Arhitectura CPU 8086. Lucrul cu memoria

În acest capitol se prezintă diverse exerciţii şi probleme ce abordează noţiuni de bază cu privire la elementele componente ale arhitecturii unui sistem bazat pe procesorul 8086. Astfel, este absolut necesară cunoaşterea modului de lucru cu regiştrii procesorului 8086: cu regiştrii de uz general AX,BX,CX,DX, cu regiştrii index DI, SI, cu regiştrii pentru stivă BP şi SP, cu regiştrii segment DS, CS, SS, ES, cu pointerul la instrucţiunea următoare IP, dar şi cu registrul de flaguri FLAGS; astfel, sunt abordate mai multe exerciţii şi probleme ce vizează:

(1) modul de lucru cu regiştrii procesorului pe 8 şi 16 biţi;

(2) acomodarea cu folosirea corectă a regiştrilor şi de dimensiune potrivită în funcţie de mărimea datelor, respectiv gama de valori; de exemplu, utilizarea regiştrilor de 8 biţi pentru valori cuprinse în intervalul [0, 255], sau a regiştrilor de 16 biţi dacă se utilizează valori cuprinse între [256, 65535] (numere fără semn);

(3) înţelegerea indicatorilor de condiţie ai procesorului (flag-urile) şi modul în care aceştia sunt afectaţi în urma operaţiilor aritmetice; este necesară cunoaşterea teoretică a semnificaţiei acestor flaguri, precum şi a modului de setare/ resetare a acestora; în plus, interpretarea lor este diferită la numere fără semn vs cu semn; e indicată şi cunoaşterea instrucţ. de modificare a acestor flaguri (cmc, clc, stc).

Ca o continuare a capitolului anterior, s-au introdus directivele de definire a datelor: db, dw, dd şi modul de lucru cu aceste date. Un aspect foarte important abordat în acest capitol este lucrul cu memoria (şi stiva); e necesară înţelegerea conceptului de adresare segmentată (lucrul cu segmentele) şi a modului de adresare a datelor din memorie (adresare directă, indirectă, calculul adresei din memorie); astfel, s-au abordat exerciţii şi probleme care au vizat: (1) modul de calcul al adreselor locaţiilor de memorie, în funcţie de valoarea registrului segment şi a offsetului; (2) definirea datelor în memorie pe octet, cuvânt, dublucuvânt; (3) definirea variabilelor individuale, dar şi a şirurilor cu mai multe elemente; (4) accesul la datele definite în memorie, folosind adresa variabilelor sau un registru index pentru adresarea relativă la zona de început a variabilei/ vectorului din memorie; (5) accesul la octeţii individuali ai datelor definite pe cuvânt şi dublucuvânt, conform convenţiei *Little Endian*; (6) directive pentru conversia tipului de date atunci când ele sunt încărcate în locaţii de dimensiuni diferite (directivele de forţare a tipului byte ptr, word ptr).

Sunt abordate şi primele instrucţiuni de transfer (mov, lea), cele pentru lucrul cu stiva (push, pop), cele pentru lucrul cu registrul FLAGS (lahf, sahf, pushf, popf), dar şi o instrucţiune aritmetică (add) pentru a înţelege mai uşor lucrul cu indicatorii de condiţii.

1) **Arhitectura procesorului 8086**: regiștrii interni, lucrul cu memoria, lucrul cu stiva, codificarea și decodificarea instrucțiunilor
2) Despre **flagurile CPU**: afectarea lor de catre operatiile de adunare si scadere
    a. Regulile de corectie a rezultatului folosind flagurile CPU -> la nr unsigned si la nr signed
3) Segmente -> **adresarea segmentata**: cod/ date/ stiva -> AL si AF
4) Calculul adresei unui operand din memorie: AE =[BX/BP] + [SI/DI] + [deplasament]
5) Definirea datelor in memorie: directivele db, dw, dd
6) Definirea sirurilor de valori -> fortarea tipului prin *byte ptr* si *word ptr*
7) Instructiunile cu stiva **push** si **pop** -> vizualizarea conținutului stivei

Arhitectura procesorului 8086: regiștrii interni, lucrul cu memoria, lucrul cu stiva, codificarea și decodificarea instrucțiunilor

3. Arhitectura microprocesorului 8086. Lucrul cu memoria

2. Reprezentarea datelor în sisteme cu microprocesor 80x86

Cum se reprezintă informaţia din memoria SC ?

Informaţia din SC, reprezentată sub forma numerelor binare, poate ocupa un anumit număr (finit) de biţi, uzual folosindu-se următoarele entităţi:

- ***bit*** - poate fi 0 sau 1,
- ***nibble*** - cu ajutorul unui nibble se pot reprezenta 16 valori diferite;
- ***octet*** - un număr de 8 biţi cu ajutorul cărora pot fi reprezentate 256 valori diferite;
- ***cuvânt*** - un număr de 16 biţi sau 2 octeţi desemnează un cuvânt, numărul valorilor reprezentabile fiind 65 536;
- ***dublucuvânt*** – un nr de 4 octeţi, deci 32 biţi - permite reprezentarea a 2^{32} valori,
- ***cvadruplucuvânt*** – un nr de 8 octeţi, deci 64 biţi – se pot reprezenta 2^{64} valori

Astfel, în timp, dimensiunile uzuale ale operanzilor în PC au fost: ***octet*** (în engleză *byte*), ***cuvânt*** (în engleză *word*), ***dublucuvânt*** (în engleză *doubleword*) şi ***cvadruplucuvânt*** (în engleză *quadword*), aşa cum apare în Figura 2.4; pentru a fi cât mai uşor de urmărit, s-au reprezentat biţii grupaţi în octeţi.

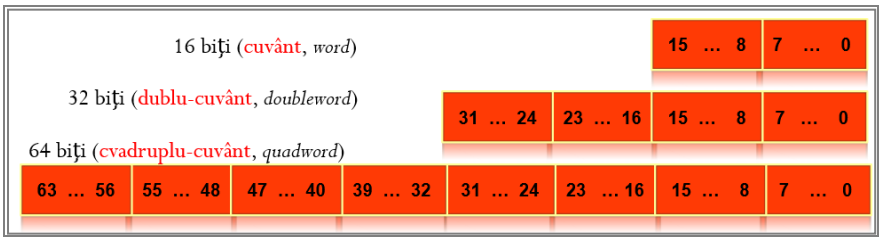

Figura 2.4 Multipli uzuali ai octetului: cuvânt, dublu-cuvânt şi cvadruplu-cuvânt

Cum putem identifica octeţii din cadrul unui cuvânt, dublucuvânt, etc ?

Uneori, se mai practică și numerotarea octeţilor sau tetradelor din cadrul structurii multi-octet, de exemplu pentru un dublucuvânt, despre octetul având biţii 31...24 se spune că este **de rang 3**, numerotarea începând de la rangul 0 prin octetul cu biţii 7...0.

Operanzii folosiţi în PC pot avea şi dimensiuni mai mari, dar numai multipli de dimensiunea octetului şi în plus, aceştia sunt în general doar puteri ale lui 2: 2^0 octeţi = 1 octet, 2^1 octeţi = 1 cuvânt, 2^2 octeţi = 1 dublucuvânt, etc

Cum putem accesa informaţia din memorie?

Memoria internă a unui SC este văzută ca ***o succesiune de locaţii***, fiecare locaţie având un număr fix de biţi (analogie cu un dulap cu sertare, sertarele fiind toate de aceeaşi dimensiune). ***Locaţia*** este unitatea elementară de adresare a unei memorii, accesarea conţinutului realizându-se prin adresarea locaţiilor de memorie; aceasta înseamnă că ***folosind adresa, vom avea acces la conţinut*** (analogie cu o etichetă (postit) care precizează un număr de ordine pentru sertar: consultând eticheta accesăm conţinutul sertarului).

Câţi octeţi putem accesa la un moment dat ?

Reprezentarea din Figura 2.5 se referă la modul cum se depune în memorie octetul ***21h***, cuvântul ***43 21h*** sau dublucuvântul ***87 65 43 21h*** începând de la adresa 126 și deci ocupând 1 locaţie, 2 locaţii sau 4 locaţii succesive. Memoria la procesoarele din familia x86 (indiferent că vorbim de un procesor pe 8 biți, pe 16 biți, pe 32 biți sau pe 64 biți) este întotdeauna organizată la nivel de octet (spunem că este adresabilă la nivel de octet - "byte-addressable").

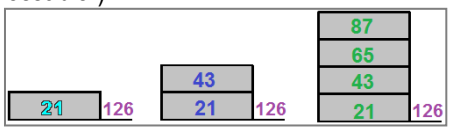

Figura 2.5 Octetul 21h, cuvântul 4321h și dublucuvântul 87654321h stocate în memorie începând de la adresa 126

Cum putem accesa octeţii din memorie folosind adrese ?

Dacă, de exemplu, se doreşte accesarea conţinutului 21h, atunci la adresare se va folosi locaţia unde se află acel conţinut, deci 126. O posibilitate de a realiza acest lucru este prin folosirea parantezelor: prin scrierea [126] ne vom referi la *conţinutul locaţiei de memorie de la adresa 126.* Dacă dorim să accesăm ***doar octetul 21h*** (adică să deschidem *un singur sertar* al dulapului), va trebui specificată cumva ***adresarea la nivel de octet***, deoarece este posibilă și ***adresarea la nivel de cuvânt*** (deschidem *2 sertare*) sau ***dublucuvânt*** (deschidem *4 sertare* consecutive) sau chiar la nivel de ***cvadruplucuvânt*** (deschidem *8 sertare* consecutive)[1] plecând de la o anumită adresă. În general, dimensiunea datelor accesate se va considera astfel încât să potrivească celei a registrului folosit în operaţia de accesare a acelei date din memorie.

De exemplu, o instrucţiune de forma:

[1] Ținând cont de procesorul folosit

mov AL, [126] ; *va accesa octetul din memorie* de la *adresa 126* și îl va copia (sau muta după cum sugerează instrucţiunea ***mov***) în *registrul AL*, care așa cum vom vedea ulterior, este un registru pe 8 biţi al procesorului 8086. Similar,
mov AX, [126] ; va accesa *cuvântul* din memorie de la *adresa 126* și îl va copia în *registrul AX*, care este un registru pe 16 biţi al procesorului 8086.
De exemplu, pe un processor de 32 biți este posibilă scrierea instrucțiunii:
mov EAX, [126] ; va accesa *dublu-cuvântul* din memorie de la *adresa 126* și îl va copia în *registrul EAX*, care este un registru pe 32 biţi al procesorului 80386.

Câţi biţi putem accesa minim din memorie ?

La procesoarele Întel, ***unitatea minimă de adresare*** (locaţia de memorie) ***are 8 biţi***, nu se poate accesa mai puţin, așa cum e cazul microcontrolerelor de exemplu (acces la nivel de bit). În schimb, memoria *se poate adresa și la nivel de* cuvânt (16 biţi) sau dublu-cuvânt (32 biţi), deci în general un multiplu de 8 biţi. Astfel, în Figura 2.6 se poate adresa un octet de la oricare din adresele 126, 127, 128 sau 129, sau un cuvânt, de exemplu cel format de adresele 126 şi 127 sau 127 și 128 (obligatoriu locaţiile se vor considera succesive în sens crescător), sau un dublucuvânt, aşa cum apare la locaţiile 126 ...129.

