CUPRINS

PREFAŢĂ

Volumul de faţă reprezintă o reeditare adăugită, completată şi îmbunătăţită a lucrării [1], fiind un ghid de lucrări practice, aplicaţii şi chestionare.Lucrarea se adresează în mod special studenţilor de la profilul electric care studiază disciplina de microprocesoare, dar şi tuturor celor interesaţi a se iniţia în programarea în limbajul de asamblare al procesoarelor Intel x86.

Inainte de toate, înţelegerea detaliată a funcţionătii microprocesoarelor fără limbajul de asamblare specific este practic imposibilă. Deşi tendinţa actuală de îndepărtare de limbajul de asambare este într-un fel justificată de anumite inconveniente: este dificil de învăţat şi înţeles, este dificil de scris programe şi de depanat, nu este portabil şi cere cunoaşterea arhitecturii procesorului. Totusi acesta prezintă anumite beneficii incontestabile: programele scrise în limbaj de asamblare sunt mai rapide, ocupă spaţiu mai redus, pot realiza funcţii care în limbajele de nivel înalt sunt dificil de realizat sau chiar imposibil. De asemenea demn de mentionat, legat de limbajele de asamblare, este faptul ca in topul limbajelor de programare folosite la sistemele embedded, limbajele de asamblare se afla in primele 5 limbaje cele mai folosite, conform IEEE Spectrum 2018.

Astfel, aplicaţiile sau secvenţele de program critice din punct de vedere al timpului de execuţie sau care necesită un spaţiu de memorie limitat sunt dezvoltate în limbaj de asamblare. Multe medii de dezvoltare a aplicaţiilor (IDE) şi compilatoare prezintă facilităţi de inserare de linii sursă scrise direct în limbaj de asamblare. Volumul de faţă conţine 15 capitole prin care cei interesaţi sunt iniţiaţi în mod treptat în domeniul programării în limbaj de asamblare şi se pot autoevalua pe baza testelor şi chestionarelor.

Prima lucrare face o introducere în domeniul arhitecturilor de prelucrare şi a procesoarelor Intel. Partea practică tratează reprezentarea numerelor în calculator şi operaţiile aritmetice cu numere reprezentate în diferite formate.

Lucrarea a doua prezintă noţiunile mai importante legate de arhitectura procesorului 8086 necesare pentru înţelegerea aspectelor legate de setul

de instrucţiuni şi abordează de asemenea şi modurile de adresare ale procesorului.

Următoarele trei capitole se bazează pe emulatorul de microprocesor *„EMU 8086”* şi au rolul de a familiariza cititorul cu arhitectura procesorului şi mnemonicile instrucţiunilor. Grafica care însoţeşte aplicaţiile are darul de a facilita înţelegerea şi asimilarea acestor noţiuni.

Lucrările 6, 7 şi 8 prezintă setul de instrucţiuni de bază al procesoarelor x86 grupate după funcţiile pe care le îndeplinesc, însoţite de exemple şi aplicaţii. Dezvoltarea aplicaţiilor în limbaj de asamblare utilizand macroinstructiuni este tratată în următoarea lucrare.

Lucrarea 10 abordează aspectele legate de folosirea subrutinelor, întreruperilor şi a serviciilor BIOS şi DOS oferite de sistemul de operare.

Următoarea lucrare tratează interfaţarea aplicaţiilor în asamblare cu sistemul de operare DOS însoţită de exemple şi programe.

O selectie de instrucţiuni din setul extins x86 evidenţiind îmbunătăţirile aduse de anumite instrucţiuni şi noutăţile apărute este prezentată in *Lucrarea 12.*

In *Capitolul 13* este tratata dezvoltarea aplicatiilor C si asamblare (in line) folosind facilitatile oferite de Visual Studio IDE.

Urmatorul capitol reprezintă o colecţie importantă de peste 20 de probleme şi aplicaţii, de diferite grade de complexitate şi dificultate, din care o bună parte sunt rezolvate, iar restul sunt propuse ca teme, cu sugestii şi indicaţii pentru rezolvare.

Capitolul 15 cuprinde peste 140 de întrebări și teste, iar pentru o parte dintre ele sunt oferite și soluțiile. Obiectivul acestui capitol este evaluarea însuşirii cunoştinţelor prezentate în lucrare.

Sperăm ca lucrările, aplicaţiile şi chestionarele propuse să fie de folos tuturor celor care vor avea interesul sau curiozitatea să le parcurgă sau să le utilizeze.

Cluj-Napoca
1 Noiembrie 2018 Autorii

1. Introducere în microprocesoare

1.1 Generalităţi

Elementul de bază al unui calculator este reprezentat de microprocesor, un *chip* deosebit de complex plasat de obicei pe placa de bază a sistemului de calcul. Microprocesorul asigură procesarea datelor, adică interpretarea, prelucrarea şi controlul acestora, supervizează transferurile de informaţii şi controlează activitatea generală a celorlalte componente care alcătuiesc sistemul de calcul. Structura internă a microprocesorului este compusă din mai multe micromodule interconectate prin intermediul unor căi de comunicaţie numite magistrale sau busuri interne care pot transfera *date* şi *instrucţiuni* (sau *comenzi*). Datele şi instrucţiunile formează un program care este executat pe un sistem de calcul şi reprezintă informaţia procesată.

Microcalculatoarele tipice folosesc o unitate centrală de prelucrare (CPU – Central Processing Unit), un ceas (clock) şi interfeţe cu memoria şi cu dispozitivele externe de intrare/ieşire. Unităţile sunt interconectate prin magistrale care transferă informaţiile între acestea.

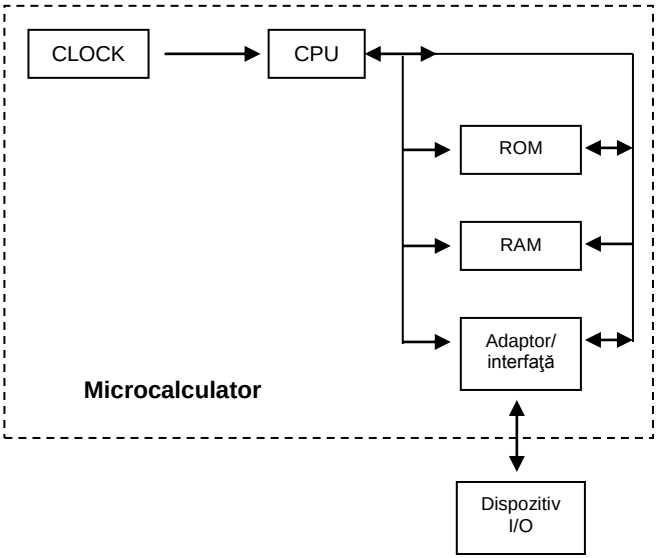


Fig.1.1. Exemplu de arhitectură de microcalculator

1.2 Exemplu de arhitectură de microprocesor

Microprocesorul conţine unitatea aritmetică şi logică (UAL) şi unitatea de control (UC). Acesta este conectat la memorie şi la dispozitivele de intrare/ieşire prin magistrale.
Informaţia este transmisă între unităţile microcalculatorului prin magistrale (bus-uri). Există câte o linie pentru fiecare bit de informaţie transmis, deci 16 linii pentru o magistrală de 16 biţi. Magistralele pot fi *de adrese*, *de date* şi *de control*.

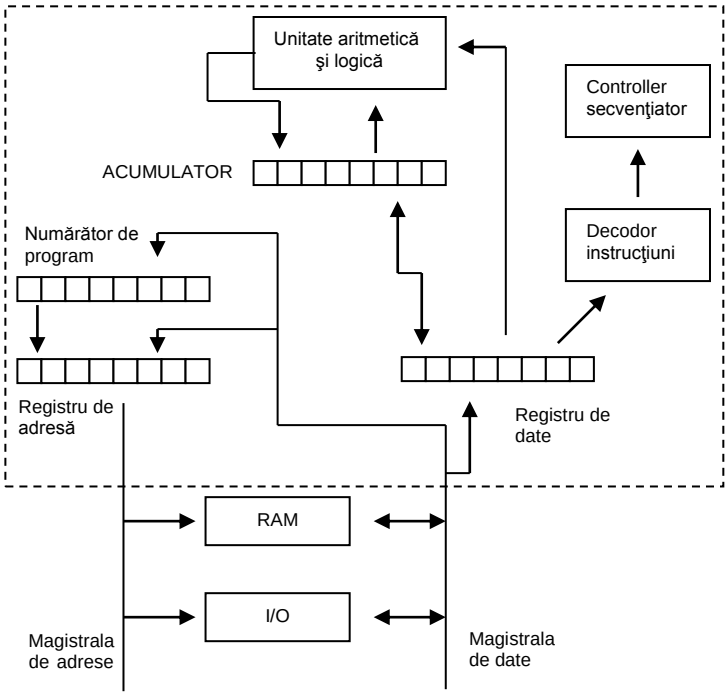


Fig.1.2. Exemplu de arhitectură de microprocesor

1.2.1 Unitatea aritmetică şi logică (UAL)

Toate operaţiile aritmetice/logice ale unui microprocesor au loc în unitatea aritmetică şi logică. Folosind o combinaţie de porţi şi bistabile interne, numerele pot fi adunate în mai puţin de o microsecundă, chiar şi în procesoarele mici. Operaţia ce trebuie executată este specificată de semnalele generate de unitatea de control prin decodificarea instrucţiunilor.

Datele asupra cărora se realizează operaţia pot proveni din memorie sau de la o intrare externă (port). Datele numerice sunt procesate pe baza unui set de instrucţiuni, având ca operaţii de bază adunarea, scăderea şi operaţiile logice.

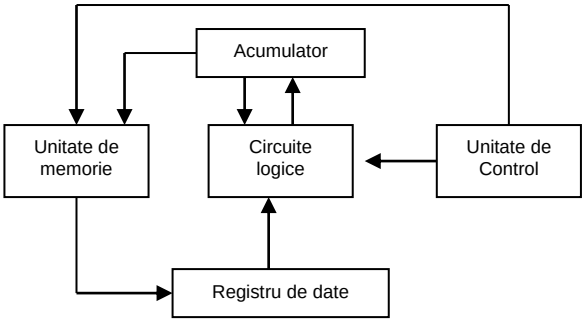


Fig.1.3. Unitatea aritmetică şi logică

1.2.2 Acumulatorul (Acc)

Acumulatorul este registrul principal al unităţii aritmetice şi logice a microprocesorului. Regiştrii sunt elemente de memorie care pot păstra datele. Acumulatorul conţine de obicei primul operand din datele implicate într-un calcul. De exemplu, dacă un număr din memorie este adunat la aceste date, suma înlocuieşte datele originale din acumulator. Acesta este locul de depozitare pentru rezultatele operaţiilor aritmetice, care pot fi apoi transferate în memorie sau la dispozitive periferice.

1.2.3 Unitatea de control a microprocesorului (UC)

Unitatea de control a microprocesorului coordonează operaţiile celorlalte unităţi prin generarea semnalelor de temporizare şi control. Funcţia microcalculatorului este de a executa programele stocate în memorie. Unitatea de control conţine logica necesară interpretării instrucţiunilor şi generării semnalelor necesare execuţiei acestor instrucţiuni. Cuvintele descriptive “fetch” şi “execute” sunt folosite pentru a descrie acţiunile unităţii de control. Aceasta citeşte instrucţiunea (fetch) prin generarea unei adrese şi a unei comenzi de citire către unitatea de memorie. Instrucţiunea de la adresa respectivă este transferată unităţii de control pentru decodare. Aceasta generează apoi semnalele necesare pentru execuţia instrucţiunii.

Cu toate că sunt componente electronice extrem de complexe, microprocesoarele execută instrucţiunile secvenţial. Microprocesoarele ce pot executa în paralel mai mult de o instrucţiune au o arhitectură superscalară, dispunând de două unităţi aritmetice şi logice. Succesiunea instrucţiunilor unui program este executată în general secvenţial. Printr-o

structură *pipeline* se pot executa instrucţiuni secvenţiale paralele, situaţie în care faze diferite ale unor instrucţiuni succesive se execută în acelaşi timp.
Toate instrucţiunile pe care le poate executa un microprocesor formează *setul de instrucţiuni al microprocesorului*. Acest set a fost proiectat şi optimizat pentru fiecare procesor în parte. Toate microprocesoarele Intel 80x86 inclusiv Pentium, au setul de instrucţiuni compatibil cu versiunile anterioare.

Microprocesor	Set de instrucţiuni
8088/8086	115
80186	126
80286	142
80386	200
80486	206
Pentium	211
Pentium MMX	211+ 57

Tabel 1.1. Procesoarele Intel 80x86 şi setul de instrucţiuni

Instrucţiunile limbajelor evoluate (High Level Language) (C, Pascal, Prolog, Lisp, etc) necesită transformarea lor de către compilator în instrucţiuni simple, interpretabile de către microprocesor. Astfel, pentru fiecare instrucţiune a limbajului de nivel înalt, se generează una sau mai multe instrucţiuni aparţinând setului de instrucţiuni al microprocesorului.

Setul de instrucţiuni pe care un procesor îl poate executa şi care este limbajul de programare la nivel scăzut al microprocesorului formează *limbajul maşină* sau *codul maşină*. Pentru a facilita utilizarea limbajului maşină, la care instrucţiunile reprezintă o înşiruire de biţi, se utilizează **mnemonicile** care abreviază operaţia executată de instrucţiune. Setul de instrucţiuni prezentat sub forma mnemonicilor reprezintă *limbajul de asamblare*. Modul în care partea hardware este comandată de utilizator prin intermediul software-ului, utilizând diverse nivele de limbaj, este prezentată în figura 1.4.

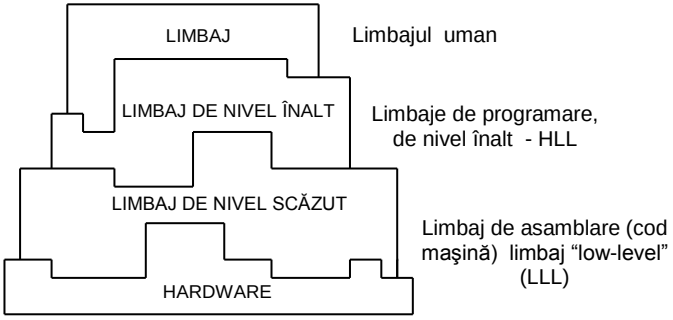


Fig.1.4. Ierarhizarea limbajelor

1.3 Arhitecturi de prelucrare

La baza majorităţii calculatoarelor actuale stau cele cinci criterii enunţate de von Neumann şi prezentate în figura 1.5.

Un calculator cu program memorat trebuie să posede:

- Intrare pentru un număr nelimitat de date şi instrucţiuni;
- Memorie din care se pot citi instrucţiuni şi operanzi şi în care se depun rezultate;
- Ieşire care să pună rezultatele la dispoziţia utilizatorului;
- Unitate de calcul (UAL – unitate aritmetică şi logică sau UE - unitate de execuţie) care să execute operaţii aritmetice şi logice asupra datelor din memorie;
- Unitate de comandă (sau control) - UC care să interpreteze instrucţiunile extrase din memorie şi să aleagă diferite acţiuni pe baza rezultatelor calculelor.

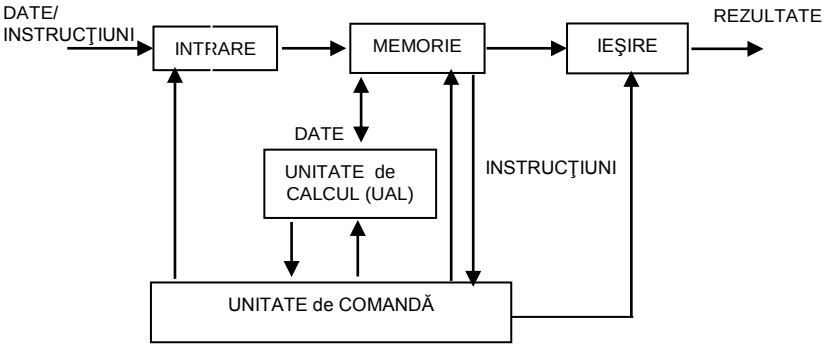


Fig. 1.5. Schema bloc a calculatorului cu program memorat

După modul în care se conectează procesorul la memorie, se extrag datele şi instrucţiunile sau se optimizează prelucrările, în practică se întâlnesc mai multe arhitecturi de procesare.

Arhitectura "von Neumann" sau SISD (single instruction single data) are următoarele caracteristici:

- extragerea datelor şi instrucţiunilor se face pe aceeaşi magistrală;
- instrucţiunile se extrag şi se execută secvenţial.

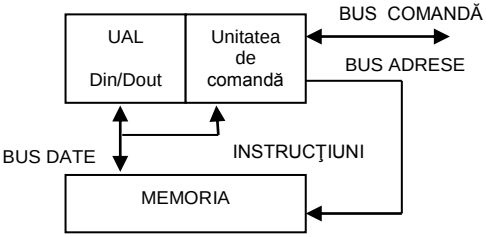


Fig. 1.6. Arhitectura SISD

Arhitectura Harvard are memoria partajată în memorie de date şi memorie de program, permiţând o prelucrare mai eficientă şi executarea paralelă a diferitelor faze ale instrucţiunilor (pipeline). Este utilizată cu precădere la procesoarele de semnal (DSP) şi microcontrolere.

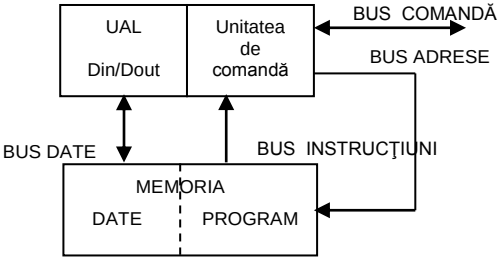


Fig. 1.7. Arhitectura Harvard

Arhitectura SIMD (single instruction multiple data) numită şi arie de procesoare sau *tablou sistolic* permite execuţia aceleiaşi instrucţiuni pe mai multe date simultan, ceea ce implică existenţa mai multor unităţi aritmetice. Arhitectura e utilă şi la prelucrări multimedia, fiind întâlnită la Pentium MMX.

Arhitectura MIMD (multiple instruction multiple data) sau Data Flow conţine mai multe procesoare care rulează programe diferite operând cu date diferite în paralel, conlucrând la rezolvarea unei aplicaţii (task)

1.4 Reprezentarea numerelor

În calculator, memoria conţine numere. Datorită logicii booleene pe care este conceput hardware-ul, calculatoarele stochează informaţia în format binar, nu zecimal. *Sistemul zecimal* foloseşte 10 digiţi (0-9), fiecare digit al unui număr având asociată o putere a lui 10 în funcţie de poziţia sa, deci sistemul de numerotaţie este un sistem poziţional:

$$234=2*10^2+3*10^1+4*10^0$$

*Sistemul **binar*** foloseşte doar două cifre, 0 şi 1, fiecare cifră a unui număr având asociată o putere a lui 2 în funcţie de poziţia sa:

$$11001_2=1*2^4+1*2^3+0*2^2+0*2^1+1*2^0=16+8+1=25_{10}$$

Adunarea în sistemul binar este intuitivă:

$$\begin{array}{r} 11011_2 \\ \underline{+10001_2} \\ 101100_2 \end{array}$$

*Sistemul **hexazecimal*** foloseşte baza 16 şi este folosit pentru reprezentarea prescurtată a numerelor binare. Se folosesc cifrele de la 0 la 9, iar ca cifre suplimentare se folosesc caracterele A – F. Modul de transformare în zecimal este asemănător:

$$2BD_{16}=2*16^2+11*16^1+13*16^0=512+176+13=701_{10}$$

Conversia hexa-binar şi invers se face foarte uşor: fiecare cifră hexa este transformată într-un număr binar pe 4 cifre, şi invers, grupuri de câte 4 cifre binare se transformă într-o cifră hexa:

0110	0000	0101	1010	0111	1110
6	0	5	A	7	E

Convenţiile de reprezentare a numerelor întregi

Numerele întregi pot fi reprezentate a) fără semn sau b) cu semn.

*a) Reprezentarea **fără semn*** este intuitivă: numărul 200_{10} va fi reprezentat în binar ca 11001000_2, în hexazecimal ca şi C8h.

b) Numerele întregi ***cu semn*** (pozitiv sau negativ) sunt însă mai complicat de reprezentat. Există 3 convenţii de reprezentare a numerelor cu semn, toate folosesc ca *bit de semn* bitul cel mai semnificativ al reprezentării. Bitul de semn este 0 pentru un număr pozitiv şi 1 pentru un număr negativ.

Modul şi semn: numărul va fi reprezentat din două parţi: bit de semn şi valoare absolută. Numărul 56_{10}=38h va fi reprezentat ca $\underline{0}0111000_2$, cu bitul de semn subliniat, iar -56 va fi $\underline{1}0111000_2$. Cea mai mare valoare reprezentabilă pe octet va fi +127=$\underline{0}1111111_2$, respectiv cea mai mică valoare -127=$\underline{1}1111111_2$. Această metodă are dezavantaje în organizarea logică a UC. Zero nu este nici pozitiv nici negativ, deci reprezentările $\underline{1}0000000_2$ şi $\underline{0}0000000_2$ sunt echivalente.

Complement faţă de 1 (C1): se calculează prin complementarea fiecărui bit din reprezentare. Reprezentarea lui +56, $\underline{0}0111000_2$ va fi deci $\underline{1}1000111_2$. Prin urmare, -56 va fi $\underline{1}1000111_2$. Bitul de semn a fost schimbat automat prin complementare. Complementul faţă de 1 are aceleaşi dezavantaje ca şi metoda de reprezentare în modul şi semn: două reprezentări echivalente pentru zero şi aritmetică complicată.

Complement faţă de 2 (C2): se obţine astfel: se adună 1 la reprezentarea în C1. C1 s-a folosit în primele calculatoare, iar C2 este reprezentarea standard folosită în calculatoarele de azi pentru numerele întregi. Reprezentarea C2 a lui 56 este: $\underline{0}0111000_2$, iar pentru +56 avem:

$$\begin{array}{ll} \text{-> C1} & \underline{1}1000111_2 \\ & \underline{+\qquad 1} \\ & \underline{1}1001000_2 \end{array}$$

Există o metodă rapidă de calcul a complementului faţă de 1, pentru a nu trece prin reprezentarea binară: se scade din F fiecare cifră din reprezentarea hexazecimală a numărului. +56 este 38h. Scăzând fiecare cifră din Fh obţinem C7h care se reprezintă ca 11000111_2 (acelaşi rezultat pentru C1).
Adunarea a două numere în C2 poate produce transport (Carry), dar acesta nu este folosit. Toate datele sunt reprezentate pe o lungime fixă (ca număr de biţi), deci adunarea a doi octeţi va produce ca rezultat tot un octet:

$$\begin{array}{ll} \underline{1}1111111_2 & \\ \underline{+\qquad 1} & \\ \underline{0}0000000_2 & \text{Carry=1} \end{array}$$

Ca o consecinţă, în reprezentarea C2 există o singură notaţie pentru zero. Vom vedea mai târziu cum detectăm acest transport (Carry), momentan este suficient de ştiut că nu este reprezentat în rezultat.
Prin folosirea sistemului C2, pe 8 biţi se pot reprezenta numere întregi de la -128 la +127; pe 16 biţi se pot reprezenta numerele între -32768 şi +32767.

Număr	*C2 pe 8 biţi*	*C2 pe 16 biţi*
+32767	nu se poate	7FFFh
-32768	nu se poate	8000h
-128	80h	FF80h
-1	FFh	FFFFh

UC nu ştie ce anume reprezintă un anumit octet (sau cuvânt). În limbaj de asamblare nu există tipuri de date ca în limbajele de nivel înalt. Modul de reprezentare al datelor este determinat de instrucţiunea folosită. Valoarea FF_{16} este considerată ca reprezentând -1 în reprezentarea cu semn sau 255 în reprezentarea fără semn, în funcţie de aplicaţie. Dacă vom considera numerele reprezentate pe trei biţi, vom avea în cele 3 convenţii valorile:

Număr binar	Modul şi semn	Complement faţă de 1	Complement faţă de 2
000	0	0	0
001	1	1	1
010	2	2	2
011	3	3	3
100	-0	-3	-4
101	-1	-2	-3
110	-2	-1	-2
111	-3	-0	-1

1.5 Reprezentarea datelor în memorie

Folosind un număr de 8 biţi se pot reprezenta 2^8=256 valori diferite. În figura 1.8 s-a ilustrat grafic *gama numerelor* pe sistemul de axe, cu poziţionarea celor 256 valori diferite ***fără semn*** versus ***cu semn*** care se pot scrie ***folosind un octet***.

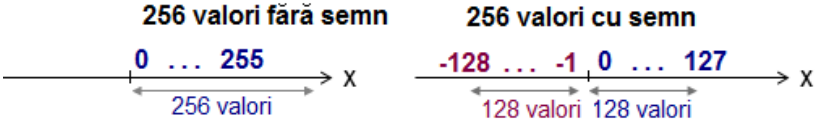


Fig. 1.8. Reprezentarea a 256 valori fără semn vs. cu semn

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 1.9; pentru a fi cât mai uşor de urmărit, s-au reprezentat biţii grupaţi în octeţi.

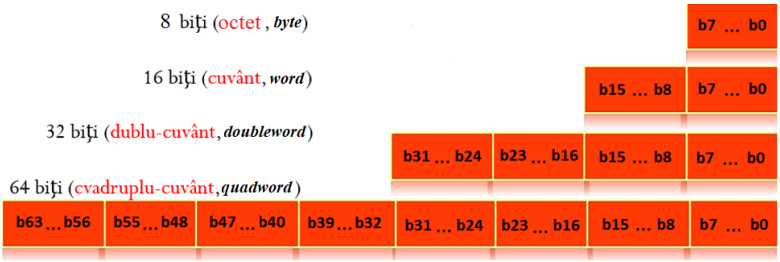


Fig. 1.9 Octetul şi multiplii uzuali ai octetului: cuvânt, dublucuvânt şi cvadruplucuvânt

Reprezentarea din figura 1.10 se referă la modul cum se depune în memorie octetul ***21h***, cuvântul ***43 21h***, şi respectiv dublucuvântul ***87 65 43 21h***, începând de la adresa 126 (ocupă 1 locaţie, 2 locaţii sau 4 locaţii succesive).

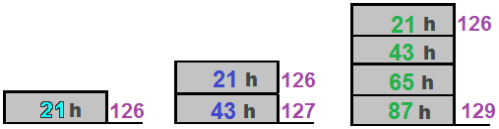


Fig. 1.10 Octetul 21h, cuvântul 4321h şi dublucuvântul 87654321h stocate în memorie începând de la adresa 126

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 *Intel* – figura 1.11a);

- Big Endian: octetul **LSB** se depune în memorie la locaţia cu **adresa cea mai mare (“Big”),** convenţia fiind specifică procesoarelor din familia *Motorola* – figura 1.11b).

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

Fig. 1.11 Depunerea dublucuvântului 87654321h în memorie după
a) convenţia Little Endian (stânga); b) convenţia Big Endian (dreapta)

Interacţiunea dintre utilizator și SC se realizează prin intermediul dispozitivelor de intrare-ieșire, de exemplu al tastaturii și al ecranului. Pentru a interacţiona cu acestea, SC vehiculează *coduri ASCII* atât la preluarea unui caracter de la tastatură cât și la afișarea unei valori pe ecran.

Exemple de coduri Ascii: cifrele 0...9 au codurile Ascii 30h ...39h, literele mici încep de la valoarea 61h ce corespunde lui ‚a’, iar literele mari încep de la valoarea 41h ce corespunde lui ‚A’.

Exemplu: Valoarea numerică a şirului ASCİİ „**Salut**” este **53h 61h 6Ch 75h 74h**. Valoarea 53h, în zecimal e 83, iar în binar pe 7 biţi se scrie 1010011b, dar în memorie va fi stocată ca octet de valoare 01010011b. Un program care face depanare ar putea afişa această valoare ca „53” (fără să mai precizeze şi sufixul ***h*** de la hexa), dar dacă această valoare ar fi copiată în zona de memorie video, atunci pe ecran apare „S”, deoarece 53h este codul ASCİİ al lui „S”.

Există o mare diferenţă între ***valori binare*** și ***coduri Ascii*** din punct de vedere al interpretării. De exemplu, dacă se definește o valoare sau un element al unui șir (așa cum apare în exemplul de mai jos) ca **1**, acesta nu e identic cu ***‘1’***. În primul caz e ***valoarea 1***, interpretată ca și codul Ascii al caracterului ☺ în figura 1.12 (adresa 07103h), pe când în cel de-al doilea caz e ***codul Ascii al caracterului ‚1’***, adică valoarea 31h (adresa 07105h).
O altă diferenţă este observabilă când ne referim la valoarea A în hexazecimal, sau mai corect 0Ah și ‘A’. În primul caz, valoarea 0Ah este echivalentă numărului 10 (la adresa 07107h), pe când ‘A’ este caracterul având codul Ascii 41h (la adresa 07108h).

Adresa	in hexa	in zecimal	Cod Ascii caracter
07102:	00	000	NULL
07103:	01	001	☺
07104:	02	002	☻
07105:	31	049	1
07106:	32	050	2
07107:	0A	010	NEWL
07108:	41	065	A
07109:	42	066	B

Fig. 1.12. Exemple de valori interpretate ca numere binare sau coduri Ascii

Variabilele identifică datele, formând operanzi pentru instrucţiuni. Pentru declararea variabilelor se utilizează directive care alocă şi iniţializează memoria în unităţi de octeţi, cuvinte, dublu-cuvinte, astfel:

- directiva ***db*** *(Define Byte)* declară octeţi sau şiruri de octeţi;
- directiva ***dw*** *(Define Word)* declară cuvinte sau şiruri de cuvinte;
- directiva ***dd*** *(Define Doubleword)* declară dublucuvinte sau şiruri de dublucuvinte;

De exemplu, pentru obţinerea datelor aşa cum apar în figura 1.12 s-a folosit definirea unei variabile cu numele „sir” de tip octet, de forma:

sir db 0,1,2,'1','2', 0Ah, 'A','B'

1.6 Exerciţii şi teme

1. Se vor studia modalităţile de conversie a numerelor din bazele 2, 8,10, 16 şi operaţii aritmetice corespunzătoare.

2. Să se convertească următoarele numere din baza 10 în baza 8 şi 16:
   267; 1089; 530; 104; 708

3. Reprezentaţi în C2 pe 8 şi 16 biţi numerele:
   -204; -23; -67;

4. Să se calculeze în baza 16:

8Ah+	E6h-	-3*	5B16h+	22CAh+	3C51h-
56h	18h	6	8FEh	43Bh	2FDh

5. Realizaţi un desen care să ilustreze conţinutul memoriei dacă se va depune (după convenţia Little End-ian) dubucuvântul 12345678h, urmat de cuvântul ABCDh în memorie, începând de la adresa 102h; specificaţi şi adresa în hexazecimal.

6. Să se convertească următoarele numere din baza 10 în baza indicată, ţinând cont de faptul că sunt numere fără semn:
   a) 249; b) 251; c) 254; d) 247;
   Numărul: _____d = _____b = _____h

7. Scrieţi următoarele numere în binar folosind minim 8 biţi, considerând numerele fără semn:
a) 1= .....b; b) 7= .....b; c) 2516= .....b;
d) 1250=.....b; e) 258= .....b;

8. Repetaţi exerciţiul 6. considerând numerele cu semn.

9. Repetaţi exerciţiul 7. considerând numerele cu semn.

10. Scrieţi următoarele numere în hexazecimal folosind minim 2 cifre hexazecimale:
a) 1=.....h; b) 7=.....h; c) 2516=.....h; d) 1250=.....h; e) 258=.....h;

11. Scrieţi următoarele numere din binar în zecimal, ştiind că sunt numere fără semn:
a) 0000 0010b = ... ; b) 000000000001b = ...;
1111110000b = ...; 111110000001b = ...;
c) 000000000000111b = ...; d) 0000000000111b = ...;
10000000b = ...; 1010101010b = ...;

12. Scrieţi următoarele numere din binar în zecimal, ştiind că sunt numere cu semn:
a) 0000 0010b = ... ; b) 000000000001b = ...;
1111110000b = ...; 111110000001b = ...;
c) 000000000000111b = ...; d) 0000000000111b = ...;
10000000b = ...; 1010101010b = ...;

13. Scrieţi următoarele numere din binar în hexazecimal:
a) 0000 0010b = ... ; b) 000000000001b = ...;
1111110000b = ...; 111110000001b = ...;
c) 000000000000111b = ...; d) 0000000000111b = ...;
10000000b = ...; 1010101010b = ...;

14. Câţi octeţi se ocupă în memorie, dacă se folosește următoarea directivă?
a) a db 12h,34h,90h b) a dw -1,4,-4 c) a db 127,-128
b dw 56h,78h b db 2,-8 b dw -127,128

Nr. octeţi pt a: ____
Nr. octeţi pt b: ____

15. Scrieţi o directivă prin care să definiţi variabila *nume* de tip octet care să conţină șirul Ascii al caracterelor care determină prenumele dvs. Ilustraţi printr-un desen modul cum apare această variabilă în memorie.

2. Microprocesorul 8086

2.1. Scurt istoric

Microprocesoarele "pe n biţi" sunt acele procesoare la care lungimea cuvântului prelucrat (dimensiunea regiştrilor, lungimea uzuală a unui operand) este de n biţi. Cele mai răspândite microprocesoare în calculatoarele personale sunt cele din familia INTEL. În continuare este prezentată o evoluţie a acestei familii:

8080/8085	microprocesor pe 8 biţi, 64Ko de memorie
8086/8088	microprocesor pe 16 biţi maximum 1 Mo de memorie, mod de lucru real, single task arhitectură de prelucrare secvenţial-paralelă (pipeline)
80286	magistrală internă şi externă pe 16 biţi maximum 16 Mo de memorie, mod de lucru real/protejat suport pentru sistemele de operare multitasking
80386DX	microprocesor pe 32 de biţi, 4 Go de memorie mod de lucru real, protejat 16/32 de biţi, real virtual 8086 4GB de memorie principală posibil
80386SX	microprocesor pe 32 de biţi magistrală externă pe 16 biţi, 16 Mo de memorie
80486	dispune de coprocesor matematic FPU (Floating Point Unit) şi memorie cache pe acelaşi chip cu procesorul
Pentium	microprocesor pe 32 de biţi, 4 Go de memorie arhitectura superscalară (RISC) permite execuţia a mai mult de o instrucţiune/tact în anumite condiţii.

Caracteristicile diferiţilor reprezentanţi sunt prezentate în anexa 1.

2.2 Arhitectura software a microprocesorului 8086

Schema bloc internă a microprocesorului 8086 este prezentată în figura 4.3. Se compune din:

- EU – *Execution Unit* – Unitate de execuţie – execută calculele prin intermediul componentei ALU (UAL - Unitatea Aritmetică şi Logică); EU are în componenţă Unitatea Aritmetică şi Logică, regiştrii generali, regiştrii de adresare, regiştrii temporari, unitatea de decodificare şi comandă şi registrul de flaguri;
- BIU – *Bus Interface Unit* – Unitate de interfaţă cu bus-urile– componenta care pregăteşte execuţia fiecărei instrucţiuni; în esenţă, aceasta extrage o instrucţiune din memorie, o depune în coada de instrucţiuni şi calculează adresa din memorie a unui eventual operand.

Cele două componente lucrează în paralel în sensul că în timp ce EU execută instrucţiunea curentă, BIU pregăteşte instrucţiunea următoare; aceasta este o arhitectură secvenţial-paralelă.

Regiştrii generali sunt folosiţi pentru a stoca temporar operanzi şi rezultate în UC. Regiştrii de uz general AX, BX, CX, DX sunt regiştri pe 16 biţi. O utilizare implicită a acestor regiştri este următoarea:

- AX – registru acumulator
- BX – registru de bază în adresare
- CX – registru contor
- DX – registru de date (extensia acumulatorului), adresare indirectă a porturilor

Regiştrii pot fi accesaţi pe 16 biţi ca AX, BX, CX, DX sau pe 8 biţi, având partea inferioară AL, BL, CL, DL şi partea superioară AH, BH, CH, DH.
Regiştrii SP şi BP sunt regiştri destinaţi lucrului cu stiva. Stiva se defineşte ca o zonă de memorie (LIFO – Last In First Out) în care pot fi depuse valori, extragerea lor ulterioară facându-se în ordine inversă depunerii. SP – Stack Pointer – pointează spre ultimul element introdus în stivă, iar BP – Base Pointer – către baza stivei.
Regiştrii DI şi SI sunt regiştri index (Destination Index şi Source Index) destinaţi lucrului cu şiruri de octeţi sau cuvinte.
Registrul PSW/Flags este registrul indicatorilor de stare şi control. Un *flag* este un indicator reprezentat pe un bit. Indicatorii de stare ai registrului de flaguri sintetizează execuţia ultimei instrucţiuni. Pentru 8086 acest registru are 16 biţi din care sunt folosiţi numai 9. Structura acestui registru este prezentată mai jos.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-	-	-	-	**O**	**D**	**I**	**T**	**S**	**Z**	-	**A**	-	**P**	-	**C**

Indicatorii de stare sunt:
C – Carry – indică un transport în afara domeniului de reprezentare a rezultatului;
P – Parity – este stabilit astfel încât împreună cu numărul de biţi de 1 din rezultat să rezulte un număr impar de cifre de 1;
A – Auxiliary – indică valoarea transportului de la bitul 3 la bitul 4 (între cifrele hexazecimale);
Z – Zero – are valoarea 1 dacă rezultatul ultimei instrucţiuni este egal cu zero;
S – Sign – are valoarea 1 când rezultatul ultimei operaţii este un număr strict negativ, adică copiază bitul MSB al rezultatului;
O – Overflow – indică depăşire de gamă: dacă rezultatul ultimei instrucţiuni a depăşit spaţiul rezervat rezultatului, în cazul operanzilor consideraţi numere cu semn.

$$O=C \oplus transport_{MSB-1\ MSB}$$

Indicatorii de control sunt:
T – Trap – flag pentru depanare; dacă are valoarea 1, atunci procesorul se opreşte după fiecare instrucţiune;
I – Interrupt – flag de întrerupere; permite sau invalidează acceptarea întreruperilor externe mascabile care apar pe intrarea INT a procesorului;
D – Direction – flag folosit la lucrul cu şiruri pentru a indica direcţia de parcurgere de-a lungul şirului, D=0 – adrese crescătoare, D=1 – adrese descrescătoare.

Exemple:

Valorile biţilor de Carry, Zero, Sign şi Overflow imediat după executarea următoarelor adunări de către un microprocesor pe 8 biţi sunt următoarele:

	Rezultat	**C**	**Z**	**S**	**O**
02 + 02	04	0	0	0	0
02 + 7D	7F	0	0	0	0
04 + 7F	83	0	0	1	1
80 + 80	00	1	1	0	1

2.3. Organizarea şi adresarea memoriei

Prin definiţie, *adresa unei locaţii de memorie* este numărul de ordine între începutul memoriei RAM şi locaţia respectivă. Dată fiind capacitatea de 1Mo a memoriei la 8086, o adresă trebuie să se reprezinte pe 20 de biţi, dar capacitatea regiştrilor şi a cuvintelor este de 16 biţi. Pentru rezolvarea situaţiei a apărut conceptul de segment de memorie, respectiv adresarea segmentată.

Segmentul de memorie reprezintă o succesiune continuă de octeţi care are următoarele proprietăţi: începe la o adresă multiplu de 16 (paragraf), are lungimea multiplu de 16 octeţi şi maximum 64 Kocteţi. Deoarece adresa de început a fiecărui segment este multiplu de 16, cei mai puţin semnificativi 4 biţi ai acestei adrese sunt zero.

Offset-ul sau deplasamentul reprezintă adresa unei locaţii faţă de începutul segmentului. Deoarece un segment are maximum 64 Ko, pentru exprimarea offsetului sunt suficienţi 16 biţi.

Adresa logică este o pereche de numere pe câte 16 biţi fiecare, unul reprezentând adresa de început a segmentului şi celălalt reprezentând offsetul în cadrul segmentului.

Adr.Logică = Reg.Segment:offset

Adresa fizică pe 20 de biţi se obţine din configuraţia de 16 biţi care localizează începutul segmentului înmulţită cu 16 la care se adună valoarea offsetului. Acest calcul este efectuat de unitatea de adresare din BIU.

$$AF_{20}=16 \bullet Reg.Segment_{16}+offset_{16}$$

Arhitectura 8086 permite existenţa a patru tipuri de segmente:

- CS – segment de cod – care conţine instrucţiuni;
- DS – segment de date – care conţine date care se prelucrează conform instrucţiunilor;
- SS – segment de stivă;
- ES – segment de date suplimentar (extrasegment).

În fiecare moment al execuţiei este declarat activ câte un singur segment din fiecare tip. Regiştrii CS (Code Segment), DS (Data Segment), SS (Stack Segment) şi ES (Extra Segment) din BIU reţin adresele de început ale segmentelor active, corespunzătoare fiecărui tip de segment.

Registrul IP – *Instruction Pointer* – conţine offsetul instrucţiunii curente în cadrul segmentului de cod curent, el fiind manipulat exclusiv de către BIU.

