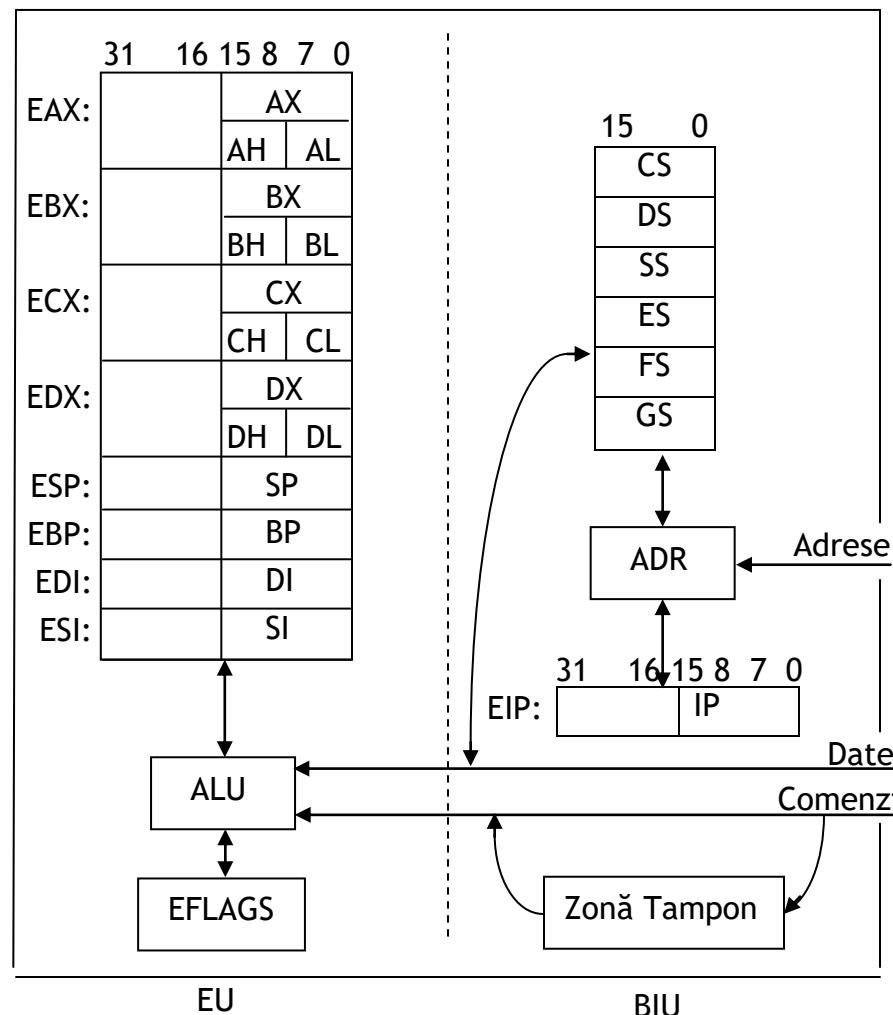


2.6. ARHITECTURA MICROPROCESOARELOR x86 (IA-32)

2.6.1. Structura micropresorului

Micropresorul x86 este format din două componente principale:

- **EU (Executive Unit)** - execută instr. mașină prin intermediul componentei **ALU (Aritmetic and Logic Unit)**.
- **BIU (Bus Interface Unit)** - pregătește execuția fiecărei instrucțiuni mașină. Citește o instrucțiune din memorie, o decodifică și calculează adresa din memorie a unui eventual operand. Configurația rezultată este depusă într-o zonă tampon cu dimensiunea de 15 octeți, de unde va fi preluată de EU.



EU și **BIU** lucrează în paralel - în timp ce **EU** execută instrucțiunea curentă, **BIU** pregătește instrucțiunea următoare. Cele două acțiuni sunt sincronizate - cea care termină prima aşteaptă după cealaltă.

2.6.2. Regiștrii generali EU

Registrul **EAX** este *registratorul acumulator*. El este folosit de către majoritatea instrucțiunilor ca unul dintre operanzi.

Registrul **EBX** - *registru general* (provine ca denumire de la Base Register – registru de baza – folosit în aceasta acceptiune la programarea pe 16 biți).

Registrul **ECX** - *registru de numărare (registru contor)* pt instrucțiuni care au nevoie de indicații numerice.

Registrul **EDX** - *registru de date*. Împreună cu EAX se folosește în calculele ale căror rezultate depășesc un dublucuvânt (32 biți).

"Word size" refers to the number of bits processed by a computer's CPU in one go (these days, typically 32 bits or 64 bits). Data bus size, instruction size, address size are usually multiples of the word size.

Just to confuse matters, for backwards compatibility, Microsoft Windows API defines a WORD as being A DATA TYPE on 16 bits, a DWORD being A DATA TYPE on 32 bits and a QWORD as 64 bits, regardless of the processor.

Regiștrii **ESP** și **EBP** sunt regiștri destinați lucrului cu *stiva*. O stivă se definește ca fiind o zonă de memorie în care se pot depune succesiv valori, extragerea lor ulterioară făcându-se în ordinea inversă depunerii.

Registrul **ESP** (*Stack Pointer*) punctează spre elementul ultim introdus în stivă (elementul din *vârful stivei*).

Registrul **EBP** (*Base pointer*) punctează spre primul element introdus în stivă (indică *baza stivei*).

Regiștrii **EDI** și **ESI** sunt *registri de index* utilizati de obicei pentru accesarea elementelor din siruri de octeți sau de cuvinte. Denumirile lor (*Destination Index* și *Source Index*) precum și rolurile lor vor fi clarificate în cap. 4.

Fiecare dintre regiștrii EAX, EBX, ECX, EDX, ESP, EBP, EDI, ESI au capacitatea de 32 biți. Fiecare dintre ei poate fi privit în același timp ca fiind format prin concatenarea (alipirea) a doi (sub)regiștri de câte 16 biți. Subregistru superior, care conține cei mai semnificativi 16 biți ai registrului de 32 biți din care face parte, nu are denumire și nu este disponibil separat. Subregistru inferior poate însă fi accesat individual, având astfel regiștrii de 16 biți **AX**, **BX**, **CX**, **DX**, **SP**, **BP**, **DI**, **SI**. Dintre aceștia, regiștrii AX, BX, CX, și DX sunt fiecare la rândul lor, formați din câte doi alți subregiștri a câte 8 biți. Există astfel regiștrii **AH**, **BH**, **CH**, **DH**, conținând cei 8 biți superiori (partea HIGH a regiștrilor AX, BX, CX și DX), respectiv **AL**, **BL**, **CL**, **DL**, conținând cei 8 biți inferiori (partea LOW).

2.6.3. Flagurile

Un *flag* este un indicator reprezentat pe un bit. O configurație a *registrului de flaguri* indică un rezumat sintetic a execuției fiecărei instrucțiuni. Pentru x86 registrul EFLAGS (the *status register*) are 32 biți dintre care sunt folosiți uzuale numai 9.

31	30	...	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	...	x	OF	DF	IF	TF	SF	ZF	x	AF	x	PF	x	CF

CF (*Carry Flag*) este flagul de transport. Are valoarea 1 în cazul în care în cadrul ultimei operații efectuate (UOE) s-a efectuat transport în afara domeniului de reprezentare a rezultatului și valoarea 0 în caz contrar. De exemplu, pt

1001 0011 +	147 +		93h +	-109 +
<u>0111 0011</u>	<u>115</u>	rezulta un transport de cifra semnificativa și	<u>73h</u>	<u>115</u>
1 0000 0110	262	valoarea 1 este depusa automat în CF (fara semn)	106h	06 (hexa)

Flagul CF semnalează depășirea în cazul interpretării FĂRĂ SEMN.

PF (*Parity Flag*) - Valoarea lui se stabilește a.î. împreună cu numărul de biți 1 din octetul cel mai puțin semnificativ al reprezentării rezultatului UOE să rezulte un număr impar de cifre 1.

AF (*Auxiliary Flag*) indică valoarea transportului de la bitul 3 la bitul 4 al rezultatului UOE. De exemplu, în adunarea de mai sus transportul este 0.

ZF (*Zero Flag*) primește valoarea 1 dacă rezultatul UOE este egal cu zero și valoarea 0 la rezultat diferit de zero.

SF (*Sign Flag*) primește valoarea 1 dacă rezultatul UOE este un număr strict negativ și valoarea 0 în caz contrar.

TF (*Trap Flag*) este un flag de depanare. Dacă are valoarea 1, atunci mașina se oprește după fiecare instrucțiune.

IF (*Interrupt Flag*) este flag de întrerupere. Detalii în cap.5 din manual.

DF (*Direction Flag*) - pt operare asupra șirurilor de octeți sau de cuvinte. Dacă are valoarea 0, atunci deplasarea în șir se face de la început spre sfârșit, iar dacă are valoarea 1 este vorba de deplasări de la sfârșit spre început.

OF (*Overflow Flag*) este flag pentru depășire **CU SEMN**. Dacă rezultatul ultimei instrucțiuni în interpretarea CU SEMN a operanzilor nu a încăput în spațiul rezervat operanzilor (intervalul de reprezentare admisibil), atunci acest flag va avea valoarea 1, altfel va avea valoarea 0. Pentru exemplul de mai sus, OF=0.

Categorii de flag-uri

Flag-urile se pot împărți în două categorii :

- a). flag-uri ale căror valori sunt setate ca efect direct al execuției Ultimei Operații Efectuate (UOE) : CF, PF, AF, ZF, SF și OF
 - b). flag-uri ale căror valori sunt setate de către programator în vederea influențării modului de operare al instrucțiunilor care urmează acestor setări : CF, TF, DF și IF.
-
- a). Flag-uri ce raportează efectul generat de UOE : CF, PF, AF, ZF, SF, OF (unde și CUM se tine cont în program de aceste valori ?) – ADC, salturi CONDITIONATE (cea mai frecventă utilizare)
 - b). Flag-uri cu efect ULTERIOR setării lor de către programator asupra instrucțiunilor ce urmează : CF, TF, IF, DF ; CUM le setam ??

Instrucțiuni specifice de setare a valorilor unor flag-uri

Având în vedere categoria b) este normal ca limbajul de asamblare să ne pună la dispoziție instrucțiuni specifice de setare a valorii flag-urilor care vor avea un efect ulterior. Aceste instrucțiuni sunt în număr de 7:

CLC – efectul fiind CF=0, STC – efectul fiind CF=1, CMC – complementarea valorii din CF

CLD – având ca efect DF=0, STD – având ca efect DF=1 ; CLI – având ca efect IF=0, STI – având ca efect IF=1

Având în vedere riscul major de setare accidentală a valorii din TF precum și rolul său absolut special în dezvoltarea de depanatoare, NU există disponibile instrucțiuni de acces direct la valoarea din TF !!

Operations Performed by ALU

1. **Arithmetic Operators**: It refers to bit subtraction and addition, despite the fact that it does multiplication and division. Multiplication and division processes, on the other hand, are more expensive to do. Addition can be used in place of multiplication, while subtraction can be used in place of division.
2. **Bit-Shifting Operators**: involves shifting the location of a bit to the right or left by a particular number of places. It is responsible for a multiplication operation.
3. **Logical Operations**: These consist of AND, OR, XOR, NOT

Complementul față de 2. Discuție și exemple.

Matematic, REPREZENTAREA unui număr NEGATIV în complement față de doi este valoarea $2^n - V$, unde V este valoarea absolută a numărului reprezentat.

Motto:

Cu complementul față de 2, “Interpretăm reprezentări și reprezentăm interpretări”

Obs. Notațiile din acest document care se referă la configurații în baza 2 NU vor conține sufixul b, deoarece această cerință există doar la nivel de sintaxă a limbajului de asamblare. Acest material nu conține cod sursă în limbaj de asamblare. În schimb, referirea la baza 16 va conține sufixul h pentru a se distinge de valorile din baza 10 (ex: 93h vs. 93).

Complementul față de 2. Discuție și exemple.

Matematic, reprezentarea unui număr NEGATIV în complement față de doi este valoarea $2^n - V$, unde V este valoarea absolută a numărului reprezentat.

1001 0011 (= 93h = 147), deci în interpretarea FARA SEMN 1001 0011 = 147

Fiind un număr care începe cu 1, în interpretarea CU SEMN, acest număr este unul negativ. Cât este valoarea lui ?

Raspuns: Valoarea sa este: $-(\text{complementul față de 2 al configurației binare inițiale}) = -(2^n - V)$

Ca urmare trebuie să determinăm complementul față de 2 al configurației 1001 0011

Cum se obține complementul față de 2 al unui număr (reprezentat în memorie, deci reprezentat în baza 2) ?

Varianta 1 (Oficială): Se scade binar conținutul (evidenț binar) al locației de complementat din 100 ...00, unde numărul de după cifra binară 1 are atâtea zerouri câtă biți are locația de complementat.

$$\begin{array}{r} 1 \ 0000 \ 0000 - \\ \underline{1001 \ 0011} \\ \hline 0110 \ 1101 \end{array} \quad \begin{array}{l} 2^n - \\ V \end{array} = 6Dh = 96 + 13 = 109 \text{ (deci complementul față de 2 pe 8 biți al numărului 147 este 109)}$$

Ca urmare, valoarea în interpretarea CU SEMN a numărului 1001 0011 este -109

Varianta 2 (derivată din definiția complementului față de 2 – mai rapidă dpdV practic): se inversează valorile tuturor biților (valoarea 0 devine 1 și valoarea 1 devine 0) din locația de reprezentare, după care se adaugă 1 la valoarea obținută.

Cf acestei reguli, plecăm deci de la 1001 0011 și inversăm valorile tuturor biților, obținând 0110 1100 după care adaugam 1 la valoarea obținută: $0110 \ 1100 + 1 = 0110 \ 1101 = 109$; Ca urmare, valoarea în interpretarea CU SEMN a numărului 1001 0011 este -109

Varianta 3 (MULT mai rapidă dpdv practic, cea mai rapidă pt obținerea configurației binare a complementului față de 2): Se lasă neschimbați biții începând din dreapta reprezentării binare până la primul bit 1 inclusiv, iar restul biților se inversează.

Aplicând această regulă, plecăm de la 1001 0011 și lăsăm neschimbați biții din dreapta până la primul bit 1 inclusiv (în cazul nostru asta înseamnă doar primul bit 1 din dreapta – care ramâne aşadar neschimbat), iar restul biților se inversează, deci obținem... 0110 1101 = 6DH = 109

Deci, valoarea în interpretarea CU SEMN a numărului 1001 0011 este -109

Varianta 4 (CEA mai rapidă dpdv practic, dacă ne interesează doar valoarea absolută în baza 10 a complementului față de 2):

Regulă derivată din definiția complementului față de 2: Suma valorilor absolute a celor două valori complementare este cardinalul mulțimii reprezentabile pe acea dimensiune.

- Pe 8 biți se pot reprezenta 2^8 valori = 256 valori ([0..255] sau [-128..+127])
- Pe 16 biți se pot reprezenta 2^{16} valori = 65536 valori ([0..65535] sau [-32768,+32767])
- Pe 32 biți se pot reprezenta 2^{32} valori = 4.294.967.296 valori (...)

Deci pe 8 biti , complementul fata de 2 a lui 1001 0011 (= 93h = 147) este $256 - 147 = 109$, ca urmare valoarea în interpretarea CU SEMN pentru 1001 0011 este -109

[0..255] – interval de reprezentare admisibil pt “intreg FARA semn reprezentat pe 1 octet” (UNSIGNED BYTE)
[-128..+127] – interval de reprezentare admisibil pt “intreg CU semn reprezentat pe 1 octet” (SIGNED BYTE)

[0..65535] – interval de reprezentare admisibil pt “intreg FARA semn repr. pe 1 Word = 2 octeți” (UNSIGNED WORD)
[-32768..+32767] – interval de reprezentare admisibil pt “intreg CU semn repr. pe 1 Word = 2 octeți” (SIGNED WORD)

La ce ne trebuie complementul față de 2 nouă ca programatori în ASM ?

1001 0011 (= 93h = 147), deci în interpretarea FARA SEMN 1001 0011= 147

Care este valoarea în interpretarea cu semn a lui 1001 0011 ? a). 01101101 b). -109 c). 6Dh d). +147

Care este valoarea în interpretarea cu semn a lui 93h ? a). 01101101 b). -109 c). 6Dh d). +147

Care este valoarea în interpretarea cu semn a lui 147 din baza 10 ? a). 01101101 b). -109 c). 6Dh d). +147 (nici unul dintre răspunsuri, deoarece întrebarea este un nonsens – pentru că 147 **ESTE DEJA O INTERPRETARE !!!**)

1 0000 0000 –

1001 0011

01101101

= 6Dh = 96+13 = 109 (deci complementul față de 2 pe 8 biți al numărului 147 este 109)

Ca urmare, valoarea în interpretarea CU SEMN a numărului 1001 0011 este... -109

Deci 147 și 109 sunt 2 valori complementare, în sensul că 1001 0011 = fie 147, fie -109 în funcție de interpretare.

Deci complementul lui 147 este -109. Este valabil oare și invers ? Este -147 complementul lui 109 ?... NU !!!!!!

Și dacă nu, care este complementul lui 109 atunci ?

Să verificam... $109 = 01101101$, complementul față de 2 a lui 01101101 este $10010011 = 147$, deci.... Ce facem dpdv practic cu informația aceasta ? ...și care este concluzia atunci ?...

Matematic, reprezentarea unui număr NEGATIV în complement față de doi este valoarea $2^n - V$, unde V este valoarea absolută a numărului reprezentat. Deci toată discuția despre complementul față de 2 are sens dpdv practic DOAR ATUNCI când ne referim la REPREZENTAREA BINARA a unui număr NEGATIV din baza 10 !!! Adică la INTERPRETAREA CU SEMN a unui număr binar care începe cu 1 !! Adică DOAR atunci când discutăm despre INTERPRETAREA de numere care încep cu 1 în baza 2 !!!!!! Atunci când avem un număr binar care începe cu 0, INTERPRETAREA lui VA FI ACEEASI atât în CU SEMN cât și fără semn, adică $01101101 = 109$ IN AMBELE INTERPRETARI !! Ca urmare, faptul că 147 este în sine “complementul lui 01101101 (109 în baza 10) față de 2 ca valoare absolută” nu ne ajută la nimic și nu are nici o urmare practică pentru nimic într-un astfel de context.

Deci: toată discuția despre valori complementare are sens doar dacă atenția noastră se concentrează pe problematica NUMERELOR NEGATIVE !! Ex: se pleacă de la o reprezentare în baza 2 care începe cu 1 și ne întrebăm care va fi numărul negativ asociat în interpretarea CU SEMN !! Sau: plecăm de la o valoare absolută (109 sau 147) și ne întrebăm care este reprezentarea în baza 2 pentru numărul -109 sau pentru -147 !! Deci totul se învârte în jurul numerelor NEGATIVE !!

O abordare care pleacă de la o reprezentare în baza 2 care începe cu 0 și se întreabă sec DOAR care va fi “valoarea complementară a acelei reprezentări” (fără ca această valoare complementară să fie folosită apoi la ceva concret) nu își are sensul scoasă din contextul interpretării unui număr CU SEMN ca negativ.

Faptul că există un interval de valori pentru care interpretarea unei configurații binare este aceeași în ambele interpretări se întâmplă (luăm ca exemplu dimensiunea unui octet) deoarece $[0..255] \cap [-128..+127] = [0..+127]$ și astfel orice număr din intervalul $[0..+127]$ va reprezenta atât varianta CU SEMN cât și cea FARA SEMN a configurației binare date ! Aceasta REPREZENTARE în baza 2 va fi o secvență de biți care începe cu 0, o astfel de secvență fiind caracterizată de faptul că în baza 10 valoarea sa va fi aceeași (pozitivă!) în ambele interpretări. Aceste numere fac parte din intersecția interpretării fără semn $[0..255]$ cu cea cu semn $[-128 ..+127]$.

Deci intersecția intervalelor de reprezentare admisibile pe dimensiunea N este constituită DOAR din valorile care în binar încep cu bitul 0 ! Ca urmare, valorile binare care încep cu bitul 1 NU sunt comune acestor intervale “complementare”, ceea ce înseamnă că interpretările cu semn și fără semn ale oricărei configurații binare care începe cu 1 VOR FI INTOTDEAUNA DIFERITE și ele NU vor fi NICIODATA părți ale aceluiași interval de reprezentare admisibil !!! Valorile absolute ale celor două interpretări reprezintă două valori complementare. Ex: -128, 128; 147, -109; -1, 255; -3, 253; **-127, 129** NU vor face parte niciodată din același interval de reprezentare admisibil. (NU se aplică însă și pentru **127, -129 !!!** – explicațiile urmează... ☺)

Așadar... care este reprezentarea binara pt -147 ? Sau cât este 10010011 în interpretarea cu semn (= -109) ? 147 = 10010011, deci ... cum il obtinem pe -147 ?

Mai concret, hai să vedem care sunt **tipurile de întrebări** pe care le putem pune plecând de la o configurație binară dată și să fixăm astfel care sunt situațiile în care apare necesitatea utilizării complementului față de 2:

- a) Dacă avem o REPREZENTARE **în baza 2** de tip $\overline{0xxx}....$ de valoare $+abc$ în interpretarea FARA semn, ce valoare va avea această REPREZENTARE în interpretarea CU SEMN din **BAZA 10** ? (b2 – b10)

R: Tot atâta ! Un număr care începe cu 0 în baza 2 are aceeași valoare atât în interpretarea cu semn cât și în cea fără semn, fiind un număr pozitiv (109 este tot 109 în ambele interpretări).

- b). Dacă avem o REPREZENTARE de forma $\overline{0xxx}....$ de valoare $+abc$, care va fi REPREZENTAREA binară a valorii $-abc$? (Ex: dacă plecăm de la 109, cum se reprezintă binar -109 ?) (b2 – b2)

R: De abia într-o astfel de întrebare începe “complementul față de 2 să joace un rol”, iar răspunsul este: REPREZENTAREA sa va fi **”complementul față de 2 al configurației binare inițiale”**. Pentru valoarea $109 = 01101101$, complementul față de 2 a lui 01101101 este 10010011 , deci $-109 = 10010011$

Ca urmare putem concluziona că valoarea complementară a unui întreg care începe cu 0, va începe cu 1 (excepție făcând doar valoarea 0) și va încăpea ca valoare complementară în interpretarea CU SEMN pe aceeași dimensiune de reprezentare ca și valoarea inițială !! (-109 este tot un byte ca și 109).

- c). Dacă avem o REPREZENTARE de tip $\overline{1xxx}....$ de valoare $+abc$ în interpretarea FĂRĂ semn, ce valoare va avea această REPREZENTARE în interpretarea CU SEMN din baza 10 ? (b2 – b10)

R: Valoarea sa este: $-$ (complementul față de 2 al configurației binare inițiale). Pentru exemplul nostru, avem: $10010011 = 147$ (fără semn) $= -$ (complementul față de 2 al configurației 10010011) $= -(01101101) = -109$.

- d). Dacă avem o REPREZENTARE de forma $\overline{1xxx}....$ de valoare $+abc$, care va fi REPREZENTAREA binară a valorii $-abc$? (Ex: dacă plecăm de la $10010011 = +147$, cum se reprezintă binar -147 ?) (b2 – b2)

R: Răspunsul nu poate fi decât unul similar cu cel de la b) : REPREZENTAREA sa va fi "complementul față de 2 al configurației binare inițiale". Numai că: dacă un număr începe cu 1 în reprezentarea lui în baza 2 și are valoarea $+abc$ în interpretarea FARA SEMN, atunci tot cu 1 va trebui să înceapă și varianta lui negativă – \overline{abc} ca REPREZENTARE (pentru că în caz contrar nu ar mai fi un număr negativ în interpretarea CU SEMN). Dar, complementarea unei valori binare de forma $\overline{1xxx}....$ va furniza prin complementare în mod natural o valoare binară CARE INCEPE CU 0 pe o dimensiune de reprezentare identică cu cea de pornire !!! - excepție făcând DOAR valorile de forma $\overline{100}....$ (-128, +128, -32768, +32768 etc).

Ca urmare, concluzionăm că dacă plecăm de la o reprezentare de forma $\overline{1xxx}....$ de valoare $+abc$ NU PUTEM OBȚINE valoarea – \overline{abc} PE O ACEEAȘI DIMENSIUNE DE REPREZENTARE !!!!!

Dovada : $147 = 10010011$ ($147 \in [0..255]$, însă $-147 \notin [-128..+127]$, deci -147 NU este reprezentabil pe un octet chiar dacă $+147$ este !!!!!!)

Deci nu numai că, complementarea nu poate fi făcută corect pe aceeași dim. de reprezentare ca a val. inițiale ca METODOLOGIE, dar și analiza intervalelor de reprezentare admisibile ne confirmă asta dpdv SEMANTIC !!!!

Ca urmare, obținerea lui -147 plecând de la $147 = 10010011$, se face astfel:

- i). Reprezentarea binară a lui 147 începe cu 1, dar trebuie să ținem cont și că $-147 \notin [-128..+127]$, însă $-147 \in [-32768..+32767]$ de unde rezultă că -147 NU este reprezentabil ca octet CI DOAR CA ȘI CUVÂNT !!
- ii). Pe o dimensiune de tip WORD, $147 = 00000000\ 10010011$ (deci un număr binar care începe cu 0) și conform răspunsului de la b), avem că -147 = "complementul față de 2 al configurației binare inițiale".

Complementul față de 2 a configurației $00000000\ 10010011$ este $11111111\ 01101101$, deci
 $-147 = 11111111\ 01101101 = FF6Dh$

Verificare: 11111111 01101101 = FF6Dh = 65389 în interpretarea FĂRĂ semn, suma valorilor absolute a celor două valori complementare fiind $65389 + 147 = 65536$ = cardinalul mulțimii numerelor reprezentabile pe 1 WORD, deci cele 2 interpretări ale configurației binare 11111111 01101101 sunt corecte și consistente !!

Ca urmare, putem deduce că implicarea “complementului față de 2” se manifestă în **3 cazuri** :

Format binar nr	Interpretare	Valoare zecimală	În ce fel este implicat “complementul față de 2”	Răspuns
0xxx	Fără semn	+abc	-	-
	Cu semn	+abc	Cum se reprezintă -abc ? (b)	Compl față de 2 a lui 0xxx
1xxx	Fără semn	+def	-	-
	Cu semn		Ce valoare va avea 1xxx în interpretarea cu semn ? (c)	-(compl față de 2 a lui 1xxx)
	Cu semn		Cum se reprezintă -def ? (d)	Compl față de 2 a lui 1xx extins fără semn pe $2 * \text{sizeof}(1xxx)$

(excepție fac reprezentările de forma de forma $\overline{100}.... (-128, +128, -32768, +32768 etc)$.

Noțiunea de “complement față de 2” NU este implicată în nici un fel atunci când abordăm doar reprezentările **fără semn** ! Exemplu:

Nr binar	Interpretare	Valoare	Implicare “complement față de 2”	Răspuns
01101101	Fără semn	+109	-	-
	Cu semn	+109	Cum se reprezintă -109 ? (b)	10010011
10010011	Fără semn	+147	-	-
	Cu semn		Ce valoare va avea 10010011 în interpretarea cu semn ? (c)	- (01101101) = -109
	Cu semn		Cum se reprezintă -147 ? (d)	Complementul față de 2 a lui 00000000 10010011 care este 11111111 01101101

Coloanele 3 și 4 de mai sus pot fi rezumate astfel:

Numărul X în reprezentare binară începe cu	-X începe cu	-X se reprezintă LA NIVELUL ARHITECTURII și al LIMBAJULUI DE ASAMBLARE:	Exemple:
0	1	Pe aceeași dimensiune ca X	$109 = 01101101 ; -109 = 10010011$
1	1	$2 * \text{sizeof}(X)$	$147 = 10010011; -147 = 11111111 01101101$

(excepție fac reprezentările de forma de forma $\overline{100}....$ ($-128, +128, -32768, +32768$ etc)).

Ca și concluzii:

* pentru o configurație binară de valoare VAL care începe cu **0** în baza 2, -VAL se va reprezenta pe aceeași dimensiune de reprezentare ca și reprezentarea inițială (109 și -109 se REPREZINTĂ pe aceeași dimensiune de reprezentare - 1 octet).

* pentru o configurație binară de valoare VAL care începe cu **1** în baza 2, -VAL se va reprezenta pe o dimensiune de reprezentare MAI MARE decât cea inițială - de dimensiune DUBLĂ ca tip de date în limbaj de asamblare (147 reprezentabil pe 1 octet, -147 reprezentabil pe 1 WORD), dar în fapt doar cu 1 bit mai mult “pe hârtie” dpdv al “teoriei” regulilor de complementare ($147 = 10010011 = 8$ biți, $-147 = 1 01101101 = 9$ biți necesari MINIMAL pentru reprezentare). Deci -147 nu se poate reprezenta pe octet !!

Care este nr MINIM de biți pe care se poate reprezenta -147 ?

- Pe n biti se reprezinta 2^n valori:
 - fie valorile $[0..2^n - 1]$ in interpretarea FARA SEMN
 - fie valorile $[-2^{(n-1)}, 2^{(n-1)}-1]$ in interpretarea CU SEMN

Pe 8 biti astfel se pot reprezenta 2^8 valori ($=256$ valori), fie $[0..2^8-1] = [0..255]$ in interpretarea FARA SEMN, fie valorile $[-2^{(8-1)}, 2^{(8-1)}-1] = [-2^7, 2^7-1] = [-128..+127]$ in interpretarea CU SEMN

Pe 9 biti... $[0..511]$ sau $[-256..+255]$ si cum $-147 \in [-256..+255]$ rezulta ca numarul MINIM de biti pe care se poate reprezenta -147 este 9, iar reprezentarea lui -147 este:

(...Pe 9 biti se pot reprezenta 512 numere, $512-147 = 365 = 1\ 6Dh = 1\ 0110\ 1101 \dots$)

Deci $1\ 0110\ 1101 = 16Dh = 256 + 6*16 + 13 = 256 + 96 + 13 = 365$ in interpretarea FARA SEMN !

$1\ 0110\ 1101 = -(complementul\ fata\ de\ 2\ a\ lui\ 1\ 0110\ 1101) = -(0\ 1001\ 0011) = -(093h) = -147$

Ca si TD in ASM, evident ca avem byte, word sau dword, deci $-147 \in [-32768..+32767]$ si cf. celor de mai sus avem $-147 = 11111111\ 01101101 = FF6Dh$ ca valoare reprezentata pe cuvant (word) = 2 octeti.

Care este numărul minim de biți pe care se poate reprezenta 3 ?

Răspuns: pe 2 biți; $3 = 11b$

Pe 2 biti se pot reprezenta 2^2 valori (=4 valori), fie $[0..2^2-1] = [0..3]$ in interpretarea FARA SEMN, fie valorile $[-2^{(2-1)}, 2^{(2-1)}-1] = [-2^1, 2^1-0] = [-2..+1]$ in interpretarea CU SEMN

Care este numărul minim de biți pe care se poate reprezenta -3 ?

Răspuns: pe 3 biți; deoarece pe 2 biți NU se poate datorită explicației de mai sus și pe 3 biți avem că:

Pe 3 biti se pot reprezenta 2^3 valori (=8 valori), fie $[0..2^3-1] = [0..7]$ in interpretarea FARA SEMN, fie valorile $[-2^{(3-1)}, 2^{(3-1)}-1] = [-2^2, 2^2-1] = [-4..+3]$ in interpretarea CU SEMN

Ca urmare, cum

(...Pe 3 biti se pot reprezenta 8 numere, $8-3 = 5 = 101b\dots$) - deci 101b este reprezentarea lui -3 pe 3 biți

101b = 5 in interpretarea FARA SEMN !

101b = -(complementul fata de 2 a lui 101) = -(011) = -3

De lămurit: Cine este “complementul față de 2” pentru cine ? Si față de cine ?...

O configurație binară față de alta ? Sau un număr zecimal cu semn față de unul fără semn ? Sau două valori absolute una față de alta ?... Vom vedea că se folosesc în practică toate cele 3 exprimări...

“Complementul față de 2” se referă la REPREZENTĂRI sau la INTERPRETĂRI... ?

Pt că este “FAȚĂ DE 2” se referă LA BAZA 2, deci ne referim în primul rând ca definiție la REPREZENTĂRI

- complementul față de 2 a lui **01101101** este 10010011 (asta mă ajută să răspund la întrebarea “care este reprezentarea lui -109 ?”) (b)
- complementul față de 2 a lui 10010011 este **01101101** (asta mă ajută să răspund la întrebarea “care este valoarea în interpretarea CU SEMN a lui 10010011”) (c)

Pe de altă parte ne referim ȘI LA INTERPRETĂRI, mai exact la INTERPRETĂRILE FĂRĂ SEMN ale celor două reprezentări binare complementare - 01101101 și 10010011 de exemplu – spunând că “109 și 147 sunt 2 valori complementare” (echivalent semantic și cu “Valorile absolute ale celor două interpretări ale unui număr binar ce începe cu 1 reprezintă două valori complementare”). Adică 147 și 109 sunt 2 valori complementare, în sensul că REPREZENTAREA 1001 0011 = fie +147, fie -109 în funcție de interpretare. INSA NU există o reprezentare care să aibă fie valoarea -147, fie 109 !!

De aceea, pentru a distinge claritatea a ceea ce se vrea a fi subliniat, se forțează de multe ori exprimarea într-un sens poate incorect cumva relativ la definiție dar relevant ca și concluzie în cazuri ca cel de mai sus, astfel: 147 și 109 sunt 2 valori complementare, în sensul că “**-109 este complementul lui +147**”, însă “**-147 NU este complementul lui +109** !!! Forțarea aici este reprezentată de apariția semnului MINUS în exprimare... însă pe de altă parte devine clar ceea ce se vrea exprimat.

Deci... NU are sens discuția despre valori complementare în baza 10 pornind de la o REPREZENTARE care începe cu 0 în baza 2 !! ... de exemplu în sensul că pornind de la **01101101** (109) valoarea sa complementară este 10010011 (147). Ca aceste concepte să aibă sens trebuie să fie implicate în discuție DIN START “reprezentarea unui număr negativ” sau “interpretarea unei configurații binare CE INCEPE CU BITUL 1 ca număr cu semn”.

Deci implicarea “complementului față de 2” este una UNIDIRECȚIONALĂ !!! având sens doar pornind dinspre interpretarea CU SEMN și a reprezentării numerelor negative ! Asta exprima de fapt “forțarea” de mai sus...

2.6.4. Registrrii de adresă și calculul de adresă

Adresa unei locații – nr. de **octeți** consecutivi dintre începutul memoriei RAM și începutul locației respective.

O succesiune continuă de locații de memorie, menite să deservească scopuri similare în timpul execuției unui program, formează un *segment*. În consecință, un segment reprezintă o diviziune logică a memoriei unui program, caracterizată prin *adresa de bază* (început), *limita* (dimensiune) și *tipul* acesteia. Atât adresa de bază cât și dimensiunea unui segment au valori reprezentate pe 32 biți.

In the family of 8086-based processors, the term **segment** has two meanings:

1. A block of memory of discrete size, called a *physical segment*. The number of bytes in a physical memory segment is
 - o (a) 64K for 16-bit processors
 - o (b) 4 gigabytes for 32-bit processors.
2. A variable-sized block of memory, called a *logical segment* occupied by a program's code or data.

Vom numi *offset* sau *deplasament* adresa unei locații față de începutul unui segment, sau, cu alte cuvinte, numărul de octeți aflați între începutul segmentului și locația în cauză. Un offset se consideră valid dacă și numai dacă valoarea sa numerică, pe 32 biți, nu depășește limita (dimensiunea) segmentului la care se raportează.

Vom numi *specificare de adresă* sau *adresă logică* o pereche formată dintr-un *selector de segment* și un offset. Un **selector de segment** este o valoare numerică de 16 biți care identifică (indică/selectează) în mod unic segmentul accesat și caracteristicile acestuia. **Un selector de segment este definit și furnizat de catre sistemul de operare !!** În scriere hexazecimală o adresă se exprimă sub forma:

S₃S₂S₁S₀ : 0706050403020100

În acest caz, selectorul S₃S₂S₁S₀ indică accesarea unui segment a cărui adresă de bază este de forma b₇b₆b₅b₄b₃b₂b₁b₀ și având o limită l₇l₆l₅l₄l₃l₂l₁l₀. Baza și limita sunt determinate de către procesor în urma aplicării mecanismului de segmentare.

Pentru a fi permis accesul către locația specificată, este necesar să fie îndeplinită condiția:

$$0706050403020100 \leq l_7l_6l_5l_4l_3l_2l_1l_0.$$

Determinarea *adresei de segmentare* din specificarea de adresă se face printr-un *calcul de adresă* cf. formulei:

$$a_7a_6a_5a_4a_3a_2a_1a_0 := b_7b_6b_5b_4b_3b_2b_1b_0 + 0706050403020100$$

unde $a_7a_6a_5a_4a_3a_2a_1a_0$ este adresa calculată (scrisă în hexazecimal). Adresa rezultată din calculul de mai sus, poartă numele de *adresă liniară* (sau *adresă de segmentare*).

O specificare de adresă mai poartă și numele de adresă FAR (îndepărtată). Atunci când o adresă se precizează doar prin offset, spunem ca este o adresă NEAR (apropiată).

Un exemplu concret de specificare de adresă este:

8:1000h

Pentru a calcula adresa liniară ce-i corespunde acestei specificări, procesorul va proceda după cum urmează:

1. Verifică dacă segmentul ce corespunde valorii de selector 8 a fost definit de către sistemul de operare și se blochează accesul dacă nu a fost definit un astfel de segment;
2. Extragă adresa de bază (B) și limita acestui segment (L), de exemplu, ca rezultat am putea avea $B = 2000h$ și $L = 4000h$; (este o operație la ale cărei detalii NU avem acces, ea derulându-se exclusiv între procesor și SO)
3. Verifică dacă offsetul depășește limita segmentului: $1000h > 4000h$? În caz de depășire accesul ar fi fost blocat (*memory violation error*);
4. Adună offsetul cu B, obținând în cazul nostru adresa liniară $3000h$ ($1000h + 2000h$). Acest calcul este efectuat de către componenta **ADR** din **BIU**.

Acest mecanism de adresare poartă numele de *segmentare*, vorbind astfel despre *modelul de adresare segmentată*.

În cazul în care segmentele încep la adresa 0 și au dimensiunea maximă posibilă (4GiB), orice offset este automat valid și segmentarea nu contribuie efectiv în calculul adreselor. Astfel, având $b_7b_6b_5b_4b_3b_2b_1b_0 = 00000000$, calculul de adresă pentru adresa logică $s_3s_2s_1s_0 : 0706050403020100$ va rezulta în adresa liniară:

$$\mathbf{a_7a_6a_5a_4a_3a_2a_1a_0 := 00000000 + 0706050403020100}$$

$$\mathbf{a_7a_6a_5a_4a_3a_2a_1a_0 := 0706050403020100} \\ \Rightarrow$$

Acest mod particular de utilizare a segmentării, folosit de către majoritatea sistemelor de operare moderne poartă numele de *model de memorie flat*.

Procesoarele x86 suportă și un mecanism de control al accesului la memorie numit *paginare*, independent de adresarea segmentată. Paginarea implică împărțirea memoriei *virtuale* în *pagini*, care sunt asociate (translatate) memoriei fizice disponibile (1 page = 4 KB = 2^{12} bytes = 4096 bytes).

Configurarea și controlul mecanismelor de segmentare și paginare sunt sarcina sistemului de operare. Dintre cele două, doar segmentarea intervine în specificarea de adrese, paginarea fiind complet transparentă din perspectiva programelor de utilizator.

Atât calculul de adrese cât și folosirea mecanismelor de segmentare și paginare sunt influențate de *modul de execuție* al procesorului, procesoarele x86 suportând următoarele moduri de execuție mai importante:

- *mod real*, pe 16 biți (folosind cuvânt de memorie de 16 biți și având memoria limitată la 1MB);
- ***mod protejat pe 16 sau 32 biți, caracterizat prin folosirea paginării și segmentării;***
- *mod virtual 8086*, permite rulare programelor de tip mod real alături de cele de mod protejat;
- *long mode*, pe 64 sau 32 biți, unde paginarea este obligatorie în timp ce segmentarea este dezactivată.

În cadrul cursului nostru ne vom concentra asupra arhitecturii și comportamentului procesoarelor din familia Intel x86 în modul protejat pe 32 de biți.

Arhitectura x86 permite folosirea a patru tipuri de segmente cu roluri diferite:

- *segment de cod*, care conține instrucțiuni mașină;
- *segment de date*, care conține date asupra cărora se acționează în conformitate cu instrucțiunile;
- *segment de stivă*;
- *segment suplimentar de date* (extrasegment).

Fiecare program este compus din unul sau mai multe segmente, de unul sau mai multe dintre tipurile de mai sus. În fiecare moment al execuției este activ cel mult câte un segment din fiecare tip. Regiștrii **CS** (*Code Segment*), **DS** (*Data Segment*), **SS** (*Stack Segment*), **ES** (*Extra Segment*) din **BIU** conțin valorile selectorilor segmentelor active, corespunzător fiecărui tip. Deci regiștrii CS, DS, SS și ES determină adresele de început și dimensiunile segmentelor active: de cod, de date, de stivă și suplimentar. Regiștrii **FS** și **GS** pot reține selectori indicând către segmente suplimentare, fără însă a avea roluri predeterminate. Datorită utilizării lor, CS, DS, SS, ES, FS și GS poartă denumirea de *regiștri de segment* (sau *regiștri selectori*). Registrul **EIP** (care oferă și posibilitatea accesării cuvântului său inferior prin subregistru **IP**) conține offsetul instrucțiunii curente în cadrul segmentului de cod curent, el fiind manipulat exclusiv de către **BIU**.

Cum noțiunile asociate adresării sunt fundamentale înțelegерii funcționării procesoarelor x86 și programării în limbaj de asamblare, este foarte importantă cunoașterea acestora. Pentru aceasta, le recapitulăm pe scurt în vederea clarificării:

Noțiune	Reprezentare	Descriere
Specificare de adresă, Selector ₁₆ :offset ₃₂ adresă logică, adresă FAR		Definește complet atât segmentul cât și deplasamentul în cadrul acestuia
Selector de segment	16 biți	Identifică unul dintre segmentele disponibile. Ca valoare numerică acesta codifică poziția descriptorului de segment selectat în cadrul unei tabele de descriptori.
Offset, adresă NEAR	Offset ₃₂	Definește doar componenta de offset (considerând segmentul cunoscut ori folosirea modelului de memorie flat)
Adresă liniară (adresă de segmentare)	32 biți	Inceput segment + offset, reprezintă <u>rezultatul calculului de adresă</u>
Adresă fizică efectivă	Cel puțin 32 biți	Rezultatul final al segmentării plus, eventual, paginării. Adresa finală obținută de către BIU, indicând în memoria fizică (hardware)

2.6.6. Adrese FAR și NEAR

Pentru a adresa o locație din memoria RAM sunt necesare două valori: una care să indice segmentul, alta care să indice offsetul în cadrul segmentului. Pentru a simplifica referirea la memorie, microprocesorul derivă, în lipsa unei alte specificări, adresa segmentului din **unul dintre registrii de segment CS, DS, SS sau ES**. Alegerea implicită a unui registru de segment se face după niște reguli proprii instrucțiunii folosite.

Prin definiție, o adresă în care se specifică doar offsetul, urmând ca segmentul să fie preluat implicit dintr-un registru de segment poartă numele de *adresă NEAR* (adresă apropiată). O adresă NEAR se află întotdeauna în interiorul unuia din cele patru segmente active.

O adresă în care programatorul indică explicit un selector de segment poartă numele de *adresă FAR* (adresă îndepărtată). O adresă FAR este deci o SPECIFICARE COMPLETA DE ADRESA și ea se poate exprima la nivelul unui program în trei moduri:

- $s_3s_2s_1s_0$: specificare_offset unde $s_3s_2s_1s_0$ este o constantă;
- registru_segment : specificare_offset, registru segment fiind CS, DS, SS, ES, FS sau GS;
- FAR [variabilă], unde variabilă este de tip QWORD și conține cei 6 octeți constituind adresa FAR. (ceea ce numim variabila pointer în limbajele de nivel înalt)

Formatul intern al unei adrese FAR este: la adresa mai mică se află offsetul, iar la adresa mai mare cu 4 (cuvântul care urmează după dublucuvântul curent) se află cuvântul ce conține selectorul care indică segmentul.

Reprezentarea adreselor respectă principiul reprezentării little-endian expus în capitolul 1, paragraf 1.3.2.3: partea cea mai puțin semnificativă are adresa cea mai mică, iar partea cea mai semnificativă are adresa cea mai mare.

2.6.7. Calculul offsetului unui operand. Moduri de adresare

În cadrul unei instrucțiuni 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 mașinii; mov eax, 17
- *modul imediat*, atunci când în instrucțiune se află chiar valoarea operandului (nu adresa lui și nici un registru în care să fie conținut); mov eax, 17
- *modul adresare la memorie*, dacă operandul se află efectiv undeva în memorie. În acest caz, offsetul lui se calculează după următoarea formulă:

$$\text{adresa_offset} = [\text{bază}] + [\text{index} \times \text{scală}] + [\text{constanta}]$$

Deci *adresa_offset* se obține din următoarele (maxim) patru elemente:

- conținutul unuia dintre registrii EAX, EBX, ECX, EDX, EBP, ESI, EDI sau ESP ca bază;
- conținutul unuia dintre registrii EAX, EBX, ECX, EDX, EBP, ESI sau EDI drept index;
- factor numeric (scală) pentru a înmulți valoarea registratorului index cu 1, 2, 4 sau 8
- valoarea unei constante numerice, pe octet, cuvant sau dublucuvânt.

De aici rezultă următoarele moduri de adresare la memorie:

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

Cele trei moduri de adresare a memoriei pot fi combinate. De exemplu, poate să apară adresare directă bazată, adresare bazată și scalat-indexată etc

Adresarea care NU este directă se numește **adresare indirectă** (bazată și/sau indexată). Deci o adresare indirectă este cea pt care avem specificat cel puțin un registru între parantezele drepte.

La instrucțiunile de salt mai apare și un alt tip de adresare numit adresare *relativă*.

Adresa relativă indică poziția următoarei instrucțiuni de executat, în raport cu poziția curentă. Poziția este indicată prin numărul de octeți de cod peste care se va sări. Arhitectura x86 permite atât adrese relative scurte (SHORT Address), reprezentate pe octet și având valori între -128 și 127, cât și adrese relative apropriate (NEAR Address), pe dublucuvânt cu valori între -2147483648 și 2147483647.

Jmp MaiJos ; aceasta instrucțiune se traduce (vezi OllyDbg) de obicei în Jmp [0084]↓

.....

.....

MaiJos:

Mov eax, ebx

Aritmetica de pointeri

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

Raspuns: Orice operație care are sens... aceasta însemnând orice operație ce exprimă ca rezultat o localizare în memorie corectă și utilă ca informație pt programator.