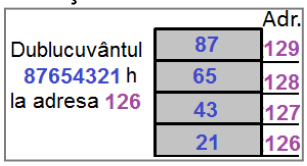

Figura 2.6 Dublucuvântul 87654321h stocat în memorie la adresele 129,...,126

Convenţional, ***bitul c.m.p.s.*** (cel mai puţin semnificativ, în engleză LS - *least significant* bit) al unei valori, numerotat ca bitul b_0, se află în poziţia cea mai din dreapta a valorii, iar ***bitul c.m.s.*** (cel mai semnificativ, în engleză MS - *most significant* bit) în poziţia cea mai din stânga, aşa cum se poate urmări în Figura 2.7.

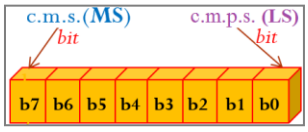

Figura 2.7 Numerotarea biţilor într-un octet: cel mai semnificativ (c.m.s.) şi cel mai puţin semnificativ (c.m.p.s.) *bit dintr-un octet*

Această specificare e importantă atunci când se consideră operaţiile logice la nivel de bit, precum cele de deplasare sau rotire deoarece atunci se dorește acces în interiorul octetului. Reamintesc aici că biţii nu-și modifică niciodată poziţia la accesare, bo rămâne LSb (în extrema dreaptă), iar b7 rămâne MSb (în extrema stângă). Această organizare a biţilor într-un octet poate fi generalizată şi asupra octeţilor (aşa cum arată Figura 2.8), a cuvintelor, etc.

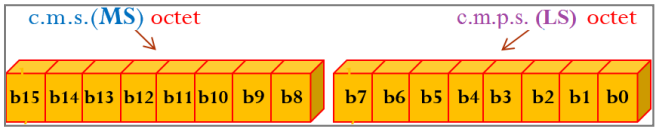

Figura 2.8 Cel mai semnificativ (c.m.s.) şi cel mai puţin semnificativ (c.m.p.s.) *octet dintr-un cuvânt*

În care sertar va ajunge sacoul ?

Revenind la analogia cu dulapul cu sertare, să presupunem că trebuie sa depozităm în acest dulap un costum din 2 piese: sacou şi pantaloni. În care sertar va ajunge sacoul: în cel de deasupra sertarului cu pantaloni, sau în cel de sub acesta? Este importantă această organizare, deoarece la un moment dat am putea ruga pe cineva să ne livreze conţinutul sertarului 5 de exemplu (fiind siguri că acolo e sacoul). Această problemă se pune și în cazul organizării informaţiei în memorie. De exemplu, cuvântul din Figura 2.8, cum se va aranja în memorie? Octetul c.m.s. deasupra celui c.m.p.s. sau invers?

Cum se depun octeţii unui cuvânt în memorie? (în sus sau în jos?)

La adresa curentă e octetul c.m.s. sau c.m.p.s.(dintr-un cuvânt, de exemplu)?

Răspunsul la această întrebare, dacă nu se specifică familia din care face parte procesorul sau mai bine ***convenţia utilizată la reprezentare***, poate fi greşit/ ambiguu. Aceasta, deoarece rezultatul poate arăta ca în Figura 2.9a) sau ca în Figura 2.9b), în funcţie de convenţia Little End-ian sau Big End-ian folosită de sistem.

Dublu-cuvântul 87654321h se depune în memorie folosind una din convenţiile:

- Little Endian: octetul **LSB**, adică cel de la sfârşitul structurii (“**END**”-ian) se depune în memorie la locaţia cu **adresa cea mai mică (“Little”)** – această convenţie este specifică procesoarelor din familia *Întel* – Figura 2.9a);

- Big Endian: octetul **LSB** se depune în memorie la locaţia cu **adresa cea mai mare (“Big”),** convenţia fiind specifică procesoarelor din familia *Motorola* – Figura 2.9b).

Adresa	Conţinutul		Adresa	Conţinutul
0003	87	Dublucuvântul **87.65.43.21h** în memorie de tip Little sau Big END-ian	**0003**	**21**
0002	65		0002	43
0001	43		0001	65
0000	**21**		0000	87
Little END-ian			**Big END-ian**	

Figura 2.9 Depunerea dublucuvântului 87654321h în memorie după a) convenţia Little Endian (stânga); b) convenţia Big Endian (dreapta)

Exercițiu: începând de la adresa 17102h, reprezentați pe foaia de hârtie conținutul memoriei dacă se definesc în ordine: doi octeți de valoare 12h și 21h, două cuvinte de valoare 3456h și 7890h și un dublucuvânt de valoare FEDCBAh (folosiți directive de definire a datelor în memorie). Știind că 81h la nivel de octet reprezintă valoarea 129 ca număr fără semn, resp. -127 ca număr cu semn, definiți încă 2 variabile în memorie, pe cuvânt, una cu valoare în hexacimal și una cu valoare negativă în zecimal. Reprezentați-le în memorie.

Rezolvare:

Urmărind convenția Little END-ian (vom folosi întotdeauna procesor de tip Intel sau AMD), datele se vor depune în memorie așa cum arată Figura 2.10.

Pentru a defini aceste date, vom scrie directive de definire a datelor, sub forma:

```
a db 12h
b db 21h
cuv1 dw 3456h
cuv2 dw 7890h
var dd 0FEDCBAh
e dw 81h
f dw -127
```

Observații:

Directiva **db** (provine de la ***define byte***) se folosește pentru a defini octeți sau șiruri de octeți în memorie. În locul celor 2 directive de definire a octeților, am fi putut scrie o singură directivă sub forma: **a db 12h, 21h**

Directiva **dw** (provine de la ***define word***) se folosește pentru a defini cuvinte sau șiruri de cuvinte în memorie. În locul celor 2 directive de definire a cuvintelor, am fi putut scrie o singură directivă sub forma: **cuv dw 3456h, 7890h**

Directiva **dd** (provine de la ***define doubleword***) se folosește pentru a defini dublucuvinte sau șiruri de dublucuvinte în memorie. In general, în fața valorilor hexazecimale care încep cu literă, se pune un 0 pentru ca asamblorul să nu confunde aceste valori cu așa numitele "etichete".

17102h	12h
17103h	21h
17104h	56h
17105h	34h
17106h	90h
17107h	78h
17108h	BAh
17109h	DCh
1710Ah	FEh
1710Bh	00h
1710Ch	81h
1710Dh	00h
1710Eh	81h
1710Fh	FFh
17110h	

Figura 2.10 Depunerea a doi octeți în memorie de valoare 12h și 21h, a două cuvinte de valoare 3456h și 7890h și a unui dublucuvânt de valoare FEDCBAh

Din figură se observă că la definirea ***variabilei e***, extensia s-a realizat cu bit de 0, deoarece valoarea a fost specificată în hexazecimal pe un singur octet; în schimb, la definirea ***variabilei f***, deoarece s-a folosit valoarea în zecimal ca număr cu semn, s-a realizat extensia cu bitul de semn al valorii.

Cum putem ilustra conţinutul zonei de memorie?
(înspre adrese crescătoare sau înspre adrese descrescătoare?)

Deşi pare foarte simplu, trebuie acordată o deosebită atenţie la ilustrarea schematică a conţinutului memoriei în acest caz. În Figura 2.11 puteţi observa un *octet la adresa 118*, un *cuvânt la adresele 122-123* şi un *dublucuvânt la adresele 126-129*.

Conţinutul memoriei este identic în cele 2 cazuri prezentate în figură, doar că în partea stângă (Figura 2.11a)) s-a desenat zona de memorie de la adrese mai mari înspre adrese mai mici, iar în partea dreaptă (Figura 2.11b)) este invers.

Desenul este valabil pentru un procesor din familia Intel, de exemplu, deci care respectă formatul ***Little Endian***.

Se recomandă utilizarea desenului cu ***adrese crescătoare*** întrucât şi în cadrul simulatorului EMU8086 acestea sunt prezentate sub această formă. Recomandarea ţine mai mult de practică, pentru a nu greşi în interpretarea datelor din memorie.

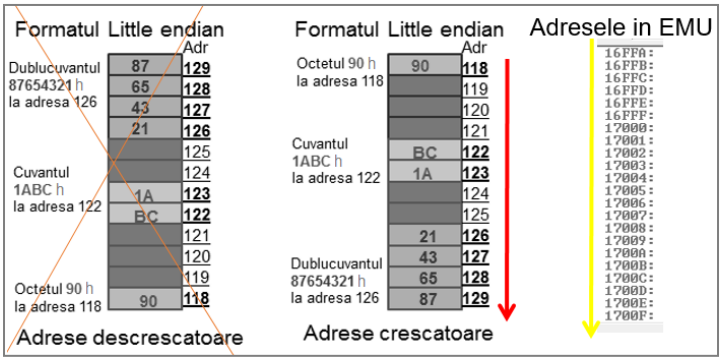

Figura 2.11 Formatul Little Endian ilustrat la adrese a) descrescătoare sau b) crescătoare

3. Interpretarea rezultatelor obținute de către procesor la adunarea și scăderea valorilor

Există vreo metodă de a verifica dacă rezultatul obţinut în urma unei operaţii aritmetice este corect ?

În general, atunci când se realizează o operaţie aritmetică, datorită dublei interpretări a valorilor (*fără semn* sau *cu semn*) rezultatul obţinut poate să nu fie atât de sugestiv pe cât ar trebui pentru a fi interpretat corect.

De exemplu, să considerăm o operaţie de adunare pe 3 biţi; dacă adunăm la nr 3 încă un 1, din 011b se transformă în 100b pentru care sunt posibile cele 2 interpretări:

- dacă numărul se consideră fără semn, atunci rezultatul obţinut este corect 100b=4 (am adunat 3+1 și am obţinut 4);
- în schimb, dacă se consideră interpretarea cu semn, atunci rezultatul nu mai este corect, pentru că interpretarea lui 100b ca număr cu semn este - 4, și nu 4 ! Altfel spus, am adunat 3 cu 1 și am obţinut -4 în loc de 4; aceasta din cauză că rezultatul ar fi avut nevoie de încă un bit pentru a fi stocat în mod corect (ca 0100b)– spunem că *am depășit domeniul sau gama de reprezentare a numărului,* sau în engleză se folosește termenul ***overflow***.

În concluzie, dacă am fi avut un bit suplimentar, rezultatul ar fi putut fi interpretat corect în ambele situaţii (în primul caz folosind extensia valorii). Problema este că regiștrii existenţi în CPU nu-și modifică forma; la 8086 aceștia rămân tot timpul de 8 sau 16 biţi, nu putem avea regiștri de 9 sau 17 biţi pentru a stoca date. Totuși, avem la dispoziţie, ca programatori, un indicator (“flag”) care poate juca acest rol de bit suplimentar.

De fapt, multiple situaţii care pot să apară după execuţia operaţiilor aritmetice sunt ilustrate în cadrul CPU prin niște indicatori de 1 bit numiţi *flag* în engleză – aceste flaguri trebuie văzute asemănător cu niște leduri - ele ne pot arăta diverse situaţii excepţionale care pot să apară după efectuarea unei operaţii, aprinzându-se (devin 1, sau spunem că se setează). Dacă situaţia respectivă nu apare sau nu se mai menţine, flagul corespunzător se va stinge (devine 0, sau spunem că se resetează).

Ce sunt flagurile aritmetice și cum ne pot ajuta ele în interpretarea rezultatelor?

