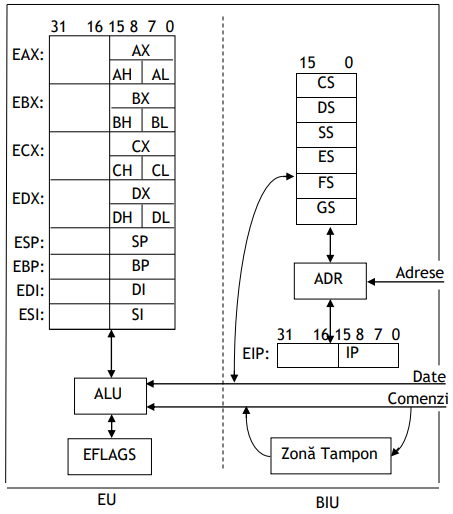
**Capitolul 2**

**1. Structura microprocesorului**

Microprocesorul x86 este format din 2 componente principale:  
 - **EU** (***Executive Unit***) – execută instrucțiunile mașină prin intermediul **ALU** (***Arithmetic and Logic Unit***)  
 - **BIU** (***Bus Interface Unit***) – pregătește execuția fiecărei instrucțiuni mașină. Citește o instrucțiune, o decodifică și calculează adresa din memorie a unui eventual operand. Configurația rezultată este depusă într-o zonă tampon cu dimensiunea de 15 octeți unde va fi preluată de **EU**.



**EU** și **BIU** lucrează în paralel, **EU** prelucrează instrucțiunea curentă, **BIU** pregătește instructiunea următoare. Ele sunt sincronizate, are termină prima așteaptă după cealalta.

**2. Regiștrii generali EU**

Registrul **EAX** – ***registrul acumulator***, folosit în majoritatea instrucțiunilor ca unul dintre operanzi.

Registrul **EBX** – ***registrul general*** (***Base Register***), fiind registrul de bază la programarea pe 16 biți.

Registrul **ECX** – ***registrul contor*** (***de numărare***), folosit pentru instrucțiunile care necesită indicații numerice.

Registrul **EDX** – ***registrul de date***, împreună cu **EAX** se folosește în calculele ale căror rezultate depășesc dublucuvântul (32 de biți).

Regiștrii **ESP** și **EBP** sunt 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 ultimul element introdus în stivă.

Registrul **EBP** – ***Base Pointer***, punctează spre primul element introdus în stivă.

Regiștrii **ESI** (***Source Index***) și **EDI** (***Destination Index***) – ***regiștrii de index***, utilizați pentru accesarea elementelor din șiruri de octeți sau cuvinte.

Fiecare dintre regiștrii **EAX**, **EBX**, **ECX**, **EDX**, **ESP**, **EBP**, **ESI** au capacitatea de 32 de biți. Fiecare dintre ei poate fi privit în același timp ca fiind format din concatenarea a doi subregiștrii de câte 16 biți. Subregistrul superior, care conține cei mai semnificativi 16 biți ai registrului de 32 de biți din care fac parte, nu are o denumire și nu este disponibil separat. Subregistrul inferior poate fi accesat individual, având astfel regiștrii de 16 biți **AX**, **BX**, **CX**, **DX**, **SP**, **BP**, **SI**. Dintre aceștia, **AX**, **BX**, **CX** și **DX** sunt fiecare la rândul lor, formați din câte doi alți subregiștrii 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**).

**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 de biți din care sunt folosiți uzual numai 9.

**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. **Flagul CF semnalează depășirea în cazul interpretării FĂRĂ SEMN**.

**PF** – ***Parity Flag***, valoarea lui se stabilește astfel încât împreună cu numărul de biți 1 din octetul cel mai puțin semnificativ al reprezentării rezultatului **UOE** să rezulte un număr impar de cifre 1. **Cu alte cuvinte, PF are valoarea 1 dacă octetul cel mai puțin semnificativ al rezultatului UOE are un număr par de cifre 1 și valoarea 0 în caz contrar**.

**AF** – ***Auxiliary Flag***, indică **valoarea transportului de la bitul 3 la bitul 4** al reprezentării **UOE**.

**ZF** – ***Zero Flag***, primește valoarea 1 dacă rezultatul **UOE** este **egal cu 0** și **valoarea 0** la **rezultatul diferit de 0**.

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

**IF** – ***Interrupt Flag***, este un flag de întrerupere.

**DF** – ***Direction Flag***, pentru operarea asupra șirurilor de octeți sau 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 sfărșit spre început**.

**OF** – ***Overflow Flag***, dacă rezultatul ultimei instrucțiuni în interpretarea **CU SEMN** a operanzilor **nu a încăput în spațiul rezervat operanzilor** (interval de reprezentare admisibil), atunci acest flag va avea **valoarea 1**, **altfel** va avea **valoarea 0**. **Flagul OF semnalează depășirea în cazul interpretării CU SEMN**.