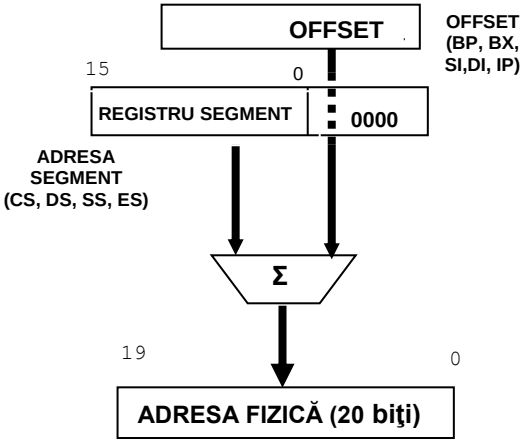


Fig.2.1. Calculul adresei fizice

Exemplu:

Dacă CS=24F6h şi IP=634Ah

- adresa logică este 24F6h:634Ah
- offsetul este 634Ah
- adresa fizică este 2B2AAh = 24F60h+634Ah

O adresă fizică poate fi obţinută din mai multe combinaţii de adrese logice:

Adresa logică (hexa)	Adresa fizică (hexa)
1000:5020	15020
1500:0020	15020
1502:0000	15020
1400:1020	15020
1302:2000	15020

O prezentare sintetică a regiştrilor microprocesorului 8086 se găseşte în tabelul următor.

Registru	Biţi	Nume registru
General	16	**AX, BX, CX, DX**
(date)	8	**AL, AH, BL,BH, CL, CH, DL, DH**
Pointer	16	**SP, BP**
Index	16	**SI, DI**
Segment	16	**CS, DS, SS, ES**
Instrucţiune	16	**IP** (contor instrucţiuni)
Flag-uri	16	**PSW**

Tabelul 2.1. Regiştrii procesorului 8086

2.4 Moduri de adresare

În cadrul unei instrucţiuni există mai multe moduri de a calcula adresa efectivă sau offsetul unui operand pe care aceasta îl solicită:

- modul registru – dacă operandul este un registru;
- modul imediat – atunci când în instrucţiune se află chiar valoarea operandului;
- modul de adresare cu memoria – dacă operandul se află undeva în memorie.

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

Adresarea directă în cazul regiştrilor înseamnă folosirea valorii reale din interiorul registrului în momentul execuţiei instrucţiunii. În plus, există unele instrucţiuni care pot fi folosite numai cu operanzi regiştri şi instrucţiuni care pot fi folosite numai cu anumiţi regiştri. În cazul adresării cu regiştri memoria nu este accesată.

Exemple:

```
mov BX, DX      ; BX = DX
mov ES, AX      ; ES = AX
add AL, BH      ; AL = AL+BH
```

Observaţie: sursa şi destinaţia trebuie să aibă aceeaşi dimensiune !

Adresarea imediată are ca operand sursă o constantă. Acest mod nu poate fi folosit pentru încărcarea datelor în regiştri segment şi în flaguri.

Exemple:
```
mov AX, 2550h  ; AX = 2550h
mov CX, 625    ; CX = 625
add BL, 40h    ; BL = BL+40h
```

Când un operand se află în memorie, procesorul calculează adresa efectivă (offsetul) a datelor care vor fi prelucrate. Calcularea acestei adrese depinde de modalitatea în care este specificat operandul. Nu sunt admise operaţiile pentru care atât sursa cât şi destinaţia sunt operanzi din memorie.

A. Operandul cu adresare directă este o constantă sau un simbol care reprezintă adresa (segment şi deplasament) unei instrucţiuni sau a unor date. Deplasamentul unui operand cu adresare directă este calculat în momentul asamblării. Adresa fiecărui operand raportată la structura programului este calculată în momentul editării de legături. Adresa efectivă este calculată în momentul încărcării programului pentru execuţie. Adresa efectivă este întotdeauna raportată la un registru de segment.

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

```
mov AL, 50h
mov [4320h], AL
```

Adr.Fizică = 18A20h ; [18A20h] = 50h.

B. Operanzii cu adresare indirectă utilizează regiştri pentru a indica adrese din memorie. Adresarea indirectă este folosită în manipularea dinamică a datelor, deoarece valorile din regiştri se pot modifica. În cazul microprocesoarelor 8086 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. Forma generală pentru accesarea indirectă a unui operand de memorie este:

[BX | BP] + [DI | SI] + [deplasament 8/16 biţi]

adică adunând următoarele trei elemente, sau numai unele dintre ele:
- conţinutul unuia dintre regiştrii BX sau BP
- conţinutul unuia dintre regiştrii DI sau SI
- un deplasament.

Rezultă astfel următoarele moduri de adresare la memorie:
- *directă* – atunci când apare numai constanta
- *bazată* – atunci cand în calcul apare un registru de bază
- *indexată* – atunci când în calcul apare un registru index

sau combinaţii ale acestora.
Dacă BX este 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 DS va fi folosit ca registru segment implicit. Dacă BP este folosit oriunde în calculul adresei operandului, segmentul implicit este SS.

Se permit diverse modalităţi de a specifica operanzi cu adresare indirectă, folosind orice operator care indică adunarea (+, [ ], .). De exemplu următoarele moduri de specificare sunt echivalente:

```
table [BX][DI] + 6
6 + table [BX+DI]
[table+BX+DI] + 6
table [BX+6][DI]
```

Când se utilizează modul de adresare bazat-indexat, unul dintre regiştri trebuie să fie registru de bază, iar celălalt să fie registru index. Următoarele instrucţiuni sunt **incorecte:**

```
mov AX, table [BX][BP]        ; doi regiştri de bază!
mov AX, table [SI][DI]        ; doi regiştri index!
```

Pentru adresarea indirectă, esenţială este specificarea între paranteze drepte a cel puţin unuia dintre elementele componente ale formei prezentate.

Codificarea instrucţiunilor se realizează în funcţie de modul de adresare folosit. Formatul unei instrucţiuni este de forma:

Cod	d	w	octet - mod de adresare	depl L	depl H

cod – este codul mnemonicii (operaţiei)
d – indică dacă locaţia de memorie este sursă sau destinaţie
d=0 destinaţie, d=1 sursă
w – indică dacă operaţia se face pe octet sau pe cuvânt
w=0 octet, w=1 cuvânt
octetul mod de adresare – este alcătuit din 3 părţi distincte:

7	6	5	4	3	2	1	0
mod			reg		r/m		

mod =
00 – adresare cu memoria, fără deplasament
01 – adresare cu memoria, deplasament pe un octet
10 – adresare cu memoria, deplasament pe 2 octeţi
11 – adresare registru

reg – indică registrul operand în instrucţiune

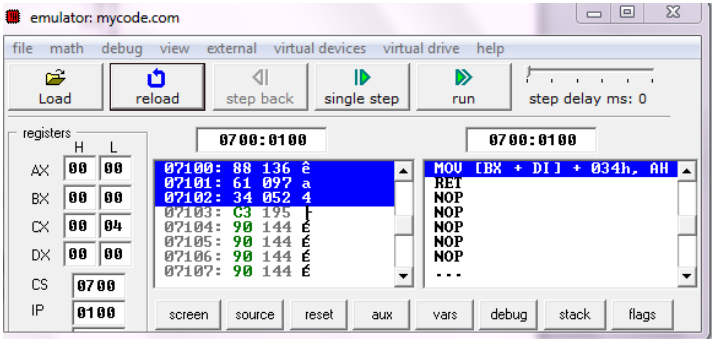
r/m – indică modul de calcul al adresei efective
(registrul segment implicit)

reg	w=0	w=1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Exemplu: `mov [BX+DI+34h], AH ;`

100010	0	0	01	100	001	0011	0100
cod “**mov**”	d	w	mod	reg (AH)	r/m (BX+DI+depl8)	3	4

Folosind simulatorul EMU8086 verificati exemplul de mai sus.



Exemplu: `mov [bx+di+34h], ah ;`

8	8		6	1		3	4
100010	0	0	01	100	001	0011	0100
cod “**mov**”	d	w	mod	reg (AH)	r/m (bx+di+depl8)	3	4

Fig.2.2. Codificarea instrucţiunii mov [BX+DI+34h], AH

Analizati manual dar şi verificaţi codul următoarelor instrucţiuni :

```
mov [BX+DI+1234h], AH ;
mov [BX+DI+1234h], AX ;
```

2.5 Exerciţii şi teme

1. Daţi exemplu de 3 adrese logice ce se pot scrie pt adresa fizică 23456h.

2. Să se calculeze adresa fizică ce corespunde adresei logice 89ABh : 89ABh.

3. Să se calculeze componenta offset corespunzătoare adresei fizice 10000h dacă se dă componenta segment 1000h. Dar dacă adresa fizică este 1FFFFh ?

4. Să se calculeze componenta segment corespunzătoare adresei fizice 0ABC10h dacă se dă componenta offset 600h.

5. Verificaţi dacă adresa fizică 31FFFh aparţine segmentului care are componenta segment 2200h. Dar dacă adresa fizică este 21000h?

6. Conţinutul căror locaţii de memorie este mutat în AX în instrucţiunile următoare? Ce mod de adresare este folosit? Se considera BX=12ABh, BP=1300h, SI=1A2Bh, DI=340Ch, DS=2100h, SS=3F00h, CS=5A00h.

```
mov AX, [BX+3]
mov AX, 4[BX+SI]
mov AX, [SI]+10h
mov AX, 5[BP]
mov AX, [DI+8]
mov AX,2345h
```

7. Ştiind CS=1200h, SS=3F00h, IP=2000h şi SP=2002h, specificaţi adresele logice şi fizice pentru a) elementul din vârful stivei şi b) instrucţiunea curentă.

8. 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
- 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
- mai precis, operandul este cuvântul din memorie, de la adresa 1200h:1A3Bh (partea LOW) şi 1200h:1A3Ch (partea HIGH)

```
mov [DI], AX
mov [SI], AL
mov AX, [SI+BX]
mov AL, [BX] [SI]
mov AX, DS: [BP+2]
```

9. Precizaţi dacă următoarele instrucţiuni sunt corecte şi justificaţi răspunsul:

```
mov AX, [BP+BX]
mov AL, [BX]
mov AX, [BP+SI]
mov AX, [SI+DI]
mov [SI+DI], AX
mov [SI+BP], AX
mov AX, [BP+5]
```

```
mov AX, 7[BP]
mov AX, [6]
```

10. Analizaţi secvenţa de mai jos şi scrieţi atât rezultatul obţinut cât şi codificarea pentru următoarele secvenţe de instrucţiuni; comparaţi rezultatele obţinute între ele, specificând efectul fiecărei instrucţiuni în parte:

```
org 100h
.data ; în zona de date se definesc cele 2 variabile de mai jos
sir1 db 1,2,3,4,5,6
sir2 dw 7,8,9,10,11,12,13,14,15

.code
mov al, sir1 [2];   _______          mov bl, sir1 [3]; ________

mov ax, sir2 [2];  ________          mov bx, sir2 [3]; ________

mov sir1 [2], ah ; _______          mov sir1 [3], bh; ________

mov sir2 [2], ax ; ________          mov sir2 [3], bx; ________
```

11. Fie o variabilă definită în memorie *sir db 0,1,2,3,4,5,6,7,8,9.* Scrieţi câte un exemplu de instrucţiune *mov* folosind adresare a) bazată, b) indexată şi c) bazată-indexată astfel încât să se depună în registrul AL al 5-lea element al şirului.

12. Precizaţi dacă următoarele instrucţiuni sunt corecte şi justificaţi răspunsul:

```
mov BX, [AX]
mov AX, [BX]
mov AX, [CX+4]
mov AL, sir[3+BX]
mov AX, sir[3+BL]
mov AL, sir [CH+5]
mov BL, sir 2[BH]
mov AX, [DX]
```

13. Folosind regulile specificate în material, specificaţi modul de codificare al următoarelor instrucţiuni:

```
mov [BX+DI+1234h], AL
mov [BX+DI+1234h], AX
mov AX, [BX+DI+1234h]
mov BX,CX
mov BH,CH
mov BL,CH
mov BP,SP
mov [BX+10],5678h
```

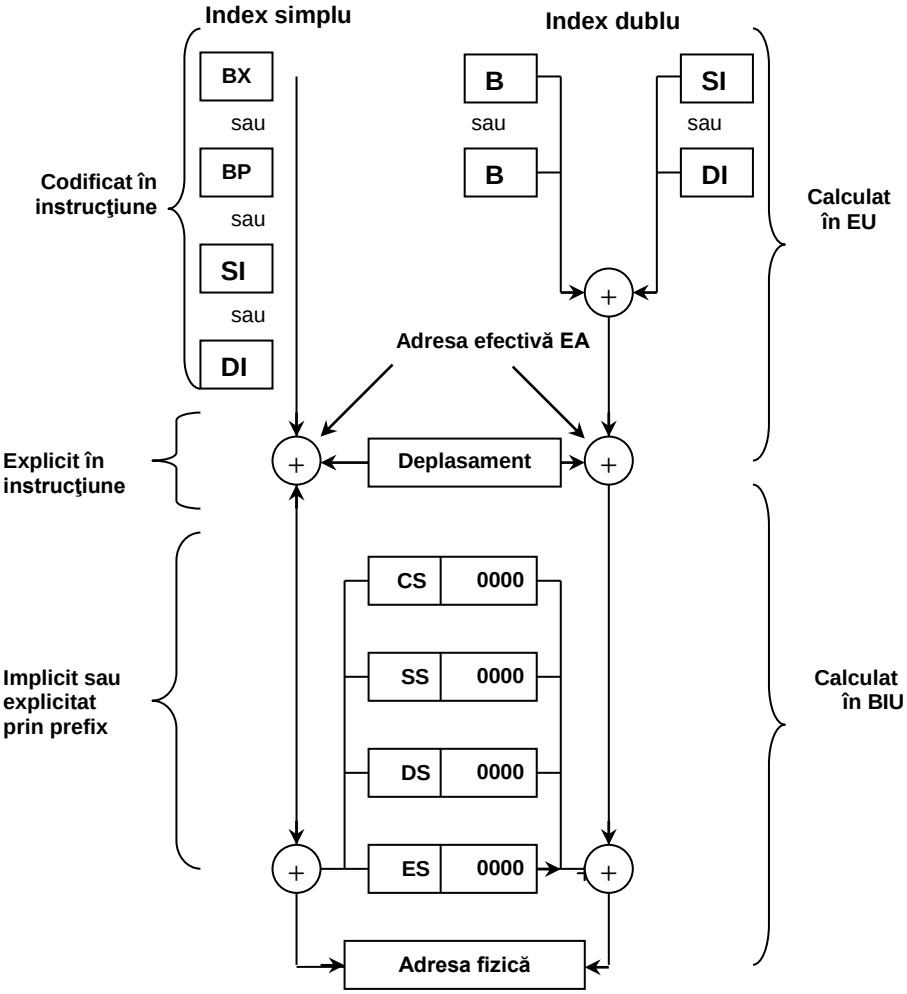


Fig. 2.3. Variante de calcul ale adresei fizice de memorie

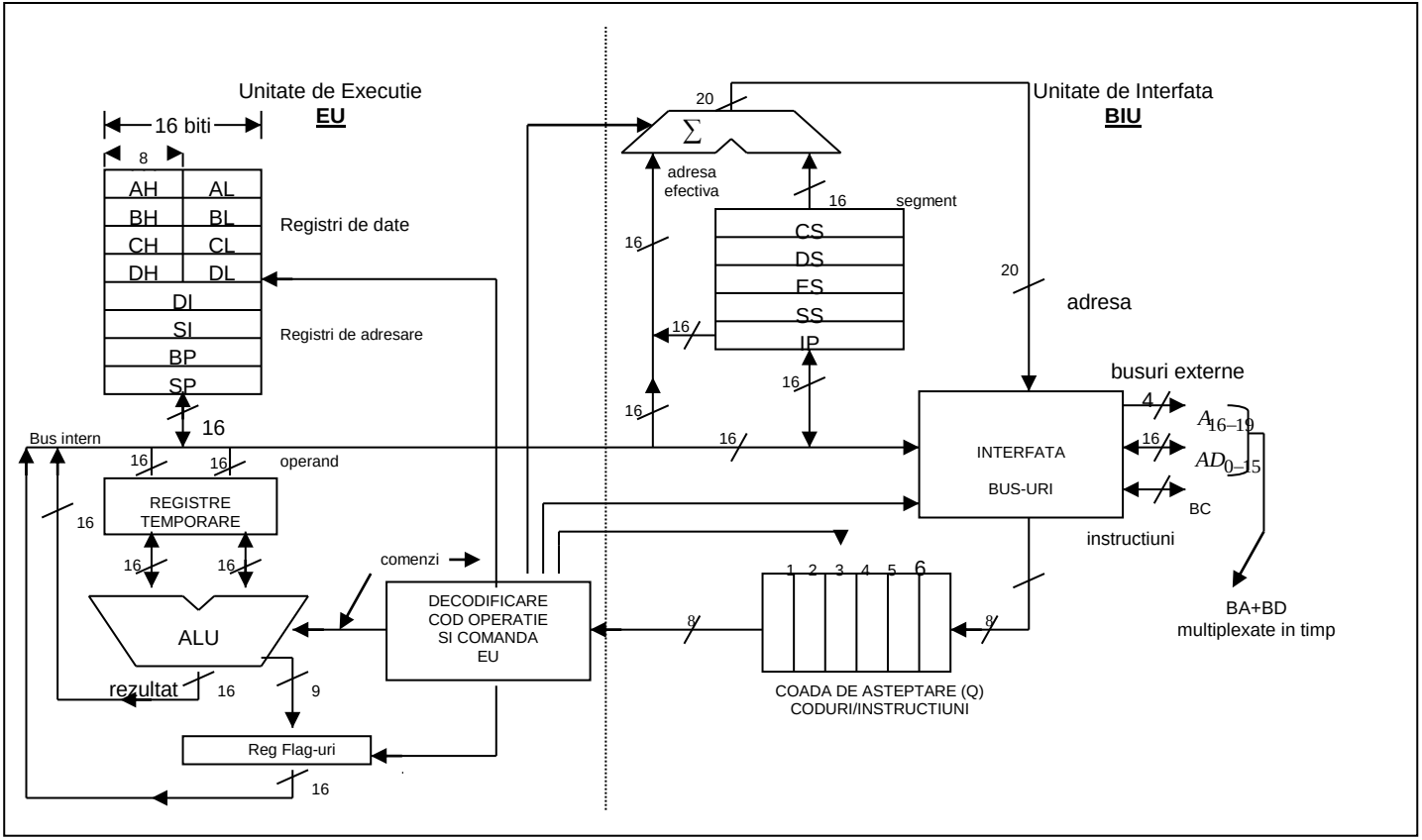


Fig. 2.3 Schema bloc internă simplificată a microprocesorului 8086

3. Simulator de microprocesor (I)

Simulatorul Emu8086 (www.emu8086.com) este un emulator pentru microprocesor ce conţine (are integrat) un Asamblor 8086. Emulatorul rulează programele pe o Maşină Virtuală, emulând un hardware real precum ecranul monitorului, memoria şi dispozitivele de intrare/ieşire. Setul de instrucţiuni 8086 stă la baza tuturor microprocesoarelor, inclusiv Pentium şi Athlon. Toate instrucţiunile Intel, chiar şi directive precum MASM şi TASM sunt suportate de Emu8086, acesta oferind o soluţie completă în învăţarea limbajului de asamblare. Emu8086 rulează programele ca un microprocesor 8086 real: codul sursă este asamblat şi executat de către emulator pas cu pas, existând posibilitatea de a urmări modificările apărute în regiştri, flag-uri şi memorie în timpul rulării programelor.

Emu8086 include şi un Tutorial, plus o mulţime de programe date ca model. Emu include de asemenea şi câteva dispozitive externe virtuale, precum un robot, un motor pas cu pas, afişaj cu led-uri, intersecţie gestionată de semafoare, etc; aceste dispozitive pot fi modificate sau clonate, codul lor sursă fiind disponibil. Emulatorul permite crearea unui nou proiect (opţiunea **new**), vizualizarea unor exemple deja existente (opţiunea **code examples**), urmărirea tutorialului (opţiunea **quick start tutor**) sau deschiderea fişierelor recent utilizate (opţiunea **recent files**) aşa cum se poate urmări în figura 3.1.

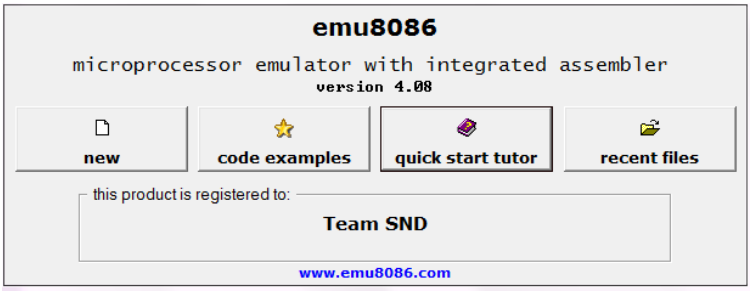


Fig. 3.1. Fereastra de început a Emu8086

După scrierea codului sursă (al aplicaţiei) acesta poate fi compilat, obţinându-se astfel un fişier binar cu extensia .bin ce poate fi salvat şi apoi executat.

3.1. Generalităţi despre Emu8086

Structura de bază a unui computer este prezentată în figura 3.2; se pot observa arhitectura şi componentele principale ale unui sistem de calcul.

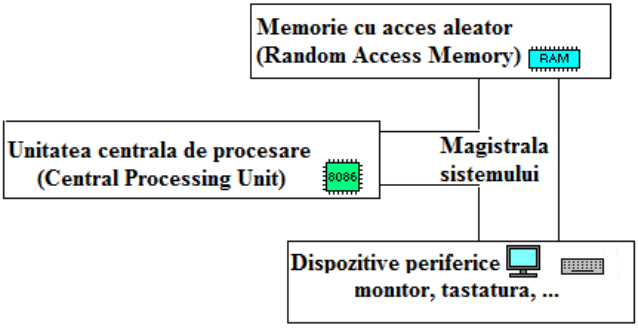


Fig.3.2. Arhitectură tipică de computer

Unitatea centrală de procesare (CPU) este "creierul" computerului. Toate calculele, deciziile şi mişcările datelor se realizează aici. CPU are în componenţă locaţii de stocare specifice numite *regiştri* şi *o unitate aritmetică şi logică* (ALU) unde se realizează procesările. Datele sunt luate din regiştri, procesate, iar rezultatele sunt stocate tot în regiştri. Există mai multe instrucţiuni, fiecare având un scop precis; colecţia tuturor instrucţiunilor se numeşte ***set de instrucţiuni***.

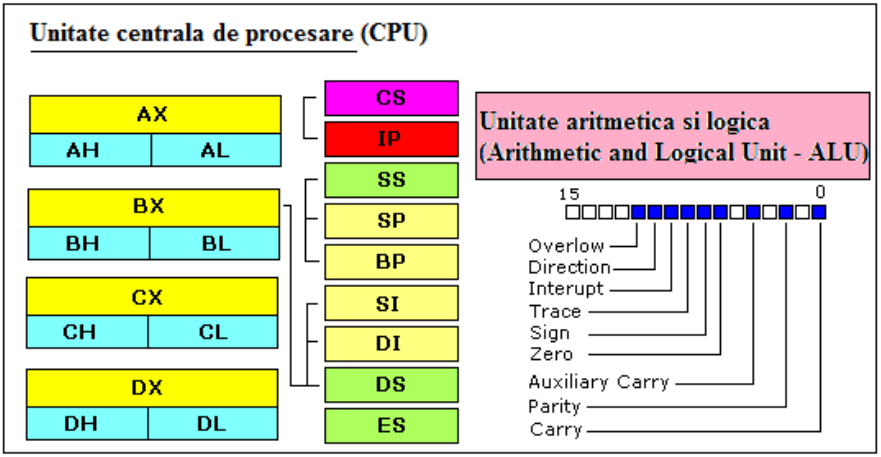


Fig.3.3. Unitatea centrală de procesare

CPU a familiei 8086 are 8 regiştrii de uz general, după cum urmează:

- **AX** - registru *acumulator* (divizat în **AH / AL**).
- **BX** – registru adresă de *bază* (divizat în **BH / BL**).
- **CX** – registru *contor* sau numărător (divizat în **CH / CL**).
- **DX** – registru de *date* (divizat în **DH / DL**).

- **SI** – registru *index sursă*.
- **DI** – registru *index destinaţie*.
- **BP** – pointer la *baza stivei*.
- **SP** – pointer la *stivă (adresa curentă)*.

Regiştrii de uz general de date sunt **AX, BX, CX, DX**; mărimea lor fiind de 16 biţi, ei pot păstra numere fără semn în domeniul 0..+65535 sau numere cu semn în domeniul –32768..+32767. Aceştia se folosesc ca locaţii temporare, mai degrabă decât ca locaţii de memorie, pentru că accesul la regiştrii este mai rapid. Regiştrii AL, BL, CL, DL şi AH, BH, CH, DH sunt părţile low şi high ale regiştrilor corespunzători pe 16 biţi.
*Regiştrii **DI** şi **SI*** sunt regiştri index (Destination Index şi Source Index) destinaţi lucrului cu şiruri de octeţi sau cuvinte.
*Regiştrii **SP** şi **BP*** sunt regiştri destinaţi lucrului cu stiva. Stiva se defineşte ca o zonă de memorie (LIFO – Last In First Out) în care pot fi depuse valori, extragerea lor ulterioară realizându-se în ordine inversă depunerii lor.
SP – Stack Pointer – pointează spre ultimul element introdus în stivă, iar
BP – Base Pointer – pointează către baza stivei.

Regiştrii cu scop special sunt **IP** şi **PSW**.
Registrul **IP** conţine adresa instrucţiunii curente care se execută; după ce s-a executat instrucţiunea curentă, IP este incrementat automat pentru a pointa spre instrucţiunea următoare. Instrucţiunile de salt modifică valoarea acestui registru astfel încât execuţia programului se mută la o nouă poziţie. Registrul IP se mai numeşte şi PC (Program Counter).
PSW (Program Status Word) este un registru ce conţine flagurile (1 bit fiecare) ce raportează starea CPU după execuţia fiecărei instrucţiuni. Aceşti regiştri cu scop special nu pot fi accesaţi direct, ei fiind modificaţi de către CPU pe durata execuţiei instrucţiunii.

Principalii indicatori de stare sunt: **C, S, O, Z**:
C (Carry) indică un transport în afara domeniului de reprezentare al rezultatului;
S (Sign) are valoarea 1 când rezultatul ultimei operaţii este un număr strict negativ, deci copiază bitul MSB al rezultatului;
O (Overflow) indică depăşire de gamă: dacă rezultatul ultimei instrucţiuni a depăşit spaţiul rezervat rezultatului, în cazul operanzilor consideraţi numere cu semn;
Z (Zero) are valoarea 1 dacă rezultatul ultimei instrucţiuni este egal cu zero.

Procesorul 8086 conţine 4 regiştri segment (**CS, DS, ES** şi **SS**), pe 16 biţi fiecare. Aceştia se folosesc pentru a selecta blocuri (numite segmente) din memorie. Regiştrii CS (Code Segment), DS (Data Segment), SS (Stack Segment) şi ES (Extra Segment) din BIU reţin adresele de început ale segmentelor active, corespunzătoare fiecărui tip de segment.

CPU poate accesa până la 1 MB de memorie RAM, adresele RAM fiind date uzual între paranteze drepte; de exemplu [7Ch] se citeşte ca "datele de la locaţia cu adresa 7Ch", unde 7Ch este un număr hexazecimal.

Magistralele sunt seturi de linii folosite pentru transportul semnalelor. Un computer pe 16 biţi are uzual regiştri pe 16 biţi şi 16 fire/linii într-un bus (magistrală).
Bus-ul de date se foloseşte pentru transportul datelor între CPU, RAM şi porturile I/O. Simulatorul are un bus de date de 16 biţi.
Bus-ul de adrese se foloseşte pentru a specifica ce adresă RAM sau port I/O va fi folosit. Simulatorul are un bus de adrese de 16 biţi.
Bus-ul de control are o linie pentru a determina dacă se accesează RAM sau porturi I/O; are de asemenea o linie pentru a determina dacă datele sunt scrise sau citite: CPU citeşte date când acestea intră în CPU şi scrie date când acestea ies din CPU către RAM sau porturi.
Ceasul sistem constă în impulsuri periodice generate astfel încât componentele să se sincronizeze. Simulatorul lucrează cu o viteză de cateva instrucţiuni pe secundă, ajustabilă în limite mici.

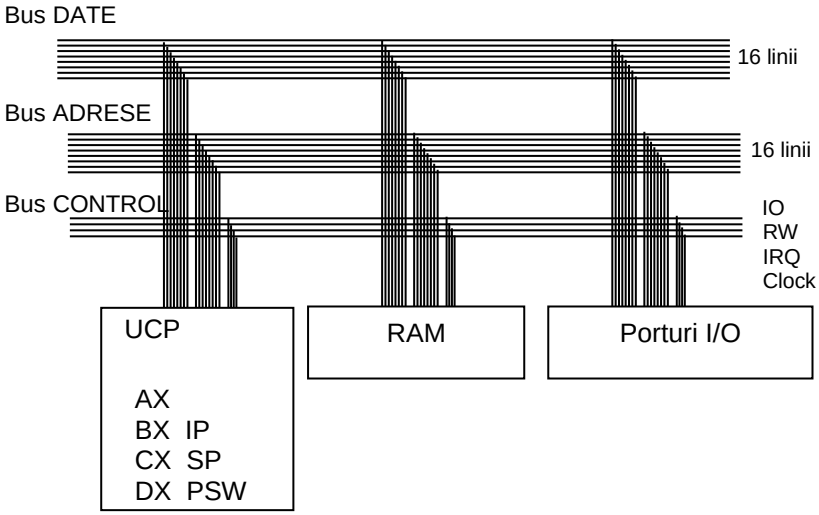


Fig. 3.4. Arhitectură tipică de calculator

3.2. Caracteristicile emulatorului:

- CPU de 16 biţi
- Pe 16 biţi se pot adresa 2^{16} porturi I/O;
- Periferice simulate pe unele dintre aceste porturi
- Asamblor
- Help on-line

- Rulare pas cu pas a programului
- Rulare continuă a programului
- Posibilitatea de modificare a ceasului procesor

Utilizarea emulatorului

La pornirea emulatorului, va apărea o fereastră asemănătoare celei din figura 3.5, în care se poate observa zona de editare a programului sursă (fişier de tip asm), iar în partea de sus meniul cu opţiunile: file, edit, bookmarks, assembler, emulator, math, ascii codes, help.

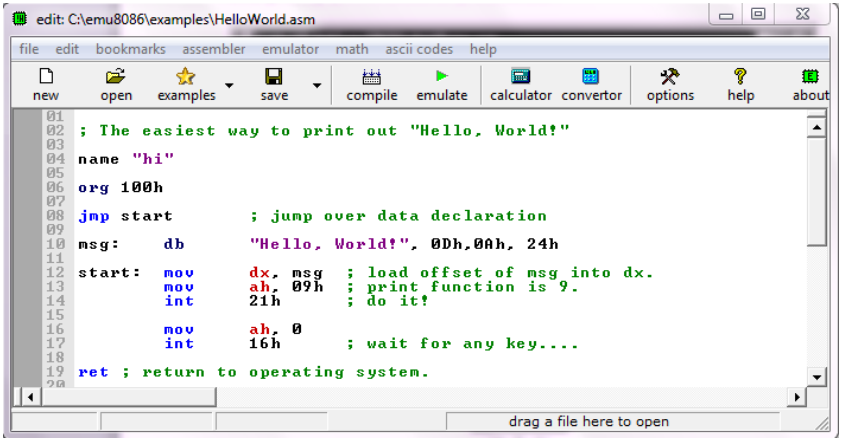
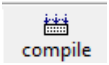


Fig.3.5. Fereastra de editare a emulatorului

Pentru a scrie şi a rula un program nou folosind emulatorul, se va folosi optiunea „new” şi se va selecta un şablon de tip COM, aşa cum se poate urmări în figura 3.6.

Codul scris se numeşte *limbaj de asamblare* (*.asm), iar trecerea lui într-un limbaj care să fie înţeles de către CPU se obţine prin compilare, cu opţiunea



În urma acestei operaţii, va rezulta un fişier de tip .com ce se va salva local, iar dacă se doreşte încărcarea codului în emulator, se va folosi butonul

După încărcarea programului în emulator, va apare fereastra din figura 3.7 în care se pot urmări în partea de jos, dinspre stânga spre dreapta, în cele 3 zone distincte:

1) regiştrii emulatorului,

2) adresa fizică pe 20 de biţi (exprimată cu 5 cifre hexazecimale) şi conţinutul de la acea adresă (specificat în hexazecimal, zecimal şi Ascii)

3) codul sursă al programului în limbaj maşină, înainte de a fi asamblat.

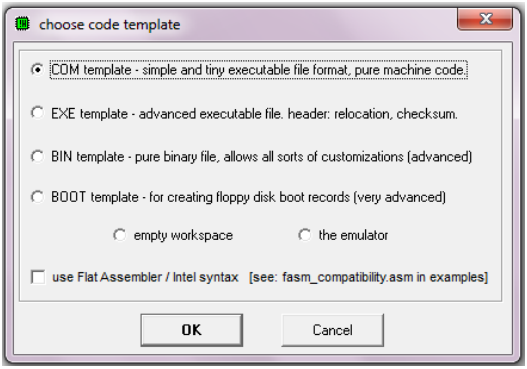


Fig. 3.6. Alegerea şablonului pentru un nou program

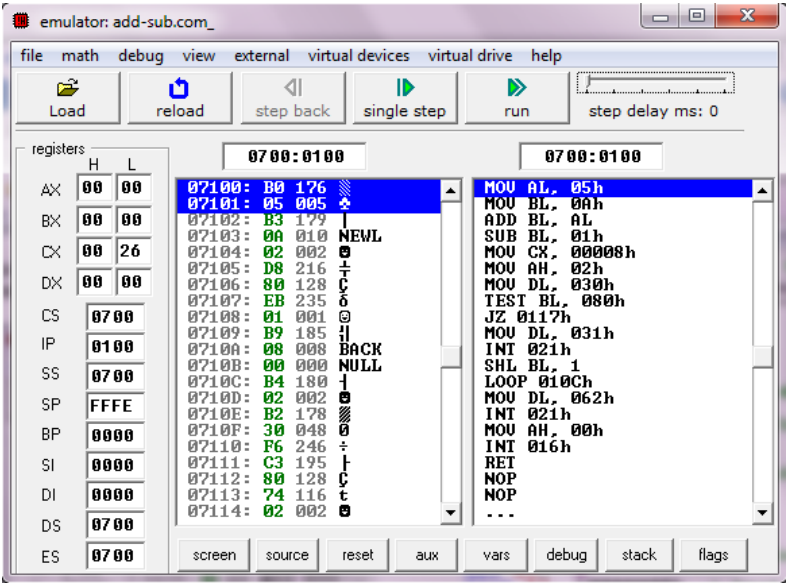
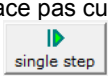
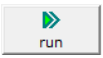


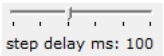
Fig.3.7. Incărcarea unui fişier în emulator

Rularea unui program se poate face pas cu pas, apăsând în mod repetat



sau în mod continuu, cu



Viteza de execuţie poate fi selectată cu ajutorul cursorului 

In figura 3.7, dacă se va da dubluclick în caseta corespunzătoare regiştrilor, se va deschide o nouă fereastră *Extended value viewer*, ce conţine valoarea acelui registru exprimată în hexazecimal, binar şi zecimal. Prin intermediul acestei ferestre, valoarea din registru poate fi modificată direct, în timpul rulării. O operaţie de dubluclick asupra unei zone din memorie, va lista cuvântul din memorie aflat la locaţia selectată. Reamintim că octetul cel mai puţin semnificativ se găseşte la adrese mai mici (conform convenţiei *Little END-ian*).

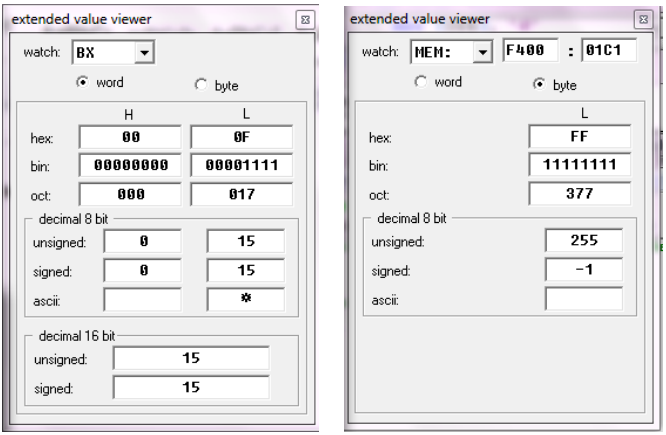


Fig.3.8. Fereastra de vizualizare şi modificare a regiştrilor (stânga) sau a unei zone din memorie (dreapta)

Emulatorul rulează programele pe un computer virtual, fapt care blochează accesarea hardware-ului real, precum driverele hard şi memoria.

În urma execuţiei fiecărei instrucţiuni, putem urmări modificarea conţinutului regiştrilor din CPU. De asemenea, există posibilitatea de a vizualiza conţinutul memoriei sau al ALU, valorile flag-urilor, aşa cum se poate urmări în figurile 3.9 şi 3.10. Selectarea informaţiei dorite spre vizualizare se realizează din meniul View al emulatorului (figura 3.7, sus)

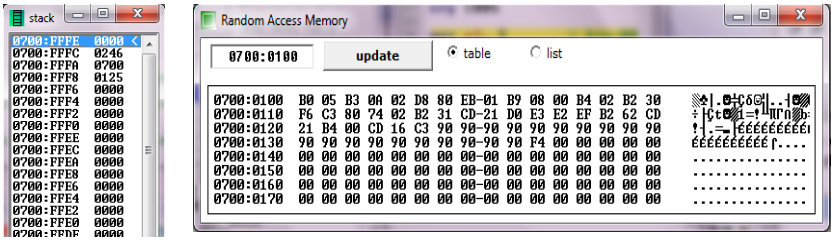


Fig.3.9. Ferestre de vizualizare a conţinutului stivei şi al memoriei

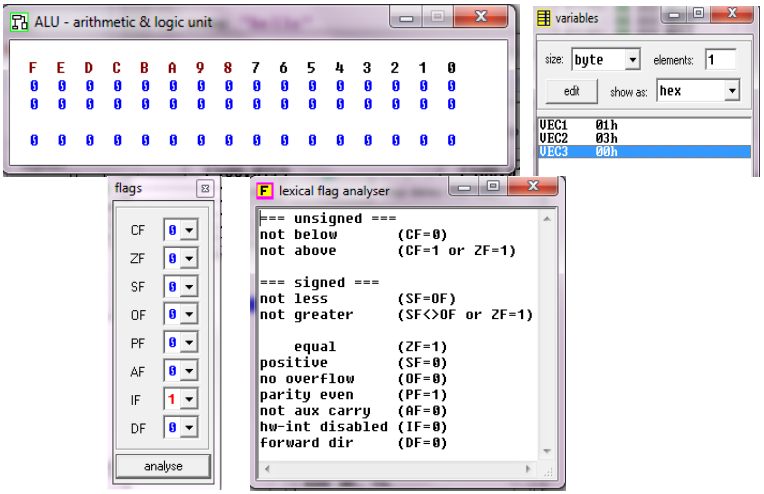


Fig.3.10. Ferestre de vizualizare a conţinutului ALU, al variabilelor definite în memorie şi al flag-urilor

Ecranul emulatorului (obţinut tot din meniul View, fereastra din figura 3.7.) poate fi folosit pentru datele de ieşire, fiind suportat şi modul color.

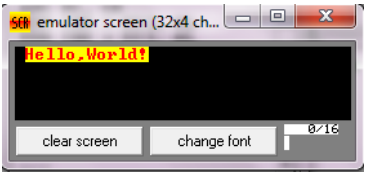


Fig.3.11. Fereastra de vizualizare a ecranului

Din meniul Math, se pot deschide ferestrele din fig. 3.12, corespunzătoare:
- calculatorului (*expression evaluator*) ce poate fi folosit pentru operaţii logice şi aritmetice cu valori hexazecimale, octale, binare şi zecimale şi resp.
- convertorului (*base convertor*), prin care numerele pot fi convertite dintr-o bază de numeraţie în alta.

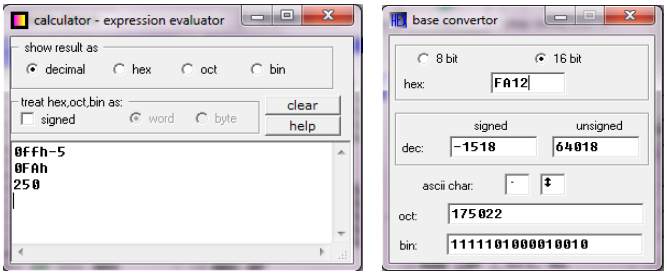


Fig.3.12. Ferestrele calculatorului şi convertorului

Emu8086 include câteva dispozitive virtuale cu ajutorul cărora se pot realiza experimente: acestea includ un termometru, un ecran virtual, un sistem semafor, un motor pas cu pas, un robot, un afişaj cu cristale lichide, o imprimantă şi altele, cu interfeţe aşa cum se poate urmări în figura 3.13.

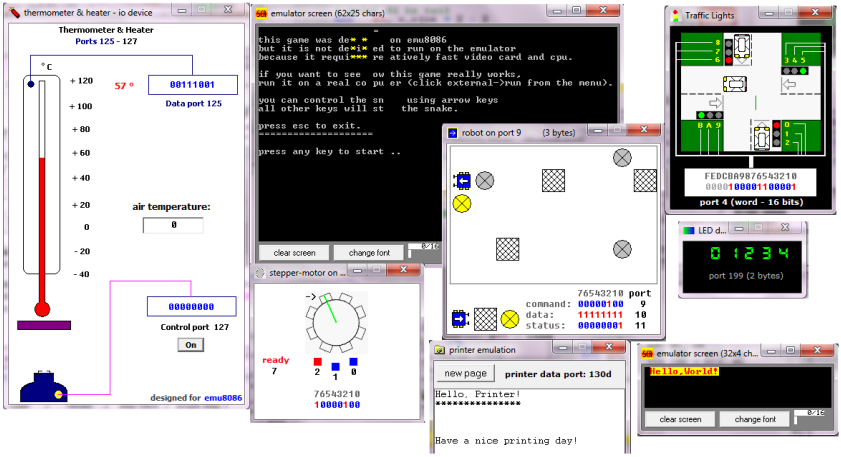


Fig.3.13. Dispozitive virtuale existente în emulator

Exemplele prezentate în tutorial reprezintă o introducere pas cu pas în comenzile şi tehnicile de programare de nivel scăzut.