Ce trebuie să reţinem din paragrafele anterioare este că există o posibilitate de a observa diverse situaţii excepţionale care pot să apară după efectuarea operaţiilor – în special cele aritmetice (de adunare și scădere) în SC; pentru a ajuta la gestionarea corectă a acestor situaţii excepţionale, se pot interpreta așa numitele ***flaguri aritmetice***. Acestea pot arăta una din următoarele situaţii:

Overflow – apare (sau spunem că se setează flagul corespunzător) când se depășește gama de reprezentare a numărului: dacă rezultatul obţinut nu a încăput în gama destinată stocării lui, atunci există un indicator care va avea valoarea 1; altfel, acest indicator va avea valoarea 0.

Carry sau ***Borrow*** - apare atunci când se realizează un transport: există un indicator ce va avea valoarea 1 în cazul în care ultima operaţie efectuată a generat un transport în/ din afara domeniului de reprezentare a numărului și valoarea 0 în caz contrar;

- operaţie de transport *în afara rezultatului*, probabil *obţinut printr-o operaţie de adunare*, sau
- operaţie de transport *din afara rezultatului* (caz în care își schimbă denumirea în borrow, în română împrumut) și probabil s-a *obţinut printr-o operaţie de scădere*;

Zero - semnalizează dacă rezultatul ultimei operaţii este egal cu zero.

Sign - semnalizează dacă rezultatul ultimei operaţii este un număr strict negativ.

Exemple: Se vor considera operaţiile realizate pe 4 biţi, deci și rezultatul obținut va trebui considerat tot pe 4 biţi:

a) 2h+2h=4h -> **C=0, Z=0, S=0, O=0**;

b) 3h+7h=Ah -> **C=0, Z=0, S=1, O=1** (a apărut depășire de gamă: +10 s-ar fi scris corect pe 5 biţi, folosind gama [-16;+15], iar ca numere fără semn nu a apărut transport înafara domeniului de reprezentare, numărul 10 este scris corect);

c) 7h+Bh=2h -> **C=1, Z=0, S=0, O=0** (ca numere cu semn, se operează 7+(-5) =2, dar ca numere fără semn, a apărut transport înafara domeniului de reprezentare);

d) Ah+Bh=5h -> **C=1, Z=0, S=0, O=1** (a apărut depășire de gamă: se operează (-6)+(-5)=-11 și care s-ar fi scris corect pe 5 biţi, folosind gama [-16;+15], iar ca numere fără semn a apărut transport înafara domeniului de reprezentare, numărul 10+11=21 s-ar fi scris corect pe 5 biţi, în gama [0; 31]);

e) 8h+8h=0h -> **C=1, Z=1, S=0, O=1** (a apărut depășire de gamă: se operează (-8)+(-8)=-16 și care s-ar fi scris corect pe 6 biţi, folosind gama [-32;+31], iar ca numere fără semn a apărut transport înafara domeniului de reprezentare, numărul 8+8=16 s-ar fi scris corect pe 5 biţi, în gama [0; 31]).

Ce semnificaţie vrem noi să dăm numerelor din PC, doar noi știm: CPU nu are de unde să știe acest lucru. CPU reprezintă numărul în binar și atât; ce poate face ulterior cu acest număr este să interpreteze bitul c.m.s. al acestei valori ca fiind un bit ce arată semnul numărului (sau să nu îl interpreteze astfel). De exemplu, valoarea care pentru noi oamenii înseamnă -1 și se reprezintă în binar ca FFFFh, pentru CPU poate însemna la fel de bine și 65535 (adică în reprezentarea fără semn). Astfel, va trebui să acordăm o foarte mare importanţă la interpretarea valorilor cu care lucrăm.

La 8086, există instrucţiuni care să sugereze modul de interpretare al valorilor de către CPU?

În interpretarea numerelor, există unele ***instrucţiuni care sugerează modul de interpretare al valorilor de către CPU***; de exemplu, operaţia de înmulţire are 2 variante (am putea spune că este o instrucţiune dual): pentru operare numere fără semn, respectiv pentru operare numere cu semn (programatorul va sugera modul cum dorește ca CPU să considere valorile binare respective); identic și la împărţire. Totuși, cele mai multe instrucţiuni nu ţin cont de această interpretare. De exemplu, adunarea: nu avem de unde ști că adunăm două numere pozitive și rezultatul intră în depășire și deci obţinem (pe același număr de biţi) un rezultat negativ, *decât dacă interpretăm noi (ca programatori) corect rezultatele și flagurile*.

Exemplu: 3h + 7h= Ah – care e un număr negativ dacă e interpretat în convenţia cu semn; am adunat două nr pozitive (se vede simplu, după c.m.s. bit care este 0) și am obţinut un număr negativ: 3 cu 7 și am obţinut 10 (ca număr fără semn) sau -6 (ca număr cu semn), depinde cum vrem să-l interpretăm. Programatorul este cel care trebuie să gestioneze toate aceste posibile situaţii excepţionale. Vom vedea unde sunt stocate aceste flaguri și mai ales cum le putem consulta în cadrul simulatorului, în continuare. Deocamdată, le-am urmărit la modul simplist, intuitiv, doar din prisma reprezentării cu gama numerelor.

Reamintesc aici gama numerelor:

Tabelul 2.1. Gama numerelor fără semn

Reprezentare	Nr octeţi	Gama numerelor fără semn	Puterea lui 2
octet	1	0 … 255	0 … 2^8-1
cuvânt	2	0 … 65 535	0 … 2^{16}-1
dublucuvânt	4	0 … 4 294 967 295	0 … 2^{32}-1
cvadruplucuvânt	8	0 … 18 446 824 753 389 551 615	0 … 2^{64}-1

Tabelul 2.2. Gama numerelor cu semn

Reprezentare	Nr octeţi	Gama numerelor cu semn	Puterea lui 2
octet	1	-128 … +127	-2^7 … $+2^7$-1
cuvânt	2	-32 768 … +32 767	-2^{15} …$+2^{15}$-1
dublucuvânt	4	-2 147 483 648 … +2 147 483 647	-2^{31} …$+2^{31}$-1
cvadruplucuvânt	8	-9 223 412 376 694 775 808 ... +9 223 412 376 694 775 807	-2^{63} …$+2^{63}$-1

3. Arhitectura de bază a procesoarelor x86 pe 16 biţi

În structura de bază a unui computer (sau SC) pot exista diferite componente, însă ***cel puţin 3 categorii*** sunt necesare pentru buna funcţionare a sistemului:

1) una sau mai multe ***unităţi de prelucrare*** (în general, fiecare unitate implică unul sau mai multe procesoare),
2) ***memorie*** şi
3) ***periferice.***

În general, când vorbim despre microprocesor în contextul ***organizării şi arhitecturii sistemelor de calcul*** se înţelege că acesta reprezintă **unitatea centrală de prelucrare UCP** (în engleză CPU – Central Processing Unit), sau simplu ***procesor***.

3.1. Scurt istoric al familiei x86:

- Primul microprocesor **4004** era pe 4 biţi (apărut în 1970);
- doi ani mai târziu i-a urmat **8008** și apoi **8080** care erau procesoare pe 8 biţi;
- **8086** a apărut în 1978 şi apoi i-a urmat **80286** (procesoare pe 16 biţi). Un caz special este procesorul **8088**, cu aceeaşi structură ca 8086, dar care comunică în exterior printr-o magistrală de 8 biţi şi nu de 16 biţi ca 8086, proiectat astfel din economie. Îmbunătăţirile aduse de la un procesor la altul de-a lungul timpului s-au bazat pe tendinţa de a obţine o putere de calcul mai mare prin *creşterea numărului de biţi prelucraţi* la momentul curent.
- în următoarea etapă s-a trecut la prelucrări pe 32 biţi (începând cu **80386** și continuând apoi cu ***80486***, apoi seria ***Pentium I, II,*** *etc*)
- mai recent, au apărut cele pe 64 biţi (de la **Core 2** spre **Core i3, i5, i7**).

Figura 3.7 prezintă evoluţia familiei x86 mai detaliat. Totodată, s-au realizat în permanenţă inovaţii în cadrul arhitecturii interne, toate acestea ducând la o creştere a vitezei de prelucrare. Pentru a studia arhitectura unei generaţii de procesoare, abordarea cea mai des utilizată este de a porni de la studiul unui procesor considerat “de referinţă”, iar în acest caz acesta este procesorul 8086.

Când scriem programe (indiferent ca folosim HLL sau LLL[2]) acestea cuprind *secvenţe de instrucţiuni* necesare îndeplinirii unei anumite sarcini. Aceste *instrucţiuni* sunt apoi "traduse" în *secvenţe echivalente de instrucţiuni* în limbaj mașină (de către asamblor) pe care procesorul le înţelege; ulterior, S.O. încarcă programul în memoria principală (cu ajutorul unui program încărcător, numit loader), îi indică procesorului locaţia respectivă și "îl ghidează" spre execuţia lui.

3.2. Codificarea/ decodificarea instrucţiunilor

Codificarea instrucţiunilor trebuie realizată pentru că toate instrucţiunile (scrise în cadrul unui program) trebuie transformate (de către asamblor) în cod mașină și depuse în memorie, ca de acolo CPU să le poată lua și executa; astfel, orice instrucţiune (indiferent că e scrisă în LLL sau HLL) se va transforma în șiruri de 0 și 1 grupate în octeţi. Acești octeţi sunt depuși în memorie (tot programul va «suferi» aceleași modificări) și deci va ajunge undeva în memorie (în Figura 3.2 aceasta e sugerată ca începând de la adresa X). Procesul de transformare în acest sens se numește codificare a instrucţiunilor, iar cel invers poartă numele de decodificare a instrucţiunilor.

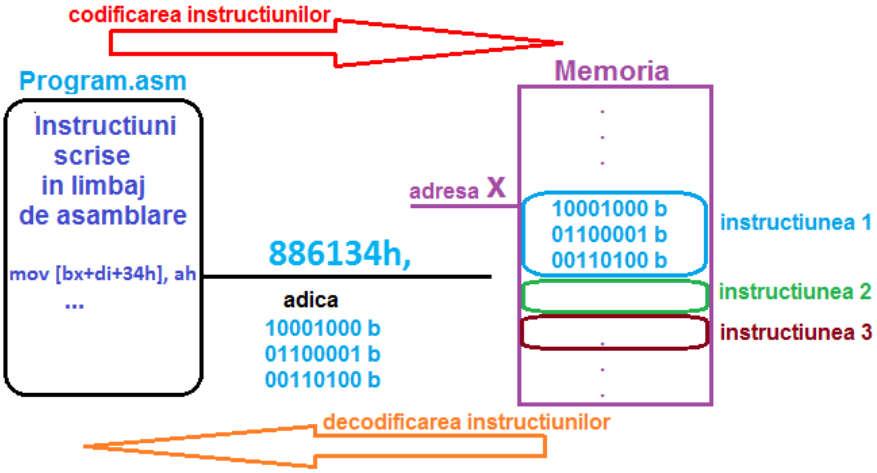

Figura 3.2. Codificarea și decodificarea instrucţiunilor

Ciclii mașină și tacţii

Fiecare instrucţiune (de exemplu, *mov ax,bx;*) implică mai multe operaţii interne ("ciclii mașină") ce trebuie executate de UCP, toate realizându-se sincronizat cu ceasul intern al UCP. O instrucţiune conţine în general mai mulţi ciclii mașină (de la 3 în sus), iar fiecare ciclu mașină durează în general mai mulţi tacţi ("stări").