**Categorii de flag-uri**

Flagurile se pot împărți în două categorii:  
a) cu efect anterior generat de către **Ultima Operație Efectuată** (**UOE**): **CF**, **PF**, **AF**, **ZF**, **SF** și **OF**.  
b) cu efect ulterior setării lor de către programator pentru **influențarea modului de operare al instrucțiunilor care urmează**: **CF**, **TF**, **DF** și **IF**.

**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** – setează valoarea **CF** la 0  
**STC** – setează valoarea **CF** la l  
**CMC** – complementarea valorii din **CF**  
**CLD** – setează valoarea **DF** la 0  
**STD** – setează valoarea **DF** la 1  
**CLI** – setează valoarea **IF** la 0  
**STI** – setează valoarea **IF** la 1

Pentru **TF** nu există instrucțiuni de acces direct deoarece are un rol absolut special în dezvoltarea de depanatoare.

**4. Complementul față de 2**

**Varianta 1 (oficială)**.e scade binar conținutul (evident binar) al locației de complement din 100..00, unde numărul de după cifra binară 1 are atâtea 0-uri câți biți are locația de complementat.

**Varianta 2 (derivată din definiția complementului față de 2)**. 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.

**Varianta 3 (cea mai rapidă pentru obținea configurației binare)**. Se lasă neschimbați biții începând din dreapta reprezentării binare până la primul bit 1 inclusiv, iar restul biților se inversează.

**Varianta 4 (cea mai rapidă pentru obținerea valoarea absolută în baza 10)**. Suma valorilor absolute a celor două valori complementare este cardinalul mulțimii reprezentabile pe acea dimensiune.  
 - pe 8 biți se pot reprezenta 2­8 valori: [0...+255] sau [-128...+127]  
 - pe 16 biți se pot reprezenta 216 valori: [0...65535] sau [-32768…+32767]  
 - pe 32 biți se pot reprezenta 232 valori: 4.294.967.296 valori (…)

**Două valori complementare nu vor fi niciodată parte a aceluiași interval de reprezentare admisibil!**

**Baza 2** → **Baza 10** ← **Baza 16**

(**reprezentare**) **INTERPRETARE** (**reprezentare**)

**a reprezentărilor in baza 2**

↙ ↘

**FĂRĂ SEMN** **CU SEMN**

(**valoare absolută**) (**pozitiv sau/și negativ**)

**5. Conceptul de depășire**

**Definiție**: O depășire este o condiție / situație matematică ce exprimă faptul că rezultatul unei operații nu a încăput în spațiul rezervat acestuia.

La nivelul procesorului și a limbajului de asamblare, o depășire este o **condiție / situație matematică ce exprimă faptul că rezultatul UOE nu a încăput în spațiul rezervat acestuia** SAU **acest rezultat nu aparține intervalului de reprezentare admisibil** 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**.

Flagurile ce semnalează o depășire sunt **CF** și **OF**. **CF** este folosit pentru semnalarea unei depășiri în cadrul interpretării **FĂRĂ SEMN**, iar **OF** este folosit pentru semnalarea unei depășiri în cadrul interpretării **CU SEMN**.

Prin setarea ambelor flag-uri CF și OF la valoarea 1, procesorul ne ”transmite mesajul” că ambele interpretări în baza 10 ale operației binare sunt **operații matematice incorecte**.

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

Semantic, cele două situații exprimă imposibilitatea acceptării matematice a celor 2 operații, nu putem aduna două numere pozitiv și să obținem unul negativ și nici nu putem aduna două numere negative și să obținem unul pozitiv.

0.......+ sau 1.......+  
0....... 1.......  
1....... 0.......

**În cazul scăderii**, avem de asemenea două reguli de depășire **în interpretarea cu semn**, consecința celor două reguli de la depășirea în cazul adunării:

1......– sau 0.......–  
0....... 1.......  
0....... 1.......

Semantic, cele două situații exprimă imposibilitatea acceptării matematice a celor 2 operații, nu putem scădea un număr pozitiv dintr-un număr negativ și să obținem unul pozitiv și nici invers.

**CF** va fi setat la **valoarea 1** (**unsigned overflow**) dacă **în urma operației de adunare** va **rezulta un transport de cifră semnificativă** (rezultatul **nu va încăpea în rezultatul alocat acestuia**) sau **dacă în urma operației de scădere** avem **nevoie de împrumut de cifră semnificativă** pentru efectuarea scăderii (**consecință a depășirii în cazul adunării**).

Operația de înmulțire **NU furnizează depășire** la nivelul arhitecturii 80x86, spațiul rezervat pentru rezultat fiind suficient pentru ambele interpretări. Totuși, pentru a nu rămîne 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 ambele flag-uri **vor fi setate la valoarea 1** (**“multiplication overflow”, CF = OF = 1**).

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 împărțirea word / byte, word pentru împărțirea doubleword / word și respectiv doubleword pentru împărțirea 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’**. În 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 niciun sens pentru nimeni să se întrebe ce valoare au flag-urile CF și OF.

