2.6.4. Regiştrii de adresă şi calculul de adresă

Adresa unei locaţii – nr. de **octeţi** consecutivi dintre începutul memoriei RAM şi începutul locaţiei respective.

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

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
   - (a) 64K for 16-bit processors
   - (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 locaţii faţă de începutul unui segment, sau, cu alte cuvinte, numărul de octeţi aflaţi între începutul segmentului şi locaţia în cauză. Un offset se consideră valid dacă şi numai dacă valoarea sa numerică, pe 32 biţi, nu depăşeşte limita (dimensiunea) segmentului la care se raportează.

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

$$s_3s_2s_1s_0 : o_7o_6o_5o_4o_3o_2o_1o_0$$

În acest caz, selectorul $s_3s_2s_1s_0$ indică accesarea unui segment a cărui adresă de bază este de forma $b_7b_6b_5b_4b_3b_2b_1b_0$ şi având o limită $l_7l_6l_5l_4l_3l_2l_1l_0$. Baza şi limita sunt determinate de către procesor în urma aplicării mecanismului de segmentare.

Pentru a fi permis accesul către locaţia specificată, este necesar să fie îndeplinită condiţia:

$$o_7o_6o_5o_4o_3o_2o_1o_0 \leq l_7l_6l_5l_4l_3l_2l_1l_0.$$

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

$$a_7a_6a_5a_4a_3a_2a_1a_0 := b_7b_6b_5b_4b_3b_2b_1b_0 + o_7o_6o_5o_4o_3o_2o_1o_0$$

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 şi 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 şi se blochează accesul dacă nu a fost definit un astfel de segment;
2. Extrage adresa de bază (B) şi limita acestui segment (L), de exemplu, ca rezultat am putea avea B = 2000h şi L = 4000h; (este o operaţie la ale cărei detalii NU avem acces, ea derulându-se exclusiv între procesor şi SO)
3. Verifică dacă offsetul depăşeşte limita segmentului: 1000h > 4000h ? în caz de depăşire accesul ar fi fost blocat (*memory violation error*);
4. Adună offsetul cu B, obţinând 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 şi au dimensiunea maximă posibilă (4GiB), orice offset este automat valid şi 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 $s_3s_2s_1s_0 : o_7o_6o_5o_4o_3o_2o_1o_0$ va rezultă în adresa liniară:

$$a_7a_6a_5a_4a_3a_2a_1a_0 := 00000000 + o_7o_6o_5o_4o_3o_2o_1o_0$$

$$a_7a_6a_5a_4a_3a_2a_1a_0 := o_7o_6o_5o_4o_3o_2o_1o_0$$

⇨

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

Configurarea şi controlul mecanismelor de segmentare şi 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 şi folosirea mecanismelor de segmentare şi paginare sunt influenţate de *modul de execuţie* al procesorului, procesoarele x86 suportând următoarele moduri de execuţie mai importante:

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

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

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

- *segment de cod*, care conţine instrucţiuni maşină;
- *segment de date*, care conţine date asupra cărora se acţionează în conformitate cu instrucţiunile;
- *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 execuţiei este activ cel mult câte un segment din fiecare tip. Regiştrii **CS** (*Code Segment*), **DS** (*Data Segment*), **SS** (*Stack Segment*), **ES** (*Extra Segment*) din **BIU** conţin valorile selectorilor segmentelor active, corespunzător fiecărui tip. Deci regiştrii CS, DS, SS şi ES determină adresele de început şi dimensiunile segmentelor active: de cod, de date, de stivă şi suplimentar. Regiştrii **FS** şi **GS** pot reţine selectori indicând către segmente suplimentare, fără însă a avea roluri predeterminate. Datorită utilizării lor, CS, DS, SS, ES, FS şi GS poartă denumirea de *regiştri de segment* (sau *regiştri selectori*). Registrul **EIP** (care oferă şi posibilitatea accesării cuvântului său inferior prin subregistrul **IP**) conţine offsetul instrucţiunii curente în cadrul segmentului de cod curent, el fiind manipulat exclusiv de către **BIU**.

Cum noţiunile asociate adresării sunt fundamentale înţelegerii funcţionării procesoarelor x86 şi programării în limbaj de asamblare, este foarte importantă cunoaştera acestora. Pentru aceasta, le recapitulăm pe scurt în vederea clarificării:

Noţiune	**Reprezentare**	**Descriere**
Specificare de adresă, adresă logică,adresă FAR	$Selector_{16}:offset_{32}$	Defineşte complet atât segmentul cât şi deplasamentul în cadrul acestuia
Selector de segment	16 biţi	Identifică unul dintre segmentele disponibile. Ca valoare numerică acesta codifică poziţia descriptorului de segment selectat în cadrul unei tabele de descriptori.
Offset, adresă NEAR	$Offset_{32}$	Defineşte doar componenta de offset (considerând segmentul cunoscut ori folosirea modelului de memorie flat)
Adresă liniară (adresă de segmentare)	32 biţi	Inceput segment + offset, reprezintă rezultatul calculului de adresă
Adresă fizică efectivă	Cel puţin 32 biţi	Rezultatul final al segmentării plus, eventual, paginării. Adresa finală obţinută de către BIU, indicând în memoria fizică (hardware)

2.6.5. Reprezentarea instrucţiunilor maşină

O instrucţiune maşină x86 reprezintă o secvenţă de 1 până la 15 octeţi, care prin valorile lor specifică o operaţie de executat, operanzii asupra cărora va fi aplicată, precum şi modificatori suplimentari care controlează modul în care aceasta va fi executată.