De exemplu, operaţia denumită ***fetch*** se regăsește la toate instrucţiunile, la începutul execuţiei; această operaţie de fetch ***extrage sau aduce înspre CPU instrucţiunea din memorie*** (așa cum este ea, sub formă codificată, în memorie); aceasta se aduce din memorie sau de la port, folosind operaţie de RD memorie sau RD port, etc.

În general se spune că orice instrucţiune este de fapt o combinaţie de ciclii mașină și începe cu un ciclu de tip FETCH - pentru a se extrage din memorie sau de la port, deci pentru a aduce înspre CPU codul instrucţiunii. E nevoie de această operaţie de fetch pentru ca UCP să știe ce operaţie are de realizat.

Performanţa execuţiei unei instrucţiuni este, în general, specificată în *ciclii* de *ceas* (în loc de a fi exprimată în secunde).

Când se menţionează simplu termenul "*ceas*" se face referire la ceasul sistem (ceasul folosit de UCP). Totuși, există anumite busuri care deţin și ele propriul ceas – și referirea la acestea se va face precizând acest aspect (în general acestea au durată mai mare decât durata ceasului UCP, ducând la ușoare întârzieri, dar care nu afectează major performanţa sistemului).

3.3. Arhitectura software a microprocesorului I8086

Procesorul are o parte hardware și una software. D.p.d.v. hardware ne referim la infor-maţia despre construcţia fizică a acestuia; de exemplu, că numărul de tranzistoare al procesorului 8086 este de 29000 și că a fost fabricat în tehnologie de 3 µm. Termenul „arhitectură software" se referă la componentele arhitecturale care fac posibilă execuţia instrucţiunilor pe care le suportă (deci pe care le știe executa) acel procesor.

Deci dacă vrem să dăm instrucţiuni procesorului, cea mai des utilizată abordare este să scriem acele instrucţiuni în ***programe în limbaj de asamblare***, să le asamblăm și să le depunem în memorie sub formă de fișier **.com** sau **.exe** (deci fișiere executabile); de acolo, CPU le va prelua și apoi le va executa, instrucţiune cu instrucţiune. Care este setul de instrucţiuni suportat de procesorul 8086 și care dintre acestea au fost implementate în cadrul simulatorului, se va prezenta în secţiunile următoare.

[2] se referă la High și Low *Level Languages*, adică limbaje de programare de nivel ridicat, respectiv scăzut

3.3.1. Ce înseamnă procesor pe 16 biţi

Faptul că **I8086** este un **procesor pe 16 biţi** înseamnă că **registrii săi interni** sunt de dimensiune **16 biţi**; cu alte cuvinte, ei pot stoca un număr care să se scrie pe maxim 16 biţi, adică:

valori în gama [0; 65535] dacă se consideră numere *fără semn*, respectiv

valori în gama [-32768; +32767] dacă sunt considerate numere *cu semn*.

Registrul se definește ca o structură internă de bază a CPU, necesară în efectuarea operaţiilor. UCP 8086 are mai mulţi regiștri interni de 16 biţi care pot fi folosiţi.

Exemplu:

instrucţiunea: *mov AX, 2;* va încărca în registrul AX (care are dimensiunea de 16 biţi), valoarea 0002h. Unii dintre acești regiștri pot fi folosiţi și ca regiștri de 8 biţi, ca parte HIGH (deci AH) sau parte LOW (deci AL); astfel:

instrucţiunea *mov AL, 2;* va încărca în AL (fiind registru de 8 biţi) valoarea 02h.

Mai multe detalii despre lucrul cu regiștrii procesorului se vor specifica în secţiunile următoare.

3.3.2. Schema bloc internă

Începând cu microprocesoarele pe 16 biţi (8086, 80286), unitatea de prelucrare nu mai urmează strict schema descrisă de arhitectura von Neumann (extrage o instrucţiune, o decodifică, o execută ş.a.m.d). Ca element nou, s-a divizat unitatea de prelucrare în alte două unităţi (Figura 3.4) care să poată lucra în paralel:
- Unitatea de execuţie (Execution Unit - EU)
- Unitatea de interfaţă cu magistrala (Bus Interface Unit - BIU)

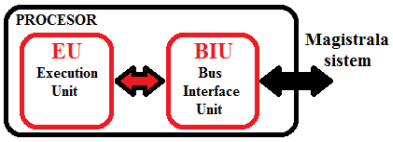

Figura 3.4. Cele două componente principale ale procesorului

Figura 3.4 prezintă detaliat structura internă a celor 2 unităţi în cadrul schemei bloc interne a procesorului 8086.

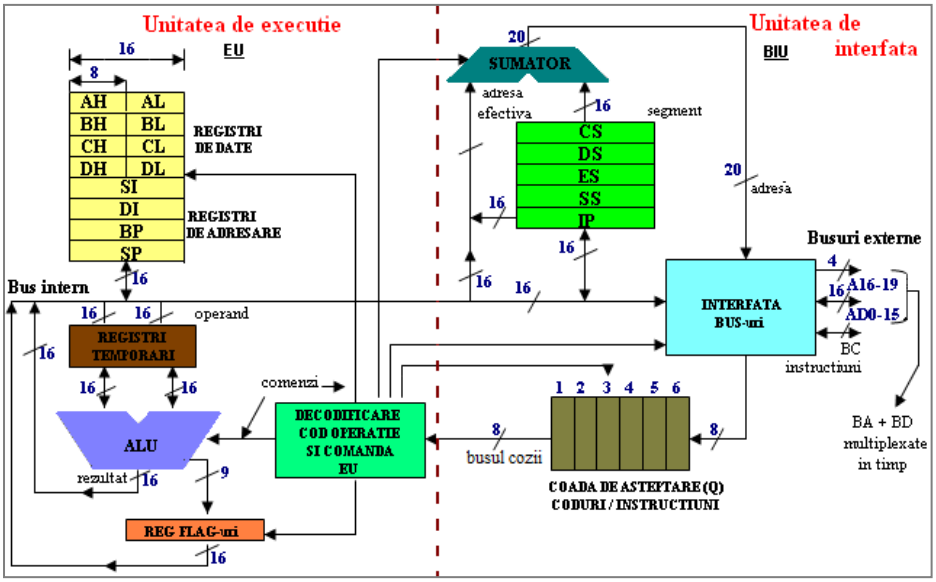

Figura 3.5 Schema bloc internă a microprocesorului I8086

Pipeline cu EU şi BIU : **ALU** (Arithmetic and Logic Unit în engleză sau UAL în română) și **UC** (în engleză Command Unit) prezentate în Figura 3.3 – se regăsesc în cadrul EU în Figura 3.5 cu **albastru** și **verde**. Cele două unităţi EU și BIU sunt legate între ele cu o conductă (*pipeline*) prin care sunt transferate instrucţiunile extrase (din program) de către BIU spre EU; unitatea de execuţie EU are doar rolul de a executa instrucţiunile extrase de BIU, neavând nici o legătură cu magistrala sistemului. În timp ce EU îşi îndeplineşte sarcina de a executa instrucţiuni, BIU extrage noi instrucţiuni pe care le organizează într-o coadă de aşteptare (*queue*). BIU pregăteşte execuţia fiecărei instrucţiuni astfel: extrage o instrucţiune din memorie, o depune în coada de instrucţiuni şi calculează adresa din memorie a unui eventual operand. La terminarea execuţiei unei instrucţiuni, EU are deja la dispoziţie o nouă instrucţiune din coada de aşteptare construită de BIU. Cele două unităţi, EU şi BIU, lucrează astfel în paralel, existând momente de sincronizare şi aşteptare între ele, funcţionarea paralelă a celor două unităţi fiind transparentă utilizatorului. Această arhitectură se mai numeşte şi arhitectură cu *prelucrare secvenţial – paralelă* de tip ***pipeline cu 2 etaje***.

Unitatea de execuţie EU conţine o unitate aritmetică și logică (ALU) și Unitatea de Comandă sau Control (UC) a EU de 16 biţi (ambele prezentate anterior în Figura 3.3), registrul indicatorilor de stare (FLAGS), registrul operatorilor (TEMPORARI) şi REGIŞTRII GENERALI (de date și adresare), așa cum se poate urmări în Figura 3.4.

Unitatea de interfaţă BIU conţine indicatorul de instrucţiuni IP (Instruction Pointer), REGIȘTRII SEGMENT (CS, DS, SS, ES), un bloc de CONTROL ȘI INTERFAŢARE al magistralei, un bloc SUMATOR de generare a adresei pe 20 biţi şi o memorie organizată sub forma unei COZI, în care sunt depuse instrucţiunile extrase (Instruction Queue, de 6 locaţii-octeţi). Singurele momente în care coada trebuie reiniţializată, şi deci EU trebuie să aştepte efectiv citirea unei instrucţiuni (BIU nu prea ajută EU în astfel de momente), sunt cele care urmează după execuţia unei instrucţiuni de salt.

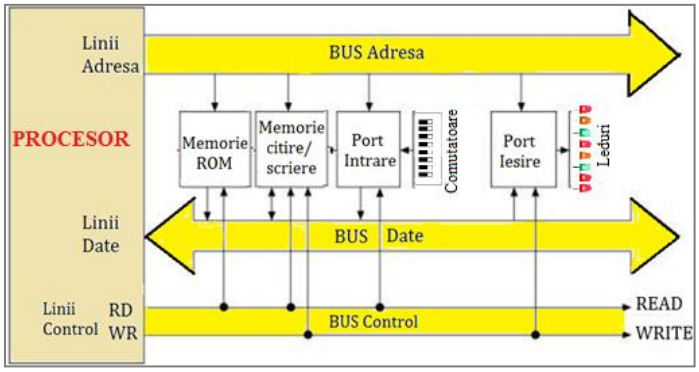

Figura 3.6 Interfaţarea cu exteriorul prin periferice de intrare și ieșire

Magistralele sunt seturi de fire sau linii folosite pentru transportul semnalelor.

Un SC pe 16 biţi are uzual regiştri pe 16 biţi şi 16 fire/ linii paralele într-un bus (magistrală). Uzual, ***bus-ul de date* (BD)** se foloseşte pt transportul datelor între CPU, RAM şi porturile I/O (unde se conectează dispozitivele periferice), ***bus-ul de adrese* (BA)** se foloseşte pt a specifica ce adresă RAM sau port I/O va fi folosit, iar ***bus-ul de control* (BC)** are la dispoziţie mai multe linii, dintre care: una pt a determina dacă se accesează RAM sau porturi I/O (semnalul IO); are de asemenea o linie pt a determina dacă datele sunt scrise sau citite, considerând că *CPU citeşte* date când acestea intră în CPU şi că *CPU scrie* date când acestea ies din CPU către RAM sau porturi.

Procesorul 8086 are o magistrală de 20 biţi, așa cum se poate observa în Figura 3.5, din care cei mai puţin semnificativi 16 biţi ***AD_{0-15} sunt partajaţi*** – termenul folosit este „multiplexaţi în timp", adică sunt folosiţi în comun, dar nu în același timp, de către ***busul de date*** și ***cel de adresă***.
Deoarece sunt 16 biţi care circulă pe BD, operaţiile se pot realiza doar la nivel de 16 biţi; faptul că sunt 20 biţi pt adresă oferă posibilitatea ca ***dimensiunea memoriei (la adresare) să fie de 1MB***
(1 Mega Octet – adică 10^6 octeţi sau mai bine, echivalent în puteri binare cu 2^{20}).
Termenul corect ar fi fost MebiB și nu MegaB, întrucât diferenţa între 10^6 și 2^{20} nu este neglijabilă (48.576 octeţi).