**6. Regiștrii de adresă și calcul de adresă**

**Adresa unei locații** = numărul 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 de biți.

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 de 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 către sistemul de operare!** În scrierea hexazecimală o adresă se exprimă sub forma:

**s3s2s1s0 : o7o6o5o4o3o2o1o0**

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 processor în urma aplicării mecanismului de segmentare.

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

**o7o6o5o4o3o2o1o0 ≤ l7l6l5l4l3l2l1l0**

Determinarea adresei de segmentare din specifiacarea de adresă se face printr-un calcul de adresă conform formulei:

**a7a6a5a4a3a2a1a0 := b7b6b5b4b3b2b1b0 + o7o6o5o4o3o2o1o0**

unde a7a6a5a4a3a2a1a0 este **adresa calculată** (***scrisă 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 că 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. Extrage **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**.

Majoritatea sistemelor de operare moderne folosesc un mod particular de utilizare a segmentării, segmenetele începând la adresa 0 și având dimensiunea maximă posibilă de 4GiB (b7b6b5b4b3b2b1b0 = 00000000 și a7a6a5a4a3a2a1a0 := o7o6o5o4o3o2o1o0), acest mod poartă numele de **model de memorie flat**.

Procesoarele x86 suportă un mechanism de control al accesului la memorie numit **paginare**, independent de adresa segmentată. Paginarea implică împărțirea memoriei virtuale în pagini, care sunt associate (translatate) memoriei fizice disponibile (1 page = 4KB = 212 bytes = 4096 bytes).

Configurarea și controlul mecanismelor de segmentare și paginare sunt sarcina sistemului de operare. Dintre cele două, doar segmentarea intervene 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 memorie limitată la 1MB);  
- ***mod protejat*, pe 16 sau 32 biți, caracterizat prin folosirea paginării și segmentării;**  
- *mod virtual 8086*, permite rularea 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 **mod 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, corespunzători 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 subregistrul **IP**) conține offsetul instrucțiunii curente în cadrul segmentului de cod curent, el fiind manipulat exclusiv de către **BIU**.

|  |  |  |
| --- | --- | --- |
| Noțiune | Reprezentare | Descriere |
| Specificare de adresă, adresă logică, adresă FAR | Selector16:offset32 | 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 table de descriptori. |
| Offset, adresă NEAR | Offset32 | Definește doar componenta de offset (considerând segmentul cunoscut ori folosirea modelului de memorie flat) |
| Adresă liniară (adresă de segmentare) | 32 biți | Început segment + offset, reprezintă rezultatul calcului de adresă |
| 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) |

**7. 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, operanzii asupra cărora va fi aplicată, precum și modificatorii suplimentari care controlează modulul în care aceasta va fi executat.

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-uun registru al EU, fie este o constantă întreagă. Astfel, o instrucțiune are forma:

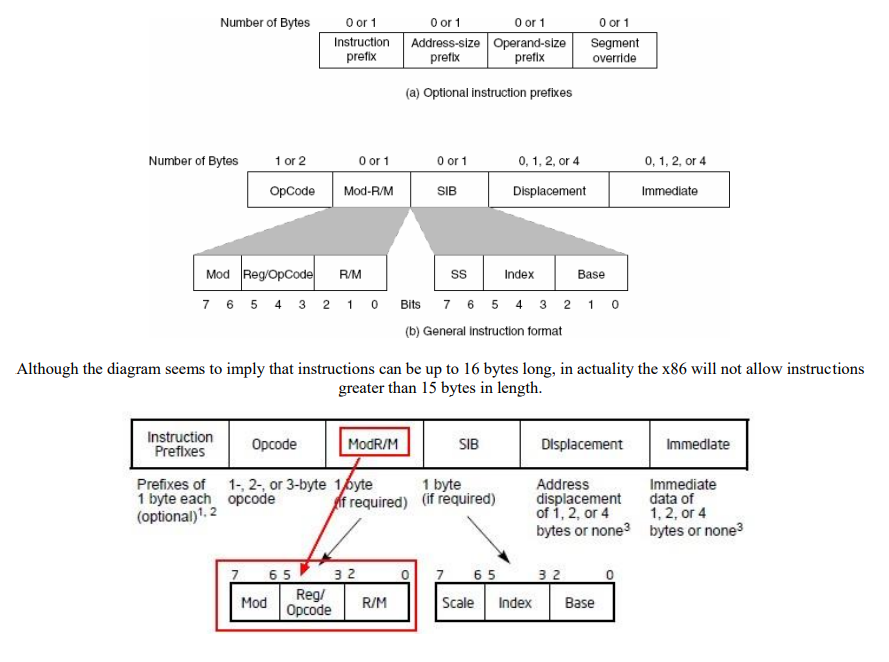
**numeinstrucțiune** destinație, sursă

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 și ocupă câte un octet fiecare. De exemplu, acestea pot solicita execuția repetată (în buclă) a instrucșiunii curente sau pot bloca magistrala de adrese pe parcursul execuției pentru a nu permite accesul concurent 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.