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

- adunări și scăderi de adrese ?
 - Adunare de adrese = ??? CE reprezinta ? NIMIC !!!!
 - Scădere de adrese = ??? CE reprezinta ? $q-p$ = nr **octeți** dintre cele două adrese de memorie (niciodată nu depăşim dim memoriei ; valoarea obținută este o CONSTANTA NUMERICA !!!!)
- adunări și scăderi de constante la o/dintr-o adresă = necesare și utile pt accesarea elementelor dintr-un array
- înmulțirea a două adrese ?- nepermisă (in majoritatea cazurilor valoarea obtinuta este dincolo de limita maxima a memoriei posibile a fi accesata).
- Inmultirea cu o constantă (in majoritatea cazurilor valoarea obtinuta este dincolo de limita maxima a memoriei posibile a fi accesata). In plus, CE reprezinta valoarea obtinuta ?... Nimic util !!
- Impărțire ?... No way !
 - Singura exceptie de la regulile aritmeticii de pointeri o constituie formula de calcul a offsetului unui operand unde sunt permise adunări de valori de registri (NU adunări de pointeri!!)... In rest nu exista excepții

$$a[7] = *(a+7) = *(7+a) = 7[a] \quad - \text{atât în C cât și în asamblare !}$$

Pointer arithmetic...? **DOAR 3 operații sunt permise cu POINTERI:**

1). Scăderea a două adrese

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

2) Adunarea unei constante numerice la o adresă

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

3). Scăderea unei constante numerice dintr-o adresă

Adresa – constanta numerică - $a[-4]$, $p-7$

- scăderea a 2 pointeri – valoare SCALARĂ !!! (valoare numerică constantă imediată)
- adunarea unei constante la un pointer → POINTER !
- scăderea unei constante dintr-un pointer → POINTER !

(ultimele două sunt utile pt referirea de elemente dintr-un array/zonă de memorie)

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

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

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 că a-b este o operație corectă cu pointeri !!!

[EBX+ECX*2 + v -7] – ok
SIB depl. const.

[EBX+ECX*2 + a-b-7]

SIB const.

mov eax, [EBX+ECX*2+(-7)] – ok.

L-value; R-value. Valoare stangă vs. valoare dreaptă a unei atribuirii.

Atribuire: $i := i + 1$ LHS vs. RHS

(Adresa lui I \leftarrow valoarea lui I + 1)

LHS(i) = adresa lui I := RHS(i) = (continutul de la adresa I) + 1

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

Sintaxele majorității limbajelor de programare prevăd că:

Symbol := expression_value, adică Identifier := expresie

In fapt, sunt limbaje (C++, ASAMBLARE !!!) care permit mai general sintaxa:

Expresie_calcul_de_adresa := expresie

(mov [ebx+2*EDX+v-7], a+2)

Dereferențierea (extragerea valorii de la o adresa) este implicită în 99% din limbaje. Exemplu exceptie – limbajul BLISS – unde dereferențierea trebuie menționată explicit intotdeauna; $i \leftarrow *i + 1$
(+ unele situații în Algol 68)

Symbol := expression_value (99% of the cases...)

Address_computation_Expression := expression_value

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

Variabilele “referință C++” (C++ reference variables) au 3 utilizări:

- 1) `Int& j = i; // j devine ALIAS pt i`
- 2) Transmiterea de variabile prin referință la apelul de subprograme
`float f(int&x, y);.....`
- 3) Returnarea de L-valori prin intermediul funcțiilor

`Int& f(x,i) {....return v[i];}` – Funcția f returnează o LHS (valoare stângă)
`F(a,7) = 79;` înseamnă că `v[7]=79 !!!`

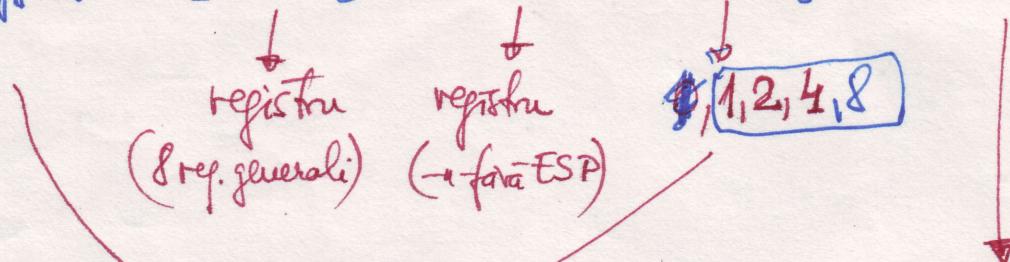
De asemenea, separat de acestea se permite și utilizarea operatorului condițional ternar pe post de valoare stângă:

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

Explanare

Formula de calcul a offsetului unui operand:

$$\text{adr-offset} = [\text{bază}] + [\text{index} * \text{scala}] + [\text{constantă}]$$



Adresare INDIRECTĂ!

Prin valoarea constantă determinată la momentul asamblării
(Adresare DIRECTĂ!)

<code>mov eax, [ebx]</code>	<code>mov eax, [2*edx - 7]</code>
<code>mov eax, EBX</code>	<code>mov eax, [edx + 4*edx + 18]</code>

`mov eax, 23` - ? respectă formula? Nu!!

`mov eax, [23]` - oh! dar se adresează memoria în OllyDbg....

`mov eax, ebx` - oh? respectă formula?

`mov edx, [ecx + 2*esp + 7]` → syntax error! ESP nu poate fi index!

`mov eax, [ebx * 2]`; oh!

`mov eax, [ebx * 3]`; oh.

`mov eax, [ebx + ebx * 2]`

`push dword [eax * 9 + 12]`

`[eax + eax * 8 + 12]`

`mov eax, [esp * 5]`
(syntax error!)

ESP nu poate fi index!

`mov eax, [ebp * 7]`
- syntax error!

`mov eax, [v]`; ?

mod registru?

mod immediat?

Mod adresare la memorie?

`mov eax, [ebx + 2]`
`mov eax, ebx + 2`

`mov eax, [ebx + edx]`
`mov eax, ebx + edx`

`mov eax,`

`[ebx - edx]`

invalid effective address.

ANALIZA CONCEPTULUI DE DEPASIRE (OVERFLOW)

CF (*Carry Flag*) este flagul de transport. Are valoarea 1 în cazul în care în cadrul ultimei operatii efectuate (UOE) s-a efectuat transport în afara domeniului de reprezentare a rezultatului și valoarea 0 în caz contrar. De exemplu, pt

1001 0011 +	147 +	93h +	-109 +
<u>0111 0011</u>	<u>115</u>	<u>73h</u>	<u>115</u>
1 0000 0110	262	106h	06
(fără semn)		(hexa)	
CF=1		(cu semn)	
		OF=0	

rezulta un transport de cifra semnificativa si
valoarea 1 este depusa automat in CF

flagul CF semnalează depășirea în cazul interpretării FĂRĂ SEMN (CF=1).

OF (*Overflow Flag*) este flag pentru depășire **CU SEMN**. Dacă rezultatul ultimei instrucțiuni în interpretarea CU SEMN a operanzilor nu a încăput în spațiul rezervat operanzilor (intervalul de reprezentare admisibil), atunci acest flag va avea valoarea 1, altfel va avea valoarea 0. Pentru exemplul de mai sus, OF=0.

Definiție (general, comprimată și incompletă). O *depășire* este o condiție/situatie matematică ce exprimă faptul că rezultatul unei operații nu a încăput în spațiul rezervat acestuia. (nici -147 nu “începe” în intervalul [-128..+127] și nici pe un byte însă e mai dificil de intuit că definiția cuprinde și se referă și la acest caz..)

Definiție mai exactă și completă. La nivelul procesorului și a limbajului de asamblare o *depășire* este o condiție/situatie matematică ce exprimă faptul că rezultatul UOE nu a încăput în spațiul rezervat acestuia SAU că acest rezultat nu aparține intervalului de reprezentare admisibil pe acea dimensiune de reprezentare SAU că operația efectuată este un nonsens matematic în respectiva interpretare (cu semn sau fără semn) și nu poate fi astfel acceptată drept o operație matematică corectă.

CF vs. OF. Conceptul de depășire.

$\begin{array}{r} 1001\ 0011 \\ + 1011\ 0011 \\ \hline 1\ 0100\ 0110 \end{array}$ <p>(reprezentare binară)</p>	$147 + 179 = 326$ <p>(interpretare fără semn)</p>	$\begin{array}{r} 93h \\ + B3h \\ \hline 1\ 46h \end{array}$ <p>(reprezentare hexa)</p>	$\begin{array}{r} -109 \\ - 77 \\ \hline - 186 \end{array} !!!!$ <p>(interpretare cu semn)</p>	<p>(asta ne-am aștepta să obținem ca rezultat și în program !!!)</p> <p>- adică 326 și -186 !</p>
--	---	---	--	---

- 326 și -186 sunt rezultatele corecte în baza 10 ale celor două interpretări ale OPERANZILOR binari de mai sus

Din discuția asupra intervalelor de reprezentare admisibile și respectiv din analiza tipului de probleme « Care este numărul minim de biți pe care se poate reprezenta... 326 și apoi respectiv -186 » va rezulta că

326 € [0..511] și -186 € [-256..+255] și că astfel numărul MINIM de biți pe care se pot reprezenta 326 și respectiv -186 este 9, iar reprezentarea lui -186 este: $512 - 186 = 326 = 1\ 46h = 1\ 0100\ 0110$

Ca urmare, TOATE operațiile de mai sus se desfășoară CORECT MATEMATIC pe 9 biți și operanze și rezultatele finale INCAP în spațiul rezervat, DACA operațiile se desfășoară pe 9 biți !!

Insă din păcate, adunarea de mai sus se desfășoară la nivel de procesor pe 8 biți (deoarece în limbaj de asamblare avem că ADD b+b → b) și ca urmare dptv MATEMATIC, aceasta NU se va desfășura corect pe 8 biți, nici 326 și nici -186 neîncăpând pe 1 octet !! (CARE NU se va desfășura corect pe 8 biți mai exact ? – base 2, base 10, base 16 ?... NICI UNA nu se va desfășura corect pe 8 biți !!! și de aceea CF și OF = 1 ambele...)

Acest lucru este semnalat SIMULTAN de către flag-urile CF (pt interpretarea fără semn) și respectiv OF (pt interpretarea CU semn), ambele flag-uri fiind setate la valoarea 1.

Ca urmare, ceea ce vom obține ca și rezultate și mai ales ca EFECTE pe 1 byte în program va fi :

1001 0011 +	147 +	93h +	-109 +
<u>1011 0011</u>	<u>179</u>	<u>B3h</u>	<u>- 77</u>
1 0100 0110	+70	1 46h	+ 70 !!!!
(fără semn)		(hexa) (cu semn)	
CF=1		OF=1	

(Deci nu obținem din păcate 326 și -186 aşa cum ar trebui corect matematic, ci +70 în ambele interpretări, adică operații incorecte matematic în ambele interpretări !!!!)

Prin setarea flag-urilor CF și OF la valoarea 1, procesorul ne « transmite mesajul » că ambele interpretări în baza 10 ale operației binare de adunare de mai sus sunt operații matematice incorecte !

0101 0011 +	83 +	53h +	83 +
<u>0111 0011</u>	<u>115</u>	<u>73h</u>	<u>115</u>
1100 0110	198	C6h	198 !!!!
(interpretarea fără semn a operanzilor)		(hexa) (interpretarea cu semn a operanzilor)	

- 198 este rezultatul corect în baza 10 pentru ambele interpretări ale OPERANZILOR binari din adunarea de mai sus, INSA trebuie să vedem acum dacă rezultatul începe pe 8 biți (DA – începe, de aceea vom avea CF=0) și respectiv dacă rezultatul operației binare în interpretarea cu semn este unul consistent cu corectitudinea operației matematice efectuate (NU este, deoarece 11000110 NU este un număr pozitiv în interpretarea CU semn !), deci ceea ce vom obține ca și rezultate pe 1 octet va fi :

0101 0011 +	83 +	53h +	83 +
<u>0111 0011</u>	<u>115</u>	<u>73h</u>	<u>115</u>
1100 0110	198	C6h	-58 !!!!
(fără semn)		(hexa) (cu semn)	
CF=0		OF=1	

Setarea CF=0 exprimă faptul că interpretarea fără semn în baza 10 a adunării în baza 2 de mai sus este o operație corectă, atât la nivelul interpretării OPERANZILOR cât și a REZULTATULUI. OF va fi însă setat de către procesor la valoarea 1, acest lucru însemnând că interpretarea cu semn în baza 10 a adunării în baza 2 de mai sus reflectă o operație incorrectă (mai exact REZULTATUL obținut este unul incorrect în interpretarea cu semn!)

OF va fi setat la valoarea 1 (*signed overflow*) dacă pentru operația de adunare ne aflăm în una din următoarele două situații (*regulile de depășire la adunare pentru interpretarea cu semn*). Sunt singurele două situații care provoacă depășire la adunare în interpretarea cu semn:

0.....+	sau	1.....+	(Semantic, cele două situații exprimă imposibilitatea acceptării matematice a celor
0.....		1.....	2 operații : nu putem aduna două numere pozitive și să obținem unul negativ și nici
-----		-----	nu putem aduna două numere negative și să obținem unul pozitiv).
1.....		0.....	

In cazul scăderii, avem de asemenea două reguli de depășire în interpretarea cu semn, consecințe a celor două reguli de la depășirea în cazul adunării :

1.....-	sau	0....-	(Semantic, cele două situații exprimă imposibilitatea acceptării matematice a celor 2
0.....		1.....	operații: nu putem scădea un număr pozitiv dintr-un nr.negativ și să obținem unul pozitiv
-----		-----	și nici nu putem scădea dintr-un nr pozitiv unul negativ și să obținem unul negativ).
0.....		1.....	

1 0110 0010 -	98 -	62h -	98 -
<u>1100 1000</u>	<u>200</u>	<u>C8h</u>	<u>-56</u>
1001 1010	-102	9Ah	154

(în interpretarea fără semn A OPERANZILOR) (hexa) (în interpretarea cu semn A OPERANZILOR)
 (REZULTATELE MATEMATICE CORECTE sunt precizate mai sus- ignorând INTERPRETAREA în vreun fel a configurației binare 1001 1010)

Insă, dacă luăm în considerare și **interpretările** REZULTATULUI obținut în program, avem din păcate :

1	0110 0010 -	98 -	62h -	98 -
	<u>1100 1000</u>	<u>200</u>	<u>C8h</u>	<u>-56</u>
	1001 1010	154	9Ah	-102 !!!!
	(fără semn)	pt efectuarea scăderii și de aceea valoarea 1 este depusa în CF	(hexa)	(cu semn)
	CF=1			OF=1

Ambele interpretări ale rezultatului obținut în baza 2 SUNT INCORECTE MATEMATIC, deci CF și OF vor fi ambele setate la valoarea 1.

Nici una dintre cele 2 interpretări nu este consistentă în baza 10 : 98-200 (interpretarea fără semn a scăderii) ar fi trebuit să furnizeze -102 ca rezultat matematic corect (valoarea aceasta fiind disponibilă doar în interpretarea cu semn !!), valoarea 154 nefiind în mod evident un rezultat corect ! Interpretarea CU SEMN furnizează $98 - (-56) = -102$ (rezultat evident incorrect !), deoarece $98 + 56 = 154$ (acesta ar fi trebuit să fie rezultatul corect, însă interpretarea 154 pt rezultat este valabilă doar în interpretarea fără semn). Ca urmare, se constată că pentru a fi corecte matematic rezultatele finale **ar fi trebuit să fie exact invers repartizate celor 2 interpretări**, însă nu este aşa, cele 2 operații matematice de mai sus (adică cele 2 interpretări asociate scăderii din baza 2) fiind ambele incorecte dpdV matematic. Ca urmare și ca reacție a mP 80x86 la această situație vom avea **CF=1** și respectiv **OF=1**.

Tehnic vorbind, microprocesorul setează OF=1 doar în una din cele 4 situații prezentate mai sus (2 situații pt adunare și respectiv 2 situații pt scădere) plus încă o situație pt înmulțire care va fi explicată în cele ce urmează.

Operatia de înmulțire NU furnizează depășire la nivelul arhitecturii 80x86, spațiul rezervat pt rezultat fiind suficient pentru ambele interpretări. Totuși, pt a nu rămâne neutilizate flag-urile CF și OF în cazul înmulțirii s-a luat decizia ca în cazul în care în cadrul operației de înmulțire dimensiunea rezultatului se întâmplă să fie identică cu cea a operanzilor ($b^*b = b$, $w^*w = w$ sau $d^*d = d$) flag-urile CF și OF să fie setate ambele la valoarea 0 (« no multiplication overflow », $CF = OF = 0$), iar dacă avem în mod real una dintre situațiile $b^*b = w$, $w^*w = d$, $d^*d = qword$, atunci $CF = OF = 1$ (« multiplication overflow »).

Cel mai grav efect al unei situații de depășire se manifestă în cazul împărțirii : în cazul acestei operații, dacă câtul obținut nu încape în spațiul rezervat (spațiul rezervat de către asamblor fiind byte pentru împărțire word/byte, word pentru împărțire doubleword/word și respectiv doubleword pentru împărțire quadword/doubleword) atunci se va semnala situație de « depășire la împărțire » cu efectul ‘Run-time error’ și cu emiterea din partea sistemului de operare a unuia dintre cele 3 mesaje echivalente : ‘Divide overflow’, ‘Division by zero’ sau ‘Zero divide’.

In cazul unei împărțiri care se efectuează corect, adică fără a se semnala depășire, CF și OF sunt nedefinite. Dacă avem însă depășire, programul « crapă », execuția lui se încheie, deci practic nu mai are nici un sens pentru nimeni să se întrebe ce valoare au la acel moment flag-urile CF și OF...

w/b → b 1002/3 = 334 = situație de depășire (overflow) în cazul împărțirii – fatal – Run time error (‘Divide overflow’, ‘Division by zero’ sau ‘Zero divide’ – oare DE CE se emite un astfel de mesaj care sugerează împărțirea la zero cu toate că aici am împărțit la 3 ??).

Pt.că dpdv al procesorului indiferent că împărțim la ZERO sau la ceva diferit de zero dar rezultatul NU ÎNCAPE în spațiul rezervat, situația în care se ajunge este ACEEAȘI : run-time error pe motiv că NU ÎNCAPE!!

334 nu încape pe un byte, iar INFINITE NU ÎNCAPE IN NIMIC !!!!!!!!

număr/0 – Zero divide = ESTE OVERFLOW PT CA INFINITE NU ÎNCAPE IN NIMIC !!!!!!!!

De ce am nevoie SIMULTAN de CF și OF in EFLAGS ?? Nu ajunge un singur flag pt a îmi arăta PE RAND daca am sau nu depășire fie în interpretarea cu semn fie în cea fără semn ? NU, pt că în momentul efectuării unei operații de adunare sau scădere în baza2 se efectuează de fapt SIMULTAN 2 operații în baza10: una în interpretarea cu semn și cealaltă în interpretarea fără semn.

În consecință e nevoie **SIMULTAN** de două flaguri diferite care să se ocupe fiecare **SEPARAT** de câte una din cele 2 interpretări posibile în baza 10:

- CF – pt interpretarea fără semn ; OF – pt interpretarea cu semn

Aceasta se întâmplă deoarece operația de adunare sau scădere exprimată IN BAZA 2 se efectuează IDENTIC, la fel deci, INDIFERENT DE INTERPRETAREA cu semn sau fără semn a operanzilor și a rezultatului !!!! Acesta este și motivul pt care în limbaj de asamblare **NU EXISTA IADD sau ISUB** ! Pt că ele și dacă ar exista NU ar funcționa diferit de ADD și respectiv SUB !

ADD = IADD, SUB = ISUB – Pt că în baza 2 operația exprimată se efectuează la fel INDIFERENT DE INTERPRETARE !!!

- DE CE AM NEVOIE DE IMUL și IDIV ??

Pentru că spre deosebire de adunare și scădere, care funcționează la fel în baza2 , indiferent de interpretare (cu semn sau fără semn) înmulțirea și împărțirea CU SEMN și FARA SEMN funcționează diferit în cazul cu semn comparativ cu cazul fără semn !!

Ca urmare la adunare și la scădere nu există necesitatea de a preciza ANTERIOR desfășurării lor cum dorim să fie interpretați operanții și rezultatul, deoarece cele 2 operații oricum funcționează la fel BINAR indiferent de cum dorim să le interpretăm. Este suficient să ne decidem ULTERIOR efectuării operației cum dorim să fie interpretați operanții și rezultatul.

In schimb înmulțirea și împărțirea NU funcționează BINAR la fel în cele 2 interpretări, aici existând necesitatea de a preciza ANTERIOR unei înmulțiri sau împărțiri cum dorim să fie interpretați operanții, iar acest lucru se face tocmai prin precizarea MUL și DIV (dacă dorim operanți fără semn) sau respectiv IMUL și IDIV (dacă dorim operanți cu semn).

CAPITOLUL 3

ELEMENTELE DE BAZA ALE LIMBAJULUI DE ASAMBLARE

Limbajul mașină al unui sistem de calcul (SC) este format din totalitatea instrucțiunilor mașină puse la dispoziție de procesorul SC. Acestea se reprezintă sub forma unor șiruri de biți cu semnificație prestabilită.

Limbajul de asamblare al unui calculator este un limbaj de programare în care setul de bază al instrucțiunilor coincide cu operațiile mașinii și ale cărui structuri de date coincid cu structurile primare de date ale mașinii. **Limbaj simbolic. Simboluri - Mnemonice + etichete.**

Elementele cu care lucrează un **asamblor** sunt:

- * **etichete** - nume scrise de utilizator, cu ajutorul cărora se pot referi date sau zone de memorie.
- * **instrucțiuni** - scrise sub forma unor mnemonice care sugerează acțiunea. Asamblorul generează octeți care codifică instrucțiunea respectivă.
- * **directive** - sunt indicații date asamblorului în scopul generării corecte a octetilor. Ex: relații între modulele obiect, definirea unor segmente, indicații de asamblare condiționată, directive de generare a datelor.
- * **contor de locații** - număr întreg gestionat de asamblor. În fiecare moment, valoarea contorului coincide cu numărul de octeți generați corespunzător instrucțiunilor și directivelor deja întâlnite în cadrul segmentului respectiv (deplasamentul curent în cadrul segmentului). Programatorul poate utiliza această valoare (accesare doar în citire!) prin simbolul '\$'. **Fiecare segment are propriul său contor de locații !!!**

NASM supports two special tokens in expressions, allowing calculations to involve the current assembly position: the \$ and \$\$ tokens. \$ evaluates to the assembly position at the beginning of the line containing the expression; so you can code an infinite loop using JMP \$.

\$\$ evaluates to the start of the current section; so you can tell how far into the section are by using (\$-\$).

Directiva SECTION

```
section .data  
db 'hello'  
db  'h', 'e', 'l','l','o'  
data_segment_size equ $$-$$
```

$\$-\$\$$ = the distance from the beginning of the segment AS A SCALAR (constant numerical value)!!!!!!

$\$$ - is an offset = POINTER TYPE !!!! It is an address !!!

$\$\$$ - is an offset =POINTER TYPE !!!! It is an address !!!

$\$$ means "address of here".

$\$\$$ means "address of start of current section".

So $\$-\$\$$ means "current size of section".

For the example above, this will be 10, as there are 10 bytes of data given.

Daca nu am definite sectiuni explicite , atunci $\$\$$ =inceputul segmentului respectiv

3.1. FORMATUL UNEI LINII SURSA

Formatul unei linii sursă în limbajul de asamblare x86 este următorul:

[etichetă[:]] [prefixe] [mnemonică] [operanzi] [;comentariu]

Ilustrăm conceptul prin intermediul a câteva exemple de linii sursă:

aici: jmp acolo ; avem etichetă + mnemonică + operand + comentariu
repz cmpsd ; prefix + mnemonică + comentariu

start: ; etichetă + comentariu
; doar un comentariu (care putea lipsi și el)
a dw 19872, 42h ; etichetă + mnemonică + 2 operanzi + comentariu

Caracterele din care poate fi constituită o *etichetă* sunt următoarele:

- Litere, atât A-Z cât și a-z;
- Cifre de la 0 la 9;
- Caractere _, \$, \$\$, #, @, ~, . și ?

Ca și prim caracter al unei etichete sunt permise doar litere, _ și ?

ACESTE REGULI SUNT VALABILE PENTRU TOȚI *IDENTIFICATORII* VALIZI (denumiri simbolice, precum nume de variabile, etichete, macro, etc).

Identifierii NASM sunt *case sensitive*, limbajul diferențiind literele mari de cele mici privitor la denumirile utilizator. Aceasta înseamnă că un identifier Abc este diferit de identifierul abc. Pentru denumirile implicit parte a limbajului, cum ar fi cuvintele cheie, mnemonicile și numele registrilor, nu se diferențiază literele mari de cele mici (acestea sunt case insensitive).

La nivelul limbajului de asamblare se întâlnesc două categorii de etichete:

- 1). **etichete de cod**, care apar în cadrul secvențelor de instrucțiuni cu scopul de a defini destinațiile de transfer ale controlului în cadrul unui program. **Pot apărea și în segmente de date !**
- 2). **etichete de date**, care identifică simbolic unele locații de memorie, din punct de vedere semantic ele fiind echivalentul noțiunii de *variabilă* din alte limbaje. **Pot apărea și în segmente de cod !**

Valoarea unei etichete în limbaj de asamblare este un număr întreg reprezentând adresa instrucțiunii, directivei sau datelor ce urmează etichetei.

Distincția dintre referirea adresei unei variabile sau a conținutului asociat acesteia în NASM se face după regulile:

- Când este specificat **între paranteze drepte, numele variabilei desemnează valoarea variabilei**, de exemplu [p] specifică accesarea valorii variabilei p, similar cu modul în care *p semnifică dereferențierea unui pointer (accesul la conținutul indicat prin valoarea pointerului) în C;
- În orice alt context **numele variabilei reprezintă adresa variabilei**, spre exemplu, p este întotdeauna adresa variabilei p;

Exemple:

mov EAX, et ; încarcă în registrul EAX **adresa** datelor sau a codului marcat cu eticheta et (4 octeți)
mov EAX, [et] ; încarcă în registrul EAX **conținutul** de la adresa et (4 octeți)
lea eax, [v] ; încarcă în registrul eax **adresa** (offsetul) variabilei v (4 octeți)

Ca generalizare, **folosirea parantezelor pătrate indică întotdeauna accesarea unui operand din memorie**. De exemplu, mov EAX, [EBX] semnifică un transfer în EAX a conținutului memoriei a cărei adresă este dată de valoarea lui EBX.

Există două tipuri de *mnemonice*: mnemonice de *instrucțiuni* și nume de *directive*. **Directivele dirijează asamblorul**. Ele specifică modul în care asamblorul va genera codul obiect. **Instrucțiunile dirijează procesorul**.

Operanții sunt parametri care definesc valorile ce vor fi prelucrate de instrucțiuni sau de directive. Ei pot fi **registri, constante, etichete, expresii, cuvinte cheie sau alte simboluri**. Semnificația operanților depinde de mnemonica instrucțiunii sau directivei asociate.

3.2. EXPRESII

expresie - operanzi + operatori. *Operatorii* indică modul de combinare a operanzilor în scopul formării expresiei. **Expresiile sunt evaluate în momentul asamblării** (adică, valorile lor sunt determinabile la momentul asamblării, cu excepția acelor părți care desemnează conținuturi de regiștri și care vor fi determinate la execuție).

3.2.1. Moduri de adresare

Operanzii instrucțiunilor pot fi specificați sub forme numite *moduri de adresare*. Cele trei tipuri de operanzi sunt: **operaṇzi imediați, operaṇzi registru și operaṇzi în memorie**.

Valoarea operanzilor este calculată în momentul asamblării pentru operanzii imediați și pentru offset-urile reprezentând adresarea directă, în momentul încărcării programului pentru adresarea directă (**adresa FAR**) și în momentul execuției pentru operanzii registru și cei adresați indirect.

??:offset (assembly time) 0708:offset (loading time)

3.2.1.1. Utilizarea operaṇzilor imediați

Operaṇzi imediați sunt formați din date numerice constante calculabile la momentul asamblării.

Constantele întregi se specifică prin valori binare, octale, zecimale sau hexazecimale. Adițional, este permisă folosirea caracterului _ (underscore) pentru a separa grupuri de cifre. Baza de numerație poate fi precizată în mai multe moduri:

- Folosind sufixele H sau X pentru hexazecimal, D sau T pentru zecimal, Q sau O pentru octal și B sau Y pentru binar; în aceste cazuri numărul trebuie să înceapă obligatoriu cu o cifră între 0 și 9, pentru a nu exista confuzii între constante și simboluri, de exemplu, OABCH este interpretat ca număr hexazecimal, dar ABCH este interpretat ca simbol
- În stil C, prin prefixare cu 0x sau 0h pentru hexazecimal, 0d sau 0t pentru zecimal, 0o sau 0q pentru octal, respectiv 0b sau 0y pentru binar;

Exemple:

- constanta hexazecimală B2A poate fi exprimată ca 0xb2a, 0xb2A, 0hb2a, 0b12Ah, 0B12AH, etc;
- valoarea zecimală 123 poate fi specificată ca 123, 0d123, 0d0123, 123d, 123D, ...
- 11001000b, 0b11001000, 0y1100_1000, 001100_1000Y reprezintă diferite exprimări ale nr bin 11001000

Deplasamentele etichetelor de date și de cod reprezinta valori determinabile la momentul asamblării care rămân constante pe tot parcursul execuției programului

mov eax, et ; transfer în registrul EAX a adresei (offsetului) asociate etichetei et
va putea fi evaluată la momentul asamblării drept de exemplu

mov eax, 8 ; distanță de 8 octeți față de începutul segmentului de date

“Constanța” acestor valori derivă din regulile de alocare adoptate de limbajele de programare în general și care statuează că ordinea de alocare în memorie a variabilelor declarate (mai precis distanța față de începutul segmentului de date în care o variabilă este alocată) sau respectiv distanțele salturilor destinație în cazul unor instrucțiuni de tip **goto** sunt valori constante pe parcursul execuției unui program.

Adică: o variabilă odată alocată în cadrul unui segment de memorie nu își va schimba niciodată locul alocării (adică poziția sa față de începutul acelui segment) iar această informație determinabilă la momentul asamblării derivă din ordinea specificării variabilelor la declarare în cadrul textului sursă și din dimensiunea de reprezentare dedusă pe bază informației de tip asociate.

3.2.1.2. Utilizarea operanzilor registru

Modul de *invocare/accesare directă* - **mov eax, ebx**

Invocare/accesare *indirectă* - pentru a indica locațiile de memorie - **mov eax, [ebx]**

3.2.1.3. Utilizarea operanzilor din memorie

Operanții din memorie : cu *adresare directă* și cu *adresare indirectă*.

Operandul cu *adresare directă* este o constantă sau un simbol care reprezintă adresa (segment și deplasament) unei instrucțiuni sau a unor date. Acești operanzi pot fi *etichete* (de ex: jmp et, var1 dw 324, add eax,[b]), *nume de proceduri* (de ex: call proc1) sau *valoarea contorului de locații* (de ex: b db \$-a).

Deplasamentul unui operand cu adresare directă este calculat în momentul asamblării (assembly time). Adresa fiecărui operand raportată la structura programului executabil (mai precis stabilirea segmentelor la care se raportează deplasamentele calculate) este calculată ***în momentul editării de legături*** (linking time). Adresa fizică efectivă este calculată ***în momentul încărcării programului pentru execuție*** (loading time – acest proces final de ajustare a adreselor numindu-se RELOCAREA ADRESELOR = Address Relocation).

Un deplasament utilizat ca operand în cadrul unui program este întotdeauna raportat la un registru de segment. Acest registru poate fi specificat explicit sau, în caz contrar, se asociază de către asamblor în mod implicit un registru de segment. Regulile limbajului de asamblare pentru asocierile implice sunt:

- **CS** pentru etichete de cod destinație ale unor salturi (jmp, call, ret, jz etc);
- **SS** în adresări SIB ce foloseste EBP sau ESP drept bază (indiferent de index sau scală);
- **DS** pentru restul accesărilor de date

Specificarea explicită a unui registru de segment se face cu ajutorul operatorului de prefixare segment (notat ":" și care se mai numește, 'operatorul de specificare a segmentului'). ES poate fi utilizat numai în specificări explicate (ca de exemplu ES:[Var] sau ES:[ebx+eax*2-a]) sau în cadrul unor instrucțiuni pe siruri (MOVSB)

JMP FAR CS:...

JMP FAR DS:... or JMP FAR [label2]

3.2.1.4. Operanzi cu adresare indirectă

Operanzii cu *adresare indirectă* utilizează regiștri pentru a indica adrese din memorie. Deoarece valorile din regiștri se pot modifica la momentul execuției, adresarea indirectă este indicată pentru a opera în mod dinamic asupra datelor.

Forma generală pentru accesarea indirectă a unui operand de memorie este dată de formula de calcul a offset-ului unui operand:

$$[\text{registru_de_bază}] + [\text{registru_index} * \text{scală}] + [\text{constantă}]$$

Constanta este o expresie a cărei valoare este determinabilă la momentul asamblării. De exemplu, $[\text{ebx} + \text{edi} + \text{table} + 6]$ desemnează un operand prin adresare indirectă, unde atât *table* cât și 6 sunt constante.

Operanzii *registru_de_bază* și *registru_index* sunt folosiți de obicei pentru a indica o adresă de memorie referitoare la un tablou. În combinație cu factorul de scalare, mecanismul este suficient de flexibil pentru a permite acces direct la elementele unui tablou de înregistrări, cu condiția ca dimensiunea în octeți a unei înregistrări să fie 1, 2, 4 sau 8. De exemplu, octetul superior al elementului de tip DWORD cu index dat în ecx, parte a unui vector de înregistrări al cărui adresă (a vectorului) este în edx poate fi încărcat în dh prin intermediul instrucțiunii

```
mov dh, [edx + ecx * 4 + 3]
```

Din punct de vedere sintactic, atunci când operandul nu este specificat prin formula completă, lipsind unele dintre componente (de exemplu lipsește “* scală”), asamblorul va rezolva ambiguitatea care rezultă printr-un proces de analiză a tuturor formele echivalente de codificare posibile și alegerea celei mai scurte dintre acestea. Cu alte cuvinte, având

```
push dword [eax + ebx] ; salvează pe stivă dublucuvântul de la adresa eax+ebx
```

asamblorul are libertatea de a considera eax drept bază și ebx drept index sau invers, ebx drept bază și eax drept index. Analog, pentru

```
pop DWORD [ecx] ; restaurează vârful stivei în variabila cu adresa dată de ecx
```

asamblorul poate interpreta ecx fie ca bază fie ca index. Ce este realmente important de reținut este faptul că toate codificările luate în considerare de către asamblor sunt echivalente iar decizia finală a asamblorului nu are impact asupra funcționalității codului rezultat.

De asemenea, în plus față de rezolvarea unor astfel de ambiguități, asamblorul permite și exprimări non-standard cu condiția ca acestea să fie transformabile într-un final în forma standard de mai sus.

Alte exemple:

```
lea eax, [eax*2] ; încarcă în eax valoarea lui eax*2 (adică, eax devine 2*eax)
```

În acest caz, asamblorul poate decide între codificare de tip bază = eax + index = eax și scală = 1 sau index = eax și scală = 2.

```
lea eax, [eax*9 + 12] ; eax ia valoarea eax * 9 + 12
```

Deși scală nu poate fi 9, asamblorul nu va emite aici un mesaj de eroare. Aceasta deoarece el va observa posibila codificare a adresei drept: bază=eax + index=eax cu scală=8, unde de această dată valoarea 8 este corectă pentru scală. Evident, instrucțiunea putea fi precizată mai clar sub forma

```
lea eax, [eax + eax * 8 + 12].
```

Să reținem deci că pentru adresarea indirectă, esențială este specificarea între paranteze drepte a cel puțin uneia dintre elementele componente ale formulei de calcul a offsetului.

3.2.2. Utilizarea operatorilor

Operatori - pentru combinarea, compararea, modificarea și analiza operanzilor. Unii operatori lucrează cu constante întregi, alții cu valori întregi memorate, iar alții cu ambele tipuri de operanzi.

Este importantă înțelegerea diferenței dintre operatori și instrucțiuni. **Operatorii efectuează calcule cu valori constante SCALARE determinabile la momentul asamblării** (valori scalare = valori imediate), **cu excepția adunării și scaderii unei constantă la un/dintr-un pointer (care va furniza întotdeauna “pointer data type”)** și **cu excepția formulei de calcul al offset-ului unui operand (care permite operatorul ‘+’)**. Instrucțiunile efectuează calcule cu valori ce pot fi necunoscute până în momentul execuției. Operatorul de adunare (+) efectuează adunarea în momentul asamblării; instrucțiunea ADD efectuează adunarea în timpul execuției.

Operatorii disponibili pentru construcția expresiilor sunt asemănători celor din limbajul C, atât ca sintaxă cât și din punct de vedere semantic. **Evaluarea expresiilor numerice se face pe 64 de biți**, rezultatele finale fiind ulterior ajustate în conformitate cu dimensiunea de reprezentare disponibilă în contextul de utilizare al expresiei.

În tabelul de mai jos sunt prezentate în ordinea priorității operatorii ce pot fi folosiți în cadrul expresiilor limbajului de asamblare x86 (NASM !!).

Prioritate	Operator	Tip	Rezultat
7	-	Unar, prefixat	Complement față de 2 (negare): $-X = 0 - X$
7	+	Unar, prefixat	Fără efect (oferit pentru simetrie cu „-“): $+X = X$
7	~	Unar, prefixat	Complement față de 1: $\text{mov al, } \sim 0 \Rightarrow \text{mov AL, } 0\text{xFF}$
7	!	Unar, prefixat	Negare logică: $!X = 0$ când $X \neq 0$, altfel 1
6	*	Binar, infix	Înmulțire: $1 * 2 * 3 = 6$
6	/	Binar, infix	Câțul împărțirii fără semn: $24 / 4 / 2 = 3$ ($-24/4/2 = 0\text{FDh}$)
6	//	Binar, infix	Câțul împărțirii cu semn: $-24 // 4 // 2 = -3$ ($-24 / 4 / 2 \neq -3!$)
6	%	Binar, infix	Restul împărțirii fără semn: $123 \% 100 \% 5 = 3$
6	%%	Binar, infix	Restul împărțirii cu semn: $-123 \% \% 100 \% \% 5 = -3$
5	+	Binar, infix	Însumare: $1 + 2 = 3$
5	-	Binar, infix	Scădere: $1 - 2 = -1$

4	<code><<</code>	Binar, infix	Deplasare pe biți către stânga: $1 << 4 = 16$
4	<code>>></code>	Binar, infix	Deplasare pe biți la dreapta: $0xFE >> 4 = 0x0F$
3	<code>&</code>	Binar, infix	$\$I: 0xF00F \& 0xFF6 = 0x0006$
2	<code>^</code>	Binar, infix	SAU exclusiv: $0xFF0F ^ 0xFF = 0x0FF0$
1	<code> </code>	Binar, infix	SAU: $1 2 = 3$

Operatorul de indexare are o utilizare largă în specificarea operanzilor din memorie adresați indirect. Paragraful 3.2.1 a clarificat rolul operatorului `[]` în adresarea indirectă.

Acest tabel este de o importanță covârșitoare în anumite situații și e bine să îl avem „la îndemână”. Iată de ce:

$5|6+7\&8 = (5|6)+(7\&8) = 7+0 = 7 ??$ NU !!!!!! datorită precedenței operatorilor avem;

$5|6+7\&8 = 5|(6+7)\&8 = 5|13\&8 = 5|8=13 = 0Dh !!!$

3.2.2.3. Operatori de deplasare de biți

expresie >> cu_cât și *expresie << cu_cât*

mov ah, 01110111b << 3 ; desemnează valoarea 10111000b
add bh, 01110111b >> 3 ; desemnează valoarea 00001110b

AX	00000011 10111000	!!!!
BX	000000000 00001110b	

3.2.2.4. Operatori logici pe biți

Operatorii pe biți efectuează operații logice la nivelul fiecărui bit al operandului (operanzilor) unei expresii. Expresiile au ca rezultat valori constante.

OPERATOR	SINTAXA	SEMNIFICATIE
~	~ expresie	complementare biți
&	expr1 & expr2	ȘI bit cu bit
	expr1 expr2	SAU bit cu bit
^	expr1 ^ expr2	SAU exclusiv bit cu bit

Exemple (presupunem că expresia se reprezintă pe un octet):

~ 11110000b ; desemnează valoarea 00001111b ; ~0f0h = ~...
 01010101b & 11110000b ; are ca rezultat valoarea 01010000b
 01010101b | 11110000b ; are ca rezultat valoarea 11110101b
 01010101b ^ 11110000b ; are ca rezultat valoarea 10100101b
 ! – negare logică (similar cu limbajul C) ; !0 = 1 ; !(orice diferit de zero) = 0

3.2.2.6. Operatorul de specificare a segmentului

Operatorul de specificare a segmentului (:) comandă calcularea adresei FAR a unei variabile sau etichete în funcție de un anumit segment. Sintaxa este: ***segment:expresie***

[ss: ebx+4] ; deplasamentul e relativ la SS // [es:082h] ; deplasamentul e relativ la ES
 10h:var ; segmentul este indicat de selectorul 10h, iar offsetul este valoarea etichetei var.

3.2.2.7. Operatori de tip

Specifică tipurile unor expresii și a unor operanzi păstrați în memorie. Sintaxa pentru aceștia este

tip expresie

unde specificatorul de tip este unul dintre cuvintele cheie **BYTE, WORD, DWORD, QWORD**.

Această construcție sintactică forțează ca *expresie* să fie tratată ca având dimensiunea de reprezentare *tip*, fără însă a-i modifica definitiv (destructiv) valoarea în sensul precizat de conversia dorită. De aceea, aceștia sunt considerați **operatori de conversie (temporară) nedestructivă**. Pentru operanzii păstrați în memorie, *tip* poate fi **BYTE, WORD, DWORD, QWORD** având dimensiunile de reprezentare 1, 2, 4, 8 octeți. Pentru etichetele de cod el poate fi **NEAR** (adresă pe 4 octeți) sau **FAR** (adresă pe 6 octeți). Expresia **byte [A]** va indica doar primul octet de la adresa indicată de A. Analog, **dword [A]** indică dublucuvântul ce începe la adresa A.

Specificatorii BYTE / WORD / DWORD / QWORD au intotdeauna doar rol de a clarifica o ambiguitate (inclusiv cand este vorba despre o variabilă de memorie, faptul de a preciza **mov BYTE [v], 0** sau **mov WORD [v], 0** este tot o clarificare a ambiguității, cum nasm nu asociază faptul că v este byte/ word / dword).

mov [v],0 ; syntax error – operation size not specified

Specificatorul QWORD nu intervene niciodată explicit în cod pe 32 de biți.

Exemple unde e necesar un specificator de dimensiune al operanzilor:

- **mov [mem], 12**

- **(i)div [mem] ; (i)mul [mem]**

- **push [mem] ; pop [mem]**

- **push 15** - aici este o inconsistență în NASM, asamblorul nu va emite eroare/warning ci va face
- **push DWORD 15**

Exemple de operanzi IMPLICITI efectiv pe 64 biți (în cod pe 32):

- **mul dword [v]** ; înmulțește eax cu dword-ul de la adresa v și depune în EDX:EAX rezultatul
- **div dword [v]** ; împărțire EDX:EAX la v

3.3. DIRECTIVE

Directivele indică modul în care sunt generate codul și datele în momentul asamblării.

3.3.1.1. Directiva SEGMENT

Directiva SEGMENT permite direcționarea octetilor de cod sau date emiși de către un asamblor înspre segmentul precizat, segment care poartă un nume și are asociate diverse caracteristici.

SEGMENT *nume* [*tip*] [*ALIGN=aliniere*] [*combinare*] [*utilizare*] [*CLASS=clasă*]

Numelui segmentului i se asociază ca valoare adresa de segment (32 biți) corespunzătoare poziției segmentului în memorie în faza de execuție. În acest sens, asamblorul NASM pune la dispoziție și simbolul special \$\$ care este echivalent cu adresa segmentului curent, acesta având însă avantajul că poate fi utilizat în orice context, fără a fi necesar să fie cunoscut numele segmentului în care ne aflăm.

Cu excepția numelui, toate celelalte câmpuri sunt opționale atât din punct de vedere a prezenței cât și a ordinii în care sunt specificate.

Argumentele opționale *tip*, *aliniere*, *combinare*, *utilizare* și 'clasa' dau editorului de legături și asamblorului indicații referitoare la modul de încărcare și atributele segmentelor.

Tip permite selectarea unui model de folosire al segmentului, având la dispoziție următoarele opțiuni:

- **code** (sau **text**) - segmentul va conține cod, conținutul nu poate fi scris dar se poate citi sau executa

- **data** (sau **bss**) - segment de date permitând citire și scriere însă nu și execuție (valoare implicită)
- **rdata** - segment din care se poate doar citi, menit să conțină definiții de date constante

Argumentul opțional *aliniere* specifică multiplul numărului de octeți la care trebuie să înceapă segmentul respectiv. Alinierile acceptate sunt puteri a lui 2, între 1 și 4096.

Dacă argumentul *aliniere* lipsește, atunci se consideră implicit că este vorba despre o aliniere ALIGN=1, adică segmentul poate începe la orice adresă.

Argumentul opțional *combinare* controlează modul în care segmente cu același nume din cadrul altor module vor fi combinate cu segmentul în cauză la momentul editării de legături. Valorile posibile sunt:

- **PUBLIC** - indică editorului de legături să concateneze acest segment cu alte eventuale segmente cu același nume, obținându-se un unic segment a cărei lungime este suma lungimilor segmentelor componente.
- **COMMON** - specifică faptul că începutul acestui segment trebuie să se suprapună peste începutul tuturor segmentelor ce au același nume. Se obține un segment având dimensiunea egală cu cea a celui mai mare segment având același nume.
- **PRIVATE** - indică editorului de legături că acest segment nu este permis a fi combinat cu altele care poartă același nume.
- **STACK** - segmentele cu același nume vor fi concatenate. În faza de execuție segmentul rezultat va fi segmentul stivă.

Implicit, dacă nu se specifică o metodă de combinare, orice segment este considerat PUBLIC.

Argumentul *utilizare* permite optarea pentru altă dimensiune de cuvânt decât cea de 16 biți, care este implicită în lipsa precizării acestui argument.

Argumentul '*clasa*' are rolul de a permite stabilirea ordinii în care editorul de legături plasează segmentele în memorie. Toate segmentele având aceeași clasă vor fi plasate într-un bloc contiguu de memorie indiferent de

ordinea lor în cadrul codului sursă. Nu există o valoare implicită de initializare pentru acest argument, el fiind nedefinit în lipsa specificării, ducând în consecință la evitarea concatenării într-un bloc continuu a tuturor segmentelor definite astfel.

```
segment code use32 class=CODE      segment data use32 class=DATA
```

3.3.2. Directive pentru definirea datelor

definire date = declarare (specificarea atributelor) + *alocare* (rezervarea sp. de memorie necesar).

(UNICA !!!)

(NU e unica !!!)

(UNICA !!!!)

In C – 17 module (fisiere separate) ; A1- variabilă globală (int A1 + 16 declarații de data extern int A1)
LINKER-ul este responsabil pt verificarea DEPENDENTELOR dintre module !

Structura unei Variable = ([nume], set_de_atribute, [adresa/referință, valoare])

Variabilele dinamice NU AU NUME !!!!

P=new(...); p=malloc(...); ...free... (Diferența between POINTER and DYNAMIC variables !!!!)

(Nume, set_de_atribute) = parametrii formali ai funcțiilor și procedurilor !!!