3.3.3. Setul de regiştri

Procesoarele din familia 80x86 au ***3 categorii*** principale de *regiștri*:

I. regiștri de UZ GENERAL (care pot fi de date, pointer sau index)
- regiștrii de date în general sunt identificaţi prin litera X de la sfârșit, regiștrii pointer - *litera P, și regiștrii* index - *litera I*, toţi aceștia fiind numiţi și GPR (General Purpose Registers în engleză);
II. regiștri SEGMENT (identificabili ușor prin *litera S*);
III. regiștri SPECIALI: cuprinde registrul pointer la instrucţiune (IP) și registrul indicatorilor de condiţii (PSW).
Setul de regiștri al fiecărui procesor este de fapt un *superset* al procesoarelor apărute anterior; de exemplu, procesorul 386 are în componenţă toţi regiștrii pe care îi avea 8086 la momentul respectiv și în plus, mai are și acei regiștri specifici lui.

Tabelul 3.1. Regiștrii procesorului 8086

Registru	Nr biţi	Nume Registru	Apărut de la
Regiștri de uz general (GPR)			
Date	8	AL,BL,CL,DL,AH,BH,CH,DH	8086↑
	16	AX,BX,CX,DX	8086↑
Pointer	16	SP,BP	8086↑
Index	16	SI,DI	8086↑
Regiștri segment			
Segment	16	CS,DS,SS,ES	8086↑
Regiștri speciali			
Pointer la instrucţiune	16	IP	8086↑
Indicatori de condiţii	16	Flags	8086↑

I. Cei **8 *regiştri de uz general*** (**AX, BX, CX, DX, SI, DI, BP, SP**): mărimea lor fiind de 16 biţi, ei pot păstra numere fără semn în domeniul 0...65535 sau numere cu semn în domeniul –32768...+32767.

Deoarece accesul la regiştri este mai rapid (decât la locaţiile de memorie), aceştia se folosesc mai intens ca locaţii temporare; reamintesc aici că instrucţiunile care folosesc regiştri se execută mai rapid, lungimea lor este mai mică.

Regiştrii ***AL, BL, CL, DL*** *şi* ***AH, BH, CH, DH*** sunt părţile low şi respectiv high ale regiştrilor corespunzători pe 16 biţi şi pot fi accesaţi chiar şi aşa, doar pe 8 biţi.

Regiştrii DI şi SI sunt regiştri index (Destinaţion Index şi Source Index) destinaţi lucrului cu şiruri de octeţi sau cuvinte. Aceştia nu pot fi accesaţi decât pe 16 biţi.

Regiştrii SP şi BP sunt regiştri destinaţi lucrului cu stiva și la fel, pot fi accesaţi doar ca 16 biţi deodată. Stiva se defineşte ca o zonă de memorie (LIFO–Last în First Out) în care pot fi depuse valori, extragerea lor ulterioară realizându-se în ordine inversă depunerii. Registrul SP (Stack Pointer) pointează spre ultimul element introdus în stivă, iar BP (Base Pointer) către baza stivei.

Noţiunea de stivă poate fi explicată prin analogie cu un sac de haine în care se depun cămăși frumos împăturate. După ce depunem mai multe astfel de cămăși, BP va fi pointer la prima cămașă depusă în sac, iar SP va fi pointer la ultima. Nu e permis să stricăm ordinea din sac, astfel că toate articolele vor fi extrase din sac doar în ordine inversă depunerii lor (luăm de deasupra).

II. Procesorul 8086 conţine ***4 regiştri segment*** pe 16 biţi (**CS, DS, ES** şi **SS**) ce se folosesc pentru a selecta blocuri (numite segmente) din memorie. Arhitectura I8086 permite deci existenţa a patru tipuri de segmente: segment de cod, segment de date, segment de stivă şi segment de date suplimentar / extrasegment.

Regiştrii CS, DS, SS şi ES din BIU reţin adresele de început ale segmentelor active, corespunzătoare fiecărui tip de segment. Registrul IP conţine offsetul instrucţiunii curente în cadrul segmentului de cod (CS) curent, el fiind manipulat exclusiv de către BIU. Astfel, programatorul nu poate și nici nu trebuie să modifice conţinutul lui IP.

III. ***Regiştrii cu scop special*** sunt **IP** şi **PSW**.

Registrul IP conţine adresa instrucţiunii curente care se execută și se mai numeşte şi PC (Program Counter). După ce s-a executat instrucţiunea curentă, IP este incremen-tat *automat* pentru a pointa spre *instrucţiunea următoare*. Instrucţiunile de salt modifică valoarea acestui registru astfel încât execuţia programului se mută la o nouă poziţie (care nu se cunoaște în general decât atunci când se ajunge în situaţia de a realiza saltul; în plus, nu se poate prezice dinainte dacă se execută sau nu saltul respectiv).

Tabelul 3.2. Adrese specifice în 8086

Adresa logică	Segment	Offset Implicit	Semnificaţie
CS:IP	CS	IP	Adresa instrucţiunii curente
SS:SP	SS	SP	Adresa ultimului element introdus în stivă
SS:BP	SS	BP	Adresa primului element introdus în stivă
DS:offset	DS	BX sau nimic, DI sau SI, (+deplasament)	Adresa datei

Registrul *PSW* (Program Status Word) conţine 9 flaguri (1 bit fiecare flag) ce raportea-ză *starea CPU* la momentul curent, deci după execuţia fiecărei instrucţiuni.

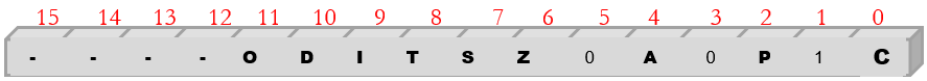

Figura 3.7 Conţinutul registrului PSW (Flags)

Aceşti regiştri cu scop special nu pot fi accesaţi direct, ei fiind modificaţi automat de către CPU pe durata execuţiei instrucţiunii.

PSW sau Flags conţine indicatori de condiţie (bistabili) a căror stare se modifică în urma execuţiei unor instrucţiuni; totuși, nu toate instrucţiunile afectează aceste flaguri, unele putând chiar altera valoarea lor în mod nespecific.

Indicatorii „de stare”și „de control”

Indicatorii „de stare” arată *starea/ situaţia* în care a ajuns UCP în urma execuţiei unei instrucţiuni; așa cum rezultă și din Figura 3.7 aceștia sunt implementaţi în Flags pe poziţiile 0,2,4,6,7 și 11 ca fiind C, P, A, Z, S, O și sunt detaliaţi în continuare:

CF ***(Carry Flag)*** este flagul care arată un posibil transport: acesta are valoarea 1 în cazul în care în cadrul ultimei operaţii efectuate a apărut un transport (carry) sau un împrumut (borrow) în/ din afara domeniului de reprezentare al rezultatului și valoarea 0 în caz contrar;
PF ***(Parity Flag)*** are valoarea astfel încât împreună cu numărul de biţi de 1 din reprezentarea rezultatului, să existe un număr impar de cifre 1. Altfel spus, flagul se setează dacă numărul de biţi de 1 ai rezultatului este un număr par;

AF (Auxiliary Flag) indică valoarea transportului (carry/ borrow) de la bitul b3 la bitul b4 al rezultatului obţinut în urma calculelor - folosit în special la cifrele BCD;

ZF (Zero Flag) are valoarea 1 dacă rezultatul ultimei operaţii este egal cu zero şi valoarea 0 dacă s-a obţinut un rezultat diferit de zero; menţionat în *secţiunea 2.2.6*;

SF (Sign Flag) are valoarea 1 dacă rezultatul ultimei operaţii este un număr strict negativ şi valoarea 0 în caz contrar; adică va copia bitul MSb al rezultatului.

OF (Overflow Flag) indică depăşire de gamă: dacă rezultatul ultimei instrucţiuni nu a încăput în gama alocată pentru stocarea numărului în cazul operanzilor consideraţi numere cu semn (dacă s-a modificat semnul rezultatului), atunci acest flag va avea valoarea 1; altfel, va avea valoarea 0 (apare în *secţiunea 2.2.6*). Pentru a determina valoarea flagului Overflow, se poate scrie o formulă pe baza flagului Carry și a unui posibil transport între bitul MSb-1 și bitul MSb; în general se folosește relaţia:

$$O=C \oplus transport_{MSb-1,\ MSb}$$

Indicatorii „de control" se folosesc pt *a controla* modul de acţiune al procesorului. Aceștia se regăsesc în registrul ***Flags*** pe biţii 8,9,10, așa cum reiese din Figura 3.6:

T – Trap – flag pentru depanare; dacă are valoarea 1, atunci procesorul se opreşte după fiecare instrucţiune, permiţând execuţia pas cu pas (la depanare);

I – Interrupt – flag de întrerupere ce controlează răspunsul CPU la cereri de întreruperi mascabile: permite (dacă IF=1) sau invalidează (dacă IF=0) acceptarea întreruperilor externe mascabile care apar pe o intrare specială (pe pinul INT) a procesorului; acest flag nu afectează întreruperile interne sau pe cele externe nemascabile;

D - Direction – flag folosit la lucrul cu şiruri pentru a indica direcţia de parcurgere de-a lungul şirului, (sau direcţia de deplasare la adresarea pe șiruri)

D=0 pentru adrese crescătoare,
D=1 pentru adrese descrescătoare.

Trap și Interrupt se mai numesc și flaguri *de sistem*.
Începând cu 80286, au fost implementaţi și indicatorii de sistem de pe biţii 12, 13 (IOPL) și 14 (NT), iar începând de la procesoarele 80386, registrul indicatorilor de condiţii a fost extins la 32 biţi (și se numește EFLAGS).

Exemple: după modelul exerciţiilor prezentate anterior, analizaţi următoarele operaţii și efectul lor; realizaţi calculele în binar după modelul prezentat:
Model: operaţiile se consideră pe 8 biţi; calculul 02h+02h=04h, în binar se scrie:
0000 0010b +
<u>0000 0010b</u>
0000 0100b, unde biţii se adună poziţie cu poziţie, iar prin însumarea 1+1 a rezultat un bit de 0 și o unitate (ca bază) s-a transportat mai departe, înspre stânga, exact ca la algoritmul aplicat în clasele primare când adunam în zecimal și transportam un 1 (adică o unitate, și se spunea „1 ducem mai departe"). Similar se procedează și în cazul scăderii în binar, doar că în acest caz se va realiza un împrumut de la cifra dinspre stânga înspre cea din dreapta („aducem un 1", adică o unitate care va fi 2 aici).
a) 02h+02h=04h -> C=0, Z=0, S=0, O=0
b) 03h+7Ch=7Fh -> C=0, Z=0, S=0, O=0
c) 04h+7Ch=80h -> C=0, Z=0, S=1, O=1
unde $O=C \oplus transport_{MSb-1,\ MSb}=0+1=1$
d) 80h+80h=00h -> C=1, Z=1, S=0, O=1
unde $O=C \oplus transport_{MSb-1,\ MSb}=1+0=1$

Pentru numerele reprezentate în convenţia fără semn, indicatorul Carry joacă un rol foarte important în cadrul operaţiilor aritmetice; acesta este folosit ca Borrow flag la operaţii de scădere. De exemplu, când se vor compara doi operanzi folosind instrucţiunea ***cmp opStânga, opDreapta*** se realizează de fapt o scădere a celor doi operanzi și se analizează rezultatul obţinut:
dacă opStânga – opDreapta = 0, atunci se setează ZF=1 și CF este lăsat pe 0, CF=0.
dacă opStânga – opDreapta > 0 atunci ZF=0 și CF=0, iar
dacă opStânga – opDreapta < 0, atunci ZF=0 și CF=1 (a fost împrumut - borrow).
Pentru numerele reprezentate în convenţia cu semn, indicatorii Carry și Overflow au un rol important; de asemenea, SF va interveni pentru a arăta semnul rezultatului. Ca în analiza precedentă,
dacă opStânga – opDreapta = 0, atunci ZF=1 și CF este lăsat pe 0;
dacă opStânga – opDreapta > 0 atunci ZF=0, CF=0, iar OF=SF;
dacă opStânga – opDreapta < 0, atunci ZF=0, CF=1 și OF=not SF.