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.

Pentru cazuri mai complexe de adresare decât codificabile direct prin ModR/M, combinarea acestuia cu octetul SIB (Scale – Index – Base) permite următoarea **formulă generală de definire a unui offset**:

**offset = [bază] + [index ∙ scală] + [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 permiși 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.

**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 dublucuvânt (32 biți).

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**.

**8. Adrese FAR și NEAR**

Pentru a adresa o locație din memorie 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 segmetului din **unul dintre regiștrii din 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 dintre 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 COMPLETĂ DE ADRESĂ** și ea se poate exprima în trei moduri:  
- s3s2s1s0 : specificare\_offset, unde s3s2s1s0 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 variabilă 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, partea cea mai puțin semnificativă are adresa cea mai mică, iar partea cea mai semnificativă are adresa cea mai mare.

**9. Calculul offsetului unui operand. Moduri de adresare**

În cadrul unei instrucțiuni există 3 moduri de a specifica un operant pe care 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:

***adresa\_offset*** = [ bază ] + [ index ∙ scală ] + [ constantă ]

Deci ***adresa\_offset*** se obține din următoarele (maxim) patru elemente:  
- conținutul unuia dintre regiștrii EAX, EBX, ECX, EDX, EBP, ESI, EDI sau ESP ca bază;  
- conținutul unuia dintre regiștrii EAX, EBX, ECX, EDX, EBP, ESI sau EDI drept index;  
- factor numeric (scală) pentru a înmulți valoarea registrului index cu 1, 2, 4 sau 8  
- valoarea unei constante numerice pe octet, cuvânt sau dublucuvânt

De aici rezultă următoarele moduri de adresare la memorie:  
- directă, atunci când apare numai constanta;  
- bazată, dacă în calcul apare unul dintre regiștrii bază;  
- scalat-indexată, dacă în calcul apare unul dintre regiștrii index;

Adresarea care NU este directă se numește **adresare indirectă** (bazată și/sau indexată). Deci o adresare indirectă este cea pentru 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 releative apropiate (NEAD Address), pe dublucuvânt cu valor între -2147483648 și 2147483647.

Jmp MaiJos ;aceasta instrucțiune se traduce de obicei în Jmp [0084]↓  
 ……………….  
 ……………….  
 MaiJos:  
 Mov eax, ebx

**Capitolul 3**

**1. Elementele de bază 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 (nume de instrucțiuni sau directive) + etichete**.

Elementele cu care lucrează un **asamblor** sunt:  
 **\* etichete** – nume scrise de utilizator, cu ajutorul cărora se pot referi date (nume variabile, de obicei segmentul de date, etichete de date) sau zone de memorie (destinatie a unui salt, de obicei in segmentul de cod, etichete de cod)  
 **\* instrucțiuni** – scrise sub forma unor mnemonice care sugerează acțiunea. Asamblorul generează octeții care codifică instrucțiunea respectivă  
 **\* directive** – sunt indicații date asamblorului în scopul generării corecte a octeților. 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 ‘$’.

**2. Formatul unei linii sursă**

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 construită o **etichetă** sunt următoarele:  
- Litere, atât A-Z cât și a-z;  
- Cifre, de la 0 la 9  
- Caracterele \_, $, $$, #, @, ~, . și ?

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

Aceste reguli sunt valabile pentru toți ***identifiactorii*** valizi (denumiri simbolice, precum nume de variabile, etichete, macro, etc).

Identificatorii NASM sunt **case sensitive**, limbajul diferențiind literele mari de cele mici privitor la denumirile utilizator. Aceasta înseamnă că un identificator Abc este diferit de un identificator ABC. Pentru denumirile implicit parte a limbajului, cum ar fi cuvintele cheie, mnemonicile și numele regiștrilor, nu se diferențiează 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 ununi 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 segment de cod!**

**Valoarea unei etichete** în limbaj de ansamblare este un număr întreg reprezentând **adresa (FAR) 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 varibabilei**, 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) – echivalent cu mov EAX, v

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**.

**Operanzii** sunt parametri care definesc **valorile ce vor fi preluate de instrucțiuni sau de directive**. Ei pot fi **regiștri**, **constante**, **etichete**, **expresii**, **cuvinte cheie** sau **alte simboluri**. Semnificația operanzilor depinde de mnemonica instrucțiunii sau directivei asociate.

**3. 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).

**3.1. Moduri de invocare, evocare, accesare, specificare a operanzilor**

Operanzii instrucțiunilor pot fi specificați sub forme numite **moduri de accesare** / **invocare** / **evocare** / **specificare**.

Cele trei tipuri de operanzi sunt: operanzi imediați, operanzi registru și operanzi în memorie.

