedurilor este ***cdecl.*** In acest standard:
1. Stiva este curatata (adica parametrii sunt eliminati) de catre apelant.
2. Transmiterea argumentelor se face de la dreapta spre stanga.
3. Registrii EAX, ECX, EDX sunt salvati de catre procedura apelanta iar ceilalti de catre procedura apelata.

Transmiterea parametrilor. Parametrii pot fi transmisi prin intermediul registrilor de uz general, pe stiva, in ordine inversa.
Accesarea parametrilor actuali. Daca exista, primul parametru se gaseste la adresa *EBP+8.* Datorita alinierii stivei la 4 octeti, urmatorii parametri se gasesc pe stiva, la adresele *EBP+12, EBP+16,* etc (din 4 in 4 octeti).
Returnarea valorilor. Valorile numerice intregi (int, short, etc) si adresele sunt returnate prin intermediul registrului *EAX*.

Ex

```
int suma(int a, int b, int c){
   return a+b+c;
}

void main(){
   int s;
   s = suma( 10,20,30 );
}

//-------Echivalent ASM-------------------------

int suma(int,int,int){
   _asm{
      mov eax, [ebp+8] //primul parametru
      add eax, [ebp+12]// al II-lea parametru
      add eax, [ebp+16]// al III-lea parametru
      //se returneaza prin intermediul registrului EAX
   }
}
void main(){
   _asm{
      //se pun pe stiva parametrii in ordine inversa
      push 30
      push 20
      push 10
      call suma //apelul procedurii
      add esp, 12 //eliminarea parametrilor de pe stiva
   }
}
```

Cadrul de stiva (Stack Frame) reprezinta o zona de memorie de pe stiva, asociata unei proceduri, care contine spatiul necesar salvarii parametrilor actuali, a variabilelor locale, a contextului de apel (adresa de revenire) si alte variabile temporale. Cand procedura apelata se termina de executat, cadrul de stiva asociat este eliminat iar executia threadului curent este reluata cu prin continuarea procedurii apelante.

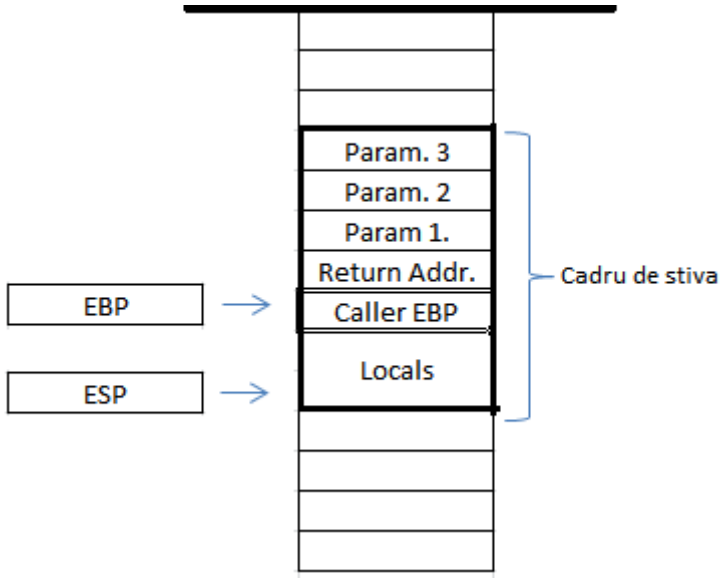
In cazul multor compilatoare (inclusiv MS VS) partea de inceput a functiilor este una standard:

```
PUSH EBP       //se salveaza base pointer pentru functia apelanta
MOV EBP, ESP //EBP se suprascrie cu adresa varfului curent al stivei
SUB ESP, X     //se face loc pentru variabilele locale
```

La fel si partea de sfarsit:

```
MOV ESP, EBP
POP EBP
RET
```

Astfel, in timpul executiei unei functii, stiva arata astfel:



* vectori 1D şi 2D (reprezentare în memorie şi metode de acces la un element)
* tablouri şi pointeri

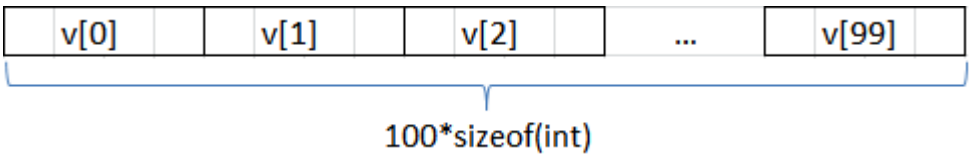
Instructiunea **LEA dest, source** (load efective address)

Vector 1D

int v[100]; //static

sau

int v = (int*)malloc(100*sizeof(int)); //dinamic*



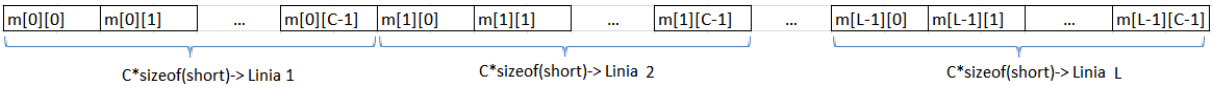
Accesarea elementelor

```
v[i]=5;
*(v+i)=5;
*((int*)((unsigned)v+sizeof(int)*i))=5;

_asm{
    LEA edi, v
    MOV ecx, i
    MOV dword ptr [edi+4*ecx],5
}
```