De analizat: (exerciții rezolvate)

Ex1. Să se determine rezultatul următoarelor secvențe de instrucțiuni:

```
mov AL, 10          mov AL, -10
mov BL, 20          mov BL, -20
add AL,BL           add AL,BL
```

Răspuns: ***Instrucţiunea ADD*** adună doi operanzi, rezultatul fiind apoi plasat în primul operand. În acest caz, va trebui să fim atenţi la dimensiunea operanzilor şi la valorile acestora, pentru a nu rezulta depăşiri de capacitate. Adunarea a două valori pe octet generează un rezultat corect în exemplele de mai jos:

a) valori pozitive

```
mov AL, 10  ; AL=10d=0Ah
mov BL, 20  ; BL=20d=14h
add AL, BL  ; AL=30d=1Eh
```

Valoarea flagurilor:
CF=0, ZF=0, SF=0, OF=0, AF=0, PF=1

b) valori negative

```
mov AL, -10  ; AL=-10d=F6h
mov BL, -20  ; BL=-20d=ECh
add AL, BL   ; AL=-30d=E2h
```

Valoarea flagurilor:
CF=1, ZF=0, SF=1, OF=0, AF=1, PF=0

Ex2. Să se determine rezultatul următoarelor secvențe de instrucțiuni:

ca numere fără semn:

```
mov AL, 120
mov BL, 100
add AL, BL
```

ca nr. cu semn:

```
mov AL, 120
mov BL, 100
add AL, BL
```

ca nr. cu semn:

```
mov AL, -120
mov BL, -100
add AL, BL
```

Răspuns: În exemplele date, adunarea va fi corectă dacă vom considera numerele fără semn (rezultatul va încăpea în registrul de 8 biţi), dar va genera o depășire de capacitate dacă vom considera numerele cu semn, deci în acest caz rezultatul nu va fi unul corect:

a) numere fără semn

```
mov AL, 120  ; AL=120d=78h
mov BL, 100  ; BL=100d=64h
add L,BL     ; AL=220d=DCh
```

Valoarea flagurilor:
CF=0, ZF=0, SF=1, OF=1, AF=0, PF=0

b) nr. cu semn, valori pozitive

```
mov AL, 120  ; AL=120d=78h
mov BL, 100  ; BL=100d=64h
add AL,BL    ; AL=DCh=-36d
```

*<rezultat incorect> dar citit ca **0**DCh va fi corect, unde **0**=CF*

Valoarea flagurilor:
CF=0, ZF=0, SF=1, OF=1, AF=0, PF=0

c) nr. cu semn, valori negative

```
mov AL, -120  ; AL=-120d=88h
mov BL, -100  ; BL=-100d=9Ch
add AL,BL     ; AL=36d=24h
```

*<rezultat incorect>, dar citit ca **1**24h va fi corect, unde **1**=CF*

Valoarea flagurilor:
CF=1, ZF=0, SF=0, OF=1, AF=1, PF=1

Obs:

La interpretarea numerelor *fără semn*,
în caz că apare ***CF=1***:
CF *va trebui interpretat ca*
al n+1 -lea bit *al rezultatului*
(pt ca rezultatul să fie corect)

La interpretarea numerelor *cu semn*,
în caz că apare ***OF=1***:
CF *va trebui interpretat ca*
al n+1 -lea bit *al rezultatului*
(pt ca rezultatul să fie corect)

4. Organizarea şi adresarea memoriei

Datorită dimensiunii de 16 biţi a regiştrilor interni ai UCP, nu se poate manevra informaţie (indiferent că vorbim de date sau adrese) mai mare de 16 biţi în mod direct, decât dacă se apelează la diverse artificii. Astfel, fiecare aplicaţie (program aflat în memorie) are la dispoziţie un spaţiu maxim de 64KB pentru codul instrucţiunilor (segmentul de cod), 64 KB pentru stivă (segment de stivă) şi 128 KB pentru date (segmentul de date şi extra segmentul). Unele aplicaţii pot însă gestiona un spaţiu de memorie mult mai mare, manevrând segmentele după propriile necesităţi.

Împărţirea memoriei în segmente de 64KB provine din faptul că microprocesoarele pe 8 biţi anterioare gestionau un spaţiu de numai 64KB. Proiectanţii de la Intel au căutat ca şi noile microprocesoare de la vremea aceea (cele pe 16 biţi) să poată executa programe scrise pentru microprocesoarele anterioare, dorind astfel să asigure compatibilitatea oricărui procesor nou cu modelul anterior ("backward compatibility"); astfel, s-a ajuns la adoptarea acestei soluţii a segmentării logice a memoriei.

Calculatorul de referinţă al IBM a fost lansat pe piaţă în anul 1981 (calculatorul personal IBM-PC/XT) având o versiune de I8086 mai ieftină şi memorie de 1 MB. IBM Personal Computer, prescurtat IBM PC, a fost primul calculator personal care a fost produs, acesta fiind versiunea originală și precursoare a tuturor platformelor compatible PC din prezent; toate variantele de calculatoare care au apărut ulterior au păstrat din considerente de compatibilitate împărţirea memoriei ca la IBM-PC/XT.

4.1. Moduri de adresare a memoriei

Dată fiind dimensiunea memoriei de 1 Moctet la I8086, o adresă trebuie să se reprezinte pe 20 de biţi (2^{20}=1Moctet), dar capacitatea regiştrilor şi a cuvintelor este de 16 biţi. Pentru rezolvarea situaţiei de a nu putea accesa mai mult de 2^{16}=64Kocteţi o dată, a apărut conceptul de ***segment de memorie***, respectiv ***adresarea segmentată***. Acest mod de gestionare a memoriei este unul simplu, constă în utilizarea registrelor de segment şi a primit ulterior numele de ***adresare în mod real***.

Începând cu procesorul 80286, a fost implementat un nou mod de adresare, și anume ***modul protejat***, iar odată cu apariţia 80386 au mai apărut încă două moduri de adresare: ***modul paginat*** și ***modul virtual 8086***, toate acestea fiind introduse pentru a permite adresarea de către IBM PC a mai mult de 1 MB de memorie. Ulterior, de la procesoarele pe 64 biţi a apărut și ***modul long,*** cu 2 submoduri: *submodul pe 64 biţi* și *submodul compatibilitate*.

4.2. Adresarea memoriei la 8086. Adresarea segmentată

Revenind la modul de adresare al procesoarelor 8086, prin definiţie, *adresa unei locaţii de memorie* este numărul de ordine între începutul memoriei RAM (având dimensiunea de 1MB în cazul sistemelor cu 8086) şi locaţia respectivă.

Pentru a adresa toate locaţiile de memorie de până la 1 Moctet, adresele trebuie scrise pe 20 biţi, deci cu 5 cifre hexazecimale, şi nu pe doar 16 biţi cât este capacitatea regiştrilor şi a cuvintelor la I8086. Această valoare pe 20 biţi se numește ***adresă fizică*** și identifică în mod unic fiecare locaţie din spaţiul de adresare de 1 MB. Adresa fizică (scrisă deci pe 20 biţi) se găseşte în domeniul 00000_h-$FFFFF_h$ şi se mai numeşte adresă absolută. Pentru a nu depinde de locul unde se află codul în memorie, se folosesc aşa-numitele adrese logice, diferite de cele fizice. **Adresa logică** este scrisă tot cu 20 biţi, dar de fapt aceasta se constituie dintr-o valoare de segment şi o valoare de offset; aceasta se scrie ca o combinaţie de două valori pe 16 biţi:

- una pentru a stabili începutul segmentului (reţinută într-un registru segment) şi

- una numită offset (deplasament) pentru a reţine locaţiile de memorie din interiorul acelui segment, așa cum sugerează Figura 3.9.

Segmentul de memorie reprezintă astfel o succesiune continuă de octeţi care începe la o adresă multiplu de 16 (numită paragraf) şi are lungimea multiplu de 16 octeţi (maximum 64 Kocteţi). Pentru orice locaţie de memorie, valoarea de bază a segmentului este adresa primului octet al segmentului care conţine locaţia respectivă. Deoarece adresa de început a fiecărui segment este multiplu de 16, cei mai puţin semnificativi 4 biţi ai acestei adrese sunt zero (deci ultima cifră hexazecimală este 0).

Offset-ul/ deplasamentul reprezintă ***adresa unei locaţii faţă de (sau raportat la) începutul segmentului***, sau altfel spus, ***offsetul din interiorul acelui segment***; este totodată distanţa în octeţi de la începutul segmentului până la locaţia respectivă.

Adresa de bază şi deplasamentul sunt valori pe 16 biţi fără semn, așa cum se poate deduce din Figura 3.10.

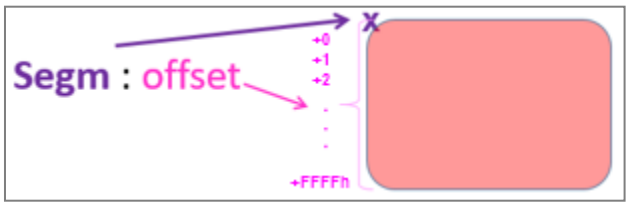

Figura 3.10 Ilustrarea noţiunilor de segment și offset în cadrul memoriei la 8086

Adresa logică este o pereche de numere pe câte 16 biţi fiecare, unul reprezentând adresa de început a segmentului (dată de RS - registrul segment), iar celălalt reprezentând offsetul în cadrul segmentului; se foloseşte scrierea:

Adresa Logică = RS:offset (1)

Adresa fizică reprezintă locaţia sau zona de memorie fizică, este pe 20 de biţi şi se obţine din configuraţia de 16 biţi care localizează începutul segmentului înmulţită cu 16 (prin deplasare spre stânga cu 4 poziţii binare, adică o cifră hexa) la care se adună valoarea offsetului aşa cum se poate urmări în Figura 3.11.

Reveniţi la Figura 3.5 și localizaţi pe schemă acest bloc de calcul al adresei, în partea dreaptă sus, în blocul BIU.

Calculul din relaţia (2) este efectuat de unitatea de adresare din BIU: aceasta generează întotdeauna o adresă fizică dintr-o adresă logică, după mecanismul prezentat detaliat în Figura 3.10 și care implementează regula:

$$\text{Adresa Fizică}_{20\text{biți}} = 16 \bullet RS_{16\text{ biți}} + offset_{16\text{ biți}} \qquad (2)$$

Calculul adresei fizice se realizează prin deplasarea bazei segmentului (conţinută într-un registru segment) cu 4 poziţii spre stânga (ceea ce echivalează cu o înmulţire cu 16 sau 10h) şi adunarea valorii deplasamentului.