Valoarea operanzilor este calculată **în momentul asamblării** pentru operanzii imediați și pentru operanzii cu adresare directă (e vorba de offset aici!), **în momentul încărcării** programului pentru adresarea directă (partea de segment – adresa FAR – acest pas implică un proces de ajustare a adreselor numite REALOCARE a adreselor, address realocation – determinarea adresei fizice de început de segment pe baza selectorului de segment) și **în momentul execuției** (***run-time***) pentru operanzii registru și cei adresați indirect. ????:offset (assembly time) | 0708:offset (loading time)

**3.2 Utilizarea operanzilor imediați**

Operanzii 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: 0xb2a, 0xb2A, 0hb2a, 0b12Ah, 0B12AH, etc.  
 - valoarea zecimală 123: 123, 0d123, 0d0123, 123d, 123D, etc.  
 - valoarea binară 11001000: 11001000b, 0b11001000, 0y1100\_1000, 00\_1100\_1000Y

**Deplasamentele** (**offset-urile**) etichetelor de date și de cod reprezintă **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 exemplu:

mov eax, DS:[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.

Cu alte cuvinte, 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 de declarare în cadrul textului sursă și din dimensiunea de reprezentare dedusă pe bază informației de tip asociate.

**3.3. Utilizarea operanzilor registru**

Modul **de invocare / accesare directă**: **mov eax, ebx**  
 **Invocare / accesare indirectă** – pentru a indica locațiile de memorie – **mov eax, [ebx]**

**3.4 Utilizarea operanzilor din memorie**

Operanzi 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 procedori** (de ex: **call proc1**) sau **valoarea contorului de locații** (de ex: **b db $-a**).

**Deplasamentul unui operand cu adresare directă este calculat în momentul asamblări**i (***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 numindu-se **REALOCAREA ADRESELOR** = **Address Realocation**).

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 explicit un registru de segment. **Regulile limbajului de asamblare pentru asocierile implicite** sunt:  
 - **CS** pentru etichete de cod destinație ale unor salturi (jmp, call, ret, jz etc);  
 - **SS** în adresări SIB ce folosesc EBS 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 prefixare segment (notat “**:**” și care se mai numește ‘**operatorul de specificare a segmentului**’).

ES poate fi utilizat doar în specificări explicite de adresă (ex: ES:[Var] sau ES:[ebx+eax\*2-a]) sau în cazul unor instrucțiuni specific pe siruri (MOVSB).

JMP FAR CS:…  
 JMP FAR DS:… sau JMP FAR [label2]

**3.5. 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 indirectă 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**:

**[ registru\_de\_bază ] + [ registru\_index ∙ scală ] + [ constantă ]**

**Constanta** este o expresie a cărei valoare este determinabilă la momentul asamblării. De exemplu, [ebx + edi + 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 următoarei instrucțiuni:

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 formelor 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 eax \* 2)

În acest caz, asamblorul poate decide între codificarea 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 unuia dintre elementele componente ale formulei de calcul a offsetului.

**4. 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 (cu excepția adunării și scăderii de constante la un pointer care furnizează o valoare de tip adresă și cu excepția formulei de calcul a offset-ului unui operand)**. Instrucțiunile efectuează calcule de 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 prezentați 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 effect (oferit pentru simetrie cu “-“): +X = X |
| 7 | ~ | Unar, prefixat | Complement față de 1: mov AL, ~0 => 0xFF |
| 7 | ! | Unar, prefixat | Negare logică: !X = 0 când X ≠ 0, altfel 1 |
| 6 | \* | Binar, infix | Înmulțire: 1 \* 2 \* 3 = 6 |
| 6 | / | Binar, infix | Câtul împărțirii FĂRĂ SEMN: 24 / 4 / 2 = 3 |
| 6 | // | Binar, infix | Câtul împărțirii CU SEMN: -24 / 4 / 2 = -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 | << | Binar, infix | Deplasare biți către stânga: 1 << 4 = 16 |
| 4 | >> | Binar, infix | Deplasare biți către dreapta: 0xFE >> 4 = 0x0F |
| 3 | & | Binar, infix | ȘI: 0xF00F & 0x0FF6 = 0x0006 |
| 2 | ^ | Binar, infix | XOR (sau exclusiv): 0xFF0F ^ 0xF0FF = 0x0FF0 |
| 1 | | | Binar, infix | SAU: 1 | 2 = 3 |

**Operatorul de indexare** are o utilizare largă în specificarea operanzilor din memorie adresați indirect. Paragraful anterior a clarificat rolul operatorului [] în adresare indirectă.