3.3 Aplicaţii

Fiecare program are unul sau mai multe exerciţii asociate, unele dintre ele simple, altele mai complexe.

3.3.1 Operaţii aritmetice simple: **2_sample.asm**

Programul **2_sample.asm** foloseşte instrucţiunile MOV, ADD, SUB. Pentru a rula programul apăsaţi Single Step până la terminarea programului. Observaţi modificările din regiştri în timpul rulării pas cu pas. În momentul modificării valorii într-un registru, acest lucru este semnalat prin colorare în albastru.

Observaţii:

Comentarii – orice text aflat după ";" nu este parte a programului şi este ignorat de către simulator. Comentariile se folosesc pentru a explica ce face programul. Un bun programator foloseşte multe comentarii, iar acestea nu trebuie să repete pur şi simplu codul, ci să furnizeze explicaţii suplimentare.

Directiva *name „nume":* name "add-sub" este folosită pentru a specifica numele fişierului de tip .com ce se va crea în urma compilării.

Directiva *org valoare:* org 100h se foloseşte pentru a specifica adresa la care se va asambla programul. Valoarea 100h este 256 în zecimal.

MOV – este prescurtarea pentru operaţia de mutare MOVE. În acest exemplu, numerele sunt copiate în regiştri pentru a putea realiza operaţia aritmetică. MOV copiază datele dintr-o locaţie în alta, datele de la locaţia iniţială rămânând nemodificate.
Aritmetica – ADD se foloseşte pentru a aduna doi operanzi, în cazul de faţă conţinutul a doi regiştri. O altă versiune este folosirea sa pentru a aduna un număr la un registru. Rezultatul adunării se va depune în operandul destinaţie, cel din stânga. SUB se foloseşte pentru a scădea conţinutul a doi regiştri: din operandul din stânga se scade cel din dreapta, rezultatul fiind depus apoi în cel din stânga.
END – dacă apare în program orice text după această directivă va fi ignorat, END fiind ultima comandă, specifică emulatorului.

Programul **2_sample.asm** este prezentat în continuare:

```
name "add-sub"
org 100h
;______CALCULEAZĂ 5 PLUS 10 ŞI 15 MINUS 1_____________________

       MOV AL, 5      ;copiază în reg. AL valoarea 5=00000101b
       MOV BL, 10     ;copiază în reg. BL valoarea 10=00001010b
       ADD BL, AL     ;5+10=15 (000Fh) valoare stocată în BL
       SUB BL, 1      ;15-1=14 (000Eh) valoare stocată în BL
```

Programul continuă cu instrucţiuni necesare afişării rezultatului în binar, care vor fi ignorate în lucrarea de faţă şi se vor studia într-o lucrare ulterioară. Pentru a studia aritmetica, se vor consulta valorile regiştrilor aşa cum se sugerează în figura 3.8. Pentru a urmări rezultatele obţinute, se poate consulta conţinutul ALU, aşa cum arată figura 3.9. De asemenea, se pot folosi calculatorul şi convertorul din figura 3.11 pentru verificarea corectitudinii calculelor.

Exerciţii şi teme (I)

1. Scrieţi un program care realizează scăderea a două numere, folosind instrucţiunea SUB;
2. Scrieţi un program care realizează înmulţirea a două numere, folosind instrucţiunea MUL;
3. Scrieţi un program care realizează împărţirea a două numere, folosind instrucţiunea DIV;
4. Scrieţi un program care realizează împărţirea la zero. Amintiţi-vă să nu faceţi această împărţire altă dată.

Majoritatea programelor prezentate includ o parte de exerciţii pentru învăţare, pentru a garanta înţelegerea exemplului. În acest caz, puteţi folosi toţi regiştrii de 8 biţi sau de 16 biţi şi operaţiile ADD, SUB, MUL şi DIV. Studiaţi aceste instrucţiuni (din Help-ul simulatorului) înainte de a le utiliza. Observaţi ce se întâmplă dacă încercaţi să faceţi împărţirea la zero.

3.3.2. Intrări şi ieşiri de date: **simple_io.asm**

Programul **simple_io.asm** se studiază pentru a arăta cum se pot accesa porturile (virtuale), având adresele posibile de la 0 la 0FFFFh. Programul foloseşte instrucţiuni specifice porturilor de ieşire (OUT) şi de intrare (IN) şi date de dimensiuni diferite: când intervine registrul AL, datele sunt pe octet, iar când se foloseşte registrul AX, datele sunt pe cuvânt. Pentru a rula programul, apăsaţi Single Step până la terminarea programului. Urmăriţi fereastra corespunzătoare portului virtual:

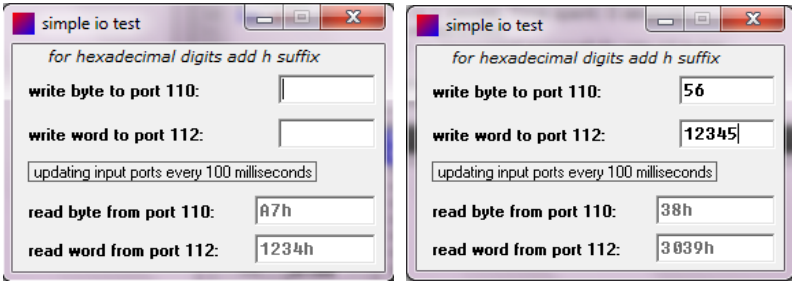


Fig.3.14. Interfaţa aplicaţiei Simple_io înainte şi după introducerea datelor de la utilizator

Observaţii:

OUT val,AL sau OUT val,AX – această instrucţiune trimite conţinutul registrului AL sau AX (deci octet sau cuvânt), la portul de ieşire cu adresa *val*. Circuitul virtual este legat pe portul de ieşire 110 (pentru octet) sau 112 (pentru cuvânt).

IN AL, val sau *IN AX, val* – preia un octet sau un cuvânt de la portul cu adresa *val*. Emulatorul aşteaptă introducerea unui octet sau cuvânt şi scrie (eventual şi transformă) valoarea în hexazecimal în registrul AL sau AX.

Programul **simple_io.asm** este prezentat în continuare:

```
#start=simple.exe#
#make_bin#
name "simple"

;______CITESTE OCTET şi CUVANT DE LA PORT_____________________
       MOV AL, 0A7h   ;scrie octetul 0A7h la portul 110
       OUT 110, AL

       MOV AX, 1234h  ;scrie cuvântul de valoare 1234h la
       OUT 112, AX    ;portul cu adresa 112

       MOV AX, 0      ;reset registru.

       IN AL, 110     ;citeşte în AL octet de la portul 110
       IN AX, 112     ;citeşte în AX cuvant de la portul 112
```

Exerciţii şi teme (II)

1. Modificaţi programul a.î. valoarea implicită pe port pentru octet să fie 12h.
2. Modificaţi programul astfel încât valoarea implicită pe port pentru cuvânt să fie 3456h.

3.3.3. Utilizarea numerelor hexazecimale: **traffic_lights2.asm**

Programul **traffic_lights2.asm** foloseşte instrucţiunile MOV, OUT, ROL şi JMP. Luminile semafoarelor sunt controlate prin trimiterea datelor la portul 4, deci se va folosi instructiunea *OUT 4, AX*.

Avem de controlat 12 becuri in figura 3.15: roşu, galben, verde (în această ordine) pentru cele 4 semafoare (primul-cel de jos, al doilea-cel din dreapta, al treilea-cel de sus, al patrulea-cel din stânga). Acest lucru poate fi realizat cu ajutorul a doi octeţi, deci 16 biţi, din care nu vom folosi cei mai semnificativi 4 biţi. Prin setarea unui bit la 1, becul corespunzător se aprinde.

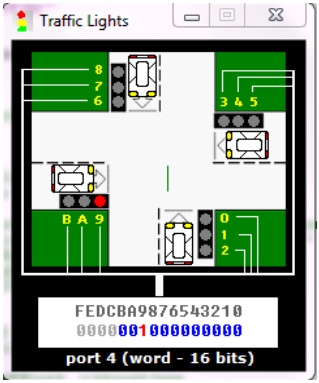


Fig.3.15. Interfaţa aplicaţiei semafor (Traffic Lights)

Observaţii:

Etichete şi instrucţiunea JMP – etichetele marchează în program poziţii care vor fi folosite de comenzile de salt. În acest program, toate instrucţiunile sunt repetate la infinit sau până la apăsarea butonului Stop. Numele de etichete trebuie să înceapă cu o literă sau cu caracterul "_", în nici un caz cu o cifră.

O instrucţiune de genul *JMP start* va cauza un salt în program la eticheta *start* şi reluarea instrucţiunilor cuprinse între cele 2. Eticheta destinaţie a saltului se încheie cu ":", de exemplu *start*:

OUT 4, AX – această instrucţiune copiază conţinutul registrului AX la portul de ieşire 4, unde este legat circuitul de semafoare. Un 1 binar are ca efect aprinderea becului, iar un 0 stingerea lui.

Controlul becurilor – se poate observa din figura 3.15 care este becul (din cele 12) controlat de fiecare bit. Prin setarea bitului corespunzător (binar) şi transformarea în hexazecimal a întregului număr, se poate schimba modul de funcţionare al semafoarelor.

Directiva *EQU* este folosită pentru a defini constante, de exemplu *red EQU 0000_0001b* defineşte constanta red cu valoarea binară 0000_0001b.

Instrucţiunea *nop* provine de la *no operation* şi introduce o întârziere în prelucrarea datelor.

ROL – bitul MSb trece atât în C (Carry) cât şi în bitul LSb din operand.

Instructiunea *ROL* este de fapt cea care se execută atunci când apare semnul „<<”. De exemplu: *MOV AX, green << 3* va încărca în registrul AX valoarea constantei green, deplasată spre stânga cu 3 poziţii; deci 0000_0000_0000_0100b va deveni 0000_0000_0010_0000b; astfel, bitul 5 va deveni 1, adică becul verde de la al doilea semafor va fi aprins.

Instrucţiunea jmp start asigură rularea (execuţia) continuă a programului.

```
#start=Traffic_Lights.exe#
name "traffic2"

;____CONTROLUL BECURILOR SEMAFORULUI _2________
       yellow_and_green      equ     0000_0110b
       red                   equ     0000_0001b
       yellow_and_red        equ     0000_0011b
       green                 equ     0000_0100b
       all_red               equ     0010_0100_1001b

start:
       nop

       mov ax, green                 ; controlează biţii 0,1,2
       out 4, ax
       mov ax, yellow_and_green
       out 4, ax
       mov ax,red
       out 4, ax
       mov ax, yellow_and_red
       out 4, ax

       mov ax, green << 3            ; controlează biţii 3,4,5
       out 4, ax
       mov ax, yellow_and_green << 3
       out 4, ax
       mov ax,red << 3
       out 4, ax
       mov ax, yellow_and_red << 3
       out 4, ax
```

```
        mov ax, green << 6            ; controlează biţii 6,7,8
        out 4, ax
        mov ax, yellow_and_green << 6
        out 4, ax
        mov ax,red << 6
        out 4, ax
        mov ax, yellow_and_red << 6
        out 4, ax
        mov ax, green << 9            ; controlează biţii 9,A,B
        out 4, ax
        mov ax, yellow_and_green << 9
        out 4, ax
        mov ax,red << 9
        out 4, ax
        mov ax, yellow_and_red << 9
        out 4, ax

        mov ax, all_red       ;aprinde toate becurile roşii
        out 4, ax
        mov ax, all_red << 1  ;aprinde toate becurile galbene
        out 4, ax
        mov ax, all_red << 2  ;aprinde toate becurile verzi
        out 4, ax
jmp start
```

Exerciţii şi teme (III)

1. Modificaţi programul astfel încât secvenţa de funcţionare a becurilor semaforului să fie realistă; se vor da comenzi astfel (într-un singur program):

1 - se vor aprinde becurile roşii de la semafoarele 2 şi 4, împreună cu becurile verzi de la semafoarele 1 şi 3, apoi

2 - se vor aprinde becurile roşu şi galben de la semafoarele 2 şi 4, împreună cu becurile galbene de la semafoarele 1 şi 3, apoi

3 - se vor aprinde becurile verzi de la semafoarele 2 şi 4, împreună cu becurile roşii de la semafoarele 1 şi 3, apoi

4 - se vor aprinde becurile galbene de la semafoarele 2 şi 4, împreună cu becurile roşii şi galbene de la semafoarele 1 şi 3 şi apoi ciclul se reia.

2. Propuneţi o altă funcţionare pentru semafor, astfel încât să nu ducă la ciocniri de autovehicule.

3. Consideraţi introducerea de întârzieri în execuţia secvenţelor de instrucţiuni.

4. Simulator de microprocesor (II)

4.1. Definirea datelor în memorie: **traffic_lights.asm**

Programul **traffic_lights2.asm** foloseşte instrucţiunile MOV, OUT, ROL şi JMP. Luminile semafoarelor sunt controlate prin trimiterea datelor la portul 4, deci se va folosi instructiunea *OUT 4, AX*. Avem de controlat 12 becuri în figura 4.1: roşu, galben, verde (in aceasta ordine) pentru cele 4 semafoare.

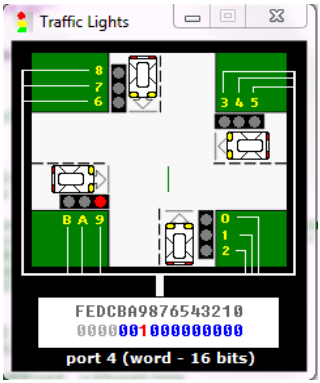


Fig.4.1. Interfaţa aplicaţiei semafor Traffic Lights

O variantă îmbunătăţită a programului **traffic_lights2.asm** prezentată în lucrarea anterioară, poate fi urmărită în programul **traffic_lights.asm,** care este prezentat în continuare:

```
 #start=Traffic_Lights.exe#
name "traffic"
;_______CONTROLUL BECURILOR SEMAFORULUI__________

        MOV AX, all_red
        OUT 4, AX
        MOV SI, offset situation
next:   MOV AX, [SI]
        OUT 4, AX

        ; secvenţa necesară introducerii unei întârzieri:
        MOV     CX, 4CH         ; asteaptă 5 secunde
        MOV     DX, 4B40H       ; 004C4B40h = 5,000,000
        MOV     AH, 86H
        INT     15H
        ADD SI, 2               ; trece la situaţia următoare
        CMP SI, sit_end         ; se verifică dacă s-a ajuns la
                                ; sfârşit
        JB  next                ; dacă nu, sare la eticheta next
        MOV SI, OFFSET situation  ;dacă da, reia de la început
        JMP next                ; salt necondiţionat la et.next
```

```
                                ;FEDC_BA98_7654_3210
situation       dw      0000_0011_0000_1100b
s1              dw      0000_0110_1001_1010b
s2              dw      0000_1000_0110_0001b
s3              dw      0000_1000_0110_0001b
s4              dw      0000_0100_1101_0011b
sit_end = $
all_red         equ     0000_0010_0100_1001b
```

Observaţii:

Programul foloseşte un tabel de date, creat prin intermediul directivei dw (define word). Secvenţa de mai sus defineşte *o zonă de memorie*, începând de la cuvântul adresabil prin numele „situation” de valoare 030Ch, urmat apoi de cuvântul s1 de valoare 069Ah, apoi s2 de valoare 0861h şi aşa mai departe, încheindu-se cu caracterul „$”.

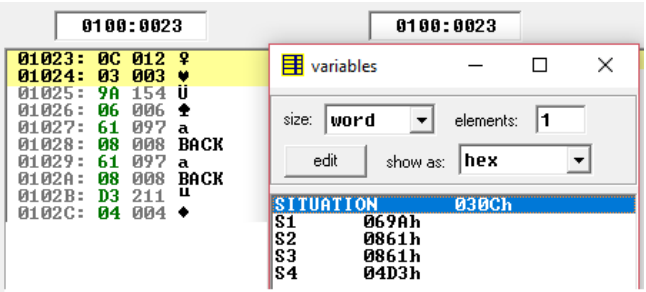


Fig.4.2. Vizualizarea zonei de memorie pentru aplicaţia semafor

Exerciţii şi teme (I)

1. Analizaţi programul şi variabilele definite în memorie. Urmăriţi funcţionarea semaforului, executând programul cu „run”.

2. Modificaţi variabilele din memorie astfel încât secvenţa de funcţionare a becurilor semaforului să respecte următorul algoritm: un singur semafor va afişa culoarea verde; semaforul opus celui „activat” pe verde, va afişa culoarea galben, iar celelalte 2 semafoare adiacente vor afişa culoarea roşu; apoi, următorul semafor va funcţiona după algoritmul propus şi tot aşa, pe rând, până la funcţionarea fiecăruia din cele 4 semafoare după algoritmul propus. Bucla se reia apoi şi continuă până la apăsarea unei taste, când se asigură ieşirea din program.

3. Propuneţi o secvenţă care să fie formată din cel puţin încă 2 variabile în plus. Observăti aceste variabile în memorie, în ambele moduri, aşa cum se sugerează în figura 4.2.

4.2. Instrucţiuni de comparare: meniul **examples**, **compare numbers**

Instrucţiunea **CMP op1,op2** funcţionează în felul următor: procesorul calculează diferenţa op1-op2 şi în funcţie de rezultat, setează flagurile (indicatorii):
Z dacă rezultatul este zero (deci dacă dacă op1=op2),
S dacă rezultatul este negativ (deci dacă op1<op2),
iar dacă op1 este mai mare decât op2 nu se setează nici unul.

Flagul C (Carry) indică un transport în afara domeniului de reprezentare a rezultatului, iar flagul A (Auxiliary) indică valoarea transportului de la bitul 3 la bitul 4 (între cifrele hexazecimale).
Se sugerează revenirea la lucrarea 2 în vederea revizuirii definiţiei flagurilor aritmetice şi a exemplelor prezentate.
Pentru vizualizarea flagurilor, din meniul View al ferestrei corespunzătoare fişierului de tip .com se va selecta *flags*, iar pentru vizualizarea rezultatului operaţiei de scădere se poate consulta conţinutul ALU din cadrul aceluiaşi meniu. Pentru o mai bună înţelegere a exemplelor prezentate, se va urmări şi fereastra *lexical flag analyzer*.

```
; 4 este egal cu 4
mov ah, 4       ; AH=4
mov al, 4       ; AL=4
cmp ah, al   ;AH-AL=0, deci Z=1, iar S=0, C=0

; (cu semn/ fără semn)
; 4 este mai mare (greater/above) decât 3
mov ah, 4       ;AH=4
mov al, 3       ;AL=3
cmp ah, al      ;AH-AL=1, deci Z=0, iar S=0, C=0

; (cu semn)
; 1 este mai mare (greater) decât -5
mov ah, 1       ; AH=1
mov al, -5      ; AL=-5 = 251 = 0fbh
cmp ah, al      ; AH-AL=1-(-5)=6, deci trebuie interpretat ca o
                ; scădere:
                        0000 0001b-
                        1111 1011b
                        0000 0110b, C=1, A=1, Z=0, S=0

; (fără semn)
; 1 este mai mic (below) decat 251
mov ah, 1       ; AH=1
mov al, 251     ; AL=251
cmp ah, al      ; AH-AL=1-251=6 trebuie interpretat ca o scădere
                ; cu împrumut
                        0000 0001b-
                        1111 1011b
                         0000 0110b, C=1, A=1, Z=0, S=0
```

```
; (cu semn)
; -3 este mai mic (less) decât -2
mov ah, -3      ; AH=-3
mov al, -2      ; AL=-2
cmp ah, al      ; AH-AL=-3-(-2)=-3+2=-1=0FFh, O=0,S=1,Z=0,A=1

; (cu semn)
; -2 este mai mare(greater) decât -3
mov ah, -2      ; AH=-2
mov al, -3      ; AL=-3
cmp ah, al      ; AH-AL=-2+3=+1=01h, O=0,S=0,Z=0,A=0

; (fără semn)
; 255 este mai mare(above) decât 1
mov ah, 255     ; AH=255=0FFh
mov al, 1       ; AL=1
cmp ah, al      ; AH-AL=254=0FEh, S=1, Z=0, C=0
```

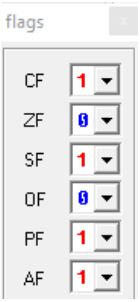


Fig.4.3 Vizualizarea flagurilor aritmetice pentru cel de-al 5-lea exemplu

Exerciţii şi teme (II)

Analizaţi ultimele 3 exemple individual, efectuând operaţiile în binar şi specificaţi valorile flagurilor şi abia apoi rulaţi exemplele cu emulatorul.

4.3. Folosirea întreruperilor: **z02.asm**

Întreruperea este un semnal transmis sistemului de calcul prin care acesta este anunţat de apariţia unui eveniment care necesită atenţie. În BIOS sunt scrise o serie de subrutine legate de echipamentele periferice ale sistemului, apelarea lor într-o aplicaţie făcându-se prin întreruperi, cu instrucţiunea **INT**, cu sintaxa **INT n**. O întrerupere poate avea mai multe servicii asociate, serviciul selectându-se prin încărcarea în registrul AH a unui număr specific acelui serviciu, înainte de apelarea întreruperii. Alţi parametri de apel ai serviciului se încarcă în anumiţi regiştri, după caz, aşa cum se va vedea într-o lucrare ulterioară, în care se vor studia întreruperile mai pe larg.

Aplicaţia de faţă foloseşte întreruperea INT 21h cu serviciul 09h, care afişează un şir de caractere aflat la locaţia adresată de DS:DX, şir care trebuie să se termine cu caracterul $. (a se consulta documentaţia, la secţiunea Tutoriale, lista întreruperilor).

```
.stack 64h      ; segmentul de stivă
.data           ; segmentul de date

msg db "Hello, World", 24h    ;se defineşte un şir de octeţi,
;în segmentul de date, putând fi identificat prin numele
;variabilei msg şi fiind iniţializat cu: codul Ascii al
;caracterului ‚H', urmat de codul Ascii al caracterului ‚e',
;s.a.m.d.; şirul se termină cu caracterul ce are codul Ascii
;24h, adică $.
.code
    mov ax, @data
    mov ds, ax

    mov dx, offset msg
    mov ah, 9
    int 21h
.exit
```

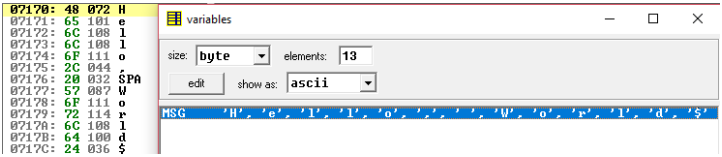


Fig.4.4 Vizualizarea variabilei de tip şir în memorie

Exerciţii şi teme (III)
Analizaţi lista întreruperilor din tutorial şi propuneţi o altă metodă de afişare a unui şir de caractere pe ecran (folosind o altă întrerupere şi/sau serviciu).

4.4. Citirea unor date de la tastatură: **keybrd.asm**

Programul **keybrd.asm** foloseşte instrucţiunile IN, CMP, JNZ, JZ şi ilustrează folosirea funcţiilor tastaturii. Aplicaţia foloseşte bufferul tastaturii (de 16 biţi, vizualizat jos în fereastră, lângă opţiunea change font) atunci când se tipăreşte foarte repede. Codul aplicaţiei se repetă în buclă până la apăsarea tastei „esc", orice alt caracter fiind afişat pe ecran (Figura 4.5).

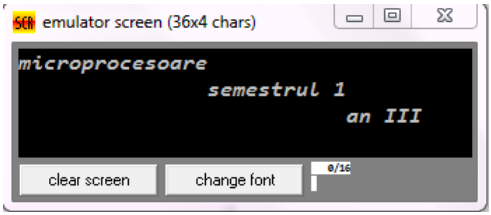


Fig.4.5 Fereastra aplicaţiei keybrd.asm

Observatii:
CMP AL,1Bh – compară conţinutul registrului AL cu codul ASCII al tastei Esc (codul ASCII al tastei Esc este 1Bh).

JZ Rep – JZ vine de la Jump if Zero, adică salt dacă flagul Z este setat. Programul va face un salt la adresa marcată de eticheta Rep. O instrucţiune asemănătoare este JNZ, adică Jump if Not Zero, în cazul în care flagul Z nu este setat. În acest program, instrucţiunea CMP setează flag-urile. Instrucţiunile aritmetice setează de asemenea stările flag-urilor.

Programul **keybrd.asm** este prezentat în continuare:

```
name "keybrd"
org     100h

;_______INTRARE DE LA TASTATURA__________
mov dx, offset msg     ; afişează mesaj de primire
       mov ah, 9
       int 21h
;============================
wait_for_key:                  ; buclă infinită pt preluarea şi
                               ; afişarea tastelor
       mov   ah, 1             ; verifică dacă există tastă
                               ; nepreluată din buffer
       int     16h
   jz   wait_for_key
       mov     ah, 0           ; preia tasta de la tastatură,
                               ; ştergând-o din buffer
       int     16h
       mov     ah, 0eh         ; afisează pe ecran tasta
       int     10h
       cmp     al, 1bh         ; se verifică dacă s-a apăsat
                               ; 'esc' pentru a ieşi
       jz      exit
       jmp  wait_for_key
;============================
exit:    ret
msg  db "Type anything...", 0Dh,0Ah
     db "[Enter] - carriage return.", 0Dh,0Ah
     db "[Ctrl]+[Enter] - line feed.", 0Dh,0Ah
     db "You may hear a beep", 0Dh,0Ah
     db "    when buffer is overflown.", 0Dh,0Ah
     db "Press Esc to exit.", 0Dh,0Ah, "$"
end
```

Exerciţii şi teme (VI)
1. Rulaţi programul keybrd.asm setând viteza maximă la execuţie (întârziere de 0 msec), dar şi la o viteză mai mică (de exemplu întârziere de 200 msec) din cursorul pentru timp. Urmăriţi ambele situaţii.

2. Tastaţi cât mai repede cu putinţă, introducând mai mult de 16 caractere. Ce observaţi?

5. Simulator de microprocesor (III)

5.1. Salturi în program – **thermometer.asm**

Aplicaţia foloseşte un termometru care trebuie menţinut la o temperatură cuprinsă între 60° şi 80° cu ajutorul unui radiator de căldură. Aplicaţia foloseşte 2 porturi de intrare: de date (adresa 125) şi de control (adresa 127). La început, temperatura este 0 (portul 125 iniţializat cu 0), utilizatorul putând interveni şi controla (porni/opri) radiatorul de căldură (portul 127) sau temperatura aerului. Temperatura creşte repede de la valoarea 0, până depăşeşte 80°, apoi radiatorul de căldură se opreşte din program (automat) şi până ce temperatura nu ajunge să fie mai mică de 60° nu mai porneşte.

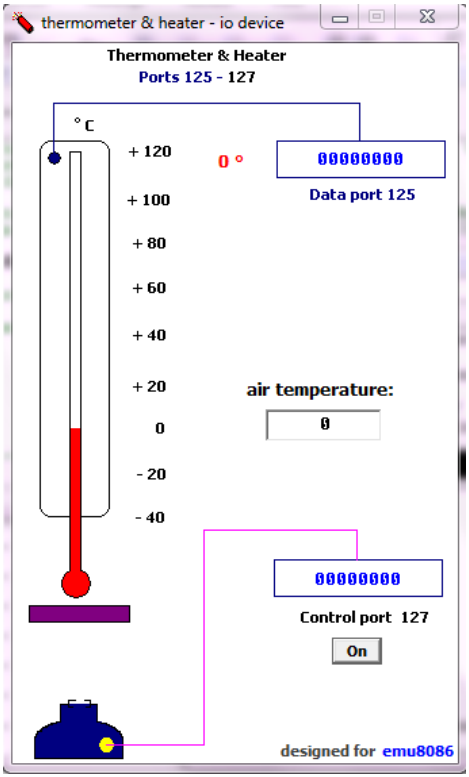


Fig.5.1. Fereastra aplicaţiei thermometer.asm

Programul **thermometer.asm** este prezentat în continuare:

```
; setează adresa segmentului de date înspre segmentul de cod:
mov ax, cs
mov ds, ax
start:                ;eticheta START

in al, 125            ;preia valoarea temperaturii de pe portul
                      ;de date cu adresa 125
cmp al, 60            ;compara valoarea temp. curente cu 60°
jl  low               ;daca temp. este < 60° sare la et. LOW

cmp al, 80      ; compară valoarea temp. curente cu 80°
jle  ok         ; daca temperatura este <= 80°,
                ; sare la eticheta OK
jg   high       ; daca temp. este > 80°, sare la eticheta HIGH
low:            ; eticheta LOW
mov al, 1       ; AL=1
out 127, al     ; pune radiatorul de caldura pe "on",
                ; trimitand 1 pe port
jmp ok          ; salt neconditionat la eticheta OK

high:
mov al, 0       ; AL=0
out 127, al     ; pune radiatorul de caldura pe "off",
                ; trimitand 0 pe port

ok:
jmp start       ; salt înapoi la START,
                ; deci bucla se reia la infinit.
```

Exerciţii şi teme (I)

1. Modificaţi limitele între care se încearcă păstrarea temperaturii la 70° şi 90°.
2. Modificaţi valoarea ce se încarcă în registrul AL la eticheta LOW să fie 2. Salvaţi şi rulaţi din nou. Ce observaţi ? Cum explicaţi ?

5.2. Operaţii de incrementare/decrementare: **examples**, **LED display test**

Aplicaţia este una foarte simplă, foloseşte operaţii de incrementare/decrementare asupra registrului AX şi salturi necondiţionate în program. Acest exemplu foloseşte portul 199 pentru a emula existenţa unui dispozitiv virtual precum cel din figura 5.2.

Fig.5.2. Fereastra aplicaţiei LED display

Programul **LED_display_test.asm** este prezentat în continuare:

```
mov ax, 1234    ;AX=1234
out 199, ax     ;continutul reg AX se trimite la portul 199,
                ;deci se va vizualiza pe display valoarea 1234
mov ax, -5678   ;AX=-5678
out 199, ax     ; pe display se va vizualiza valoarea -5678
; bucla infinita
mov ax, 0       ; AX=0
x1:             ; eticheta x1
  out 199, ax   ; se afiseaza pe display valoarea din AX
  inc ax        ; se incrementeaza continutul registrului AX
jmp x1          ; salt automat la eticheta x1
hlt
```

Exerciţii şi teme (II)

1. Modificaţi aplicaţia astfel încât să funcţioneze asemănător termometrului: valoarea să se incrementeze până ajunge la 50, apoi să descrească până ajunge la 40 şi tot aşa. Sugestie: folosiţi instrucţiuni de salt condiţionat.

5.3. Scrierea datelor în memoria video: **„Hello, world" (examples)**

Fişierul asm se deschide din meniul emulatorului, folosind opţiunea examples, şi se găseşte sub numele „Hello, world", efectul fiind cel ilustrat în figura urmatoare.

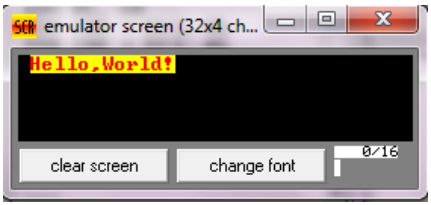


Fig.5.3. Fereastra de vizualizare a ecranului

Emulatorul foloseste 8 pagini de memorie video, cuprinse intre adresele 0B8000h-0C0000h. Ecranul emulatorului poate fi redimensionat, astfel incat sa fie necesara mai putina memorie pentru fiecare pagină.

Folosind acest program se afişează în ecranul emulatorului mesajul „hello world” prin scriere directa în memoria video. O caracteristică a memoriei VGA este că primul octet are semnificaţie de cod Ascii al caracterului de afişat, iar octetul următor (consecutiv) reprezintă atributul lui de culoare.

Atributul de culoare este scris pe octet, cei mai semnificativi 4 biţi specificând culoarea de fond (background), iar cei mai puţin semnificativi 4 biţi specificând culoarea de scriere (foreground), după regula: primul bit – dacă este intermitent sau nu, iar următorii 3 culoarea în format RGB. Dintre cele mai uzuale culori, se pot specifica: 0000-negru, 0001-albastru, 0010-verde, 0100-rosu, 0111-gri deschis, 1000-gri inchis, 1001-albastru deschis, ..., 1110-galben, 1111-alb.

Observatii

MOV [02h],'H' – va copia codul Ascii al literei H la adresa [02h] în memoria video.

MOV [BL],AL – copiază AL la adresa dată de BL. Registrul BL poate fi făcut pointer la prima poziţie în memoria video, apoi prin incrementare cu 2 se trece la următoarea locaţie specifică codului caracterului de afişat.

LOOP este specific buclelor: se execută instrucţiunile din cadrul buclei atât timp cât CX este diferit de 0. Instrucţiunea LOOP lucrează împreună cu registrul CX, pe care il decrementează la fiecare trecere prin buclă, dar şi cu o etichetă care specifică locul de salt.

Programul **hi-world.asm** este prezentat în continuare:

```
name "hi-world"
org 100h
;_______SCRIERE IN MEMORIA VIDEO__________

mov ax, 3    ;setare mod video
int 10h      ;mod text 80coloane x 25linii, 16 culori, 8 pagini
            ;ah=0, al=3)
mov ax, 1003h          ;invalidare intermitentă
                       ;permiterea celor 16 culori
mov bx, 0
int 10h

mov     ax, 0b800h     ;setare registru segment
mov     ds, ax

;pe masură ce se scrie în memoria video, datele se afişează pe
;ecran
;se afişează mesajul Hello, World
;primul octet (la adrese pare) e codul Ascii,
;al doilea octet e atributul de culoare (la adrese impare).
```

```
mov [02h], 'H'
mov [04h], 'e'
mov [06h], 'l'
mov [08h], 'l'
mov [0ah], 'o'
mov [0ch], ','
mov [0eh], 'W'
mov [10h], 'o'
mov [12h], 'r'
mov [14h], 'l'
mov [16h], 'd'
mov [18h], '!'

; colorează toate caracterele:
mov cx, 12            ;numărul de caractere
mov di, 03h           ;adresa culoare litera ,H'

c:  mov [di], 11101100b   ;roşu deschis(1100)/fond galben(1110)
    add di, 2             ;adresa culoare litera urmatoare
    loop c

; aşteaptă apăsarea unei taste
mov ah, 0
int 16h
ret
```

Exerciţii şi teme (III)

1. Executaţi aplicaţia pas cu pas, modificând atributul de culoare. Folosiţi pentru fond culoare roşu deschis, iar pentru scris culoarea verde.

2. Scrieţi un nou program, în care să înseraţi următoarele instrucţiuni:

```
MOV AX, 0B800h     ; AX= B800h.
MOV DS, AX         ; DS=B800h
MOV CL, 'A'        ; CL =codul Ascii al lui A, adică 41h.
MOV CH, 1101_1111b ; CH =11011111b.
MOV BX, 15Eh       ; BX=15Eh.
MOV [BX], CX       ; in memorie, la adr. B800:015E se depune CX
```

Rulaţi pas cu pas, explicaţi codul şi specificaţi efectul programului. Modificaţi astfel încât să se afiseze 3 caractere B de culori diferite: unul roşu, unul verde, unul albastru, toate pe fond gri.

3. Scrieţi un program care să reproducă imaginea din figura 5.3, adică să apară textul respectiv pe linii diferite; în plus, modificaţi atributele de culoare să fie diferite de pe un rând pe altul.

6. Setul de instrucţiuni 8086 (I)

Setul complet de comenzi pe care un microprocesor le înţelege şi le poate executa este cunoscut sub numele de **set de instrucţiuni**.
În următoarele două lucrări sunt prezentate instrucţiunile din setul de bază al familiei de procesoare Intel. Acestea dispun de un set puternic şi diversificat de instrucţiuni specifice procesoarelor de tip CISC (Complex Instruction Set Computer). Instrucţiunile pot fi clasificate în funcţie de mai multe criterii:

a)După **numărul de operanzi:**
- instrucţiuni fără operanzi
- instrucţiuni cu un operand
- instrucţiuni cu doi operanzi

b)După **lungimea instrucţiunii**:
- 1-6 octeţi

c)După **funcţia instrucţiunii**:
- instrucţiuni de transfer: MOV, PUSH, POP, IN, OUT — datele sunt copiate între memorie sau porturi I/O şi regiştrii procesorului, fără a fi prelucrate
- instrucţiuni aritmetice şi logice: ADD, INC, AND, CMP— datele sunt prelucrate în format numeric
- instrucţiuni de manipulare a şirurilor: MOVS, CMPS, REP
- instrucţiuni specifice întreruperilor: INT
- instrucţiuni de ramificare: CALL, JMP
- instrucţiuni de control al microprocesorului: CLC, STC, NOP, HLT

Se vor folosi următoarele **notaţii:**

reg – conţinutul unui registru oarecare de 8 sau 16 biţi, exceptând regiştrii segment
reg8 – conţinutul unui registru de 8 biţi
reg16 – conţinutul unui registru de 16 biţi
sreg – conţinutul unui registru segment
acc – conţinutul acumulatorului
mem – conţinutul unei locaţii de memorie pe unul sau mai mulţi octeţi, adresată cu unul dintre modurile de adresare permise pentru memoria de date, cu excepţia adresării imediate
mem8 – conţinutul unei locaţii de memorie pe un octet
[reg] – conţinutul unei locaţii de memorie a cărei adresă se află într-un registru
[mem] – conţinutul unei locaţii de memorie a cărei adresă se află într-o altă locaţie de memorie

port – adresa, numărul de ordine al unui port de intrare sau de ieşire
imed – operandul este o valoare constantă, ce se adresează în mod imediat
(X) – conţinutul locaţiei X
((X)) – conţinutul locaţiei de memorie adresate de X

6.1 Instrucţiuni de transfer

Aceste instrucţiuni permit copierea/transferul datelor (pe octet sau pe cuvânt) de la o sursă la o destinaţie, sursa fiind întotdeauna operandul al doilea.

Sursa	**Destinaţia**
reg	reg
mem	mem
imed	-
port (intrare)	port (ieşire)

Instrucţiunea MOV (*Data movement* - Transfer de date)

MOV — este instrucţiunea cel mai des folosită.

MOV destinaţie, sursă; (destinaţie) ← sursă

Operanzi: reg,imed
mem,imed
reg,reg
reg,mem
mem,reg

Observaţii:

- Operanzii trebuie să aibă dimensiune egală;
- Este interzis ca ambii operanzi să fie locaţii de memorie;
- Nu sunt folosiţi regiştrii IP şi FLAGS;
- Registrul CS nu poate fi destinaţie;
- Nu afectează nici un flag.

Exemplu:

```
MOV   AL,    BH        ;AL=BH – transfer pe octet
MOV   AX,   [ADR]      ;AX = (DS:ADR), DS implicit
                       ;transfer pe cuvant
MOV   AX,   ES:[BX]    ;AX = (ES:(BX))
```

Cuvântul pe 16 biţi aflat în memorie în segmentul suplimentar, indicat prin registru segment ES la un offset dat de conţinutul lui BX faţă de începutul acestui segment este transferat în registrul AX.

```
MOV   byte ptr [BX+100],  15  ;in memorie, la octetul de
                              la locatia data de segment DS,
                              offset BX+100 se depune valoarea 15
```

Operatorul **ptr** permite modificarea atributului unei valori. Fără utilizarea acestuia, ultima instrucţiune ar putea fi interpretată în două moduri: se depune valoarea 15 la octetul de la adresa DS:BX+100 sau la cuvântul de la adresa DS:BX+100. Forma *byte ptr* precizează că este vorba de un transfer pe octet.

Instrucţiuni incorecte:

```
MOV   AX,BH      ;lungime operanzi diferită
MOV   [BX],[SI]  ;ambii operanzi în memorie
MOV   CS,BX      ;reg CS apare la destinaţie
```

Instrucţiuni specifice stivei

Stiva reprezintă un concept abstract de structură de date, o listă de tip LIFO- "Last in first out" şi se găseşte în segmentul SS. Adresa curentă, numită şi adresa vârfului stivei, se găseşte în registrul intern SP (Stack Pointer).

Instrucţiunile PUSH (Push Value onto Stack) şi **POP (Pop Value off Stack)** sunt folosite pentru salvarea unor date şi respectiv refacerea lor în ordine inversă.

PUSH sursă	*POP destinaţie*
SP ← SP – 2	*SS : ((SP) + 1) → high (destinaţie)*
SS : ((SP) + 1) ← high (sursă)	*SS : ((SP)) → low (destinaţie)*
SS : ((SP)) ← low (sursă)	*SP ← SP + 2*

Operanzi:

imed, reg16, sreg, mem sreg, reg16, mem

În urma utilizării instrucţiunii PUSH registrul SP este decrementat cu 2, după care operandul sursă este salvat în octeţii de la adresele (SP)+1→octetul high, respectiv (SP)→octetul low.

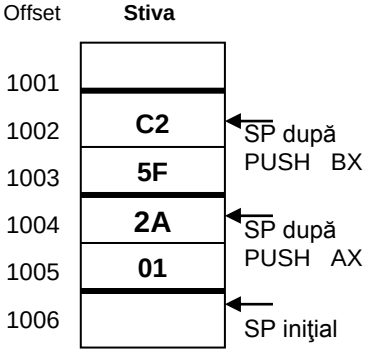
Instrucţiunea POP copiază conţinutul vârfului stivei, adică octeţii de la adresele (SP)+1 şi (SP) în operandul destinaţie, mecanism urmat de incrementarea lui SP cu 2. În urma unei secvenţe de refacere, indicatorul SP trebuie adus la valoarea sa de dinaintea secvenţei de salvare (numărul operaţiilor POP coincide cu cel al instrucţiunilor PUSH).