Matrice statica

```
#define L 100
#define C 50
short m[L][C];
```



Accesarea elementelor

```
m[i][j] = 5;
*(((short*)m[i])+j)=5;
*((short*)m + i*C+j)=5;
*((short*)((unsigned)m + i*C*sizeof(short)+j*sizeof(short)))=5;

_asm{
    LEA edi,m
    mov eax,C
    mov ebx,i
    mov ecx,j
    mul ebx
    add eax,ecx
    shl eax,1
    add edi,eax
    mov word ptr [edi],5
}
```

Transmiterea matricei ca parametru unei functii

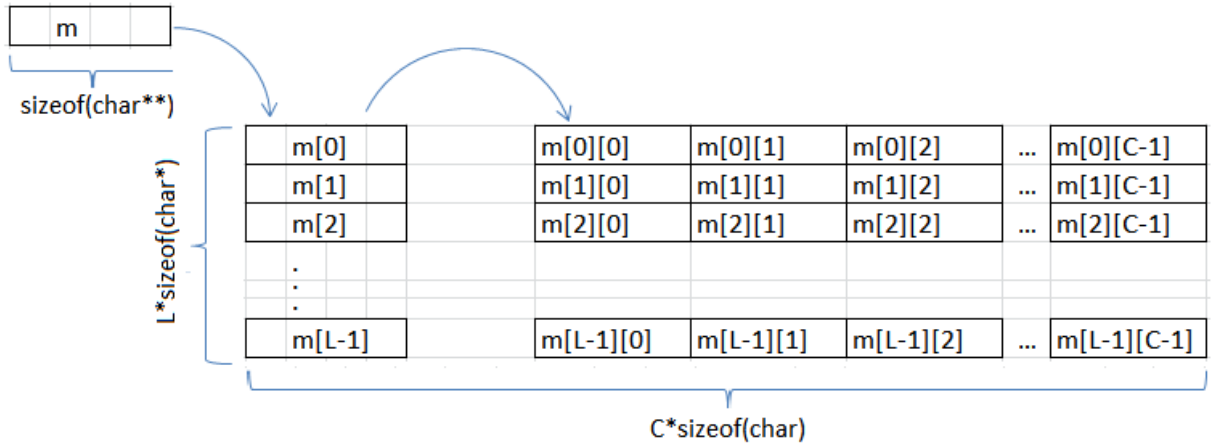
```
#include <stdio.h>
#include <malloc.h>
#define L 10
#define C 20


int f(int m[][C]){
 //returnarea lui m[0][0]
 _asm{
 mov eax,[ebp+8]
 mov eax,[eax]
 }
}
void main()
{
 int m[L][C];
 m[0][0]=1024000;
 int x = f(m);//vectorii si matricele se transmit prin referinta
 printf("%d\n",x); // se afiseaza 1024000
}
```

Matrice dinamica

```
#define L 10
#define C 20

char** m;
m = (char**)malloc(L*sizeof(char*));
for(int idx=0;idx<L;idx++){
    m[idx] = (char*)malloc(C*sizeof(char));
}
```



Accesarea elementelor

```
m[i][j]=5;
*(*(m+i)+j)=5;
*((char*)((unsigned)(*((char**)((unsigned)m+i*sizeof(char*))))+j*sizeof(char)))=5

_asm{
 mov edi,m
 mov ebx,i
 mov ecx,j
 mov edi,[edi+4*ebx]
 mov byte ptr [edi+ecx],5
}
```

Structuri, aliniere

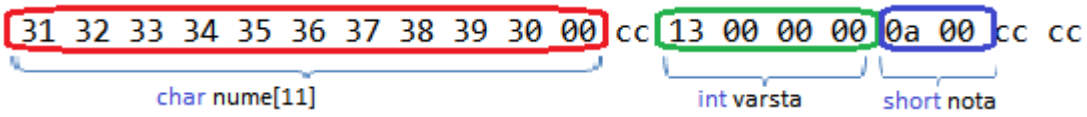
* Alinierea membrilor de date se realizeaza dependent de compilator;
* Datele membru sunt situate in memorie in ordinea in care s-a specificat in cod;
* In mod uzual, deplasamentul fiecarui membru fata de inceputul structurii este multiplu de dimensiunea sa;
* Structura se completeaza la sfarsit cu spatiu astfel incat marimea sa sa fie multiplu de cel mai mare dimensiune al vre-unui membru;

Exemplu

```
#include <stdio.h>
#include <string.h>

struct Persoana{
 char nume[11];
 int varsta;
 short nota;
};


void main(){
 Persoana p;
 strcpy(p.nume,"1234567890");
 p.nota = 10;
 p.varsta = 19;
 printf("%d",sizeof(p)); //se afiseaza 20
}
```


31 32 33 34 35 36 37 38 39 30 00 cc 13 00 00 00 0a 00 cc cc
char nume[11]
int varsta
short nota