Set_de_atribute = (TD, Domeniu_de_vizibilitate (scope), durata de viață (lifetime/extent), clasa de memorie)
Memory class (in C) = (auto, register, static, extern)

tipul de dată = dimensiunea de reprezentare – octet, cuvânt, dublucuvânt, quadword

Forma generală a unei linii sursă în cazul unei declarații de date este:

[*nume*] *tip_data lista_expresii* [*;comentariu*]

sau [*nume*] *tip_alocare factor* [*;comentariu*]

sau [*nume*] **TIMES** *factor tip_data lista_expresii* [*;comentariu*]

unde *nume* este o etichetă prin care va fi referită data. Tipul rezultă din tipul datei (dimensiunea de reprezentare) iar valoarea este adresa la care se va găsi în memorie primul octet rezervat pentru data etichetată cu numele respectiv.

factor este un număr care indică de câte ori se repetă lista de expresii care urmează în paranteză.

Tip_data este o directive de definire a datelor, una din următoarele:

DB - date de tip octet (BYTE)

DW - date de tip cuvânt (WORD)

DD - date de tip dublucuvânt (DWORD)

DQ - date de tip 8 octeți (QWORD - 64 biți)

DT - date de tip 10 octeți (TWORD - utilizate pentru memorarea constantelor BCD sau constantelor reale de precizie extinsă)

De exemplu, secvența urmatoare definește și initializează cinci variabile de memorie:

segment data

```
var1 DB  'd'    ;1 octet
      .a DW  101b ;2 octeți
var2 DD  2bfh ;4 octeți
      .a DQ  307o ;8 octeți (1 quadword)
      .b DT  100   ;10 octeți
```

Variabilele var1 și var2 sunt definite folosind etichete obișnuite, cu vizibilitate la nivelul întregului cod sursă, în timp ce .a și .b sunt etichete locale, accesul la aceste variabile impunând următoarele constrângeri:

- acestea se pot accesa cu numele local, adică .a sau .b, până în momentul definirii unei alte etichete obișnuite (ele fiind locale etichetei ce le preced);
- pot fi accesate de oriunde prin numele lor complet: var1.a, var2.a sau var2.b.

Valoarea de inițializare poate fi și o expresie, ca de exemplu în
vartest DW (1002/4+1)

După o directivă de definire a datelor pot să apară mai multe valori, permitându-se astfel declararea și inițializarea de tablouri. De exemplu, declarația Tablou DW 1,2,3,4,5

crează un tablou de 5 întregi reprezentați pe cuvinte având valorile respectiv 1,2,3,4,5. Dacă valorile de după directivă nu încap pe o singură linie se pot adăuga oricâte linii este necesar, linii ce vor conține numai directiva și valorile dorite. Exemplu:

Tabpatrate DD 0, 1, 4, 9, 16, 25, 36
DD 49, 64, 81
DD 100, 121, 144, 169

Tip_alocare este o directivă de rezervare de date neinițializate:

- RESB** - date de tip octet (BYTE)
- RESW** - date de tip cuvânt (WORD)
- RESD** - date de tip dublucuvânt (DWORD)
- RESQ** - date de tip 8 octeți (QWORD - 64 biți)
- REST** - date de tip 10 octeți (TWORD – 80 biți)

factor este un număr care indică de câte ori se repetă tipul alocării precizate

De exemplu Tabzero RESW 100h rezervă 256 de cuvinte pentru tabloul *Tabzero*

NASM does not support the MASM/TASM syntax of reserving uninitialized space by writing DW ? or similar things: this is what it does instead. The operand to a RESB-type pseudo-instruction is a **critical expression** (toți operanții care intervin în calcul trebuie să fie cunoscute în momentul în care expresia este întâlnită). Ex:

```
buffer:    resb  64   ; reserve 64 bytes
wordvar:   resw  1    ; reserve a word
realarray: resq  10   ; array of ten reals
```

Directiva TIMES permite asamblarea repetată a unei instrucțiuni sau definiții de date:

TIMES *factor tip_data expresie*

De exemplu Tabchar TIMES 80 DB ‘a’

crează un tablou de 80 de octeți inițializați fiecare cu codul ASCII al caracterului 'a'.

matrice10x10 times 10*10 dd 0

va furniza 100 de dublucuvinte dispuse continuu în memorie începând de la adresa asociată etichetei matrice10x10.

TIMES can also be applied to instructions:

TIMES *factor instrucțiune*

TIMES 32 add eax, edx ; having as effect EAX = EAX + 32*EDX

3.3.3. Directiva EQU

Directiva EQU permite atribuirea, în faza de asamblare, unei valori numerice sau sir de caractere unei etichete fără alocarea de spațiu de memorie sau generare de octeți. Sintaxa directivei EQU este

nume EQU expresie

Exemple:

END_OF_DATA	EQU '!"
BUFFER_SIZE	EQU 1000h
INDEX_START	EQU (1000/4 + 2)
VAR_CICLARE	EQU i

Prin utilizarea de astfel de echivalări textul sursă poate deveni mai lizibil. Se observă asemănarea etichetelor echivalate prin directiva EQU cu constantele din limbajele de programare de nivel înalt.

Expresia pentru echivalarea unei etichete definite prin directiva EQU poate conține la rândul ei etichete definite prin EQU:

TABLE_OFFSET	EQU 1000h
INDEX_START	EQU (TABLE_OFFSET + 2)
DICTIONAR_STAR	EQU (TABLE_OFFSET + 100h)

Contor de locatii

Segment data

a db 17, -2, 0ffh, 'xyz',...

db

db....

;lga db \$-a (mov [lga],....); ok //aritmetica de pointeri – scăderea a 2 pointeri = scalar (constanta numerică) - lga=variabila de memorie (mov [lga],...)

;lga dw \$-\$; Corect numai DACA a este primul element definit în data segment !!!!

;lga EQU \$-a ; ok ! însa mov [lga],... este syntax error !!! pt ca lga NU este o variabilă alocată... ci o constantă! (fără alocare de memorie)

;lga dw \$-data ; corect in TASM/MASM, INCORECT in NASM sub 32 biți !!! syntax error – “Expression is not simple or relocatable”

;lga dw lga-a !!!!!!!

b EQU 27 ; b NU este un offset !!!!

c dd 12345678h

;lga dw b-a ; syntax error !!!!! b is NOT an address !!!

;lga dw c-a ; ok !!!!

lga dw \$-a-4 ; ok !!!

lg dw \$-a ; length (a) + 4 !!!

Dacă nu se utilizează nici o directivă section în mod explicit, simbolul \$\$ se va evalua implicit la offset-ul începutului de segment.

Elementul sintatic ":" se pune obligatoriu când definim etichete de cod (ex: "start:") însă nu trebuie pus dacă definim o etichetă de date (ex: definirea de variabile "a db 17")

Exemplu ilustrând regulile implicate de prefixare ale unui offset cu segmentul aferent

Mov eax, [v] ; mov eax, DWORD PTR DS:[405000]

Mov eax, [ebx] ; mov eax, DWORD PTR DS:[ebx]

Mov eax, [ebp] ; mov eax, DWORD PTR SS:[ebp]

Mov eax, [ebp*2] ; mov eax, DWORD PTR SS:[ebp+ebp]

Mov eax, [ebp*3] ; mov eax, DWORD PTR SS:[ebp+ebp*2]

Mov eax, [ebp*4] ; mov eax, DWORD PTR DS:[ebp*4]

Mov eax, [ebx+esp] ; eax \leftarrow dword care incepe la adresa [SS:esp+ebx]

Mov eax, [esp + ebx] ; eax \leftarrow dword care incepe la adresa [SS:esp+ebx]

Mov eax, [ebx+esp*2] ; syntax error – ESP nu poate fi index !

Mov eax, [ebx+ebp*2] ; eax \leftarrow dword care incepe la adresa [DS:ebx + 2*ebp]

Mov eax, [ebx+ebp] ; EAX \leftarrow ...DS:...

Mov eax, [ebp+ebx] ;SS:.....

Mov eax, [ebx*2+ebp] ;SS:....

Mov eax, [ebx*1+ebp] ;... SS:....

Mov eax, [ebp*1+ebx] ; ...DS:....

Mov eax, [ebx*1+ebp*1] ; SS... ; Primul gasit cu * e considerat index ! EBP - baza

Mov eax, [ebp*1+ebx*1] ; DS....; Primul gasit cu * e considerat index! EBX - baza

Mov eax, [ebp*1+ebx*2]; ...SS...

Operații și operatori pe biți

In computer programming, a bitwise operation operates on a bit string, a bit array or a binary numeral at the level of its individual bits. It is a fast and simple action, basic to the higher-level arithmetic operations and directly supported by the processor.

Atenție la diferența dintre operatori și instrucțiuni !!

Mov ah, 01110111b << 3 ; AH :=10111000b

Vs.

Mov ah, 01110111b

Shl ah, 3

In descrierile de mai jos x reprezintă UN BIT, 0 și 1 reprezintă valori de bit iar $\sim x$ reprezintă valoarea complementară a bitului de valoare x. Secvențele descriptive de mai jos exemplifică modul de acțiune al **operațiilor AND, OR și XOR LA NIVEL DE BIT** ca și MECANISM de acțiune, indiferent dacă operația respectivă este declanșată la nivel de cod sursă de OPERATORUL respectiv sau de INSTRUCTIUNEA corespondentă.

& - operatorul SI bit cu bit

AND – instrucțiune

x AND 0 = 0

x AND 1 = x

; x AND x = x

; x AND $\sim x$ = 0

Operație utilă pt forțarea valorii anumitor biți la 0 !!!!

| - operatorul SAU bit cu bit

OR – instrucțiune

x OR 0 = x

x OR 1 = 1

; x OR x = x

; x OR $\sim x$ = 1

Operație utilă pt forțarea valorii anumitor biți la 1 !!!!

\wedge - op. SAU EXCLUSIV bit cu bit;

XOR – instrucțiune

x XOR 0 = x

x XOR 1 = $\sim x$

x XOR x = 0

x XOR $\sim x$ = 1

Operație utilă pt complementarea valorii anumitor biți !!!

XOR ax, ax ; AX=0 !!!

Utilizarea operatorilor ! si ~

- ! Negare logică: !X = 0 când X ≠ 0, altfel 1
- ~ Complement față de 1: mov al, ~0 => mov AL, 0ffh

a d?....

b d?...

Mov eax, ![a] ; expression syntax error pt că... [a] NU este o constantă de la mom asamblării

Mov eax, ![a] ; ! can only be applied to SCALAR values !!!!!
a = POINTER !!!!!!

Mov eax, !a ; ! can only be applied to SCALAR values !!!!!
a = POINTER !!!!!!

Mov eax, !(a+7) ; ! can only be applied to SCALAR values !!!!!
a(+7) = POINTER !!!!!!

Mov eax, !(b-a) ; OK !!!! a,b – pointers, dar b-a = SCALAR !

Mov eax, ![a+7] – expression syntax error !

Mov eax, !7 ; EAX = 0 ;

Mov AH, ~7 ; 7 = 00000111b deci ~7 = 11111000b = f8h
Mov eax, ~7 ; EAX = -8 (FFFF FFF8h)

Mov eax, !ebx ; syntax error !

aa equ 2

Mov ah, !aa ; AH = 0 !!!! – MERGE !!!!!!

Mov AH, 17^(~17) ; AH = 1111111b = 0ffh

Mov ax, value ^ ~value ax=0ffffh

Operatori de tip și tipuri de date asociate operanzilor

Operatorii efectuează calcule cu valori constante SCALARE determinabile la momentul asamblării (valori scalare = valori imediate), cu excepția adunării și scăderii unei constante la un/dintr-un pointer (care va furniza intotdeauna “pointer data type”) și cu excepția formulei de calcul al offset-ului unui operand (care permite operatorul ‘+’).

Specificatorii BYTE / WORD / DWORD / QWORD au rolul de a clarifica o ambiguitate.

v d?
a d?...
b d?...

Push v - ok, stack ← offset v (pe 32 biți)

Push [v] - syntax error – Operation size not specified ! (PUSH pe o stivă pe 32 biți acceptă atât operanzi pe 32 biți cât și pe 16 biți)

Push dword [v] – ok
Push word [v] - ok

Mov eax,[v] - ok ; EAX = dword ptr [v], în Olly dbg “mov eax, dword ptr [DS:v]”

Push [eax] – syntax error.... Operation size not specified !
Push byte [eax] – syntax error....
Push word [eax] – ok

Push 15 ; PUSH DWORD 15

Pop [v] ; Op size not specified (a POP from the stack accepts both 16 and 32 bits values as stack operands) ;

Pop word/dword [v] – ok !!

Pop v ; sintaxa este POP destinatie; destinatie must be a L-value !!!!!
...dar v este R-value !!!! ; acest pop v este similar ca operatie cu 2:=3 !!!
(Invalid combination of opcode and operands)

Pop dword b ; syntax error !

Pop [eax] ; Op size not specified

Pop (d)word [eax] ; ok!

Pop 15 ; 15 is NOT a L-value !! – syntax error

Pop [15] ; syntax error - Op size not specified

Pop dword [15] ; syntactic ok , cel mai probabil **run-time error** deoarece probabil [DS:15] va provoca Access violation !!

Mov [v],0 ; syntax error - Op size not specified

Mov byte [v], 0 ; ok !

Mov [v], byte 0

Div [v] ; op. size ?

Div word [v]; ok !

Imul [v+2] ; op. size ?

Imul word [v+2]; DX:AX = AX*word de la adresa v+2

a d?...

b d?...

Mov a,b ; syntax error pt că a NU este L-value, ci R-value fiind un offset determinabil ca și constantă la momentul asamblării

Mov [a], b ; syntax error – op. size not specified !

Mov byte [a], b or mov [a], byte b – SYNTAX ERROR ! because AN OFFSET is EITHER a 16 bits value or a 32 bits value, NEVER an 8 bit value !!!!! (the same effect as mov ah, v)

Mov word [a], b ; ok !! – 2 octeți inferiori din valoarea offset-ului lui b !

Mov dword [a], b ; full offset 32 bits

Mov a,[b] ; Invalid comb. of opcode and operands (a = R-value)

Mov [a], [b] ; Invalid comb. Of opcode and operands (NU putem avea 2 operanze simultan din memorie)

Mul v – syntax error – MUL reg/mem

Mul word v - syntax error – MUL reg/mem

Mul [v] - op. size not specified

Mul dword [v]; ok !

Mul eax ; ok !

Mul [eax] ; op. size not specified

Mul byte [eax] ; ok !!!

Mul 15 ; Invalid comb. of opcode and operands – MUL reg/mem

Pop byte [v] – Invalid combination of opcode and operands

Pop qword [v] – Instruction not supported in 32 bit mode !

Clasificarea erorilor in informatică

- **Eroare de sintaxă – ea este diagnosticată de asamblor/compilator ! (eroare de asamblare)**
- **Run-time error (eroare la execuție) – programul “crapă” – programul se opreste = program crash !!**
- **Eroare logică = programul funcționează până la capăt sau ramâne blocat în ciclu infinit, însă GRESIT dpdv LOGIC obținându-se cu totul alte rezultate decât cele așteptate...**
- **Fatal: Linking Error !!! (de ex în cazul unei definiții duble de variabilă... 17 module și o variabilă trebuie să fie DEFINITA DOAR într-un singur modul ! Dacă ea este definită în 2 sau mai multe module se va obține Fatal: Linking Error !!! – Duplicate definition for symbol A1 !!!)**

The steps followed by a program from source code to run-time:

- Syntactic checking (done by assembler/compiler/interpreter)
- OBJ files are generated by the assembler/compiler
- Linking phase (performed by a LINKER = a tool provided by the OS, which checks the possible DEPENDENCIES between this OBJ files/modules); The result → .EXE file !!!
- You (the user) are activating your exe file (by clicking or entering...)
- The LOADER of the OS is looking for the required RAM memory space for your EXE file. When finding it, it loads the EXE file AND performs ADDRESS RELOCATION !!!!
- In the end the loader gives control to the processor by specifying THE PROGRAM's ENTRY POINT (ex: the start label) !!! The run-time phase begins NOW...

Mark Zbirkowski – semnătura EXE = 'MZ'

Tipuri de date asociate operanzilor

Directivele de definire a datelor in NASM NU sunt mecanisme de definire a tipurilor de date !!

a db 17,19

b dw 1234h ; 34h 12h

c dd....

Rolul directivelor de definire a datelor NU este în NASM de a preciza tipul de date al variabilelor definite, ci DOAR de a genera octeții corespunzători acelor zone de memorie pe care le ocupă în conformitate cu directiva specifică aleasă și respectând ordinea de plasare de tip little-endian !!!

Deci a NU este de tip byte – ci doar un offset/deplasament și atât... un simbol desemnând începutul unei zone de memorie FARA VREUN TIP ASOCIAT !

Iar b NU este de tip word – ci doar un offset/deplasament și atât ... un simbol desemnând începutul unei zone de memorie FARA VREUN TIP ASOCIAT !

Si nici c NU este de tip doubleword – ci doar un offset/deplasament și atât ... un simbol desemnând începutul unei zone de memorie FARA VREUN TIP ASOCIAT !

Atunci DE CE mai asociem în definiție o directivă de tip de dată ???? Pt a INDICA ASAMBLORULUI CUM să populeze cu date/initializeze zona de memorie respectivă !!! (fie ca o secvență de bytes, fie ca una de words, fie ca una de doublewords !) Directivele de date se referă la modalitatea de inițializare concretă a unei zone de memorie și nu asociază atributul de tip de dată cu un simbol !

Deci: directiva de definire a unei date NU este un mecanism de asociere de tip de dată pentru o variabilă, ci doar un mecanism de inițializare a zonei de memorie alocate variabilei cu valorile dorite !!!

Stiute, dar bine de a fi reamintite:

The **name of a variable** is **associated** in assembly language **with its offset relative to the segment** in which its **definition appears**. **The offsets of the variables** defined in a program are **always constant** values, determinable at assembly/compiling time.

Assembly language and **C** are **value oriented languages**, meaning that everything is reduced in the end to a numeric value, this being a low level feature.

In a **high-level programming language**, the **programmer can access the memory only** by using **variable names**, in contrast, in **assembly language**, the **memory is/can/must be accessed ONLY** by using the **offset computation formula** ("formula de la două noaptea") where **pointer arithmetic** is also used (pointer arithmetic is also used in C !).

mov ax, [ebx] – the source operand **doesn't** have an **associated data type** (it represents only a start of a memory area) and because of that, in the case of our MOV instruction the **destination operand** is the one that **decides the data type of the transfer (a word in this case)**, and the transfer will be made accordingly to the little endian representation.

Directive de definire de date (Data definition directives)

OllyDbg – CODE segment începe la offset-ul 00402000
OllyDbg – DATA segment începe la offset-ul 00401000

Segment data

a1 db 0,1,2,’xyz’ ; 00 01 02 ‘x’ ‘y’ ‘z’ ; offset(a1) - determinat la încărcarea lui OllyDbg = 00401000; offset(a1) determinat la asamblare de către NASM = 0 !!!
78 79 7A - codurile ASCII

db 300, “F”+3 ; 2C 49 - ‘ascii code F + 3’ - Warning – byte data (300 = 1 2Ch) exceeds bounds!

a2 TIMES 3 db 44h ; 44 44 44 ; offset a2 = 00401008, însă offsetul determinat la asamblare de către NASM va fi 8 !!

a3 TIMES 11 db 5,1,3 ; 05 01 03 ... de 11 ori (33 octeți)

a4 dw a2+1, ‘bc’ ; offset(a2)=00401008h; a2+1=00401009h; deci se generează
09 10 ‘b’ ‘c’ = 09 10 62 63 (2 cuvinte – words)

a41 dw a2+1, ‘b’, ‘c’ ; ‘b’ 00 ‘c’ 00 = 09 10 62 00 63 00 (3 cuvinte – words)

a42 db a2+1 ; – syntax error – OBJ format can only handle 16 or 32 bits relocation !

09 10 (corect, DAR... această valoare particulară 10h este calculabilă doar DUPĂ INCARCAREA PROGRAMULUI (LOADING) !!! – deci offset-ul începuturilor de segmente este și el determinabil doar la momentul încărcării programului – LOADING TIME !!!)

Offset-ul variabilelor față de începutul segmentelor în care apar sunt constante (de tip pointer !, NU scalar !) determinabile la momentul asamblării !!

a44 dw 1009h ; 09 10

a5 dd a2+1, ‘bcd’ ; 09 10 40 00 | 62 63 64 00
a6 TIMES 4 db ‘78’ ; 37 38 37 38 37 38 37 38
a61 TIMES 4 db ‘7’,‘8’ ; 37 38 37 38 37 38 37 38 !!!
a62 TIMES 4 dw ‘78’ ; 37 38 37 38 37 38 37 38 !!!

a7 db a2 ; syntax err. **OBJ format can only handle 16- or 32- relocation** (echiv. cu **mov ah,a2**)

a8 dw a2 ; 08 10

a9 dd a2 ; 08 10 40 00

a10 dq a2 ; 08 10 40 00 00 00 00 00

a11 db [a2] ; - expression syntax error – pt că [a2] NU este o expresie validă acceptată de către asamblor, nereprezentând “o valoare constantă determinabilă la momentul asamblării” ! Dereferențierea implicată aici este ceea ce deranjează asamblorul !! **Conținutul unei zone de memorie sau conținutul unui registru NU sunt valori constante determinabile la momentul asamblării !!!! Acestea sunt accesibile și determinabile DOAR la momentul execuției !! (run-time).**

a12 dw [a2] ; expression syntax error

a13 dd dword [a2] ; expression syntax error

a14 dq [a2] ; expression syntax error

a15 dd eax; expression syntax error

a16 dd [eax]; expression syntax error

mov ax, v ; Warning – 32 bit offset in 16 bit field !!!

Segment code (incepe intotdeauna la offset 00402000 - CINE decide asta ?)

Linkeditorul ia deciziile de acest tip. Adresa de baza pentru incarcarea PE-urilor, cel putin cea implicita (setata de catre linkeditorul de la Microsoft si nu numai) este 0x400000 in cazul executabilelor (respectiv 0x10000000 pentru biblioteci). Alink respecta aceasta conventie si completeaza in campul ImageBase al structurii IMAGE_OPTIONAL_HEADER din fisierul P.E. nou construit valoarea 0x400000. Cum fiecare "segment"/sectiune din program poate prevedea drepturi diferite de acces (codul este executabil, putem avea segmente read-only etc...), acestea sunt planificate sa inceapa fiecare la adresa cate unei noi pagini de memorie (4KiB, deci multiplu de 0x1000), fiecare pagina de memorie putand fi configurata cu drepturi specifice de catre incarcatorul de programe. In cazul unor programe de mici dimensiuni, implicatia este ca se va obtine urmatoarea harta a programului in memorie (la executare):

- programul este planificat a fi incarcat in memorie la exact adresa 0x400000 (insa aici vor ajunge structurile de metadate ale fisierului, nu codul sau datele programului in sine)
- primul "segment" va fi incarcat la 0x401000 (pun ghilimele deoarece nu este un segment propriu-zis ci doar o diviziune logica a programului, nu este asociat direct "segmentul" unui registru de segment – din aceasta pricina se prefera de multe ori denumirea de sectiune in loc de cea de segment)
- al doilea "segment" va fi incarcat la 0x402000 (segmentul pe care il va folosi procesorul pentru segmentare incepe la adresa 0 si are limita de 4GiB, indiferent de adresele si dimensiunile sectiunilor)
- va fi pregatit "segment" (sectiune) de importuri, "segment" de exporturi si "segment" de stack in ordinea decisa de catre likeditor (si de dimensiuni prevazute tot de catre acesta), segmente ce vor fi incarcate de la 0x403000, 0x404000 si asa mai departe (incremente de 0x1000 cat timp au dimensiune suficient de mica, in caz contrar fiind nevoie a se folosi pentru increment cel mai mic multiplu de 0x1000 care permite suficient spatiu pentru continutul intregului segment)

Conform logicei de decizie a adreselor de inceput ale sectiunilor, putem concluziona ca aici avem o sectiune (de date probabil) inaintea celei de cod, continand sub 0x1000 octeti, motiv pentru care codul porneste imediat dupa, de la 0x402000, harta programului fiind la final: metadate (antete) de la 0x400000, cod la 0x401000 si date la 0x402000 (urmat bineintele de alte "segmente" pentru stiva, importuri si, optional, exporturi).

Segment code (starts always at offset 00402000)

Start:

Jmp Real_start	(2 octeți)	- offset(instr. JMP) = 00402000
a db 17		- offset(a) = 00402002
b dw 1234h		- offset(b) = 00402003
c dd 12345678h		- offset(c) = 00402005

Real_start:

.....

Mov eax, c ; EAX = 00402005

Mov edx, [c] ; mov edx, DWORD PTR DS:[00402005]

.....

Mov edx, [CS:c] ; mov edx, DWORD PTR CS:[402005]

Mov edx, [DS:c] ; mov edx, DWORD PTR DS:[402005]

Mov edx, [SS:c] ; mov edx, DWORD PTR SS:[402005]

Mov edx, [ES:c] ; mov edx, DWORD PTR ES:[402005]

Efectul fiind in toate cele 5 cazuri **EDX:=12345678h DE CE ???**

Explicația este direct legată de **modelul de memorie flat** – toate segmentele descriu în realitate întreaga memorie, începând de la 0 și până la capătul primilor 4GiB ai memoriei. Ca atare, [CS:c] sau [DS:c] sau [SS:c] sau [ES:c] vor accesa aceeași locație de memorie însă cu drepturi de acces potențial diferite. Deși toți selectorii (potențiali DIFERITI – vezi OllyDbg !!!) indică segmente IDENTICE ca adresă și dimensiune, aceștia pot avea diferențe în cum le sunt completate alte câmpuri de control și de acces ale descriptorilor de segment indicați de către ei.

Modelul flat ne asigura ca mecanismul de segmentare este transparent pentru noi, noi nu sesizam diferențe intre segmente si, ca atare, scapam complet de grija segmentarii (insa ne intereseaza impartirea logica in segmente a programului, motiv pentru care folosim sectiuni/"segmente" separate pentru cod / date). Acest lucru este valabil insa doar cat timp ne limitam la CS/DS/ES si SS! Selectorii FS si GS indica inspre segmente speciale care nu respectă întotdeauna modelul flat (rezervate interacțiunii programului cu S.O-ul), mai precis, [FS:c] **nu garantează** indicarea aceleiași zone de memorie ca și [CS:c]!

De verificat:

Mov edx, [FS:c] ; mov edx, DWORD PTR FS:[401005] ?

Mov edx, [GS:c] ; mov edx, DWORD PTR GS:[401005] ?

(GS se pare ca merge – ES=SS=DS=GS = 002B !!)

FS = 0053 – Olly Dbg “descompune aiurea” aceste 2 ultime instr., deci ele trebuie verificate într-un .EXE care să fie rulat complet separat și nu “step by step”)

C A P I T O L U L 4

INSTRUCȚIUNI ALE LIMBAJULUI DE ASAMBLARE

Forma generală a unui program în NASM + scurt exemplu:

```
global start          ; solicităm asamblorului să confere vizibilitate globală simbolului denumit start  
                      ; (eticheta start va fi punctul de intrare în program)  
  
extern ExitProcess, printf ; informăm asamblorul că simbolurile ExitProcess și printf au proveniență străină,  
                          ; evitând astfel a fi semnalate erori cu privire la lipsa definirii acestora  
  
import ExitProcess kernel32.dll ; precizăm care sunt bibliotecile externe care definesc cele două simboluri:  
                                ; ExitProcess e parte a bibliotecii kernel32.dll (bibliotecă standard a sistemului de operare)  
import printf msvcrt.dll       ; printf este funcție standard C și se regăsește în biblioteca msvcrt.dll (SO)  
  
bits 32                ; solicităm asamblarea pentru un procesor X86 (pe 32 biți)  
  
segment code use32 class=CODE      ; codul programului va fi emis ca parte a unui segment numit code  
  
start:  
    ; apel printf("Salut din ASM")  
    push dword string ; transmitem parametrul funcției printf (adresa sirului) pe stivă (așa cere printf)  
    call  [printf]      ; printf este numele unei funcții (etichetă = adresă , trebuie indirectată cu [])  
  
    ; apel ExitProcess(0), 0 reprezentând "execuție cu succes"  
    push dword 0  
    call  [ExitProcess]  
  
segment data use32 class=DATA ; variabilele vor fi stocate în segmentul de date (denumit data)  
string: db "Salut din ASM!", 0
```

4.1. MANIPULAREA DATELOR/ 4.1.1. Instrucțiuni de transfer al informației

4.1.1.1. Instrucțiuni de transfer de uz general

MOV d,s	<d> <-> <s> (b-b, w-w, d-d)	-
PUSH s	ESP = ESP - 4 și depune <s> în stivă (s – dublucuvânt)	-
POP d	extrage elementul curent din stivă și îl depune în d (d – dublucuvânt) ESP = ESP + 4	-
XCHG d,s	<d> ↔ <s> s,d – trebuie sa fie L-values !!!	-
[reg_segment] XLAT	AL ← <DS:[EBX+AL]> sau AL ← <reg_segment:[EBX+AL]>	-
CMOVcc d, s	<d> ← <s> dacă cc (cod condiție) este adevărat	-
PUSHA / PUSHAD	Depune pe stiva EAX, ECX, EDX, EBX, ESP, EBP, ESI si EDI	-
POPA / POPAD	Extrage EDI, ESI, EBP, ESP, EBX, EDX, ECX si EAX de pe stiva	-
PUSHF / PUSHFD	Depune EFlags pe stivă	-
POPF / POPFD	Extrage vârful stivei și il depune în EFlags	-
SETcc d	<d> ← 1 dacă cc este adevărat, altfel <d> ← 0	-

Dacă operandul destinație al instrucțiunii MOV este unul dintre cei 6 registrii de segment atunci sursa trebuie să fie unul dintre cei opt registri generali de 16 biți ai UE sau o variabilă de memorie. Încăr cătorul de programe al sistemului de operare preinițializează în mod automat registrii de segment, iar schimbarea valorilor acestora, deși posibilă din punct de vedere al procesorului, nu aduce nici o utilitate (un program este limitat la a încărca doar valori de selectori ce indică înspre segmente preconfigurate de către sistemul de operare, fără a putea să definească segmente adiționale).

Instrucțiunile **PUSH** și **POP** au sintaxa

PUSH s și **POP d**

Operanții trebuie să fie reprezentați pe dublucuvânt, deoarece stiva este organizată pe dublucuvinte. Stiva crește de la adrese mari spre adrese mici, din 4 în 4 octeți, ESP punctând întotdeauna spre dublucuvântul din vârful stivei.

Funcționarea acestor instrucțiuni poate fi ilustrată prin intermediul unei secvențe echivalente de instrucțiuni MOV și ADD sau SUB:

push eax	\Leftrightarrow	sub esp, 4 ; pregătim (alocăm) spațiu pentru a stoca valoarea mov [esp], eax ; stocăm valoarea în locația alocată
pop eax	\Leftrightarrow	mov eax, [esp] ; încarcăm în eax valoarea din vârful stivei add esp, 4 ; eliberăm locația

În perspectiva evaluării efectului unor instrucțiuni ca **PUSH ESP** sau **POP dword [ESP]** trebuie precizat și mai clar ordinea în care se efectuează (sub)operatiile componente ale instrucțiunilor PUSH și POP:

- a). Se evaluatează operandul **sursă** al instrucțiunii (ESP pt PUSH sau respectiv elementul din vârful stivei pt POP)
- b). Se actualizează corespunzător ESP (ESP := ESP-4 pt PUSH și respectiv ESP := ESP+4 pt POP)
- c). Se efectuează atribuirea implicată de efectul instrucțiunii asupra operandului **destinație** (noul vârf al stivei pt PUSH și respectiv dword [ESP] (acesta fiind acum după scăderea ESP de la pct b) elementul de sub vârful initial al stivei) - pt POP

Presupunând că situația initială este ESP = 0019FF74, după **PUSH ESP** vom avea ESP = 0019FF70 și continutul din varful stivei va fi acum 0019FF74.

Presupunând că situația initială este ESP = 0019FF74 și că în aceasta locație se află valoarea 7741FA29 (varful stivei), după **POP dword [ESP]** vom avea ESP = 0019FF78 și continutul acestei locații (continutul locației din varful stivei) va fi 7741FA29 (deci am putea spune că „vf.stivei se mută cu o poziție mai jos”!!).

Instrucțiunile PUSH și POP permit doar depunerea și extragerea de valori reprezentate pe cuvânt și dublucuvânt. Ca atare, PUSH AL nu reprezintă o instrucțiune validă (syntax error), deoarece operandul nu este permis a fi o valoare pe octet. Pe de altă parte, secvența de instrucțiuni

```
PUSH ax ; depunem ax  
PUSH ebx ; depunem ebx  
POP ecx ; ecx <- dublucuvântul din vârful stivei (valoarea lui ebx)  
POP dx ; dx <- cuvântul ramas în stivă (deci valoarea lui ax)
```

este corectă și echivalentă prin efect cu

```
MOV ecx, ebx  
MOV dx, ax
```

Adițional acestei constrângeri (inerentă tuturor procesoarelor x86), sistemul de operare impune ca operarea stivei să fie obligatoriu făcută doar prin accese pe dublucuvânt sau multipli de dublucuvânt, din motive de compatibilitate între programele de utilizator și nucleul și bibliotecile de sistem. Implicația acestei constrângeri este că o instrucțiune de forma PUSH operand₁₆ sau POP operand₁₆ (de exemplu PUSH word 10), deși este suportată de către procesor și asamblată cu succes de către asamblor, nu este recomandată, putând cauza ceea ce poartă numele de eroare de dezalinierie e stivei: stiva este corect aliniată dacă și numai dacă valoarea din registrul ESP este în permanentă divizibilă cu 4!

Instrucțiunea **XCHG** permite interschimbarea conținutului a doi operanzi de aceeași dimensiune (octet, cuvânt sau dublucuvânt), cel puțin unul dintre ei trebuind să fie regisztr. Sintaxa ei este

XCHG *operand1, operand2*

Instrucțiunea **XLAT** "traduce" octetul din AL într-un alt octet, utilizând în acest scop o tabelă de corespondență creată de utilizator, numită *tabelă de translatare*. Instrucțiunea are sintaxa

[reg_segment] XLAT

tabelă_de_translatare este **adresa directă** a unui sir de octeți. Instrucțiunea XLAT pretinde la intrare adresa far a tabelei de translatare furnizată sub unul din următoarele două moduri:

- DS:EBX (implicit, dacă lipsește precizarea regisztrului segment)
- regisztr_segment:EBX, dacă regisztrul segment este precizat explicit

Efectul instrucției **XLAT** este înlocuirea octetului din AL cu octetul din tabelă ce are numărul de ordine valoarea din AL (primul octet din tabelă are indexul 0).

De exemplu, secvența

```
    mov ebx, Tabela  
    mov al,6  
    ES xlat           AL ← < ES:[EBX+6] >
```

depune conținutul celei de-a 7-a locații de memorie (de index 6) din *Tabela* în AL.

Dăm un exemplu de secvență care translatează o **valoare** zecimală 'numar' cuprinsă între 0 și 15 în **cifra** hexazecimală (codul ei ASCII) corespunzătoare:

```
segment data use32  
    . . .  
    TabHexa db '0123456789ABCDEF'  
    numar resb 1  
    . . .  
  
segment code use32  
    mov ebx, TabHexa  
    . . .  
    mov al, numar  
    xlat           ; AL ← < DS:[EBX+AL] >
```

O astfel de strategie este des utilizată și se dovedește foarte utilă în cadrul pregătirii pentru tipărire a unei valori numerice întregi (practic este vorba despre o conversie *valoare numerică registru – string de tipărit*).

4.1.1.3. Instrucțiunea de transfer al adreselor LEA

LEA <i>reg_general, continutul unui operand din memorie</i>	<i>reg_general <- offset(mem)</i>	-
--	--------------------------------------	---

Instrucțiunea **LEA** (*Load Effective Address*) transferă deplasamentul operandului din memorie *mem* în registrul destinație. De exemplu

lea eax,[v]

încarcă în EAX offsetul variabilei v, instrucțiune echivalentă cu

mov eax, v

Instrucțiunea **LEA** are însă avantajul că operandul sursă poate fi o expresie de adresare (spre deosebire de instrucțiunea **mov** care nu acceptă pe post de operand sursă decât o variabilă cu adresare directă într-un astfel de caz). De exemplu, instrucțiunea

lea eax,[ebx+v-6] având ca efect „**mov eax, ebx+v-6**”

nu are ca echivalent direct o singură instrucțiune **MOV**, instrucțiunea

mov eax, ebx+v-6

fiind incorectă sintactic deoarece expresia **ebx+v-6** nu este determinabilă la momentul asamblării.

Prin utilizarea directă a valorilor deplasamentelor ce rezultă în urma calculelor de adrese (in contrast cu folosirea memoriei indicate de către acestea), **LEA** se evidențiază prin versatilitate și eficiență sporite: versatilă prin combinarea unei înmulțiri cu adunări de registri și/sau valori constante și eficiență ridicată datorată execuției întregului calcul într-o singură instrucțiune, fără a ocupa

circuitele ALU care rămân astfel disponibile pentru alte operații (timp în care calculul de adresă este efectuat de către circuite specializate, separate, ale BIU).

Exemplu: inmulțirea unui număr cu 10

```
mov eax, [număr]      ; eax <- valoarea variabilei număr  
lea eax, [eax * 2]    ; eax <- număr * 2  
lea eax, [eax * 4 + eax] ; eax <- (eax * 4) + eax = eax * 5 = (număr * 2) * 5
```

4.1.1.4. Instrucțiuni asupra flagurilor

Următoarele patru instrucțiuni sunt *instrucțiuni de transfer* al indicatorilor:

Instrucțiunea **LAHF** (*Load register AH from Flags*) copiază indicatorii SF, ZF, AF, PF și CF din registrul de flag-uri în biții 7, 6, 4, 2 și respectiv 0 ai registrului AH. Conținutul biților 5,3 și 1 este nedefinit. Indicatorii nu sunt afectați în urma acestei operații de transfer (în sensul că instrucțiunea LAHF nu este ea însăși generatoare de efecte asupra unor flag-uri – ea doar transferă valorile flag-urilor și atât).

Instrucțiunea **SAHF** (*Store register AH into Flags*) transferă biții 7, 6, 4, 2 și 0 ai registrului AH în indicatorii SF, ZF, AF, PF și respectiv CF, înlocuind valorile anterioare ale acestor indicatori.

Instrucțiunea **PUSHF** transferă toți indicatorii în vârful stivei (conținutul registrului Flags se transferă în vârful stivei). Indicatorii nu sunt afectați în urma acestei operații. Instrucțiunea **POPF** extrage cuvântul din vârful stivei și transferă din acesta indicatorii corespunzători în registrul de flag-uri.

Limbajul de asamblare pune la dispoziția programatorului niște *instrucțiuni de setare* a valorii indicatorilor de condiție, pentru ca programatorul să poată influența după dorință modul de acțiune a instrucțiunilor care exploatează flaguri.

CLC	CF=0	CF
CMC	CF = ~CF	CF
STC	CF=1	CF
CLD	DF=0	DF
STD	DF=1	DF

CLI, STI – actioneaza asupra flagului de intrerupere (IF). Functioneaza efectiv doar in programarea sub 16 biti, aici la programarea sub 32 biti SO interzicand accesul la flag-ul de intreruperi.

4.3.2.3. Exemple comentate

Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

mov al,80h ;al := 128 = 10000000b = -128 ! (Interesant! – remarcă faptul că datorită regulilor de reprezentare în cod complementar 128 și -128 au aceeași reprezentare binară și anume 10000000b!)

(*) **cmp al,0** ;instrucțiunea cmp nu interpretează în nici un fel valoarea din AL (ca fiind ;cu semn sau fără semn) ci doar realizează scăderea fictivă al-0 și afecteazăcorespunzător flagurile: SF=1 , CF=ZF=OF=PF=AF=0.

jl et ;utilizarea instrucțiunii JL (Jump if Less than) provoacă interpretarea ;comparației **al<0 cu semn** (vezi tabelul 4.2), adică cf. tabelului 4.1 se ;testează dacă SF≠OF și cum SF=1 iar OF=0 se decide îndeplinirea condiției ;și saltul la eticheta et. Deducem deci că interpretarea valorilor comparate a ;stat la latitudinea programatorului care prin utilizarea instrucțiunii JL a ;decis că dorește să compare -128 cu 0 și cum -128 este “less than” 0 ;condiția a fost îndeplinită (echiv. cu jnge et). În contrast, **jnl et** sau **jge et** ;(care vor testa dacă SF=OF) NU vor fi îndeplinite și NU vor provoca saltul ;la eticheta specificată.

jb et ;utilizarea instrucțiunii JB (Jump if Below) provoacă interpretarea ;comparației **al<0 fără semn** (vezi tabelul 4.2), adică cf. tabelului 4.1 se ;testează dacă CF=1 și cum CF=0 se decide neîndeplinirea condiției deci nu ;se va face saltul la eticheta et. Deducem deci că interpretarea valorilor comparate a stat la latitudinea programatorului care prin utilizarea instrucțiunii JB a decis că dorește să compare 128 cu 0 și cum 128 NU este “below” 0 condiția NU a fost îndeplinită (echivalent cu jnae et sau jc et).

jae et1 ;se testează **fără semn** dacă **al ≥ 0** (**128 ≥ 0?**) - CF=0 deci condiție ;îndeplinită (echivalent cu jnc et1 sau jnb et1) – se efectuează saltul la ;eticheta et1

jbe et2 ;se testează **fără semn** dacă **al ≤ 0** (**128 ≤ 0?**) – CF = ZF = 0 deci condiția ;(CF=1 sau ZF=1) NU este îndeplinită și ca urmare nu se va face saltul la ;eticheta et2 – rezultat consistent cu jb et, deoarece **jbe** implică **jb** ;(echivalent cu jna et2)

ja et3 ;se testează **fără semn** dacă **al > 0** (**128 > 0?**) – CF = ZF = 0 deci condiția ;(CF=0 și ZF=0) este îndeplinită și ca urmare se

va face saltul la eticheta **et3** ;(echivalent cu **jnbe et3**) și rezultat consistent cu **jbe et2**, deoarece dacă ;**jbe** nu este îndeplinită atunci **ja** trebuie să fie.

je et4 ;se testează dacă **al** = 0 ($128 = 0 ?$) – nu se pune problema semnului dacă se ;testează egalitatea! – cum **ZF=0**, condiția **ZF=1** nu este îndeplinită deci nu ;se va efectua saltul la eticheta **et4** (echivalent cu **jz et4**). În contrast, **jne et4** sau **jnz et4** (care vor testa dacă **ZF=1**) vor fi îndeplinite și vor provoca ;saltul la eticheta specificată.

jle et5 ;se testează cu semn dacă **al** ≤ 0 ($-128 \leq 0 ?$) – **OF = ZF = 0** și **SF=1** deci ;condiția (**ZF=1** sau **SF≠OF**) este îndeplinită și ca urmare se va face saltul la ;eticheta **et5** (echivalent cu **jng et5**) și rezultat consistent cu **jl et**, deoarece ;**jle** implică **jl**.

jg et6 ;se testează cu semn dacă **al** > 0 ($-128 > 0 ?$) – **OF = ZF = 0** și **SF=1** deci ;condiția (**ZF=0** și **SF=OF**) NU este îndeplinită și ca urmare NU se va face ;saltul la eticheta **et6** (echivalent cu **jnle et6**) și rezultat consistent cu **jle et5**, deoarece dacă **jg** nu este îndeplinită atunci **jle** trebuie să fie.

jp et7 ;se testează dacă **PF=1** - **PF=0** deci condiție neîndeplinită – nu se efectuează ;saltul (echivalent cu **jpe et7** – *Jump if Parity Even*). În contrast, **jnp et7** ;(care testează dacă **PF=0** – echivalentă cu **jpo et7** – *Jump if Parity Odd*) va ;fi îndeplinită și saltul se va efectua.

jo et8
efectuează ;se testează dacă **OF=1** - **OF=0** deci condiție neîndeplinită – nu se ;saltul (nu există depășire). În contrast, **jno et8** (care testează dacă **OF=0**) va ;fi îndeplinită și saltul se va efectua.

js et9 ;se testează dacă în interpretarea cu semn rezultatul comparației are semn ;negativ (deoarece aşa cum specificam în cadrul prezentării instrucțiunii ;**CMP**, nu este vorba de a interpreta cu semn sau fără semn **operanții** ;scăderii fictive *d-s*, ci **rezultatul** final al acesteia !) adică testăm dacă **SF=1** -;condiție îndeplinită în cazul nostru și ca urmare saltul se va efectua !. În ;contrast, **jns et9** (care testează dacă **SF=0**) NU va fi îndeplinită și saltul ;NU se va efectua.

cmp 0,al ;eroare de sintaxă : “*Illegal immediate*” deoarece sintaxa instrucțiunii **cmp** ;interzice specificarea ca prim operand a unei valori imediate (constante). ;dacă totuși dorim forțarea unei

comparații de acest tip (0-al) putem utiliza ;pe post de prim operand un registru inițializat cu valoarea 0.

`mov bl,0`

`cmp bl, al` ;realizează scăderea fictivă bl-al (0-al = 0-80h = 0-10000000b = ;10000000b) și afectează corespunzător flagurile: CF=SF=OF=1, ;ZF=PF=AF=0.

Exercițiu propus: Reluați discuția efectului tuturor instrucțiunilor de salt condiționat de mai sus (analizate pentru cazul comparației (*) `cmp al,0`) în condițiile în care această comparație e înlocuită de ultimele două instrucțiuni prezentate, adică în cazul în care se efectuează `cmp bl,al` cu `bl=0`.

Care ar fi însă justificarea faptului că în cazul `cmp bl,al` avem $CF = OF = SF = 1$ iar în cazul `cmp al,0` doar $SF=1$ iar $CF = OF = 0$?

Pentru a justifica modurile de setare diferite ale flag-urilor trebuie să luăm în discuție regulile practice de setare a acestor flag-uri. Aceste reguli generale sunt:

- SF ia valoarea bitului de semn al rezultatului obținut;
- CF ia valoarea cifrei de transport : dacă e vorba despre o adunare se analizează dacă rezultatul obținut a provocat ($CF=1$) sau nu ($CF=0$) un transport în afara spațiului de reprezentare; dacă e vorba despre o scădere $d-s$, avem: dacă $|d| \geq |s|$ atunci $CF=0$ (nu e nevoie de cifră de împrumut pentru efectuarea scăderii) iar dacă $|d| < |s|$ atunci $CF=1$ (este nevoie de cifră de împrumut pentru efectuarea scăderii și acest lucru se reflectă în CF)
- OF este setat la valoarea 1 dacă există depășire în interpretarea cu semn a rezultatului (“*OF is set if there exists a signed overflow*”), adică dacă rezultatul obținut nu se încadrează în intervalul de interpretare admis (acesta fiind [-128..+127] dacă este vorba despre octeți și respectiv [-32768..+32767] pentru cuvinte interpretate cu semn).

Ultimele două reguli derivă de fapt din modul de implementare a conceptului de **depășire** (*overflow*) la nivelul procesorului 80x86.

În cazul operațiilor/operanzilor fără semn depășirea va fi semnalată prin setarea indicatorului CF (*carry flag*). În cazul operațiilor/operanzilor cu semn depășirea va fi semnalată prin setarea indicatorului OF (*overflow flag*).

Cum să detectăm însă situațiile de depășire în cazul operațiilor de adunare și scădere ?
Care sunt regulile practice de aplicat pentru a înțelege și a putea justifica corect setările de flag-uri pe care le remarcăm în cadrul programelor rulate ? În discuțiile ce urmează ne vom concentra în principal pe justificarea modului de setare a flag-ului OF (*overflow flag*) deoarece și datorită numelui său acesta este principalul factor răspunzător de caracterizarea unei situații din partea programatorilor ca fiind depășire sau nu.