Observaţii:

- Registrul CS nu poate apărea ca destinaţie;
- Sursa şi destinaţia sunt operanzi pe 16 biţi

Exemplu:

```
SP=1006h; AX=012Ah; BX=5FC2h;

        PUSH   AX;
        PUSH   BX;
        POP    BX;
        POP    AX;
```



Instrucţiuni incorecte:

```
        PUSH    AH; AH are doar 8 biţi, ar altera stiva
        POP     CS;
```

Transferuri bilaterale XCHG (Exchange)

Instrucţiunea **XCHG** constă în interschimbarea conţinutului celor doi operanzi.

XCHG destinaţie, sursă

Operanzi: reg,reg
reg,mem
mem,reg

Observaţie:

- Regiştrii segment nu pot apărea ca şi operanzi
- Operanzii trebuie sa aibă dimensiune egală
- Cel puţin un operand trebuie să fie registru

Exemplu:

Dacă ambii operanzi sunt aflaţi în memorie la locaţiile `mem1` şi `mem2` şi trebuie interschimbaţi, se va folosi următoarea secvenţă:

```
MOV      reg,  mem1
XCHG     reg,  mem2
MOV      mem1,  reg
```

Instrucţiunea XLAT

Nu are operanzi. Se încarcă în AL conţinutul locaţiei din segmentul de date de la offset-ul lui BX adunat cu conţinutul lui AL. Este utilă în conversia unor tipuri de date şi se foloseşte împreună cu tabele de translatare.

AL ← (DS : (BX + AL))

Exemplu:

```
Var db 'ABCDEF'; Var este un sir de octeti
               ;initializati cu codurile Ascii ale
               ; literelor A...F
LEA  BX, Var   ;incarca in BX adresa efectiva a lui Var
MOV  AL, 2     ; AL=2
XLAT           ; AL=43h, adica codul Ascii al lui 'C'
```

Instrucţiuni de transfer cu porturile: IN (Input Byte/ Word)
OUT (Output to Port)

Oferă posibilitatea de a schimba informaţii cu perifericele prin intermediul porturilor de intrare/ieşire.

IN destinaţie, port *OUT port, sursă*

Transferurile sunt făcute din sau în acumulator, pe cuvânt sau pe octet (AX, AL). Adresa portului poate fi specificată explicit pentru primele 256 de porturi (00h—FFh) sau prin intermediul registrului DX.

Exemplu:

```
IN    AL,70H    ;AL = (port 70H)
MOV   DX,3ECH
OUT   DX,AX     ;(port 3ECH) =AX
```

Instrucţiuni de transfer specifice adreselor

Instrucţiunea **LEA (Load Effective Address)**

LEA destinaţie, sursă;

Operanzi: reg, mem

Permite copierea adresei efective a sursei (operand aflat în memorie) în registrul general specificat. Operaţia se face în faza de execuţie.

Exemplu:

```
LEA   BX,VAL1          ;BX = offset  VAL1
LEA   DI,[AX][CX]      ;se adună conţinuturile lui AX şi
                       ;CX şi rezultatul se depune în DI
```

În mod asemănător se poate obţine adresa efectivă folosind operatorul **OFFSET** şi instrucţiunea MOV. Atribuirea se face la asamblare.

```
MOV   BX,OFFSET VAL1
```

Instrucţiunile **LDS (Load Data Segment)** şi **LES (Load Extra Segment)**

LDS registru, sursă *LES registru, sursă*

Exemplu:

```
x db 2468h          ;în segmentul de date se definesc
y dw 1357h          ;variabilele

lds si, x         ; încarcă SI=2468h şi DS=1357h
```

Aceste instrucţiuni transferă o adresă completă într-o pereche de regiştri. Perechea de regiştri DS:registru / ES:registru este încărcată cu adresa completă de 32 de biţi conţinută în sursă, definită ca operand double-word în memorie.

Instrucţiuni specifice flagurilor

Instrucţiunile **LAHF (Load AH with Flags)** şi **SAHF (Store AH into Flags)**

$(AH) \leftarrow (PSW)_L$ $(PSW)_L \leftarrow (AH)$

Nu au operanzi. Se încarcă registrul AH cu octetul low al registrului PSW, respectiv se depune conţinutul registrului AH în octetul low al PSW.

Exemplu:

```
LAHF
SHL     AH,7      ;deplasare logică stânga cu 7 poziţii
AND     AH,80H    ;AH conţine flagul CF
```

Instrucţiunile **PUSHF (Push Flags)** şi **POPF (Pop Flags)**

SP ← SP – 2 *PSW ← SS : ((SP) + 1 : (SP))*
SS : ((SP) + 1: (SP)) ← PSW *SP ← SP + 2*

Nu au operanzi. Efectul lor este salvarea registrului PSW pe stivă, respectiv refacerea acestuia de pe stivă.

Exemplu:

```
PUSHF           ;încarcă stiva cu valorile Flags
POP AX          ;sunt luate de pe stivă şi depuse în AX
OR AX,01        ;se setează Carry,
PUSH AX         ;se depune în stivă cu Carry modificat
POPF            ;se ia din stivă înapoi în Flags, cu CF=1
```

6.2 Instrucţiuni aritmetice şi logice

Instrucţiuni specifice adunării

Sunt instrucţiuni cu doi operanzi iar rezultatul se depune în primul operand. Se modifică flag-urile C, S, Z, P, O, A, de unde şi denumirea de flaguri aritmetice. Semnificaţia tuturor flagurilor a fost prezentată în lucrarea anterioară.

Instrucţiunea **ADD (Integer Addition)**

ADD destinaţie, sursă; (destinaţie) = (destinaţie) + (sursă)

Operanzi: reg,imed
mem,imed
reg,reg
reg,mem
mem,reg

Se adună conţinutul sursei la destinaţie şi rezultatul se depune în operandul destinaţie, vechea valoare pierzându-se. Flag-urile S, Z, P se modifică conform rezultatului operaţiei.

Observaţii:

- Operanzii trebuie să aibă dimensiuni egale;
- Este interzis ca ambii operanzi să fie locaţii de memorie.

Exemplu:

```
ADD AX,CX;
ADD byte ptr [DI],4    ;destinaţia în memorie pe octet,
                       ;sursa imediată
```

Exemplu:

```
MOV AX, 8FFEh      ; AX=8FFEh
MOV BX, 0C001h     ; BX=0C001h
ADD AX,BX          ; AX=4FFFh şi CF=1
```

Instrucţiunea **ADC (Add with Carry)** este asemănătoare cu ADD, singura diferenţă este că se adună şi bitul de transport. Este utilă în cazul execuţiei unor adunări pe lungimi mai mari decât cuvântul. Toate flagurile aritmetice sunt afectate.

(destinaţie) = (destinaţie) + (sursă) + C

Exemplu:

```
MOV AX, 8FFEh      ; AX=8FFEh
MOV BX, 0C001h     ; BX=0C001h
ADC AX,BX          ; AX=5000h şi CF=0
```

Instrucţiunea **INC (Increment)** are ca efect incrementarea cu valoarea 1 a destinaţiei. Se modifică toate flag-urile aritmetice, mai puţin Carry.

INC destinaţie; (destinaţie) = (destinaţie) + 1

Exemplu:

```
mov AX, 0FFFFh ; AX=0FFFFh
inc AX         ; AX=0000h, iar CF=0 şi ZF=1
inc byte [7]   ; se incrementează octetul din
               ; memorie de la adresa DS:7
```

Instrucţiunile **DAA (Decimal Adjust for Addition)** şi **AAA (Ascii Adjust for Addition)**

dacă ($AL_{0,3}$) > 9 sau A = 1
  atunci (AL) ← (AL) + 6
    A ← 1,
  altfel A ← 0

dacă ($AL_{4,7}$) > 9 sau C = 1
  atunci (AL) ← (AL) + 60h
    C ← 1,
  altfel C ← 0

dacă ($AL_{0,3}$) > 9 sau A = 1
  atunci (AL) ← (AL) + 6
    (AH) ← (AH) + 1
    A ← 1
    C ← 1
    (AL) ← (AL) AND 0Fh
  altfel A ← 0, C ← 0

Nu au operanzi. Se execută corecţia zecimală a acumulatorului AL, respectiv AX după o operaţie de adunare cu operanzi BCD împachetaţi (2 cifre pe un octet), respectiv despachetaţi (o cifră pe octet).

Instrucţiuni specifice scăderii

Instrucţiunea **SUB (Subtraction)** poate fi interpretată ca şi o adunare a destinaţiei cu complementul faţă de 2 al sursei. Se inversează rolul bistabilului Carry. Toate flagurile aritmetice sunt afectate.

SUB destinaţie, sursă; (destinaţie) = (destinaţie) – (sursă)

Exemplu:
```
MOV AX,4000h   ; AX= 4000h
MOV BX, 0B000h ; BX=0B000h
SUB AX, BX     ; AX= 9000h şi CF=1
```

În cazul instrucţiunii ***SBB*** **(Subtract with Borrow)** se ţine cont de un împrumut anterior. Se foloseşte la scăderi de operanzi pe mai multe cuvinte.

SBB destinaţie, sursă; (destinaţie) = (destinaţie) – (sursă) - C

Exemplu:
```
MOV AX,4000h   ; AX= 4000h
MOV BX, 0B000h ; BX=0B000h
SUB AX, BX     ; AX= 9000h şi CF=1
SBB AX,1       ; AX=8FFEh şi CF=0
```

Instrucţiunea **DEC** are ca efect decrementarea cu valoarea 1 a destinaţiei. Se modifică toate flag-urile aritmetice mai putin Carry.

DEC destinaţie; (destinaţie) = (destinaţie) – 1

Exemplu:
```
MOV AX, 01h    ; AX=0001h
DEC AX         ; AX=0000h şi CF=0, iar ZF=1
```

```
MOV AX, 00h     ; AX= 0000h
DEC AX          ;AX=0FFFFh şi CF=0 (instrucţ. echivalentă
                ;SUB AX,1 ar fi setat CF), OF=0, iar ZF=0
```

Instrucţiunea **NEG** realizează complementul faţă de 2 al destinaţiei. Toate flagurile aritmetice sunt afectate.

NEG destinaţie; (destinaţie) = 0 – (destinaţie)

Exemplu:
```
MOV AX, 0F0Fh  ;AX=0F0Fh
NEG AX         ;AX=F0F1h şi CF=1 (borrow)
```

Exemplu:
```
;următoarea secvenţă calculeaza modulul unei valori
OR AX,AX       ;se testează semnul
JNS et         ;salt dacă numărul e pozitiv
NEG AX         ;negarea numărului negativ
et:...
```

Semnificaţia instrucţiunii **CMP** este execuţia unei scăderi fictive, fără modificarea operanzilor dar cu poziţionarea tuturor flagurilor aritmetice.

CMP destinaţie, sursă;
dacă destinaţie = sursă → Z=1
destinaţie < sursă → C=1
destinaţie > sursă → C+Z=0 → C=Z=0

Exemplu:
```
MOV AX, 2000h  ;AX=2000h
MOV BX, 400h   ;BX=400h
CMP AX, BX     ;AX=2000h, BX=400h şi OF=0, SF=0, CF=0
```

Instrucţiunile **DAS (Decimal Adjust for Subtraction)** şi **AAS (Ascii Adjust for Subtraction)**

dacă ($AL_{0,3}$) > 9 sau A = 1
  atunci (AL) ← (AL) - 6
         A ← 1,
  altfel A ← 0

dacă ($AL_{4,7}$) > 9 sau C = 1
  atunci (AL) ← (AL) - 60h
         C ← 1 ,
  altfel C ← 0

dacă ($AL_{0,3}$) > 9 sau A = 1
  atunci (AL) ← (AL) - 6
         (AH) ← (AH) - 1
         A ← 1
         C ← 1
         (AL) ← (AL) AND 0Fh
  altfel A ← 0, C ← 0

Nu au operanzi. Se execută corecţia zecimală a acumulatorului AL, respectiv AX după o operaţie de scădere cu operanzi BCD impachetaţi, respectiv despachetaţi.

Instrucţiuni specifice înmulţirii

Operaţiile specifice înmulţirii se fac între acumulator şi un alt operand; rezultatul obţinut este pe 16 sau 32 de biţi.

Instrucţiunea **CBW (Convert Byte to Word)** converteşte octetul la cuvânt, adică bitul de semn din AL se extinde la tot registrul AH. Efectul instrucţiunii este echivalent cu reprezentarea registrului AL în complement faţă de doi pe un număr dublu de biţi. Nu este afectat nici un flag.

dacă $AL_7 = 1 \rightarrow AH = 0FFh$
altfel AH = 0

Instrucţiunea **CWD (Convert Word to Doubleword)** converteşte cuvântul la dublu-cuvânt, adică bitul de semn din AX se extinde la tot registrul DX. Nu este afectat nici un flag.

dacă $AX_{15} = 1 \rightarrow DX = 0FFFFh$
altfel DX= 0

Prin cele două instrucţiuni se face extensia semnului acumulatorului în acumulatorul extins.

Exemplu:

```
MOV AL, 54h     ; AL=54h
CBW             ; AX=0054h
CWD             ; DX:AX=0000 0054h
```

Instrucţiunile **MUL**—înmulţire fără semn şi **IMUL**—înmulţire cu semn

MUL sursă; *IMUL sursă;*
*(acumulator extins) = (acumulator) * (sursă)*

AX = AL * (reg8/mem8)
DX:AX = AX * (reg/mem)

Destinaţia este implicit acumulatorul iar sursa un registru sau o locaţie de memorie. Ambii operanzi sunt fără semn, respectiv cu semn. Înmulţirea NU duce la depăşiri. Sunt afectate flagurile O şi C.

Exemplu:

```
A db 5h
B dw 300h
MOV AL,10h
MUL A           ;( AX)  ← (AL) * A
MOV AX,100h
MUL B           ;( DX:AX) ← (AX ) * B
```

Exemplu:
```
mov AL,-16      ; AL=0F0h
mov BL,2        ; BL=2
mul BL          ;AX=01E0h=480, 0F0h=240 nr fara semn
```

Exemplu:
```
mov AL,-16      ; AL=0F0h
mov BL,2        ; BL=2
imul BL         ;AX=FFE0h=-32, 0F0h=-16 nr cu semn
```

Instrucţiunea **AAM (Ascii Adjust for Multiply)** nu are operanzi şi realizează o corecţie a acumulatorului AX, după o operaţie de înmulţire pe 8 biţi cu operanzi BCD despachetaţi.

(AH) ← (AX) / 10;
(AL) ← (AX) MOD 10;

Exemplu:
```
MOV AL,8
MOV BL,7
MUL BL           ;( AX ) = 38h
AAM              ;AX = 0506h
```

Instrucţiuni specifice împărţirii

Instrucţiunile **DIV şi IDIV**

DIV sursă;
(acumulator) = (acumulator extins) / (sursă)
(extensia acumulatorului) = (acumulator extins) MOD (sursă)

AL = AX / (reg8/mem8)
AH = AX MOD (reg/mem8)

AX = (DX:AX) / (reg/mem)
DX = (DX:AX) MOD (reg/mem)

Destinaţia este implicit acumulatorul sau acumulatorul extins iar sursa un registru general sau o locaţie de memorie. Ambii operanzi sunt fără semn, respectiv cu semn. Împărţirea presupune că lungimea deîmpărţitului este dublă faţă de cea a împărţitorului. Poate duce la depăşiri când împărţitorul este zero sau când câtul depăşeşte dimensiunea rezervată rezultatului. În urma unei depăşiri se iniţiază o întrerupere de nivel 0. Flagurile nu sunt afectate.

Exemplu:
```
A db 5
mov AX,51h      ; AX=51h=5h*10h+01h=81_10=5_10*16_10+1_10
div A           ; AL=10h (câtul), iar AH=01h (restul)
```

Exemplu:

```
B dw 300h
mov AX, 14h     ;AX=0014h =20₁₀
mov DX, 03h     ;DX=0003h = 3₁₀
div B           ;DX:AX=3₁₀*16⁴+20₁₀=196628₁₀ se împarte
                ;la 300h=768₁₀ => AX=0100h=256₁₀(cât)
                ;şi DX=0014h=20₁₀ (rest)
```

Instrucţiunea **AAD (ASCII Adjust for Division)** nu are operanzi şi realizează o corecţie a acumulatorului AX, interpretat ca două cifre BCD despachetate (o cifră pe octet). Efectul este următorul:

AL = AH * 10 + AL
AH = 0

6.3 Exerciţii şi teme

6.3.1. Analizaţi secventele de mai jos şi apoi executaţi-le manual pentru a completa valorile în tabelele alăturate. Apoi introduceţi-le pe calculator şi executaţi-le cu emulatorul *emu8086* sau cu Turbo Debugger-ul pentru a verifica corectitudinea rezultatelor.

Exemplu:

```
MOV     AX, BBBBH       ;AX=BBBBH
ADD     AX, 4445H       ;AX=0000H

 BBBB    1011 1011 1011 1011
+4445    0100 0100 0100 0101
-----------------------------
 0000    0000 0000 0000 0000
```

AX	0000H
Sign flag	0
Carry flag	1
Parity flag	1
Zero flag	1

a.
```
mov al,64h
mov bl,0A6h
add al,bl
```

AL	
Sign flag	
Carry flag	
Overflow flag	
Zero flag	

b.
```
mov al,38h
mov bl,62h
sub al,bl
```

AL	
Sign flag	
Carry flag	
Overflow flag	
Zero flag	

c.
```
mov al,57h
mov bl,39h
sub al,bl
```

AL	
Sign flag	
Carry flag	
Overflow flag	
Zero flag	

d.
```
mov ax,2B54h
mov bx,100h
mov [bx],ax
```

AX	
[DS:100h]	
[DS:101h]	
Overflow flag	
Zero flag	

e.
```
mov ax,2B54h
mov bx,100h
mov 20h[bx],ax
```

AX	
[100h]	
[120h]	
[121h]	
Zero flag	

f.
```
mov al,'a'
mov ah,'A'
```

AL	
AH	
Carry flag	
Overflow flag	
Zero flag	

g.
```
mov bx,2863h
mov sp,0102h
push bx
```

BX	
SP	
[SS:101h]	
[SS:100h]	
Zero flag	

h.
```
mov ax,2B54h
mov bx,100h
mov dx, 1234h
mov [bx+20h], dx
mov ax,[bx+20h]
```

AX	
[100h]	
[120h]	
[121h]	
Zero flag	

6.3.2. Adunări şi scăderi

- Să se evalueze expresia *r=(x+y)-(z+15)-(t-10)*. Variabilele sunt reprezentate pe 16 biţi cu semn.

```
Date:
        x dw 100
        y dw 500
        z dw 1000
        t dw -200
        r dw ?

Cod:
        mov ax, x
        add ax, y               ;ax = x+y
        mov bx, z
        add bx, 15              ;bx = z+15
        sub ax, bx              ;ax =(x+y) - (z+15)
        mov bx, t
        sub bx, 10              ;bx = t-10
        sub ax, bx              ;ax = =(x+y)-(z+15)-(t-10)
        mov r, ax
```

6.3.3. Inmulţiri şi împărţiri

- Să se evalueze *r=(x-y*z)/t*, unde *x, y, z, t* şi *r* vor fi reprezentate pe 16 biţi cu semn.

```
Date:
        x dw 2000
        y dw -500
        z dw 200
        t dw 300
        r dw ?
```

```
Cod:
        mov ax, y
        imul z                  ;dx:ax = y*z
        mov cx, dx
        mov bx, ax              ;cx:bx = y*z , avem nevoie de dx:ax
                                ;pt. conversia lui x
        mov ax, x
        cwd                     ; dx:ax = a
        sub ax, bx
        sbb dx, cx              ;dx:ax = x-y*z
        idiv t                  ;ax = (x-y*z)/t
        mov r, ax
```

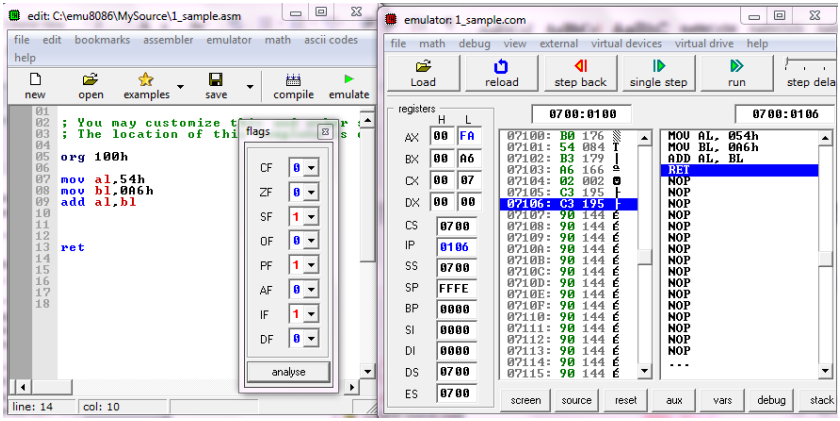


Figura 6.1. Verificarea corectitudinii rezultatelor cu EMU8086

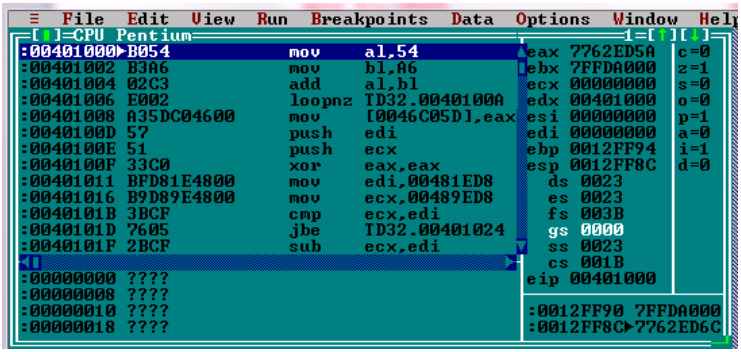


Figura 6.2. Verificarea corectitudinii rezultatelor cu TD32

7. Setul de instrucţiuni 8086 (II)

7.1 Instrucţiuni logice

Aşa cum am văzut instrucţiunile aritmetice privesc operanzii ca şi valori numerice. Instrucţiunile logice îi consideră simple şiruri de biţi. O funcţie logică se va aplica tuturor biţilor sau perechilor de biţi corespunzători. Nu există transport.

Instrucţiunea **NOT (Negare logică bit cu bit)** are ca efect negarea tuturor biţilor operandului destinaţie sau altfel spus calculează complementul faţă de 1 al acestuia. Nu modifică nici un flag.

NOT destinaţie;

Exemplu:

```
MOV AX, 1234h  ; AX=1234h =0001 0010 0011 0100b
NOT AX         ; AX=0EDCBh=1110 1101 1100 1011b
```

Restul operaţiilor au câte doi operanzi.
Instrucţiunea **AND (Şi logic bit cu bit)**

AND destinaţie, sursă; (destinaţie) = (destinaţie) AND (sursă)

Destinaţia poate fi un registru general sau o locaţie de memorie iar sursa un registru general, o locaţie de memorie sau o valoare imediată. Este des utilizată când se doreşte mascarea anumitor biţi.

Instrucţiunile **OR (Sau logic bit cu bit)** şi **XOR (Sau-exclusiv bit cu bit)** au aceiaşi operanzi ca şi instrucţiunea AND.

OR destinaţie, sursă; (destinaţie) = (destinaţie) OR (sursă)

XOR destinaţie, sursă; (destinaţie) = (destinaţie) XOR (sursă)

Bit1	Bit2	OR	XOR	AND
0	0	0	0	0
0	1	1	1	0
1	0	1	1	0
1	1	1	0	1

Exemplu: Presupunând AX=1234h şi BX=0F0Fh, operaţiile logice AND, OR, şi XOR (realizate independent una de alta) vor furniza

```
MOV AX, 1234h  ; AX= 1234h = 0001 0010 0011 0100b
MOV BX, 0F0Fh  ; BX= 0F0Fh = 0000 1111 0000 1111b
AND AX, BX     ; AX= 0204h = 0000 0010 0000 0100b
OR AX, BX      ; AX= 1F3Fh = 0001 1111 0011 1111b
XOR AX, BX     ; AX= 1D3Bh =0001 1101 0011 1011b
```

Instrucţiunea **TEST** realizează un AND fictiv între destinaţie şi sursă iar flagurile se modifică la fel ca şi la AND.

TEST destinaţie, sursă; (destinaţie) AND (sursă) → FLAGS

Exemplu:

```
mov AX, 1234h  ; AX=1234h=0001 0010 0011 0100b
mov BX, 0F0Fh  ; BX=0EDCBh=1110 1101 1100 1011b
test AX, BX    ; AX AND BX=0000h=0000 0000 0000 0000b ,
               ; AX=1234h, BX=0EDCBh, SF=0, ZF=1
```

7.2 Instrucţiuni pentru deplasări şi rotaţii

Instrucţiunile au structura următoare:

Mnemonica operand, contor

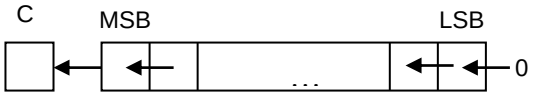
unde *operand* este un registru sau o locaţie de memorie (8 sau 16 biţi) iar *contor* poate fi constanta 1 sau registrul CL:

reg, imed
mem, imed
reg, CL
mem, CL

În cazul operaţiilor de deplasare sunt afectate toate flagurile cu excepţia lui A, iar în cazul unei rotaţii doar C şi O.

Instrucţiunea **SHL/SAL** (**Shift logic/arithmetic left)**

SHL/SAL operand, contor;



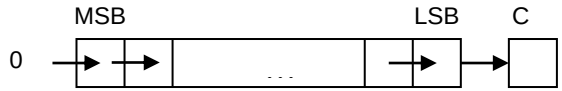
Cel mai semnificativ bit al operandului trece în C iar ceilalti biţi se deplasează la stânga cu o poziţie (înmulţire cu doi). Această operaţie se repetă de un număr de ori egal cu valoarea contorilui.

Exemplu:

```
MOV AX,5555h     ; AX=  5555h  = 0101 0101 0101 0101b
SHL AX,1         ; AX=0AAAAh=1010 1010 1010 1010b, CF=0
```

Instrucţiunea ***SHR*** **(Shift Logic Right)**

SHR operand, contor;



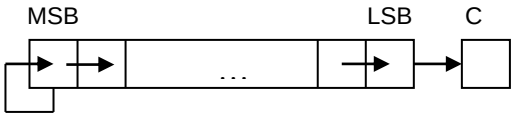
Cel mai puţin semnificativ bit al operandului trece în C iar ceilalţi biţi se deplasează la dreapta cu o poziţie (impărţire cu doi). Aceasta operaţie se repetă de un număr de ori egal cu valoarea din contor.

Exemplu:

```
MOV AX,5555h      ; AX=  5555h = 0101 0101 0101 0101b
SHR AX,1          ; AX=  2AAAh=0010 1010 1010 1010b, CF=1
```

Instrucţiunea **SAR (Shift Arithmetic Right)**—diferenţa faţă de SHR este faptul că semnul se păstrează.

SAR operand, contor;

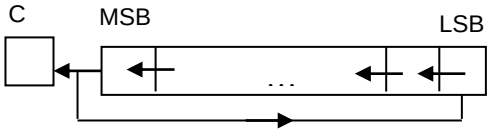


Exemplu:

```
MOV AX,5555h      ; AX=  5555h=0101 0101 0101 0101b
SAR AX,1          ; AX=  2AAAh=0010 1010 1010 1010b, CF=1
```

Instrucţiunea **ROL (Rotate Left)**

ROL operand, contor;



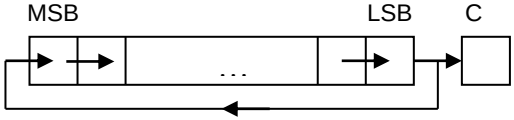
Exemplu:

```
MOV AX,5555h      ; AX=  5555h  =0101 0101 0101 0101b
ROL AX,1          ; AX=0AAAAh=1010 1010 1010 1010b, CF=0
```

Bitul MSB trece atât în C cât şi în bitul LSB din operand.

Instrucţiunea **ROR (Rotate Right)**

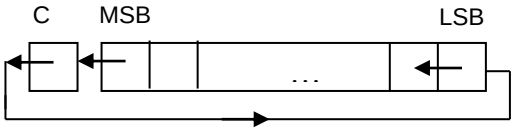
ROR operand, contor;



Exemplu:
```
MOV AX,5555h     ; AX=  5555h  =0101 0101 0101 0101b
ROR AX,1         ; AX=0AAAAh=1010 1010 1010 1010b, CF=1
```

Instrucţiunea **RCL (Rotate Left through Carry)** —C partipă efectiv la rotaţie.

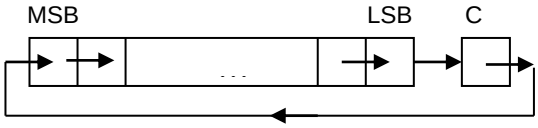
RCL operand, contor;



Exemplu:
```
CLC              ; şterge flagul CF, adică CF=0
MOV AX,5555h     ; AX=  5555h  =0101 0101 0101 0101b
RCL AX,1         ; AX=0AAAAh=1010 1010 1010 1010b, CF=0
```
Exemplu:
```
STC              ; setează flagul CF, adică CF=1
MOV AX,5555h     ; AX=  5555h=0101 0101 0101 0101b
RCL AX,1         ; AX=0AAABh=1010 1010 1010 1011b, CF=0
```

Instrucţiunea **RCR (Rotate Right through Carry)**

RCR operand, contor;



Exemplu:
```
CLC              ; şterge flagul CF, adică CF=0
MOV AX,5555h     ; AX=  5555h  =0101 0101 0101 0101b
RCR AX,1         ; AX=  2AAAh=0010 1010 1010 1010b, CF=1
```

Exemplu:
```
STC              ; setează flagul CF, adică CF=1
MOV AX,5555h     ; AX=  5555h   =0101 0101 0101 0101b
RCR AX,1         ; AX= 0AAAAh=1010 1010 1010 1010b, CF=1
```

7.3. Exerciţii şi teme

1. Analizaţi exemplele de mai jos apoi introduceţi-le pe calculator pentru a verifica corectitudinea rezultatelor.

a.
```
mov al,73h
mov bl,36h
xor al,bl
```

AL	
Sign flag	
Carry flag	
Overflow flag	
Zero flag	

b.
```
mov al,54h
not al
```

AL	
Sign flag	
Carry flag	
Overflow flag	
Zero flag	

c.
```
mov ax,1f54h
mov bx,5a36h
add al,bl
```

AL	
Sign flag	
Carry flag	
Overflow flag	
Zero flag	

d.
```
mov ax,2B54h
mov bx,0236h
mov cl,3
shr ax,1
shl bx,cl
```

AX	
BX	
Carry flag	
Overflow flag	
Zero flag	

e.
```
mov ax,2B54h
mov bx,0236h
mov cl,3
sar bx,cl
```

AX	
BX	
Carry flag	
Overflow flag	
Zero flag	

f.
```
mov ax, 2B54h
mov bx,0236h
mov cl,3
ror ax,cl
rcl bx,cl
```

AX	
BX	
Carry flag	
Zero flag	

g.
```
mov al,54h
mov bl,66h
cmp al,bl
```

AL	
BL	
Carry flag	
Overflow flag	
Zero flag	

h.
```
mov al,32
mov ah,53
mov bx,236
xor ax,bx
```

AX	
BX	
Carry flag	
Zero flag	

2. Pornind de la conţinutul regiştrilor AX=5555h şi CL=4, executaţi următoarele instrucţiuni, specificând valorile regiştrilor şi flagurilor care se modifică:

```
a) SHL AX,CL;          b) shr AX,CL;          c) sar AX,CL;

d) rol AX,CL;          e) ror AX,CL

f) CTC                 g) STC                 h) STC
   rcl AX,CL              rcl AX,CL              rcr AX,CL
```

3. Studiaţi următoarele exemple în pereche şi specificaţi diferenţa dintre ele:

```
a) mov AX,0                        Xor ax,ax

b) MOV  AX, -9                     MOV AX,-9
   MOV  BL, 2                      MOV BL, 2
   IDIV BL                         SAR AX,1
```

4. Comentaţi fiecare secvenţă şi specificaţi valorile regiştrilor şi flagurilor care se modifică:

```
a) Sir DB 2,5,6,8,4,3,75,12
MOV BX, OFFSET SIR
MOV AL,03             ; AL=3
XLAT                  ; extrage al 4-lea element din şir şi
                      ; îl salvează în  AL
SUB AH,AH             ;AH=0
MOV SI,AX             ;SI=AX
MOV AL,[BX+SI]        ;AL=?

b) var db '123456789ABCDEF'     c)sir db 1,2,3,4,5,6,7,8,9,10,11
xor AH, AH                      mov SI,2
lea BX,var                      mov BX,3
mov AL,12                       lea AX, sir[SI][BX]
xlat
```

5. Presupunând că SS=0700h şi SP=FFFEh, se dă urmatoarea secvenţă de instrucţiuni:

```
mov AX,1234h
mov BX, 5678h
push AX
push BX
```

Comentaţi fiecare instrucţiune şi specificaţi cum va arăta zona de memorie unde este alocată stiva (adresele şi conţinutul).

Răspunsuri:

2.

```
a)      shl AX,CL       ; AX=  5550h=0101 0101 0101 0000b, CF=1
b)      shr AX,CL       ; AX=  0555h=0000 0101 0101 0101b, CF=0
c)      sar AX,CL       ; AX=  0555h=0000 0101 0101 0101b, CF=0
d)      rol AX,CL       ; AX=  5555h=0101 0101 0101 0101b, CF=1
e)      ror AX,CL       ; AX=  5555h=0101 0101 0101 0101b, CF=0
f) CF=0 => rcl AX,CL    ; AX=  5552h=0101 0101 0101 0010b, CF=1
g) CF=1 => rcl AX,CL    ; AX=  555Ah=0101 0101 0101 1010b, CF=1
h) CF=1 => rcr AX,CL    ; AX=0B555h=1011 0101 0101 0101b,  CF=0
```

3. a) Cele 2 instrucţiuni sunt identice: ambele depun în registrul AX valoarea 0, dar XOR este mai rapidă.

b) IDIV furnizeaza FCh=-4, SAR furnizează FBh= -5 =>
SAR rotunjeşte numerele negative în jos iar IDIV le rotunjeşte în sus.

5.

```
mov AX,1234h    ; AX=1234h
mov BX, 5678h   ; BX=5678h
push AX         ; SP=0FFFCh, (SS:SP)=1234h
push BX         ; SP=0FFFAh ,(SS:SP)=5678h, deci stiva va arăta
                ;(începând de la adresa fizica 16FFDh, pe octet)
                ; astfel: 12,34,56,78
pop CX          ; CX=5678h, SP=0FFFCh
pop DX          ; DX=1234h, SP=FFFEh
```

8. Setul de instrucţiuni 8086 (III)

8.1 Instrucţiuni pentru şiruri

Operaţii primitive

Reprezintă un set de operaţii pe octet sau pe cuvânt asupra unor locaţii escries e în memorie. Deci operanzii sunt locaţii de memorie pe 8 sau 16 biţi. Distincţia între instrucţiunile pe octet sau pe cuvânt se face prin adaugarea sufixelor B (byte) sau W (word).
Instrucţiunile pentru şiruri nu au operanzi, regiştrii DS:SI sunt folosiţi drept adresă sursă iar ES:DI adresă destinaţie. Parcurgerea şirului se poate face atât înainte cât şi înapoi, deci regiştrii SI, DI sunt actualizaţi fie prin incrementare fie prin decrementare. Sensul de parcurgere al şirului (în ordine crescătoare sau descrescătoare) este determinat de starea flagului D din PSW. Dacă D=0 adresele sunt incrementate cu 1, dacă operaţia este la nivel de octet şi cu 2, dacă e la nivel de cuvânt. Dacă D=1 adresele sunt decrementate în mod similar. Flagul D poate fi setat, respectiv şters prin utilizarea instrucţiunilor fără operanzi STD (Set Direction) respectiv CLD (Clear Direction).

Instrucţiunile de copiere sau transfer (Move String) **MOVSB, MOVSW –** transferă în ((ES :DI)) conţinutul locaţiei de memorie ((DS :SI)) urmată de actualizarea adreselor.

((ES:DI))← ((DS:SI))
şi (SI) = (SI) ± (d), unde d = 1/ 2 (B / W)
(DI) = (DI) ± (d)

Instrucţiunile de comparare şir (Compare String) **CMPSB, CMPSW**— testează egalitatea şirurilor. Se execută o scădere fictivă între octeţii (cuvintele) de la adresele (DS:SI) şi (ES:DI), fără modificarea operanzilor dar cu poziţionarea tuturor flagurilor.

((DS:SI)) – ((ES:DI)) → FLAGS
şi (SI) = (SI) ± (d)
(DI) = (DI) ± (d)

Instrucţiunile de încărcare a elementelor din şir (**Lo**a**d S**tring) **LODSB, LODSW** — se încarcă în AL, respectiv AX octetul, respectiv cuvântul de la adresa (DS:SI), apoi se actualizează adresa.

(AL/AX) ←((DS:SI))
şi (SI) = (SI) ± (d)

Instrucţiunile de memorare şir (**Sto**re **S**tring) **STOSB, STOSW** —se încarcă AL, respectiv AX în octetul, respectiv cuvântul de la adresa (ES:DI), apoi se actualizează adresa.

(AL/AX) →((ES:DI))
şi (DI) = (DI) ± (d)

Instrucţiunile de scanare (**Sca**n **S**tring) **SCASB, SCASW** — testează/caută un anumit octet, cuvânt într-un şir. Se execută diferenţa fictivă dintre AL, respectiv AX şi octetul, respectiv cuvântul de la adresa (ES:DI), fără modificarea operanzilor dar cu poziţionarea tuturor flagurilor.

(AL/AX) – ((ES:DI)) → FLAGS
şi (DI) = (DI) ± (d)

OBS: Începând cu 386 instrucţiunile anterioare se pot realiza şi pe dublu-cuvânt (sufixul este D iar d=4)

Instrucţiunile pentru şiruri realizează inclusiv actualizarea adreselor, însă nu repetă operaţia de un număr de ori egal cu numărul de elemente ale şirului de prelucrat. De aceea, instrucţiunile pentru lucrul cu şiruri sunt combinate cu algoritmi de repetare (gen prefixe de repetare sau bucle cu salt condiţionat).

Prefixe de repetare

Permit execuţia repetată a unor operaţii primitive cu şiruri în funcţie de un contor sau un contor şi o condiţie logică. Formează instrucţiuni compuse alături de operaţiile primitive anterior descrise. Nu sunt instrucţiuni în sine.

REP/ REPE/ REPZ—operaţia primitivă se execută de un număr maxim de ori dat de conţinutul registrului contor CX.

REP/ REPE/ REPZ operaţie_primitivă;

REP este utilizat cu instrucţiuni de tip MOVS, STOS, LODS.
REPE şi REPZ este utilizat cu instrucţiuni de tip CMPS, SCAS.

REPNE/ REPNZ—se iese din buclă dacă rezultatul primitivei este zero. Deci bucla este executată atât timp cât rezultatul este nenul dar nu mai mult de valoarea conţinută de registrul CX.

REP/ REPE/ REPZ operaţie_primitivă;

8.2 Instrucţiuni de salt

Un program se execută prin extragerea din memorie a câte unui octet (fetch). Următoarea instrucţiune de executat se află în segmentul de cod CS, cu un offset dat de către registrul IP (Instruction Pointer). Derularea secvenţială de la o instrucţiune la alta se obţine prin incrementarea registrului IP numit şi Program Counter.
Instrucţiunile de salt ne permit să modificăm derularea secvenţială a unui program. Acestea pot fi clasificate după mai multe criterii.

Salturi – *scurte (SHORT)* sau relative
- *intrasegment (NEAR)* →saltul se face în interiorul segmentului de cod, se modifică IP;
- *intersegment (FAR)* →saltul se poate face oriunde în memorie; se modifică IP şi CS;

Salturi – *condiţionate* →funcţie de valoarea unui anumit bit din PSW;
– *necondiţionate* →derularea e alterată întotdeauna.

8.2.1 Instrucţiunea de salt necondiţionat

JMP—determină întotdeauna un salt în spaţiul de 1 Moctet. Există trei moduri de adresare:
1. Modul relativ: saltul se face la o etichetă a cărei adresă este în domeniul [-128,127] faţă de adresa instrucţiunii JMP;

2. Modul direct: este specificată efectiv adresa unde se sare; mod folosit pentru salturile intra şi inter segment;
NEAR — adresa ţintă este pe 2 octeţi IP_L şi IP_H.
FAR — adresa ţintă este pe 4 octeţi IP_L, IP_H, CS_L şi CS_H.

3. Modul indirect: adresa de salt este rezultatul unor calcule şi se poate modifica dinamic; mod folosit pentru salturile intra şi inter segment
JMP target;

unde target poate fi un registru care conţine offsetul IP-ului, o variabilă care ne dă offsetul sau o adresă de memorie.
Mecanismul de apelare al subrutinelor precum şi instrucţiunile corespunzătoare **CALL** şi **RET** vor fi tratate în lucrarea 9.

8.2.2 Instrucţiuni de salt condiţionat

Condiţiile de salt sunt determinate de starea anumitor flaguri din PSW, afectate de operaţiile aritmetice, logice sau de control.
Modul de adresare este întotdeauna de tip SHORT.

Pentru testarea aceleiaşi condiţii se pot utiliza mnemonici diferite.
Bistabilii de condiţie nu se modifică.