O instrucţiune maşină x86 are maximum doi operanzi. Pentru cele mai multe dintre instrucţiuni, cei doi operanzi poartă numele de *sursă*, respectiv *destinaţie*. 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 instrucţiune are forma:

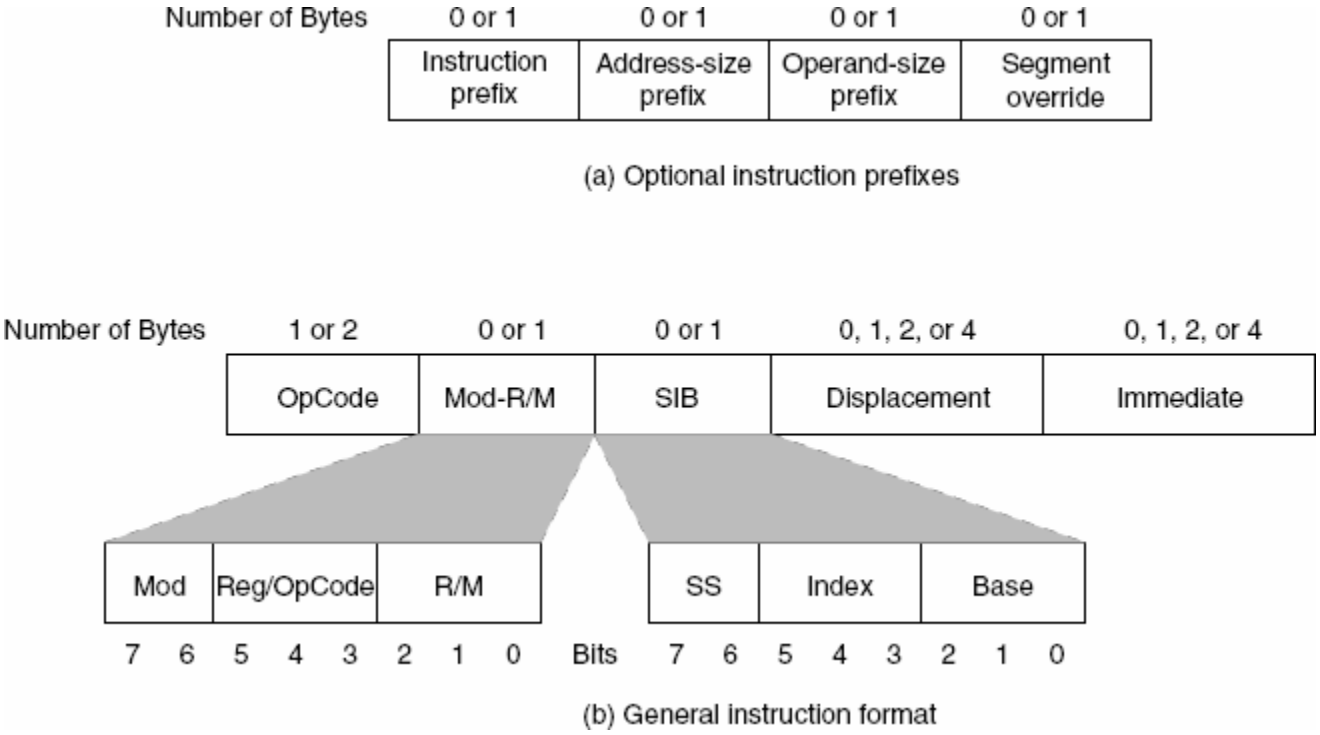
numeinstrucţiune *destinaţie****,*** *sursă*

Formatul intern al unei instrucţiuni este variabil, el putând ocupa între 1 şi 15 octeţi, având următoarea formă generală de reprezentare (*Instructions byte-codes from OllyDbg*):