Atragem însă atenția asupra a ceea ce se ignoră de multe ori în acest context și anume faptul că o situație de tipul CF=1 (cu OF=0) semnalează la rândul ei o depășire, însă pentru cazul numerelor interpretate fără semn.

Pentru ADUNARE: dacă se adună două numere de același semn și rezultatul este de semn diferit atunci se semnalează depășire (OF=1), în caz contrar nu (OF=0). Aceasta este deci ceea ce am putea numi *regula depășirii la adunare* (RDA) în cazul interpretării cu semn.

De exemplu, la nivel de octet, dacă vom considera adunarea $100 + 50 = 150$ vom obține depășire (!) cu semn (pare surprinzător, nu-i aşa ?). Justificare: $100 (= 64h = 01100100b) + 50 (= 32h = 00110010b) = 150 (= 96h = 10010110b)$. Operanzii au același semn dar rezultatul este de semn diferit, deci conform RDA vom avea OF=1. Intuitiv, depășirea se poate justifica prin faptul că $150 \notin [-128..127]$ deci se obține o eroare de tip “*out of range*”. Deși s-ar putea replica faptul că $150 = 10010110b = -106$ (în interpretarea cu semn), iar $-106 \in [-128..127]$, această ultimă interpretare nu poate fi acceptată deoarece operanzii (100 și 50) au valori pozitive în ambele interpretări (bitul de semn fiind 0). Ca urmare, suma a două numere pozitive nu poate da un număr negativ și astfel singura interpretare ce poate fi acceptată în acest context pentru $10010110b$ este $150 \notin [-128..127]$ deci se setează OF=1.

Pe de altă parte, CF = 0 (nu există cifră de transport în afara spațiului de reprezentare) deci nu avem depășire în interpretarea fără semn: rezultatul adunării $100 + 50 = 150 \in [0, 255]$ (intervalul de interpretare admis pentru numere fără semn).

Analog, în interpretarea cu semn, suma a două numere negative nu poate furniza un număr pozitiv. Luăm exemplul:

$$\begin{array}{r} 10010110 + \\ 10000010 \\ \hline 1\ 00011000 \end{array}$$

Se observă din reprezentarea binară că există un transport de cifră 1 în afara spațiului de reprezentare admis al celor 8 biți, deci intuitiv este suficient de justificat depășirea. Din punct de vedere al aplicării RDA obținem pe 8 biți în interpretarea cu semn că suma a două numere negative (ele sunt negative deoarece bitul de semn este 1 pentru ambele numere) ar trebui să furnizeze un număr pozitiv: $00011000b = 18h = 24$. Această valoare este de fapt o trunchiere a valorii binare corecte (pe 9 biți!) ce ar fi trebuit obținută ($100011000b = 118h$), iar trunchierarea are loc tocmai datorită depășirii. Ca urmare, nu se poate obține un număr pozitiv prin adunarea a două numere negative (decât printr-o trunchiere iar necesitatea trunchierii înseamnă de fapt depășire!). Se observă că o astfel de trunchieră înseamnă întotdeauna și apariția unei cifre de transport 1 în afara dimensiunii de reprezentare a rezultatului, deci vom avea automat și CF=1.

$10010110b = 96h = -106$ (în interpretarea cu semn) = $+150$ (în interpretarea fără semn)
 $10000010b = 82h = -126$ (în interpretarea cu semn) = $+130$ (în interpretarea fără semn)

În interpretarea fără semn avem $150 + 130 = 280 \notin [0..255]$ (justificarea intuitivă a depășirii). Tehnic, am văzut deja că CF = 1 și rezultă astfel clar că avem depășire în interpretarea fără semn.

Nu putem avea deci $-106 + (-126) = 24!$ (pentru că $00011000b = 18h = 24$ în ambele interpretări) Acesta este sensul în care se aplică RDA aici. Un alt mod de justificare intuitivă a depășirii în acest tip de situație este:

În interpretarea cu semn avem $-106 + (-126) = -232 \notin [-128..127]$ deci OF=1.

Această ultimă motivație este mai intuitivă pentru justificarea depășirii însă astfel de justificări sunt mai greu de exprimat la nivelul unui algoritm. Tehnic vorbind, RDA rămâne “cea mai rapid aplicabilă regulă practică din punct de vedere algoritmic” dacă ne putem exprima așa... (și iată că am putut!)

Rezultă că în cazul în care adunăm două numere de semne diferite nu se va semnala niciodată depășire. De asemenea, dacă adunăm două numere de același semn dar rezultatul are același semn cu operanții nu se va semnala nici în acest caz depășire (înseamnă că nu a fost nevoie de trunchiere pentru reprezentarea rezultatului pe aceeași dimensiune ca și cea a operanților). Se poate verifica ușor din punct de vedere matematic că în nici unul din aceste cazuri nu ieșim din intervalul de interpretare admis.

Pentru SCĂDERE: se interpretează **operanții** respectiv **cu semn**, se efectuează scăderea solicitată asupra configurațiilor corespunzătoare de biți și dacă rezultatul obținut interpretat cu semn nu se încadrează în intervalul de interpretare admis (intervalul [-128..127] pentru octetii cu semn și respectiv [-32768..32767] pentru cuvinte interpretate cu semn) atunci se semnalează depășire (*overflow*) și astfel OF=1. Această formulare o putem numi *regula depășirii la scădere* (RDS) pentru cazul interpretării cu semn.

În cazul depășirii la scădere fără semn: necesitatea efectuării unei scăderi cu împrumut de cifră este semnalată de către procesor prin setarea CF=1, pe care o putem interpreta semnificativ “depășire la scădere în interpretarea fără semn”.

Să analizăm în continuare mai multe exemple menite să clarifice aplicarea regulilor de mai sus precum și impactul lor asupra modului de setare al flag-urilor.

Exemple:

- i). **mov ah,82h ;** $82h = 130$ (interpretarea fără semn) = -126 (interpretarea cu semn)
; = $10000010b$ (bitul de semn fiind 1 cele două interpretări diferă)
mov bh,2ah ; $2ah = 42$ (atât în interpretarea cu semn cât și în cea fără semn)
; = $00101010b$ (bitul de semn fiind 0 cele două interpretări coincid)
cmp ah,bh ; se realizează scăderea fictivă $ah-bh=10000010b - 00101010b = 01011000b$
; = $58h = 88$ (atât în interpretarea cu semn cât și în cea fără semn deoarece ;bitul de semn este 0)

Această scădere setează flag-urile astfel:

SF = 0 (deoarece bitul de semn pentru rezultatul $58h = 01011000b$ este 0)

CF = 0 (deoarece $|82h| > |2ah|$ nu se pune problema unei scăderi cu împrumut de cifră; deci

nu vom avea depăşire în interpretarea fără semn care se efectuează: $130 - 42 = 88$)

OF = 1 (se efectuează scăderea în interpretarea cu semn, adică $ah-bh = -126 - 42 = -168$ și

cum $-168 \notin [-128..127]$ se semnalează *signed overflow* și ca urmare OF=1)

cmp bh,ah ;se realizează scăderea fictivă $bh-ah = 00101010b-10000010b = 10101000b$

; = A8h = 168 (în interpretarea fără semn) = -88 (în interpretarea cu semn)

Această scădere setează flag-urile astfel:

SF = 1 (deoarece bitul de semn pentru rezultatul $A8h = 10101000b$ este 1)

CF = 1 (deoarece $|2ah| < |82h|$ se pune problema unei scăderi cu împrumut de cifră; în interpretarea fără semn scăderea devine $42 - 130 = 168$ (!) provenită de fapt din necesitatea unei scăderi de tipul $(256 + 42) - 130 = 168$ și ca urmare a necesității împrumutului se va semnala depăşire în interpretarea fără semn, înțeleasă aici ca “nu se poate efectua corect această scădere fără utilizarea unei cifre de împrumut”)

OF = 1 (se efectuează scăderea în interpretarea cu semn, adică $bh-ah = 42-(-126) = +168$ și

cum $+168 \notin [-128..127]$ se semnalează *signed overflow* și ca urmare OF=1)

ii). **mov ah,126** ;echivalent cu **mov ah,7eh** deoarece $126 = 7Eh = 01111110b$ (bitul de ;semn fiind 0 cele două interpretări coincid, ca urmare conținutul lui AH este ;126 atât în interpretarea cu semn cât și în cea fără semn)

mov bh,2ah ; $2ah = 42$ (atât în interpretarea cu semn cât și în cea fără semn)

; = 00101010b (bitul de semn fiind 0 cele două interpretări coincid)

cmp ah,bh ;se realizează scăderea fictivă $ah-bh = 01111110b-00101010b = 01010100b$

; = 54h = 84 = $126 - 42$ (atât în interpretarea cu semn cât și în cea fără semn deoarece bitul de semn al rezultatului este 0)

Această scădere setează flag-urile astfel:

SF = 0 (deoarece bitul de semn pentru rezultatul $54h = 01010100b$ este 0)

CF = 0 (deoarece $|126| > |42|$ nu se pune problema unei scăderi cu împrumut de cifră, deci nu se va semnala depăşire în interpretarea fără semn)

OF = 0 (se efectuează scăderea în interpretarea cu semn, adică $ah-bh = 126 - 42 = 84$ și

cum $84 \in [-128..127]$ NU se semnalează *signed overflow* și ca urmare OF=0)

cmp bh,ah ;se realizează scăderea fictivă $bh-ah = 00101010b-0111110b = 10101100b$

; = $42-126 = ACh = 172$ (în interpretarea fără semn) = -84 (în interpretarea ;cu semn)

Această scădere setează flag-urile astfel:

SF = 1 (deoarece bitul de semn pentru rezultatul ACh = 10101100b este 1)

CF = 1 (deoarece $|42| < |126|$ se pune problema unei scăderi cu împrumut de cifră ; în interpretarea fără semn scăderea devine $42 - 126 = 172$ (!) provenită de fapt din necesitatea unei scăderi de tipul $(256 + 42) - 126 = 172$ și ca urmare a necesității împrumutului se va semnala depășire în interpretarea fără semn prin setarea flagului carry)

OF = 0 (se efectuează scăderea în interpretarea cu semn, adică $bh-ah = 42-126 = -84$ și

cum $-84 \in [-128..127]$ NU se semnalează *signed overflow* și ca urmare OF=0)

Ca regulă generală să observăm că din punctul de vedere al reprezentării binare, dacă rezultatul scăderii $a-b \in [-127..127]$ atunci și $b-a \in [-127..127]$ (situația particulară în care $a-b = -128$ o tratăm mai jos). Analog pentru reprezentări de tip cuvânt la nivelul intervalului $[-32767..32767]$ cu discuție asupra cazului particular -32768 . Ca urmare se poate concluziona faptul că instrucțiunile **cmp a,b** și **cmp b,a** vor furniza întotdeauna aceeași valoare pentru OF.

iii). - discuție asupra cazurilor **cmp 80h,0** și **cmp 0,80h**

mov ah,80h ; $80h = 128$ (interpretarea fără semn) = -128 (interpretarea cu semn)

; = $10000000b$ (bitul de semn fiind 1 cele două interpretări diferă)

mov bh,0 ; $bh:=0$

cmp ah,bh ;se realizează scăderea fictivă $ah-bh = 10000000b-00000000b = 10000000b$

; = $80h = 128$ (interpretarea fără semn) = -128 (interpretarea cu semn)

Această scădere setează flag-urile astfel:

SF = 1 (deoarece bitul de semn pentru rezultatul $80h = 10000000b$ este 1)

CF = 0 (deoarece $|80h| > |0|$ nu se pune problema unei scăderi cu împrumut de cifră, deci nu

poate fi vorba despre depășire în interpretarea fără semn)

$OF = 0$ (se efectuează scăderea în interpretarea cu semn, adică $bh - ah = 00000000b - 10000000b = -128$ și

cum $-128 \in [-128..127]$ NU se semnalează *signed overflow* și ca urmare $OF=0$)

cmp bh,ah ; se realizează scăderea fictivă $bh - ah = 00000000b - 10000000b = 10000000b$

; $= 80h = 128$ (interpretarea fără semn) $= -128$ (interpretarea cu semn)

Această scădere setează flag-urile astfel:

$SF = 1$ (deoarece bitul de semn pentru rezultatul $80h = 10000000b$ este 1)

$CF = 1$ (deoarece $|0h| < |80h|$ se pune problema unei scăderi cu împrumut de cifră; în interpretarea fără semn scăderea devine $0 - 128 = 128$ (!) provenită de fapt din necesitatea unei scăderi de tipul $(256 + 0) - 128 = 128$ și ca urmare a necesității împrumutului se va semnala depășire în interpretarea fără semn prin setarea flagului carry)

$OF = 1$ (se efectuează scăderea în interpretarea cu semn, adică $bh - ah = 0 - (-128) = +128$ și

cum $+128 \notin [-128..127]$ se semnalează *signed overflow* și ca urmare $OF=1$)

$CF = 1$ în cazul **cmp 0,80h** deoarece se efectuează o scădere cu împrumut de tipul :

$$\begin{array}{r} 0 - 10000000b = \\ \hline 10000000 \\ \hline 010000000 \end{array}$$

și cifra de împrumut se transferă în CF.

Să analizăm în acest context ce înseamnă și cum s-a ajuns la domeniul “numerelor cu semn posibil a fi reprezentate pe 1 octet” respectiv domeniul “numerelor cu semn posibil a fi reprezentate pe 1 cuvânt”.

Pe 1 octet se pot reprezenta 256 de valori, indiferent că vorbim despre interpretarea cu semn sau interpretarea fără semn. În interpretarea fără semn aceste valori sunt cele din intervalul [0..255]. Care sunt însă cele 256 de valori reprezentabile în interpretarea cu semn? Este vorba despre intervalul [-128..127] sau despre intervalul [-127..128]? Pentru că nu poate fi vorba despre intervalul [-128..128] deoarece în acest interval sunt 257 de valori! Cu alte cuvinte cineva a trebuit să aleagă una dintre cele două variante și totodată să facă precizarea că numerele -128 și +128 nu pot coexista între limitele aceluiasi interval de reprezentare al aceluiași tip de dată! (reamintim că în limbaj de asamblare tip de dată = dimensiune de reprezentare)

În acest sens este de observat și impactul acestui mod de reprezentare asupra limbajelor de nivel înalt: de exemplu atât **shortint** cât și **byte** în Turbo Pascal acceptă valoarea 80h (-128 ca **shortint** și +128 ca **byte**) însă 80h **nu poate avea două interpretări distincte în**

cadrul aceluiasi tip de dată ! Nu vom întâlni la nivelul nici unui limbaj de programare de nivel înalt valorile -128 și +128 ca fiind prezente în cadrul aceluiasi tip de dată !

Ca urmare, s-a luat decizia ca intervalul acceptat al valorilor cu semn reprezentabile pe 1 octet să fie intervalul [-128..+127] (care este exact domeniul de valori și a tipului de dată **shortint** din Turbo Pascal): deci **+128 nu este acceptat ca valoare cu semn reprezentabilă pe 1 octet !**

Totuși, după cum putem verifica foarte ușor, instrucțiunile **mov ah, 128 și mov ah,-128** sunt amândouă acceptate de către asamblor, efectul fiind în ambele cazuri încărcarea în *ah* a configurației binare 10000000b ! Aceasta deoarece în primul caz va fi vorba de fapt despre interpretarea fără semn pentru 80h iar în al doilea caz va fi vorba despre interpretarea cu semn. Simpla încărcare a unui registru cu o anumită configurație binară nu presupune și necesitatea interpretării respectivei configurații într-un anumit fel. Sarcina interpretării acelei configurații drept cu semn sau fără semn va cădea în sarcina instrucțiunilor ce urmează și care vor folosi ca operanzi aceste valori. De exemplu, utilizarea lui IMUL în loc de MUL va provoca interpretarea configurației binare respective drept un operand cu semn în loc de unul fără semn. Analog, utilizarea lui DIV în loc de IDIV va provoca interpretarea aceluiasi operand ca fără semn și.a.m.d.

În cazul **cmp 80h,0** se efectuează $80h - 0 = 80h = 10000000b$ ($128 - 0 = 128$ în interpretarea fără semn) fără a fi nevoie de o cifră de transport împrumutată pentru a putea efectua scăderea, deci nu avem depășire în interpretarea fără semn și astfel CF = 0. În interpretarea cu semn a operanzilor și a rezultatului final avem $-128 - 0 = -128 \in [-128..127]$ deci nu avem depășire nici în interpretarea cu semn și astfel OF = 0.

Pe de altă parte, avem evident în ambele cazuri SF=1. Justificarea *intuitivă*: în interpretarea cu semn valoarea 10000000b reprezintă un număr strict negativ adică -128. Justificarea *tehnică*: bitul de semn al reprezentării binare 10000000b este 1 deci SF=1.

iv). Să analizăm în continuare modurile în care putem compara valorile 0 și 1 (și apoi 0 și -1) și ce efecte are asupra flagurilor instrucțiunea **cmp** în fiecare dintre situații.

Situată **cmp 1,0** (evidențiată la nivelul unui text sursă de exemplu prin **cmp ah,0** cu **ah=1**) va efectua scăderea fictivă $1-0 = 1 = 00000001b$. Efectul asupra flag-urilor va fi CF = SF = OF = ZF = PF = AF = 0. Justificările sunt evidente pe baza discuțiilor din exemplele anterioare.

Situată **cmp 0,1** (evidențiată la nivelul unui text sursă de exemplu prin **cmp ah,1** cu **ah=0**) va efectua scăderea fictivă $0-1 = -1 = 11111111b$:

$$\begin{array}{r} 0 - 00000001b \\ = \quad 1\ 00000000 - \\ \underline{- \quad 00000001} \\ 0\ 11111111 \end{array}$$

Efectul asupra flag-urilor va fi CF = SF = PF = AF = 1 și ZF = OF = 0. Justificarea valorilor din CF și SF este și aici evidentă pe baza discuțiilor din exemplele anterioare iar OF=0 deoarece rezultatul în interpretarea cu semn este -1, iar $-1 \in [-128..127]$.

Situată **cmp -1,0** (evidențiată la nivelul unui text sursă de exemplu prin **cmp ah,0** cu ah = -1) va efectua scăderea fictivă $-1 - 0 = -1 = 11111111b$. Efectul asupra flag-urilor va fi SF = PF = 1 și CF = OF = ZF = AF = 0. SF=1 deoarece bitul de semn este 1. OF=0 deoarece rezultatul în interpretarea cu semn este -1, iar $-1 \in [-128..127]$. CF=0 deoarece nu se impune efectuarea unei scăderi cu împrumut.

Situată **cmp 0,-1** (evidențiată la nivelul unui text sursă de exemplu prin **cmp ah,-1** cu ah = 0) va efectua scăderea fictivă $0 - (-1) = +1 = 00000001b$:

$$0 - 11111111b = 1\ 00000000 - \\ \underline{11111111} \\ 0\ 00000001$$

Efectul asupra flag-urilor va fi CF = AF = 1 și OF = SF = ZF = PF = 0. SF = 0 deoarece bitul de semn este 0. OF=0 deoarece $0 - (-1) = +1 \in [-128..127]$. CF = 1 deoarece se impune efectuarea unei scăderi cu împrumut. Putem justifica și așa: în interpretarea fără semn această scădere înseamnă de fapt $0 - 255 = 1$ (!), care trebuie justificată prin $(256+0) - 255 = 1$, deci e nevoie de cîfră de împrumut și astfel se semnalează depășire în cazul interpretării fără semn, deci CF = 1.

v). Cazurile studiate anterior (i-iv) s-au referit la operații de scădere datorită analizei pe care am avut-o în vedere asupra efectelor instrucțiunii **Cmp**. Să analizăm în continuare și cazul unei depășiri furnizate de operația de adunare revenind astfel la discuția asupra aplicării regulii RDA:

```
mov ah,126 ;126 = 01111110b = 7eh (aceeași valoare 126 în ambele interpretări)
add ah, 2    ; 2 = 2h = 00000010b ; AH := 01111110b + 00000010b = 7eh + 02h
=
; 10000000b = 80h (= 128 fără semn = -128 cu semn)
CF = 0 deoarece: 01111110 +
00000010
10000000 - nu există transport în afara spațiului de reprezentare
al rez.
```

SF = 1 deoarece bitul de semn al rezultatului este 1 (în interpretarea cu semn rezultatul operației efectuate este strict negativ = -128).

OF = 1 deoarece:

- justificare *tehnică* - conform RDA se adună două numere de același semn (bitul de semn este 0 pentru amândouă) iar rezultatul este de semn diferit (bitul de semn este 1).

- justificare *intuitivă* - adunăm două numere fără semn a căror sumă este $126 + 2 = 128$. Însă numărul $+128 \notin [-128..127]$ deci se semnalează *signed overflow* și ca urmare $OF=1$.

vi). Unul dintre efectele surprinzătoare ale interpretărilor cu semn sau fără semn se referă la situația în care programatorul își initializează operanții cu anumite valori inițiale dorite (cu semn sau fără semn, conform necesităților problemei în cauză) și se așteaptă la obținerea unor rezultate sau reacții în conformitate cu valorile furnizate. Atenție însă! De obicei aceste valori au o dublă interpretare posibilă și nu vor fi interpretate în orice situație sub forma furnizată la inițializare!

Utilizarea ulterioară a unor instrucțiuni care forțează prin modul de lor de acțiune interpretarea complementară (cu semn/fără semn) celei de la inițializare poate provoca apariția unor situații în care un utilizator la prima vedere fie să suspecteze erori din partea asamblorului (!) fie din punct de vedere al exprimării în baza 10 să se ajungă la interpretări hilare... Aceasta se întâmplă dacă nu se ține cont în permanență de dubla interpretare posibilă a configurațiilor binare manipulate. Să luăm un exemplu:

```
mov al, 200 ; al = 11001000b = 0C8h = 200 (fără semn) = -56 (cu semn)
mov bl, -1   ; bl = 11111111b = OFFh = 255 (fără semn) = -1 (cu semn)
cmp al, bl  ; al-bl = 11001001b = C9h = -55 (cu semn) = 201 (fără semn)
              (și se setează corespunzător OF=ZF=0 și CF=SF=1)
```

Deci pe cine comparăm de fapt aici? Pe 200 cu -1 sau cum precizează valorile de la inițializare?

Sau poate pe 200 cu 255? Sau pe -56 cu -1? Sau pe -56 cu 255?

Răspuns: comparăm întotdeauna pe 0C8h cu OFFh sau în exprimare binară pe 11001000 cu 11111111. Efectul va fi unul singur: afectarea corespunzătoare a flag-urilor în urma efectuării scăderii fictive AL-BL. Modul de exprimare corect al comparației efectuate în baza 10 nu este dedus din acțiunea instrucțiunii CMP (care nu distinge absolut de loc între cele 4 variante posibile de comparare de mai sus) ci pe baza unor eventuale instrucțiuni ulterioare care vor avea ele rolul de a interpreta în unul din cele 4 moduri de mai sus comparația efectuată. Să urmărim în acest sens variantele de comparare de mai jos identificate prin utilizarea instrucțiunilor corespunzătoare de salt condiționat:

jl et1 ; evident că $200 < -1$ deci la prima vedere pare că nu este îndeplinită condiția necesară pentru efectuarea saltului... să nu uităm însă faptul că JL (Jump If Less) interpretează rezultatul comparației ca fiind cu semn (deci -55) aceasta însemnând implicit și faptul că scăderea este interpretată ca $(-56 - (-1))$ deci și operanții vor fi amândoi interpretati cu semn... cum $-56 < -1$ iată că și intuitiv condiția se verifică (pe lângă justificarea tehnică a îndeplinirii condiției de salt SF \neq OF) și deci saltul se va efectua! Deci chiar dacă programatorul a furnizat la inițializare valorile 200 și -1, utilizarea instrucțiunii JL a provocat interpretarea comparației ca fiind între -55 și -1 și nu între 200 și -1! (explicația de aici și faptul că saltul se va efectua vă poate ajuta să "demonstrați" unor colegi cum 200 poate fi mai mic decât -1 !!!)

ja et2 ; deoarece $200 > -1$ în acest caz ne-am așteptă ca saltul să se efectueze... însă utilizarea instrucțiunii JA (Jump if Above) impune interpretarea fără semn, deci varianta de comparație corectă aici este comparația lui 200 cu 255 și cum $200 > 255$ condiția nu este îndeplinită și deci saltul nu se va efectua (iată deci cum se poate “demonstra” că 200 nu este superior valorii -1 !!!). Ca o confirmare, se poate vedea că nici condiția tehnică impusă de JA nu este îndeplinită: ar trebui să avem $CF=ZF=0$, însă în cazul nostru $CF=1$ deci saltul nu se va efectua.

jb et3 ; intuitiv $200 < 255$, iar tehnic $CF=1$ deci saltul se efectuează

jg et4 ; intuitiv $-56 > -1$, iar tehnic deși $ZF=0$ nu este îndeplinită și condiția $SF = OF$ deci

; saltul nu se va efectua

Ca urmare din cele 4 situații teoretic posibile de mai sus, vom întâlni concret numai două:

- comparație fără semn (200 cu 255) - impusă de “above” sau “below”
- comparație cu semn (-56 cu -1) – impusă de “less than” sau “greater than”

Nu putem aşadar compara de fapt pe 200 cu -1 așa cum au fost specificate valorile la initializare și nici pe -56 cu 255 deoarece **interpretarea este ori cu semn ori fără semn pentru ambii operanzi!**

vii). Am studiat în exemplele anterioare modalitatea de reacție (de interpretare) a procesorului 80x86 legată de noțiunea de depășire în cazul operațiilor de adunare și de scădere. Când și cum semnalează însă procesoarele din familia 80x86 depășirea la înmulțire și respectiv la împărțire ?

“Depășirea” la înmulțire. Instrucțiunile MUL și IMUL setează $CF=1$ și $OF=1$ dacă “jumătatea” superioară a produsului (octetul superior dacă este vorba despre produs-cuvânt sau cuvântul superior dacă este vorba despre produs-dublucuvânt) este o valoare diferită de zero. Aceasta este definiția noțiunii de “depășire la înmulțire” în cazul arhitecturii 80x86. Să remarcăm faptul că nu se face distincție între MUL și IMUL și de aceea nici între CF și OF. Ori vor fi amândouă flag-urile setate la valoarea 1 cu semnificația de “depășire la înmulțire” în sensul precizat mai sus, ori vor primi amândouă valoarea 0. Iată un exemplu pe 8 biți:

```
mov al, 5
mov bl, 170
mul bl      ;AX := AL * BL = 5 * 170 = 850 = 0352h și vom avea CF=1 și
              ;deoarece octetul superior AH = 03 ≠ 0.
```

Varianta cu IMUL va furniza:

```
mov al, 5
mov bl, 170 ;170 = 0aah = -86 în interpretarea cu semn
imul bl     ;AX := AL * BL = 5 * (-86) = - 430 = 0fe52h și vom avea CF=1 și
```

;OF=1 deoarece octetul superior AH = 0feh ≠ 0.

În cazul unor operanzi pe 16 biți putem avea de exemplu:

```
val1 DW 2000h  
val2 DW 0100h  
...  
mov ax, val1  
mul val2      ;DX:AX = 00200000h și vom avea CF=1 și OF=1 deoarece  
jumătatea ;superioară a produsului DX:AX, adică registrul DX conține valoarea  
0020h ≠ 0.
```

Aceste setări nu trebuie să le interpretăm drept erori. Nu este în nici un caz vorba despre o potențială pierdere de informație ca și în cazul celorlalte depășiri - adunare, scădere sau împărțire. Aceasta deoarece chiar dacă înmulțim valorile maximale posibil a fi reprezentate pe dimensiunea operanzilor ($255 * 255$ pentru octeți și respectiv $65535 * 65535$ pentru cuvinte) tot nu se depășește dublul dimensiunii de reprezentare a operanzilor, adică spațiul pe care îl avem oricum la dispoziție prin definiție, deoarece $255 * 255 = 65025 < 65535$ (numărul maximal fără semn reprezentabil pe un cuvânt) iar $65535 * 65535 = 4\ 294\ 836\ 225 < 4\ 294\ 967\ 295$ (numărul maximal fără semn reprezentabil pe un dublucuvânt).

În cazul înmulțirii cu semn (instrucțiunea IMUL) justificarea este similară: $127 * 127 = 16129 < 32767$ (numărul maximal cu semn ce poate fi reprezentat pe 1 cuvânt), iar $32767 * 32767 = 1\ 073\ 676\ 289 < 2\ 147\ 483\ 647$ (numărul maximal cu semn reprezentabil pe un dublucuvânt).

Depășirea în cazul înmulțirii la nivelul limbajului de asamblare 80x86 este doar o semnalare a faptului că plecându-se de la operanzi octeți (respectiv cuvinte) produsul nu începe tot într-un octet (respectiv într-un cuvânt) ci este realmente nevoie de o dimensiune dublă pentru memorarea rezultatului. În acest sens, a se vedea și capitolul 1, în care din punct de vedere matematic s-a specificat clar că înmulțirea nu provoacă de fapt depășire, tocmai din cauza alocării unui spațiu suficient pentru reprezentarea produsului. În concluzie, se poate spune că din punct de vedere matematic singura operație care nu provoacă depășire este înmulțirea, însă procesoarele 80x86 promovează totuși noțiunea de "depășire la înmulțire" pentru a diferenția între situațiile în care produsul începe într-un spațiu de dimensiunea operanzilor și în care nu.

Situațiile în care produsul începe pe dimensiunea operanzilor vor fi caracterizate de setările $CF = OF = 0$ (nu avem deci depășire la înmulțire). Iată un exemplu:

```
mov al, 5  
mov bl, 51  
mul bl      ; AX := AL * BL = 5 * 51 = 255 = 00ffh și vom avea CF=0 și  
OF=0          ; deoarece octetul superior AH = 0.
```

Depășirea la împărțire. În cazul împărțirii, specificarea acestei operații sub forma

(I)DIV *operand*

presupune că operandul specificat este împărțitorul (posibil a fi reprezentat fie pe 8 fie pe 16 biți) iar deîmpărțitul este considerat implicit în AX (dacă *operand* este octet) sau în DX:AX (dacă împărțitorul este cuvânt). Efectuarea operației are ca efect:

AX : operand pe 8 biți = câtul în AL și restul în AH;
DX:AX / operand pe 16 biți = câtul în AX și restul în DX;

În cazul împărțirii depășirea apare atunci când rezultatul împărțirii nu începe în spațiul rezervat conform definiției pentru reprezentare, mai exact, când câtul nu începe în AL sau respectiv AX. Într-o astfel de situație, procesorul 80x86 emite o intrerupere 0, execuția terminându-se cu un mesaj furnizat de către rutina de tratare a intreruperii 0, de genul “Divide by zero”, “Zero divide” sau “Divide overflow” (în funcție de tipul de procesor și/sau de SO instalat). Pare ciudat la prima vedere că o împărțire prin 0 (de genul *div bh* cu *bh* = 0) ce practic nu se poate efectua din punct de vedere matematic este tratată similar ca efect din punct de vedere al limbajului de asamblare cu o împărțire care matematic se poate efectua. Secvența

```
mov ax,60000  
mov bl,2  
div bl
```

ar trebui să furnizeze din punct de vedere matematic câtul 30000. Însă conform definiției împărțirii DIV acest cât trebuie memorat în registrul AL, de dimensiune octet. Cum cea mai mare valoare reprezentabilă pe 1 octet este 255, este evident astfel că din punct de vedere al limbajului de asamblare împărțirea de mai sus nu se poate nici ea efectua (similar cu o situație de tip *div 0*) și ca urmare înțelegem acum decizia proiectanților de a trata tot prin emiterea unei intreruperi 0 și o situație de genul celei de mai sus. Să remarcăm în acest sens și faptul că mesajul “Divide overflow” (depășire la împărțire) este acceptat în acest context ca similar unui “Divide by zero”.

viii). Una dintre erorile logice frecvente pe care o fac programatorii neexperimentați este de a confunda exprimările “*numere cu semn*” și “*numere fără semn*” cu exprimările “*numere negative*” și respectiv “*numere pozitive*”. Numere cu semn nu înseamnă automat numere negative ! Numerele cu semn sunt fie pozitive, fie negative. Numerele fără semn sunt întotdeauna pozitive.

Ce concluzii vom trage relativ la modul de interpretare (cu semn sau fără semn) din enunțul unei probleme care cere efectuarea unei anumite acțiuni “*dacă numărul v este (strict) negativ?*” În primul rând vom concluziona că este vorba despre interpretarea cu semn. Se pune însă întrebarea: cum vom testa practic dacă un număr cu semn este negativ sau nu? (să presupunem că *v* este octet). Fiind vorba despre interpretarea cu semn, dacă primul bit al configurației binare este 1 atunci numărul este negativ. Deci totul se reduce la un test asupra

primului bit din reprezentarea numărului. Iată două alternative pentru realizarea unui astfel de test:

- a). Realizăm o deplasare a primului bit în CF și testăm valoarea sa printr-o instrucțiune adecvată de salt condiționat. Secvența

```
mov al,v      ;pentru a nu afecta destructiv conținutul variabilei v  
shl al,1     ;shift stânga cu 1 poziție pentru ca primul bit să treacă în CF.  
jc este_negativ ;dacă CF=1 atunci salt la eticheta este_negativ
```

asigură testarea faptului dacă variabila *v* este sau nu un număr negativ.

- b). Utilizăm instrucțiunea **cmp** pentru o comparație în raport cu 0:

```
cmp v,0      ;scădere fictivă v-0  
jl este_negativ ;dacă v<0 atunci salt la eticheta este_negativ
```

sau alternativ

```
cmp 0,v      ; scădere fictivă 0-v  
jg este_negativ ;dacă 0>v atunci salt la eticheta este_negativ
```

ix). Am văzut că la nivelul efectuării operațiilor de adunare sau scădere procesorul 80x86 nu diferențiază între adunări/scăderi cu semn sau fără semn (tehnic vorbind ele se efectuează drept operații binare cu rezultat interpretabil **ulterior** drept cu semn sau fără). Totuși, în momentul în care se pune problema exprimării în baza 10 a unei operații de adunare sau scădere ne punem întrebarea: cum să exprimăm semantic corect operanzei operației respective pentru ca aceste exprimări să fie consistente cu interpretarea rezultatului final obținut? Mai concret:

00000101 + <u>11111110</u> (1) 00000011	(= 5 în ambele interpretări) (= 254 fără semn și -2 în interpretarea cu semn) (= 3 în ambele interpretări ale configurației pe 8 biți)
---	--

reprezintă $5 + 254 = 259$ ($= 1\ 00000011$ – configurație pe 9 biți!) sau reprezintă $5 + (-2) = 3$? După cum vom vedea și aici răspunsul este că putem interpreta în ambele moduri și să justificăm astfel ca două reacții separate modul de setare al flag-urilor CF și respectiv OF.

Datorită cifrei de transport vom avea CF=1 (independent de interpretarea operanzei sau a rezultatului final drept cu semn sau fără semn, deoarece este vorba despre o consecință tehnică a modului de efectuare a operației binare de adunare). Ca urmare în interpretarea fără semn avem depășire (evident, deoarece $259 > 255$, adică decât numărul maxim reprezentabil pe 1 octet).

Ce se întâmplă cu OF ? Rularea secvenței

```
mov al, 5    ; = 5 în ambele interpretări  
mov bl, 254  ; = -2 în interpretarea cu semn  
add al, bl   ; AL := AL+BL = 5+(-2) = 3
```

nu setează flagul OF la valoarea 1, deci situația de mai sus nu este considerată “depășire” în interpretarea cu semn! Din punct de vedere al justificării modului de setare a flag-ului OF secvența de mai sus ar fi mai corectă dacă ar fi scrisă:

```
mov al, 5  
mov bl, -2  
add al, bl   ; deci 5 + (-2) = 3
```

și este evident că în această interpretare nu este vorba despre nici o depășire (și de aceea și OF = 0).

Să ne reamintim în acest context și exemplele date la prezentarea RDA și RDS de la paginile ??-??: adunarea $100 + 50 = 150$ va semnala depășire (*signed overflow* - conform RDA), iar scăderile $130 - 42$ (interpretată ca $-126 - 42 = -168 \notin [-128..127]$) și $42 - 130$ (interpretată ca scăderea $42 - (-126) = +168 \notin [-128..127]$) produc la rândul lor *signed overflow* și ca urmare OF=1.

4.1.2. Instrucțiuni de conversie (distructivă)

CBW	conversie octet conținut în AL la cuvânt în AX (extensie de semn)	-
CWD	conversie cuvânt conținut în AX la dublu cuvânt în DX:AX (extensie de semn)	-
CWDE	conversie cuvânt din AX în dublucuvânt în EAX (extensie de semn)	-
MOVZX d, s	încarcă în d (REGISTRU !), de dimensiune mai mare decât s (registru sau operand din memorie!), conținutul lui s fară semn (zero extension)	-
MOVSX d, s	încarcă în d (REGISTRU !), de dimensiune mai mare decât s (registru sau operand din memorie!), continutul lui s cu semn (sign extension) http://www.c-jump.com/CIS77/ASM/DataTypes/T77_0270_sext_example_mosvx.htm !!	-

Instrucțiunea **CBW** convertește octetul cu semn din AL în cuvântul cu semn AX (extinde bitul de semn al octetului din AL la nivelul cuvântului din AX, modificând distructiv conținutul registrului AH). De exemplu,

```
mov al, -1      ; AL = 0FFh
cbw              ;extinde valoarea octet -1 din AL în valoarea cuvânt -1 din AX (0FFFFh).
```

Analog, pentru conversia cu semn cuvânt - dublu cuvânt, instrucțiunea **CWD** extinde cuvântul cu semn din AX în dublucuvântul cu semn DX:AX. Exemplu:

```
mov ax,-10000    ; AX = 0D8F0h
cwd              ;obține valoarea -10000 în DX:AX (DX = 0FFFFh ; AX = 0D8F0h)
cwde             ;obține valoarea -10000 în EAX   (EAX = 0FFFFD8F0h)
```

Conversia fără semn se realizează prin zerorizarea octetului sau cuvântului superior al valorii de la care s-a plecat. (de exemplu, prin mov ah,0 sau mov dx,0 - efect similar se obține prin aplicarea instr. **MOVZX**)

De ce coexistă CWD cu CWDE ? CWD trebuie să rămână din rațiuni de backwards compatibility și din rațiuni de funcționalitate a instrucțiunilor (I)MUL și (I)DIV.

MOV ah, 0c8h		
MOVSX ebx, ah	; EBX = FFFFFFFC8h	MOVSX ax,[v] ; MOVSX ax, byte ptr DS:[offset v]
MOVZX edx, ah	; EDX = 000000C8h	MOVZX eax, [v] ; syntax error – op.size not specified

Atenție ! NU sunt acceptate sintactic:

CBD	CWDE EBX, BX	MOVSX EAX, [v]
CWB	CWD EDX, AX	MOVZX EAX, [EBX]
CDW	MOVZX AX, BX	MOVSX dword [EBX], AH
CDB !!! (super-înghesuire!! ☺)	MOVSX EAX, -1	CBW BL

4.1.3. Impactul reprezentării little-endian asupra accesării datelor (pag.119 – 122 – curs).

Dacă programatorul utilizează datele consistent cu dimensiunea de reprezentare stabilită la definire (ex: accesarea octetilor drept octeți și nu drept secvențe de octeți interpretate ca și cuvinte sau dublucuvinte, accesarea de cuvinte ca și cuvinte și nu ca perechi de octeți, accesarea de dublucuvinte ca și dublucuvinte și nu ca secvențe de octeți sau de cuvinte) atunci instrucțiunile limbajului de asamblare vor ține cont în mod AUTOMAT de modalitatea de reprezentare little-endian. Ca urmare, dacă se respectă această condiție programatorul nu trebuie să intervină suplimentar în nici un fel pentru a asigura corectitudinea accesării și manipulării datelor utilizate. Exemplu:

```
a db 'd', -25, 120
b dw -15642, 2ba5h
c dd 12345678h
...
mov al, [a] ;se încarcă în AL codul ASCII al caracterului 'd'
```

`mov bx, [b]` ;se încarcă în BX valoarea -15642; ordinea octetilor în BX va fi însă inversată față de reprezentarea în memorie, deoarece numai reprezentarea *în memorie* folosește reprezentarea *little-endian!* În regiștri datele sunt memorate conform reprezentării structurale normale, echivalente unei reprezentări *big endian*.

`mov edx, [c]` ;se încarcă în EDX valoarea dublucuvânt 12345678h

Dacă însă se dorește accesarea sau interpretarea datelor sub o formă diferită față de modalitatea de definire atunci trebuie utilizate conversii explicite de tip. În momentul utilizării conversiilor explicite de tip programatorul trebuie să își asume însă întreaga responsabilitate a interpretării și accesării corecte a datelor. În astfel de situații programatorul este obligat să conștientizeze particularitățile de reprezentare little-endian (ordinea de plasare a octetilor în memorie) și să utilizeze modalități de accesare a datelor în conformitate cu aceasta Ex pag.120-122.

segment data

`a dw 1234h` ;datorită reprezentării little-endian, în memorie octetii sunt plasați astfel:
`b dd 11223344h` ;34h 12h 44h 33h 22h 11h
; adresa a a+1 b b+1 b+2 b+3

`c db -1`

segment code

`mov al, byte [a+1]` ;accesarea lui a drept octet, efectuarea calculului de adresă a+1, selectarea octetului de la adresa a+1 (octetul de valoare 12h) și transferul său în registrul AL.

`mov dx, word [b+2]` ;dx:=1122h

`mov dx, word [a+4]` ;dx:=1122h deoarece b+2 = a+4 , în sensul că aceste expresii de tip pointer desemnează aceeași adresă și anume adresa octetului 22h.

`mov dx, [a+4]` ;această instrucțiune este echivalentă cu cea de mai sus, nefiind realmente necesară ;utilizarea operatorului de conversie WORD

mov bx, [b]	;bx:=3344h
mov bx, [a+2]	;bx:=3344h, deoarece ca adresa b = a+2.
mov ecx, dword [a]	;ecx:=33441234h, deoarece dublucuvântul ce începe la adresa a este format din octeții 34h 12h 44h 33h care (datorită reprezentării little-endian) înseamnă de fapt ;dublucuvântul 33441234h.
mov ebx, [b]	; ebx := 11223344h
mov ax, word [a+1]	; ax := 4412h
mov eax, word [a+1]	; ax := 22334412h
mov dx, [c-2]	; DX := 1122h deoarece c-2 = b+2 = a+4
mov bh, [b]	;bh := 44h
mov ch, [b-1]	;ch := 12h
mov cx, [b+3]	; CX := 0FF11h

4.2. OPERAȚII

4.2.1. Operații aritmetice

Operanzii sunt reprezentați în cod complementar (vezi 1.5.2.). Microprocesorul realizează adunările și scăderile "văzând" doar configurații de biți și nu numere cu semn sau fără. Regulile de efectuare a adunării și scăderii presupun adunarea de configurații binare, fără a fi nevoie de a interpreta operanzii drept cu semn sau fără semn anterior efectuării operației! Deci, la nivelul acestor instrucțiuni, interpretarea "cu semn" sau "fără semn" rămâne la latitudinea programatorului, nefiind nevoie de instrucțiuni separate pentru adunarea/scăderea cu semn față de adunarea/scăderea fără semn.

Adunarea și scăderea se efectuează întotdeauna la fel (adunând sau scăzând configurații binare) indiferent de semnul (interpretarea) acestor configurații! După cum vom vedea acest lucru nu este valabil și pentru înmulțire și

împărțire. În cazul acestor operații trebuie să știm apriori dacă operanții vor fi interpretați drept cu semn sau fără semn. De exemplu, fie doi operanzi A și B reprezentați fiecare pe câte un octet:

$$A = 9Ch = 10011100b \quad (= 156 \text{ în interpretarea fără semn și } -100 \text{ în interpretarea cu semn})$$
$$B = 4Ah = 01001010b \quad (= 74 \text{ atât în interpretarea fără semn cât și în interpretarea cu semn})$$

Microprocesorul realizează adunarea $C = A + B$ și obține

$$C = E6h = 11100110b \quad (= 230 \text{ în interpretarea fără semn și } -26 \text{ în interpretarea cu semn})$$

Se observă deci că simpla adunare a configurațiilor de biți (fără a ne fixa neapărat asupra unei interpretări anume la momentul efectuării adunării) asigură corectitudinea rezultatului obținut, atât în interpretarea cu semn cât și în cea fără semn.

INSTRUCTIUNI ARITMETICE – tabel pag.123 (curs)

4.2.1.3. Exemple și exercitii propuse – pag.129-130 (curs)

4.2.2. Operații logice pe biți(AND, OR, XOR și NOT) – pag.131 (curs)

Instrucțiunea AND este indicată pentru izolarea unui anumit bit sau pentru forțarea anumitor biți la valoarea 0.

Instrucțiunea OR este indicată pentru forțarea anumitor biți la valoarea 1.

Instrucțiunea XOR este indicată pentru schimbarea valorii unor biți din 0 în 1 sau din 1 în 0.

Instrucțiunea NOT este indicată pentru complementarea conținutului operandului (reg/mem).

4.2.3. Deplasări și rotiri de biți

Instrucțiunile de *deplasare* de biți se clasifică în:

- Instrucțiuni de deplasare logică
 - stânga - **SHL**
 - dreapta - **SHR**

Instrucțiunile de *rotire* a biților în cadrul unui operand se clasifică în:

- Instrucțiuni de rotire fără carry
 - stânga - **ROL**
 - dreapta - **ROR**

- Instrucțiuni de deplasare aritmetică
 - stânga - **SAL**
 - dreapta - **SAR**

Pentru a defini deplasările și rotirile să considerăm ca și configurație inițială un octet $X = abcdefgh$, unde a-h sunt cifre binare, h este cifra binară de rang 0, a este cifra binară de rang 7, iar k este valoarea existentă în CF ($CF=k$). Atunci avem:

```

SHL X,1 ;rezultă X = bcdefgh0 și CF = a
SHR X,1 ;rezultă X = 0abcdefg și CF = h
SAL X,1 ; identic cu SHL
SAR X,1 ;rezultă X = aabcdefg și CF = h
ROL X,1 ;rezultă X = bcdefgha și CF = a
ROR X,1 ;rezultă X = habcdefg și CF = h
RCL X,1 ;rezultă X = bcdefghk și CF = a
RCR X,1 ;rezultă X = kabcdefg și CF = h

```

INSTRUCȚIUNILE DE DEPLASARE ȘI ROTIRE DE BIȚI – tabel – pag.134 (curs)

4.3. RAMIFICĂRI, SALTURI, CICLURI

4.3.1. Saltul necondiționat

În această categorie intră instrucțiunile JMP (echivalentul instrucțiunii GOTO din alte limbaje), CALL (apelul de procedură înseamnă transferul controlului din punctul apelului la prima instrucțiune din procedura apelată) și RET (transfer control la prima instrucțiune executabilă de după CALL).

JMP <i>operand</i>	Salt necondiționat la adresa determinată de operand	-
CALL <i>operand</i>	Transferă controlul procedurii determinată de operand	-
RET <i>[n]</i>	Transferă controlul instrucțiunii de după CALL	-

4.3.1.1. Instrucțiunea JMP

Instrucțiunea de salt necondiționat **JMP** are sintaxa

JMP *operand*

unde *operand* este o **etichetă**, un **registru** sau o **variabilă de memorie ce conține o adresă**. Efectul ei este transferul necondiționat al controlului la instrucțiunea ce urmează etichetei, la adresa dată de valoarea registrului sau constantei, respectiv la adresa conținută în variabila de memorie. De exemplu, după execuția secvenței

```

    mov ax,1
    jmp AdunaDoi
AdunaUnu:   inc  eax
              jmp  urmare
AdunaDoi:   add  eax,2
urmare: .   .   .
  