Mnemonica	Condiţie de salt	Interpretare
JE JZ	Z = 1	Equal , Zero
JL JNGE	S ≠ O	Less, Not Greater or Equal
JLE JNG	S ≠ O sau Z = 1	Less or Equal, Not Greater
JB JNAE JC	C = 1	Below, Not Above or Equal, Carry
JBE JNA	C = 1 sau Z = 1	Below or Equal, Not Above
JP JPE	P = 1	Parity, Parity Even
JO	O = 1	Overflow
JS	S = 1	Sign
JNE JNZ	Z = 0	Not Zero, Not Equal
JNL JGE	S = O	Not Less, Greater or Equal
JNLE JG	S = O şi Z = 0	Not Less or Equal, Greater
JNB JAE JNC	C = 0	Not Below, Above or Equal, Not Carry
JNBE JA	C = 0 şi Z = 0	Not Below or Equal, Above
JNP JPO	P = 0	Not Parity, Parity Odd
JNO	O = 0	Not Overflow
JNS	S = 0	Not Sign
JCXZ	Reg CX = 0	CX register is zero

Instrucţiunea JCXZ nu testează indicatori de condiţie ci conţinutul registrului CX.

8.2.3 Instrucţiuni de buclare

În programe apare necesitatea execuţiei unei secvenţe de instrucţiuni, în mod repetat. Secvenţa care se repetă se numeşte buclă (loop) sau iteraţie. Instrucţiunile de control al buclelor sunt prezentate în următorul tabel:
Toate aceste instrucţiuni utilizează registrul CX ca număr de iteraţii, adică se decrementează CX şi dacă acesta e diferit de zero se sare la eticheta specificată. Instrucţiunile de LOOP condiţionat înainte de a testa dacă

registrul contor CX a ajuns la zero, verifică şi indicatorul Z. Eticheta unde se face saltul se află în domeniul [-128, 127] faţă de instrucţiunea curentă.

LOOPxx etichetă;

Mnemonică	Interpretare
LOOP	CX = CX – 1 If (CX ≠ 0) then jump Else continue
LOOPE LOOPZ	CX = CX – 1 If (CX ≠ 0 and Z = 1) then jump Else continue
LOOPNE LOOPNZ	CX = CX – 1 If (CX ≠ 0 and Z = 0) then jump Else continue

8.3 Instrucţiuni pentru controlul procesorului

Aceste instrucţiuni nu au operanzi. O categorie de instrucţiuni se referă la controlul explicit al unor bistabili de condiţie din PSW:

Mnemonică	Acţiune
CLC (Clear carry flag)	C = 0
CLD (Clear direction flag)	D = 0
CLI (Clear interrupt flag)	I = 0
CMC (Complement carry flag)	C = not (C)
STC (Set carry flag)	C = 1
STD (Set direction flag)	D = 1
STI (Set interrupt flag)	I = 1

Dacă dorim ca o anumită secvenţă de program să nu fie întreruptă, aceasta se protejează printr-o instrucţiune CLI înainte şi una STI după.

Cea de-a doua categorie de instrucţiuni realizează operaţii speciale asupra procesorului:

HALT — Oprire procesor; se poate ieşi doar prin întreruperi externe sau reset general.

LOCK — Blocare magistrală; se foloseşte înaintea oricărei instrucţiuni iar pe durata acesteia accesul unui alt dispozitiv la magistrala sistemului hardware este blocat.
WAIT — Aşteaptă; folosită la sincronizarea procesorului cu un alt dispozitiv, de obicei coprocesor matematic.

Instrucţiunile specifice întreruperilor **INT** şi **IRET** vor fi tratate în lucrarea 9.

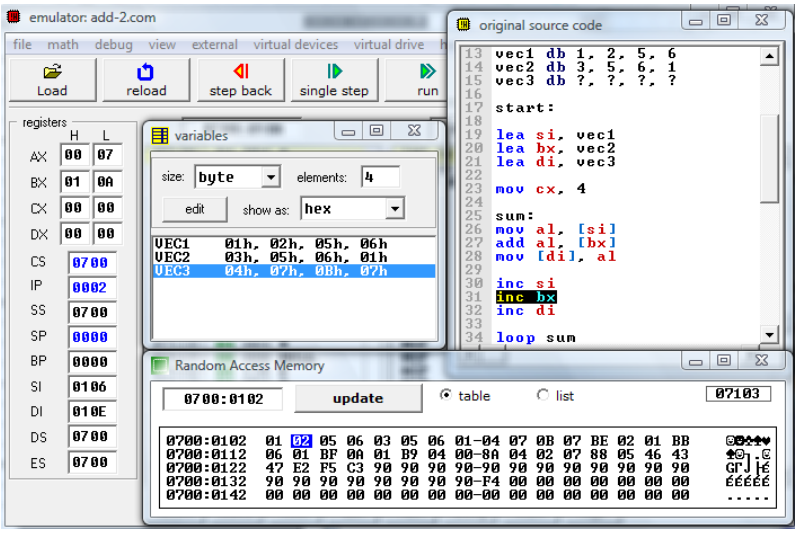
8.4 Exerciţii şi teme

1. Instrucţiuni de salt condiţionate: Calculaţi expresia *r=x-y+z*, numerele fiind reprezentate pe 16 biţi fără semn, testând eventualele depăşiri. In registrul BX puneţi 0 dacă nu sunt depăşiri şi 1 dacă sunt.

```
        mov ax, x
        sub ax, y
        jc et
        add ax, z
        jc et
        mov bx, 0
        jmp gata
et:     mov bx, 1
gata:   ....
```

2. Urmatorul exemplu calculeaza suma a doi vectori iar rezultatul il depune in alt vector

```
        org 100h
        jmp start
        vec1 db 1, 2, 5, 6
        vec2 db 3, 5, 6, 1
        vec3 db ?, ?, ?, ?
        start:
        lea si, vec1
        lea bx, vec2
        lea di, vec3
        mov cx, 4
        sum:
            mov al, [si]
            add al, [bx]
            mov [di], al
            inc si
            inc bx
            inc di
            loop sum
        ret
```



3. Sa se copieze un şir sursa (SIRs) format din 3 elemente definite pe octet într-un alt şir destinatie (SIRd).

```
        ;regiştrii index SI respectiv DI trebuie să pointeze
        ;spre primul element din fiecare şir

        ;pentru fiecare element se executa instrucţiunea MOVSB


      SIRs DB 1,2,3     ; şirul destinaţie cu 3 elemente pe octet
      SIRd DB 3 DUP(0) ;şirul sursă cu 3 elemente, neiniţilizat
        ...
      lea SI, SIRs      ; SI=adr de început a SIRs
      lea DI, SIRd      ; DI=adr de început a SIRd
      mov CX,3          ; CX=3 numărul de elemente
      cld               ; DF=0, şiruri parcurse în sens crescător
et:   movsb             ;eticheta eti se foloseşte pentru a
                        ;asigura revenirea în acest punct
      dec cx            ; după execuţia mosvb (mutare element şi
                        ; actualizare adrese), CX=CX-1
        jnz et          ; dacă încă CX nu a ajuns în zero(jump if
                        ;not ZF->dacă ZF=0,face salt), reia bucla
```

Obs1: Secvenţa de buclare: se poate înlocui cu:

et: movsb dec cx jnz et	et: movsb loop et	rep movsb

Obs2: Programul anterior poate fi rescris folosind instr. LODSB şi STOSB (elementele şirului sursă trebuie preluate în acumulator şi apoi depuse în şirul destinaţie).

```
et: lodsb   ;AL = element curent din SIRs
    stosb   ;din AL se depune elementul curent în SIRd
    dec cx  ;CX=CX-1
    jnz et  ;dacă CX diferit de zero, se reia bucla.
```

4. Sa se testeze egalitatea a 2 şiruri de dimensiuni egale.

```
; se compară sirurile element cu element, iar dacă înainte de a
;ajunge la sfârşit se întâlneşte o nepotrivire, concluzionăm că
;şirurile nu sunt egale(DI contine poziţia primei nepotriviri.

        SIRs DB 0,1,2,3,2,4,2,5,2,6 ;
        SIRd DB 0,1,2,5,2,4,2,5,2,6 ;
        ...
        lea SI, SIRs            ; SI=adr de început a SIRs
        lea DI, SIRd            ; DI=adr de început a SIRd
        mov CX, 10              ; CX= numarul de elemente
        cld                     ; DF=0
        xor bx,bx               ; BX=0
        et1: inc BX
        cmpsb                   ;  verifica  egalitatea  elementelor
        loope et1
```

5. Rezolvaţi problemele de mai jos şi verificaţi-le cu debugger-ul.

a. Scrieţi o secvenţă în limbaj de asamblare care conţine o funcţie care adună conţinutul regiştrilor AX şi BX şi pune rezultatul în registrul CX.

b. Scrieţi o secvenţă care încarcă valorile 00h, 01h, ... FEh, FFh în memorie începând de la adresa DS:0018h. O idee de bază este dată mai jos:

```
        mov al,00h
        mov bx,18h
et:     .....
        jle et
```

c. Scrieţi o secvenţă care încarcă valorile FFh, FEh, FDh, ... 01h, 00h în memorie începând de la adresa DS:0400h.

d. Scrieţi o secvenţă care mută un bloc din memorie de la DS:0220h până la 0300h la adresele începând de la 0400h.

e. Scrieţi o secvenţă care determină cel mai mare octet din memorie între locaţiile DS:0000h şi DS:0050.

9. Macroinstructiuni

9.1 Tipuri de macroinstrucţiuni

Macroinstrucţiunile reprezintă secvenţe de program (instrucţiuni, definiţii de date, directive) asociate unui nume.
Folosirea numelui macroinstrucţiunii în program are ca efect înlocuirea acestuia cu secvenţa de program asociată (expandare). Procesul are loc înaintea asamblării propriu-zise.
Un asamblor care implementează macrouri este Macroasamblor. Spre deosebire de proceduri folosirea macrourilor nu duce la micşorarea programului, ci textul sursă devine mai clar şi mai scurt. Folosirea macrourilor implică două etape :

- definirea macroinstrucţiunilor
- apelul (invocarea) macroinstrucţiunilor.

Macroinstrucţiunile pot fi:
- fără parametri
- cu parametri
- repetitive

9.1.1 Definirea macroinstrucţiunii fără parametri

```
Nume_macro MACRO
    ; corp macro
ENDM
```

Mod de utilizare: se scrie în textul sursă numele macroinstrucţiunii.

Exemple:

```
Save    MACRO                       Rest        MACRO
            PUSH AX                             POP     DI
            PUSH BX                             POP     SI
            PUSH CX                             POP     DX
            PUSH DX                             POP     CX
            PUSH SI                             POP     BX
            PUSH DI                             POP     AX
ENDM                                ENDM
```

Utilizare:

```
Some_proc       PROC            NEAR
                Save
                .....
                rest
                RET
Some_proc       ENDP
```

```
Go_dos  MACRO
        MOV     ax,4C00h
        INT 21h
ENDM
```

9.1.2 Definirea macroinstrucţiunilor cu parametri

```
Nume_macro      MACRO   P1,P2,…Pn
        ;
        corp macro
        ;
ENDM
```

Utilizare: `nume_macro  A1,A2,…An ,`

unde identificatorii Pi sunt parametri formali, iar Ai cei actuali.
La invocare pe lângă expandare are loc şi înlocuirea parametrilor formali cu cei actuali.

Exemple: apel de servicii DOS

```
Int_Dos         MACRO   N
                MOV     AH,N
                INT     21h
ENDM
```

Utilizare: `Int_Dos 9`

```
Aduna   MACRO   OP1,OP2,SUMA
        MOV     AX,OP1
        ADD     AX,OP2
        MOV     SUMA,AX
ENDM
```

Utilizare*:* aduna bx,cx,dx

9.1.3 Macroinstrucţiuni repetitive

Sunt predefinite şi generează secvenţe repetitive de program.

```
REPT    N
        ;corp macro
ENDM
```

Exemplu: Secvenţa următoare generează codurile ASCII pentru cifrele 0-9

```
        N=0
        Cifre Label Byte
REPT    10
        DB      '0' +n
        N=n+1
ENDM
```

Repetarea de un număr nedefinit de ori:

```
IRP     p_formal, <lista_param_actuali>
        ;corp macro
ENDM
```

Se repetă de un număr de ori egal cu numărul de elemente conţinut de lista de parametri actuali.

Exemplu:

```
IRP x,<'1','2','3'>    ;se va expanda în DB '1', DB '2', DB '3'
DB x
ENDM
```

Macroinstrucţiunile pot fi păstrate în fişiere separate (fişier de includere) şi folosite în diferite fişiere sursă prin includerea lor folosind directiva INCLUDE cu sintaxa:

INCLUDE identif_fisier

identif_fisier este un fişier de includere care conţine instrucţiuni corecte acceptate de asamblor.
De obicei fişierele de includere conţin macrouri, echivalări sau definiţiii standard de segmente.

```
INCLUDE fct.inc          ; specificator de fişier
INCLUDE c:\libs\seg.inc  ; specificator complet (calea)
INCLUDE ..\libs\tim.inc
```

Exemplu: c:\tasm\libs\ timer.inc conţine proceduri specifice timer

```
                ……
                include         ..\libs\timer.inc
seg_program     ends
                end     start
```

9.2 Exerciţii şi teme

1. Scrieţi un program complet care determină maximul dintr-un şir.

a. Definiţi un şir de 10 octeţi iniţializaţi cu numere aleatoare;
b. Definiţi o variabilă “maxim” pe octet;
c. Scrieţi o secvenţă de program care determină maximul dintr-un şir;
d. Scrieţi sursa (nume.asm);
e. Asamblaţi aplicaţia; generaţi şi listingul;
f. Linkeditaţi aplicaţia;
g. Executaţi programul cu td.exe.

2. După modelul prezentat, scrieţi un program care determină minimul dintr-un şir de 10 numere.

3. Scrieţi un program care calculează suma elementelor unui şir de numere. Parcurgeţi şirul în două moduri: prin adresare bazată indexată şi folosind instrucţiuni specifice şirurilor.

4. Dezvoltaţi programele pentru reuniunea, intersecţia şi diferenţa a două şiruri.

5. Scrieţi aplicaţiile care ordonează un şir de 10 elemente, definite ca octeţi în memorie, crescător (descrescător) considerând elementele ca numere fără semn şi cu semn.

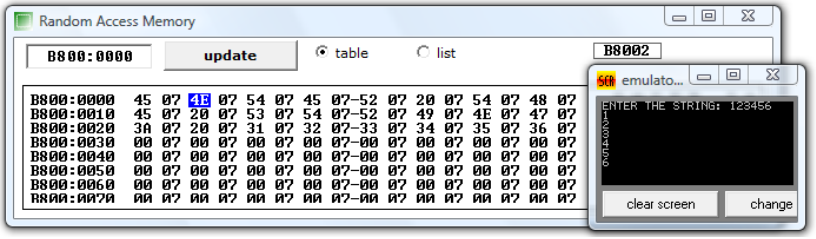
9.3 Probleme rezolvate, folosind macroinstructiuni

1. Sa se afiseze pe ecran valoarea din registru AL in formatele: zecimal fara semn, respectiv binar (de exemplu pentru AL=0FEH pe ecran se va tipari pe prima linie *254* iar pe cea de-a doua linie *11111110b*)

```
name "printAL"
org 100h

mov al, 0FEh
call print_al       ; afiseaza pe prima linie valoarea din Al
                    ; in format zecimal fara semn
call print_nl       ; trece la urmatoarea linie
call print_al_bin   ; afiseaza pe a doua linie valoarea din Al
                    ; in format binar
ret
print_al proc
    cmp al, 0
    jne print_al_r
    push ax
    mov al, '0'
    mov ah, 0eh
    int 10h
    pop ax
    ret
print_al_r:
    pusha
    mov ah, 0
    cmp ax, 0
    je pn_done
    mov dl, 10
    div dl
    call print_al_r
    mov al, ah
    add al, 30h
    mov ah, 0eh
    int 10h
    jmp pn_done
pn_done:
    popa
```

```
    ret
endp

print_al_bin proc
    pusha
    mov cx, 8
    mov bl, al
p1: mov ah, 2
    mov dl, '0'
    test bl, 10000000b
    jz zero
    mov dl, '1'
zero:  int 21h
    shl bl, 1
    loop p1
    mov dl, 'b'
    int 21h
    mov dl, 0Dh
    int 21h
    mov dl, 0Ah
    int 21h
    popa
    ret
endp

print_nl proc
    push ax
    push dx
    mov ah, 2
    mov dl, 0Dh
    int 21h
    mov dl, 0Ah
    int 21h
    pop dx
    pop ax
    ret
endp
```

2. Următorul program preia un şir de la tastatură si apoi afişează pe ecran elementele şirului, câte unul pe linie:



```
name "charchar"
org  100h

print_new_line macro
    mov dl, 13
    mov ah, 2
    int 21h
    mov dl, 10
    mov ah, 2
    int 21h
endm

    mov dx, offset msg1
    mov ah, 9
    int 21h
    mov dx, offset s1
    mov ah, 0ah
    int 21h

    xor cx, cx
    mov cl, s1[1]
    print_new_line

    mov bx, offset s1[2]
print_char:
    mov dl, [bx]
    mov ah, 2
    int 21h
    print_new_line
    inc bx
    loop print_char

    mov ax, 0
    int 16h

    ret

msg1    db  "ENTER THE STRING: $"
s1      db 100,?, 100 dup(' ')
end
```

10. Subrutine, întreruperi, şi servicii

10.1 Subrutine

Subrutina este o secvenţă de instrucţiuni scrisă separat, care poate fi apelată din diferite puncte ale unui program. Mecanismul de implementare a subrutinelor este realizat cu ajutorul instrucţiunilor **CALL** şi **RET**. Subrutinele pot fi NEAR (în acelaşi segment cu programul apelant/intrasegment) sau FAR (într-un segment diferit/extrasegment). La apelarea unei subrutine (prin CALL), adresa unde urmează a se face revenirea este salvată pe stivă, iar la revenirea din rutină (prin RET) adresa este refăcută din stivă iar registrul IP (eventual şi CS) se reîncarcă.

Mecanismul de apel a unei subrutine este ilustrat în figura 10.1.

1. program principal
2. apel subrutină
3. execuţie subrutină
4. întoarcere din subrutină
5. reluare program principal

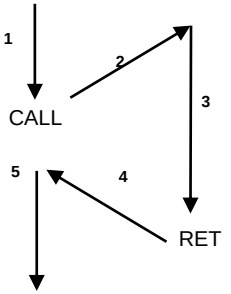


Fig. 10.1. Mecanismul de apel al unei subrutine

Apelurile intrasegment salvează doar offsetul adresei de revenire, iar la RET această valoare este reîncărcată în IP. Apelurile intersegment salvează şi conţinutul registrului CS şi cel al registrului IP, astfel că, la revenire, se vor reface ambele. Pentru cele două cazuri, instrucţiunile CALL şi RET au coduri diferite.
Instrucţiunea RET nu are operanzi; instrucţiunea CALL are un singur operand. Modurile de adresare sunt identice cu cele de la JMP, cu excepţia adresării relative. În cazul în care subrutina alterează (foloseşte) regiştrii a căror valoare este necesară în continuare în programul ppellant, aceşti regiştri trebuie salvaţi pe stivă cu PUSH şi apoi refăcuţi cu POP înainte de revenirea din subrutină.

10.2. Întreruperi

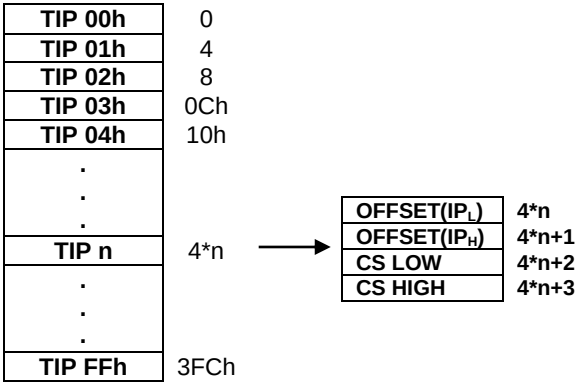
Întreruperea este un semnal transmis sistemului de calcul prin care acesta este anunţat de apariţia unui eveniment care necesită atenţie.

Atunci când evenimentul s-a produs, au loc, în ordine, următoarele acţiuni:
- suspendarea programului în curs de desfăşurare; salvarea pe stivă a adresei de revenire (IP, CS);
- lansarea în execuţie a unei rutine specializate numită rutină de tratare a întreruperii care deserveşte întreruperea;
- reluarea execuţiei programului suspendat prin refacerea de pe stivă a adresei de revenire.

Cauzele acestor evenimente pot fi de două tipuri: interne şi externe. O întrerupere este luată în considerare numai între execuţiile a două instrucţiuni maşină succesive. Dacă apar simultan două întreruperi, circuitele hard ale sistemului de calcul decid care dintre ele va fi servită prima.

La apariţia unei întreruperi, sistemul de calcul trebuie, în această ordine:
- să determine tipul evenimentului care a generat întreruperea (intern, extern),
- să afle care este cauza întreruperii,
- să determine adresa rutinei de tratare a întreruperii (RTI).

Pentru fiecare tip de eveniment şi pentru fiecare cauză posibilă se construieşte câte o RTI. Metoda folosită pentru localizarea rapidă a RTI este vectorizarea întreruperilor. Aceasta constă în a asocia pentru fiecare întrerupere o locaţie de memorie cu adresă fixă. În această locaţie se trece adresa RTI corespunzătoare întreruperii. Microprocesorul 8086 dispune de 256 întreruperi numerotate de la 00h la FFh. Vectorizarea acestora se realizează astfel: la începutul memoriei RAM sunt rezervate 256 de dublucuvinte; fiecare dublucuvânt conţine o adresă FAR a unei RTI. Primul dublucuvânt, aflat la adresa *0000:0000*, conţine adresa RTI pentru intreruperea 00h; al doilea, aflat la adresa *0000:0004* conţine adresa RTI pentru întreruperea 01h, etc. Pentru o întrerupere k, dublul cuvânt care conţine adresa RTI se află la adresa *0000:4*k*



Întreruperile se clasifică în întreruperi hard şi soft. Întreruperea hard 00h provine de la microprocesor (internă) şi apare la tentativa de împărţire la zero; întreruperea 08h provine de la circuitul de temporizare şi este folosită pentru contorizarea timpului; întreruperea 02h semnalizează eroare de paritate la accesarea unei locaţii de memorie şi este o întrerupere nemascabilă, adică declanşarea ei nu poate fi controlată prin flagul *Interrupt.* Întreruperile soft oferă accesul la servicii BIOS şi servicii DOS; sunt foarte mult utilizate datorită facilităţii oferite, o bază de programe care poate fi folosită ca o librărie de programe (rutine) gata scrise, care uşurează mult munca programatorului în limbaj de asamblare. Aceste rutine poartă numele de servicii. Se vor exemplifica servicii pentru câteva întreruperi folosite intensiv în scrierea programelor.

Mecanismul de tratare a unei întreruperi este asemănător cu cel al subrutinelor:

- la lansarea unei întreruperi, indiferent de felul acesteia, starea curentă a microprocesorului este salvată pe stivă; se salvează pe stivă şi PSW; alte întreruperi sunt dezactivate;
- microprocesorul identifică adresa unde se află subrutina de tratare a întreruperii; în acest scop, numărul asociat întreruperii (tipul întreruperii) este folosit ca index în tabloul vectorilor de întreruperi;
- se încarcă în CS şi IP adresa subrutinei din poziţia corespunzătoare a tabloului vectorilor de întrerupere; se execută rutina de tratare a întreruperii până la întâlnirea instrucţiunii IRET;
- se revine din întrerupere prin reîncărcarea lui IP, CS şi PSW cu valorile salvate pe stivă la apelare.

10.3 Instrucţiuni specifice întreruperilor

O întrerupere soft poate fi apelată prin instrucţiunea **INT**, cu sintaxa INT n, provocând activarea handler-ului corespunzător întreruperii cu numărul n. Ea realizează patru acţiuni succesive:

- pune în stivă flagurile (PSW);
- pune în stivă adresa FAR de revenire (CS, IP);
- pune 0 în flagurile TF şi IF;
- apelează prin adresare indirectă handlerul asociat întreruperii.

Instrucţiunile **STI** şi **CLI** acţionează asupra flagului de întreruperi IF, indicând procesorului cum să se comporte la apariţia unei întreruperi. După CLI (Clear Interrupt, IF=0), procesorul nu mai acceptă vreo întrerupere. Apare de obicei la începutul unui handler pentru a evita perturbarea activităţii acestuia. STI (Set Interrupt) permite procesorului să accepte întreruperi, IF=1.

OBS. Întreruperile nemascabile nu ţin cont de starea flagului IF!

Instrucţiunea **IRET** provoacă revenirea dintr-o întrerupere. Ea este ultima instrucţiune executată într-un handler, având efect complementar instrucţiunii INT:

- reface flagurile din stivă;
- revine la instrucţiunea a cărei adresă FAR se află în vârful stivei

```
INT N                          IRET
SP ← Sp-2                      IP ← [SP]
[SP] ← PSW                     SP ← SP+2
I ← 0                          CS ← [SP]
SP ← SP-2                      SP ← SP+2
[SP] ← CS                      PSW ← [SP]
SP ← SP-2                      SP ← SP+2
[SP] ← IP
IP ← [4*N+1 4*N]
CS ← [4*N+3 4*N+2]
```

10.4 Întreruperi şi servicii BIOS şi DOS

BIOS-ul răspunde de gestionarea echipamentelor de intrare/ieşire. Deci, în BIOS sunt scrise o serie întreagă de subrutine legate de aceste echipamente. Apelarea lor într-o aplicaţie se face prin întreruperi; în cadrul fiecărei întreruperi se pot executa mai multe servicii, selecţia serviciului dorit fiind realizată prin încărcarea în registrul AH a unui număr specific serviciului, înainte de apelarea întreruperii. Parametrii de apel ai serviciului se încarcă în anumiţi regiştri, după caz.

Câteva **întreruperi BIOS** sunt prezentate mai jos:

INT 10 h - servicii de ecran / video
INT 13 h - servicii de disc
INT 14 h - servicii de comunicaţii seriale
INT 16 h - servicii de tastatură
INT 20 h - terminare program

Servicii video - INT 10 h

Funcţia 00 - setarea modului video
AH = 00
AL = codul modului video (vezi Anexa 3)
Funcţia 02 - setarea poziţiei cursorului
AH = 02
BH = numărul paginii video (0 pentru modul grafic)
DH = rândul
DL = coloana
Funcţia 09 - scrierea caracterului la cursor (fără deplasarea cursorului)
AH = 09

AL = codul ASCII al caracterului de scris
BH = pagina video
BL = atribut de culoare

Funcţia 0ch - scrierea unui pixel grafic la coordonate
AH = 0ch
AL = culoarea
BH = pagina video
CX = coloana
DX = rândul

Funcţia 0eh - scrierea unui caracter în mod teletype (cu deplasarea cursorului)
AH = 0eh
AL = codul ASCII al caracterului de scris
BH = pagina video

Servicii tastatură - INT 16h

Funcţia 00 - aşteaptă o tastă şi citeşte caracterul tastat
AH = 00
AL = (returnat) codul ASCII al caracterului tastat

Funcţia 01 - citeşte starea tastaturii
AH = 01
Z = 0 - s-a apăsat o tastă; Z = 1 - nu s-a apăsat tastă

Funcţii DOS

Principala întrerupere DOS este **21h**. Sarcinile unora dintre întreruperile amintite mai sus au fost preluate şi uneori extinse de către unele din funcţiile acestei întreruperi.

Funcţia 01 - citire caracter de la tastatură
AH=01
AL = (returnat) caracterul citit

Funcţia 09 - scrierea unui şir de caractere terminat cu "$"
AH = 09
DS:DX - pointer la un şir ce se termină cu caracterul "$"

Funcţia 4ch - terminarea procesului cu cod de retur
AH = 4ch
AL = cod de retur
metodă de terminare a unui program; nu este suportată de versiuni DOS sub 2.x.

Redirectarea unei întreruperi

Rutina de tratare a unei întreruperi oarecare poate fi rescrisă de către utilizator, respectând anumite reguli, legate în primul rând de structura rutinei respective, apoi de manipularea adreselor acestor rutine. În scopul înlocuirii unei RTI cu o rutină scrisă de utlizator, se va înlocui la adresa

corespunzătoare rutinei în TVI cu adresa noii rutine. Ca o măsură firească de precauţie, cel care modifică adresa unui handler trebuie să păstreze adresa veche şi să o refacă atunci când nu mai doreşte folosirea handlerului propriu.
Funcţia 35 a întreruperii 21h permite citirea adresei RTI din TVI pentru o anumită întrerupere dorită de utilizator. Se pune în AH codul funcţiei (35h) iar în AL tipul întreruperii. După apelul INT 21h în regiştrii ES:BX se obţine adresa FAR a handlerului.
Funcţia 25h a întreruperii 21h permite modificarea adresei RTI în TVI pentru o anumită întrerupere dorită. În AH se pune codul funcţiei (25h), în AL se pune tipul întreruperii dorite, iar în DS:DX se pune adresa FAR a noului handler.

10.5 Exerciţii şi teme

1. Studiaţi programul `afisare.asm`; programul face afişarea textului "TASM" în diagonală pe ecran. Modificaţi atributele de culoare pentru afişarea textului.

```
.model small
.stack 200h
.data
        text db 'TASM',0
.code

linie   proc    near
        mov     cx,1
        mov     dx,0
linie_iar:
        lea     si,text
iar:
        mov     ah,2
        int     10h
        mov     al,[si]
        cmp     al,0
        jz      endtext
        mov     ah,9
        int     10h
        inc     dl
        inc     si
        cmp     dl,80
        jnz     iar
endtext:
        inc     dh
        cmp     dh,25
        jc      linie_iar
        ret

linie   endp
```

```
main label

        mov     ax,@data
        mov     ds,ax
        mov     ah,0
        mov     al,2
        int     10h
        mov     bh,0
        mov     bl,1fh
        call    linie
        mov     ax,4c00h
        int     21h

end main
```

2. Scrieţi un program care afişează un mesaj de test definit în segmentul de date, folosind pentru afişare funcţii ale întreruperii 10h şi 21h; Nu omiteţi încheierea programului! (int 21h, serviciul 4ch sau prin ret)

```
org 100h
.data
    msg db "Hello, World", 24h

.code
    mov ax, @data
    mov ds, ax

    mov dx, offset msg
    mov ah, 9
    int 21h
.exit
```

3. Completaţi programul de adunare a două numere prin afişarea acestora şi a sumei pe ecran. În acest scop trebuie realizată conversia valorilor din binar în ASCII, pentru a putea fi afişate folosind funcţiile prezentate.

4. Considerăm următorul program în limbaj de asamblare:

```
        org     100h
        mov     si,0
        mov     dl,30h
        mov     ah,2
lop:    int     21h
        inc     dl
        inc     si
        cmp     si,8h
        jne     lop
        int     20h
```

a. Ce conţin regiştrii SI şi DL în momentul în care programul iese din buclă?

b. Ce se va afişa pe ecranul PC-ului la terminarea programului?

```
SI conţine 08H
DI conţine 38H
```

Pe ecran va fi scris: `01234567`

5. Scrieţi un program care începe să citească caractere de la tastatură şi să stocheze codul lor ASCII în memorie începând de la adresa 2000h, până la apăsarea tastei Enter (0Dh).

```
        org     100h
        mov     si,0
        mov     ah,1
lop:    int     21h
        cmp     al,0dh
        je      out1
        mov     2000h[si],al
        inc     si
        jmp     lop
out1:   int     20h
```

6. Comentaţi linie cu linie următorul program şi apoi explicaţi ce se va afişa pe ecranul PC-ului. Pentru verificare rulaţi programul cu emu8086:

```
name "colors"
org     100h

        mov     ax, 3
        int     10h

        mov     ax, 1003h
        mov     bx, 0
        int     10h

        mov     dl, 0
        mov     dh, 0

        mov     bl, 0

        jmp     next_char

next_row:
        inc     dh
        cmp     dh, 16
        je      stop_print
        mov     dl, 0
```

```
next_char:
        mov     ah, 02h
        int     10h

        mov     al, 'a'
        mov     bh, 0
        mov     cx, 1
        mov     ah, 09h
        int     10h

        inc     bl

        inc     dl
        cmp     dl, 16
        je      next_row
        jmp     next_char

stop_print:

        mov     dl, 10
        mov     dh, 5
        mov     ah, 02h
        int     10h

        mov     al, 'x'
        mov     ah, 0eh
        int     10h

        mov ah, 0
        int 16h

ret
```

11. Interfaţarea aplicaţiilor în limbaj de asamblare cu sistemul de operare DOS

11.1 Execuţia aplicaţiilor DOS

Sub sistemul de operare DOS există trei tipuri de fişiere care se pot lansa în execuţie: fişiere de tip .BAT, fişiere de tip .EXE şi fişiere de tip .COM. Fişierele de tip BAT sunt fişiere text care conţin comenzi DOS şi eventual directive care controlează ordinea de exEcuţie a comenzilor. La lansarea în execuţie a unui fişier de tip BAT sunt executate una după alta toate comenzile conţinute în fişier, respectându-se semnificaţiile directivelor. Fişierele COM şi EXE sunt fişiere binare ce conţin instrucţiuni în limbaj maşină; din punct de vedere istoric, primele apărute sunt fişierele de tip COM, fişiere având o lungime mai mică de 64Ko (un singur segment). La încărcarea unui program executabil, harta memoriei arată în felul următor:

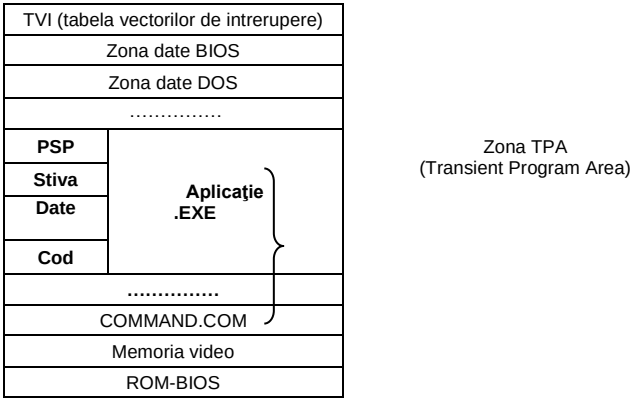


Fig.11.1. Harta memoriei PC-ului

11.2 Prefixul unui program executabil (PSP)

Imaginea în memorie a unui program de tip EXE sau COM începe cu un antet numit PSP (Program Status Prefix). În momentul încărcării programului, imaginea lui în memorie este completată cu acest tabel de 256 octeţi. Informaţiile din PSP sunt utilizabile direct de către sistemul de operare DOS şi indirect de către utilizator. Structura acestuia este indicată în tabelul următor (detaliile sunt prezentate în anexa 2).

În PSP există o serie de zone care în prezent sunt mai puţin folosite. Ele au fost introduse la prima versiune de DOS şi nu mai sunt folosite începând cu DOS 2.0, dar sunt păstrate pentru a asigura compatibilitatea programelor executabile DOS din versiunile mai noi cu cele din prima versiune.
Fiecare program, pe lângă codul lui propriu zis mai are o zonă de memorie în care este descris contextul în care lucrează programul. Acest context include, printre altele, informaţii referitoare la numele discului (este implicit), numele directorului (este implicit), calea spre interpretorul de comenzi COMMAND.COM etc. Această zonă de context este un segment numit *mediu (environment)* şi PSP conţine pointer la începutul lui.

00h	**Codul instrucţiunii INT 20h**
02h	**Sfârşitul memoriei ocupate de program**
04h	Un octet rezervat
05h	Codul instrucţiunii INT 21h
0Ah	Adresa FAR a RTI 22h
0Eh	Adresa FAR a RTI 23h
12h	Adresa FAR a RTI 24h
16h	21 octeţi rezervaţi
2Ch	Adresa segmentului de mediu
2Eh	46 octeţi rezervaţi
5Ch	FCB1 şi FCB2, câte 16 octeţi pentru fişierele standard de intrare şi ieşire (azi evitaţi)
80h	**Lungimea cozii liniei de comandă**
81h	**Coada liniei de comandă (<127 octeţi)**

Tabelul 11.1. Structura PSP

Se ştie că o comandă DOS are forma :
. . . >numecomanda *param1, . . . , paramn*
Porţiunea *param1, . …, paramn* se numeşte coada liniei de comandă şi ea este memorată în jumătatea a doua a tabelei PSP.

Un program de tip COM are o structură simplă. El conţine imaginea binară a conţinutului ce va fi încărcat în memorie după PSP. Un program de tip EXE poate avea oricâte segmente de tip cod, date sau stivă, toate fiind plasate după PSP, însă ordinea lor nu este importantă.

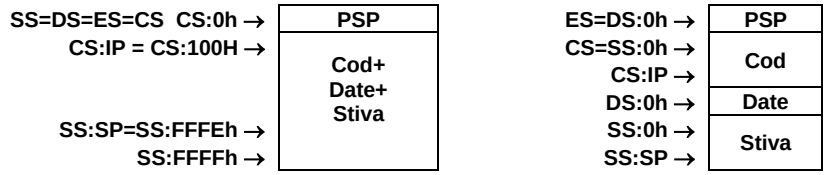


Fig.11.2 Imaginea unui program COM şi a unui program EXE în memorie

11.3 Exerciţii şi teme

1. Următoarea aplicaţie afişează parametrii din linia de comandă, scrişi cu majuscule:

```
; afişare mesaj din linia de comandă (PSP)
; >psp ***** 5 caractere cu litere mici
; programul le va scrie cu litere mari
; scad din codul ASCII 20h
; se face şi verificarea liniei de comandă

.model small
.stack 200h
.data
        mesaj db 'Eroare în linia de comanda!!', '$'
.code
main label
        mov ax,@data
        mov ds,ax

;verificare parametri: numărul de caractere introduse
;(6=spaţiu +5 caractere)

       mov bx, 80h
       mov al, es:[bx]
       cmp al,6
       jz next
       mov dx, offset mesaj
       mov ah, 09h
       int 21h
       jmp iesire

;afisare din PSP

next:  mov bx, 82h
       mov ah,0eh
       mov cx,0

repeta:mov al,es:[bx]
       sub al, 20h
       int 10h
       inc bx
       inc cx
       cmp cx,5
       jnz repeta

iesire:mov ax,4c00h
       int 21h


end main
```

2. Modificaţi programul `afisare.asm` prezentat în lucrarea anterioară, pentru a afişa un text introdus în linia de comandă.

3. Scrieţi un program care adună două numere de câte două cifre introduse din linia de comandă sub forma `aduna ab cd` şi afişează rezultatul. În acest scop va trebui făcută conversia din ASCII în binar pentru realizarea operaţiei de adunare şi apoi din binar în ASCII pentru afişare.

```
; conversie ASCII - Binar pentru un număr între 0 şi 99, aflat
; în DX
convAsciiBin proc far
   xor ah,ah
   mov cl,10
   and dx,0f0fh
   mov al,dh
   mul cl
   mov bl,al
   add bl,dl
   ret
convAsciiBin endp

; conversie Binar - ASCII pentru un număr între 0 şi 255 aflat
; în AL
convBinAscii proc near
   xor ah,ah
   mov cl,10
   div cl              ; studiaţi modul de lucru al div
   add ah,30h
   mov [ASCII+2],ah    ; unităţi
   xor ah,ah
   div cl
   add ah,30h
   mov[ASCII+1],ah     ; zeci
   add al,30h
   mov [ASCII],al      ; sute
   ret
convBinAscii endp
```

4. Scrieţi şi o procedură de verificare a corectitudinii parametrilor daţi în linia de comandă (două numere din câte două cifre şi spaţiu între ele). Codurile ASCII ale cifrelor 0-9 sunt 30h-39h.

12. Setul extins de instrucţiuni x86

12.1 Generalităţi

Evoluţia arhitecturii microprocesoarelor INTEL de la 16 la 32 de biţi implică anumite modificări ale dimensiunii regiştrilor precum şi adăugarea unor resurse arhitecturale suplimentare.
În figura următoare este prezentată arhitectura simplificată a unui procesor Pentium. Procesorul Pentium are o arhitectură *superscalară* ceea ce îi permite în anumite condiţii să execute două instrucţiuni în acelaşi timp prin cele două pipeline-uri de procesare paralelă a datelor.

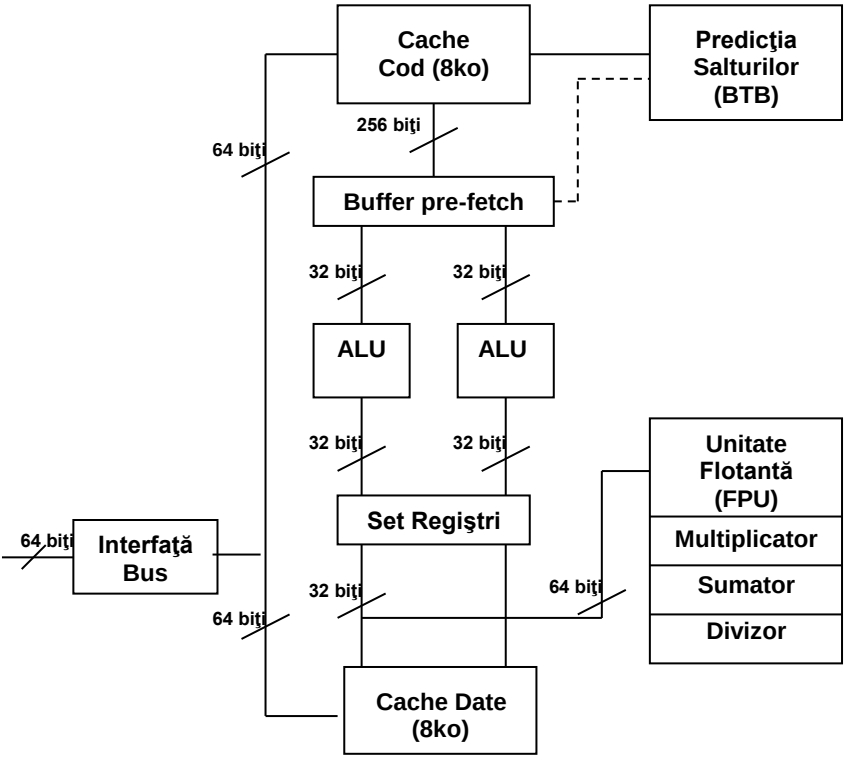


Fig. 12.1. Arhitectura simplificată a procesorului Pentium

Principalele elemente arhitecturale ale schemei sunt: regiştrii, unităţile aritmetice şi logice, memoria cache de date şi cod (8Ko+8Ko), unitatea în virgulă flotantă, blocul pentru predicţia salturilor şi interfaţa cu magistralele. *Setul de regiştri* este cel descris în tabelul 12.1.