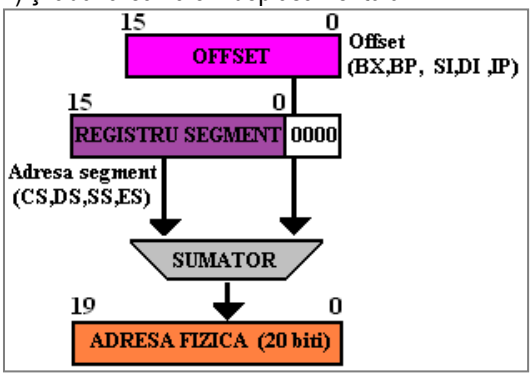

Figura 3.11. Calculul adresei fizice de către BIU

Exemplu: Dacă CS este 2104h şi IP este 2000h, atunci:

- adresa logică se va scrie 2104h:2000h, unde registrul segment conţine valoarea 2104h şi offsetul este 2000h;
- adresa fizică se va scrie 23040h, obţinută prin însumarea: 2104**0**h+2000h.

Mai multe adrese logice pot corespunde aceleiaşi locaţii fizice dacă se află în segmente diferite, ceea ce poate crea anumite probleme de inconsistenţă a datelor. Altfel spus, o locaţie fizică poate să aparţină unuia sau mai multor segmente, deoarece mai multe adrese logice (folosind segmente diferite) pot conduce înspre aceeaşi adresă fizică.

Exemplu:

Adresa logică (hexa)	Adresa fizică (hexa)
2000:3040	23040
2104:2000	23040
2200:1040	23040
2302:0020	23040

Reamintesc aici că segmentele curente oferă un spaţiu de lucru de 64 Kocteţi de cod, 64 Kocteţi de stivă şi 128 Kocteţi de date. Segmentele pot fi adiacente, disjuncte, parţial suprapuse sau suprapuse complet.

4.3. Moduri de adresare

Modul cum ne referim la datele cu care lucrăm desemnează așa numitele *moduri de adresare*. Această adresare se poate referi la operanzi stocaţi în memorie, la operanzi stocaţi în regiștri sau care provin în interiorul sau exteriorul sistemului prin intermediul porturilor. În principal, modurile de adresare sunt împărţite în 2 mari direcţii: adresare directă și adresare indirectă, așa cum se prezintă în Fig. 3.12.

ADRESARE DIRECTA:
- se specifica in mod direct *registrul*, *valoarea imediata* **sau** *zona din memorie*

```
mov AX, BX          ; cu registru
mov AX, 1234h       ; cu o constanta (valoare imediata)
mov AX, zonaMem     ; cu memoria,
                    unde zonaMem a fost definita cu directiva dw
```

ADRESARE INDIRECTA: Adresa = RS : offset [BX/-]=>RS=DS [BP] =>RS=SS

- se folosesc *registrii* pentru a indica adrese din memorie

- iar **offset** poate fi format din 3 categorii: deplasament + [BX]/[BP] + [SI]/[DI]

```
mov AX, [BX]        ; bazata fara deplasament
mov AX, [SI]        ; indexata fara deplasament
mov AX, [BX][SI]    ; bazata-indexata fara deplasament
mov AX, [BX][SI][3] ; bazata-indexata cu deplasament
```

Figura 3.12. Principalele moduri de adresare la 8086

Instrucţiunile care au doi sau mai mulţi operanzi operează întotdeauna de la dreapta spre stânga. Astfel, operandul din dreapta se numeşte operand sursă (specifică datele care vor fi folosite, dar nu şi modificate), iar operandul din stânga se numeşte operand destinaţie (specifică datele care vor fi folosite şi modificate de către o anumită instrucţiune). Datele imediate (constantele) nu sunt admise ca operand destinaţie.

Exemplu: *mov opStanga, opDreapta*; în acest caz din operandul din dreapta (numit sursă) se va copia informaţia în operandul din stânga, numit destinaţie; instrucţiunea ***mov*** provine prin prescurtarea (sub formă de mnemonică) de la move (a muta).

Exemplu: mov 2, AX – este ilegală, întrucât 2 este o constantă; nu putem muta într-o constantă (nu e localizată nici în vreun registru, nici în memorie) o cantitate de 16 biţi.

În cadrul unei instrucţiuni există mai multe **moduri de a calcula adresa efectivă** sau offsetul unui operand pe care aceasta îl solicită; astfel, se poate vorbi de mai multe **tipuri de operanzi.** Operanzii pot fi 1) imediaţi sau pot fi conţinuţi 2) în regiştrii, 3) în porturile de intrare/ ieşire sau 4) în memorie.

4.3.1. Adresarea imediată, cu regiştrii şi cu porturile

Regiştri folosiţi şi modul de adresare (memorie sau registru) sunt codificaţi în interiorul instrucţiunii. În practică există următoarele tipuri de adresare:

Adresare directă

1d. **Cu operand imediat -** atunci când operandul (numit și *imediat* sau *constantă*) este specificat chiar în instrucţiune.

```
mov AX,1234h     ; unde 1234h este operand imediat
```

2d. **Cu operand registru –** în instrucţiune participă valoarea stocată în registru.

```
mov BX,5
mov AX, BX       ; atât operandul sursă cât și cel destinaţie sunt regiștri – operand registru se referă la valoarea 5 stocată în registrul BX;
```

3d. **Cu operand de tip adresă port** (pentru adrese pe 8 biţi, în gama 0..255)
Operandul se află în portul de la adresa specificată în instrucţiune după codul operaţiei.

```
in AX, 10h       ; operandul sursă este portul de intrare aflat la adresa numerică 10h
out PORT1, AL    ; operandul destinaţie este portul de ieşire aflat la adresa PORT1
```

Adresare indirectă

1i. **Adresare indirectă** prin valoare imediată

```
mov DOI, AX      ; operandul destinaţie este o constantă simbolică DOI de tip imediat
```

2i. **Adresare indirectă** prin regiştri - offsetul este furnizat de unul dintre regiştrii BP, BX, SI sau DI, sau o constantă numită deplasament; registrul segment implicit este DS sau SS, în funcţie de regula de la adresarea indirectă cu memoria.

```
mov AL,[BX]   ; registrul segment este DS, iar offsetul este specificat de conţinutul reg. BX. Mai multe detalii se vor preciza în secţiunea următoare la adresarea cu memoria.
```

3i. **Adresarea indirectă a porturilor prin registrul DX**
Operandul se află în portul de la adresa specificată în registrul DX.

```
out DX, AL       ;operandul destinaţie este portul de ieşire a cărui adresă se află în DX
```

Observaţie: spaţiul de adrese al porturilor este 0...FFFFh (64kocteţi).

4.3.2. Adresarea operanzilor din memorie

Datele din memorie care formează operanzii instrucţiunilor pot fi adresate în mai multe moduri. Operaţiile care implică date numai din regiştri sunt cele mai rapide, nefiind nevoie de utilizarea magistralei pentru acces la memorie (acces cu consum mai mare de resurse fizice și de timp). Regiştrii folosiţi şi modul de adresare (memorie sau registru) sunt codificaţi în interiorul instrucţiunii. În practică există mai multe tipuri de adresare, așa cum am văzut anterior, dar în cele ce urmează ne vom referi doar la cele care folosesc memoria:

Adresarea cu memoria: dacă operandul se află undeva în memorie, va fi necesar transferul pe bus: operanzii din memorie sunt accesaţi mai lent, deoarece mai întâi se calculează adresa efectivă (de către BIU) a operandului, apoi se calculează adresa fizică a acestuia şi în final se transferă datele.

a) Operandul cu adresare directă la memorie – operandul cu adresare directă este o constantă sau un simbol care reprezintă adresa unei instrucţiuni sau a unor date. Deplasamentul unui operand cu adresare directă este calculat în momentul asamblării programului, dar adresa fiecărui operand raportată la structura programului este calculată în momentul editării de legături. Adresa efectivă, care este întotdeauna raportată la un registru de segment, este calculată în momentul încărcării programului pentru execuţie.
Exemplu: S-a definit o variabilă octet în memorie, neiniţializată folosind directiva:
a db ?, adică un octet definit în segm. de date, având numele *a;* acesta va fi un operand cu adresare directă la memorie.
mov a, 2

Exemplu: Calculaţi adresa fizică şi conţinutul locaţiei respective de memorie după execuţia următoarelor instrucţiuni, presupunând DS=1470h:

mov AL, 50h ; registrul AL se încarcă cu valoarea 50h

mov [4320h], AL ; adresa fizică va fi AF = 18A20h, deci [18A20h] = 50h.

Observaţie: Nu sunt admise operaţiile pentru care atât sursa cât şi destinaţia sunt operanzi din memorie.

b) Operand cu adresare indirectă la memorie - operanzii cu adresare indirectă utilizează regiştri pentru a indica adrese din memorie. Acest mod de adresare este folosit în manipularea dinamică a datelor (pt că valorile din regiştri se pot modifica).

În cazul microprocesoarelor I8086 numai patru regiştri pot fi folosiţi în adresarea indirectă: regiştrii de bază **BX** şi **BP** şi regiştrii index **SI** şi **DI**. Regiştrii de bază sau index pot fi folosiţi împreună sau separat, cu sau fără specificarea unui deplasament, fiecare dintre cei trei termeni din relaţia (3) fiind opţional. Astfel, adresa efectivă a unui operand din memorie se poate scrie sub forma:

$$AE=[\ BX\ |\ BP\ ]_{opt} + [\ DI\ |\ SI\ ]_{opt} + [\text{deplasament 8/16 biţi}]_{opt} \quad (3)$$

unde specificarea "opt" arată că un termen este opţional, iar semnul "|" specifică faptul că doar unul dintre cei doi regiştri de bază (***BX*** sau ***BP***) respectiv index (***DI*** sau ***SI***) se va putea folosi la adresare.

În Figura 3.13 sunt prezentate diferite **moduri de calcul a adresei fizice de memorie.**

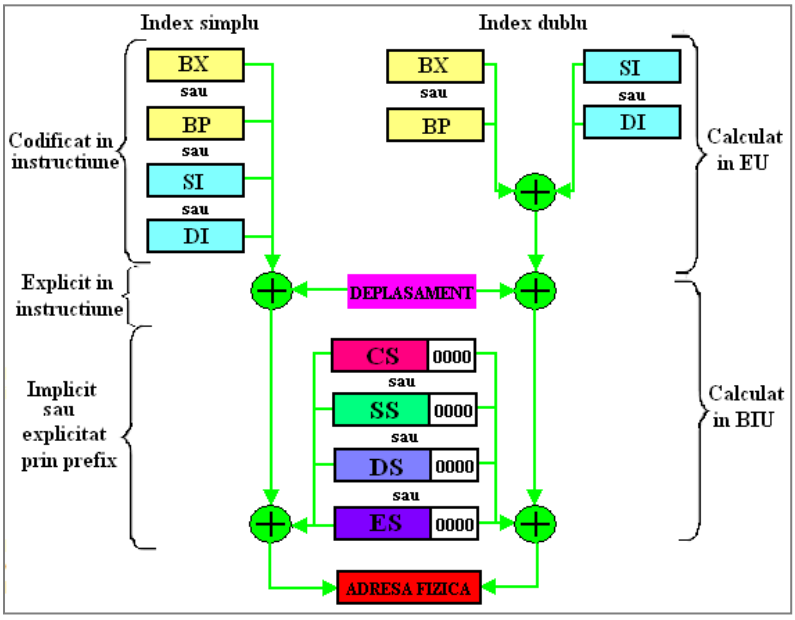

Figura 3.13. Variante de calcul a adresei fizice de memorie

Există 2 posibilităţi:

I) Dacă ***BX*** e folosit ca registru de bază sau dacă ***nu este specificat nici un registru de bază***, la calculul adresei efective a unui operand cu adresare indirectă

registrul segment implicit va fi ***DS***.