```

registru AX va conține valoarea 3. Instrucțiunile **inc** și **jmp** dintre etichetele *AdunaUnu* și *AdunaDoi* nu se vor executa, decât dacă se va face salt la *AdunaUnu* de altundeva din program.

După cum am menționat, saltul poate fi făcut și la o adresă memorată într-un registru sau într-o variabilă de memorie. Exemple:

(1) mov eax, etich
 jmp eax ;operand registru
 ; jmp [eax] ?
 etich: . . .

(2) segment data
 Salt DD Dest ;Salt := offset Dest
 . . .
 segment cod
 . . .
 jmp [Salt] ;salt NEAR
 . ;operand variabilă de memorie
 Dest : . . .

Dacă în cazul (1) dorim înlocuirea operandului destinație registru cu un operand destinație variabilă de memorie, o soluție posibilă este:

(1') b resd 1
 . . .
 mov [b], DWORD etich ; b := offset etich
 jmp [b] ; salt NEAR – operand variabilă de memorie JMP DWORD PTR DS:[offset_b]

Exemplul 4.3.1.2. – pag.142-143 (curs) – modul de transfer al controlului la o eticheta.

4.3.2. Instrucțiuni de salt condiționat

4.3.2.1. Comparări între operanzi

CMP <i>d,s</i>	comparație valori operanzi (nu modifică operanzele) (execuție fictivă <i>d - s</i>)	OF,SF,ZF,AF,PF și CF
TEST <i>d,s</i>	execuție fictivă <i>d AND s</i>	OF = 0, CF = 0 SF,ZF,PF - modificați, AF - nedefinit

Instrucțiunile de salt condiționat se folosesc de obicei în combinație cu instrucțiuni de comparare. De aceea, semnificațiile instrucțiunilor de salt rezultă din semnificația operanzilor unei instrucțiuni de comparare. În afara testului de egalitate pe care îl poate efectua o instrucțiune CMP este de multe ori necesară determinarea relației de ordine dintre două valori. De exemplu, se pune întrebarea: numărul 11111111b (= FFh = 255 = -1) este mai mare decât 00000000b (= 0h = 0)? Răspunsul poate fi și da și nu! Dacă cele două numere sunt considerate fără semn, atunci primul are valoarea 255 și este evident mai mare decât 0. Dacă însă cele două numere sunt considerate cu semn, atunci primul are valoarea -1 și este mai mic decât 0.

Instrucțiunea CMP nu face distincție între cele două situații, deoarece aşa după cum am precizat și în 4.2.1.1. adunarea și scăderea se efectuează întotdeauna la fel (adunând sau scăzând configurații binare) indiferent de semnul (interpretarea) acestor configurații. Ca urmare, nu este vorba de a interpreta cu semn sau fără semn *operanzii* scăderii fictive *d-s*, ci **rezultatul** final al acesteia! Rolul de a interpreta în mod diferit (cu semn sau fără semn) rezultatul final al comparației revine diverselor instrucțiuni de salt condiționat prezentate în 4.3.2.2.

4.3.2.2. Salturi conditionate de flaguri

În tabelul 4.1. (pag.146 – curs) se prezintă instrucțiunile de salt condiționat împreună cu semnificația lor și cu precizarea valorilor flagurilor în urma cărora se execută salturile respective. Precizăm că pentru toate instrucțiunile de salt sintaxa este aceeași, și anume

<instrucțiune_de_salt> etichetă

Semnificația instrucțiunilor de salt condiționat este dată sub forma "**salt dacă operand1 <> relație >> față de operand2**" (unde cei doi operanzi sunt obiectul unei instrucțiuni anterioare CMP sau SUB) sau referitor la valoarea concretă setată pentru un anumit flag. După cum se observă și din condițiile ce trebuie verificate, instrucțiunile ce se află într-o aceeași linie a tabelului sunt echivalente.

Când se compară două numere cu semn se folosesc termenii "**less than**" (mai mic decât) și "**greater than**" (mai mare decât), iar când se compară două numere fără semn se folosesc termenii "**below**" (inferior, sub) și respectiv "**above**" (superior, deasupra, peste).

4.3.2.3. Exemple comentate..... pag.148-162 (curs).

- discutie si analiza comparativa a conceptelor de : reprezentari cu semn vs. fara semn, depasire, modul de actiune al instructiunilor de salt conditionat.

4.3.3. Instrucțiuni de ciclare (pag.162 – 164 – curs).

Ele sunt: **LOOP**, **LOOPE**, **LOOPNE** și **JECXZ**. Sintaxa lor este

<instrucțiune> etichetă

Instrucțiunea **LOOP** comandă reluarea execuției blocului de instrucțiuni ce începe la *etichetă*, atât timp cât valoarea din registrul ECX este diferită de 0. **Se efectuează întâi decrementarea registrului ECX și apoi se face testul și eventual saltul.** Saltul este "scurt" (max. 127 octeți - atenție deci la "distanța" dintre LOOP și etichetă!). (short jump is out of range!).

În cazul în care condițiile de terminare a ciclului sunt mai complexe se pot folosi instrucțiunile **LOOPE** și **LOOPNE**. Instrucțiunea **LOOPE** (*LOOP while Equal*) diferă față de LOOP prin condiția de terminare, ciclul terminându-se fie dacă ECX=0, fie dacă ZF=0. În cazul instrucțiunii **LOOPNE** (*LOOP while Not Equal*) ciclul se va termina fie dacă ECX=0, fie dacă ZF=1. Chiar dacă ieșirea din ciclu se face pe baza valorii din ZF, decrementarea lui ECX are oricum loc. **LOOPE** mai este cunoscută și sub numele de **LOOPZ** iar **LOOPNE** mai este cunoscută și sub numele de **LOOPNZ**. Se folosesc de obicei precedate de o instrucțiune CMP sau SUB.

JECXZ (*Jump if ECX is Zero*) realizează saltul la eticheta operand numai dacă ECX=0, fiind utilă în situația în care se dorește testarea valorii din ECX înaintea intrării într-o buclă. În exemplul următor instrucțiunea JECXZ se folosește pentru a se evita intrarea în ciclu dacă ECX=0:

... . . .

jecxz MaiDepart Bucla: Mov BYTE [esi],0 inc esi loop Bucla MaiDepart: . . .	;dacă ECX=0 se sare peste buclă ;inițializarea octetului curent ;trecere la octetul următor ;reluare ciclu sau terminare
--	---

La întâlnirea instrucțiunii LOOP cu ECX=0, ECX este decrementat, obținându-se valoarea 0FFFh (= -1, deci o valoare diferită de 0), ciclul reluându-se până când se va ajunge la valoarea 0 în ECX, adică de încă 4.294.967.296 ori!

Este important să precizăm aici faptul că nici una dintre instrucțiunile de ciclare prezentate nu afectează flag-urile.

loop Bucla	dec ecx și jnz Bucla
------------	-------------------------

deși semantic echivalente, nu au exact același efect, deoarece spre deosebire de LOOP, instrucțiunea DEC afectează indicatorii OF, ZF, SF și PF.

4.3.4. Instrucțiunile CALL și RET

Apelul unei proceduri se face cu ajutorul instrucțiunii **CALL**, acesta putând fi *apel direct* sau *apel indirect*. Apelul direct are sintaxa

CALL operand

Asemănător instrucțiunii JMP și instrucțiunea **CALL** transferă controlul la adresa desemnată de operand. În plus față de aceasta, înainte de a face saltul, instrucțiunea CALL salvează în stivă adresa următoarei instrucțiuni de după CALL (adresa de revenire). Cu alte cuvinte, avem echivalență

CALL operand A: . . .	\Leftrightarrow push A jmp operand
--------------------------	--

Terminarea execuției secvenței apelate este marcată de întâlnirea unei instrucțiuni **RET**. Aceasta preia din stivă adresa de revenire depusă acolo de CALL, predând controlul la instrucțiunea de la această adresă. Sintaxa instrucțiunii RET este **RET [n]** unde *n* este un parametru optional. El indică eliberarea din stivă a *n* octeți aflați sub adresa de revenire.

Instrucțiunea RET poate fi ilustrată prin echivalență

RET n (revenire near) \Leftrightarrow	B dd ? .	.	.
	pop [B] add esp,[n] jmp [B]		

De cele mai multe ori, după cum este și natural, instrucțiunile CALL și RET apar în următorul context

etichetă_procedură:

.	.	.
	ret n	
.	.	.
	CALL	<i>etichetă_procedură</i>

Instrucțiunea CALL poate de asemenea prelua adresa de transfer dintr-un registru sau dintr-o variabilă de memorie. Un asemenea gen de apel este denumit *apel indirect*. Exemple:

call ebx	;adresă preluată din registru
call [vptr]	;adresă preluată din memorie (similar cu apelul funcției <i>printf</i>)

Rezumând, operandul destinație al unei instrucțiuni CALL poate fi:

- numele unei proceduri
- numele unui registru în care se află o adresă
- o adresă de memorie

MNEMONICA	SEMNIFICATIE (salt dacă..<<relație>>)	Condiția verificată
JB JNAE JC	este inferior nu este superior sau egal există transport	CF=1
JAE JNB JNC	este superior sau egal nu este inferior nu există transport	CF=0
JBE JNA	este inferior sau egal nu este superior	CF=1 sau ZF=1
JA JNBE	este superior nu este inferior sau egal	CF=0 și ZF=0
JE JZ	este egal este zero	ZF=1
JNE JNZ	nu este egal nu este zero	ZF=0
JL JNGE	este mai mic decât nu este mai mare sau egal	SF \neq OF
JGE JNL	este mai mare sau egal nu este mai mic decât	SF=OF
JLE JNG	este mai mic sau egal nu este mai mare decât	ZF=1 sau SF \neq OF
JG JNLE	este mai mare decât nu este mai mic sau egal	ZF=0 și SF=OF
JP JPE	are paritate paritatea este pară	PF=1
JNP JPO	nu are paritate paritatea este impară	PF=0
JS	are semn negativ	SF=1
JNS	nu are semn negativ	SF=0
JO	există depășire	OF=1
JNO	nu există depășire	OF=0

Tabelul 4.1. Instrucțiunile de salt condiționat

SHORT JMP vs. LONG JMP

Dinnou:

Mov eax, 89

.....
Jmp Maideparte ; JMP is not restricted to any “distance”

Resd 1000h; The distance between LOOP and the label Dinnou is > 127 bytes so it is not a short jump !

Maideparte:

Mov ebx, 17

.....

Loop Dinnou ; syntax error: short jump is out of range + warning: byte data exceeds bounds

Daca inlocuim Loop Dinnou cu echivalentul

```
dec ecx  
jnz Dinnou
```

NU mai obtinem eroare deoarece

In TASM and MASM the short jump condition (ie maximum 127 bytes distance) is imposed both at the level of LOOP type instructions and at the level of conditional jump instructions.

In NASM the restriction is valid only for LOOP type instructions, the conditional jump instructions are no longer subject to this restriction.

Totusi, exista o diferenta importanta intre cele 2 variante: Loop NU afecteaza flag-urile, in timp ce DEC DA !!

Daca vrem o varianta echivalenta care sa NU afecteze flag-urile:

Dinnou:

Mov eax, 89

.....

Jmp Maideparte

Resd 1000h; The distance between LOOP and the label Dinnou is > 127 bytes so it is not a short jump !

MaiAproape:

Jmp Dinnou

Maideparte:

Mov ebx, 17

.....

Loop MaiAproape ; short jump

Constante de tip string. Reprezentare in memorie si utilizare in cadrul unor instructiuni de transfer.

In cazul initializarii unei zone de memorie cu valori de tip constante string (sizeof > 1) tipul de data utilizat in definire (dw, dd, dq) are rol doar de rezervare a spatiului dorit, **ordinea de “umplere” a zonei de memorie respective fiind ordinea in care apar caracterele (octetii) in cadrul constantei de tip string:**

a6 dd '123', '345','abcd' ; se vor defini 3 dublucuvinte continutul lor fiind
31 32 33 00|33 34 35 00|61 62 63 64|

a6 dd '1234' ; 31 32 33 34

a6 dd '12345' , 'abc'; 31 32 33 34|35 00 00 00| 61 62 63 00|

a7 dw '23','45' | 32 33 | 34 35| - 2 cuvinte = 1 doubleword

a7 dw '2345' - 2 cuvinte - 32 33|34 35|

a7 dw '23456' - 3 cuvinte - 32 33|34 35|36 00|

In C ‘x’(1 byte) “x” (2 bytes = ‘x’ ‘\0’)

In ASAMBLARE '...' = "..."

a8 dw '1', '2', '3' - 3 cuvinte - 31 00|32 00|33 00

a9 dw '123' - 2 cuvinte - 31 32|33 00

Urmatoarele definitii produc aceeasi configuratie de memorie

```
dd  'ninechars'      ; constanta string doubleword
dd  'nine','char','s' ; 3 dublucuvinte
db  'ninechars',0,0,0 ; “umplere” zona prin secenta de octeti
```

Definitia din documentatia oficiala spune:

A character constant with more than one byte will be arranged (WHERE ???) with little-endian order in mind: if you code

```
        mov eax, 'abcd'  (EAX = 0x64636261)
```

then the constant generated is not 0x61626364, but 0x64636261, so that if you were then to store the value into memory, it would read abcd rather than dcba. This is also the sense of character constants understood by the Pentium's CPUID instruction.

Esenta acestei definitii este ca VALOAREA asociata unei constante de tip string 'abcd' este de fapt 'dcba' (adica acesta este modul de STOCARE al acestei constante în TABELA DE CONSTANTE)

Mov dword [a], '2345' va aparea in OllyDbg astfel:

```
mov dword ptr DS:[401000],35343332
```

iar in memoria rezervata pt a va aparea |32 33 34 35|

....dar daca folosim a data definition like a7 dd '2345' the corresponding memory layout will be NO little-endian representation, but |32 33 34 35|

Astfel, comparativ si in rezumat (constante tip string vs. constante numerice) :

```
a7 dd '2345'      ; |32 33 34 35|
a8 dd 12345678h ; |78 56 34 12|
```

.....

```
mov eax, '2345' → EAX = '5432' = 35 34 33 32
mov ebx, [a7]   → EBX = '5432' = 35 34 33 32
```

```
mov ecx, 12345678h ; ECX = 12345678h (sizeof <12345678h> = 4 bytes)
mov dword [var1], '12345678' ; var1 = '1234' (31 32 33 34)
                           ( sizeof '12345678' = 8 bytes)
```

```
mov edx, [a8]    → EDX = 12345678h
```

In cazul in care se foloseste DB ca directiva de definire a datelor e normal ca ordinea octetilor data in specificarea constantei sa se regaseasca si in memorie in mod similar, deci acest caz nu comporta analiza si discutii suplimentare.

a66 TIMES 4 db '13' ; 31 33 31 33 31 33 31 33 echiv cu ...db '1','3'

a67 TIMES 4 dw '13' ; 31 33 31 33 31 33 31 33 - cele doua moduri de definire diferite produc acelasi rezultat !!!

a68 TIMES 4 dw '1','3' ; 31 00 |33 00| 31 00 |33 00|31 00| 33 00| 31 00|33 00

a69 TIMES 4 dd '13' ; 31 33 00 00 | 31 33 00 00 | 31 33 00 00 | 31 33 00 00

Deci, un sir constant in NASM se comporta ca si cum ar exista o „zonă de memorie” alocată anterior acestor constante (ea exista !! si se numeste TABELA DE CONSTANTE!!), unde acestea sunt stocate folosind reprezentarea little-endian !!

Dpdv al REPREZENTARII valoarea asociată unei constante de tip string este INVERSA SA !!! (vezi și “definiția oficială” de mai sus!).

Practic, dacă inițializăm o zonă de memorie cu o constantă de tip string (fie prin directive de definire a datelor, fie prin mov [zona_de_memorie], constanta_string) ordinea în care caracterele vor fi depuse în memorie este ordinea în care ele apar în scrierea pe hârtie a constantei de tip string !!!

Dacă inițializăm conținutul unui registru cu o constantă de tip string, caracterele se vor depune în ordine inversă apariției lor în scrierea pe hârtie a constantei de tip string !

From a NUMERIC VALUE point of view ??? (true both in C and assembler ???)

“abcd” = THE ADDRESS FROM MEMORY OF THIS CONSTANT (in C)

“abcd”[1] = ‘b’

“abcd”[251] = ??

Clasificarea conversiilor

a). Distructive - cbw, cwd, cwde, cdq, movzx d,s, movsx d,s, mov ah,0, mov dx,0, mov edx,0 (instructiuni)

Nedistructive – byte, word, dword, qword (operatori)

b). Cu semn - cbw, cwd, cwde, cdq, movsx d,s

Fara semn – movzx d,s, mov ah,0, mov dx,0, mov edx,0, byte, word, dword, qword

c). prin largire (by enlargement) – toate cele distructive, word, dword, qword

prin ingustare (by narrowing) – byte, word, dword

e = a+b+c – integer to float = conversii implice

- Float to integer se realizeaza NU prin conversii ci prin aplicare de functii predefinite ale limbajului (floor, ceil, trunc etc).

Contorul de locații și

aritmetică de pointeri

```
SEGMENT data
```

```
a db 1,2,3,4 ; 01 02 03 04
```

```
lg db $-a ; 04
```

lg db \$-data ; syntax error – expression is not simple or relocatable

(în TASM \$-data=4 – deci același efect ca mai sus, deoarece în TASM, MASM offset(nume_segment)=0 !!!; în NASM NU, offset(data) = 00401000 !!!)

lg db a-data ; syntax error – expression is not simple or relocatable

lg2 dw data-a; asamblorul ne lasă, însă obținem eroare la link-editare – “Error in file at 00129 – Invalid fixup recordname count....”

```
db a-$ ; = -5 = FB
```

```
c equ a-$ ; 0-6 = -6 = FA
```

```
d equ a-$ ; 0-6 = -6 = FA
```

```
e db a-$ ; 0-6 = -6 = FA
```

```
x dw x ; 07 10 !!!!
```

```
x db x ; syntax error !
```

```
x1 dw x1 ; x1 = offset(x1) 09 00 la asamblare si in final 09 10
```

```
db lg-a ; 04
```

```
db a-lg ; = -4 = FC
```

```
db [$-a] ; expression syntax error
```

```
db [lg-a] ; expression syntax error
```

lg1 EQU lg1 ; lg1 = 0 NASM consideră că e corect ! (IT IS A NASM BUG !!!)

lg1 EQU lg1-a ; lg1 = 0 – DE CE ????? Bug NASM ! Orice se evalueaza la zero în dreapta este acceptat chiar daca este definiție recursivă ! (de asta e BUG !) Dacă a este primul element definit în segment consideră a=0 (offset-ul față de începutul segmentului) și va furniza lg1=0; dacă a este definit altundeva și offset(a) ≠ 0, atunci va furniza eroare de sintaxă = “Macro abuse, recursive EQUs”

```
g34 dw c-2 ; -8 = F8
```

b dd a-start ; syntax error ! (a definit aici, start definit altundeva) expression is not simple or relocatable !!!

dd start-a ; MERGE !!! (**start definit altundeva și a definit aici !**) – rez=POINTER !!!
deoarece etichetele NU fac parte din același segment și este interpretată ca scădere FAR =
pointer !!!

dd start-start1 ; MERGE !!! (deoarece amandouă etichetele sunt definite în același
segment !!!) ; rez=SCALAR !!!! pt că aici avem scădere de offset-uri definite în același segment !

segment code use32

start:

```
mov ah, lg1    ; AH = 0
mov bh, c      ; BH = -6 = FA
```

```
mov ch, lg      ; OBJ format can handle only 16 or 32 byte relocation
                  (offset NU începe în 1 byte !!!)
mov ch, lg-a    ; CH = 04
mov ch, [lg-a]   ; mov ch, byte ptr DS:[4] – Mem. access violation – cel mai probabil...
```

```
mov cx, lg-a    ; CX = 4
mov cx, [lg-a]   ; mov WORD ptr DS:[4] – Mem. access violation – cel mai probabil...
```

mov cx, \$-a ; invalid operand type !!!!! (\$ generat aici, a altundeva)

mov cx, \$\$-a ; invalid operand type !!! ; aici avem \$\$ din code și a din data segment!!!!
mov cx, a-\$; OK !!!!!! Merge !! (**a definit altundeva și \$ generat aici !**)

mov ch, \$-a ; invalid operand type !!!!! (\$ generat aici, a altundeva)
mov ch, a-\$; OBJ format can handle only 16 or 32 byte relocation
a-\$ is OK, dar a-\$ = POINTER !! – syntax error ! (pt că offset NU începe în 1 byte !!)

mov cx, \$-start ; ok !!!
mov cx, start-\$; ok !!! (ambele scăderi ok - pt că etichetele sunt din același segment !!!)

mov ch, \$-start ; ok !!! – pt că REZ este scalar ! (dacă era pointer nu mergea !!)
mov ch, start-\$; ok !!!

mov cx, a-start ; ok !!! (**a definit altundeva și start definit aici !**)
mov cx, start-a ; invalid operand type !!! (start definit aici, a altundeva)

start1:

mov ah, a+b ; MERGE !!!!!!! DAR NU ESTE ADUNARE DE POINTERI !!!
E adunare de scalari !!! $a+b = (a-\$) + (b-\$)$

mov ax, b+a ; AX = (b-\$) + (a-\$) – adunare de SCALARI !!!

mov ax, [a+b] ; INVALID EFFECTIVE ADDRESS !!!! – ASTA CHIAR E ADUNARE DE POINTERI !!!!! - deci e INTERZISA !!! – deci e syntax error !!!

var1 dd a+b ; syntax error ! - expression is not simple or relocatable

(deci NASM nu permite ca "a+b" să apară într-o definiție de date ca expresie de inițializare, ci NUMAI ca OPERAND AL UNEI INSTRUCTIUNI – cum se observă mai sus !)

Concluzie:

Expresiile de tip et1 – et2 (unde et1 și et2 sunt etichete – fie de cod, fie de date) sunt acceptate sintactic de către NASM,

- Fie dacă ambele sunt definite în același segment
- Fie dacă et1 aparține unui segment diferit față de cel în care apare expresia, iar et2 este definită în segmentul în care apare expresia. Într-un astfel de caz, TD asociat expresiei et1-et2 este POINTER și NU SCALAR (constantă numerică) ca și în cazul etichetelor ce fac parte din același segment. (Deci ALTUNDEVA – AICI merge !!!!, însă AICI – ALTUNDEVA NU !!!!!)

Scădere de offset-uri ce se raportează la același segment = SCALAR
Scădere de pointeri din segmente diferite = POINTER

Numele unui segment este asociat cu "Adresa segmentului în memorie în faza de execuție" însă aceasta nu ne este disponibilă/accesibilă, fiind decisă la momentul încărcării programului pt execuție de către încărcătorul de programe al SO. De aceea, dacă folosim nume de segmente în expresii din program în mod explicit vom obține fie eroare de sintaxă (în situații de tipul \$/a-data - "AICI – ALTUNDEVA") fie de link-editare (în situații de tipul data-a - ALTUNDEVA – AICI), deoarece practic numele unui segment este asociat cu adresa FAR al acestuia (adresă care nu va fi cunoscută decât la momentul încărcării programului, deci va fi disponibilă doar la run-time; observăm astfel că adresa_segment este considerată ca "ALTUNDEVA") și NU este asociat cu "Offset-ul segmentului în faza de asamblare, fiind o constantă determinată la momentul asamblării" (așa cum este în programarea sub 16 biți de exemplu, unde nume_segment în faza de asamblare = offset-ul său = 0). Ca urmare, se constată că în progr. sub 32 de biți numele de segmente NU pot fi folosite în expresii !!

Mov eax, data ; Segment selector relocations are not supported in PE files (relocation error occurred) - syntax error!

Sub 32 biți offset (data) = vezi OllyDbg – DATA segment începe la offset-ul 00402000, iar CODE segment la offset 00401000 (sau eventual invers, depinde de link-editor, versiunea de SO etc). Deci offset-urile începuturilor de segment nu sunt 0 la fel ca și la programarea sub 16 biți. Din acest punct de vedere sub 32 biți este o mare diferență între numele de variabile (al căror offset poate fi determinat la asamblare) și numele de segmente (al căror offset NU poate fi determinat la momentul asamblării ca și o constantă, această valoare fiind cunoscută la fel ca și adresa de segment doar la momentul încărcării programului pt execuție - loading time).

Dacă avem mai mulți operanzi pointeri, asamblorul va încerca să încadreze expresia într-o combinație validă de operații cu pointeri:

Mov bx, [(v3-v2) ± (v1-v)] – OK !!! (adunarea și scăderea de valori imediate este corectă !!)

Dacă expresia nu va putea fi încadrată într-o combinație validă de operații cu pointeri, vom obține eroarea de sintaxă “Invalid effective address”:

Mov bx, [v2-v1-v] ; Invalid effective address !

Mov bx, v2-v1-v ; Invalid operand type !

Mov bx, v2-v1-v ; Invalid operand type ! mov bx, v2-(v1+v) – de unde concluzionăm că v1+v este acceptată ca și SCALAR în interpretarea $a+b = (a-\$) + (b-\$)$, doar dacă apare DE SINE STATATOR sau în combinație cu alți SCALARI , dar NU și nu în combinație cu expresii de tip POINTER !!!

Mov bx, v2+v1-v ; ok = v2+scalar

Mov bx, [v3-v2-v1-v] ; Invalid effective address !

Mov bx, v3-v2-v1-v ; OK !!! – deoarece ?... Mov bx, v3-v2-v1-v = Mov bx, (v3-v2)-(v1+v) = scădere de valori immediate (SCALARE!)

Varianta de adresare directă vs. indirectă este mai permisivă ca operații aritmetice în NASM (din cauza acceptării expresiilor de tip “a+b”) însă asta nu înseamnă că este mai permisivă IN OPERAȚIILE CU POINTERI !!!

Mov bx, (v3±v2) ± (v1±v) – OK !!! (adunarea și scăderea de valori imediate este corectă !!)

Analiza instrucțiunii JMP. Salturi NEAR și salturi FAR.

Exemplul 4.3.1.2. Prezentăm în continuare un exemplu edificator pentru modul de transfer al controlului la o etichetă, punând în evidență deosebirile dintre un transfer *direct* și unul *indirect*.

segment data

aici DD here ;echivalent cu aici := offsetul etichetei here din segmentul de cod

segment code

mov eax, [aici] ;se încarcă în EAX conținutul variabilei aici (adică deplasamentul lui here în cadrul segmentului code – echivalent deci ca efect cu: **mov eax, here**)

mov ebx, aici ;se încarcă în EBX deplasamentul lui aici în cadrul segmentului data ; (probabil 00401000h)

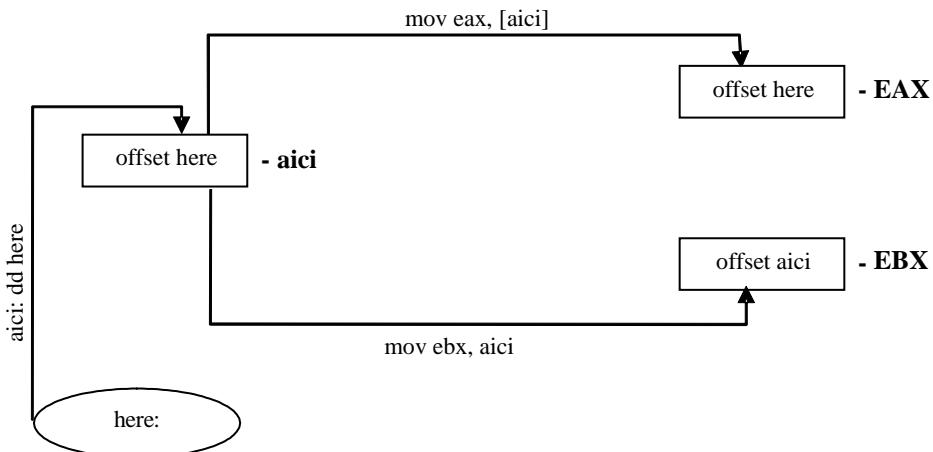


Fig. 4.4. Inițializarea variabilei aici și a regiștrilor EAX și EBX.

jmp [aici] ;salt la adresa desemnată de valoarea variabilei aici (care este adresa lui here), deci instrucțiune echivalentă cu jmp here ; ce face jmp aici ??? - același lucru ca și jmp ebx ! salt la CS:EIP cu EIP=offset (aici) din SEGMENT DATA (00401000h) ; salt la niste instr. care merg pana la primul access violation

jmp here ;salt la adresa lui here (sau, echivalent, salt la eticheta here); jmp [here] ?? – JMP DWORD PTR DS:[00402014] – cel mai probabil Access violation....

jmp eax ;salt la adresa conținută în EAX (adresare registru în mod direct), adică la here ; ce face prin comparatie jmp [eax] ??? JMP DWORD PTR DS:[EAX] – cel mai ; probabil Access violation....

jmp [ebx] ;salt la adresa conținută în locația de memorie a cărei adresă este conținută în ;EBX (adresare registru în mod indirect - singura situație de apel indirect din ;acest exemplu) – ce face prin comparatie jmp ebx ??? - salt la CS:EIP cu EIP=offset (aici) din SEGMENT DATA (00401000h) ; salt la niste instr. care merg pana la primul access violation

în EBX se află adresa variabilei **aici**, deci se accesează conținutul acestei variabile. În această locație de memorie se găsește offset-ul etichetei **here**, deci se va efectua saltul la adresa **here** - **ca urmare, ultimele 4 instrucțiuni de mai sus sunt toate echivalente cu jmp here**

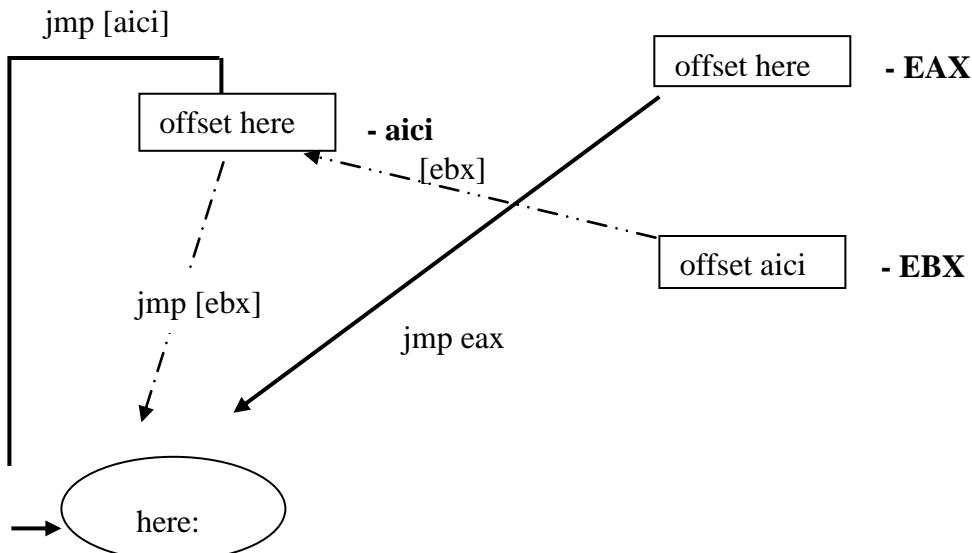


Fig. 4.5. Modalități alternative de efectuare a salturilor la eticheta *here*.

`jmp [ebp] ; JMP DWORD PTR SS:[EBP]`

.....
here:

`mov ecx, 0ffh`

Explicații asupra interacțiunii dintre regulile de asociere implicită a unui offset cu registrul segment corespunzător și efectuarea corespunzătoare a saltului la offset-ul precizat

JMP [var_mem] ; JMP DWORD PTR DS:[00401704] - salt NEAR la offset-ul din DS:[00401704]

JMP [EBX] ; JMP DWORD PTR DS:[EBX] - salt NEAR la offset-ul din DS:[EBX]
JMP [EBP] ; JMP DWORD PTR SS:[EBP] - salt NEAR la offset-ul din SS:[EBP]

JMP here ; JMP [SHORT] 00402024 - va face saltul DIRECT la offsetul respectiv in code segment

conform regulilor de asociere implicită a unui offset cu registrul segment corespunzător.

Operandul cu adresare INDIRECTĂ de după JMP ne indică DE UNDE să luăm OFFSET-ul la care să se efectueze saltul NEAR (salt în interiorul segmentului de cod curent). Ultima instrucțiune exprimă un salt DIRECT la offset-ul calculat ca valoare asociată etichetei here (salt la CS:here).

Chiar dacă folosim prefixarea explicită cu un registru segment a operandului destinație saltul NU va fi unul FAR. **FAR va fi doar ADRESA de la care se va prelua OFFSET-ul unde se va face saltul NEAR.**

jmp [ss: ebx + 12]

jmp [ss:ebp + 12] equiv with jmp [ebp + 12]

Saltul rămâne în continuare unul NEAR și se va efectua în cadrul acelaiași segment de cod, adică la **CS : valoare offset preluată din SS:[ebp+12]**

Dacă dorim însă efectuarea saltului într-un segment diferit (salt FAR) trebuie să specificăm explicit acest lucru prin intermediul operatorului de tip FAR, care va impune tratarea operandului destinație a instrucțiunii JMP drept O ADRESĂ FAR:

**jmp far [ebx + 12] => CS : EIP <- adresa far (48 biți = 6 octeți) echivalent cu
jmp far [DS: ebx + 12] => CS : EIP <- adresa far (48 biți = 6 octeți)**

(9b 7a 52 61 c2 65) → EIP = 61 52 7a 9b ; CS= 65 c2

„**Mov CS:EIP, [adr_far]**” ;

sau prin prefixare explicită: **jmp far [ss: ebx + 12] => CS : EIP <- adresa far (48 biți)**

Operatorul FAR precizează aici faptul că nu doar EIP trebuie populat cu ce se află la adresa de memorie indicată de operandul destinație (adresă NEAR = offset) ci și CS-ul trebuie încărcat cu o nouă valoare (CS:EIP = adresă FAR).

Pe scurt avem:

- valoarea pointer-ului poate fi stocată oriunde în memorie, rezultând că orice specificare de adresă validă la un mov poate apărea la fel de bine și la jmp (de exemplu **jmp [gs:ebx + esi*8 - 1023]**)
- pointer-ul în sine (deci octetii luați de la respectiva adresă de memorie) poate fi far sau near, fiind, după caz, aplicat fie lui doar lui EIP (dacă e NEAR), fie perechii CS:EIP daca saltul e FAR.

Saltul poate fi imaginat ca fiind echivalent, ipotetic, cu instrucțiuni MOV de forma:

- **jmp [gs:ebx + esi * 8 - 1023] <=> mov EIP, [gs:ebx + esi * 8 - 1023]**
- **jmp FAR [gs:ebx + esi * 8 - 1023] <=> mov EIP, DWORD [gs:ebx + esi * 8 - 1023] +
mov CS, WORD [gs:ebx + esi * 8 - 1023 + 4]**

La adresa **[gs:ebx + esi * 8 - 1023]** am gasit în exemplul concret utilizat configurația de memorie:

7B 8C A4 56 D4 47 98 B7.....

Mov CS:EIP, [memory] → **EIP** = 56 A4 8C 7B
CS = 47 D4

JMP prin etichete – always NEAR !

```
segment data use32 class=DATA
```

```
a db 1,2,3,4
```

```
start3:
```

mov edx, eax ; ok ! – controlul este transferat la start3 și această instrucțiune este executată !

.....

```
segment code use32 class=code
```

offset code segment = 00402000

```
start:
```

```
    mov edx, eax
```

```
    jmp start2 - ok – salt NEAR - JMP 00403000 (offset code1 segment = 00403000)
```

```
    jmp start3 – ok – salt NEAR – JMP 00401004 (offset data segment = 00401000)
```

jmp far start2 - Segment selector relocations are not supported in PE file – syntax error !

jmp far start3 - Segment selector relocations are not supported in PE file– syntax error !

;Cele două salturi de mai sus jmp start2 și respectiv jmp start3 se vor efectua la etichetele menționate, start2 și start3 însă ele nu vor fi considerate salturi FAR (dovada este că precizarea acestui atribut mai sus în celelalte două variante ale instrucțiunilor va furniza syntax error !)

;Ele vor fi considerate salturi NEAR datorita modelului de memorie FLAT utilizat de catre SO

```
add eax,1
```

```
final:
```

```
    push dword 0
```

```
    call [exit]
```

```
segment code1 use32 class=code
```

```
start2:
```

```
    mov eax, ebx
```

```
    push dword 0
```

```
    call [exit]
```

De ce ?... Din cauza **“Flat memory model”**

Concluzii finale.

- Salturi NEAR – se pot realiza prin oricare dintre cele 3 tipuri de operanzi (etichetă, registru, operand cu adresare la memorie)
- Salturi FAR (asta însemnând modificarea și a valorii din CS, nu numai a valorii din EIP) – se pot realiza DOAR prin intermediul unui operand adresare la memorie pe 48 de biți (pointer FAR = 6 octeți). De ce doar așa și prin etichete sau registri nu ?
- Prin etichete, chiar dacă se sare într-un alt segment nu se consideră ca este salt FAR deoarece nu se modifica CS-ul (din cauza modelului de memorie implementat – Flat Memory Model). Se va modifica doar EIP-ul și saltul se consideră dpdv tehnic ca fiind un salt NEAR.
- Prin registri nu este posibil deoarece registri sunt pe 32 de biți și se poate astfel specifica drept operand al unui JMP doar un offset (salt NEAR), deci practic suntem în imposibilitatea de a preciza un salt FAR cu un operand limitat la 32 de biți.

2.6.5. Reprezentarea instrucțiunilor mașină

O instrucțiune mașină x86 reprezintă o secvență de 1 până la 15 octeți, care prin valorile lor specifică o operație de executat, operanții asupra cărora va fi aplicată, precum și modificatori suplimentari care controlează modul în care aceasta va fi executată.

O instrucțiune mașină x86 are maximum doi operanzi. Pentru cele mai multe dintre instrucțiuni, cei doi operanzi poartă numele de *sursă*, respectiv *destinație*. Dintre cei doi operanzi, maximum unul se poate afla în memoria RAM. Celălalt se află fie într-un registru al EU, fie este o constantă întreagă. Astfel, o instrucțiune are forma:

numeinstrucțiune destinație, sursă

Formatul intern al unei instrucțiuni este variabil, el putând ocupa între 1 și 15 octeți, având următoarea formă generală de reprezentare (*Instructions byte-codes from OllyDbg*):

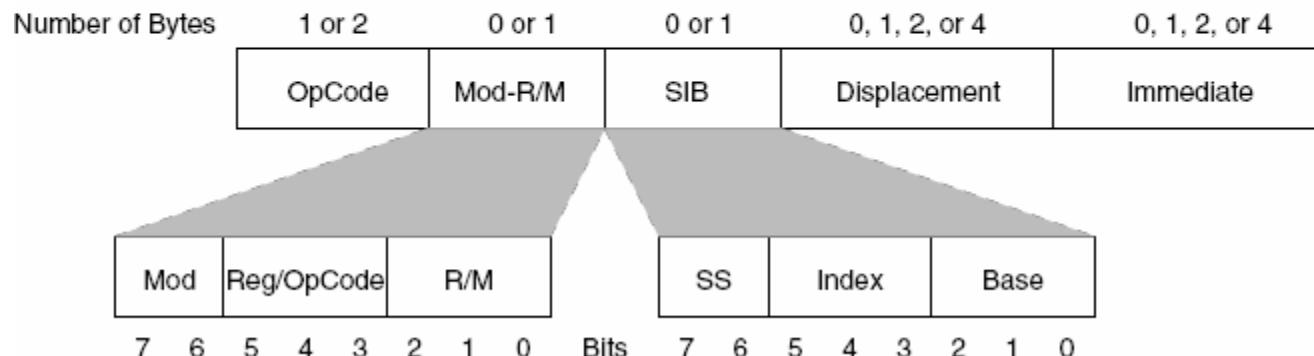
[prefixe] + cod + [ModR/M] + [SIB] + [deplasament] + [imediat]

Prefixele controlează modul în care o instrucțiune se execută. Acestea sunt opționale (0 până la maximum 4) și ocupă câte un octet fiecare. De exemplu, acestea pot solicita execuția repetată (în buclă) a instrucțiunii curente sau pot bloca magistrala de adrese pe parcursul execuției pentru a nu permite accesul concurrent la operanzi și rezultate.

Operația care se va efectua este identificată prin intermediul a 1 sau 2 octeți de *cod* (opcode), aceștia fiind singurii octeți obligatoriu prezenți, indiferent de instrucțiune.

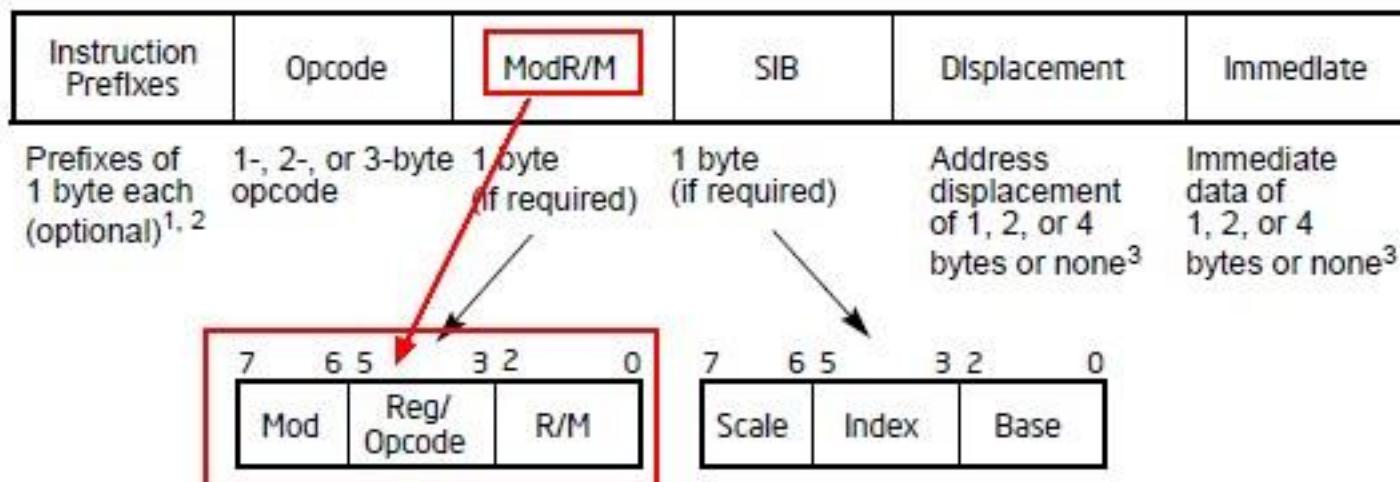
Number of Bytes	0 or 1	0 or 1	0 or 1	0 or 1
	Instruction prefix	Address-size prefix	Operand-size prefix	Segment override

(a) Optional instruction prefixes



(b) General instruction format

Although the diagram seems to imply that instructions can be up to 16 bytes long, in actuality the x86 will not allow instructions greater than 15 bytes in length.



Octetul *ModR/M* (mod registru/memorie) specifică pentru unele dintre instrucțiuni natura și locul operanzilor (registru, memorie). Acesta permite specificarea fie a unui registru, fie a unei locații de memorie a cărei adresă este exprimată prin intermediul unui offset (<http://datacadamia.com/intel/modrm>)

Pentru cazuri mai complexe de adresare decât cele codificabile direct prin ModR/M, combinarea acestuia cu octetul SIB (Scale – Index – Base) permite următoarea formulă generală de definire a unui offset:

$$\text{offset} = [\text{bază}] + [\text{index} \times \text{scală}] + [\text{constantă}]$$

(SIB) (deplasament + imediat)

unde pentru bază și index vor fi folosite valorile a doi regiștri iar scală este 1, 2, 4 sau 8. Regiștrii permisi ca bază sau / și index sunt: EAX, EBX, ECX, EDX, EBP, ESI, EDI. Registrul ESP este disponibil ca bază însă nu poate fi folosit cu rol de index. (http://www.c-jump.com/CIS77/CPU/x86/lecture.html#X77_0100_sib_byte_layout)

Majoritatea instrucțiunilor folosesc pentru reprezentare fie numai campul de cod, fie cod urmat de ModR/M.

Deplasament (displacement) apare în cazul unor forme de adresare particulare (operanzi din memorie) și urmează direct după ModR/M sau SIB, când SIB este prezent. Acest câmp poate fi codificat fie pe octet, fie pe cuvânt, fie pe dublu cuvânt (32 biți).

The most common addressing mode, and the one that's easiest to understand, is the *displacement-only* (or **direct**) addressing mode. The displacement-only addressing mode consists of a 32-bit constant that specifies the address of the target location. The displacement-only addressing mode **is perfect for accessing simple scalar variables**. Intel named this the displacement-only addressing mode because a 32-bit constant (displacement) follows the MOV opcode in memory. On the 80x86 processors, this displacement is an offset from the beginning of memory (that is, address zero).

Displacement mode, the operand's offset is contained as part of the instruction as an 8-, 16-, or 32-bit displacement. The displacement addressing mode is found on few machines because, as mentioned earlier, it leads to long instructions. In the case of the x86, the displacement value can be as long as 32 bits, making for a 6-byte instruction. Displacement addressing can be useful for referencing global variables.

Ca și consecință a imposibilității prezenței mai multor câmpuri de ModR/M, SIB și deplasament într-o instrucțiune, arhitectura 80x86 NU permite codificarea a două adrese de memorie în aceeași instrucțiune.

Valoare imediată oferă posibilitatea definirii unui operand ca fiind o constantă numerică pe 1, 2 sau 4 octeți. Când este prezent, acest câmp apare întotdeauna la sfârșitul instrucțiunii.

24. x86 Instruction Prefix Bytes

- x86 [instruction](#) can have up to 4 prefixes.
- Each prefix adjusts interpretation of the opcode:

String manipulation instruction prefixes (prefixe precizate EXPLICIT de catre programator !)

F3h = REP, REPE

F2h = REPNE

where

- **REP** repeats instruction the number of times specified by *iteration count* **ECX**.
 - **REPE** and **REPNE** prefixes allow to terminate loop on the value of **ZF** CPU flag.
-
- 0xF3 is called REP when used with **MOVS/LODS/STOS/INS/OUTS** (instructions which don't affect flags)
0xF3 is called REPE or REPZ when used with **CMPS/SCAS**
0xF2 is called REPNE or REPNZ when used with **CMPS/SCAS**, and is not documented for other instructions.
Intel's insn reference manual REP entry only documents F3 REP for MOVS, not the F2 prefix.

Instruction prefix - **REP MOVS** **F3:A4**

Related string manipulation instructions are:

- **MOVS**, move string ; **STOS**, store string
- **SCAS**, scan string ; **CMPS**, compare string, etc.

See also string manipulation sample program: [rep_movsb.asm](#)

Segment override prefix causes memory access to use *specified segment* instead of *default segment* designated for instruction operand.
(acestea sunt prefixe precizate EXPLICIT de catre programator).