Tip regiştri	Biţi	Denumire
Generali	32	EAX, EBX, ECX, EDX
	16	AX, BX, CX, DX
	8	AH, AL, BH, BL, CH, CL, DH, DL
Pointer	32	ESP, EBP
	16	SP, BP
Index	32	ESI, EDI
	16	SI, DI
Segment	16	CS, DS, SS, ES, FS, GS
Stare şi control	32	EIP, EF, CR0…CR4, DR0…DR7
Alţi regiştri	16	IP, F, MSW, GDTR, LDRT, IDTR, TR

Tabelul 12.1. Setul de regiştri al procesorului Pentium

386/486/Pentium — regiştri pe 32 de biţi

Regiştri generali si de adresă:
EAX — acumulatorul implicit
EBX — conţine implicit o adresă de bază pentru anumite moduri de adresare
ECX — contor implicit
EDX — acumulator extins implicit sau registru de date
ESI — index pentru sursă
EDI — index pentru destinaţie
EBP — indicatorul bazei în stivă
ESP — indicatorul curent în stivă

Regiştri de stare şi control:
EIP — indicator (numărător) de instrucţiuni
EF — registrul de flaguri
CR0…CR4 — regiştri de control
DR0…DR7 — regiştri pentru depanare (debug)

Regiştri segment:
FS — registru segment suplimentar (de date noi)
GS — registru segment suplimentar (de date noi)

Cele două pipeline-uri standard de procesare a instrucţiunilor dispun şi de două unităţi de calcul **ALU** întregi.
Memoriile **cache** sunt separate: pentru date (8Ko) şi pentru instrucţiuni (8Ko). Fiecare memorie cache are câte un modul TLB (Translation Lookaside Buffer) dedicat, care converteşte adresele logice succesive în adrese fizice. Conţine de asemenea şi un coprocesor matematic încorporat **FPU** *(Floating Point Unit).* Se estimează că unitatea de calcul în virgulă mobilă FPU a procesorului Pentium este de 2-10 ori mai rapidă decât cea a procesorului 486.

Modulul numit **BTB** *(Branch Target Buffer)* utilizează ca tehnică predicţia salturilor (*branch prediction*) în scopul reducerii timpului de aşteptare în canalele de procesare. La fiecare salt microprocesorul stochează adresa instrucţiunii de salt şi adresa destinaţiei saltului. BTB încearcă să prevadă apariţia unei instrucţiuni de salt şi să extragă din memorie instrucţiunile corespunzătoare ramurii la care se va face saltul. Salturile anticipate corect nu introduc întârzieri în prelucrare.

Procesorul are o magistrală de adrese pe 32 de biţi şi poate astfel să adreseze 4Go de memorie ca şi procesoarele 386DX şi 486; procesorul Pentium extinde magistrala de date la 64 de biţi, ceea ce înseamnă că poate transfera sistemului de două ori mai multe informaţii, la aceeaşi frecvenţă de ceas. În interior însă procesorul Pentium are regiştri de 32 de biţi care sunt compatibili cu cei ai procesoarelor anterioare.

12.2 Setul extins de instrucţiuni

Utilizarea setului extins de instrucţiuni în aplicaţii trebuie anunţată asamblorului folosind directivele:
.8086 .8087 .186 .286 .287 ;
.286P .386 .387 .386P .486;
.486P .586 .586P
Aceste directive nu acceptă operanzi. Directivele procesor permit utilizarea tuturor instruncţiunilor pe un procesor dat. Directivele .8087, .287, .387 activează setul de instrucţiuni în virgulă flotantă. Scopul acestor directive este să permită instrucţiuni 80287 cu setul 8086 sau 80186, sau instrucţiuni

80387 cu setul 8086, 80186 sau 80286. Directivele ce se termină cu P permit asamblarea unor instrucţiuni pentru privilegii. Acestea sunt utile celor care scriu sisteme de operare, drivere pentru anumite dispozitive şi alte rutine de sistem.

12.2.1 Instrucţiuni de transfer

MOVSX (Move with Sign Extension) – copiază un operand pe 8 biţi la o destinaţie pe 16 sau 32 de biţi, sau un operand pe 16 biţi la o destinaţie pe 32 de biţi prin extensia corespunzătoare a semnului operandului sursă. Flagurile nu se modifică.

MOVSX destinaţie, sursă *destinaţie ← sign_extend(sursă)*

Operanzi: reg,reg
reg,mem

Exemplu:

```
MOVSX       EAX, AL              ;octet → dublucuvânt
MOVSX       EDI, WORD PTR [ESI]  ;cuvânt → dublucuvânt
```

MOVZX (Move with Zero Extension) – copiază un operand pe 8 biţi la o destinaţie pe 16 sau 32 de biţi, sau un operand pe 16 biţi la o destinaţie pe 32 de biţi precum şi extensia cu zero a sursei. Extensia se realizează prin "umplerea" cu zero a biţilor superiori ai destinaţiei. Flagurile nu se modifică.

MOVZX destinaţie, sursă *;destinaţie ← sursă*

Operanzi: reg,reg
reg,mem

Exemplu:

```
MOVZX       EAX, AL              ;octet→ dublucuvânt
```

PUSHFD (Push EFLAGS Registers) – copiază registrul de flaguri **EF** în stivă. Instrucţiunea nu are operanzi şi nu afectează nici un flag.

PUSHFD *; ESP ← ESP – 4*
[SS:ESP] ← EF

POPFD (Pop Stack into EFLAGS) – aduce vârful stivei în registrul de flaguri **EF.** Instrucţiunea nu are operanzi, iar flagurile se schimbă în conformitate cu rezultatul operaţiunii.

POPFD *; EF ← [SS:ESP]*
ESP ← ESP + 4

PUSHF (Push 16-bit EFLAGS Registers) – copiază cei mai puţin semnificativi 16 biţi ai registrului de flaguri în stivă. Instrucţiunea nu are

operanzi şi nu afectează nici un flag. Asigură compatibilitatea cu procesoarele pe 16 biţi şi poate produce decalarea stivei.

```
PUSHF                    ;ESP ← ESP – 2
                         [SS:ESP] ← EFlow
```

POPF (Pop Stack into FLAGS) – aduce din stivă cei mai puţin semnificativi 16 biţi ai registrului **EF.** Instrucţiunea nu are operanzi, iar flagurile se schimbă în mod corespunzător.

```
POPFD                    ;EFlow ← [SS:ESP]
                         ESP ← ESP + 2
```

PUSHAD (Push 32-bit General Registers) – copiază toţi cei 8 regiştri pe 32 de biţi, pe stivă. Valoarea lui ESP salvată pe stivă este cea dinaintea execuţiei instrucţiunii PUSHAD. Registrul de flaguri nu se modifică.

```
temp ← ESP
PUSH   EAX
PUSH   ECX
PUSH   EDX
PUSH   EBX
PUSH   temp
PUSH   EBP
PUSH   ESI
PUSH   EDI
```

POPAD (Pop All General Registers) – reface din stivă toţi regiştrii generali pe 32 de biţi cu excepţia lui ESP. Registrul de flaguri nu se modifică.

```
POP    EDI
POP    ESI
POP    EBP
ESP ←  ESP + 4
POP    EBX
POP    EDX
POP    ECX
POP    EAX
```

OBS: Instrucţiunea **PUSHA/ POPA** copiază/ reface toţi cei 8 regiştri de 16 biţi, pe/din stivă.

SAHF (Store AH în EFLAGS) – încarcă conţinutul registrului AH în octetul cel mai puţin semnificativ al registrului EF, cu mascarea biţilor rezervaţi (7,6,4,2 şi 0).

```
EF ← EF or  (AH and 0D5H)
```

Lseg (Load Segment Register) – adresa sursă specifică un pointer pe 48 de biţi conţinând o adresă efectivă pe 32 de biţi urmată de un selector pe 16 biţi. Adresa efectivă este încărcată în registrul destinaţie, iar selectorul în registrul segment specificat prin mnemonica instrucţiunii. Flagurile nu sunt afectate.

LDS	*reg,mem*	*;destinaţie ← [sursă]*
LES	*reg,mem*	*;seg ← [sursă+4]*
LFS	*reg,mem*	
LGS	*reg,mem*	
LSS	*reg,mem*	

BSWAP (Byte Swap) – această instrucţiune se utilizează atunci când se doreşte un schimb de date între procesoare cu arhitecturi diferite. Se realizează conversia între formatele "big-endian" şi "little-endian" schimbând ordinea celor 4 octeţi care compun data conţinută de registrul precizat. (Daca ar fi posibilă folosirea unui operand pe 16 biţi, atunci de exemplu bswap AX ar fi echivalentă cu xchg AH,AL).

BSWAP reg32	*;temp*	*← reg*
	;reg[0…7]	*←* temp[*24…31]*
	;reg[8…15]	*←* temp[*16…24]*
	;reg[16…23]	*←* temp[*8…15]*
	;reg[24…31]	*←* temp[*0…7]*

Exemplu:

```
MOV EAX, 12345678h     ; EAX=12345678h
BSWAP EAX              ; EAX=78563412h
```

12.2.2 Instrucţiuni de I/O pentru şiruri

INSB, INSW, INSD (Input String from I/O Port) – destinaţia dată de (ES:EDI) sau (ES:DI) primeşte date de la portul de intrare, specificat de către DX. Registrul DI este incrementat dacă flagul DF=0 sau decrementat dacă flagul DF=1 cu dimensiunea operandului (d=1, 2 sau 4). Pot fi folosite împreună cu prefixul REP, iar în acest caz registrul ECX va fi registrul contor. Nu au operanzi.

INSB, INSW, INSD	*(ES:EDI / DI) ← port (DX)*	
	şi (EDI/ DI) = (EDI/ DI) ± (d)	*;d=1 / 2 / 4*

OUTSB, OUTSW, OUTSD (Output String) – se transferă un octet, cuvânt sau un dublucuvânt de la adresa efectivă dată de ESI sau SI la portul specificat prin registrul DX.

OUTSB, OUTSW, OUTSD	*port(DX) ← (ES:ESI/ SI)*	
	şi (ESI/ SI) = (ESI/ SI) ± (d)	*;d=1 / 2 / 4*

12.2.3 Instrucţiuni aritmetice

CDQ (Convert Double Word to Quadword) – converteşte datele pe 32 de biţi aflate în registrul EAX la 64 biţi. Nu este afectat nici un flag. Se utilizează deseori înaintea instrucţiunilor de împărţire când deîmpărţitul este de 64 de biţi.

```
CDQ             ;dacă EAX31 = 1 → EDX = 0FFFFFFFFh
                ;altfel EDX= 0
```

Exemplu:

```
mov EAX, 12345678h     ;EAX=12345678h
cdq                    ;EDX=0h
```

Exemplu:

```
MOV   EAX, [420H]        ;copiază deîmpărţitul în EAX
CDQ                      ;extensie la 64 biţi
IDIV  DWORD  PTR [30H]   ;împărţire cu semn
```

Instrucţiunea **CWDE (Convert Word to Doubleword Extended)** recunoscută de procesoarele de la 80386 şi ulterioare, converteşte cuvântul din AX la dublu-cuvântul din EAX, deci bitul de semn din AX se extinde la tot registrul EAX.

```
CWDE            ; dacă AX15 = 1 → EAX = 0FFFFFFFFh,
                ; altfel EAX= 0
```

Exemplu:

```
mov AX, 0FFA5h         ;AX=0FFA5h
cwde                   ;EAX=0FFFFFFA5h
```

CMPXCHG (Compare and Exchange) – valoarea primului operand este comparată cu valoarea acumulatorului (AX, EAX). Dacă valorile sunt egale, adică ZF=1 valoarea celui de-al doilea operand este copiată în primul. În caz contrar primul operand este copiat în acumulator.

```
CMPXCHG   operand1, operand2     ;dacă   ACC = operand1 atunci
                                 ;ZF=1     operand1 = operand2
                                 ;altfel  ZF=0    ACC = operand1
```

Operanzi: *reg,reg*
mem,reg

Exemplu:

```
cmpxchg [VAR], SI     ;compară AX cu cuvântul din memorie
                       ;adresat de VAR; dacă sunt egale
                       ;=>ZF=1 şi în locaţia adresată de
                       ;VAR se depune conţinutul lui SI
                       ;dacă sunt diferite =>ZF=0 şi în
```

```
                                ;AX se depune conţinutul memoriei
                                ;dat de VAR
```

O instrucţiune asemănătoare cu CMPXCHG este **CMPXCHG8B** care compară cei 8 octeţi ai perechii EDX:EAX cu o zonă de memorie mem64. Dacă sunt egale, ZF=1 şi în acea zonă de memorie se încarcă conţinutul perechii ECX:EBX. Altfel, ZF=0 şi conţinutul zonei de memorie e încărcat in perechea EDX:EAX.

```
CMPXCHG8B  destinaţie      ;dacă EDX:EAX= destinaţie,
                           ;atunci Z=1 şi (destinaţie)=ECX:EBX
                           ;altfel Z=0 şi EDX:EAX =(destinaţie)
```

XADD (Exchange and ADD) – se calculează suma dintre operanzi, iar suma se depune în destinaţie. Valoarea originală a destinaţiei este stocată în sursă.

XADD *destinaţie, sursă* *; destinaţie = destinaţie + sursă*

Operanzi: *reg,reg*
mem,reg

Exemplu:

```
mov EAX,4
mov EBX,6
xadd EAX,EBX   ; EAX=EAX+EBX=> EAX=0000000Ah şi
               ; EBX=00000004h - doar de la 80486↑
```

MUL (*Unsigned* Multiplication) – există un singur operand, înmulţitorul. Se realizează înmulţirea fără semn a numerelor întregi.

În cazul unui operand pe 32 biţi avem:
- deînmulţit: EAX
- produs: EDX | EAX

MUL *sursă* *;acc extins ← acc * sursă*

Operanzi: *reg*
mem

IMUL (*Integer* (*Signed*) Multiplication) – se realizează înmulţirea unor întregi cu semn. Dacă avem un singur operand pe 32 de biţi rezultatul este depus în EDX|EAX. Dacă avem doi sau trei operanzi aceştia trebuie să aibă aceeaşi dimensiune.

```
IMUL    op1
IMUL    op1,op2
IMUL    op1,op2,op3
```

Operanzi:	op1	op2	op3	
	reg			acc ← *acc * reg*
	mem			acc ← *acc * mem*
	reg,	reg		op1 ← *op1*op2*
	reg,	mem		op1 ← *op1*op2*
	reg,	imed		op1 ← *op1*op2*
	reg,	reg,	imed	op1 ← *op2*op3*
	reg,	mem,	imed	op1 ← *op2*op3*

Exemplu:

```
imul CX,DX,10  ;înmulţeşte conţinutul reg. DX cu 10 şi
               ;depunde rezultatul în registrul CX – de
               ;la 80286↑
```

Exemplu:

```
imul EAX,[2]   ;înmulţeşte conţinutul reg. EAX cu
               ;dublucuvântul din locaţia de memorie
               ;(segm DS) care începe la adresa 2,
               ;depunând rezultatul în EAX –de la 80386↑
```

Exemplu:

```
imul ESI,EDI,10        ;înmulţeşte conţ. reg. EDI cu 10
                       ;şi depunde rezultatul în reg.
                       ;ESI – de la 80386↑
```

DIV şi IDIV (Integer (Signed) Division) – valoarea aflată în acumulatorul extins este împărţită la *operand.* Câtul rezultat se stochează în partea mai puţin semnificativă a acumulatorului iar restul în partea cea mai semnificativă a acestuia. Dacă câtul e mai mare decât dimensiunea acumulatorului sau avem împărţire cu zero se produce INT 0.

În cazul unui operand pe 32 biţi avem:
- deîmpărţit: EDX│EAX
- cât: EAX
- rest: EDX

IDIV *operand*

Operanzi: reg
mem

Exemplu:

```
MOV     EAX, [ESP + 16]        ;deîmpărţitul
CDQ                            ;se converteşte la 64 biţi
IDIV    ECX                    ;apoi împarte cu semn
                               ;conţ. perechii EDX:EAX la
                               ;ECX şi depune câtul în
                               ;EAX iar restul în EDX
```

12.2.4 Operaţii logice

BSF (Bit Scan Forward) – se parcurge operandul sursă începând cu bitul cel mai puţin semnificativ şi se scrie poziţia primului bit nenul în destinaţie. Dacă operandul sursă este nul, ZF = 1, iar destinaţia este nedefinită.

BSF destinaţie, sursă

dacă (sursa = 0) atunci
ZF = 1 şi destinaţia ←*???*
altfel ZF ← *0*
temp ← *0*
while (bit(sursa, temp) =0)
temp ← *temp+1*
destinaţie ← *temp*

Operanzi: *reg,reg*
reg,mem

Exemplu:

```
MOV     AX, 85C0H
BSF     BX, AX                  ;(BX) ← 6
```

BSR (Bit Scan Reverse) – se parcurge operandul sursă începând cu bitul cel mai semnificativ şi se scrie poziţia primului bit nenul în destinaţie. Dacă operandul sursă este nul, ZF = 1 iar destinaţia este nedefinită.

BSR destinaţie, sursă

dacă (destinaţia este (AX, BX,
CX, DX, SI, DI, BP, SP)) atunci startbit ← 15
altfel
startbit ← 31
dacă (sursa = 0) atunci
ZF = 1 şi destinaţia ← ???
altfel ZF ← 0
temp ← startbit
while (bit (sursa, temp) = 0)
temp ← temp-1
destinaţie ← temp

Operanzi: *reg,reg*
reg,mem

Exemplu:

```
mov AX,1234h            ;AX=1234h=0001 0010 0011 0100b
bsr DX,AX               ;DX=12=0Ch
```

Operaţii cu flaguri

BT (Bit Test) – instrucţiunea testează un singur bit din destinaţie: cel a cărui poziţie este precizată prin index. Valoarea acestui bit este încărcată în CF şi ulterior poate fi testată.

BT destinaţie, index ; CF ← destinaţie [index]

Operanzi: reg, data
mem, data
reg,reg
mem,reg

Exemplu:

```
       mov EAX, 23     ;se verifică dacă bitul 23 este setat,
                       ;deci dacă tehnologia MMX este suportată
       mov EBX, 0387F9FFh     ; destinaţia
       bt EBX,EAX             ;EBX=0387F9FFh;
;EBX=0000 0011 1000 0111 1111 1001 1111 1111b cu b23=1,deci CF=1
       jc etSuportaMMX        ;salt la o etich. ce tratează
                              ;cazul în care proc suportă MMX
```

BTC (Bit Test and Complement) – instrucţiunea ajută la testarea unui bit din destinaţie a cărui poziţie este precizată prin index. Valoarea acestui bit este încărcată în CF şi ulterior poate fi testată. Valoarea sa este apoi inversată.

BTC destinaţie, index ;CF ← destinaţie [index]
*;destinaţie [index] ← **not** destinaţie [index]*

Operanzi: reg, data
mem, data
reg,reg
mem,reg

BTR (Bit Test and Reset) – instrucţiunea testează un bit din destinaţie a cărui poziţie este precizată prin index. Valoarea acestui bit este încărcată în CF şi ulterior poate fi testată. Valoarea sa devine zero.

BTR destinaţie, index CF ← destinaţie [index]
destinaţie [index] ← 0

Operanzi: reg, data
mem, data
reg,reg
mem,reg

BTS (Bit Test and Set) – instrucţiunea testează un bit din destinaţie a cărui poziţie este precizată prin index. Valoarea acestui bit este încărcată în CF şi ulterior poate fi testată. Valoarea sa devine unu.

BTR	*destinaţie, index*	*CF ← destinaţie [index]* *destinaţie [index] ← 1*

Operanzi: *reg, data*
mem, data
reg,reg
mem,reg

Exemplu:

```
mov EAX, 23    ;se verif. dacă bitul 23 este setat,
mov EBX, 0387F9FFh     ;destinaţia
bts EBX,EAX    ;CF=1,b23 rămâne 1, deci
               ;EBX=038 7F9FFh (neschimbat)
btr EBX,EAX    ;CF=1, b23 devine 0, deci
               ;EBX=030 7F9FFh
btc EBX,EAX    ;CF=0,b23 devine 1 prin complementare
               ;deci EBX=038 7F9FFh (neschimbat)
```

SETcc (Set Byte on Condition) – dacă condiţia este îndeplinită, instrucţiunea setează octetul precizat de destinaţie, iar în caz contrar octetul devine 0.
Condiţia *‚cc'* poate fi: A/ AE/ B/ BE/ C/ E/ G/ GE/ L/ LE/ NA/ NAE/ NB/ NBE/ NC/ NE/ NG/ NGE/ NL/ NLE/ NO/ NP/ NS/ NZ/ O/ P/ PE/ PO/ S/ Z

Exemplu:

```
SETC   destinaţie    ;Set if Carry/ CF=1
SETLE  destinaţie    ;Set if less or equal/ SF≠OF&ZF=1
```

Exemplu:

```
mov AX,2345h
mov BX,0EDCBh
add AX,BX
setnc CL       ; CF=1 -> CL=0
```

Exemplu:

```
mov EAX,2
mov EBX,3
cmp EAX,EBX    ; AX-BX<0
setle CL       ; is less - > CL=1
```

SHLD (Shift Left Double) şi SHRD (Shift Right Double) – *sursa* este concatenată cu *destinaţia* iar valoarea astfel obţinută se deplasează spre stânga respectiv dreapta. Cei mai puţin semnificativi biţi sunt stocaţi în destinaţie. Operandul *count* este mascat cu valoarea 1FH (adică se face şi logic între *count* şi 1FH) astfel că operandul *count* nu depăşeşte valoarea 31.

SHLD *destinaţie, sursă, count*

temp ← max (*count*, 31)
value ← sursă ↑destinaţie
value ← valoare * 2^{temp}
dest ← valoare

Operanzi: reg, reg, imed
mem, reg, imed
reg, reg, CL
mem, reg, CL

SHRD *destinaţie, sursă, count*

temp ← max (*count*, 31)
value ← sursă ↑destinaţie
value ← valoare div 2^{temp}
dest ← valoare

Exemplu:

```
shld AX, BX, 12        ; conţinutul registrului AX este
                       ;deplasat spre stânga cu 12 poziţii şi umplut de
                       ;la dreapta spre stânga cu cei mai semnificativi
                       ;12 biţi ai registrului BX
```

Exemplu:

```
shrd EAX, EBX, 12      ; conţinutul registrului EBX este
                       ;deplasat spre dreapta cu 14 poziţii şi umplut
                       ;de la stânga spre dreapta cu cei mai puţin
                       ;semnificativi 14 biţi ai registrului EAX
```

12.2.5 Alte instrucţiuni

CPUID (CPU Identification) – returnează informaţii de identificare a procesorului sau despre resursele lui. Executarea instrucţiunii pentru diferite valori din EAX dau o imagine completă despre procesor şi posibilităţile sale. Informaţiile returnate la executarea instrucţiunii CPUID sunt: şirul ASCII de identificare a producătorului, semnătura procesorului, numărul de serie al procesorului, flag-urile de caracteristici ale acestuia, informaţii despre memoria cache, etc.

INVD (Invalidate Cache) – invalidează memoria cache internă; instrucţiunea nu are operanzi şi nu modifică nici un flag.

INVLPG (Invalidate TLB Entry) – invalidează intrarea în TLB (Translation Look-aside Buffer) dacă adresa liniară a paginii în care se găseşte adresa *mem* se află în TLB. Registrul de flaguri nu se modifică.

INVLPG mem

dacă adresa_liniară (mem) este în TLB(i)
atunci invalidează TLB(i)

Exemplu:

```
INVLPG     [SI+4]     ;se invalidează PTE pentru această
                      ;adresă
```

MOV (Move Special) – copiază sau încarcă un registru special al CPU în sau dintr-un registru general. Regiştrii speciali sunt CR0, CR2, CR3 (Control Register), respectiv DR0, DR1, DR2, DR3, DR6, DR7 (Debug Register).

MOV destinaţie, sursă; (destinaţie) ← sursă

Operanzi: reg,reg

Exemplu:
```
MOV     EAX, CR0        ;se salvează CR0 în EAX
MOV     DR7, ECX        ;se încarcă DR7 cu conţinutul ECX
```

RDMSR (Read from Model Specific Register) – conţinutul unui registru MSR, precizat de ECX, este copiat în EDX|EAX. Cu ajutorul MSR pot fi observate şi controlate detaliile specifice procesorului.

RDMSR EDX|EAX ← MSR[ECX]

WRMSR (Write to Model Specific Register) – un operand din EDX|EAX este copiat într-un registru MSR

12.3 Exerciţii şi teme

1. Analizaţi exemplele de mai jos şi apoi executaţi-le cu debugger-ul (TD32.exe)

a).
```
INVD                        ;invalidează memoria cache
MOV    EAX, CR0             ;aduce conţinutul registrului de
                            ;control
AND    EAX, 06000000H       ;validează memoria cache
MOV    CR0, EAX             ;rescrie în registrul de control
```

b). Calculul valorii absolute
```
    MOV    EAX, valoare
    OR     EAX, EAX         ;test pentru semn
    JNS    SKIP             ;salt dacă nu există semn
    NEG    EAX              ;negarea numarului dacă e negativ
SKIP:
```

c). Scăderea pe 64 de biţi: EDX|EAX - EBX|ECX
```
    SUB    EAX, ECX         ;se scad biţii de ordin inferior
    SBB    EDX, EBX         ;se scad biţii de ordin superior,
                            ;cu posibilitate de împrumut
```

2. Studiaţi următorul program în care se verifică dacă procesorul suportă tehnologia MMX. Se foloseşte instrucţiunea `CPUID` care returnează informaţii referitoare la procesor: instrucţiunea se execută folosind diferite valori ca intrare în registrul EAX. În cazul parametrului de intrare EAX=1, rezultatul returnat în EDX va conţine flaguri cu informaţii despre procesor. Bitul 23 dă informaţia referitoare la tehnologia MMX: dacă este 1, procesorul supportă MMX.

```
code    segment
        org     100h
        assume  cs:code, ds:code, ss:code, es:code

main:
        jmp     start

        mesaj db "Setul MMX $"
        mesaj_DA db "acceptat.", 13, 10, "$"
        mesaj_NU db "nu este acceptat.", 13, 10, "$"

start:
        ; afişează mesajul de început
        mov ax, seg mesaj
        mov ds, ax
        mov dx, offset mesaj
        mov ah, 09h
        int 21h

.586P   ; selectează set de instrucţiuni 586P

        push    ebx             ; salvează regiştrii
        push    ecx
        push    edx

        ; execută cpuid cu parametrul de intrare eax=1
        mov     eax, 1
        cpuid
        mov     eax,23   ; verific bitul 23
        bt      edx,eax  ; dacă bitul 23 din edx este 1, C=1

        pop     edx      ; reface regiştrii
        pop     ecx
        pop     ebx
.8086                   ; selectează set de instrucţiuni 8086
        jnc nu          ; dacă C=0, nu suportă MMX
        mov     dx, offset mesaj_DA    ; selectez DA
        jmp     afişare

nu:     mov     dx, offset mesaj_NU    ; selectez NU

afisare:
        mov     ah, 09h                ; afişez răspunsul
        int     21h
        mov     ax,4c00h
        int     21h                    ; terminare program
code    ends
        end     main
```

Modificaţi aplicaţia astfel încât să verifice accesul la numărul de serie al procesorului. Această informaţie este dată de bitul 18 din EDX returnat pentru intrare EAX=1, în acelaşi fel: dacă bitul respectiv este 1, avem acces la numărul de serie al procesorului. Testaţi valoarea bitului folosind altă secvenţă decât cea prezentată mai sus.

3. Considerăm următorul şir definit pe octet: 0,1,2,3,4,5,6,7. Scrieţi un program care să realizeze afişarea acestuia în ordinea dată de adresarea bit-reversed. Acest tip de adresare este întâlnit la procesoarele de semnal specializate şi se foloseşte în calculul transformatei Fourier rapide.

13. *Dezvoltarea aplicatiilor in limbaj C si asamblare*

13.1. Execuţia programelor pe 32 biţi

O modalitate de a dezvolta programe pe 32 biţi este prin folosirea Visual C++ în care se poate insera cod în limbaj de asamblare inline. Astfel, se beneficiază de toate avantajele folosirii unui limbaj de programare de nivel înalt. În plus, se pot consulta regiştrii pe 32 biţi, aşa cum era posibil în limbajul de nivel scăzut.

Termenul *inline* este un cuvânt cheie în limbajul C și este folosit în declararea funcțiilor. Atunci când în programul sursă compilatorul găsește o funcție declarată ca fiind „inline”, peste tot în programul principal unde va găsi apel al acelei funcţii, va înlocui cu corpul funcției respective. Expresia “inline assembly” se referă la un set de instrucțiuni în limbaj de asamblare, scrise ca funcții inline. ***Inline assembly*** este folosit în general ca o metodă de optimizare utilizată în dezvoltarea programelor.

În programele C/C++ se pot insera instrucțiuni în limbaj de asamblare folosind cuvântul cheie “asm”, precedat de „_” şi apoi între { } se vor scrie respectivele instrucţiuni în asamblare.

13.2. Etape în crearea unui program folosind asamblarea inline

Pas1. Se creează un proiect de tip Win32 Console Application (în exemplele prezentate s-a folosit Visual Studio Express Edition), în care suntem de acord cu toate opţiunile propuse şi selectăm *Finish*, aşa cum se poate observa şi din figura 13.1.

Exemplu de program simplu, tipic în C, care calculează suma a 2 numere preluate de la tastatură:

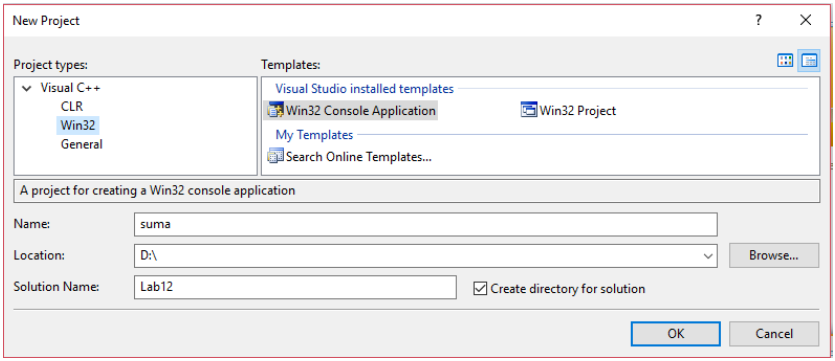


Fig 13.1. Crearea unui program în Visual Studio

Pas 2. Scriem programul de mai jos (în locul şablonului oferit automat de Microsoft Visual Studio C++) şi dăm apoi comanda *Build Solution* sau *F7*, apoi *Build suma*, *Compile*

```
#include "stdafx.h"
#include "stdio.h"
int a, b, sum;      // variabile globale
int main (void)
{
 printf ("Enter the first number: ");
 scanf ("%d", &a);  // preluarea primului numar

 printf ("Enter the second number: ");
 scanf ("%d", &b);  // preluarea celui de-al doilea numar
 // sum = a+b;      // calculeaza suma
 _asm {         MOV eax, a
                MOV ebx, b
                ADD eax, ebx
                MOV sum, eax
       }
 printf ("\n%d plus %d = %d\n", a, b, sum);
 return 0;
}
```

Pas 3. Executăm programul linie cu linie, folosind opţiunea *Step Into*:

Se va selecta *Start Debugging (F5)* şi apoi *Step Into (F11)*; în continuare, la opţiunile *Windows* (din *Debug*) vor apărea mult mai multe opţiuni. Vom parcurge întreg programul cu *F11 (Step Into)* apăsat, pentru fiecare linie din fişier, aşa cum se poate propune în figura 13.2:

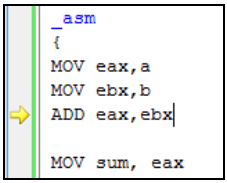


Fig 13.2. Depanare cu Visual Studio: programul propus înainte de execuţia operaţiei de adunare

Regiştrii arată ca în figura 13.3 (după execuţia MOV ebx,b); în exemplul prezentat, utilizatorul a apăsat 5, urmat de Enter şi apoi 11 urmat de Enter; aşa cum se poate urmări şi în codul sursă, valorile sunt preluate ca numere în zecimal.

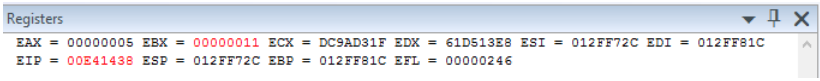


Fig 13.3. Depanare cu Visual Studio: vizualizare regiştri

Mai apăsăm încă o dată ***F11***, şi deci se va executa operaţia de adunare, care va avea ca rezultat coborârea cursorului astfel încât să indice că operaţia următoare este *MOV suma, eax* (aşa cum se prezintă în figura 13.4), iar valorile regiştrilor sunt cele din figura 13.5, unde se poate observa rezultatul în registrul EAX. Subliniem faptul că EFL este registrul de flaguri (Extended FLag).

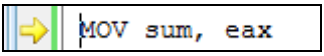


Fig 13.4. Depanare cu Visual Studio: modul cum se mută cursorul pe instrucţiunea curentă

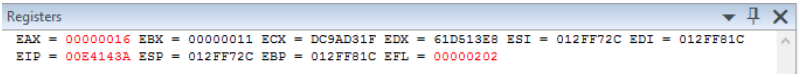


Fig 13.5. Depanare cu Visual Studio: modul cum se prezintă regiştrii după execuţia instrucţiunii *MOV sum, eax*

Pas 4. La sfârşit, când dorim să ne oprim cu depanarea, selectăm ***Stop Debugging.***

Dacă vom selecta opţiunea ***Disassembly*** (disponibilă în meniul *Debug-> Windows* după ce am dat cel puţin o dată pe *F11*, sau folosind *Alt+8*), vom vedea că inclusiv funcţiile din C (precum *printf* şi *scanf*) sunt transformate în codul echivalent în asamblare; important însă pentru noi să urmărim este faptul că tipul variabilelor folosite în program, adică *a,b* şi *sum* sunt precedate acum de construcţia *dword ptr* şi variabila apare între [ ], aşa cum arată figura 13.6.

```
_asm
{
MOV eax,a
00151432  mov         eax,dword ptr [a]
MOV ebx,b
00151435  mov         ebx,dword ptr [b]
ADD eax,ebx
00151438  add         eax,ebx

MOV sum, eax
0015143A  mov         dword ptr [sum],eax
```

Fig 13.6. Depanare cu Visual Studio: adaptarea la tipul variabilelor folosind *dword ptr*

13.3. Când se foloseste inline assembly?

În general, atunci când limbajul de nivel înalt nu are acces la anumite instrucţiuni sau facilităţi (dar care din limbaj de asamblare sunt uşor de obţinut)

1. În limbajul C există suport pentru **operații de deplasare (shiftare) pe biți,** dar nu există implementat suport pentru operații de rotație pe biți (deşi aceste operaţii există la nivelul procesorului).

2. **CPUID**: La nivelul procesoarelor moderne există o instrucțiune simplă, accesibilă doar din limbaj de asamblare, care oferă informații despre procesor; această instrucţiune este cpuid.

Operaţii pe biţi în C: Datele aflate în memoria RAM a sistemului sunt organizate ca o secvenţă de octeţi. Operatorii la nivel de bit sunt necesari atunci când dorim să prelucrăm biţii din interiorul structurii unui octet.

Limbajul C suportă 6 tipuri de operatori pe biţi („bitwise operators").
- **operatorul &** pentru realizarea operaţiei AND logic pe biţi
- **operatorul |** pentru realizarea operaţiei OR logic pe biţi
- **operatorul ^** pentru realizarea operaţiei XOR logic pe biţi
- **operatorul ~** pt realizarea operaţiei NOT logic pe biţi (complement faţă de 1)
- **operatorul <<** de deplasare spre stânga
- **operatorul >>** de deplasare spre dreapta

Exemplul 1. Deplasarea spre stânga a unui număr pe 32 biţi (citit de la tastatură în hexazecimal) cu un nr de poziţii citit de la tastatură (ca nr întreg între 0-31).

```
Introduceti un nr cu maxim 8 cifre hexa (introduceti doar cifrele lui hexa): 1234
Introduceti un nr < 32 ca fiind nr de pozitii cu cat doriti deplasare spre stanga: 8

Valoarea lui n este: 00001234h inainte de deplasarea spre stanga cu 8 pozitii.
Valoarea lui n este: 00123400h dupa deplasarea spre stanga cu 8 pozitii.
```

Fig 13.7. Fereastra execuţiei ex. 1

Program scris în C	Program în C cu inline asm
`/* Program in C - demonstrarea folosirii operatorului de deplasare spre stanga << in C.*/`	`/* Program in C cu asm inline - demonstrarea folosirii instructiunii de deplasare spre stanga SHL in ASM.*/`
`#include <stdafx.h>`	`#include <stdafx.h>`
`#include <stdio.h>`	`#include <stdio.h>`
`int main()`	`int main()`
`{`	`{`
`unsigned int n, poz;`	`unsigned int n, poz;`

```
  printf("Introduceti un nr cu maxim 8 cifre hexa (introduceti doar cifrele lui hexa): ");
  scanf("%X", &n);
  printf("Introduceti un nr < 32 ca fiind nr de pozitii cu cat doriti deplasare spre stanga: ");
  scanf("%d", &poz);
printf("\nValoarea lui n este: %08Xh inainte de deplasarea spre stanga cu %d pozitii.",n,poz);
    /*deplasarea lui n cu poz pozitii spre stanga*/
    n = (n<<poz); /*operatia*/

printf("\nValoarea lui n este: %08Xh dupa deplasarea spre stanga cu %d pozitii. \n\n",n,poz);
    return 0;
}
```

```
    printf("Introduceti un nr cu maxim 8 cifre hexa (introduceti doar cifrele lui hexa): ");
    scanf("%X", &n);
    printf("Introduceti un nr < 32 ca fiind nr de pozitii cu cat doriti deplasare spre stanga: ");
    scanf("%d", &poz);
   printf("\nValoarea lui n este: %08Xh inainte de deplasarea spre stanga cu %d pozitii.",n,poz);
    /*deplasarea lui n cu poz pozitii spre stanga*/
    _asm          /*operatia*/
        {       MOV EAX, n;
                MOV ECX, poz;
                SHL EAX, CL
                MOV n, EAX;
        }
printf("\nValoarea lui n este: %08Xh dupa deplasarea spre stanga cu %d pozitii. \n\n",n,poz);
    return 0;
}
```

Exerciţii propuse:

1. Realizaţi deplasarea spre dreapta (echivalent cu instrucţiunea ***shr***).
2. Propuneţi o metodă/ un algoritm de a implementa în C acelaşi efect ca şi cel obţinut prin execuţia instrucţiunii ***sar***.