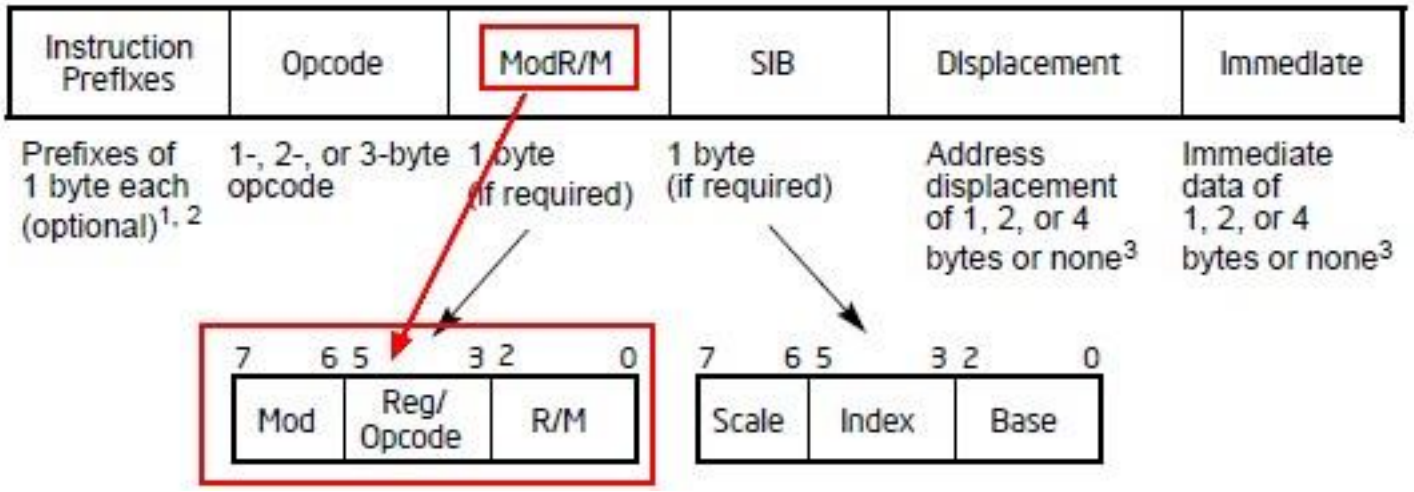
[prefixe] + cod + [ModR/M] + [SIB] + [deplasament] + [imediat]

Prefixele controlează modul în care o instrucţiune se execută. Acestea sunt opţionale (0 până la maximum 4) şi ocupă câte un octet fiecare. De exemplu, acestea pot solicita execuţia repetată (în buclă) a instrucţiunii curente sau pot bloca magistrala de adrese pe parcursul execuţiei pentru a nu permite accesul concurent la operanzi şi rezultate.

Operaţia care se va efectua este identificată prin intermediul a 1 sau 2 octeţi de *cod* (opcode), aceştia fiind singurii octeţi obligatoriu prezenţi, indiferent de instrucţiune.



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 instrucţiuni natura şi locul operanzilor (registru, memorie). Acesta permite specificarea fie a unui registru, fie a unei locaţii 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:

offset = [bază] + [index × scală] + [constantă]
(SIB) (deplasament + imediat)

unde pentru bază şi index vor fi folosite valorile a doi regiştri iar scală este 1, 2, 4 sau 8. Regiştrii permişi ca bază sau / şi 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 instrucţiunilor 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) şi 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 biţi).

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 şi consecinţă a imposibilităţii prezenţei mai multor câmpuri de ModR/M, SIB şi deplasament într-o instrucţiune, arhitectura 80x86 NU permite codificarea a două adrese de memorie în aceeaşi instrucţiune.

Valoare imediată oferă posibilitatea definirii unui operand ca fiind o constantă numerică pe 1, 2 sau 4 octeţi. Când este prezent, acest câmp apare întotdeauna la sfârşitul instrucţiunii.

2.6.6. Adrese FAR şi NEAR

Pentru a adresa o locaţie 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 regiştrii de segment CS, DS, SS sau ES**. Alegerea implicită a unui registru de segment se face după nişte reguli proprii instrucţiunii folosite.

Prin definiţie, 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 si ea se poate exprima 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 şi conţine cei 6 octeţi constituind adresa FAR. (ceea ce numim variabila pointer in limbajele de nivel inalt)

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 conţine selectorul care indică segmentul.

Reprezentarea adreselor respectă principiul reprezentării little-endian expus în capitolul 1, paragraf 1.3.2.3: partea cea mai puţin 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 instrucţiuni există 3 moduri de a specifica un operand pe care aceasta îl solicită:

- *modul registru*, dacă pe post de operand se află un registru al maşinii; mov eax, 17
- *modul imediat*, atunci când în instrucţiune se află chiar valoarea operandului (nu adresa lui şi nici un registru în care să fie conţinut); mov eax, 17
- *modul adresare la memorie*, dacă operandul se află efectiv undeva în memorie. În acest caz, offsetul lui se calculează după următoarea formulă:

$$\textit{adresa\_offset} = [\text{ bază }] + [\text{ index} \times \text{scală }] + [\text{ constanta }]$$

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

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

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

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

La instrucţiunile de salt mai apare şi un alt tip de adresare numit adresare *relativă*.

Adresa relativă indică poziţia următoarei instrucţiuni de executat, în raport cu poziţia curentă. Poziţia este indicată prin numărul de octeţi de cod peste care se va sări. Arhitectura x86 permite atât adrese relative scurte (SHORT Address), reprezentate pe octet şi având valori între -128 şi 127, cât şi adrese relative apropiate (NEAR Address), pe dublucuvânt cu valori între -2147483648 şi 2147483647.

```
Jmp MaiJos  ; aceasta instructiune se traduce (vezi OllyDbg) de obicei in Jmp [0084]↓
…….
…….
MaiJos:
   Mov eax, ebx
```