2Eh = CS

36h = SS

3Eh = DS

26h = ES

64h = FS

65h = GS

26 : D7

Segment override prefix - **ES** xlat
(ES:EBX)

mov eax, [ebx] - 8B03

mov eax, [SS:ebx] – 36:8B03

mov ax, [DS:ebx] – prefixare implicită cu 66 3E : 66 : 8B03

- prefixare explicită de catre progr. cu DS

(3E – prefix DS, 66 – prefix op.size, 8B – cod mov Gv, Ev, 03 – EBX)

mov eax, [CS:ebx] – prefixare explicită de către progr. cu CS 2E : 8B03
(2E – prefix CS, 8B – cod mov Gv, Ev, 03 – EBX)

ES LODSB – 26 : AC ; LODS byte ptr ES:[ESI]

ES CMPSB - 26 : A6 ; CMPS byte ptr ES:[ESI], byte ptr ES:[EDI]

**ES STOSB - 26 : AA ; STOS byte ptr ES:[EDI] – superfluous segment
override prefix**

MOVSB - A4 ; MOVS byte ptr ES:[EDI], byte ptr DS:[ESI]

ES MOVSB - 26 : A4 ; MOVS byte ptr ES:[EDI], byte ptr ES:[ESI]

**ES SCASB - 26 : AE ; SCAS byte ptr ES:[EDI] - superfluous segment
override prefix**

Operand override, 66h. Changes size of **data** expected by default mode of the instruction e.g. 16-bit to 32-bit and vice versa.

Address override, 67h. Changes size of **address** expected by the default mode of the instruction. 32-bit address could switch to 16-bit and vice versa.

Aceste ultime două tipuri de prefixe apar ca rezultat al unor moduri particulare de utilizare a instrucțiunilor (exemple mai jos), utilizări care vor provoca apariția acestor prefixe la nivelul formatului intern al instrucțiunii. Aceste prefixe NU se precizează EXPLICIT prin cuvinte cheie sau rezervate ale limbajului de asamblare.

Operand size prefix – 0x66

Bits 32 – default mode of the below code

cbw ; 66:98 - deoarece rez este pe 16 biți (AX)

cwd ; 66:99 - deoarece rez este format din 2 reg. pe 16 biți (DX:AX)

cwde ; 98 - aici se respectă modul default pe 32 biți – rez în EAX

push ax ; 66:50 – deoarece se încarcă pe stivă o val pe 16 biți, stiva fiind organizată implicit (default) pe 32 biți

push eax ; 50 - ok – utilizare consistentă cu modul default

mov ax, a ; 66:B8 0010 – deoarece rez. este pe 16 biți

Bits 16 – default mode of the below code

cbw ; 98 - deoarece rez este pe 16 biți (AX)

cwd ; 99 - deoarece rez este format din 2 reg pe 16 biți (DX:AX)

cwde ; 66:98 - deoarece aici NU se respectă modul default pe 16 biți – rez in EAX

Address size prefix – 0x67

Bits 32

mov eax, [bx] ; 67:8B07 - deoarece DS:[BX] este adresare pe 16 biți

Bits 16

mov BX, [EAX] ; 67:8B18 – deoarece DS:[EAX] este adresare pe 32 biți

Bits 16

push dword[ebx] ; 66:67:FF33 – Aici modul default este Bits 16;
deoarece adresarea [EBX] este pe 32 biți apare 67 și deoarece se face
push la un operand DWORD în loc de unul pe 16 biți apare 66 ca prefix

push dword[CS:ebx] ; 2E:66:67:FF33
rep push dword[CS:ebx] ; F3:2E:66:67:FF33
(rep push word ptr CS:[BP+DI] - superfluous REPxx fix ! - OllyDbg)

Bits 32

67:8B07 mov eax, [bx]; Offset_16_bitii = [BX|BP] + [SI|DI] + [const]
(mov eax, dword ptr DS:[BX]) **67** – address size override prefix

Bits 16

66:8B07 mov eax, [bx]; mov ax, word ptr DS:[edi]
(66 – operand size override prefix)

Definiție.

Prefixele de instrucțiuni sunt construcții ale limbajului de asamblare care apar optional în componența unei linii sursă (prefixe explicite) sau a formatului internal unei instrucțiuni (prefixe generate în mod implicit de către asamblor în două situații) și care modifică comportamentul standard al acestor instrucțiuni (în cazul prefixelor explicite) sau care semnalează procesorului modificarea dimensiunii implicit de reprezentare a operanzilor sau/și a adreselor, dimensiuni stabilite prin directive de asamblare (BITS 16 su BITS 32).

2.6.5. Reprezentarea instrucțiunilor mașină

O instrucțiune mașină x86 reprezintă o secvență de 1 până la 15 octeți, care prin valorile lor specifică o operație de executat, operanții asupra cărora va fi aplicată, precum și modificatori suplimentari care controlează modul în care aceasta va fi executată.

O instrucțiune mașină x86 are maximum doi operanzi. Pentru cele mai multe dintre instrucțiuni, cei doi operanzi poartă numele de *sursă*, respectiv *destinație*. Dintre cei doi operanzi, maximum unul se poate afla în memoria RAM. Celălalt se află fie într-un registru al EU, fie este o constantă întreagă. Astfel, o instrucțiune are forma:

numeinstrucțiune destinație, sursă

Formatul intern al unei instrucțiuni este variabil, el putând ocupa între 1 și 15 octeți, având următoarea formă generală de reprezentare (*Instructions byte-codes from OllyDbg*):

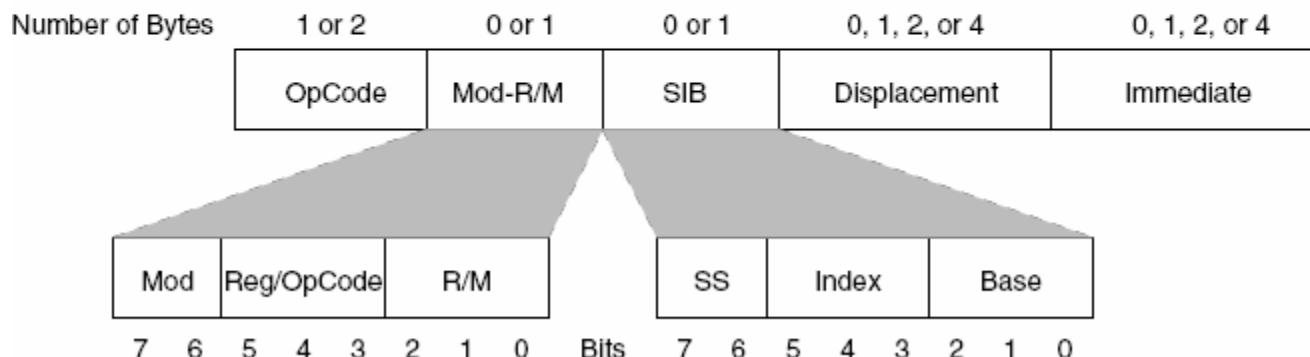
[prefixe] + cod + [ModR/M] + [SIB] + [deplasament] + [imediat]

Prefixele controlează modul în care o instrucțiune se execută. Acestea sunt opționale (0 până la maximum 4) și ocupă câte un octet fiecare. De exemplu, acestea pot solicita execuția repetată (în buclă) a instrucțiunii curente sau pot bloca magistrala de adrese pe parcursul execuției pentru a nu permite accesul concurrent la operanzi și rezultate.

Operația care se va efectua este identificată prin intermediul a 1 sau 2 octeți de *cod* (opcode), aceștia fiind singurii octeți obligatoriu prezenți, indiferent de instrucțiune.

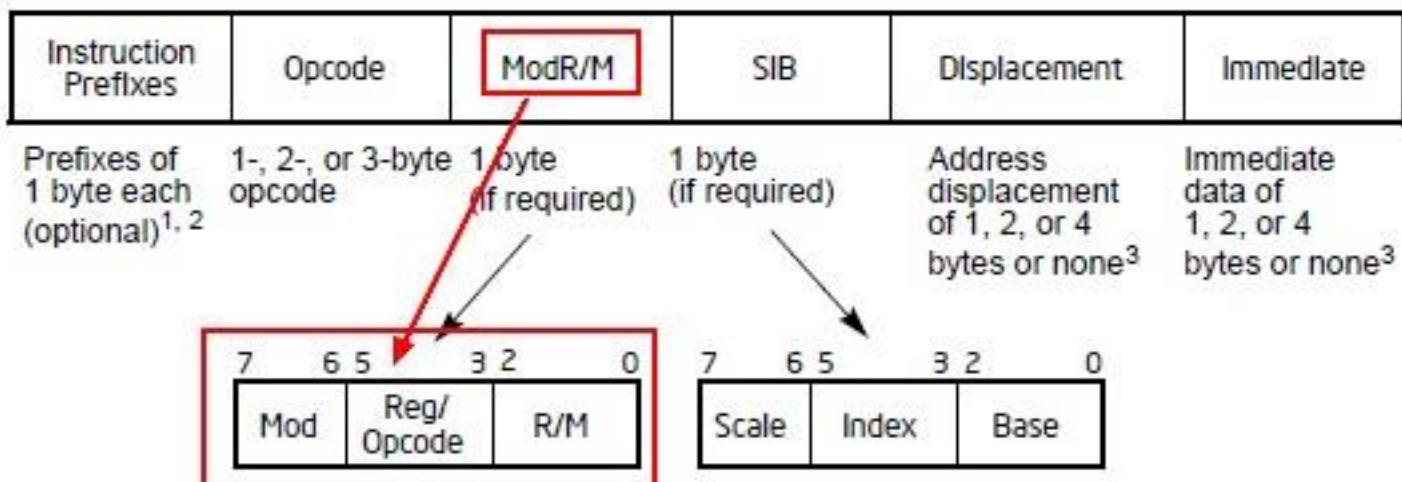
Number of Bytes	0 or 1	0 or 1	0 or 1	0 or 1
	Instruction prefix	Address-size prefix	Operand-size prefix	Segment override

(a) Optional instruction prefixes



(b) General instruction format

Although the diagram seems to imply that instructions can be up to 16 bytes long, in actuality the x86 will not allow instructions greater than 15 bytes in length.



Octetul *ModR/M* (mod registru/memorie) specifică pentru unele dintre instrucțiuni natura și locul operanzilor (registru, memorie). Acesta permite specificarea fie a unui registru, fie a unei locații de memorie a cărei adresă este exprimată prin intermediul unui offset (<http://datacadamia.com/intel/modrm>)

Pentru cazuri mai complexe de adresare decât cele codificabile direct prin ModR/M, combinarea acestuia cu octetul SIB (Scale – Index – Base) permite următoarea formulă generală de definire a unui offset:

unde pentru bază și index vor fi folosite valorile a doi registri iar scală este 1, 2, 4 sau 8. Registrii permisi ca bază sau / și index sunt: EAX, EBX, ECX, EDX, EBP, ESI, EDI. Registrul ESP este disponibil ca bază însă nu poate fi folosit cu rol de index. (http://www.c-jump.com/CIS77/CPU/x86/lecture.html#X77_0100_sib_byte_layout)

Majoritatea instrucțiunilor folosesc pentru reprezentare fie numai campul de cod, fie cod urmat de ModR/M.

Deplasament (displacement) apare în cazul unor forme de adresare particulare (operanzi din memorie) și urmează direct după ModR/M sau SIB, când SIB este prezent. Acest câmp poate fi codificat fie pe octet, fie pe cuvânt, fie pe dublu cuvant (32 biți).

The most common addressing mode, and the one that's easiest to understand, is the *displacement-only* (or *direct*) addressing mode. **The displacement-only addressing mode consists of a 32-bit constant that specifies the address of the target location.** The displacement-only addressing mode **is perfect for accessing simple scalar variables.** Intel named this the displacement-only addressing mode because a 32-bit constant (displacement) follows the MOV opcode in memory. **On the 80x86 processors, this displacement is an offset from the beginning of memory (that is, address zero).**

Displacement mode, the operand's offset is contained as part of the instruction as an 8-, 16-, or 32-bit displacement. The displacement addressing mode is found on few machines because, as mentioned earlier, it leads to long instructions. In the case of the x86, the displacement value can be as long as 32 bits, making for a 6-byte instruction. **Displacement addressing can be useful for referencing global variables.**

Ca și consecință a imposibilității prezenței mai multor câmpuri de ModR/M, SIB și deplasament într-o instrucțiune, arhitectura 80x86 NU permite codificarea a două adrese de memorie în aceeași instrucțiune.

Valoare imediată oferă posibilitatea definirii unui operand ca fiind o constantă numerică pe 1, 2 sau 4 octeți. Când este prezent, acest câmp apare întotdeauna la sfârșitul instrucțiunii.

Intel x86 Assembler Instruction Set Opcode Table

ADD Eb Gb 00	ADD Ev Gv 01	ADD Gb Eb 02	ADD Gv Ev 03	ADD AL Ib 04	ADD eAX Iv 05	PUSH ES 06	POP ES 07	OR Eb Gb 08	OR Ev Gv 09	OR Gb Eb 0A	OR Gv Ev 0B	OR AL Ib 0C	OR eAX Iv 0D	PUSH CS 0E	TWOBYTE 0F
ADC Eb Gb 10	ADC Ev Gv 11	ADC Gb Eb 12	ADC Gv Ev 13	ADC AL Ib 14	ADC eAX Iv 15	PUSH SS 16	POP SS 17	SBB Eb Gb 18	SBB Ev Gv 19	SBB Gb Eb 1A	SBB Gv Ev 1B	SBB AL Ib 1C	SBB eAX Iv 1D	PUSH DS 1E	POP DS 1F
AND Eb Gb 20	AND Ev Gv 21	AND Gb Eb 22	AND Gv Ev 23	AND AL Ib 24	AND eAX Iv 25	ES:	DAA	SUB Eb Gb 28	SUB Ev Gv 29	SUB Gb Eb 2A	SUB Gv Ev 2B	SUB AL Ib 2C	SUB eAX Iv 2D	CS:	DAS
XOR Eb Gb 30	XOR Ev Gv 31	XOR Gb Eb 32	XOR Gv Ev 33	XOR AL Ib 34	XOR eAX Iv 35	SS:	AAA	CMP Eb Gb 38	CMP Ev Gv 39	CMP Gb Eb 3A	CMP Gv Ev 3B	CMP AL Ib 3C	CMP eAX Iv 3D	DS:	AAS
INC eAX 40	INC eCX 41	INC eDX 42	INC eBX 43	INC eSP 44	INC eBP 45	INC eSI 46	INC eDI 47	DEC eAX 48	DEC eCX 49	DEC eDX 4A	DEC eBX 4B	DEC eSP 4C	DEC eBP 4D	DEC eSI 4E	DEC eDI 4F
PUSH eAX 50	PUSH eCX 51	PUSH eDX 52	PUSH eBX 53	PUSH eSP 54	PUSH eBP 55	PUSH eSI 56	PUSH eDI 57	POP eAX 58	POP eCX 59	POP eDX 5A	POP eBX 5B	POP eSP 5C	POP eBP 5D	POP eSI 5E	POP eDI 5F
PUSHA 60	POPA 61	BOUND Gv Ma 62	ARPL Ew Gw 63	FS:	GS:	OPSIZE: 66	ADSIZE: 67	PUSH Iv 68	IMUL Gv Ev Iv 69	PUSH Ib 6A	IMUL Gv Ev Ib 6B	INSB Yb DX 6C	INSW Yz DX 6D	OUTSB DX Xb 6E	OUTSW DX Xv 6F
JO Jb 70	JNO Jb 71	JB Jb 72	JNB Jb 73	JZ Jb 74	JNZ Jb 75	JBE Jb 76	JA Jb 77	JS Jb 78	JNS Jb 79	JP Jb 7A	JNP Jb 7B	JL Jb 7C	JNL Jb 7D	JLE Jb 7E	JNLE Jb 7F
ADD Eb Ib	ADD Ev Iv	SUB Eb Ib	SUB Ev Ib	TEST Eb Gb	TEST Ev Gv	XCHG Eb Gb	XCHG Ev Gv	MOV Eb Gb	MOV Ev Gv	MOV Gb Eb	MOV Gv Ev	MOV Ew Sw	LEA Gv M	MOV Sw Ew	POP Ev

80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F
NOP 90	XCHG eAX eCX 91	XCHG eAX eDX 92	XCHG G eAX eBX 93	XCHG eAX eSP 94	XCHG eAX eBP 95	XCHG eAX eSI 96	XCHG eAX eDI 97	CBW	CWD	CALL Ap 9A	WAIT 9B	PUSH F Fv 9C	POPF Fv 9D	SAHF 9E	LAHF 9F
MOV AL Ob A0	MOV eAX Ov A1	MOV Ob AL A2	MOV Ov eAX A3	MOVS B Xb Yb A4	MOVS W Xv Yv A5	CMPSB Xb Yb A6	CMPS W Xv Yv A7	TEST AL Ib A8	TEST eAX Iv A9	STOS B Yb AL AA	STOS W Yv eAX AB	LODS B AL Xb AC	LODS W eAX Xv AD	SCASB AL Yb AE	SCASW eAX Yv AF
MOV AL Ib B0	MOV CL Ib B1	MOV DL Ib B2	MOV BL Ib B3	MOV AH Ib B4	MOV CH Ib B5	MOV DH Ib B6	MOV BH Ib B7	MOV eAX Iv B8	MOV eCX Iv B9	MOV eDX Iv BA	MOV eBX Iv BB	MOV eSP Iv BC	MOV eBP Iv BD	MOV eSI Iv BE	MOV eDI Iv BF
#2 Eb Ib C0	#2 Ev Ib C1	RETN Iw C2	RET N C3	LES Gv Mp C4	LDS Gv Mp C5	MOV Eb Ib C6	MOV Ev Iv C7	ENTE R Iw Ib C8	LEAV E C9	RETF Iw CA	RETF Iw CA	INT3 CB	INT Ib CD	INTO CE	IRET CF
#2 Eb 1 D0	#2 Ev 1 D1	#2 Eb CL D2	#2 Ev CL D3	AAM lb D4	AAD lb D5	SALC D6	XLAT D7	ESC 0 D8	ESC 1 D9	ESC 2 DA	ESC 3 DB	ESC 4 DC	ESC 5 DD	ESC 6 DE	ESC 7 DF
LOOPNZ Jb E0	LOOPZ Jb E1	LOOP Jb E2	JCXZ Jb E3	IN AL Ib E4	IN eAX Ib E5	OUT lb eAX E6	OUT lb eAX E7	CALL Jz E8	JMP Jz E9	JMP Ap EA	JMP Jb EB	IN AL DX EC	IN eAX DX ED	OUT DX AL EE	OUT DX eAX EF
LOCK: F0	INT1 F1	REPN E: F2	REP: F3	HLT F4	CMC F5	#3 Eb F6	#3 Ev F7	CLC F8	STC F9	CLI FA	STI FB	CLD FC	STD FD	#4 INC/DE C FE	#5 INC/DEC FF

Legend

HAS MOD R/M
LENGTH = 1
OTHER

80x86 Instruction Format

Prefix

INSTRUCTION PREFIX	ADDRESS SIZE PREFIX	OPERAND SIZE PREFIX	SEGMENT OVERRIDE
0 OR 1	0 OR 1	0 OR 1	0 OR 1
NUMBER OF BYTES			

Required

OPCODE	MOD R/M	SIB	DISPLACEMENT	IMMEDIATE
1 OR 2	0 OR 1	0 OR 1	0,1,2 OR 4	0,1,2 OR 4
NUMBER OF BYTES				

MOD R/M BYTE

7	6	5	4	3	2	1	0
MOD	REG/OPCODE			R/M			

SIB BYTE

7	6	5	4	3	2	1	0
SCALE	INDEX			BASE			

MOD R/M 16

	0	1	2	3	4	5	6	7
0	[BX+SI] +1	[BX+DI] +1	[BP+SI] +1	[BP+DI] +1	[SI] +1	[DI] +1	[Iw] +3	[BX] +1
1	[BX+SI+Ib] +2	[BX+DI+Ib] +2	[BP+SI+Ib] +2	[BP+DI+Ib] +2	[SI+Ib] +2	[DI+Ib] +2	[BP+Ib] +2	[BX+Ib] +2
2	[BX+SI+Iw] +3	[BX+DI+Iw] +3	[BP+SI+Iw] +3	[BP+DI+Iw] +3	[SI+Iw] +3	[DI+Iw] +3	[BP+Iw] +3	[BX+Iw] +3
3	AX +1	CX +1	DX +1	BX +1	SP +1	BP +1	SI +1	DI +1

MOD R/M 32

	0	1	2	3	4	5	6	7
0	[eAX] +1	[eCX] +1	[eDX] +1	[eBX] +1	[SIB] +2	[Iv] +5	[eSI] +1	[eDI] +1
1	[eAX+Ib] +2	[eCX+Ib] +2	[eDX+Ib] +2	[eBX+Ib] +2	[SIB+Ib] +2	[eBP+Ib] +2	[eSI+Ib] +2	[eDI+Ib] +2
2	[eAX+Iv] +5	[eCX+Iv] +5	[eDX+Iv] +5	[eBX+Iv] +5	[SIB+Iv] +5	[eBP+Iv] +5	[eSI+Iv] +5	[eDI+Iv] +5
3	eAX +1	eCX +1	eDX +1	eBX +1	eSP +1	eBP +1	eSI +1	eDI +1

REGISTERS

	0	1	2	3	4	5	6	7
Reg 8	AL	CL	DL	BL	AH	CH	DH	BH
Reg 16	AX	CX	DX	BX	SP	BP	SI	DI
Reg 32	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
Segments	DS	ES	FS	GS	SS	CS	IP	

A.4 OPCODE EXTENSIONS FOR ONE-BYTE AND TWO-BYTE OPCODES

Some 1-byte and 2-byte opcodes use bits 3-5 of the ModR/M byte (the nnn field in Figure A-1) as an extension of the opcode.

mod	nnn	R/M
-----	-----	-----

Figure A-1. ModR/M Byte nnn Field (Bits 5, 4, and 3)

Opcodes that have opcode extensions are indicated in Table A-6 and organized by group number. Group numbers (from 1 to 16, second column) provide a table entry point. The encoding for the r/m field for each instruction can be established using the third column of the table.

A.4.1 Opcode Look-up Examples Using Opcode Extensions

An Example is provided below.

Example A-4. Interpreting an ADD Instruction

An ADD instruction with a 1-byte opcode of 80H is a Group 1 instruction:

- Table A-6 indicates that the opcode extension field encoded in the ModR/M byte for this instruction is 000B.
- The r/m field can be encoded to access a register (11B) or a memory address using a specified addressing mode (for example: mem = 00B, 01B, 10B).

Example A-5. Looking Up 0F01C3H

Look up opcode 0F01C3 for a VMRESUME instruction by using Table A-2, Table A-3, and Table A-6:

- 0F indicates that this instruction is in the 2-byte opcode map.
- 01 (row 0, column 1 in Table A-3) reveals that this opcode is in Group 7 of Table A-6.
- C3 is the ModR/M byte. The first two bits of C3 are 11B. This tells us to look at the second of the Group 7 rows in Table A-6.
- The Op/Reg bits [5,4,3] are 000B. This tells us to look in the 000 column for Group 7.
- Finally, the R/M bits [2,1,0] are 011B. This identifies the opcode as the VMRESUME instruction.

A.4.2 Opcode Extension Tables

See Table A-6 below.

Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number *

Opcode	Group	Mod 7,6	pxf	Encoding of Bits 5,4,3 of the ModR/M Byte (bits 2,1,0 in parenthesis)							
				000	001	010	011	100	101	110	111
80-83	1	mem, 11B		ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
8F	1A	mem, 11B		POP							
C0,C1 reg, imm D0, D1 reg, 1 D2, D3 reg, CL	2	mem, 11B		ROL	ROR	RCL	RCR	SHL/SAL	SHR		SAR
F6, F7	3	mem, 11B		TEST lb/lz		NOT	NEG	MUL AL/rAX	IMUL AL/rAX	DIV AL/rAX	IDIV AL/rAX
FE	4	mem, 11B		INC Eb	DEC Eb						
FF	5	mem, 11B		INC Ev	DEC Ev	near CALL ^{f64} Ev	far CALL Ep	near JMP ^{f64} Ev	far JMP Mp	PUSH ^{d64} Ev	
0F 00	6	mem, 11B		SLDT Rv/Mw	STR Rv/Mw	LLDT Ew	LTR Ew	VERR Ew	VERW Ew		
0F 01	7	mem		SGDT Ms	SIDT Ms	LGDT Ms	LIDT Ms	SMSW Mw/Rv		LMSW Ew	INVLPG Mb
		11B		VMCALL (001) VMLAUNCH (010) VMRESUME (011) VMXOFF (100)	MONITOR (000) MWAIT (001) CLAC (010) STAC (011) ENCLS (111)	XGETBV (000) XSETBV (001) VMFUNC (100) XEND (101) XTEST (110) ENCLU(111)					SWAPGS ⁶⁴ (000) RDTSCP (001)
0F BA	8	mem, 11B							BT	BTS	BTR
0F C7	9	mem			CMPXCH8B Mq CMPXCHG16B Mdq					VMPTRLD Mq	VMPTRST Mq
			66							VMCLEAR Mq	
			F3							VMXON Mq	
		11B								RDRAND Rv	RDSEED Rv
			F3								RDPID Rd/q
0F B9	10	mem					UD1				
		11B									
C6	11	mem		MOV Eb, Ib							
		11B									XABORT (000) Ib
C7	11	mem		MOV Ev, Iz							
		11B									XBEGIN (000) Jz
0F 71	12	mem									
		11B				psrlw Nq, Ib		psraw Nq, Ib		psllw Nq, Ib	
			66			vpsrlw Hx,Ux,Ib		vpsraw Hx,Ux,Ib		vpsllw Hx,Ux,Ib	
0F 72	13	mem				psrid Nq, Ib		psrad Nq, Ib		pslld Nq, Ib	
		11B				vpsrid Hx,Ux,Ib		vpsrad Hx,Ux,Ib		vpslld Hx,Ux,Ib	
			66								
0F 73	14	mem				psrlq Nq, Ib				psllq Nq, Ib	
		11B				vpsrlq Hx,Ux,Ib	vpsrdq Hx,Ux,Ib			vpsllq Hx,Ux,Ib	vpsldq Hx,Ux,Ib
			66								

Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number * (Contd.)

Opcode	Group	Mod 7,6	pxf	Encoding of Bits 5,4,3 of the ModR/M Byte (bits 2,1,0 in parenthesis)							
				000	001	010	011	100	101	110	111
0F AE	15	mem		fxsave	fxrstor	ldmxcsr	stmxcsr	XSAVE	XRSTOR	XSAVEOPT	cflush
		11B							lfence	mfence	sfence
			F3	RDFSBASE Ry	RDGSBASE Ry	WRFSBASE Ry	WRGSBASE Ry				
0F 18	16	mem		prefetch NTA	prefetch T0	prefetch T1	prefetch T2	Reserved NOP			
		11B						Reserved NOP			
VEX.0F38 F3	17	mem			BLSR ^V By, Ey	BLSMSK ^V By, Ey	BLSI ^V By, Ey				
		11B									

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Addressing Method Codes

- A Direct address. The instruction has no ModR/M byte; the address of the operand is encoded in the instruction; and no base register, index register, or scaling factor can be applied (for example, far JMP (EA)).
- C The reg field of the ModR/M byte selects a control register (for example, MOV (0F20, 0F22)).
- D The reg field of the ModR/M byte selects a debug register (for example, MOV (0F21,0F23)).
- E A ModR/M byte follows the opcode and specifies the operand. The operand is either a general-purpose register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.
- F EFLAGS Register.
- G The reg field of the ModR/M byte selects a general register (for example, AX (000)).
- I Immediate data. The operand value is encoded in subsequent bytes of the instruction.
- J The instruction contains a relative offset to be added to the instruction pointer register (for example, JMP (0E9), LOOP).
- M The ModR/M byte may refer only to memory (for example, BOUND, LES, LDS, LSS, LFS, LGS, CMPXCHG8B).
- O The instruction has no ModR/M byte; the offset of the operand is coded as a word or double word (depending on address size attribute) in the instruction. No base register, index register, or scaling factor can be applied (for example, MOV (A0–A3)).
- P The reg field of the ModR/M byte selects a packed quadword MMX™ technology register.

Q A ModR/M byte follows the opcode and specifies the operand. The operand is either an MMX™ technology register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement.

R The mod field of the ModR/M byte may refer only to a general register (for example, MOV (0F20-0F24, 0F26)).

S The reg field of the ModR/M byte selects a segment register (for example, MOV (8C,8E)).

T The reg field of the ModR/M byte selects a test register (for example, MOV (0F24,0F26)).

V The reg field of the ModR/M byte selects a packed SIMD floating-point register.

W An ModR/M byte follows the opcode and specifies the operand. The operand is either a SIMD floating-point register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement

X Memory addressed by the DS:SI register pair (for example, MOVS, CMPS, OUTS, or LODS).

Y Memory addressed by the ES:DI register pair (for example, MOVS, CMPS, INS, STOS, or SCAS).

Operand Type Codes

a Two one-word operands in memory or two double-word operands in memory, depending on operand-size attribute (used only by the BOUND instruction).

b Byte, regardless of operand-size attribute.

C Byte or word, depending on operand-size attribute.

d Doubleword, regardless of operand-size attribute

dq Double-quadword, regardless of operand-size attribute.

p 32-bit or 48-bit pointer, depending on operand-size attribute.

pi Quadword MMX™ technology register (e.g. mm0)

ps 128-bit packed FP single-precision data.

q Quadword, regardless of operand-size attribute.

S 6-byte pseudo-descriptor.

SS Scalar element of a 128-bit packed FP single-precision data.

Si Doubleword integer register (e.g., eax)

V Word or doubleword, depending on operand-size attribute.

W Word, regardless of operand-size attribute.



CHAPTER 2 INSTRUCTION FORMAT

This chapter describes the instruction format for all IA-32 processors.

2.1. GENERAL INSTRUCTION FORMAT

All IA-32 instruction encodings are subsets of the general instruction format shown in Figure 2-1. Instructions consist of optional instruction prefixes (in any order), one or two primary opcode bytes, an addressing-form specifier (if required) consisting of the ModR/M byte and sometimes the SIB (Scale-Index-Base) byte, a displacement (if required), and an immediate data field (if required).

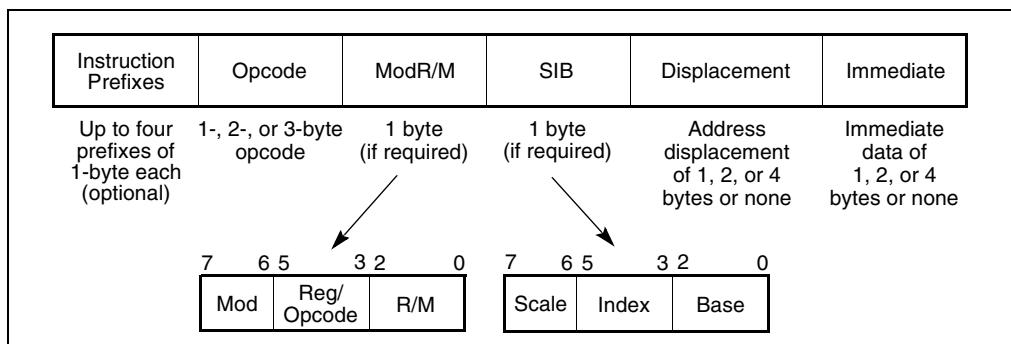


Figure 2-1. IA-32 Instruction Format

2.2. INSTRUCTION PREFIXES

The instruction prefixes are divided into four groups, each with a set of allowable prefix codes:

- Group 1
 - Lock and repeat prefixes:
 - F0H—LOCK.
 - F2H—REPNE/REPNZ (used only with string instructions).
 - F3H—REP or REPE/REPZ (use only with string instructions).
- Group 2
 - Segment override prefixes:
 - 2EH—CS segment override (use with any branch instruction is reserved).



INSTRUCTION FORMAT

- 36H—SS segment override prefix (use with any branch instruction is reserved).
- 3EH—DS segment override prefix (use with any branch instruction is reserved).
- 26H—ES segment override prefix (use with any branch instruction is reserved).
- 64H—FS segment override prefix (use with any branch instruction is reserved).
- 65H—GS segment override prefix (use with any branch instruction is reserved).
- Branch hints:
 - 2EH—Branch not taken (used only with Jcc instructions).
 - 3EH—Branch taken (used only with Jcc instructions).
- Group 3
 - 66H—Operand-size override prefix.
- Group 4
 - 67H—Address-size override prefix.

For each instruction, one prefix may be used from each of these groups and be placed in any order. Using redundant prefixes (more than one prefix from a group) is reserved and may cause unpredictable behavior.

The LOCK prefix forces an atomic operation to insure exclusive use of shared memory in a multiprocessor environment. See “LOCK—Assert LOCK# Signal Prefix” in Chapter 3, *Instruction Set Reference*, for a detailed description of this prefix and the instructions with which it can be used.

The repeat prefixes cause an instruction to be repeated for each element of a string. They can be used only with the string instructions: MOVS, CMPS, SCAS, LODS, STOS, INS, and OUTS. Use of the repeat prefixes with other IA-32 instructions is reserved and may cause unpredictable behavior (see the note below).

The branch hint prefixes allow a program to give a hint to the processor about the most likely code path that will be taken at a branch. These prefixes can only be used with the conditional branch instructions (Jcc). Use of these prefixes with other IA-32 instructions is reserved and may cause unpredictable behavior. The branch hint prefixes were introduced in the Pentium 4 and Intel Xeon processors as part of the SSE2 extensions.

The operand-size override prefix allows a program to switch between 16- and 32-bit operand sizes. Either operand size can be the default. This prefix selects the non-default size. Use of this prefix with MMX, SSE, and/or SSE2 instructions is reserved and may cause unpredictable behavior (see the note below).

The address-size override prefix allows a program to switch between 16- and 32-bit addressing. Either address size can be the default. This prefix selects the non-default size. Using this prefix when the operands for an instruction do not reside in memory is reserved and may cause unpredictable behavior.

**NOTE**

Some of the SSE and SSE2 instructions have three-byte opcodes. For these three-byte opcodes, the third opcode byte may be F2H, F3H, or 66H. For example, the SSE2 instruction CVTDQ2PD has the three-byte opcode F3 OF E6. The third opcode byte of these three-byte opcodes should not be thought of as a prefix, even though it has the same encoding as the operand size prefix (66H) or one of the repeat prefixes (F2H and F3H). As described above, using the operand size and repeat prefixes with SSE and SSE2 instructions is reserved. It should also be noted that execution of SSE2 instructions on an Intel processor that does not support SSE2 (CPUID Feature flag register EDX bit 26 is clear) will result in unpredictable code execution.

2.3. OPCODE

The primary opcode is 1, 2, or 3 bytes. An additional 3-bit opcode field is sometimes encoded in the ModR/M byte. Smaller encoding fields can be defined within the primary opcode. These fields define the direction of the operation, the size of displacements, the register encoding, condition codes, or sign extension. The encoding of fields in the opcode varies, depending on the class of operation.

2.4. MODR/M AND SIB BYTES

Most instructions that refer to an operand in memory have an addressing-form specifier byte (called the ModR/M byte) following the primary opcode. The ModR/M byte contains three fields of information:

- The *mod* field combines with the *r/m* field to form 32 possible values: eight registers and 24 addressing modes.
- The *reg/opcode* field specifies either a register number or three more bits of opcode information. The purpose of the *reg/opcode* field is specified in the primary opcode.
- The *r/m* field can specify a register as an operand or can be combined with the mod field to encode an addressing mode.

Certain encodings of the ModR/M byte require a second addressing byte, the SIB byte, to fully specify the addressing form. The base-plus-index and scale-plus-index forms of 32-bit addressing require the SIB byte. The SIB byte includes the following fields:

- The *scale* field specifies the scale factor.
- The *index* field specifies the register number of the index register.
- The *base* field specifies the register number of the base register.

See Section 2.6., “Addressing-Mode Encoding of ModR/M and SIB Bytes”, for the encodings of the ModR/M and SIB bytes.

2.5. DISPLACEMENT AND IMMEDIATE BYTES

Some addressing forms include a displacement immediately following the ModR/M byte (or the SIB byte if one is present). If a displacement is required, it can be 1, 2, or 4 bytes.

If the instruction specifies an immediate operand, the operand always follows any displacement bytes. An immediate operand can be 1, 2 or 4 bytes.

2.6. ADDRESSING-MODE ENCODING OF MODR/M AND SIB BYTES

The values and the corresponding addressing forms of the ModR/M and SIB bytes are shown in Tables 2-1 through 2-3. The 16-bit addressing forms specified by the ModR/M byte are in Table 2-1, and the 32-bit addressing forms specified by the ModR/M byte are in Table 2-2. Table 2-3 shows the 32-bit addressing forms specified by the SIB byte.

In Tables 2-1 and 2-2, the first column (labeled “Effective Address”) lists 32 different effective addresses that can be assigned to one operand of an instruction by using the Mod and R/M fields of the ModR/M byte. The first 24 effective addresses give the different ways of specifying a memory location; the last eight (specified by the Mod field encoding 11B) give the ways of specifying the general-purpose, MMX, and XMM registers. Each of the register encodings list five possible registers. For example, the first register-encoding (selected by the R/M field encoding of 000B) indicates the general-purpose registers EAX, AX or AL, MMX register MM0, or XMM register XMM0. Which of these five registers is used is determined by the opcode byte and the operand-size attribute, which select either the EAX register (32 bits) or AX register (16 bits).

The second and third columns in Tables 2-1 and 2-2 gives the binary encodings of the Mod and R/M fields in the ModR/M byte, respectively, required to obtain the associated effective address listed in the first column. All 32 possible combinations of the Mod and R/M fields are listed.

Across the top of Tables 2-1 and 2-2, the eight possible values of the 3-bit Reg/Opcode field are listed, in decimal (sixth row from top) and in binary (seventh row from top). The seventh row is labeled “REG=”, which represents the use of these 3 bits to give the location of a second operand, which must be a general-purpose, MMX, or XMM register. If the instruction does not require a second operand to be specified, then the 3 bits of the Reg/Opcode field may be used as an extension of the opcode, which is represented by the sixth row, labeled “/digit (Opcode)”. The five rows above give the byte, word, and doubleword general-purpose registers, the MMX registers, and the XMM registers that correspond to the register numbers, with the same assignments as for the R/M field when Mod field encoding is 11B. As with the R/M field register options, which of the five possible registers is used is determined by the opcode byte along with the operand-size attribute.

The body of Tables 2-1 and 2-2 (under the label “Value of ModR/M Byte (in Hexadecimal)”) contains a 32 by 8 array giving all of the 256 values of the ModR/M byte, in hexadecimal. Bits 3, 4 and 5 are specified by the column of the table in which a byte resides, and the row specifies bits 0, 1 and 2, and also bits 6 and 7.



INSTRUCTION FORMAT

Table 2-1. 16-Bit Addressing Forms with the ModR/M Byte

			AL AX EAX MM0 MM0 0 000	CL CX ECX MM1 MM1 1 001	DL DX EDX MM2 MM2 2 010	BL BX EBX MM3 MM3 3 011	AH SP ESP MM4 MM4 4 100	CH BP ¹ EBP MM5 MM5 5 101	DH SI EBP MM5 MM5 6 110	BH DI ESI MM6 MM6 7 111	
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hexadecimal)								
[BX+SI]	00	000	00	08	10	18	20	28	30	38	
[BX-DI]		001	01	09	11	19	21	29	31	39	
[BP+SI]		010	02	0A	12	1A	22	2A	32	3A	
[BP+DI]		011	03	0B	13	1B	23	2B	33	3B	
[SI]		100	04	0C	14	1C	24	2C	34	3C	
[DI]		101	05	0D	15	1D	25	2D	35	3D	
disp16 ²		110	06	0E	16	1E	26	2E	36	3E	
[BX]		111	07	0F	17	1F	27	2F	37	3F	
[BX+SI]+disp8 ³	01	000	40	48	50	58	60	68	70	78	
[BX-DI]+disp8		001	41	49	51	59	61	69	71	79	
[BP+SI]+disp8		010	42	4A	52	5A	62	6A	72	7A	
[BP+DI]+disp8		011	43	4B	53	5B	63	6B	73	7B	
[SI]+disp8		100	44	4C	54	5C	64	6C	74	7C	
[DI]+disp8		101	45	4D	55	5D	65	6D	75	7D	
[BP]+disp8		110	46	4E	56	5E	66	6E	76	7E	
[BX]+disp8		111	47	4F	57	5F	67	6F	77	7F	
[BX+SI]+disp16	10	000	80	88	90	98	A0	A8	B0	B8	
[BX-DI]+disp16		001	81	89	91	99	A1	A9	B1	B9	
[BP+SI]+disp16		010	82	8A	92	9A	A2	AA	B2	BA	
[BP+DI]+disp16		011	83	8B	93	9B	A3	AB	B3	BB	
[SI]+disp16		100	84	8C	94	9C	A4	AC	B4	BC	
[DI]+disp16		101	85	8D	95	9D	A5	AD	B5	BD	
[BP]+disp16		110	86	8E	96	9E	A6	AE	B6	BE	
[BX]+disp16		111	87	8F	97	9F	A7	AF	B7	BF	
EAX/AX/AL/MM0/XMM0 ECX/CX/CL/MM1/XMM1 EDX/DX/DL/MM2/XMM2 EBX/BX/BL/MM3/XMM3 ESP/SP/AHMM4/XMM4 EBP/BP/CH/MM5/XMM5 ESI/SI/DH/MM6/XMM6 EDI/DI/BH/MM7/XMM7	11	000	C0	C8	D0	D8	E0	E8	F0	F8	
		001	C1	C9	D1	D9	EQ	E9	F1	F9	
		010	C2	CA	D2	DA	E2	EA	F2	FA	
		011	C3	CB	D3	DB	E3	EB	F3	FB	
		100	C4	CC	D4	DC	E4	EC	F4	FC	
		101	C5	CD	D5	DD	E5	ED	F5	FD	
		110	C6	CE	D6	DE	E6	EE	F6	FE	
		111	C7	CF	D7	DF	E7	EF	F7	FF	

NOTES:

1. The default segment register is SS for the effective addresses containing a BP index, DS for other effective addresses.
2. The disp16 nomenclature denotes a 16-bit displacement that follows the ModR/M byte and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte and that is sign-extended and added to the index.



INSTRUCTION FORMAT

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(r) r16(r) r32(r) mm(r) xmm(r) /digit (Opcode) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hexadecimal)							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][-] ¹		100	04	0C	14	1C	24	2C	34	3C
disp32 ²		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8 ³	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][-]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][-]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

NOTES:

1. The [--][-] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement that follows ModR/M byte (or the SIB byte if one is present) and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows ModR/M byte (or the SIB byte if one is present) and that is sign-extended and added to the index.



INSTRUCTION FORMAT

Table 2-3 is organized similarly to Tables 2-1 and 2-2, except that its body gives the 256 possible values of the SIB byte, in hexadecimal. Which of the 8 general-purpose registers will be used as base is indicated across the top of the table, along with the corresponding values of the base field (bits 0, 1 and 2) in decimal and binary. The rows indicate which register is used as the index (determined by bits 3, 4 and 5) along with the scaling factor (determined by bits 6 and 7).

Table 2-3. 32-Bit Addressing Forms with the SIB Byte

r32 Base = Base =			EAX	ECX	EDX	EBX	ESP	[*]	ESI	EDI
Scaled Index	SS	Index	Value of SIB Byte (in Hexadecimal)							
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	89	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF
none		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

NOTE:

1. The [*] nomenclature means a disp32 with no base if MOD is 00, [EBP] otherwise. This provides the following addressing modes:

disp32[index] (MOD=00).
 disp8[EBP][index](MOD=01).
 disp32[EBP][index](MOD=10).

INSTRUCTION FORMAT

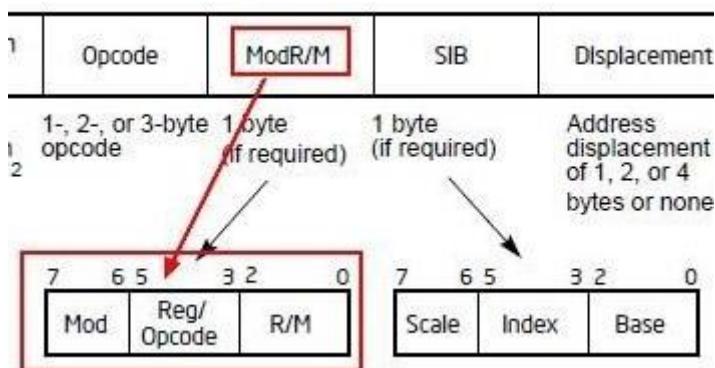


C14 - -01_Reprezentarea instr. masina - coloana 2 OllyDbg,

- 02_coduri de instructiuni (SetOpcode Table) + 02_BIS_Opcode_Extension Table

-03_Addressing Methods Codes - Iămurește cum să înțelegem OPERANZII instructiunilor din tabelele SetOpcode Table

-04 + 05 - Mod R/M byte + SIB bytes – Iămuresc modul de CODIFICARE în cadrul unei instructiuni (vezi coloana 2 din OllyDbg) al octetilor **Mod R/M** și **SIB** (structura lor este detaliată în 01_Reprezentarea instr. masina) – tabelele 2.2 (Mod R/M byte) și 2.3 (SIB byte)



Cum să citim tabelul corespunzător octetului ModR/M:

The second and third columns in Tables 2-1 and 2-2 gives the binary encodings of the Mod and R/M fields in the ModR/M byte, respectively, required to obtain the associated effective address listed in the first column. All 32 possible combinations of the Mod and R/M fields are listed.

Mod - 2 biți - sunt 4 valori posibile, prezentate pe COLOANA 2 (00, 01, 10, 11) – acest câmp indică moduri de combinație de tipuri de operanzi: 00 – “memorie (doar bază), registru” sau “registru, memorie(doar bază), 01 - “memorie (bază+disp8), registru” sau invers, 10 - “memorie (bază+disp32), registru” sau invers, 11 – “registru, registru”

Reg/Opcode - 3 biți – DACA E REGISTRU sunt 8 valori posibile (0-7, 000-111) prezentate pe LINIILE 6 și 7 - din prima linie de sus a tabelului (REG=) – acest element ALEGE **AL DOILEA OPERAND** - cel din dreapta, adică **SURSA !!!** (this denotes the second operand register if an instruction requires one !)

R/M - 3 biti - sunt 8 valori posibile (000-111) prezentate pe COLOANA 3 - **ALEGE PRIMUL OPERAND** - cel din stânga, adica **SURSA !!!** (the first operand)

Combinatiile celor 3 configurații de biti furnizeaza valoarea octetului R/M - prezentat in tabel in sectiunea din dreapta - **Value of Mod R/M Byte (in Hexadecimal)**

The ModR/M byte follows many opcodes to specify the addressing mode. This byte is fairly complicated but I'll try to explain it in this section. The diagram below shows how the byte is split into three fields: **mod selects the overall mode**, **reg selects a register**, and **r/m selects either a register or memory mode**.

Să luăm câteva exemple de valoare de octet Mod R/M (32-Bit Addressing Forms with the ModR/M Byte !!!!!) și să încercăm să deschidem ce reprezintă:

a). mov [ebp+a], ebx ; 899D 02204000 MOV Ev,Gv (89)

9Dh = 1001 1101 = 10 011 101 (Mod R/M)

Deci Mod = 10 - deci ma uit pe linia coresp. valorii 10 pt MOD ([...] + disp32)

Reg/Opcode = 011 - gasesc valoarea 011 pe coloana 4 a tabelului din dreapta (BL, BX, EBX, ...), care este alegerea depinde de tipul instructiunii (byte, word, dword - dat de octetul OPCODE). – al DOILEA operand

r/M = 101 - identifică linia [EBP]+disp32 din secțiunea Mod=10 (acesta este PRIMUL OPERAND - DESTINATIA !!!)

Deci dacă Mod R/M = 9Dh, setul de operanți va fi:

instructiune [EBP]+disp32, EBX adică o instructiune de tipul "instr memorie, registru"

E - A ModR/M byte follows the opcode and specifies the operand. The operand is either a general-purpose register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.