Exemplul 2. Setarea/resetarea unui anumit bit specificat prin poziţie (poziţia bitului se preia de la tastatură)

```
Introduceti un nr cu maxim 8 cifre hexa (introduceti doar cifrele lui hexa): 12345678
Introduceti un nr < 32 ca fiind pozitia de pe care vreti sa setati bitul: 7

Valoarea lui n este: 12345678h inainte de setarea bitului de pe pozitia 7.
Valoarea lui n este: 123456F8h dupa setarea bitului de pe pozitia 7.

Introduceti un nr < 32 ca fiind pozitia de pe care vreti sa resetati bitul: 9

Valoarea lui n este: 123456F8h inainte de resetarea bitului de pe pozitia 9.
Valoarea lui n este: 123454F8h dupa resetarea bitului de pe pozitia 9.
```

Fig 13.8. Fereastra execuţiei ex. 2

Program scris în C

```
/* C Program to set/clear some bits.*/
#include <stdafx.h>
#include <stdio.h>
int main()
{     unsigned int n, poz;
printf("Introduceti un nr cu maxim 8 cifre hexa (introduceti doar cifrele lui hexa): ");
scanf("%X", &n);
printf("Introduceti un nr < 32 ca fiind pozitia de pe care vreti sa setati bitul: ");
scanf("%d", &poz);
printf("\nValoarea lui n este: %08Xh inainte de setarea bitului de pe pozitia %d.",n,poz);
   /*seteaza bitul de pe pozitia poz*/
    n |= (1 << poz);

// sau : newN = (1 << poz) | n;

printf("\nValoarea lui n este: %08Xh dupa setarea bitului de pe pozitia %d.\n\n",n,poz);
printf("Introduceti un nr < 32 ca fiind pozitia de pe care vreti sa resetati bitul: ");
scanf("%d", &poz);
printf("\nValoarea lui n este: %08Xh inainte de resetarea bitului de pe pozitia %d.",n,poz);
   /*reseteaza bitul de pe pozitia poz*/
    n &= ~(1 << poz);

printf("\nValoarea lui n este: %08Xh dupa resetarea bitului de pe pozitia %d.\n\n",n,poz);
    return 0;
}
```

Program în C cu inline asm

```
/* C Program to set/clear some bits.*/
#include <stdafx.h>
#include <stdio.h>
int main()
{     unsigned int n, poz;
printf("Introduceti un nr cu maxim 8 cifre hexa (introduceti doar cifrele lui hexa): ");
scanf("%X", &n);
printf("Introduceti un nr < 32 ca fiind pozitia de pe care vreti sa setati bitul: ");
scanf("%d", &poz);
printf("\nValoarea lui n este: %08Xh inainte de setarea bitului de pe pozitia %d.",n,poz);
   /*seteaza bitul de pe pozitia poz*/
    _asm {      MOV EAX, n
                MOV EBX, 1
                MOV ECX, poz
                SHL EBX,CL
                OR EAX,EBX
                MOV n,EAX
          }
printf("\nValoarea lui n este: %08Xh dupa setarea bitului de pe pozitia %d.\n\n",n,poz);
printf("Introduceti un nr < 32 ca fiind pozitia de pe care vreti sa resetati bitul: ");
scanf("%d", &poz);
printf("\nValoarea lui n este: %08Xh inainte de resetarea bitului de pe pozitia %d.",n,poz);
   /*reseteaza bitul de pe pozitia poz*/
_asm    {       MOV EAX, n
                MOV EBX, 1
                MOV ECX, poz
                SHL EBX,CL
                NOT EBX
                AND EAX,EBX
                MOV n,EAX
        }
 printf("\nValoarea lui n este: %08Xh dupa resetarea bitului de pe pozitia %d.\n\n",n,poz);
    return 0;
}
```

Exerciţii propuse:

1. Modificaţi programele astfel încât să se preia de la tastatură poziţia unui bit şi să se seteze/reseteze un număr de 2/3/... biţi pornind de la acea poziţie înspre stânga / dreapta.
2. Să se preia de la tastatură poziţia şi un număr de biţi, de ex. se preia 3 şi 4: de la bitul de pe poziţia 3 vor fi afectaţi 4 biţi.
3. Repetaţi exerciţiile anterioare, dar bitul va fi modificat – adică inversat : newN = n ^ (1 << poz);
4. Sa transforme un caracter din minuscula in majuscula sau invers (se modifica doar bitul 6) – caracterele sunt litere !!! (Bit masking)

Exemplul 3. Program care să numere numărul de biţi de 1 dintr-un număr:

```
Introduceti un nr cu maxim 8 cifre hexa (introduceti doar cifrele lui hexa): 12345

Numarul total de biti de 1 este: 7
```

Fig 13.9. Fereastra execuţiei ex. 3

Program scris în C

```
/*C program to count number of 1's in a number */
#include <stdafx.h>
#include <stdio.h>
int main()
{   unsigned int n;
    unsigned                    char totalBits=sizeof(n)*8;
    int i,count=0;
printf("Introduceti un nr cu maxim 8 cifre hexa (introduceti doar cifrele lui hexa): ");
scanf("%X", &n);
for(i=0;i< totalBits;i++)
    {
        if( n & (1<< i) )
            count++;     //     se parcurge de 32 ori: totalBits=32

    }

    printf("\nNumarul total de biti de 1 este: %d\n",count);
     return 0;
}
```

Program în C cu inline asm

```
/*C program to count number of 1's in a number */
#include <stdafx.h>
#include <stdio.h>
int main()
{   unsigned int n;
    unsigned                    char totalBits=sizeof(n)*8;
    int i,count=0;
printf("Introduceti un nr cu maxim 8 cifre hexa (introduceti doar cifrele lui hexa): ");
scanf("%X", &n);
_asm
    {
      MOV EAX, n
      POPCNT EBX, EAX
      MOV count, EBX
    }
    printf("\nNumarul total de biti de 1 este: %d\n",count);
     return 0;
}
```

Exemplul 4. Inversarea biţilor unui număr.

```
Introduceti un nr cu maxim 8 cifre hexa (introduceti doar cifrele lui hexa): 1234

Numarul inversat este: 2C480000
```

Fig 13.10. Fereastra execuţiei ex. 4

Program scris in C

```
/*C program to reverse bits in
a number */
#include <stdafx.h>
#include <stdio.h>

    int main()
    {   unsigned int n;
        unsigned           char
totalBits = sizeof(n) * 8;
        unsigned           int
reversedNum = 0, i, temp;
printf("Introduceti un nr cu
maxim     8     cifre     hexa
(introduceti doar cifrele lui
hexa): ");
scanf("%X", &n);
for (i = 0; i < totalBits;
i++)
        {
        temp = (n & (1 << i));
         if(temp)
         reversedNum |= (1 <<
((totalBits - 1) - i));
        }




    printf("\nNumarul inversat
este: %X\n", reversedNum);
    return 0;
}
```

Program in C cu inline asm

```
/*C program to reverse bits in
a number */
#include <stdafx.h>
#include <stdio.h>

    int main()
    {   unsigned int n;
        unsigned           char
totalBits = sizeof(n) * 8;
        unsigned           int
reversedNum = 0, i, temp;
printf("Introduceti un nr cu
maxim     8     cifre     hexa
(introduceti doar cifrele lui
hexa): ");
scanf("%X", &n);
_asm
{       MOV ECX,32
        MOV EAX,n
        MOV EBX,0
eti:
        ROL EAX,1
        RCR EBX,1
        loop eti
        MOV reversedNum, EBX
}
    printf("\nNumarul inversat
este: %X\n", reversedNum);
    return 0;
}
```

Exemplul 5. Interschimbarea ultimelor 2 cifre (nibble) ale unui număr.

```
Introduceti un nr cu maxim 8 cifre hexa (introduceti doar cifrele lui hexa): 123456

Numarul introdus de dvs : 00123456
Numarul dupa interschimbarea ultimelor 2 tetrade: 00123465
```

Fig 13.11. Fereastra execuţiei ex. 5

Program scris in C

```
/*C program to swap two nibbles of a given byte*/
#include <stdafx.h>
#include <stdio.h>
int main()
{    unsigned int n;
    unsigned int num;
  printf("Introduceti un nr cu maxim 8 cifre hexa (introduceti doar cifrele lui hexa): ");
scanf("%X", &n);


num= ( (n & 0xFFFFFF00)<<0 |(n & 0x0F)<<4 | (n & 0xF0)>>4 );



printf("\nNumarul introdus de dvs : %08X \nNumarul dupa interschimbarea ultimelor 2 tetrade: %08X \n \n ",n,num);
    return 0;
}
```

Program in C cu inline asm

```
/*C program to swap two nibbles of a given byte*/
#include <stdafx.h>
#include <stdio.h>
int main()
{    unsigned int n;
     unsigned int num;
  printf("Introduceti un nr cu maxim 8 cifre hexa (introduceti doar cifrele lui hexa): ");
scanf("%X", &n);
 _asm
{
        MOV EAX, n
        ROL AL,4
        MOV num, EAX
}


 printf("\nNumarul introdus de dvs : %08X \nNumarul dupa interschimbarea ultimelor 2 tetrade: %08X \n \n ",n,num);
    return 0;
}
```

Exercitii propuse. Analizaţi ce se întâmplă dacă se înlocuieşte cu:
(**(n & 0xFFFF0000) <<0 |** (n<<8) &0xFF00) |((n>>8) &0x00FF).

14. Exemple şi Aplicaţii

1. Precizaţi care sunt valorile biţilor de Carry, Zero, Sign şi Overflow după executarea următoarelor adunări de către un microprocesor pe 8 biţi:

	Rezultat	C	Z	S	O
02 + 02	04	0	0	0	0
02 + 7D	7F	0	0	0	0
02 + 7E	80	0	0	1	1
02 + 7F	81	0	0	1	1
02 + FD	FF	0	0	1	0
04 + FE	02	1	0	0	0
04 + 7F	83	0	0	1	1
80 + 80	00	1	1	0	1

2. Deterinaţi valorile binare pe 8 biţi şi hexazecimale echivalente următoarelor valori zecimale cu semn: +2, -126, +16.

		Binar	Hexa
(a)	+2	00000010	02H
(b)	-126	10000010	82H
(c)	+16	00010000	10H

3. Dacă numărul binar pe 8 biţi 05h este adunat cu fiecare din următoarele numere, daţi valorile biţilor Sign şi Overflow după fiecare adunare:

(a) 02H (b) FAH (c) FBH

	Suma	Sign	Overflow
(a) 02H	07H	0	0
(b) FAH	FFH	1	0
(c) FBH	00H	0	0

4. Indicaţi valorile posibile ale unui număr pe 8 biţi N pentru care după adunarea cu 42h:

(a) Carry este 1
(b) Sign este 0
(c) Sign este 1 şi Overflow este 0

```
(a) C = 1
42H + N > FFh
```

de exemplu:
```
N > FFh - 42h
N > BDh
BEh =< N =< FFh
```

(b) S = 0

N pozitiv:

42h + N =< 7Fh
N =< 7Fh – 42h
N =< 3Dh
0 =< N =< 3Dh

N negativ:

42h + N => 100h
N => 100h – 42h
N => BEh
BEh =< N =< FFh

(c) S = 1

De la punctul b rezultă că: 3Eh =< N =< 7Fh

80h =< N =< BDh

pentru O = 0

întregul domeniu negativ: 80h =< N =< FFh

N pozitiv:

42h + N =< 7Fh
N =< 7Fh – 42h
N =< 3Dh
0 =< N =< 3Dh

Pentru a fi îndeplinite ambele condiţii este necesar ca cele două domenii să se suprapună, adică: 80h =< N =< 3Dh

5. Numărul pe 8 biţi 5Ah este adunat cu un număr pe 8 biţi N. Daţi domeniul în care poate N să ia valori astfel încât să se îndeplinească următoarele condiţii imediat după executarea adunării de către un microprocesor pe 8 biţi:

a. Bitul C să fie 0

5Ah + N <= FFh
N<= FFh - 5Ah
N<= A5h
0<= N<= A5h

b. Bitul Sign să fie 1

pentru N pozitiv, adică 0 <= N <= 7Fh:

5Ah + N > 7Fh
N > 7Fh - 5Ah
N > 25h
0 <= N <= 24h

Pentru N negativ, adică 80h <= N <= FFh:

5Ah + N <= FFh (dacă suma depăşeşte FF va fi pozitivă)
N <= FFh - 5Ah
N <= A5h
80h <= N <= A5h

Domeniul complet este: `0 <= N <= 24h, 80h <= N <= A5h`

c. Bitul Zero să fie 1
d. Bitul Overflow să fie 0
e. Biţii Carry şi Sign să fie 1
f. Biţii Overflow şi Sign să fie 0

6. Două numere pe 8 biţi aflate în DS la offset 1000h şi 1001h sunt adunate. Asupra rezultatului pe 8 biţi se realizează o operaţie logică SAU cu numărul pe 8 biţi stocat la adresa 1002h, iar rezultatul final este stocat la adresa 1100h. Scrieţi secvenţa programului în limbaj de asamblare.

```
        org     100h
        mov     ah,[1000h]
        add     ah,[1001h]
        or      ah,[1002h]
        mov     [1100h],ah
        int     20h
```

7. Opt numere pe 16 biţi stocate la adrese succesive de memorie începând cu offset 1200h sunt adunate. În continuare se realizează un SI cu numărul 8888h, iar rezultatul final este stocat la adresele de offset 1400h şi 1401h. Scrieţi secvenţa programului în limbaj de asamblare.

```
        org     100h
        mov     ax,0
        mov     si,0
loop    add     ax,1200h[si]
        add     si,2
        cmp     si,10h
        jne     loop
        and     ax,8888h
        mov     [1400h],ax
        int     20h
```

8. Operaţii pe biţi

- Următoarea secvenţă permite obţinerea aceluiaşi efect ca şi cel rezultat în urma instrucţiunii CWD

```
        and ax, 8000h       ;rezultatul va fi 0 dacă bitul de
                            ;semn este 0
        jz et
        mov dx, 0FFFFh      ;numărul este negativ
et:    mov dx,0             ;numărul este pozitiv
```

- Se dă o variabilă *var*, de tip word. Se cere să se numere biţii de 1.

```
        mov ax, var
        mov dx, 0           ;in dx vom număra biţii
        mov cx, 16          ;nr. de iteratii effectuate
iar:   shl ax, 1            ;CF=MSB din AX
```

```
       jnc et
        inc dx
et:    dec cx
       jnz bucla
```

9. Se doreşte tipărirea pe ecran a mesajului *'microprocesoare'*. Scrieţi două versiuni ale programului: una folosind INT 21H pentru a afişa caractere individuale şi una folosind INT 21H pentru a afişa un şir de caractere.

```
;Afişare caracter cu caracter
        org     100h
        mov     di,offset(string)
        mov     ah,02h
lop:    mov     dl,[di]
        cmp     dl,24h
        je      end
        int     21h
        inc     di
        jmp     lop
end     int     20h
string  db      'microprocesoare$'

;afisare sir
        org     100h
        mov     di,offset(string)
        mov     ah,09h
        mov     dx,di
        int     21h
        int     20h
string  db      'microprocesoare$'
```

10. Se introduc de la tastatură 8 caractere. Fiecare caracter trebuie verificat dacă este un cod ASCII corespunzător unei cifre zecimale (30h-39h), iar în final se va afişa pe ecran numărul de caractere zecimale introduse.

```
org     100h
        mov     ch,0
        mov     si,0
        mov     ah,01h
lop:    int     21h
        cmp     al,30h
        jb      notnum
        cmp     al,39h
        ja      notnum
        inc     ch
notnum: inc     si
        cmp     si,8h
        jne     lop
        mov     ah,2h
        mov     dl,0dh
        int     21h
        mov     dl,0ah
        int     21h
```

```
        add     ch,30h
        mov     dl,ch
        int     21h
        int     20h
```

11. Scrieţi o secvenţă de program în limbaj de asamblare în care :
    a. Utilizatorul este invitat să îşi introducă numele;
    b. Se preiau caracterele introduse de la tastatură ;
    c. Preluarea caracterelor se termină cu apăsarea tastei Enter;
    d. Se numără apariţia vocalei “e” şi se afişează pe ecran.

```
        org     100h
        mov     ch,0
        mov     ah,01h
lop:    int     21h
        cmp     al,0dh
        je      end1
        cmp     al,45h
        jne     note
        inc     ch
note:   jmp     lop
end1:   mov     ah,2h
        mov     dl,0dh
        int     21h
        mov     dl,0ah
        int     21h
        add     ch,30h
        mov     dl,ch
        int     21h
        int     20h
```

Dacă dorim să numărăm toate vocalele:

```
        org     100h
        mov     ch,0
        mov     ah,01h
lop:    int     21h
        cmp     al,0dh
        je      end
        cmp     al,41h
        jne     nota
        inc     ch
nota:   cmp     al,45h
        jne     note
        inc     ch
note:   cmp     al,49h
        jne     noti
        inc     ch
noti:   cmp     al,4fh
        jne     noto
        inc     ch
noto:   cmp     al,55h
```

```
        jne     notu
        inc     ch
notu:   jmp     lop
end1:   mov     ah,2h
        mov     dl,0dh
        int     21h
        mov     dl,0ah
        int     21h
        add     ch,30h
        mov     dl,ch
        int     21h
        int     20h
```

12. Scrieţi un program în limbaj de asamblare care vor încărca 4 numere pe câte 16 biţi aflate la adresele de la offset 0200h la 0207h în regiştrii AX, BX, CX, DX. Scădeţi BX din AX şi DX din CX. Adunaţi cele două rezultate şi transferaţi rezultatul final pe 16 biţi la adresele de offset 0220h şi 0221h.

```
        org     100h
        mov     ax,[0200h]
        mov     bx,[0202h]
        mov     cx,[0204h]
        mov     dx,[0206h]
        sub     ax,bx
        sub     cx,dx
        add     ax,cx
        mov     [0220h],ax
        int     20h
```

13. Să se citească de la tastatură un şir a cărui lungime maximă este de 30 de caractere, să se inverseze şi apoi să se afişeze rezultatul.

```
.model small
.stack 100h
.data
        şir db 30 dup (?)
.code
mov ax,@data
mov ds,ax
        mov dx, offset şir              ;se preia adresa şir
        mov si, dx ;
        mov byte ptr [si], 30           ;depune în primul octet
                                        ;lungimea maxima a
                                        ;şirului de citit
        mov ah, 0Ah                     ;citeşte şir de la un
                                        ;dispozitiv de intrare
        int 21h ;
        xor cx, cx ; CX=0
        mov cl, [si+1]
        mov bx, cx                      ;salveaza în BX lung. şir
        shr cx, 1                       ;se imparte CX la 2
        jcxz gata
        add si, 2                       ;se preia în SI adresa
```

```
                                    ;şir si o copiaza în DI
        mov di, si
        add si, bx                  ;se preia în SI adresa
                                   ;ultimului caracter din şir
        dec si
bucla:  mov al, [si]                ;se interschimba doua
                                    ;elemente din şir
        xchg al, [di]
        mov [si], al
        inc di                      ;se trece la următoarele
                                    ;elem. ce urmeaza a fi
                                    ;interschimbate
        dec si
        loop bucla                  ;bucla se repeta până CX=0
gata:   mov si, dx                  ;se preia în SI adresa şir
        mov byte ptr [si],0Ah       ;în primii 2 octeţi se
                                    ;depune CR
        mov byte ptr [si+1], 0Dh    ;si LF
        add si, bx
        add si, 2
        mov byte ptr [si], 24h      ;se depune "$" la
                                    ;sfîrsitul şirului
        mov ah, 09h                 ;se afiseaza şirul
        int 21h
        mov ax, 4c00h               ;terminare program
        int 21h
end
;_____________________________________________________________;
```

INT 21h

Funcţia 0Ah – citeşte de la intrarea standard un şir de caractere până la apăsarea tastei Enter

Intrare:

AH = 0Ahh

DS:DX = adresa de început a zonei de citire

Returnează:

Nimic

Obs: Zona de citire este o zonă conţinuă de memorie, cu următoarea structură Nr_Max, Nr_Citite, Zona_Citire, unde:

- *Nr_Max* este primul octet (offset 0) care conţine lungimea maximă a şirului care se citeşte (inclusiv Enter). Octetul trebuie completat înainte de efectuarea citirii cu o valoare între 0 si 255;
- *Nr_Citite* este al doilea octet (offset 1), neiniţalizat înaintea citirii;
- *Zona_Citire* este zona de memorie rezervată pentru caracterele citite (începe de la offset 2). Lungimea sa este egală cu Nr_Max

Se citesc caracterele de la tastatură până la întâlnirea caracterului Enter. Nu se permite introducerea de la tastatură a mai mult de Nr_Max caractere.

14. Următorul program citeşte un şir de maxim 20 caractere de la tastatură şi afişează 24 de linii a câte 80 de caractere pe linie, conţinând şirul citit *(Umplere ecran cu un şir de caractere citit de la tastatură):*

```
    .model small
    .stack 100h
    .data
        şir db 20 dup (?)
        a   db 0Dh,0Ah,'$'
        b dw ?
    .code
        mov ax,@data
        mov ds,ax
        mov dx, offset şir            ;Preia adresa şir
        mov si, dx                    ;Depune în primul octet
                                      ;lungimea maxima a şirului
        mov byte ptr [si],20
        mov ah,0Ah                    ;Citeşte şir cu funcţia
        int 21h                       ;0Ah a întreruperii 21h
        xor bx, bx                    ;Sterge BX
        mov bl,[si+1]                 ;Preia lungime şir
        mov si, dx                    ;Preia în SI adresa şir
        mov byte ptr[si+bx+2],'$'
        mov       al,80               ;mod text 25x80
        cbw
        div bl                        ;De cite ori incape şirul
                                      ;pe o linie
        xor ah,ah                     ;catul se pastraza in AL,
                                    ;restul(ah) nu ne intereseaza
        mov b, ax
        mov cx,25
et1:    push cx
        mov cx,b
eet:    mov dx, offset şir
        add dx, 2
        mov ah, 09h                   ;Afiseaza şirul rezultat
        int 21h
        loop eet
        mov dx,offset a               ;trece la linia următoare
        mov ah,09h
        int 21h
        pop cx
        loop et1
        mov ax, 4c00h                 ;terminare program
        int 21h
end
```

20. Să se scrie un program care să permită căutarea primului caracter ‘a’ dintr-un şir ale cărui elemente sunt definite pe octet. In cazul în care şirul nu conţine nici un astfel de caracter în DX va fi încărcată valoarea 0 iar în caz contrar valoarea poziţiei din şir a primului caracter găsit.

```
.model small
.stack 100h
.data
şir DB 'Sisteme cu microprocesoare!'
.code
mov ax,@data
mov ds,ax
        mov cx,size şir         ;depune în CX lungimea şirului
        mov si, offset şir-1
        mov al,'a'              ;initializeaza AL  cu 'a'
        mov dx,0
iar:    inc si                  ;bucla de cautare
        inc dx                  ;DX conţine indexul elem. gasit
        cmp al,[si]
        loopne iar              ;cautarea   continua   până   la
                                ;gaşirea primului element 'a' din
                                ;şir sau până cand CX=0
        je gata
        mov dx,0                ;dacă nu s-a gasit nici un
                                ;element DX=0
gata:    mov ax, 4c00h          ;terminare program
        int 21h
end
```

Temă: Să se scrie un program care să numere câte caractere de 'a' există într-un şir dat.

15. Următorul program verifică dacă placa video suportă **Modul 13h** (tabelul cu modurile video este prezentat în ANEXA 2)

```
.model small
.stack
.data
        No db 'Modul 13h nu este suportat.','$'
        Yes db 'Modul 13h este suportat.','$'

.code
        mov ax,@data
        mov ds,ax
        mov ax,1A00h            ;se cer informatii despre VGA
        int 10h                 ;Get Display Combination Code
        cmp al,1Ah              ;este VGA sau MCGA suportat?
        je ok
        mov ah,9                ;modul 13h nu este suportat
        mov dx,offset No
        int 21h
        jmp ieşire
ok:     mov ah,9                ;modul 13h este suportat
        mov dx,offset Yes
        int 21h
```

```
ieşire: mov ax,4C00h
        int 21h
end
;_____________________________________________________________;
```

INT 10h

Funcţia 1Ah - VIDEO DISPLAY COMBINATION (PS,VGA/MCGA)

Intrare:

AH = 1Ah

AL = 00h

Returnează:

AL = 1Ah dacă funcţia este suportată

22. Aplicaţia următoare permite desenarea unor pixeli în Modul Video 13 folosind serviciile BIOS ale întreruperii INT 10h

```
.model small
.stack
.data
VideoMode db ?
.code
        mov ax,@data
        mov ds,ax
        mov ah,0Fh              ;se salveaza modul curent
        int 10h
        mov VideoMode,al
        mov ax,13               ;se seteaza modul 13
        int 10h
        mov ah,0Ch              ;deseneaza un pixel la punctul de
                                ;coordonate (100,160)
        mov al,4                ;seteaza culoarea rosu
        mov cx,160              ;x coloana
        mov dx,100              ;y linia
        int 10h
        mov al,2                ;deseneaza o linie între:
                                ;(1,80) -> (320,80)
        mov cx,319              ;x (coloana)
        mov dx,80               ;y (linia)
et1:    int 10h
        loop et1
        xor ax,ax               ;asteapta o tasta
        int 16h
        mov al,VideoMode        ;seteaza modul video initial
        xor ah,ah
        int 10h
        mov ax,4C00h
        int 21h
end
```

Temă: Plecând de la exemplul anterior desenaţi un dreptunghi.

16. Studiaţi aplicaţia de mai jos, care propune o metodă mai rapidă de desenare a pixelilor direct în memoria video.

```
.model small
.stack
.data
.code
        mov ax,@data
        mov ds,ax
        mov ah,0
        mov al,13h
        int 10h
        mov ax,0a000h          ;segmentul VGA este 0A000h
        mov es,ax
                ;deseneaza in memoria video cu "rep stosb"
        mov dx,6
        mov di,16010           ;10+50*320
et1:    mov cx,20
        mov al,1
        rep stosb
        add di,30
        dec dx
        jnz et1
                ;desenează în memoria video cu "mov" (mai rapid)
        mov dx,6
        mov di,32020           ;20+100*320
                               ;offsetul pixelului (x,y) (de pe
                               ;linia y, coloana x) e x+(y*320)
et3:    mov cx,20              ;punctul (0,0)- coltul stanga sus
et2:    mov byte ptr es:[di],4
        inc di
        loop et2
        add di,30
        dec dx
        jnz et3

        mov ah,0
        int16h
        mov ah,0
        mov al,3
        int 10h
        mov ax,4C00h           ;terminare program
        int 21h
end
```

17. Analizaţi următoarea aplicaţie care permite generarea unui **fişier text** şi scrierea unui anumit număr de octeţi din memoria RAM în fişierul creat. Se folosesc funcţiile 3Ch si 40h ale întreruperii DOS 21h.

```
.model small
.stack
.data
    text        db 'P','r','o','c','e','s','o','a','r','e'
```

```
    Cale_Fişier db     'd:\fişier.txt',0
    mes_er      db     'A aparut o eroare! $'
    mes_ok      db     'Fişierul a fost creat! $'


.code
    mov ax,@data
    mov ds,ax
    mov ah,3ch          ;creare fişier, protejat la scriere
    mov dx,offset Cale_Fişier  ;adresa de început a şirului de
                        ;caractere ce conţine numele fişierului
    mov cx,1
    int 21h
    jc  er                      ;CF=1 – fişierul nu a fost creat
    push ax                     ;CF=0 – fişier creat cu success,
                                ;AX - identificatorul fişierului
    pop bx
    mov ah,40h                  ;scriere in fişier
    mov dx,offset text          ;adresa început de unde se preiau
                                ;octeţii pt scrierea lor infişier
    mov cx,10                   ;numărul de octeţi care se doresc
                                ;scrisi in fişier
        int     21h
        jc      er
        mov     dx,offset mes_ok
        jmp     ok
    er: mov     dx,offset mes_er
    ok: mov     ah,9
        int     21h
        mov     ax,4c00h
        int     21h
end
;_____________________________________________________________;
```

Int 21h
Funcţia 3Ch – Creare fişier

Crează un fişier. Dacă fişierul există deja, acesta este deschis şi lungimea îi este trunchiată la zero.

Intrare :

AH = 3Ch
CX = atribute de creare ale noului fişier: 0 = normal
1 = protejat la scriere,
2 = ascuns,
3 = fişier de sistem
DS:DX = adresa de început a şirului de caractere care conţine numele fişierului. Acesta trebuie să fie un şir ASCIIZ.

Returnează:

CF=0 – fişier creat cu succes; AX = identificatorul fişierului
CF=1 – fişierul nu a fost creat; AX=conţine un cod de eroare

Coduri de eroare: 3- Calea către fişier este invalidă
4- Nu există identificator de fişier disponibil
5- Nu se permite accesul la fişier

Funcţia 40h – Scriere în fişier

Scrie un anumit număr de octeţi dintr-o zonă de memorie într-un fişier specificat prin identificatorul său.

Intrare:

AH = 40h
BX = identificatorul fişierului în care se va efectua scrierea
CX = numărul de octeţi care se doresc scrişi în fişier
DS:DX = adresa de început a zonei de memorie din care se vor prelua octeţi în vederea scrierii lor în fişier

Returnează:

AX = numărul de octeţi scrişi în fişier sau cod de eroare dacă CF=1
Coduri de eroare: 5- Nu se permite accesul la fişier
6- Identificator de fişier invalid

Temă: Realizaţi o aplicaţie care să permită deschiderea unui fişier, afişarea pe ecran a conţinutului acestuia şi apoi închiderea sa (vezi funcţiile 3Dh, 3Eh si 3Fh ale întreruperii int 21h).

18. Dezvoltaţi un program care implementează **creionul din Paint**:

```
.model small
.stack 100h
.data
.code
       mov ax,@data
       mov ds,ax
       mov ah,0h       ;setează modul video
       mov al,0dh      ;modul grafic, 16 culori, dimensiune
                       ;40x25, rezoluţie 320x200
       int 10h
       mov al,1        ;setează prima culoare
repeat: push ax        ;salvează conţinutul registrului AX ca să
                       ;nu se modifice in rutina de întrerupere
       mov ax,3        ;returnează poziţia şi starea butoanelor
                       ;mouse-ului
       int 33h
       pop ax          ;refacere AX
       mov bh,bl       ;salvează starea butoanelor
       and bl,1        ;verifică dacă s-a apăsat butonul stâng
       jz next
       inc al          ;schimbă culoarea
next:  mov bl,bh       ;reface starea butoanelor
```

```
        and bl,2        ;verifică dacă s-a apăsat butonul drept,
                        ;în caz afirmativ se iese din program
        jnz gata
        mov ah,0ch   ;aprinde un pixel la coordonatele mouseului
        mov bh,0
        int 10h
        jmp repeat
gata:   mov ah,0h
        mov al,3h       ;reface modul text, 80x25
        int 10h
        mov ah,4ch
        int 21h
end
;_____________________________________________________________________;
```

INT 10h
Funcţia 0Ch – schimbă culoarea unui singur pixel

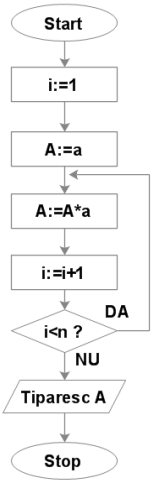
Intrare:
AH = 0Ch
AL = culoare pixel
BH =număr pagină
CX = coloana
DX = linie

INT 33h (întrerupere servicii mouse)/ AX=0003h
Intrare:
AX=0003h - returnează poziţia şi starea butoanelor mouse-ului
Returnează:
BX=1 – dacă s-a apasat butonul stâng
BX=2 – dacă s-a apasat butonul drept
BX=3 – dacă ambele butoane au fost apasate
CX = coloana x
DX = linie y

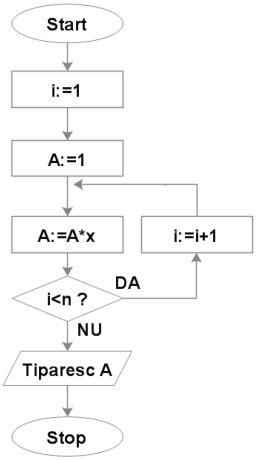
19. Scrieţi un program care să calculeze a^n şi să afişeze rezultatul, a fiind un număr dat. Urmăriţi organigrama corespunzătoare.
Indicaţie: Se notează a^n=A şi se formează o buclă în care într-un prim pas A se iniţializează cu a, iar apoi se calculează iterativ A= A*a (se va considera a=2). Rezolvaţi cerinţa pentru cazurile particulare:
a) n=3: variabila A va fi stocată în registrul AL, valoarea a se va stoca în registrul BL (pentru a realiza înmultirea), iar valoarea i va fi registrul DI.
b) pentru n=7 ce modificări trebuie aduse programului de la punctul a) ?
c) dar pentru cazul n=8? Se consideră numere fără semn.

```
Rezolvare:
a).model small
  .stack 200h
  .data
        a db 2          ; definirea variabilelor în segm de date
```

```
        n db 3
.code
main label
        mov ax,@data
        mov ds,ax       ; se iniţializează ds la începutul
                        ;segmentului de date
```



Organigrama problemei 19



Organigrama problemei 20

```
        mov di,1        ; di=1 este valoarea corespunzătoare pt i
        mov si, offset a ; si=adresa variabilei a
        mov al, [si] ; al=2(conţinutul de la adresa dată de si)
        mov bl, [si] ; bl=2

eti:    mul bl          ; ax=al*bl adică ax=4
        inc di          ; se incrementează contorul i
        cmp di,[si+1]   ; se verifică dacă i este egal cu n
                        ;(am terminat?)
        jb eti          ; dacă i<n se execută salt la eticheta
                        ; eti, deci se reia bucla pt următoarea
                        ; valoare i
afisare:                ; dacă am terminat (adică n=i)
        add al,30h      ; se adună la registrul al valoarea 30h
                        ; pt a obţine val ASCII a nr de afişat
        mov ah,0eh      ; scrie caracter în mod teletype
        int 10h
iesire:
        mov ax,4c00h    ; ieşire din program
        int 21h
end main
```

b) Modificarea programului se referă la modificarea funcţiei de afişare deoarece în cazul n=7 vom obţine 2^7=128, deci un număr cu 3 zecimale. Pentru afişare se va folosi funcţia “conversie Binar – ASCII” din cadrul Lucrării 10 (pagina 90).

```
.model small
.stack 200h
.data
        a db 2          ; definire variabile în segm de date
        n db 7
        ASCII db 3 dup(?) ; şir necesar pt depunerea
                        ; cifrelor numărului rezultat din
                        ; conversia binar -> ASCII
.code
convBinAscii proc near  ; procedura care depune în segm de date
;începand de la adresa ASCII cele 3 caract ASCII ale nr
;cuprins între 0 si 255
xor ah,ah               ; ah=00
mov cl,10               ; cl=10
div cl                  ; după div: ah=restul, al=câtul
add ah,30h              ; se adună 30h pt afişarea conţ. reg ah
mov [ASCII+2],ah        ; unităţile se depun la adresa ASCII+2
xor ah,ah               ; ah=00
div cl                  ; după div: ah=restul, al=câtul
add ah,30h
mov[ASCII+1],ah         ; zecile se depun la adresa ASCII+1
add al,30h
mov [ASCII],al          ; sutele se depun la adresa ASCII
ret                     ; revenire din procedură
convBinAscii endp

main label
        mov ax,@data
        mov ds,ax
        mov di,1
        mov si, offset a
        mov al, [si]    ; al=2
        mov bl, [si]    ; bl=2
eti:    mul bl
        inc di
        cmp di,[si+1]
        jb eti
call convBinAscii       ; apelare subrutină ce va depune în ASCII
                        ;cifrele (sute,zeci,unităţi) numărului de
                        ;afişat
        mov bx,offset ASCII ;poziţionare la încep şirului ASCII
        mov cx,3        ; avem de afişat 3 cifre

afis: mov al,[bx]       ; conţinutul adresei din reg bx se mută
                ;în reg al pentru a fi afişată în mod teletype
      mov ah,0eh
      int 10h
```

```
      inc bx
loop afis         ; cx=cx-1 şi se verifică dacă cx≠0
                  ;dacă da, se sare înapoi la "afis" pt a prelua
                  ;următoarea cifră a numărului de afişat
                  ;(bx a fost incrementat)
iesire:
       mov ax,4C00h;
       int 21h
end main
```

c) Pentru cazul n=8, se va obţine 2^8=256, număr care nu se încadrează în gama numerelor [0;255] ce se pot reprezenta pe 8 biţi. Astfel, numărul 2^8 este primul număr ce va trebui reprezentat şi cu "ajutorul" registrului AH. Pentru aceasta, subrutinei convBinAscii ii vor fi necesare şi „miimi" şi „zeci de miimi" (numerele ce se pot stoca în registrul AX aparţin gamei numerelor [0;65.535]).

```
.model small
.stack 200h
.data
        a db 2
        n db 8
        ASCII db 5 dup(0) ; există 5 zecimale (zeci de mii,
                          ; mii,sute,zecimi,unităţi)
.code

convBinAscii proc near ; procedura care depune în segm de date
                       ;începând de la adr ASCII cele 5 caract
                       ; ascii ale nr cuprins între 0 si 65.535
     xor dx,dx         ; nr-ul va fi în registrul ax,
                       ;deci se poate goli continutul reg dx
     xor cx,cx         ; cx=0
     mov cl,10         ; cx=10

     cmp ah,0          ; dacă ah=0, nr e stocat doar pe 8 biţi,
                       ;deci se va încadra în gama [0;255]
     jz et1            ; şi el se va putea scrie pe 3 caract
                       ;ASCII, nu are nevoie de 5 caract ASCII
     div cx            ; împărţim cu cx, pt a informa că ACC
                       ; este dx:ax, deci după div :
                       ; dx=restul(!dl=restul), ax=câtul
     add dl,30h
     mov [ASCII+4],dl  ; unităţi
     xor dx,dx
     div cx
     add dl,30h
     mov [ASCII+3],dl  ; zeci
     xor dx,dx
     xor ah,ah
     div cl
     add ah,30h
     mov [ASCII+2],ah  ; sute
     xor ah,ah
```

```
     div cl
     add ah,30h
     mov [ASCII+1],ah  ; mii
     xor ah,ah
     div cl
     add al,30h
     mov [ASCII],al    ; zeci de mii
     jmp ies           ; salt la revenirea din procedură
et1: div cl            ; se repetă procedura de la pct b)
                       ;dupa div: dx=restul, ax=catul
     add ah,30h
     mov [ASCII+4],ah  ; unităţi
     xor ah,ah
     div cl
     add ah,30h
     mov[ASCII+3],ah   ; zeci
     add al,30h
     mov [ASCII+2],al  ; sute
ies: ret
convBinAscii endp

main label
        mov ax,@data
        mov ds,ax
        mov di,1
        mov si, offset a
        xor bx, bx     ; bx=0
        xor dx, dx     ; dx=0
        mov al, [si]   ; al=2
        mov bl, [si]   ; bl=2

eti:    mul bl
        jo  eto        ; dacă a apărut depăşire la înmulţire,
                       ;înseamnă că va trebui să alocăm mai mult
                       ;de 8 biţi pt stocarea rezultatului
        inc di
        cmp di,[si+1]  ; am ajuns la sfârşit? (di=8?)
        jb eti         ; dacă încă nu, se reia bucla
        jmp apel       ; dacă da, sare la eticheta “apel”
eto:    xor bh,bh
        mul bx
        inc di
        cmp di,[si+1]
        jb eto
        div bx
apel:   call convBinAscii ; apelare subrutină
        mov bx,offset ASCII
        mov cx,5
afis: mov al,[bx]
      mov ah,0eh
      int 10h
      inc bx
      loop afis
```

```
iesire:
       mov ax,4C00h;
       int 21h
end main
```

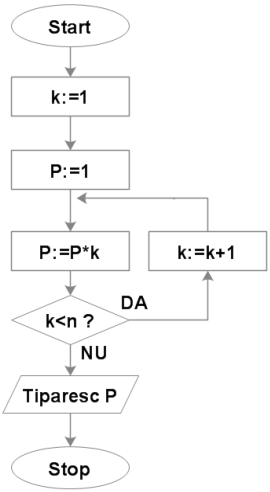
20. Scrieţi un program care să calculeze valoarea funcţiei x^n pentru x şi n daţi. In cazul particular se va considera x=3 şi n=2. Valoarea A este registrul AL, iar i va fi registrul DI. Alegeţi numerele x şi n astfel încât să obţineţi un rezultat final pe a) 8 biţi, b) 16 biţi şi c) 32 biţi si rescrieţi programele corespunzătoare. Se consideră numere fără semn. Folosiţi organigrama corespunzătoare şi problema 26 ca model.

```
Rezolvare:
a)
. . .
.data
        x db 3          ; definire variabile în segm de date
        n db 2
. . .
        mov di,0
        mov si, offset x
        mov al, 1       ; al=1 - iniţializare
        mov bl, [si]    ; bl=x valoarea cu care se înmulţeşte al
eti:    inc di          ; di=1 este folosit ca şi contor
        mul bl          ; ax=al*bl
        cmp di,2        ; am ajuns la final?
        jb eti          ; dacă încă nu, se reia bucla
afisare:                ; dacă da, se afişează rezultatul
. . .
```

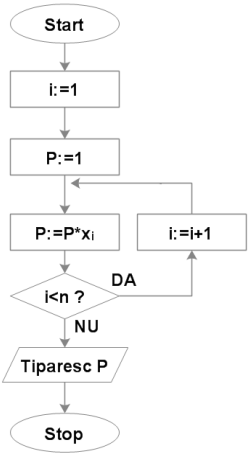
21. Să se scrie un program care să calculeze valoarea lui n! (factorial) şi să o afişeze pe ecran.

a) Se va lua cazul particular n=3 şi se va nota rezultatul final cu P (registrul AX). Valoarea k va fi stocată în registrul DI.
b) Ce modificări trebuie aduse pentru cazul n=6?
c) Dar pentru n=11 ? Se consideră numere fără semn. Folosiţi problema 26 ca model pentru afişarea rezultatelor.

```
Rezolvare:
a)
. . .
        mov bl,0    ; bl=0 rolul lui k
        mov al,1    ; al=1 rolul lui P
eti:    inc bl
        mul bl
        cmp bl,3    ; calculez 3!
        jb eti
afisare:
. . .
```



Organigrama problemei 21



Organigrama problemei 22

22. Să se scrie un program care să calculeze produsul a n numere reale date x_1, x_2, …, x_n. Valoarea produsului P va fi stocată în accumulator, i va fi registrul DI iar şirul de numere va fi definit în segmentul de date (a se vedea organigrama corespunzătoare). Pentru exemplificare se va considera cazul numerelor 2,4,1. b) Modificaţi programul astfel încât să calculeze produsul numerelor 2,4,6,8,10.

```
Rezolvare:
a) . . .
.data
        sir db 2,4,1
        lung_sir equ $-sir ; lung_sir va conţine dimensiunea
şirului "sir"
. . .
        mov di,offset sir ; poziţionare la începutul şirului sir
         mov al,[di]      ; al=2 rolul lui P
         mov cx,lung_sir ; cx=3
         dec cx          ; cx se decrementează
eti:     inc di          ; di se incrementează:
                         ; se trece la următorul element
         mov bl,[di]     ; se mută acest element in bl
         mul bl          ; ax=al*bl deci ax=2*4
         loop eti        ; cx=cx-1 si se verifică dacă cx≠0
                         ; dacă da, se sare înapoi la eti
afisare:                 ; dacă cx=0, se trece la afişare
. . .
```

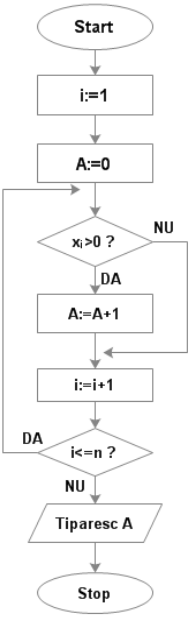
23. Să se determine numărul de numere pozitive dintr-un şir de numere date x_1, x_2, …, x_n. Se va considera şirul: `32h,84h,8fh,25h,26h,88h,0f5h,0d4h.`

```
Rezolvare:
.model small
.stack 200h
.data
     sir db 32h,84h,8fh,25h,26h,88h,0f5h,0d4h
        ; avem 5 nr. cu semnul negativ (adica MSB-ul lor e "1")
     lung_sir equ $-sir
.code
main label
        mov ax,@data
        mov ds,ax
        mov di,offset sir ;
        mov cx,lung_sir
        mov bl,0
eti:    mov al,[di]
        and al,80h
        cmp al,80h
        jnz eti1 ;dacă c=1-> nr e negativ şi nu se contorizează
        inc bl
eti1:   inc di
        loop eti
        mov al,bl
afisare:
        add al,30h
        mov ah,0eh
        int 10h
iesire: mov ax,4c00h
        int 21h
end main
```

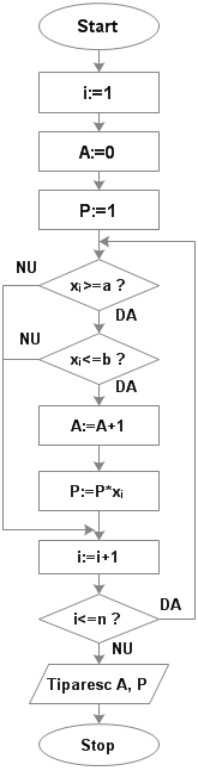
24. Se dă şirul de numere x_1, x_2, …, x_n: `2,3,5,4,6,7,8,11,23,12,34,25,10,9, 45h,50h`. Să se determine numărul termenilor şirului care aparţin intervalului [5,15] (in organigrama [a,b]) şi să se calculeze produsul lor.

```
Rezolvare:
.model small
.stack 200h
.data
     sir db 2,3,5,4,6,7,8,11,23,12,34,25,10,9,45h,50h
               ; 16 nr din care 7 sunt in intervalul [5,15]
     lim1 equ 5
     lim2 equ 15
     lung_sir equ $-sir
.code
main label
        mov ax,@data
        mov ds,ax
        mov di,offset sir
        mov cx,lung_sir
        mov dl,0       ; nr termenilor cuprinşi în interval
        mov al,1       ; produsul termenilor
eti:    mov bl,[di]    ; preiau fiecare nr din şir
```

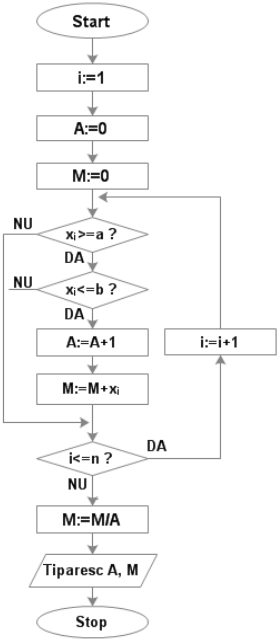
```
        cmp bl, lim1
        jng eti1  ;
        cmp bl,lim2
        jnl eti1
        inc dl
eti1:   inc di
        loop eti
        mov al,dl
afisare:
        add al,30h
        mov ah,0eh
        int 10h
iesire:
        mov ax,4c00h
        int 21h
end main
```



Organigrama problemei 23



Organigrama problemei 24



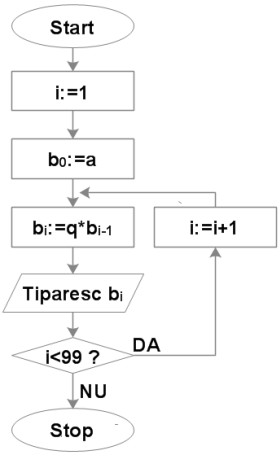
Organigrama problemei 25

25. Să se calculeze media aritmetică (rotunjită) a termenilor şirului x_1, x_2,.. x_n ce aparţin domeniului [a,b].

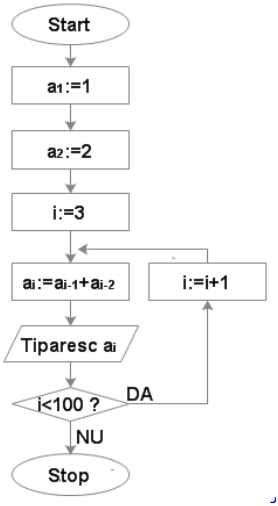
```
Solutie.

.model small
.stack 200h

.data
     sir db 2,3,5,4,6,7,8,11,23,12,34,25,10,9,45,50
                                ; în total sunt 16 nr din care
                                ; 3 sunt în intervalul [3,5]
     lim1 equ 3
     lim2 equ 5
     lung_sir equ $-sir

.code
main label
        mov ax,@data
        mov ds,ax

        mov di,offset sir

        mov cx,lung_sir
        mov dl,0       ; nr termenilor cuprinşi în interval
        mov ax,0       ; produsul termenilor

eti:    mov bl,[di]    ; preiau fiecare nr din şir
        cmp bl,lim1
        jnge eti1
        cmp bl,lim2
        jnle eti1
        add al,bl
        inc dl
eti1:   inc di
        loop eti
        div dl         ; in al se obtine câtul, în ah restul
afisare:
       add al,30h
        mov ah,0eh
        int 10h
iesire:
         mov ax,4c00h
        int 21h
end main
```

TEME:

26. Să se afiseze primii 100 termeni ai unei progresii geometrice cu primul termen a şi raţia q. Se va considera a=2 si q=2. Folosiţi organigrama corespunzătoare.