**4.1 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 00000000 00001110 !!!

**4.2 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** | **SEMNIFICAȚIE** |
| ~ | ~ expresie | complementare biți |
| & | expr1 & expr2 | ȘI bit cu bit |
| | | expr1 | expr2 | SAU bit cu bit |
| ^ | expr1 ^ expr2 | XOR bit cu bit |

Exemple (presupunem că expresia se reprezintă pe un octet):  
~ 11110000b ; desemnează valoarea 00001111b, ~0f0h = 0fh  
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

**4.3. 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 relative la SS // **[es:082h]** ; deplasamentul e relative la ES  
**10h:var** ; segmental este indicat de selectorul 10h, iar offsetul este valoarea etichetei var

**4.4 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 (distructiv) valoarea în sensul precizat de conversia dorită. De aceea, aceștia sunt considerați **operatori de conversie (temporară) nedistructivă**. 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 întotdeauna doar rol de a clarifica o ambiguitate** (inclusiv când 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**).sd

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

Specificatorul **QWORD** nu intervine 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 – este inconsistență în NASM, asamblorul va face push DWORD 15

Exemple de operanzi IMPLICIȚI efectiv pe 64 de biți (în cod pe 32):  
 mul dword [v]; înmulțește EAX cu dword-ul de la adresa v si depune în EDX:EAX rezultatul  
 div dword [v] ; împarte EDX:EAX la dword-ul de la adresa v

**5. Directive**

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

**5.1. Directiva SEGMENT**

Directiva SEGMENT permite direcționarea octeților 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 sunt **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**) – segmental va conține cod, conținutul nu poate fi scris dar se poate citi  
 sau executa  
- **data** (sau **bss**) – segment de date permițând citire și scriere însă nu și execuție (valoare  
 implicită)  
- **rdata** – segment din care se poate doar citi, menit a conține definiții de date constante

Argumentul optional **aliniere** specifică multiplul numărului de octeți la care trebuie să înceapă segmentul respectiv. Alinierile acceptate sunt puteri ale 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 fo combinate cu segmentul în cauză la momentul editării legăturii. 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 ‘**clasă**’ are rolul de a permite stabilirea ordinii în ordinea în care editorul de legături plasează segmentele în memorie. Toate segmentele având aceeasi clasă vor fi plasate într-un bloc continuu de memorie indifferent de ordinea lor în cadrul codului sursă. Nu există o valoare implicită de inițializare 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**

**5.2. Directive pentru definirea datelor**

**definire date** = **declarare** (specificarea atributelor)  
(process unic) (poate apărea de oricâte ori e necesar)  
 +  
 **alocare** (rezervarea spațiului de memorie necesar)  
 (proces unic)

**TD** = cel mai important atribut al unei variabile  
**Variabila** = (nume, set\_de\_atribute, referință/adresă, valoare)  
**Set de atribute** = (tip de dată, domeniu de vizibilitate = scope, durata de viață = extent,  
 clasă de memorie = auto, register, static, extern – clase de memorie explicite în C)

**tipul de dată** = **dimensiunea de reprezantare** – octet, cuvânt, dublucuvânt, quadword

Forma generală a unei linii sursă în cazul unei declarații de date este:  
[*nume*] **tip\_data** lista\_expresii [;*comentariu*]  
[*nume*] **tip\_alocare** factor [;*comentariu*]  
[*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 **directivă 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 următoare definește și inițializează 5 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 începutul 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 obișnuită (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, permițându-se astfel declararea și inițializarea de tablouri. De exemplu, declarația Tablou DW 1,2,3,4,5 creează un tablou de 5 întregi reprezentați pe cuvinte având valorile 1,2,3,4,5. Dacă valorile de după directive nu încap pe o singură linie se pot adăuga oricâte linii este necesar, linii ce vor contine numai directive 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 dublucuvând (**DWORD**)  
 **RESQ** – date de tip 8 octeți (**QWORD** – 64 biți)  
 **REST** – date de tip 10 octeți (**TWORD** – 80 biți)

**factor** este un numar care indică de câte ori se repeat tipul alocării precizate

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

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

**TIMES** *factor* *tip\_data* *expresie*

De exemplu, Tabchar **TIMES** 80 **DB** ‘a’  
 creează un tablou de 80 de octeți iniializați 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.

Directiva TIMES poate fi aplicată și pe instrucțiuni:

**TIMES** *factor* *instrucțiune*

**TIMES** 32 add **EAX**, **EDX** ; **EAX** = **EAX** + 32 \* **EDX**

**5.4. Directiva EQU**

**Directiva EQU** permite atribuirea, în faza de asamblare, unei valori numerice sau șir 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 echivalente prin directiva **EQU** cu constantele din limbajele de programe 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\_START EQU (TABLE\_OFFSET + 100h)

**Capitolul 4**

**1. Instrucțiuni ale limbajului de asamblare**

Forma generală a unui program NASM + scurt exemplu:

**global** start ; solicităm asamblorului să confere vizibilitate globală simbolului  
 (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 pe stivă  
 call [printf] ; printf este numele unei funcții (etichetă = adresă, trb [])  
   
 ; apel ExitProcess(0), 0 reprezentând “execuție cu succes”  
 push dword 0  
 call [ExitProcess]  
  
**segment** code **use32** class=**DATA** ; variabilele vor fi stocate în segmentul de date  
 (denumit data)  
 string: db “Salut din ASM!”, 0