G - The reg field of the ModR/M byte selects a **general register** (for example, AX (000), EBX(011) etc.)

V - Word or doubleword, depending on operand-size attribute.

b). mov [edx],ebp ; 892A MOV Ev,Gv (89)

2Ah - 0010 1010 = 00 101 010 (Mod R/M)

Deci Mod = 00 - deci ma uit pe linia coresp. valorii 00 pt MOD ([...] - fără deplasamente) = [reg]

Reg/Opcode = 101 - gasesc valoarea 101 pe coloana 6 a tabelului din dreapta (CH, BP, EBP, ...), care este exact alegerea depinde de tipul instructiunii (byte, word, dword - dat de octetul OPCODE).

r/M = 010 - identifică linia [EDX] din secțiunea Mod=00 (acesta este PRIMUL OPERAND - DESTINATIA !!!)

Deci dacă Mod R/M = 2Ah, setul de operanți va fi:

instructiune [EDX], EBP (sau CH, sau BP) - adică tot o instructiune de tipul "instr memorie, registru"

c). `mov [edx+17], ecx` = **894A 11** 89 = MOV Ev,Gv (memory address, general register)

4Ah = 0100 1010 = **01 001 010** (Mod R/M)

Mod = **01** = [reg] + disp8

Reg/Opcode = **001** = ECX (operandul sursă – al DOILEA operand !) [EDX + disp8]

r/M = **010** - identifică linia [EDX] + disp8 din secțiunea Mod=01 (acesta este PRIMUL OPERAND - DESTINATIA !!!!)

In codificarea instrucțiunii mai apare în final **11 = 17** (în baza 10) – “The disp8 nomenclature denotes an 8-bit displacement that follows ModR/M byte”

d). `mov ebx, esi ; 89F3`

F3h = 1111 0011 = **11 110 011** (mod R/M)

Deci **Mod = 11** - deci mă uit pe linia coresp. valorii 11 pt MOD - aici este vorba despre un operand de tip REGISTRU și NU din memorie !

Reg/Opcode = **110** - gasesc valoarea 110 pe coloana 7 a tabelului din dreapta (DH, SI, ESI, ...), care este exact alegerea depinde de tipul instrucțiunii (byte, word, dword - dat de octetul OPCODE).

r/M = **011** - identifica linia EBX/BX/BL din secțiunea Mod=11 (acesta este PRIMUL OPERAND - DESTINATIA !!!!)

Deci dacă Mod R/M = F3h, setul de operanți va fi:

instrucțiune EBX (sau BX sau BL), ESI (sau SI, sau DH) - adică o instrucțiune de tipul "registrator, registrator"

e). `mov ecx, [ebx] ; 8B0B` **MOV Gv, Ev (8B)** **mov registrator, memorie**

0B = **00 001 011** (ModR/M)

Deci Mod = **00** - deci ma uit pe linia coresp. valorii 00 pt MOD ([...] - fără deplasamente)

Reg/Opcode = **001** - gasesc valoarea 001 pe coloana 2 a tabelului din dreapta (CL, CX, ECX, ...), care este exact alegerea depinde de tipul instrucțiunii (byte, word, dword - dat de octetul OPCODE). Aceasta este PRIMUL OPERAND (destinatia).

G – The reg field of the ModR/M byte selects a general register (for example, CX (001))

r/M = **011** - identifica linia [EBX] din secțiunea Mod=00 (acesta este AL DOILEA OPERAND - SURSA !!!)

f). `mov ecx, [esp + ebx*4] ; 8B0C9C MOV Gv, Ev (8B) mov registru, memorie`

$0C = \text{00 } \text{001 } \text{100}$ (mod R/M byte)

- similar cu cee ace este mai sus , mai putin câmpul r/M:

r/M = **100** - identifica linia [EBX] din sectiunea Mod=00 (1. The [--][--] nomenclature means a SIB follows the ModR/M byte). -

- **SIB_Byte Values - Structura lui este SS (SCALE - 2 biti) Index (3 biti) Base (3 biti)**

$9Ch = 1001\ 1100 = \text{10 } \text{011 } \text{100}$ (SIB Byte)

SS = 10 (factor de scalare = 4) - vezi coloana SS din tabel si lista de formule de adresare disponibila de pe prima coloana

Index = 011 (vezi coloana 3 din linia coresp. Lui SS=10) deci e vorba despre expresia de index $[EBX*4]$

Base = 100 (vezi linia 1) deci registrul de baza este ESP

Ca urmare formula de calcul al offsetului este **[ESP + EBX*4]**

Ca verificare, cautam in tabelul **Table 2-3. 32-Bit Addressing Forms with the SIB Byte** valoarea **9C** si o gasim la intersecția liniei $[EBX*4]$ din coloanele SS=10 si Index = 011 si in cadrul coloanei ESP (de valoare 4 = 100)

g). `mov ecx, [esp + ebx*4 + a - 7] ; 8B8C9C FB1F4000 MOV Gv, Ev (8B) mov registru, memorie`

(`mov ecx, dword ptr SS:[ebx*4 + esp + 00401FFB]`) $00402002 - 7 = 00401FFB$

$8C = \text{10 } \text{001 } \text{100}$ (mod R/M byte)

Deci Mod = **10** - deci ma uit pe linia coresp. valorii 10 pt MOD ([...] + disp32)

Reg/Opcode = **001** - gasesc valoarea 001 pe coloana 2 a tabelului din dreapta (CL, CX, ECX, ...). Aceasta este PRIMUL OPERAND (destinatia) = **ECX**

r/M = **100** - identifica din secțiunea Mod=10 (1. The [--][--]+disp32 nomenclature means a SIB follows the ModR/M byte).

9C = SIB byte (same as above)

FB1F4000 = disp32 (= a-7 = 00401FFB)

15 Ianuarie 2025

- rolul principal al unui asamblor= generarea corespunzatoare de octeti
- la un moment dat poate fi activ numai unul din segmentele de fiecare tip(ex. doar singur segment de cod activ din N)
- sub 16 biti registrii segment cs, ds, ss, es contin adresele de INCEPUT ale segmentelor active la un moment dat**
- sub 32 biti registrii segment cs, ds, ss, es contin valorile SELECTORILOR de segment active**
- la orice moment al executiei combinatia de registrii cs:eip exprima /contine adresa instructiunii curente de executat**
- aceste valori sunt manipulate exclusiv de catre BIU
- in cadrul unei instructiuni NU putem avea ambii operanzi expliciti din memoria RAM
- BIU poate "aduce" numai un singur operand din memorie odata (ne-ar trebui 2 BIU, 2 seturi de registrii)
$$\text{adresa_offset} = [\text{bază}] + [\text{index} \times \text{scală}] + [\text{constanta}]$$

(SIB) (deplasament + imediat)

FORMATUL INTERN AL UNEI INSTRUCTIUNI ESTE:

$$[\text{prefixe}] + \text{cod} + [\text{ModR/M}] + [\text{SIB}] + [\text{deplasament}] + [\text{imediat}]$$

- octetul ModR/M poate exprima operanzi de tip regisztru și/sau de tip memorie de adresare indirectă în care apare doar [baza]; dacă apare și partea de [index*scală] atunci este nevoie SI de octetul SIB !! Deci: dacă Modr/m ne spune ca operandul e in memorie => octetul SIB apare obligatoriu NUMAI DACA AVEM SI PARTEA DE [index × scală], urmat EVENTUAL si de deplasament si/sau imediat**
- Deci primele 2 elemente din formula de calcul a offsetului unui operand (baza și index*scala) sunt exprimate sau precizate prin octetii ModR/m și SIB din formatul intern al unei instructiuni**
- al treilea element: constanta, daca apare, este exprimata de campurile deplasament sau/și imediat (displacement and/or immediate)**

- daca Modr/m imi spune ca am doar regiszru pe post de operand urmatoarele 3 campuri din formula nu mai apar (deoarece daca operandul este regiszru NU poate fi in acelasi timp si operand din memorie si operand imediat)

- campul imediat poate pe de o parte sa participe la calculul offsetului unui operand din memorie (furnizand campul constanta din formula offsetului) sau poate sa apara de sine statator exprimand valoarea imediata a unui operand (Ex: mov ebx, 12345678)

- campul deplasament dacă apare singur în formula de calcul a offset-ului exprima modul de adresare directă la memorie

- campul imediat=constante numerice

- adresarea directă presupune accesul direct la operandul din memorie pe baza deplasamentului sau, fără ca în formula de calcul a offsetului să apara vreun regiszru (deci fără baza sau index !)

- daca apar regiszrii în calculul offsetului => adresare indirectă

- în cadrul instrucțiunilor în care vom folosi doar exprimari de offseturi, acestea (offseturile) vor fi prefixate în mod IMPLICIT de către asamblor de unul dintre regiszrii de segment CS, DS,SS sau ES. (ex. în debugger push variabila -> DS:[40100...])

CS:EIP – Adresa FAR (completa) a instrucțiunii curente de executat

EIP – incrementat automat după executia instrucțiunii curente

CS – contine selectorul de segment a segmentului de cod curent (activ) și poate fi modificat numai dacă executia “sare” într-un alt segment (JMP FAR..)

Mov cs, [var] – ok sintactic (illegal instruction in OllyDbg...)

Mov eip, eax – syntax error – symbol ‘eip’ undefined

Jmp FAR unde_in_memorie; se modifica și CS și EIP !

Jmp start1 ; salt NEAR – se modifica doar offset-ul, deci numai EIP !

Pt Examenul Scris:

a). Formula:

Formatul intern al unei instrucțiuni este variabil, el putând ocupa între 1 și 15 octeți, având următoarea formă generală de reprezentare (*Instructions byte-codes from OllyDbg*):

[prefixe] + cod + [ModR/M] + [SIB] + [deplasament] + [immediat]

- Prefixele – material separat – 4 categorii (2 explicate, 2 implicate – 66h și 67h)
 - Clasificare și câteva exemple pregătite de acasă (diferite de ceea ce apare pe slides) care să poată fi explicate în scris – câte 2 din fiecare categorie; Câte prefixe pot apărea simultan în cadrul unei instrucțiuni (exemplu !)
 -

Octet cod – obligatoriu (exemplu)

CE reprezintă și rolul său în reprezentarea unei instrucțiuni mașină octetii ModR/M și SIB și care este STRUCTURA LOR ?

(câte UN exemplu de instrucțiune de tipul celor de la curs cu identificarea acestor octeți ca structură și conținut în formatul intern al acestora reflectat în OllyDbg)

Ce exprimă câmpurile *[deplasament]* și *[immediat]*

Exemple de format intern care conțin aceste câmpuri cu explicații



Bitdefender® Awake.

Programare multi-modul

Marius Vanță

mvanta@bitdefender.com

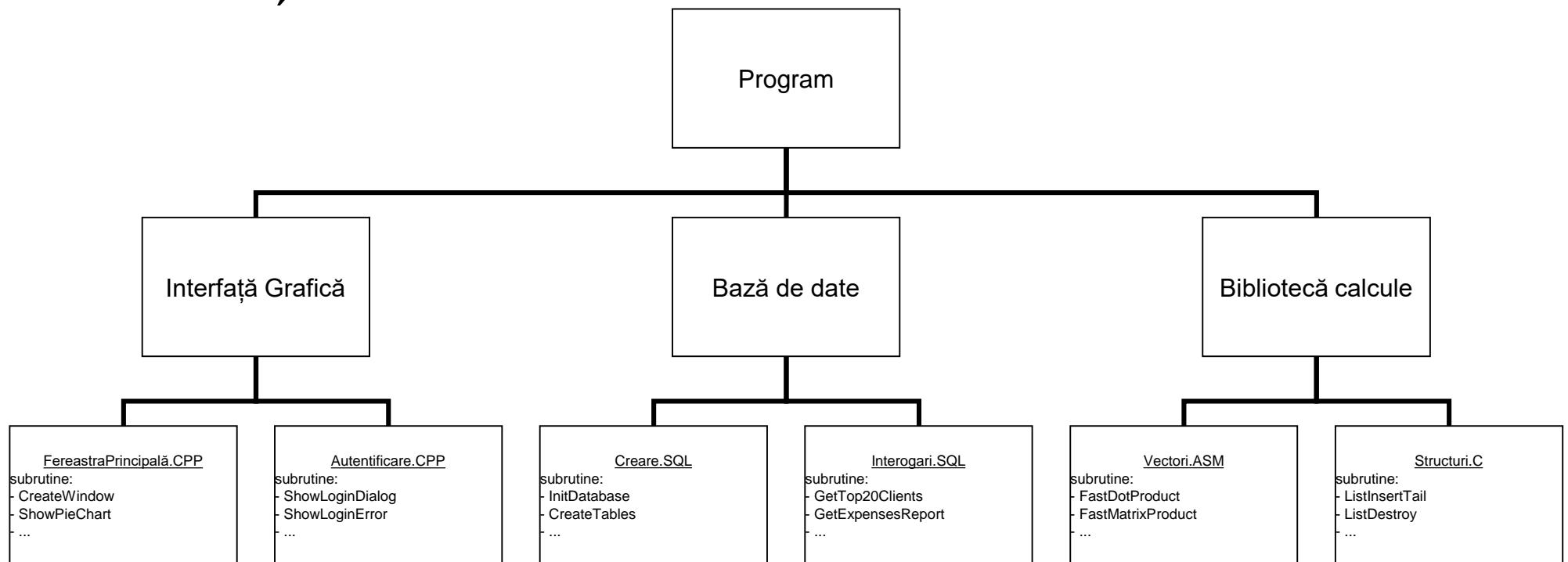
1. Arhitecturi modulare

Programare modulară

- Cum împărțim problema în sub-probleme?

- Modularizare:

- program -> unități logice
- cod (al unităților) -> fișiere distincte
- Fișiere -> subroutines



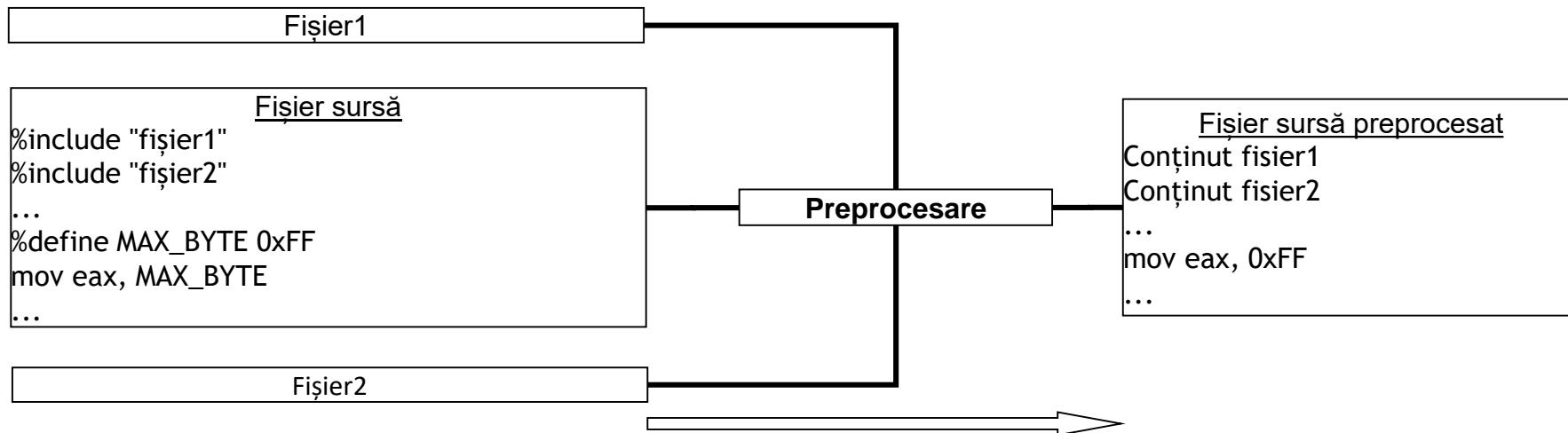
Programare modulară

- Pentru care sub-probleme există deja rezolvări disponibile?
 - Reutilizare:
 - Fișiere sursă
 - *Refolosire cod și date din asamblare*
 - Directiva %include (NU este programare multimodul, deoarece la compilare ajunge DOAR 1 SINGUR MODUL obținut prin concatenarea textuală a fisierelor incluse!!)
 - Fișiere binare
 - *Refolosire cod și date din asamblare*
 - *Cod și date din limbaje de nivel înalt*
 - *Biblioteci*
 - Existenta de fisiere binare separate implica COMPILEARE SEPARATA !!!

2. Tehnici și instrumente

Tehnici și instrumente

- Includerea **statică la compilare/asamblare**: directiva **%include**
 - Specifică limbajului (dar are echivalent și în alte limbaje)
 - Modularizare: permite doar divizarea codului scris în acel limbaj!
 - NU este programare multimodul! (aceasta necesita COMPILEARE SEPARATA !!!)
 - Reutilizare: expune codul sursă!
 - Periculos și problematic:
 - Mecanism de procesor -> concatenare textuală a fișierelor
 - Expune cu vizibilitate globală toate denumirile -> conflicte (redefiniții/redeclarări)
 - Include fișierul în întregime - și ce se folosește și ce nu!



Tehnici și instrumente

- Exemplu folosire **%include**

```
; fișierul constante.inc

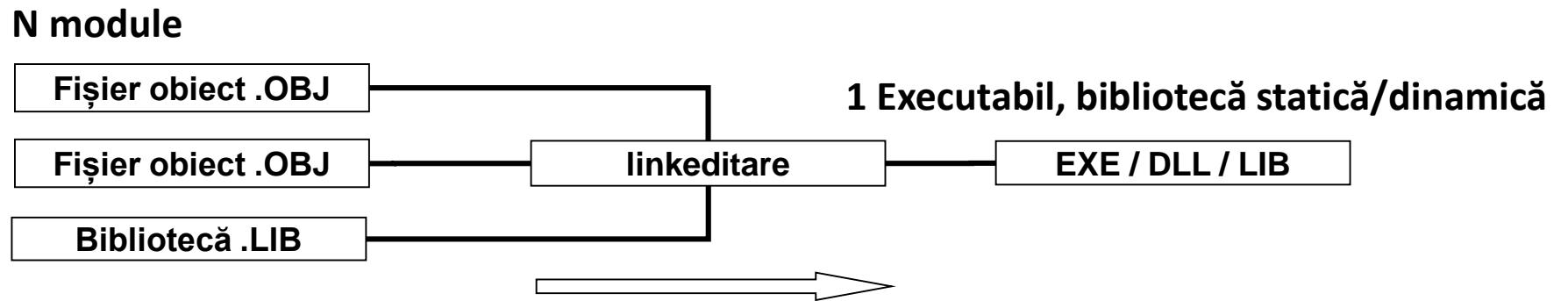
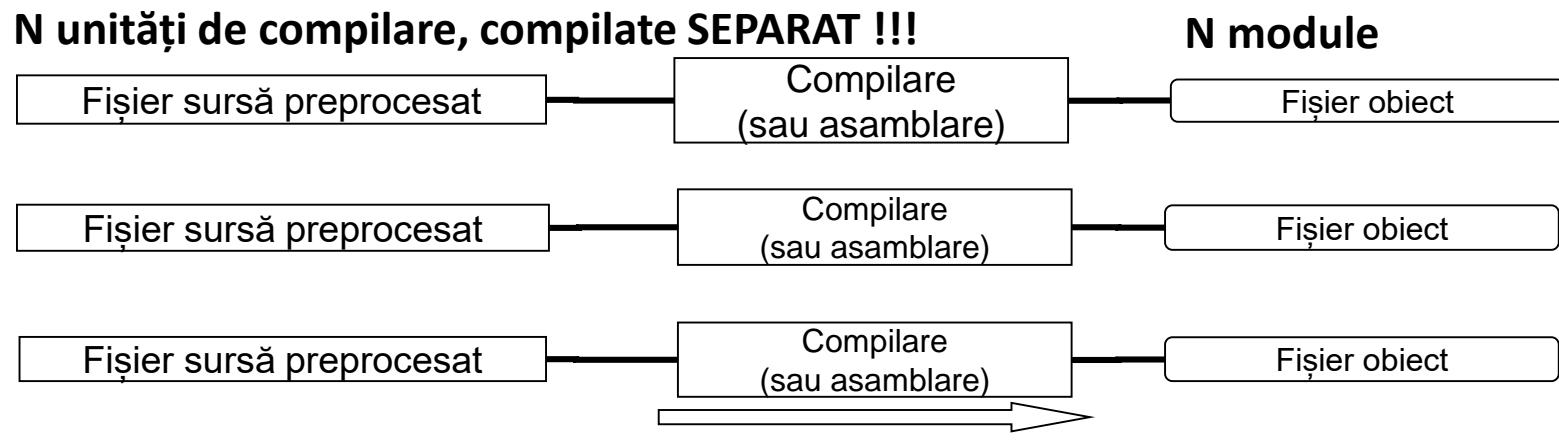
; gardă dublă-includere
#ifndef      _CONSTANTE_INC_ ; la prima includere, _CONSTANTE_INC_ nu este definit
#define       _CONSTANTE_INC_ ; definim _CONSTANTE_INC_ -> condiție falsă la viitoare inclunderi

; recomandat ca astfel de fișiere (incluse de către altele) să conțină (doar) declarații!
MAX_BYT    equ 0xFF
MAX_WORD   equ 0xFFFF
MAX_DWORD  equ 0xFFFFFFFF
MAX_QWORD  equ 0xFFFFFFFFFFFFFFFF

#endif ; _CONSTANTE_INC_
```

Tehnici și instrumente

- Legarea **statică** la **linkeditare**
 - pas realizat de către un **linkeditor** după asamblare/compilare



Tehnici și instrumente

• Legarea statică la linkeditare – sumar responsabilități

- **Preprocesor: text => text**
 - Efectuează prelucrări asupra *textului* sursă, rezultând un *text* sursă intermediu
 - Se poate imagina ca fiind o componentă a compilatorului sau asamblorului
 - Poate lipsi, multe limbaje nu au un preprocesor!
- **Asamblor: instrucțiuni (text) => codificare binară (fișier obiect)**
 - Codifică instrucțiunile și datele (variabilele) din textul sursă preprocesat și construiește un fișier obiect ce conține cod mașină și valori de variabile alături de informații despre conținut (denumiri de variabile, subrute, informații despre tipul și vizibilitatea acestora etc)
- **Compilator: instrucțiuni (text) => codificare binară (fișier obiect)**
 - Identifică secvențe de instrucțiuni de procesor prin care se pot obține funcționalitățile descrise în textul sursă *iar apoi, precum un asamblor*, generează un fișier obiect ce conține codificarea binară a acestora și a variabilelor din program
 - Asamblarea este un caz special de compilare, unde instrucțiunile de procesor sunt gata oferite direct în textul programului și ca atare nu necesită să fie alese de către compilator!
- **Linkeditor: fișiere obiect => bibliotecă sau program**
 - Construiește rezultatul final, adică un program (.exe) sau bibliotecă (.dll sau .lib) în care *împreună* (include) codul și datele binare prezente în fișierele obiect.
 - Nu este interesat în ce compilatoare sau ce limbaje au fost folosite! Legarea necesită doar ca fișierele de intrare să respecte formatul standard al fișierelor obiect!

Tehnici și instrumente

- Exemplu folosire **%include** – împachetare eax într-un BYTE/WORD/DWORD, conform magnitudinii valorii acestuia

```
; fișierul program.asm
%include "constante.inc"

    cmp    eax, MAX_BYTE
    ja     .nu_incape_in_octet      ; încape valoarea din eax într-un byte?

.nu_incape_in_octet:
    mov    [rezultat_octet], al      ; dacă da, salvăm AL în rezultat_octet
    jmp    .gata

.nu_incape_in_octet:
    cmp    eax, MAX_WORD
    ja     .nu_incape_in_cuvant    ; altfel verificăm dacă ajunge un WORD

.nu_incape_in_cuvant:
    mov    [rezultat_word], ax      ; dacă da, salvăm AX în rezultat_word
    jmp    .gata

.nu_incape_in_cuvant:
    mov    [rezultat_dword], eax    ; dacă nu ajunge un WORD, salvăm întreg eax
.gata:
```

Tehnici și instrumente

- Legarea **statică** la **linkeditare** – cerințele nasm
 - Resursele sunt partajate de comun acord
 - *Export* prin **global** nume1, nume2, ...
 - Ofer disponibilitate oricărui fișier ar fi interesat
 - *Import* prin **extern** nume1, nume2, ...
 - Solicit acces, indiferent din ce fișier va fi oferită resursa
 - Solicitare fără disponibilitate = eroare!
 - Nu se pot importa decât resurse ce sunt exportate undeva
 - Însă disponibilitate fără solicitare este caz permis. De ce?
 - Răspuns: chiar dacă niciun modul din program nu solicită/folosește, poate se va utiliza într-o versiune viitoare sau de către un alt program.
 - Limbajele de programare de nivel mai înalt oferă și ele la rândul lor construcții sintactice cu rol echivalent!
 - Exemplu: în limbajul C
 - *Disponibilitatea este automată/implicită, putându-se însă opta pentru a bloca accesul prin folosirea cuvântului cheie **static***
 - *Solicitarea de acces se face (tot) prin intermediul cuvântului cheie **extern***

Tehnici și instrumente

• Legarea statică la linkeditare

- Permite unirea mai multor **module binare** (fișiere obiect sau biblioteci statice) într-un singur fișier
 - Intrări: oricâte fișiere obiect (**.OBJ**) și/sau biblioteci statice (**.LIB**)
 - *Atenție, nu toate fișierele .LIB sunt biblioteci statice!*
 - Ieșire: .EXE sau .LIB sau .DLL (Dynamic-Link Library)
- Multimodul: oricâte fișiere pot fi compilate separat și linkeditate împreună
 - Pas realizat de linkeditor *după* compilare/asamblare -> **nu depinde de limbaj!**
- Reutilizare:
 - În formă binară - nu expune codul sursă!
 - Permite inter-operabilitate între limbi diferite!
- Alte avantaje și dezavantaje:
 - Editorul de legături *poate* identifica și elimina resurse neutilizate sau efectua alte optimizări
 - Dimensiune mare a programului: programul înglobează resursele externe reutilizate
 - Dimensiune mare a programelor: bibliotecile populare duplicate în multe programe
- NASM: directivele **global (mecanism export)** și **extern (mecanism import)**
 - global *nume* – oferirea posibilității de utilizare din exterior a acestei resurse date prin nume
 - extern *nume* – solicitare de acces la resursa specificată; necesită să fie publică!

Tehnici și instrumente

- Legarea **statică** la **linkeditare** – cerințele nasm

- Folosirea în practică a directivelor global și extern

; FIŞIER1.ASM

```
global Var1, Subrutina2  
extern Var3, Subrutina3  
Subrutina1:
```

```
....  
apel(Subrutina3)  
....  
operatii(Var3)  
....
```

Subrutina2:

```
....  
Var1 dd ...  
Var2 db ...
```

; FIŞIER2.ASM

```
extern Var1, Subrutina2  
global Subrutina3, Var3  
Subrutina3:
```

```
....  
apel(Subrutina2)  
....  
operatii(Var1)  
....
```

Subrutina1:

```
....  
Var2 db ...  
Var3 dd ...
```

Tehnici și instrumente

- Exemplu program multimodul nasm + nasm
 - Pașii necesari construirii programului executabil final
 - Se asamblează fișierul main.asm
 - *nasm.exe -fobj main.asm*
 - Se asamblează fișierul sub.asm
 - *nasm.exe -fobj sub.asm*
 - Se editează legăturile dintre cele două module
 - *alink.exe main.obj sub.obj -oPE –entry:start –subsys:console*
 - Observație: cele două module pot fi asamblate în orice ordine! Abia în timpul linkeditării este necesar ca simbolurile referite să aibă cu toate implementare disponibilă în unul dintre fișierele obiect oferite linkeditorului.
 - Linkeditarea, în mod evident, este posibilă doar după asamblare/compilare!

Tehnici și instrumente

- Legarea **statică** la **linkeditare** – cerințele nasm

- Folosirea în practică a directivelor global și extern

; FIŞIER1.ASM

global Var1, Subrutina2

extern Var3, Subrutina3

Subrutina1:

....

apel(Subrutina3)

....

operatii(Var3)

....

Subrutina2:

....

Var1 dd ...

Var2 db ...

; FIŞIER2.ASM

extern Var1, Subrutina2

global Subrutina3, Var3

Subrutina3:

....

apel(Subrutina2)

....

operatii(Var1)

....

Subrutina1:

....

Var2 db ...

Var3 dd ...

Pot fi refolosite
denumirile cât timp
nu sunt globale!

Tehnici și instrumente

- Exemplu program multimodul nasm + nasm

; MODULUL MAIN.ASM

```
global SirFinal
extern Concatenare
```

```
import printf msvcrt.dll
import exit msvcrt.dll
extern printf, exit
global start
```

```
segment code use32 public code class='code'
    start:
```

```
        mov eax, Sir1
        mov ebx, Sir2
        call Concatenare
        push dword SirFinal
        call [printf]
        add esp, 1*4
        push dword 0
        call [exit]
```

```
segment data use32
```

```
    Sir1 db 'Buna ', 0
    Sir2 db 'dimineata!', 0
    SirFinal resb 1000 ; spatiu pentru rezultat
```

; MODULUL SUB.ASM

```
extern SirFinal
global Concatenare
```

```
segment code use32 public code class='code'
; eax = adresa primului sir, ebx = adresa sirului secund
Concatenare:
    mov edi, SirFinal ; destinatie = SirFinal
    mov esi, eax ; sursa = primul sir
.sir1Loop:
    lodsb ; luam octetul urmator
    test al, al ; este terminatorul de sir (=0)?
    jz .sir2 ; daca da, trecem la sirul al doilea
    stosb ; (altfel) copiem in destinatie
    jmp .sir1Loop ; si continuam pana la nul
.sir2:
    mov esi, ebx ; sursa = sirul al doilea
.sir2Loop:
    lodsb ; acelasi proces pentru noul sir
    test al, al
    jz .gata
    stosb
    jmp .sir2Loop
.gata:
    stosb ; adaugam terminatorul de sir din al
    ret
```

Tehnici și instrumente

- Legarea **statică la linkeditare**: nasm + limbaje de nivel înalt
 - Cerințe ale editorului de legături
 - Directiva global pentru permis acces din alt limbaj către etichetele noastre
 - Directiva extern pentru obținut acces în NASM către resursele implementate în alte limbaje
 - Declararea în limbajul de nivel înalt a variabilelor și subroutines definite în nasm
 - *Exemplu C: declaratorul extern!*
 - Intrarea în procedură
 - Nealterarea valorilor unor registri
 - Transmiterea și accesarea parametrilor
 - Alocarea de spațiu pentru datele locale (optional)
 - Întoarcerea unui rezultat (optional)
- Ultimele aspecte sunt discutate in detaliu la convenții de apel!
 - vezi interfațarea cu limbajele de nivel inalt: convenții de apel

Tehnici și instrumente

- Exemplu program multimodul asm + C

```
//  
// AFISARE.C  
  
// solicita catre preprocesorul de C includerea fisierului stdio.h  
// stdio.h declara antetul (tipul de rezultat si parametri) functiei C printf  
#include <stdio.h>  
  
// declarăm funcția din fisierul asm incat compilatorul C să cunoască tipul de parametri și rezultat  
// linkeditorul se va ocupa de implementarea funcției, compilatorul necesită doar să-i cunoască antetul  
void asm_start(void); //echivalent ca efect cu extern void asm_start(void) ! Orice funcție declarată la nivelul cel mai  
// exterior al unui modul C face parte din clasa de memorie extern  
// funcția afisare este apelată de catre codul asm  
void afisare(int *vector, int numar_elemente) //orice funcție definită la nivelul cel mai exterior al  
{  
    int index;  
    for (index = 0; index < numar_elemente; index++)  
    {  
        printf("%d", vector[index]);  
    }  
    printf("\n");  
}  
  
// programul principal, acesta apelează funcția asm_start scrisă în assembly  
void main(void) // de aici începe execuția programului final  
{  
    asm_start(); // apelăm funcția din fisierul asm  
}
```

Tehnici și instrumente

- Exemplu program multimodul asm + C

```
;  
; VECTOR.NASM  
;  
  
; informam asamblorul despre existenta functiei afisare  
extern _afisare           ; atentie la adaugarea _ ca prefix al a numelor provenite din C!  
  
; informam asamblorul ca dorim ca asm_start sa fie disponibil altor unitati de compilare  
global _asm_start          ; atentie la adaugarea _ ca prefix al numelor referite de catre C!  
  
;  
; codul asm este dispus intr-un segment public, posibil a fi partajat cu alt cod extern  
segment code public code use32  
  
_asm_start:  
    push dword elemente ; parametru transmis prin valoare (se urca in stiva valoarea 5)  
    push dword vector   ; vectorul este transmis prin referinta (adresa lui este copiată pe stivă)  
    call _afisare        ; apelul functiei C, din nou cu prefix _  
    add esp, 4*2         ; afisare este o functie C (cdecl) -> necesita ca NOI să eliberam argumentele!  
    ret                  ; revenire la codul C care ne-a apelat  
  
;  
; linkeditorul poate folosi segmentul public de date si pentru date din afara  
segment data public data use32  
    vector dd 1, 2, 3, 4, 5      ; vectorul ce-l vom afisa cu rutina C  
    elemente equ ($ - vector) / 4 ; constantă egală cu 5 (numarul elementelor din vector)
```

Tehnici și instrumente

- Exemplu program multimodul asm + C
 - De ce _ ?
 - Construire executabil:
 1. Compilare/asamblare:
 - *afisare.c poate fi compilat cu orice compilator C (după preferințe) -> afisare.obj*
 - Visual C: `cl /c afisare.c`
 - `nasm.exe vector.asm -fwin32 -o vector.obj`
 - 2. Editarea legăturilor:
 - *Se apelează orice linkeditor compatibil C, solicitând:*
 - Intrări: afişare.obj și vector.obj
 - Ieşire: aplicație de consolă
 - `link vector.obj afisare.obj /OUT:afisare.exe /MACHINE:X86 /SUBSYSTEM:CONSOLE`
 - Alternativ, fișierele pot fi înglobate într-o "soluție" Visual Studio, instruind IDE-ul:
 1. Să asambleze fișierul asm: specificând de exemplu drept **Pre-Build Event** comanda de asamblare de mai sus (`nasm.exe vector.asm -fwin32 -o vector.obj`)
 2. Să includă afisare.obj drept intrare adițională la linkeditare
 3. Există *extensii* pentru Visual Studio care rezolvă automat și transparent problema!



Bitdefender® Awake.

Programare multi-modul

Marius Vanță

mvanta@bitdefender.com

3. Interfațarea cu limbajele de nivel înalt

Interfațarea cu limbajele de nivel înalt

- Transmiterea parametrilor **prin valoare**
 - Se copiază valorile octetilor ce compun datele
 - Risipitor și lent când datele ocupă multă memorie!
 - Din perspectiva apelantului, datele sunt constante!
- Transmiterea parametrilor **prin referință** (adresă(pointer))
 - Se specifică adresa (și uneori dimensiunea) datelor
 - Ineficient când datele ocupă puțin
 - 32 biți în plus pentru stocat adresa + citire pointer înainte de acces la date
 - Datele pot suferi modificări
- Decizia transmitere prin valoare sau prin referință
 - Criteriul de performanță: dimensiunea în octeți
 - Risc de a depăși memoria disponibilă? Prin referință!
 - Criteriul de accesibilitate: trebuie modificate datele?
 - Neapărat constante? Prin valoare!

Interfațarea cu limbajele de nivel înalt

- Convenții de apel (call conventions)
 - Cum transmitem parametri către subrutine?
 - Ce tipuri de parametri se pot transmite?
 - În ce ordine?
 - Câți parametri? Oricâți?
 - Ce resurse sunt volatile (poate să le altereze funcția apelată)?
 - Unde se regăsește rezultatul?
 - Ce acțiuni de curățare (cleanup) sunt necesare post-apel?
 - Cine este responsabil să le efectueze?
 - Convenții
 - Uzuale: **CDECL**, **STDCALL**, FASTCALL
 - Rar folosite sau învechite: PASCAL, FORTRAN, SYSCALL, etc...
 - Utilizator: **în asamblare toate aspectele documentate de către o convenție sunt accesibile programatorului!**

Interfațarea cu limbajele de nivel înalt

- Convenții de apel – convenția C (**CDECL**)
 - Specifică limbajului C
 - Cum transmitem parametri către subrutine? Prin împingerea lor pe stivă
 - Ce tipuri de parametri se pot transmite? Orice, dar necesită extins la minim DWORD
 - În ce ordine? Dreapta către stânga, adică, invers ordinii de la declarație
 - Câtări parametri? Oricărți? Da, C permite funcții cu oricărți parametri (ex: printf)
 - Ce resurse sunt volatile? **EAX, ECX, EDX, Eflags**
 - Unde se regăsește rezultatul? EAX, EDX:EAX sau ST0 (FPU)
 - Ce acțiuni de curățare (cleanup) sunt necesare? Eliberarea argumentelor
 - Cine este responsabil să le efectueze ? Apelantul!

Parametri			Resurse volatile	Rezultate	Eliberare
Stocare	Ordine	Număr			
Stivă	Inversă	<u>Oricărți</u>	EAX, ECX, EDX, Eflags	EAX / EDX:EAX / ST0 (FPU)	<u>Apelant</u>

Interfațarea cu limbajele de nivel înalt

- Convenții de apel – convenția **STDCALL**

- Specifică sistemului de operare Windows
 - Denumită și **WINAPI**
 - Folosită de către bibliotecile de sistem Windows
- Foarte asemănătoare convenției CDECL
 - Diferențe:
 - Număr fix de parametri
 - Eliberarea argumentelor o face funcția apelată

Parametri			Resurse volatile	Rezultate	Eliberare
Stocare	Ordine	Număr			
Stivă	Inversă	<u>Fix</u>	EAX, ECX, EDX, EFlags	EAX / EDX:EAX / ST0 (FPU)	<u>Subrutina apelată</u>

Interfațarea cu limbajele de nivel înalt

- Apelul subrutinelor
 - Etape:
 1. **Cod de apel**: pregătirea și efectuarea apelului
 2. **Cod de intrare**: intrarea în procedură și pregătirea execuției
 3. **Cod de ieșire**: revenire și eliberarea resurselor ce au expirat
 - Acțiunile depind în funcție de convenția de apel a subrutinei apelate – dar etapele rămân aceleași!
 - Etapele sunt tratate/implementate **automat** în codul generat de către compilatoarele limbajele de nivel mai înalt
 - În asamblare **rămâne totul în sarcina noastră!**

Interfațarea cu limbajele de nivel înalt

- Apelul subruteinelor – cod de apel
 - Sarcini:
 1. Salvare resurse volatile în uz: *push regisztr*
 2. Asigurare respectare constrângeri (ESP aliniat, DF=0, ...)
 3. Pregătire argumente (stivă, conform convenției): *push*
 4. Efectuare apel: *call*
 - *call subrutină când subrutină este linkeditată static*
 - *call [subrutină] dacă este dinamică (la link-time)*
 - *call regisztr sau call [variabilă] pentru dinamică la runtime*
 - Subruteinele asm folosite doar din asm pot evita (din simplitate și/sau eficiență) aceste sarcini

Interfațarea cu limbajele de nivel înalt

- Apelul subruteinelor – cod de apel

- Exemplu: apel printf din asm pentru afișare numere 0..9

```

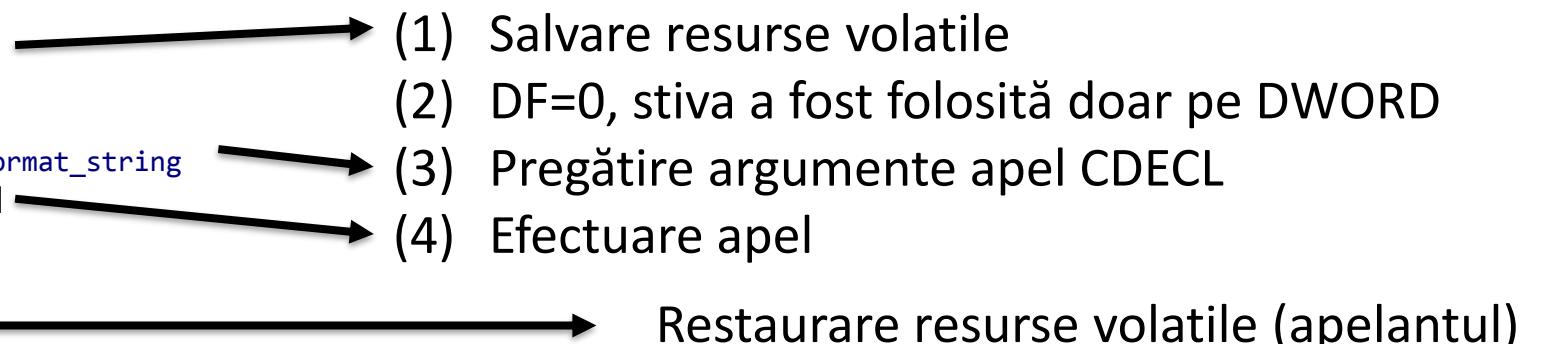
import exit msvcrt.dll
import printf msvcrt.dll
extern exit, printf
global start

segment code use32
    start:
        mov ecx, 10
        xor eax, eax
    .next:
        push eax
        push ecx
        push eax
        push dword format_string
        call [printf]
        add esp, 2*4
        pop ecx
        pop eax
        inc eax
    loop .next
    push dword 0
    call [exit]

segment data use32
    format_string db "%d", 10, 13, 0

```

Dacă apelul se făcea din C, compilatorul ar fi generat singur codul de apel!
 Cum apelul se face din asamblare, codul de apel trebuie scris de către noi!

- 
- (1) Salvare resurse volatile
 (2) DF=0, stiva a fost folosită doar pe DWORD
 (3) Pregătire argumente apel CDECL
 (4) Efectuare apel
- Restaurare resurse volatile (apelantul)

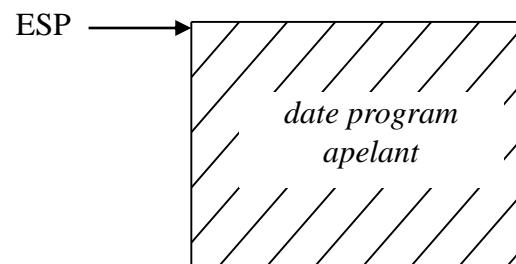
Interfațarea cu limbajele de nivel înalt

- Apelul subruteinelor – cod de apel
 - Efectul codului de apel asupra stivei

```
push eax  
push ecx
```

```
push eax  
push dword format_string  
call [printf]  
add esp, 2*4
```

Stare inițială



Interfațarea cu limbajele de nivel înalt

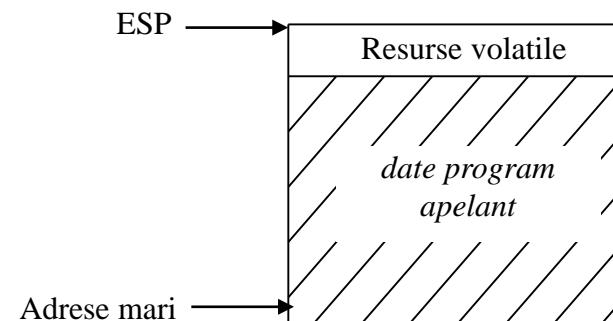
- Apelul subruteinilor – cod de apel
 - Efectul codului de apel asupra stivei

```
push eax  
push ecx
```

→ (1) Salvare resurse volatile

```
push eax  
push dword format_string  
call [printf]  
add esp, 2*4
```

(1) Salvare resurse volatile



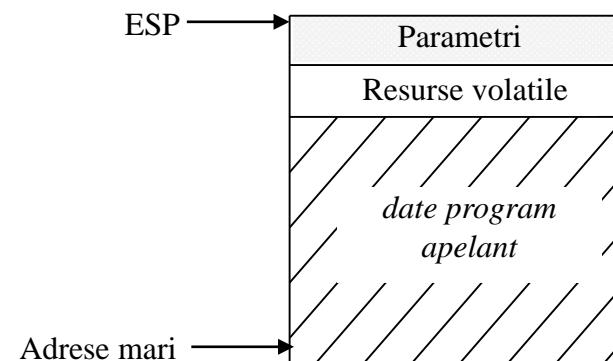
Interfațarea cu limbajele de nivel înalt

- Apelul subruteinelor – cod de apel
 - Efectul codului de apel asupra stivei

```

push eax      → (1) Salvare resurse volatile
push ecx
push eax      → (2) DF=0, stiva a fost folosită doar pe DWORD
push dword format_string
call [printf]
add esp, 2*4  → (3) Pregătire argumente apel CDECL
    
```

(3) Pregătire argumente apel CDECL

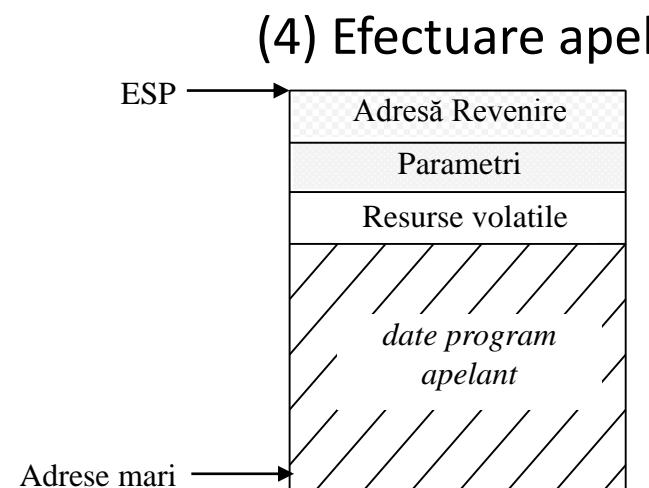


Interfațarea cu limbajele de nivel înalt

- Apelul subruteinelor – cod de apel
 - Efectul codului de apel asupra stivei

```

push eax      → (1) Salvare resurse volatile
push ecx
push eax      → (2) DF=0, stiva a fost folosită doar pe DWORD
push dword format_string
call [printf]
add esp, 2*4  → (3) Pregătire argumente apel CDECL
                  → (4) Efectuare apel
    
```



Interfațarea cu limbajele de nivel înalt

- Apelul subruteinelor – cod de intrare
 - Sarcini:
 1. Configurarea unui **cadru de stivă** (stack frame): reper ebp sau esp?
 2. Pregătire variabile locale ale funcției: sub esp, număr_octetii
 3. Salvarea unei copii a resurselor *nevolațile* modificate: push registru
 - *Orice registri cu excepția celor volatili*
 - Cadru de stivă: structură de date stocată în stivă, de dimensiune fixă (pentru o subrutină dată) și conținând:
 - *Parametrii pregătiți de apelant*
 - *Adresa de revenire (către instrucțiunea ce-i urmează celei de apel)*
 - *Copii ale resurselor nevolațile folosite de subrutină*
 - *Variabile locale*
 - Subruteinele asm folosite doar din asm pot evita (din simplitate sau eficiență) aceste sarcini

Interfațarea cu limbajele de nivel înalt

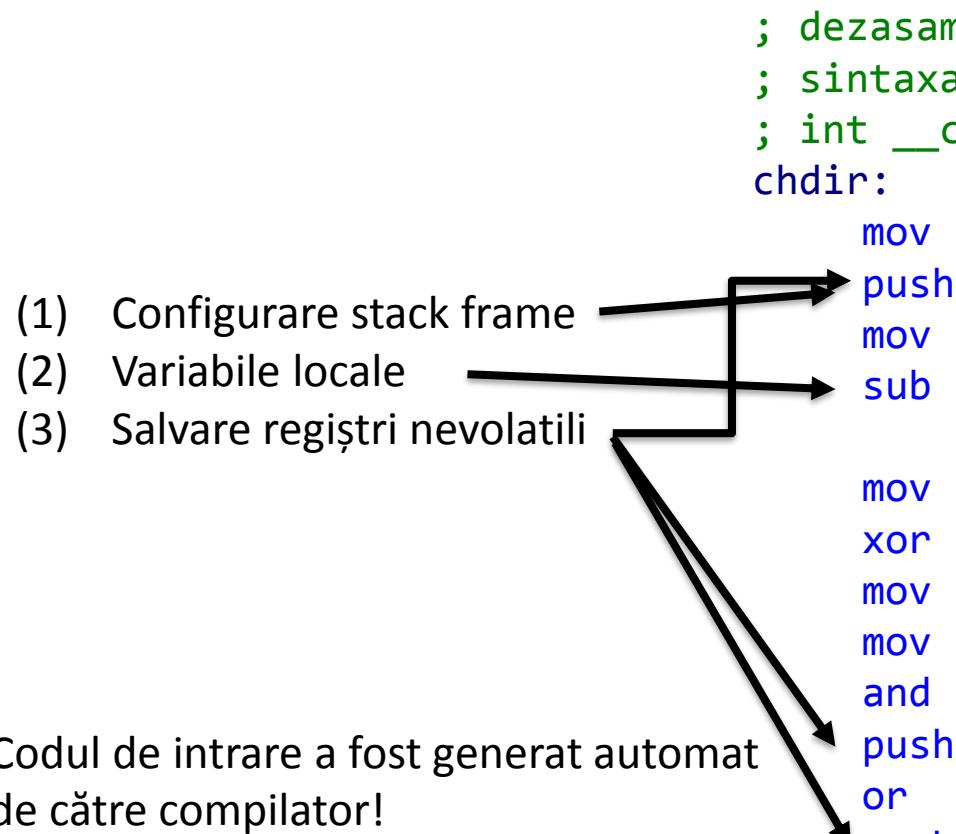
- Apelul subruteinilor – cod de intrare
 - Exemplu: cod intrare în funcția CDECL **chdir**, generată de compilatorul C

; dezasamblat cu IDA - The Interactive Disassembler
; sintaxa corespunde asamblorului MASM
; int __cdecl chdir(const char *)

```

chdir:
    mov     edi, edi ; inutil, dar permite modificare!
    push    ebp
    mov     ebp, esp
    sub    esp, 118h

    mov     eax, __security_cookie
    xor     eax, ebp
    mov     [ebp+var_4], eax
    mov     eax, [ebp+lpPathName]
    and     [ebp+var_110], 0
    push    ebx
    or      ebx, 0xFFFFFFFFh
    push    esi
;
```



(1) Configurare stack frame
(2) Variabile locale
(3) Salvare registri nevolatili

Codul de intrare a fost generat automat de către compilator!