II) Dacă ***BP*** e fol. oriunde în calculul adresei operandului, ***segmentul implicit*** va fi ***SS***.

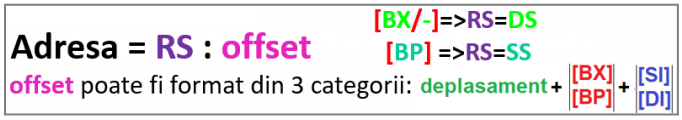

Figura 3.14. Regula de obţinere a **registrului segment** şi a **offsetului** la calculul adresei operandului din memorie

Se permit diverse modalităţi de a specifica operanzi cu adresare indirectă, folosind orice operator care indică adunarea (plus +, paranteze drepte [ ] sau punct .). Esenţială este specificarea între paranteze drepte a cel puţin unuia dintre elementele componente ale formei prezentate.

Exemplu: Următoarele moduri de specificare sunt echivalente:

sir[BX][DI]+2, 2+sir[BX+DI], [sir+BX+DI]+2, sir[BX+2] [DI] ; operandul este din memorie, din segmentul de date curent, aflat la offset-ul egal cu suma conţinuturilor registrelor BX şi DI plus 2, cu deplasarea dată de offset SIR.

Observaţie: Când se utilizează modul de adresare bazat-indexat, unul dintre regiştri trebuie să fie registru de bază, iar celălalt să fie registru index.

Exemple: Următoarele instrucţiuni sunt **incorecte:**

mov AX, SIR [BX] [BP] ; doi regiştri de bază

mov AX, SIR [SI] [DI] ; doi regiştri index

Există mai multe tipuri de adresare:

- **adresare bazată sau indexată (*index simplu*) -** ofsetul se obţine adunând la unul din regiştrii de bază (BX sau BP) sau index (SI sau DI) un deplasament constant.

Exemple: mov AX,[BX+5]
mov AX,[SI+4]

- **adresare bazată şi indexată (*index dublu*) -** este cea mai complexă formă de adresare, care combină variantele anterioare, permiţând adresarea cu 2 indecşi (conţinuturile regiştrilor de bază şi index la procesoarele pe 16 biţi).

Exemplu: mov AX,[BX+SI+7]

Daca se foloseşte BP, atunci registrul de segment implicit este SS.

Dacă se doreşte folosirea altui registru de segment decât cel implicit, în instrucţiune se poate preciza în mod explicit despre care registru de segment este vorba.

Exemplu: mov BX,ES:[BP+2] ; dacă nu s-ar fi precizat ES, implicit se folosea SS, deoarece registrul de bază este BP.

Tema LAB 03 – partea a 2-a

1. a) Realizaţi următoarele operaţii la nivel de 8 biţi. Scrieţi valorile în zecimal considerându-le numere *fără semn* și analizaţi rezultatul obţinut pe cei 8 biţi. Specificaţi și valoarea flagurilor aritmetice (realizaţi operaţia în binar).
R1, R2) 37h + 6Dh = _____d + _____d = _____h; C=__; Z=__; S=__; A=__; O=__.
R3, R4) A7h + 6Dh = _____d + _____d = _____h; C=__; Z=__; S=__; A=__; O=__.

b) Realizaţi următoarele operaţii la nivel de 8 biţi. Scrieţi valorile în zecimal considerându-le numere *cu semn* și analizaţi rezultatul obţinut pe cei 8 biţi. Specificaţi și valoarea flagurilor aritmetice.
R1, R2) 37h + 6Dh = _____d + _____d = _____h; C=__; Z=__; S=__; A=__; O=__.
R3, R4) A7h + 6Dh = _____d + _____d = _____h; C=__; Z=__; S=__; A=__; O=__.

2. a) Calculaţi adresa fizică ce corespunde adresei logice
R1) 89ABh:89ABh, **R2**) 56CDh:56CDh, **R3**) 43EFh:43EFh, **R4**) 12ABh:12ABh;

b) Daţi exemplu de 3 adrese logice ce se pot scrie pentru adresa fizică următoare:
R1) 23456h; **R2**) 65432h, **R3**) 87654h, **R4**) 45678h.

c) Să se calculeze componenta offset corespunzătoare adresei fizice menţionate dacă se cunoaște componenta segment:
R1) AF=10000h, segment=1000h; offset=_____h;
Dar dacă AF= 1FFFFh ? offset=_____h
R2) AF=20000h, segment=2000h; offset=_____h;
Dar dacă AF= 2FFFFh ? offset=_____h
R3) AF=30000h,segment=3000h; offset=_____h;
Dar dacă AF= 3FFFFh ? offset=_____h
R4) AF=40000h,segment=4000h; offset=_____h;
Dar dacă AF= 4FFFFh ? offset=_____h

d) Să se calculeze componenta segment corespunzătoare adresei fizice 0ABC10h dacă se precizează componenta offset:
R1) offset = 600h; segment= _____h; **R2**) offset= 800h; segment= _____h;
R3) offset = 0A00h; segment= _____h; **R4**) offset= 900h; segment= _____h;

e) Verificaţi dacă adresa fizică menţionată aparţine segmentului pentru care se știe componenta segment:
R1) AF=31FFFh, segm.= 2200h; R:_____ Dar adresa fizică 21000h? R:_____
R2) AF=41FFFh, segm.= 3200h; R:_____ Dar adresa fizică 31000h? R:_____
R3) AF=51FFFh, segm.= 4200h; R:_____ Dar adresa fizică 41000h? R:_____
R4) AF=61FFFh, segm.= 5200h; R:_____ Dar adresa fizică 51000h? R:_____

3. a) Pentru DS=1200h, SI=1234h, DI=4321h, AX=2ABCh şi BX=1A3Bh specificaţi adresa locaţiei de memorie accesată de fiecare dintre instrucţiunile următoare:
R1) mov [DI], AX ; AF=_____ **R2**) mov AX, [SI+BX] ; AF=_____
R3) mov AX, [SI+BX+2] ; AF=_____ **R4**) mov [DI+5], AX ; AF=_____

b) Precizaţi dacă următoarele instrucţiuni sunt corecte şi justificaţi răspunsul:

R1) a) mov AX, [BP+BX] b) mov AL, [BX]
R2) a) AX, [BP+SI] b) mov AX, 7[BP]
R3) a) mov [SI+DI], AX b) mov [SI+BP], AX
R4) a) mov AX, [BP+5] b) mov AX, [SI+DI]

c) **R1**), **R2**) Ştiind că SS=2F00h, CS=1234h, IP=2300h şi SP=3456h, specificaţi adresele logice şi fizice pt elementul din vârful stivei şi pt instrucţiunea curentă.

R3), **R4**) Ştiind că SS=3F00h, CS=1200h, IP=2000h şi SP=2002h, specificaţi adresele logice şi fizice pt elementul din vârful stivei şi pt instrucţiunea curentă.

4. Pentru SS=3F00h, CS=5A00h, ES=1400h, DS=1200h, SI=1234h, DI=340Ch, AX=2ABCh şi BX=1A3Bh, BP=1300h, menţionaţi tipul de adresare folosit şi specificaţi adresa locaţiei de memorie accesată de fiecare dintre instrucţiunile următoare, folosind modelul:
mov AX, [BX] ; s-a folosit adresarea bazată, primul operand este registrul AX, iar al doilea operand este perechea de octeţi (cuvântul) din memoria principală, din segmentul curent de date (specificat de DS), aflat la offset-ul conţinut în registrul BX (adică operandul este cuvântul din memorie, de la adresa 1200h : 1A3Bh (partea LOW) şi 1200h : 1A3Ch (partea HIGH)

i) a) mov [DI], AX b) mov AL, [BX+5]
ii) a) mov AX, [SI+BX] b) mov 2[BX] [SI], AL
iii) a) mov AX, DS: [BP+2] b) mov 5[SI], AL
iv) a) mov 2[BP] [DI], AL b) mov AX, [SI+2]

Urmăriţi modelele:
ADD AX, [BX] ; al doilea operand este octetul / perechea de octeţi (cuvântul) din memoria principală, din segmentul curent de date (specificat de DS), aflat la offset-ul dat de conţinutul din registrul BX.
ADD [SI], AL ; primul operand este octetul / perechea de octeti din memoria principală, din segmentul de date curent (specificat de DS), aflat la offsetul conţinut în registrul SI.
ADD AX, DS: [BX+2] ; al doilea operand este cuvântul din memorie, din segmentul de date curent (specificat explicit de DS), aflat la offset-ul egal cu suma dintre conţinutul registrului BX și deplasarea 2 (de tip imediat).
ADD TAB [DI], AL ; primul operand este octetul din memorie, din segmentul de date curent (specificat de DS), aflat la offset-ul egal cu suma dintre conţinutul registrului DI și deplasarea TAB (de tip imediat).
ADD AX, [BX] [SI] ; al doilea operand este cuvântul din memorie, din segmentul de date curent, aflat la offset-ul egal cu suma conţinuturilor registrelor BX și SI.
MOV MATR [BX] [SI], AX ; primul operand este cuvântul din memorie, din segmentul de date curent, aflat la offset-ul egal cu suma conţinuturilor registrelor BX și SI cu deplasarea MATR (de tip offset).

5. I. Comentaţi destinaţia din următoarele instrucţiuni după modelul exerciţiului anterior și precizaţi dacă sunt corecte aceste instrucţiuni; justificaţi răspunsul:
R1) a) mov ax, sir [BP+BX] b) mov AX, 7[BP]
R2) a) mov AX, var [BX] b) mov AX, [SI+DI]
R3) a) mov AX, sir [BP+SI] b) mov AX,[BP+5]
R4) a) mov AX, sir [SI+DI] b) mov AX, [SI+BP]

II. Conţinutul căror locaţii de memorie este mutat în AX în instrucţiunile următoare? Se dă BX=12ABh, BP=1300h, SI=1A2Bh, DI=340Ch, DS=2100h, SS=3F00h, CS=5A00h. Ce mod de adresare este folosit?
R1) a) mov AX, [bx+3], b) mov AX, 4[bx+si]
R2) a) mov AX, [si]+10h, b) mov AX, 5[bp]
R3) a) mov AX, [di+8], b) mov AX,[bx][si]
R4) a) mov AX, 5[bp+si], b) mov AX, [5][di]

6. Scrieţi instrucţiunile corespunzătoare următoarelor operaţii: (toate randurile)
a) în registrul AL să fie mutat conţinutul din memorie de la adresa 23h;
b) conţinutul registrului AL să fie mutat în memorie la adresa 25h;
c) în registrul AX să fie mutat conţinutul din memorie de la adresa 10h;
d) conţinutul registrului AX să fie mutat în memorie la adresa 12h;
e) octetul din memorie, de la locaţia 50h să fie mutat în locaţia de memorie 60h;
f) cuvântul din memorie, începând de la locaţia 20h să fie mutat începând cu locaţia de memorie 40h;