**2. Manipularea datelor**

**2.1. Instrucțiuni de transfer de uz general**

|  |  |
| --- | --- |
| **MOV** d, s | <d> 🡨 <s> (b-b, w-w, d-d) |
| **PUSH** s | ESP = ESP – 4 și depunde <s> în stivă (s – DWORD) |
| **POP** d | Extrage elementul din stivă și îl depune în d (d – dublucuvânt) ESP = ESP + 4 |
| **XCHG** d, s | <d> 🡨🡪 <s> s, d – trebuie să 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 stivă EAX, ECX, EDX, EBX, ESP, EBP, ESI și EDI |
| **POPA** / **POPAD** | Extrage EDI, ESI, EBP, ESP, EBX, EDX, ECX și EAX de pe stivă |
| **PUSHF** / **PUSHFD** | Depune EFlags pe stivă |
| **POPF** / **POPFD** | Extrage vârful stivei și îl 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 regiștrii de segment**, **atunci sursa trebuie să fie unul dintre cei opt regiștrii generali de 16 biți ai UE sau o variabilă din memorie**. Încărcătorul de programe al sistemului de operare preinițializează în mod automat regiștrii 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

Operanzii 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 ⬄ **sub** esp, 4 ; pregătim (alocăm) spațiu pentru a stoca valoarea  
 **mov** [esp], eax ; stocăm valoarea în locația alocată  
 **pop** eax ⬄ **mov** eax, [esp] ; încărcăm în eax valoarea din vârful stivei  
 **add** esp, 4 ; eliberăm locația

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

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 cuvânt, din motive de compabilitate 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 procesor și asamblată cu succes de asamblor, nu este recomandată, putând cauza ceea ce poartă numele de **eroare de dezaliniere a stivei: stiva este corect aliniată dacă și numai dacă valoarea din registrul ESP este în permanență 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 registru. 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 șir 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 registrului segment)  
 - **registru\_segment:EBX**, dacă registrul segment este precizat explicit

**Efectul instrucțiunii XLAT** **este înlocuirea octetlului 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]>  
depunde conținutul celei de-a 7-a locații de memorie (de index 6) din **Tabela** în **AL**.

Un exemplu de secvență care translatează o valoare zecimală ‘numar’ coprinsă între 0 și 15 în cifra hexazecimală (codul ei ASCII) corespunzătoare:  
 segment data use32 class=data  
 . . .  
 TabHexa db ‘0123456789ABCDEF’  
 . . .  
 segment code use32 class=code  
 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 valorică numerică registru – string de tipărit**).

**2.2 Instrucțiunea de transfer al adreselor LEA**

|  |  |
| --- | --- |
| **LEA** ***reg\_general, conținutul unui operand din memorie*** | ***Reg\_general*** 🡨 ***offset(mem)*** |

Instrucțiunea **LEA** (**L**oad **E**ffective **A**ddress) 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 accepta 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**, instructiunea cu care are acelasi effect fiind incorectă sintactic deoarece expresia ebx + v – 6 nu este determinabilă în momentul asamblării.

Prin utilizarea directă a valorilor deplasamentelor ce rezultă în urma calculelor de adrese (în contrast cu folosirea memoriei indicate de către acestea), **LEA** se evidențiează prin versalitate 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**).

**2.3. Instrucțiuni asupra flagurilor**

Următoarele patru instrucțiuni sunt instrucțiuni de transfer al indicatorilor:

Instrucțiunea **LAHF** (**L**oad register **AH** from **F**lags) 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** (în sensul că instrucțiunea **LAHF** nu este ca însăși generatoare de efecte asupra unor flag-uri – ea doar transferă valorile flag-urilor și atât)

Instrucțiunea **SAHF** (**S**tore register **AH** into **F**lags) 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** – acționează asupra flagului de întrerupere (**IF**). Funcționează efectiv doar în programarea sub 16 biți, aici la programarea de 32 biți SO interzicând accesul la flag-ul de întreruperi.

**3. Instrucțiuni de conversie**(distructivă)

|  |  |
| --- | --- |
| **CBW** | Conversie octet conținut în AL la cuvânt în AX (extensie de semn) |
| **CWD** | Conversie cuvânt din AX la dublucuvânt în DX:AX (extensie de semn) |
| **CWDE** | Conversie cuvânt din AX la 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 **FĂRĂ SEMN** |
| **MOVSX d, s** | Încarcă în d (**REGISTRU!**) de dimensiune mai mare decât s (**REGISTRU SAU OPERAND DIN MEMORIE!**) conținutul lui s **CU SEMN** |

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** = 0xFF  
 **cbw** ; **AX** = 0xFFFF

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** = 0xD8F0  
 **cwd** ; **DX:AX** = -10000 (**DX** = 0xFFFF, **AX** = 0xD8F0)  
 **cwde** ; **EAX** = -10000 (**EAX** = 0xFFFFD8F0)