Interfațarea cu limbajele de nivel înalt

- Apelul subruteinelor – cod de intrare

- Exemplu: cod intrare într-o funcție STDCALL scrisă în asm de către noi

- (1) Configurare stack frame
- (2) Variabile locale
- (3) Salvare registri nevolatili

int factorial (int n)

```
{ if (n==1) return 1;
  else
    return n * factorial (n-1);
}
```

```
factorial://in modulul C apelant se declara - extern int _stdcall factorial(int n)

push ebp          ; definim un cadru de stiva cu EBP pivot/referinta
mov ebp, esp      ; alocam spatiu pe stiva pentru o variabila DWORD temporara
sub esp, 4         ; salvam ebx pentru a-l putea restaura la sfarsit
push ebx          ; incarcam valoarea argumentului din stackframe (n-ul curent)

mov eax, [ebp + 8] ; testam coditia de terminare (n < 2)
cmp eax, 2
jae .recursiv
mov eax, 1         ; returnam 1 cand n < 2
jmp .gata

.recursiv:
push eax          ; retinem valoarea lui n sa nu-l pierdem la apelul recursiv
dec eax            ; pregatim nou parametru n-1 pt a apela factorial(n-1)
push eax            ; urcam in stiva valoarea n-1 ca parametru pt factorial(n-1)
call factorial     ; apel recursiv (STDCALL)

mov [ebp - 4], eax ; salvam in DWORD-ul temporar valoarea lui factorial(n-1)
pop eax             ; restauram valoarea lui n
mov ebx, [ebp - 4]  ; ebx <- factorial(n-1), preluat din variabila temporara
mul ebx             ; calculam (edx:)eax = (EAX=)n * (EBX=)factorial(n-1)

.gata:
pop ebx             ; refacem EBX la valoarea initiala
add esp, 4           ; eliberam memoria ocupata de variabila temporara
pop ebp              ; restauram ebp la valoarea initiala
ret 4                ; STDCALL: revenire cu eliberare memorie parametri
```

Interfațarea cu limbajele de nivel înalt

- Apelul subruteinelor – cod de intrare
 - Exemplu: cod intrare într-o funcție STDCALL scrisă în asm – stackframe

```

factorial:
    push ebp
    mov ebp, esp
    sub esp, 4
    push ebx
    ; definim un cadru de stiva cu EBP pivot/referinta
    ; alocam spatiu pe stiva pentru o variabila DWORD temporara
    ; salvam ebx pentru a-l putea restaura

    mov eax, [ebp + 8] ; incarcam valoarea argumentului din stackframe

    cmp eax, 2 ; testam coditia de terminare (n < 2)
    jae .recursiv
    mov eax, 1 ; returnam 1 cand n < 2
    jmp .gata

.recursiv:
    push eax ; retinem valoarea lui n sa nu-l pierdem la apelul recursiv

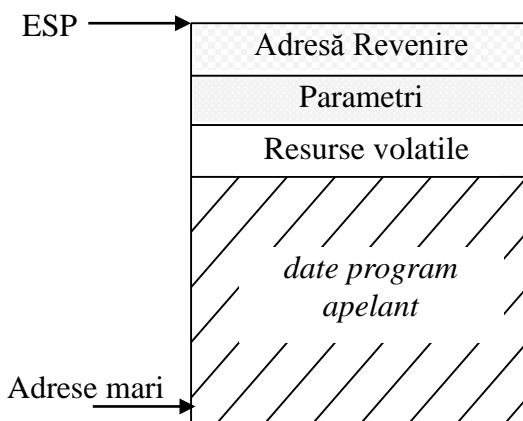
    dec eax
    push eax
    call factorial ; apelam factorial(n-1), am urcat in stiva valoarea n-1
    ; apel recursiv (STDCALL)

    mov [ebp - 4], eax ; salvam in DWORD-ul temporar valoarea lui factorial(n-1)
    pop eax ; restauram valoarea lui n
    mov ebx, [ebp - 4] ; ebx <- factorial(n-1), preluat din variabila temporara
    mul ebx ; calculam (edx:)eax = (EAX=)n * (EBX=)factorial(n-1)

.gata:
    pop ebx ; refacem EBX la valoarea initiala
    add esp, 4 ; eliberam memoria ocupata de variabila temporara
    pop ebp ; restauram ebp la valoarea initiala
    ret 4 ; STDCALL: revenire cu eliberare memorie parametri

```

Stare la intrare



Interfațarea cu limbajele de nivel înalt

- Apelul subruteinelor – cod de intrare

- Exemplu: cod intrare într-o funcție STDCALL scrisă în asm – stackframe

(1) Configurare stack frame →

```

factorial:
    push ebp
    mov ebp, esp           ; definim un cadru de stiva cu EBP pivot/referinta
    sub esp, 4             ; alocam spatiu pe stiva pentru o variabila DWORD temporara
    push ebx               ; salvam ebx pentru a-l putea restaura

    mov eax, [ebp + 8]     ; incarcam valoarea argumentului din stackframe

    cmp eax, 2             ; testam coditia de terminare (n < 2)
    jae .recursiv
    mov eax, 1             ; returnam 1 cand n < 2
    jmp .gata

.recursiv:
    push eax               ; retinem valoarea lui n sa nu-l pierdem la apelul recursiv

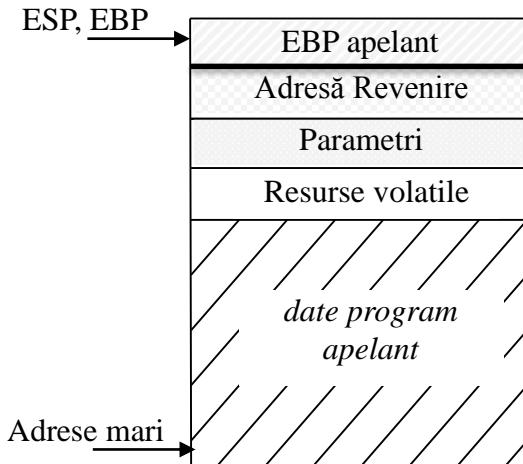
    dec eax
    push eax
    call factorial          ; apelam factorial(n-1), am urcat in stiva valoarea n-1
                            ; apel recursiv (STDCALL)

    mov [ebp - 4], eax      ; salvam in DWORD-ul temporar valoarea lui factorial(n-1)
    pop eax                 ; restauram valoarea lui n
    mov ebx, [ebp - 4]       ; ebx <- factorial(n-1), preluat din variabila temporara
    mul ebx                 ; calculam (edx:)eax = (EAX=)n * (EBX=)factorial(n-1)

.gata:
    pop ebx
    add esp, 4
    pop ebp
    ret 4                   ; refacem EBX la valoarea initiala
                            ; eliberam memoria ocupata de variabila temporara
                            ; restauram ebp la valoarea initiala
                            ; STDCALL: revenire cu eliberare memorie parametri

```

(1) Configurare stack frame



Interfațarea cu limbajele de nivel înalt

- Apelul subruteinelor – cod de intrare

- Exemplu: cod intrare într-o funcție STDCALL scrisă în asm – stackframe

- (1) Configurare stack frame →
 (2) Variabile locale →

```

factorial:
  push ebp
  mov ebp, esp
  sub esp, 4
  push ebx ←

  mov eax, [ebp + 8] ; incarcam valoarea argumentului din stackframe

  cmp eax, 2          ; testam coditia de terminare (n < 2)
  jae .recursiv
  mov eax, 1          ; returnam 1 cand n < 2
  jmp .gata

.recursiv:
  push eax            ; retinem valoarea lui n sa nu-l pierdem la apelul recursiv

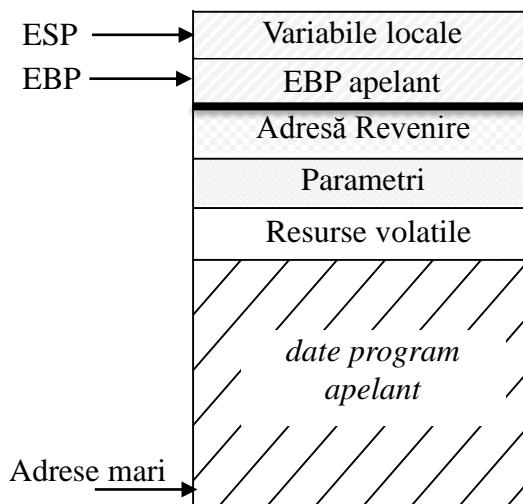
  dec eax
  push eax
  call factorial      ; apelam factorial(n-1), am urcat in stiva valoarea n-1
                       ; apel recursiv (STDCALL)

  mov [ebp - 4], eax
  pop eax
  mov ebx, [ebp - 4]
  mul ebx             ; salvam in DWORD-ul temporar valoarea lui factorial(n-1)
                       ; restauram valoarea lui n
                       ; ebx <- factorial(n-1), preluat din variabila temporara
                       ; calculam (edx:)eax = (EAX=)n * (EBX=)factorial(n-1)

.gata:
  pop ebx
  add esp, 4
  pop ebp
  ret 4               ; refacem EBX la valoarea initiala
                      ; eliberam memoria ocupata de variabila temporara
                      ; restauram ebp la valoarea initiala
                      ; STDCALL: revenire cu eliberare memorie parametri

```

(2) Variabile locale



Interfațarea cu limbajele de nivel înalt

- Apelul subruteinelor – cod de intrare

- Exemplu: cod intrare într-o funcție STDCALL scrisă în asm – stackframe

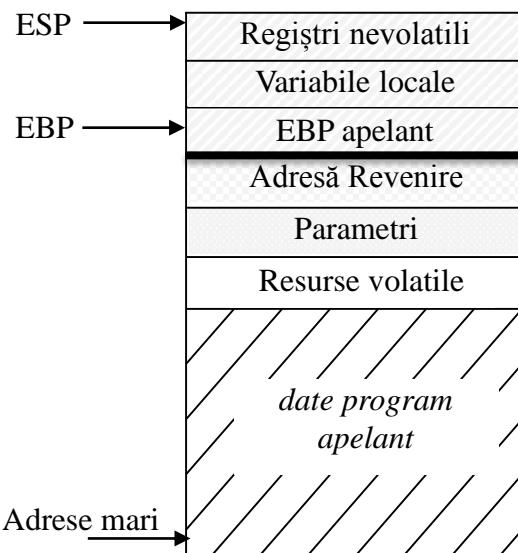
- (1) Configurare stack frame
- (2) Variabile locale
- (3) Salvare registri nevolatili

```
factorial:
    push ebp
    mov ebp, esp
    sub esp, 4
    push ebx
```

; definim un cadru de stiva cu EBP pivot/referinta
; alocam spatiu pe stiva pentru o variabila DWORD temporara
; salvam ebx pentru a-l putea restaura

```
    mov eax, [ebp + 8] ; incarcam valoarea argumentului din stackframe
    cmp eax, 2          ; testam coditia de terminare (n < 2)
    jae .recursiv
    mov eax, 1          ; returnam 1 cand n < 2
    jmp .gata
```

(3) Salvare registri nevolatili



```
.recursiv:
    push eax          ; retinem valoarea lui n sa nu-l pierdem la apelul recursiv
    dec eax
    push eax
    call factorial   ; AICI SE GENEREAZA CODUL DE APEL PENTRU URMATORUL APEL RECURSIV !!
    mov [ebp - 4], eax ; apelam factorial(n-1), am urcat in stiva valoarea n-1
    pop eax
    mov ebx, [ebp - 4] ; apel recursiv (STDCALL)
    call factorial
    mov [ebp - 4], eax ; salvam in DWORD-ul temporar valoarea lui factorial(n-1)
    pop eax           ; restauram valoarea lui n
    mov ebx, [ebp - 4] ; ebx <- factorial(n-1), preluat din variabila temporara
    mul ebx           ; calculam (edx:)eax = (EAX=)n * (EBX=)factorial(n-1)

.gata:
    pop ebx
    add esp, 4
    pop ebp
    ret 4
```

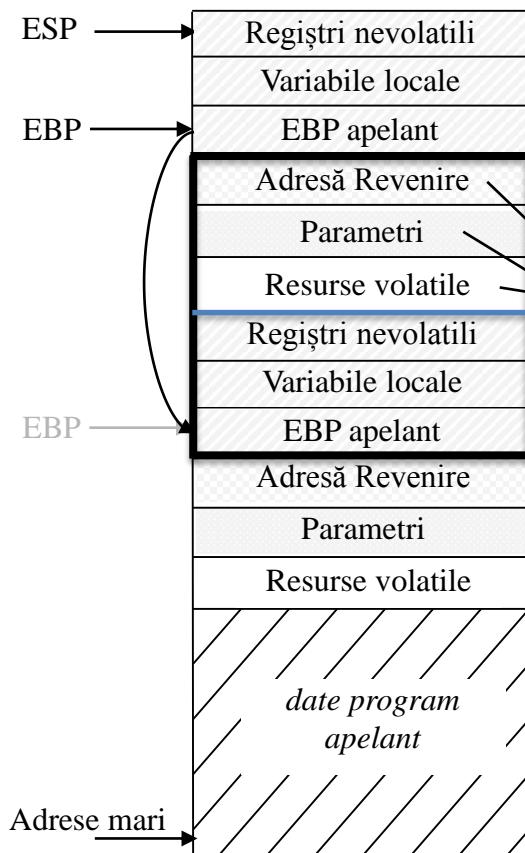
; refacem EBX la valoarea initiala
; eliberam memoria ocupata de variabila temporara
; restauram ebp la valoarea initiala
; STDCALL: revenire cu eliberare memorie parametri

Interfațarea cu limbajele de nivel înalt

- Apelul subruteinelor – cod de intrare

- Exemplu: cod intrare într-o funcție STDCALL scrisă în asm – stackframe

Evoluție stivă la apel recursiv



```

factorial:
    push ebp
    mov ebp, esp      ; definim un cadru de stiva cu EBP pivot/referinta
    sub esp, 4        ; alocam spatiu pe stiva pentru o variabila DWORD temporara
    push ebx          ; salvam ebx pentru a-l putea restaura

    mov eax, [ebp + 8] ; incarcam valoarea argumentului din stackframe

    cmp eax, 2        ; testam coditia de terminare (n < 2)
    jae .recursiv
    mov eax, 1        ; returnam 1 cand n < 2
    jmp .gata

.recursiv:
    push eax          ; retinem valoarea lui n sa nu-l pierdem la apelul recursiv
    dec eax
    push eax
    call factorial    ; AICI SE GENEREAZA CODUL DE APEL PENTRU URMATORUL APEL RECURSIV !!

    mov [ebp - 4], eax ; salvam in DWORD-ul temporar valoarea lui factorial(n-1)
    pop eax            ; restauram valoarea lui n
    mov ebx, [ebp - 4] ; ebx <- factorial(n-1), preluat din variabila temporara
    mul ebx            ; calculam (edx:)eax = (EAX=)n * (EBX=)factorial(n-1)

.gata:
    pop ebx            ; refacem EBX la valoarea initiala
    add esp, 4          ; eliberam memoria ocupata de variabila temporara
    pop ebp            ; restauram ebp la valoarea initiala
    ret 4              ; STDCALL: revenire cu eliberare memorie parametri

```

Interfațarea cu limbajele de nivel înalt

- Apelul subrutinelor – cod de ieșire

- Sarcini:

1. Restaurare resurse nevolatile alterate
2. Eliberarea variabilelor locale ale funcției
3. Dezafectarea cadrului de stivă
4. Revenirea din funcție și eliberarea argumentelor

- *CDECL:*

- Subrutina apelată: ret
 - Subprogramul apelant: add esp, dimensiune_argumente

- *STDCALL:*

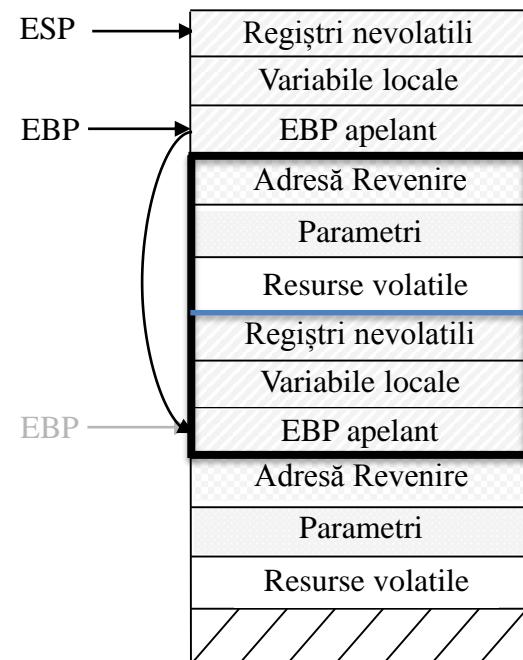
- *ret dimensiune_argumente*

- Exceptând resursele volatile și rezultatele directe ale funcției, **starea programului după acești pași trebuie să reflecte starea inițială, de dinainte de apel!**

Interfațarea cu limbajele de nivel înalt

- Apelul subruteinelor – cod de ieșire

- Exemplu: cod ieșire dintr-o funcție STDCALL asm (din apel recursiv) - stackframe



```

factorial:
    push ebp
    mov ebp, esp      ; definim un cadru de stiva cu EBP pivot/referinta
    sub esp, 4        ; alocam spatiu pe stiva pentru o variabila DWORD temporara
    push ebx          ; salvam ebx pentru a-l putea restaura

    mov eax, [ebp + 8] ; incarcam valoarea argumentului din stackframe

    cmp eax, 2         ; testam coditia de terminare (n < 2)
    jae .recursiv
    mov eax, 1         ; returnam 1 cand n < 2
    jmp .gata

.recursiv:
    push eax          ; retinem valoarea lui n sa nu-l pierdem la apelul recursiv

    dec eax
    push eax
    call factorial     ; apelam factorial(n-1), am urcat in stiva valoarea n-1
                        ; apel recursiv (STDCALL)

    mov [ebp - 4], eax ; salvam in DWORD-ul temporar valoarea lui factorial(n-1)
    pop eax            ; restauram valoarea lui n
    mov ebx, [ebp - 4] ; ebx <- factorial(n-1), preluat din variabila temporara
    mul ebx            ; calculam (edx:)eax = (EAX=)n * (EBX=)factorial(n-1)

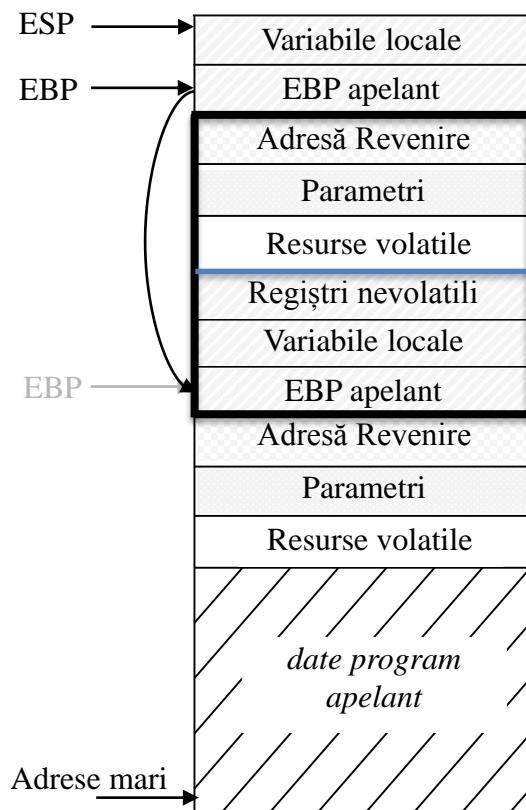
.gata:
    pop ebx            ; refacem EBX la valoarea initiala
    add esp, 4          ; eliberam memoria ocupata de variabila temporara
    pop ebp            ; restauram ebp la valoarea initiala
    ret 4               ; STDCALL: revenire cu eliberare memorie parametri

```

Interfațarea cu limbajele de nivel înalt

- Apelul subruteinelor – cod de ieșire

- Exemplu: cod ieșire dintr-o funcție STDCALL asm (din apel recursiv) - stackframe



(1) Regiștri nevolatili

```

factorial:
    push ebp
    mov ebp, esp      ; definim un cadru de stiva cu EBP pivot/referinta
    sub esp, 4        ; alocam spatiu pe stiva pentru o variabila DWORD temporara
    push ebx          ; salvam ebx pentru a-l putea restaura

    mov eax, [ebp + 8] ; incarcam valoarea argumentului din stackframe

    cmp eax, 2         ; testam coditia de terminare (n < 2)
    jae .recursiv
    mov eax, 1         ; returnam 1 cand n < 2
    jmp .gata

.recursiv:
    push eax          ; retinem valoarea lui n sa nu-l pierdem la apelul recursiv

    dec eax
    push eax
    call factorial     ; apelam factorial(n-1), am urcat in stiva valoarea n-1
                        ; apel recursiv (STDCALL)

    mov [ebp - 4], eax ; salvam in DWORD-ul temporar valoarea lui factorial(n-1)
    pop eax            ; restauram valoarea lui n
    mov ebx, [ebp - 4] ; ebx <- factorial(n-1), preluat din variabila temporara
    mul ebx            ; calculam (edx:)eax = (EAX=)n * (EBX=)factorial(n-1)

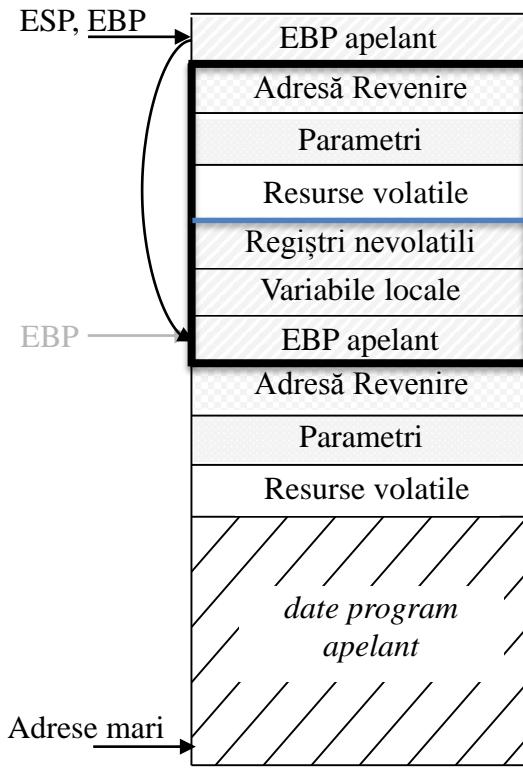
.gata:
    pop ebx            ; refacem EBX la valoarea initiala
    add esp, 4          ; eliberam memoria ocupata de variabila temporara
    pop ebp            ; restauram ebp la valoarea initiala
    ret 4               ; STDCALL: revenire cu eliberare memorie parametri

```

Interfațarea cu limbajele de nivel înalt

- Apelul subruteinelor – cod de ieșire

- Exemplu: cod ieșire dintr-o funcție STDCALL asm (din apel recursiv) - stackframe



```

factorial:
    push ebp
    mov ebp, esp      ; definim un cadru de stiva cu EBP pivot/referinta
    sub esp, 4        ; alocam spatiu pe stiva pentru o variabila DWORD temporara
    push ebx          ; salvam ebx pentru a-l putea restaura

    mov eax, [ebp + 8] ; incarcam valoarea argumentului din stackframe

    cmp eax, 2        ; testam coditia de terminare (n < 2)
    jae .recursiv
    mov eax, 1        ; returnam 1 cand n < 2
    jmp .gata

.recursiv:
    push eax          ; retinem valoarea lui n sa nu-l pierdem la apelul recursiv

    dec eax
    push eax
    call factorial     ; apelam factorial(n-1), am urcat in stiva valoarea n-1
                        ; apel recursiv (STDCALL)

    mov [ebp - 4], eax ; salvam in DWORD-ul temporar valoarea lui factorial(n-1)
    pop eax            ; restauram valoarea lui n
    mov ebx, [ebp - 4] ; ebx <- factorial(n-1), preluat din variabila temporara
    mul ebx            ; calculam (edx:)eax = (EAX=)n * (EBX=)factorial(n-1)

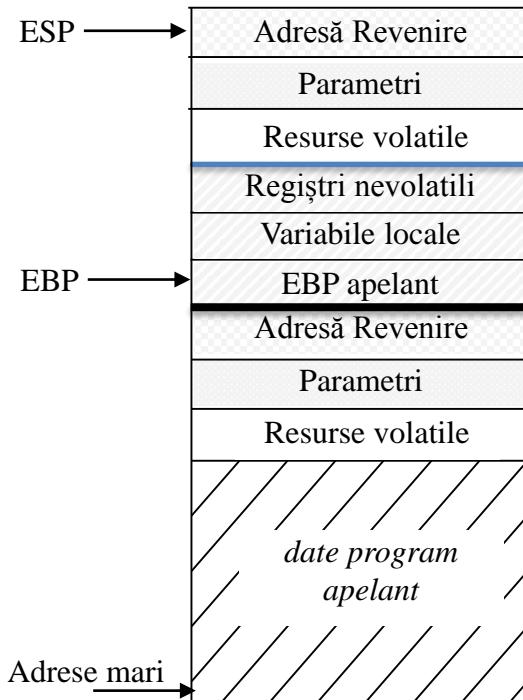
.gata:
    pop ebx            ; refacem EBX la valoarea initiala
    add esp, 4          ; eliberam memoria ocupata de variabila temporara
    pop ebp            ; restauram ebp la valoarea initiala
    ret 4              ; STDCALL: revenire cu eliberare memorie parametri

```

Interfațarea cu limbajele de nivel înalt

- Apelul subruteinelor – cod de ieșire

- Exemplu: cod ieșire dintr-o funcție STDCALL asm (din apel recursiv) - stackframe



```

factorial:
    push ebp
    mov ebp, esp      ; definim un cadru de stiva cu EBP pivot/referinta
    sub esp, 4        ; alocam spatiu pe stiva pentru o variabila DWORD temporara
    push ebx          ; salvam ebx pentru a-l putea restaura

    mov eax, [ebp + 8] ; incarcam valoarea argumentului din stackframe

    cmp eax, 2         ; testam coditia de terminare (n < 2)
    jae .recursiv
    mov eax, 1         ; returnam 1 cand n < 2
    jmp .gata

.recursiv:
    push eax          ; retinem valoarea lui n sa nu-l pierdem la apelul recursiv

    dec eax
    push eax
    call factorial     ; apelam factorial(n-1), am urcat in stiva valoarea n-1
                        ; apel recursiv (STDCALL)

    mov [ebp - 4], eax ; salvam in DWORD-ul temporar valoarea lui factorial(n-1)
    pop eax            ; restauram valoarea lui n
    mov ebx, [ebp - 4] ; ebx <- factorial(n-1), preluat din variabila temporara
    mul ebx            ; calculam (edx:)eax = (EAX=)n * (EBX=)factorial(n-1)

.gata:
    pop ebx            ; refacem EBX la valoarea initiala
    add esp, 4          ; eliberam memoria ocupata de variabila temporara
    pop ebp            ; restauram ebp la valoarea initiala
    ret 4              ; STDCALL: revenire cu eliberare memorie parametri

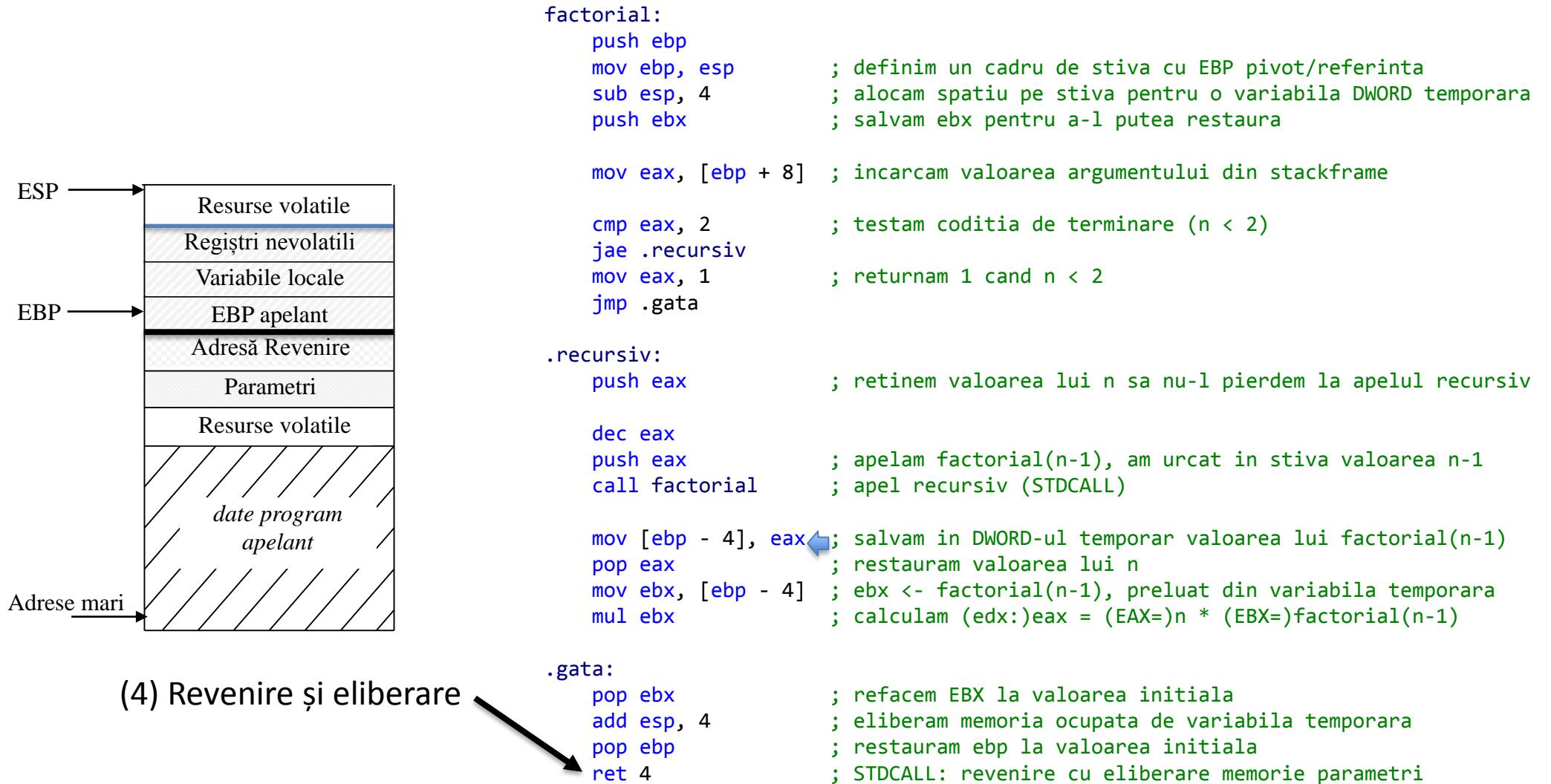
```

(3) Stackframe

Interfațarea cu limbajele de nivel înalt

- Apelul subruteinelor – cod de ieșire

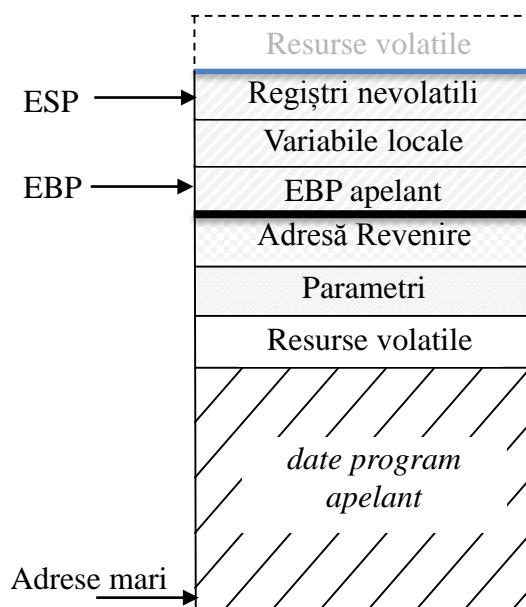
- Exemplu: cod ieșire dintr-o funcție STDCALL asm (din apel recursiv) - stackframe



Interfațarea cu limbajele de nivel înalt

- Apelul subruteinelor – cod de ieșire

- Exemplu: cod ieșire dintr-o funcție STDCALL asm (din apel initial) - stackframe



```

factorial:
    push ebp
    mov ebp, esp      ; definim un cadru de stiva cu EBP pivot/referinta
    sub esp, 4        ; alocam spatiu pe stiva pentru o variabila DWORD temporara
    push ebx          ; salvam ebx pentru a-l putea restaura

    mov eax, [ebp + 8] ; incarcam valoarea argumentului din stackframe

    cmp eax, 2        ; testam coditia de terminare (n < 2)
    jae .recursiv
    mov eax, 1        ; returnam 1 cand n < 2
    jmp .gata

.recursiv:
    push eax          ; retinem valoarea lui n sa nu-l pierdem la apelul recursiv

    dec eax
    push eax
    call factorial    ; apelam factorial(n-1), am urcat in stiva valoarea n-1
                       ; apel recursiv (STDCALL)

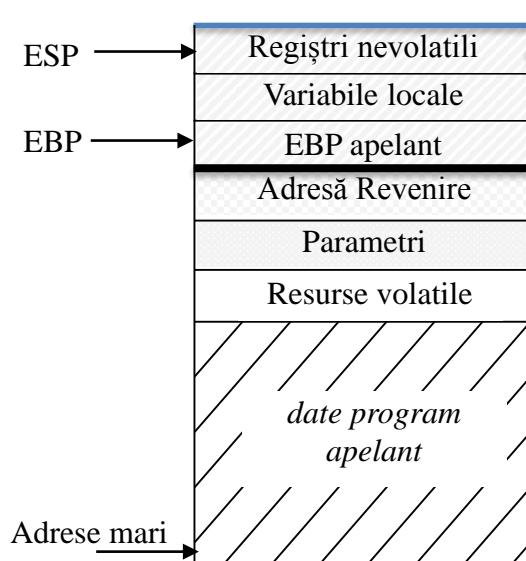
    mov [ebp - 4], eax ; salvam in DWORD-ul temporar valoarea lui factorial(n-1)
    pop eax            ; restauram valoarea lui n
    mov ebx, [ebp - 4] ; ebx <- factorial(n-1), preluat din variabila temporara
    mul ebx            ; calculam (edx:)eax = (EAX=)n * (EBX=)factorial(n-1)

.gata:
    pop ebx            ; refacem EBX la valoarea initiala
    add esp, 4          ; eliberam memoria ocupata de variabila temporara
    pop ebp            ; restauram ebp la valoarea initiala
    ret 4              ; STDCALL: revenire cu eliberare memorie parametri
  
```

Interfațarea cu limbajele de nivel înalt

- Apelul subruteinelor – cod de ieșire

- Exemplu: cod ieșire dintr-o funcție STDCALL asm (din apel inițial) - stackframe



```

factorial:
    push ebp
    mov ebp, esp      ; definim un cadru de stiva cu EBP pivot/referinta
    sub esp, 4        ; alocam spatiu pe stiva pentru o variabila DWORD temporara
    push ebx          ; salvam ebx pentru a-l putea restaura

    mov eax, [ebp + 8] ; incarcam valoarea argumentului din stackframe

    cmp eax, 2        ; testam coditia de terminare (n < 2)
    jae .recursiv
    mov eax, 1        ; returnam 1 cand n < 2
    jmp .gata

.recursiv:
    push eax          ; retinem valoarea lui n sa nu-l pierdem la apelul recursiv

    dec eax
    push eax
    call factorial    ; apelam factorial(n-1), am urcat in stiva valoarea n-1
                      ; apel recursiv (STDCALL)

    mov [ebp - 4], eax ; salvam in DWORD-ul temporar valoarea lui factorial(n-1)
    pop eax            ; restauram valoarea lui n
    mov ebx, [ebp - 4] ; ebx <- factorial(n-1), preluat din variabila temporara
    mul ebx            ; calculam (edx:)eax = (EAX=)n * (EBX=)factorial(n-1)

.gata:
    pop ebx            ; refacem EBX la valoarea initiala
    add esp, 4          ; eliberam memoria ocupata de variabila temporara
    pop ebp            ; restauram ebp la valoarea initiala
    ret 4              ; STDCALL: revenire cu eliberare memorie parametri
  
```

Interfațarea cu limbajele de nivel înalt

- Apelul subruteinelor – cod de ieșire

- Exemplu: cod ieșire dintr-o funcție STDCALL asm (din apel inițial) - stackframe

```

factorial:
    push ebp
    mov ebp, esp      ; definim un cadru de stiva cu EBP pivot/referinta
    sub esp, 4        ; alocam spatiu pe stiva pentru o variabila DWORD temporara
    push ebx          ; salvam ebx pentru a-l putea restaura

    mov eax, [ebp + 8] ; incarcam valoarea argumentului din stackframe

    cmp eax, 2        ; testam coditia de terminare (n < 2)
    jae .recursiv
    mov eax, 1        ; returnam 1 cand n < 2
    jmp .gata

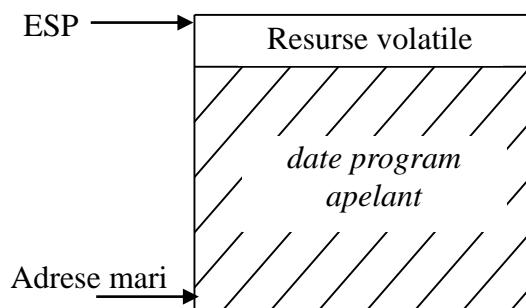
.recursiv:
    push eax          ; retinem valoarea lui n sa nu-l pierdem la apelul recursiv

    dec eax
    push eax
    call factorial    ; apelam factorial(n-1), am urcat in stiva valoarea n-1
                       ; apel recursiv (STDCALL)

    mov [ebp - 4], eax ; salvam in DWORD-ul temporar valoarea lui factorial(n-1)
    pop eax            ; restauram valoarea lui n
    mov ebx, [ebp - 4] ; ebx <- factorial(n-1), preluat din variabila temporara
    mul ebx            ; calculam (edx:)eax = (EAX=)n * (EBX=)factorial(n-1)

.gata:
    pop ebx            ; refacem EBX la valoarea initiala
    add esp, 4          ; eliberam memoria ocupata de variabila temporara
    pop ebp            ; restauram ebp la valoarea initiala
    ret 4              ; STDCALL: revenire cu eliberare memorie parametri

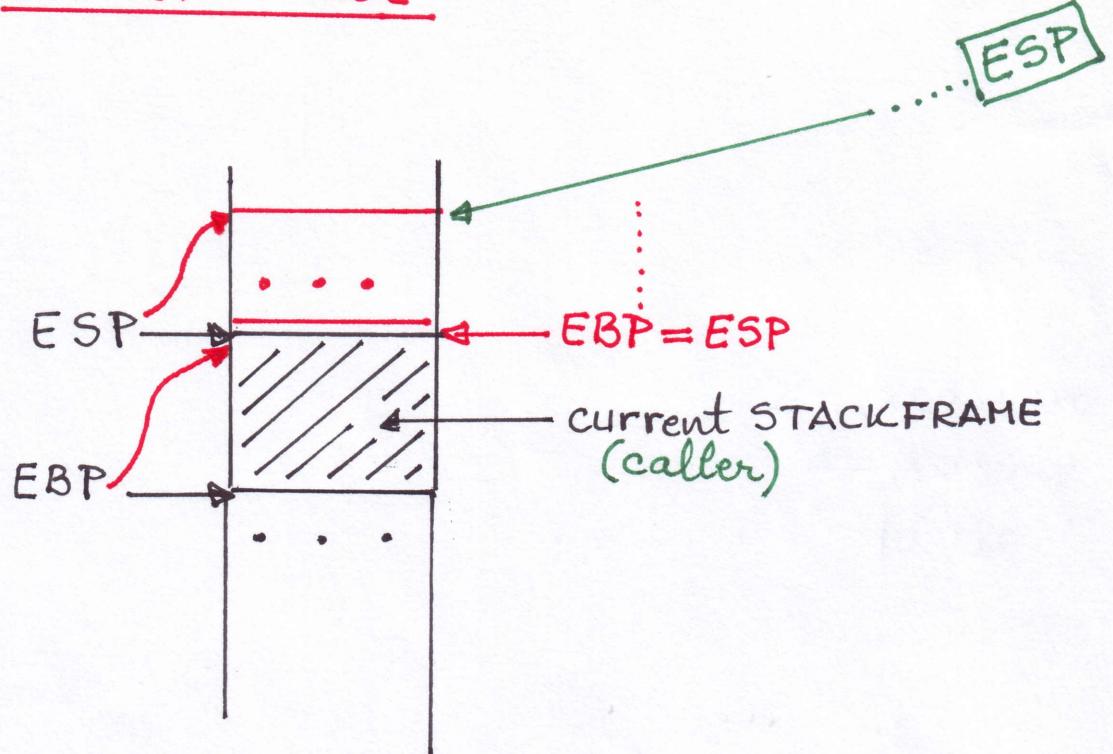
```





Bitdefender®

ENTRY PHASE



- involves the creation of a NEW STACKFRAME for the CALLED subroutine:

push EBP ; for restoring the base of the CURRENT STACKFRAME when returning

MOV EBP,ESP ; This is the BIRTH of the NEW STACKFRAME (sizeof initial = 0)
... - - - - [ESP] will start to "grow" by currently performed PUSHES

Cod apel (APELATORUL):

- a). Salvare resurse volatile (EAX, ECX, EDX, EFLAGS)
- b). Transmitere parametri
- c). Efectuare apel cu salvare adresa de revenire

Cod de intrare (APELATUL):

- a). Creare stackframe nou PUSH EBP,
 MOV EBP, ESP
- b). Alocare spațiu variabile locale SUB ESP,nr_octeti
- c). Salvare resurse nevolatile posibil a fi modificate

Cod de ieșire (APELATUL):

- a). Restaurare registri nevolatili
- b). Eliberare spațiu variabile locale [ADD ESP, nr_octeti_var_locale] – menționată aici doar ca revers al pct b) din codul de intrare , dar neobligatorie deoarece eliberarea cadrului de stivă (mov esp, ebp) include oricum dpdv practic si această etapă.
- c). Eliberare cadru de stivă MOV ESP, EBP (daca stim exact dimensiunea cadrului de stiva, ADD ESP, sizeof(stackframe) rezolva in mod similar...)
 POP EBP
 (a, b c – reversul codului de intrare)
- d). Revenirea din subprogram (RET) și scoaterea de pe stivă a parametrilor (daca este de tip STDCALL) - (reversul b + c din codul de apel)

A mai rămas de efectuat reversul pct a) din codul de apel. Este sarcina APELATORULUI să o facă alături de eventuala scoatere de pe stivă a parametrilor (daca este de tip CDECL). Aceste actiuni trebuie să se efectueze de catre codul apelant imediat după încheierea apelului.

Resurse volatile vs. resurse nevolatile

- resursele **volatile** sunt reprezentate de catre acei registri pe care **conventia de apel ii defineste ca apartinand subrutinei apelate**, ca atare apelantul avand datoria **ca parte a codului de apel** sa salveze valoarea acestora (daca-i foloseste) iar apoi sa le restaureze la valorile vechi la iesirea din apel. Deci: cine salveaza resursele volatile ? **Apelantul** (ca parte a codului de apel). Cine restaureaza resursele volatile ? Tot **apelantul**, dar NU ca parte a vreunui cod de apel/intrare/iesire, ci doar le restaureaza dupa apel in cadrul codului curent...

- resursele **nevolatile** sunt orice adrese de memorie ori registri care nu ii apartin explicit subrutinei chemate, iar ca urmare, daca aceasta doreste sa efectueze modificari este necesar sa le salveze la intrare (ca parte a codului de intrare) iar apoi sa le restaureze inainte de iesire la valorile lor originale (ca parte a codului de ieșire).

Diferenta apare in privinta cui are obligatia sa faca salvarea si resturarea:

- resurse **volatile**: **apelantul** are datoria sa aiba grija de valorile registrilor in cauza, daca-i foloseste (subrutina chemata nu garanteaza nimic)
- resurse **nevolatile**: apelantul nu are nicio datorie si ii este garantat ca valorile se pastreaza -- **codul apelat** este responsabil sa restaureze la sfarsit orice modifica

Responsabilitati generare cod apel – cod intrare – cod iesire

		Apel fctie/proc	{	}
Apelant	Apelat	Cod apel	Cod intrare	Cod iesire
C	C	Compilatorul C	Compilatorul C	Compilatorul C
C	asm	Compilatorul C	Programator asm	Programator asm
asm	C	Programator asm	Compilatorul C	Compilatorul C
asm	asm	Call (salvarea adresei de revenire)	NIMIC (totul ramane la latitudinea programatorului)	RET (recuperarea adresei de revenire + salt)