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

De ce coexistă **CWD** cu **CWDE**? **CWD** trebuie să rămână **din rațiuni de backwars compability** și din rațiuni de funcționalitate a instrucțiunilor **(I)MUL** și **(I)DIV**.  
 **MOV** ah, 0c8h  
 **MOVSX** ebx, ah ;EBX = FFFFFFC8h | **MOVSX** ax, [v] ; MOVSX ax, byte ptr DS:[offset v]   
 **MOVZX** edx, ah ;EDX = 000000C8h | **MOVSX** eax, [v], syntax error (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. Impactul representării little-endian asupra accesării datelor**

Dacă programatorul utilizează datele consistent cu dimensiunea de reprezentare stabilită la definire (ex: accesarea octeților 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 conține cont în mod AUTOMAT de modalitatea de reprezentare little-endian. Ca urmare, dacă se respectă această condiție programatorul nu trebuie să intervină suplimentar în niciun fel pentru a asigura corectitudinea accesării și manipulării datelor.

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 octeților 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  
 structural normale echivalente unei reprezentări **big endian**.

**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 assume 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 octeților în memorie) și să utilizeze modalități de accesare în conformitate cu acesta**. Exemplu:

**segment** data  
 a dw 1234h ; datorită reprezentării little-endian, în memorie octeții sunt:  
 b dd 11223344h ; 34h 12h 44h 33h 22h 11h  
 a+0 a+1 b+0 b+1 b+2 b+3  
 c db -1  
**segment** code  
 mov al, byte[a + 1] ; accesarea lui a drept octet, efectuarea calcului de adresă a+1,  
 selectarea octetului de la adresa a+1 (octetul de valoare 12h) și  
 transferal 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 utlizarea operatorului de conversie WORD  
 mov bx, [b] ; bx := 3344h  
 mov bx, [a + 2] ; bx := 3344h deoarece ca adrese 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]; eax := 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

**5. Operații**  
**5.1. Operații aritmetice**

**Operanzii sunt reprezentați în codcomplementar. 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 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 operanzii 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 = 1001 1100b (=156 **fără semn** și -100 **cu semn**)  
 B = 4Ah = 0100 1010b (= 74 **fără semn** și **cu semn**)

Microprocesorul realizează adunarea C = A + B și obține:  
 C = E6h = 1110 0110b (= 230 **fără semn** și -26 **cu semn**)

Se observă că și 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.

**5.2 Operații logice pe biți (AND; OR, XOR și NOT)**

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 invers.

**5.3. Deplasări și rotiri de biți**

**Instrucțiunile de deplasare de biți** în cadrul unui operand se clasifică în:  
- **instrucțiuni de deplasare logică**:  
 - stânga: **SHL**  
 - dreapta: **SHR**  
- **instrucțiuni de deplasare aritmetică**:  
 - stânga: **SAL** - dreapta: **SAR**

**Instrucțiunile de rotire a biților** în cadrul unui operand se clasifică în:  
- **instrucțiuni de rotire fără carry**:  
 - stânga: **ROL**  
 - dreapta: **ROR**  
- **instrucțiuni de rotire cu carry**:  
 - stânga: **RCL**  
 - dreapta: **RCR**

Pentru a defini deplasările și rotirile să considerăm ca și configurație inițială un octet X = abcdefgh, unde a-h sunt cifre binare, h este cifră 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 ; X = bcdefgh0 și CF = a  
 SHR X, 1 ; X = 0abcdefg și CF = h  
 SAL X, 1 ; identic ca SHL  
 SAR X, 1 ; X = aabcdefg și CF = h  
 ROL X, 1 ; X = bcdefgha și CF = a  
 ROR X, 1 ; X= habcdefg și CF = h  
 RCL X, 1 ; X = bcdefgh**k** și CF = a  
 RCR X, 1 ; X = kabcdefg și CF = h

**6. Ramificări, salturi, cicluri**

**6.1. Saltul necondiționat**

În această categorie intră instrucțiunile **JMP** (echivalentul instrucțiunii GOTO din alte limbaje), **CALL** (apelul 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 *operand*** | Salt necondiționat la adresa determinată de operand |
| **CALL *operand*** | Transferă controlul procedurii determinată de operand |
| **RET *[n]*** | Transferă controlul instrucțiunii de după **CALL** |

**6.2. 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 necondițional al controlului la instrucțiunea care 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: . . . . . . . .

registrul **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 (2) segment data  
 **jmp** eax ; operand registru Salt DD Dest ; Salt := offset Dest  
 ; jmp [eax] ? segment code  
 etich: . . . . . . . . . jmp [Salt] ; salt NEAR  
 . . . . . .  
 Dest: . . . . . . . . .

Dacă în cazul (1) dorim înlocuirea operandului destinație registru cu un operand destinație variabilă de memorie, o soluție posibilă este:  
(1’) b rest1  
 . . .  
 mov [b], DWORD etich ; b := offset etich  
 jmp [b] ; salt NEAR – operand variabilă de memorie JMP DWORD PTR DS: [offset\_b]

6.3. Instrucțiuni de salt condiționat