Organigrama problemei 26



Organigrama problemei 27

27. Să se calculeze primele 100 nr ale lui Fibonaci. Indicaţie: şirul lui Fibonaci: $a_n=a_{n-1} + a_{n-2}$, n=3,4,5,... cu $a_1=1$ şi $a_2=2$; rezultă şirul: 1,2,3,5,8,13,21,34... Folosiţi organigrama corespunzătoare.

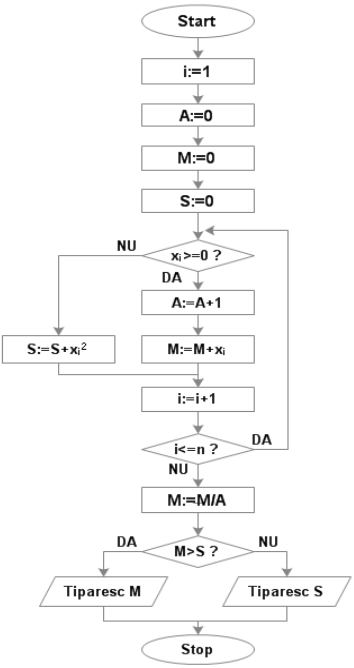
28. Se dă sirul $x_1, x_2, \ldots, x_n$. Să se calculeze:
- media aritmetică a nr pozitive (notată M)
- suma pătratelor numerelor pozitive S.

Să se tiparească cel mai mare număr dintre M şi S. Alegeţi şirul astfel încât să conţină atât numere pozitive cât şi numere negative.

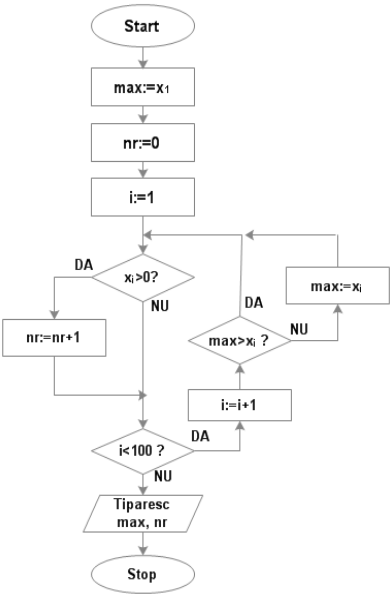
29. Să se determine maximul şi numărul de numere pozitive dintr-un şir de n de numere $x_1, x_2, \ldots, x_n$. Urmăriţi organigrama corespunzătoare (s-a considerat cazul n=100).

30. Să se ordoneze crescător numerele x_1, x_2, x_3. Indicaţie: Alegeţi numere pozitive.

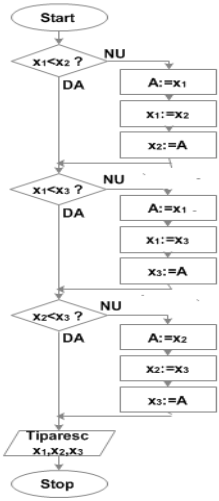
31. Să se ordoneze descrescător şirul de numere $x_1, x_2, \ldots, x_n$. Indicaţie: se aduce pe prima poziţie elementul maxim din şir, apoi următorul şi tot aşa. Deci x_1 se compară cu toate celelalte numere. Dacă 2 numere nu sunt în relaţia „>”, le vom schimba poziţia între ele şi continuăm.



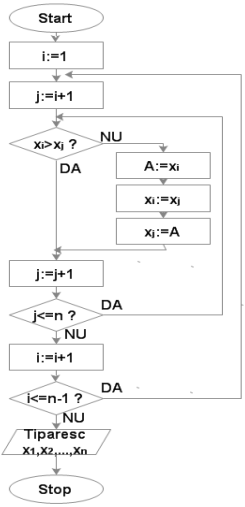
Organigrama problemei 28



Organigrama problemei 29



Organigrama problemei 30



Organigrama problemei 31

Întrebări și Teste

1. Pentru fiecare număr binar din tabel, dați valoarea echivalentă, presupunând că este reprezentat în codul indicat în capul de tabel. Dacă nu se poate reprezenta treceți EROARE.

		Cod		
Număr Hexazecimal	Număr Binar	8-biți fără semn	8-biți cu semn	ASCII paritate impară
0B5h	1011 0101b	181	-75	µ
3Bh	0011 1011b	59	+59	;

2. Care sunt elementele calculatorului cu program memorat, "*von Neumann*"?

3. Care sunt cele 3 magistrale ale unui calculator cu program memorat?

4. Un PC are 64MB de DRAM începând cu adresa 00000000H. Calculați care este adresa ultimei locații de memorie a acestui bloc de 64MB.

R: 03FFFFFH

5. Câte locații poate adresa un bus de adrese de 32 de biți ?

6. Ce arhitecturi de prelucrare cunoasteți ?

7. Ce reprezinta un microprocesor şi care sunt funcţiile lui?

8. Care este diferența dintre un microprocesor și un microcontroler ?

9. Unde este utilizată arhitectura Harvard? Ce avantaje are față de arhitectura von Neumann (SISD)?

10. Cum sunt aranjate datele arhitecturii SIMD? La ce procesor Intel apare această arhitectură?

11. Ce se înțelege prin paralelism la nivel de instrucțiune?

12. Ce se înțelege prin paralelism la nivel de procesor?

13. Ce beneficii aduce conceptul de „pipeline" în arhitectura de procesare a unui microprocesor?

14. Care sunt avantajele împărțirii interne a arhitecturii uP 8086 în BIU și EU?

15. Care sunt avantajele arhitecturii secvențial paralele la 8086?

16. Care sunt elementele componente ale unității de executie (EU) la 8086?

17. Care este rolul unității de interfață cu bus-urile (BIU)? Care sunt elementele ei componente?

18. Pentru calculatoarele cu program memorat care operează în 2 faze, acestea sunt ***FETCH*** și ***EXECUTE.*** La 8086 acești ciclii se pot suprapune sau nu? De ce?

R: Da, aceste faze se pot suprapune. Unitatile BIU şi EU sunt separate, iar o coadă de instrucțiuni în arhitectura procesorului permite extragerea de instrucțiuni în timp ce o alta se execută.

19. Care ar fi definiția actuală a unei arhitecturi superscalare?

20. Care sunt elementele principale ale arhitecturii procesorului Pentium?

21. Care sunt diferențele importante între arhitecturile RISC și CISC?

22. Ce reprezintă adresarea segmentată? Ce avantaje prezintă?

23. Ce reprezintă adresa logică?

24. Ce reprezintă adresa efectivă și unde se calculează?

25. Care este rolul regiștrilor segment în modul de calcul al adresei fizice?

26. Ce este adresa fizică și care este relatia ei cu adresa logică?

27. Care dintre adresele logice este greșită:

CS:IP SS:SI DS:DI ES:SI

28. Dati exemplu de 3 adrese logice care corespund adresei fizice 10000h.

29. Cate moduri de adresare de bază există la 8086? Enumerati.

30. Care este forma generală pentru adresarea indirecta a unui operand din memorie ? Exemplu.

31. Care este modul de adresare folosit în instrucțiunea: Mov ax, [5678h]

32. Clasificati instrucțiunile dupa numărul de operanzi și după funcția instrucțiunii.

33. Ce tipuri de date se pot intâlni în limbajul de asamblare?

34. Ce sunt etichetele? Dar constantele?

35. Explicați în ce constă setul de instrucțiuni MMX și unde sunt utile.

36. Ce reprezintă fisierele de tip BAT, COM și EXE?

37. Ce este prefixul unui program executabil (PSP)? Detalii.

38. Care sunt flag-urile aritmetice, dar cele de control ? Rolul lor.

39. Explicati rolul flagurilor C,Z,O.

40. Cum puteti controla bitul T (8 din PSW) ?

41. Explicati diferența între flagurile C și O .

42. Care din instrucțiunile de: deplasare/rotatie/logice nu afectează flagul Z?

43. De ce instrucțiunea SAR resetează întotdeauna flagul O ?

44. Indicati 3 moduri de a face Z=1.

45. Ce flag (flaguri) folosește 8086 pentru a verifica depașirea în arimetica fără semn ?

46. Ce flag se verifică pentru depașirea în aritmetica cu semn ?

47. Ce valori au AH, S și C dupa secvența de program:

```
xor ax, ax
(dec ax )
mov cl,2
sar ah,cl
```

48. Care sunt flagurile din FLAGS utilizate pentru definirea unui cod de condiţie (cc) ?

49. Ce flaguri sunt afectate de instrucţiunile LAHF şi SAHF ?

50. Ce flaguri nu pot fi modificate cu instrucţiunile SAHF/LAHF ?

51. Ce flaguri modifică instrucțiunile de mai jos și cum?

```
and al, 0           ; set Zero
or  al, 1           ; clear Zero
or  al, 80h         ; set Sign
and al, 7Fh         ; clear Sign
stc                 ; set Carry
clc                 ; clear Carry
mov al, 7Fh
inc al              ; set Overflow
or eax, 0           ; clear Overflow
```

52. Care este adresa fizică pe 20 biți pentru SS:SP, dacă avem următorul conținut (hexa) în regiștri:

DS: 0823, CS: 0824, SS: 0825, ES: 0826, BX: 000A, CX: 000D, BP: 002A, SP: 0004, SI: 0016, EAX: 40302010.

R: SS:SP = 08250h+0004h = 08254h

53. Presupunem continutul unor locații de memorie succesive ca în tabelul:

0825:0000	40	A8	2C	76	93	C5	0F	FE	**89**	2	D8	A4	8A	7C	DD	9E
0825:0010	A7	CC	9A	BD	8E	90	2C	0	E4	A0	0E	25	38	29	2C	86
0825:0020	82	A6	54	2E	9A	20	0A	98	**1C**	**90**	**0E**	**13**	8C	39	58	C6
0825:0030	90	3C	9B	83	65	19	F6	8A	C5	67	**A5**	**0**	12	BC	34	BB
0825:0040	76	D7	CA	FF	D8	71	18	24	F4	72	9	A3	29	1	D4	CE

Se presupune că avem conținutul registrelor (hexa) de mai jos astfel:

DS: 0825, CS: 0826, SS: 0827, ES: 0828, BX: 0003, BP: 001A, SP: 0008

Care este valoarea finală conținută de regiștrii ab,ax, eax.

a) mov ah, [BX+5] ; **ah = 89** *(DS:BX+5 = 08250+0003+5 = 08258)*
b) mov ax, [BP] ; **ax = 00A5** *(SS:BP = 08270+001A = 0828A)*
c) pop eax ;**eax = 130E901C** (*SS:SP = 08270+0008 = 08278)*

54. Se presupune că avem conținutul registrelor de mai jos astfel:

DS: 0823, CS: 0824, SS: 0825, ES: 0826, BX: 0003, CX: 000D, BP: 001A, SP:0008, SI: 0006, EAX: 44332211.

Indicați cum se modifică locațiile pentru următoarele instrucțiuni:

a) mov [BP+4], eax ;***SS:BP+4 = 08250+001A+4 = 0826E***
b) mov [BX+SI*3],ax ;***DS:BX+SI*3 = 08230+0003+(0006*3) = 08245***

0823:0000																
0823:0010						**11**	**22**									
0823:0020																
0823:0030															**11**	**22**
0823:0040	**33**	**44**														

55. Pentru sumele de mai jos, indicați dacă apare depășire fără semn (C), depașire cu semn (O), niciuna sau ambele:

	C	O	Explicație
08 + F0 = F8			*pozitiv + negativ nu dă O, C=0*
12 + F1 = 03	x		*pozitiv + negativ nu dă O, C=1*
48 + 63 = AB		x	*pozitiv + pozitiv = negativ -> depășire cu semn, O=1*
38 + 42 = 7A			*pozitiv + pozitiv = pozitiv, nu dă O, C=0*
D7 + F0 = E7	x		*negativ + negativ = negativ; C=1*
C2 + AA = 6C	x	x	*negativ + negativ = positiv; O=1, C=1*

56. Dacă flagul C=1 ce va face instrucţiunea "ADC CX, BX" ?
57. Ce va face instrucţiunea "ADC CX, BX" dacă flagul C=0?
58. Ce operații se inițiază la acceptarea unei intreruperi ?
59. Ce deosebiri sunt între întreruperile hardware mascabile și nemascabile?
60. Care dintre instrucțiunile de mai jos nu este acceptată de 8086 ?

```
Popa
Movsx
Push sp
Pull eax
Xchg ds,cs
```

61. Indicati instrucțiunile gresite și motivati?

```
Mov ah,cx
In 123h,al
Call phone
Mov al,dl
Clt
Pop ion
```

62. Care este diferența între instrucțiunile următoare : dec ax și sub ax,1
63. Dati 3 exemple de a obtine în BX=0
64. Cu ce instrucțiune este echivalentă secvența:

```
Add AL,1
Adc AH,0        ;R: echivalentă cu Add ax,1
```

65. Ce efect are secventa urmatoare?

```
abs: add ax,0
     jns ET1
     xor  ax,0FFFFH
     inc ax
et1:
```

66. Indicați 3 instrucțiuni din setul extins x86.
67. Ce este un ciclu mașină ?
68. Care sunt diferențele între un program .COM și unul .EXE ?
69. Clasificați întreruperile la 8086.
70. Ce este TVI-ul (tabela vectorilor de intreruperi) ? Dimensiunea și localizarea ei.
71. Clasificați instrucțiunile de salt.

72. Care sunt limitările instrucțiunilor de salt condiționat ?

73. Care instrucțiuni de salt nu țin cont de starea flagurilor de condiție ?

74. Ce reprezintă o subrutină ?

75. Ce sunt macroinstrucțiunile?

76. Care este deosebirea între o subrutină și o macroinstrucțiune ?

77. Dați un exemplu de macroinstrucțiune care initializează regiștrii generali (AX=BX=CX=DX=0).

78. Care sunt etapele în dezvoltarea unei aplicatii în limbaj de asamblare?

79. Dați un exemplu care indică faptul că rezultatul sumei a 2 numere binare de n-biți necesita n+1 biți

80. Ce pune pe stiva în plus instrucțiunea INT xx față de instrucțiunea CALL FAR ?

81. Ce instrucțiuni aveți nevoie pentru a citi portul de 8 biți 378h ? Dati exemplu.

82. Scrieti o secvențăde program (loop) care apelează rutina CALL_ME de 25 ori.

83. Care este diferența între instrucțiunile IRET și RET (FAR) ?

84. La ce este utilizată instrucțiunea JCXZ de obicei ?

85. Indicati 4 moduri diferite pentru a aduna 2 la valoarea din BX. Nici unul din moduri nu trebuie sa aiba mai mult de 2 instrucțiuni (exista cel putin 6 moduri).

86. Explicați noțiunile big endian și little endian de reprezentare a datelor în memorie.

87. Dacă imediat la intrarea într-o subrutină se execută pop ax, ce va contine ax ?

88. Să presupunem că trebuie permutate valorile din registrele CX şi BX. Scrieţi codul care să facă aceasta folosind doar instrucţiuni MOV şi registrele AX, BX şi CX.

89. Să presupunem că trebuie interschimbate valorile de 16 biţi din locaţiile de memorie X şi Y. Scrieţi trei instrucţiuni care realizeze această operație (Obs. Una din ele este XCHG).

90. Ce face instrucţiunea "LEA AX, etich[bx]"?

91. Care este diferenţa între "MOV dx, [bx]" şi "LEA dx, [bx]" ?

92. De ce nu este instrucţiunea "LEA ax, bx" corectă? Explicaţie.

93. Consideram programul în LA 8086, care ruleaza sub DOS pe un PC

```
ORG 100h
      mov si,0
      mov dl,30h
      mov ah,2
  Et: int 21h
      inc dl
      inc si
      jmp si,8
      jne Et
      int 20h
```

Se cere : a) care sunt valorile registrelor SI și DL la ieșirea din program;
b) ce este afișat pe ecran la terminarea aplicației ?

94. Ce instrucţiuni de forma L?S sunt disponibile pentru procesoarele x86 ?

95. Ce realizează instrucţiunea "LES DI, [BX+3]" ?

96. Ce înseamnă noțiunea "LIFO" (Last In First Out)?

97. Dacă ar trebui să salvaţi în stivă AX şi BX (în această ordine), care valoare ar trebui restaurată prima?

98. Dacă (E)SP are valoarea 8810h şi salvaţi în stivă registrul AX, care va fi valoarea lui (E)SP după execuţia instrucţiunii PUSH?

99. Dacă (E)SP are valoarea 8FFCh şi salvaţi în stivă registrul EAX, care va fi valoarea lui (E)SP după execuţia instrucţiunii PUSHD ?

100. Dacă restauraţi valoarea din vârful stivei în registrul BX atunci când (E)SP avea valoarea 901Ch, ce va conţine (E)SP după execuţia instrucţiunii POP ?

101. Un procesor 8086 nu permite încărcarea unei constante imediate într-un registru de segment. Cum puteţi utiliza instrucţiunile PUSH şi POP (la un procesor cel puţin 286) pentru a încărca o constantă în registrul DS.

102. Atunci când salvaţi CX urmat de DX în stivă, şi apoi restauraţi CX şi DX (în această ordine) din stivă, se vor interschimba valorile din CX şi DX ? Explicaţi.

103. Presupunem că registrul ECX are valoarea 11223344H. Dacă salvaţi registrul ECX în stivă şi apoi restauraţi registrul BX urmat de AX, care vor fi valorile din registrele BX şi AX după această secvenţă ?

104. Instrucţiunea BSWAP permută octeţii individuali dintr-un registru de 32 de biţi. Ce instrucţiune se poate utiliza pentru a permuta cele două jumătăţi ale registrului AX?

105. Care este o modalitate simplă de a realiza o extindere cu 0 ("zero extension") a lui AL în AX folosind doar o instrucţiune MOV ?

106. Presupunând că aţi răspuns la întrebarea anterioară, daţi un exemplu din care să rezulte utilitatea instrucţiunii MOVZX ?

107. Să presupunem că vreţi să împărţiţi valoarea (cu semn) din AX cu valoarea din BX folosind instrucţiunea IDIV. Instrucţiunea IDIV cere o valoare pe 32 de biţi în DX:AX înainte de împărţirea propriu-zisă. Ce instrucţiune folosiţi pentru acesta ?

108. Să presupunem că aveţi o tabelă de 128 de octeţi care conţine valorile 0, 1, 2, 3, ...80H. Dacă BX conţine valoarea adresei primului octet din tabelă, ce valoare va conţine AL după execuţia instrucţiunii XLAT ? Explicaţie.

109. Presupunem că doriţi să creaţi o funcţie care să returneze în registrul AL=1, dacă caracterul din AL este un semn de punctuaţie şi AL= 0 în caz contrar. De ce tabelă aveţi nevoie pentru a implementa această funcţie, daca folosim instrucţiunea XLAT.

110. Să presupunem că registrul șI conţine valoarea de intrare pentru funcţia descrisă în problema anterioară. Ce instrucţiune puteţi folosi pentru a realiza aceiaşi operaţie ca instrucţiunea XLAT utilizată anterior ?

111. Presupunem că avem o tabelă de 256 de octeţi, care conţine caracterele "a" ...."z" la indicii 41h până la 5Ah (valorile respective sunt codurile ASCII ale caracterelor "A" .."Z"). Explicaţi ce efect are instrucţiunea XLAT când este utilizată cu această tabelă.

112. Instrucţiunea CMP este nedestructivă, în sensul că nu modifică valoarea nici unui operand. Daţi un exemplu de instrucţiune care face acelaşi lucru ca şi CMP, dar este o operaţie destructivă.

113. Pe lângă faptul că instrucţiunile INC şi DEC nu afectează flagul Carry, care ar fi alte motive pentru a utiliza aceste instrucţiuni şi nu ADD sau SUB?

114. În ce circumstanţe instrucţiunea NEG va seta flagul Zero ?

115. Ce calculează instrucțiunea "IMUL AX, BX, 20" ?

116. Presupuneţi că aveţi un vector (o matrice tridimensională) de octeţi (caractere) cu forma: "XYZ: array [0..3, 0..4, 0..5] of char;" Ce instrucţiuni veţi folosi pentru a încărca elementul "XYZ[i, j, k]" în AL?

117. Presupuneţi că aveţi valoarea 20000h în DX:AX şi împărţiţi această valoare cu valoarea din CX. Care vor fi cele trei valori din CX care vor genera o eroare de împărţire?

118. Ce altă eroare mai poate să apară la o operaţie de împărţire, pe lângă situaţia în care câtul are o dimensiune mai mare decât registrul destinaţie?

119. Ce instrucţiune logică poate forţa toţi biţii operandului din AL în zero, cu excepţia celui mai puţin semnificativ bit (LSB) ?

120. Presupunând că L, M, N, P, şi Q sunt variabile booleene, memorate pe cate un octet (0FFh -True, 00h -False), care este codul care implementeaza funcţia logică: L := (M and N) or not (P and Q).

121. Ce instrucţiune/instrucţiuni puteţi folosi pentru a înmulţi valoarea din AX cu opt, fără să folosiţi instrucțiuni de adunare sau înmulțire?

122. Când se împarte o valoare întreagă la doi este posibil să obţineţi o valoare inexactă (un număr impar împărţit la doi). Cum puteţi să testaţi, după execuţia instrucţiunii SAR/SHR, că rezultatul este inexact ?

123. Cum se utilizeaza instrucţiunea AND pentru a face AX=0, dacă valoarea lui este pară şi respectiv AX=1, dacă valoarea lui impară ?

124. Ce altă instrucţiune operează în aceiaşi manieră nedestructivă ca instrucţiunea TEST? Ce operaţie realizează ea ?

125. Cum puteţi folosi instrucţiunea TEST pentru a afla dacă valoarea din BX este pară sau impară. De ce această metodă este mai bună decât cea care utilizează instrucţiunea AND ?

126. Cum puteţi folosi instrucţiunile BT şi ADC pentru a rotunji toate numerele impare la următoarea valoare, mai mare, pară?

127. Instrucţiunile BSF şi BSR caută doar primul bit setat (în unu) într-o valoare. Dacă vreţi să căutaţi primul bit resetat (în zero) într-o valoare, ea trebuie prelucrată astfel ca biţii în unu să fie convertiţi în zero şi vice-versa. Cu ce instrucţiune se realizează aceasta?

128. Să presupunem că vreţi să ştergeţi bitul în unu detectat de instrucţiunea "BSR AX, Bits", imediat după execuţia instrucţiunii BSR. Ce instrucţiune veţi utiliza, instrucţiune care să folosească valoarea din AX?

129. Pentru a realiza operaţia booleană NOT pe o valoare multi-bit aveţi nevoie să inversaţi doar cel mai puţin semnificativ bit, nu toţi biţi ca la instrucţiunea NOT. Ce instrucţiune puteţi folosi pentru a inversa doar cel mai puţin semnificativ bit?

130. Prefixul REP poate fi folosit cu instrucţiunea LODS, dar de ce nu prea are sens să procedăm astfel? (Obs.Ce se întâmplă dacă executăm două instrucţiuni LODSB, succesive.)

131. În anumite privinţe instrucţiunea JMP este o altă formă a instrucţiunii MOV. Explicaţie.

132. Care este diferenţa între instrucţiunile "JMP BX" şi "JMP [BX]" ? Explicaţie.

133. Scrieţi codul care să permute valorile din CX şi DX dacă DX este mai mic decât CX.

134. Să presupunem că adresa de destinaţie a instrucţiunii "JE ETCH" este în afara domeniului accesibil pentru această instrucţiune. Ce cod corectează această problemă?

135. Ce valoare va avea ax la ieşirea din bucla ?

```
       mov ax,6
       mov cx,4
  Et:  inc ax
       loop et          ; R: ax=10
```

136. Corectaţi codul de mai jos, astfel ca să nu se execute de 65536 de ori dacă cumva CX=0 la intrarea în buclă.

```
(posibil ca în acest punct CX = 0).
Loop1:     mov [bx] , ax
           add bx , 7
           loop Loop1
```

137. De câte ori se execută bucla de mai jos?

```
   mov ecx,0      dar            mov cx,0
et:inc ax                   et1: inc ax
   loop et                      loop et1
```

138. Scrieti secventa în LA pentru secvența de pseudocod următoare. Valorile sunt cu semn.

```
If (ax>bx)                cmp ax,bx
mov dl,5;                 jg L1          ;ax>bx
    else                  mov dh,6       ;dh=6
mov dh,6;                 jmp L2
                    L1:   mov dl,5       ;dl=5
                    L2: .......
```

139. Implementati secvenţa de pseudocod în asambare. Valorile sunt fără semn.

```
a)  If ( ebx <= ecx )
    {
    eax = 5000;
    edx = 6000;
    }
```

```
b)  if (al > bl) AND (bl > cl)
    Y = 5;

c)  if (al > cl) OR (cl > bl)
    Z = 2;
```

140. Implementaţi secvenţa de pseudocod în asambare. Valorile sunt cu semn pe 32 de biţi.

```
If (var1<=var2)              mov eax,var1
   var3=100;                 cmp eax,var2
else                         jle L1        ;var1<= var2
{  var3=16;                  mov var3,16   ;var3=16,daca
                                           ;var1>var2
   Var4=24;                  mov var4,24   ;var4=24
}                            jmp L2
                      L1:    mov var3,100  ;var3=100
                      L2: …
```

141. Implementaţi secvenţa de pseudocod în asambare. Valorile sunt fără semn pe 32 de biţi.

```
a)while (eax<ebx)            R:     while: cmp eax,ebx
       {                                   jae _endwhile
    eax++                                  inc eax
    if (ebx==ecx)                          cmp ebx,ecx
    q=20                                   jne _else
    Else                                   mov q,20
    q=7                                    jmp _while
       }                                 else:  mov q,7
                                               jmp _while
                                          endwhile:

b) while (ax>=cx)
       cx=cx+1 ;

d) while ( eax<=val )
       {
           eax = eax -1;
           val= val-1 ;
        }
```

142. Scrieţi un program în limbaj de asamblare care să ceară utilizatorului să introducă un număr între 1...366 şi să afişeze luna corespunzătoare zilei anului care a fost introdusă. Vezi modelul următor :

```
D:\> Introduceti un numar intre 1….366: 23
Ziua 23 a anului apartine lunii ianuarie.
```

143. Să se genereze secvenţa ‘ hailstone’ (grindină) plecand de la o valoare n, astfel : prima valoare este n, apoi daca n este par valoarea următoare este n/2, iar dacă este impar valoarea este 3n+1. Se calculează valorile până se ajunge la 1. Să se determine si numărul de elemente ale sirului. Ex. n=7, sirul are 17 elemente (7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1).

Secvenţa de cod in limajul C pentru determinarea numărului de elemente a sirului este :

```
        # include <stdio.h>

        int main(void)
{
        int n=32 ;
        int count = 0 ;
        while (n > 1)
    {
        count++ ;
        if (n & 1)
          n =3*n + 1 ;
        else
         n /=2 ;
     }
        printf (‘%d\n’ , count) ;
        return 0 ;
}
```

ANEXA 1

	8086	8088	80286	80386	80486	Pentium	Pentium Pro	Pentium II	Pentium III	Pentium 4	Pentium 4 EE
Anul	1978	1979	1982	1985	1989	1992	1995	1997	1999	2001	2004
Tact (MHz)	5-10	5-8	6-16	16-33	25-66	60/ 66	150	400	800	1700	3200-3700
Număr de tranzistoare	29 k	2 k	130 k	275 k	1.2 M	3.1 M	5.5 M	7.5 M	28 M	42 M	55M
Memorie fizică	1 Mo	1 Mo	16 Mo	4 Go	4 Go	4 Go	64 Go	64 Go	64 Go	64 Go	64 Go
Magistrală de date internă	16	16	16	32	32	32	32	32	32	32	32/64
Magistrală de date externă	16	8	16	32	32	64	64	64	64	64	64
Tipul datelor	8, 16	8, 16	8, 16	8, 16, 32	8,16, 32	8,16, 32	8, 16, 32	8, 16, 32	8, 16, 32	8, 16, 32	8,16,32, 64
Memoria Cache	--	--	--	--	8 ko L1	16 ko L1	16 ko L1 256/512ko L2	32 ko L1 256/512ko L2	32 ko L1 256/512ko L2	20 ko L1 512ko L2	+ 2Mo L3
MIPS~	0.8	0.8	2.7	10	50	100	440	440	700	3000	10000

Caracteristici generale ale unor procesoare procesoare INTEL pe 16/32 de biţi utilizate in PC-uri

ANEXA 2

Structura PSP

Offset	Dimensiune	Descriere
00h	word	codul maşină al instrucţiunii INT 20h (CDh 20h)
02h	word	vârful memoriei, în forma de segmanet (paragraf)
04h	byte	rezervat DOS, de obicei 0
05h	5bytes	codul maşină pentru long call DOS function dispatcher (obsolete CP/M)
06h	word	octeţi disponibili în segment în programele .COM (CP/M)
0Eh	dword	adresa de ieşire INT 23 Ctrl-Break; vectorul original INT 23 NU este refăcut din acest pointer (IP,CS)
12h	dword	adresa de ieşire în caz de eroare critică: vectorul INT 24 NU este refăcut din acest pointer (IP,CS)
16h	word	adresa de segment a procesului părinte; (Undoc. DOS 2.x+) COMMAND.COM are identitate de părinte zero sau propriul său PSP
2Ch	word	adresa de segment a mediului, sau zero (DOS 2.x+)
2Eh	dword	SS:SP la intrarea în ultima funcţie INT 21 function (Undoc. 2.x+) +
38h	dword	pointer la PSP anterior(deflt FFFF:FFFF, Undoc 3.x+)
3Ch	20bytes	nefolosit inainte de DOS 4.01 +
50h	3bytes	DOS function dispatcher CDh 21h CBh (Undoc. 3.x+) +
53h	9bytes	nefolosit
80h	byte	numărul de caractere din coada liniei de comandă; se numără octeţii de după numele comenzii; maxim 128
81h	127 bytes	caracterele introduse după numele programului, urmate de un octet CR

ANEXA 3

Moduri Video						
Mode	**Tip**	**Nr. Max Culori**	**Dimen-siune**	**Rezolutie**	**Nr. Max Pagini**	**Adresa de Baza**
00	Text	16	40x25	- -	8	B8000h
01	Text	16	40x25	- -	8	B8000h
02	Text	16	80x25	- -	4,8	B8000h
03	Text	16	80x25	- -	4,8	B8000h
04	Grafic	4	40x25	320x200	1	B8000h
05	Grafic	4	40x25	320x200	1	B8000h
06	Grafic	2	80x25	640x200	1	B8000h
07	Text	Mono	80x25	- -	1,8	B0000h
08	Grafic	16	80x25	160x200	1	B0000h
09	Grafic	16	40x25	320x200	1	B0000h
0A	Grafic	4	80x25	640x200	1	B0000h
0B	Rezervat (intern la EGA)					
0C	Rezervat (intern la EGA)					
0D	Grafic	16	40x25	320x200	8	A0000h
0E	Grafic	16	80x25	640x200	4	A0000h
0F	Grafic	Mono	80x25	640x350	2	A0000h
10	Grafic	16	80x25	640x350	2	A0000h
11	Grafic	2	80x25	640x480	1	A0000h
12	Grafic	16	80x25	640x480	1	A0000h
13	Grafic	256	40x25	320x200	1	A0000h
……						

A4. Dezvoltarea programelor în limbaj de asamblare (I)

Etapele prin care trece un program scris în limbaj de asamblare până a deveni un program executabil sunt prezentate în schema din figura 8.1. Procesul de obţinere a executabilului este un process repetitiv; trecerea de la o fază la alta se face numai dacă au fost eliminate toate erorile sintactice din faza curentă. În ultima fază de testare cu ajutorul depanatorului se elimină erorile de logică mergând în paralel cu microprocesorul şi comparând în diferite puncte rezultatele obţinute cu cele dorite.

Editorul de texte este folosit la introducerea programului scris în limbaj de asamblare şi respectă convenţiile şi sintaxa impusă de asamblor rezultând un program sursă (**.asm**). Se recomandă utilizarea unui editor simplu ASCII.
Asamblorul este un program de conversie în cod maşină a programelor scrise în limbaj de asamblare (.asm) obţinând un fişier obiect (.**obj**). Asamblorul bsolute asigură şi prelucrarea etichetelor simbolice astfel încât fiecare adresă simbolică este înlocuită cu o adresă relativă sau absolută.
Editorul de legături (Link-editorul) realizează legarea mai multor module obiect rezultate din asamblări sau componente de bibliotecă obţinând un program în format absolut sau relocabil (toate adresele sunt relative la o bază şi se transformă în adrese absolute la încărcare) (.**com, .exe**).
Depanatorul este utilizat la testarea programului executabil şi încărcat în memorie, pentru a înlătura erorile semantice ale programului folosind facilităţile debbuger-ului.
Programul sursă trebuie să respecte convenţiile şi sintaxa pe care o cere programul asamblor. Liniile programului sursă pot conţine instrucţiuni sau directive de asamblare (pseudoinstrucţiuni).

A4.1. Definirea şi iniţializarea datelor

Tipurile de date întâlnite în programe pot fi:

- constante
- variabile
- etichete

Instrucţiunile de transfer, aritmetice, logice (MOV, ADD, XOR) folosesc ca operanzi variabile şi constante în timp ce instrucţiunile de salt, apel (JMP, CALL) folosesc ca operanzi etichetele.

Etichetele sunt utilizate pentru identificarea codului, desemnând adrese în zona program.
Exemplu:

```
Aduna: add ax,bx
       inc bx
       jmp Aduna        ;Aduna - etichetă de tip NEAR
```

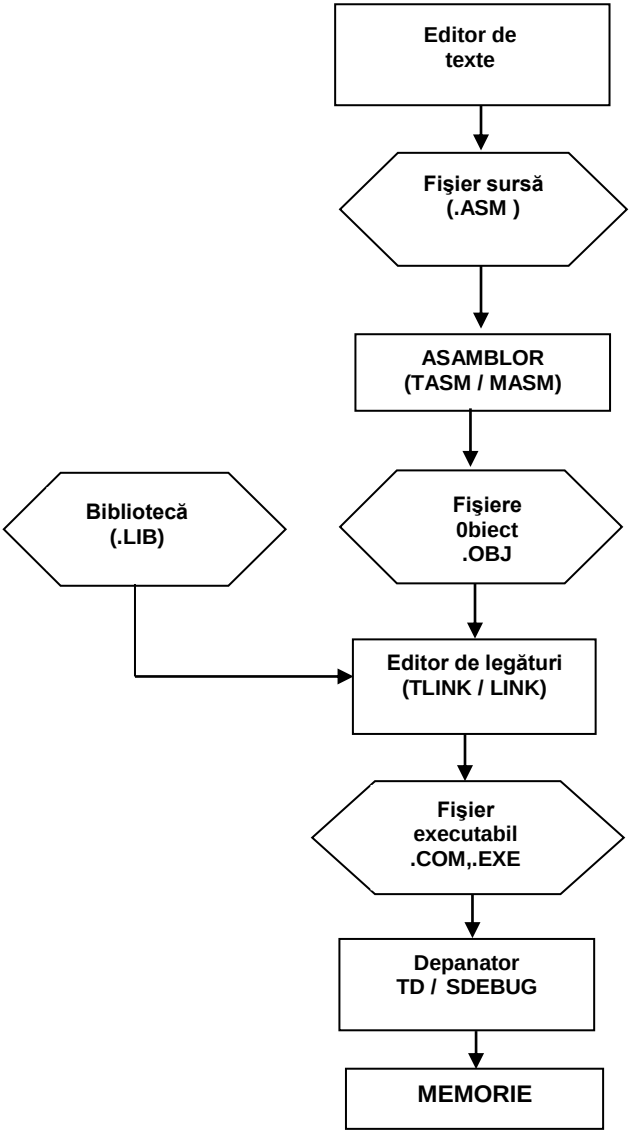


Fig.A4.1. Dezvoltarea aplicaţiilor în limbaj de asamblare

Constantele pot fi absolute (numerice) sau simbolice – nume generice asociate unor valori numerice. Constantele numerice pot fi de tipurile prezentate mai jos.

Binare	ex. 11001110B
Zecimale	ex. 28[D]
Hexa	ex. 32h, 0A5h
Octale	ex. 56o
Caracter	ex. 'alba'

Constantele simbolice se pot defini astfel:
Nume EQU expresie

Exemplu: `CR EQU 0Dh`

Variabilele identifică datele (un spaţiu de memorie rezervat) şi pot fi de următoarele tipuri:

BYTE - în memorie sau în registru de 8 biţi
- întreg cu/fără semn pe 8 biţi
- caracter ASCII

WORD - în memorie sau în registru pe 16 biţi
- întreg cu/fără semn pe 16 biţi
- secvenţă de 2 caractere ASCII
- adresă de memorie

DWORD - în memorie sau într-o pereche de regiştri pe 16 biţi
- întreg cu/fără semn pe 32 biţi
- număr real în simplă precizie
- adresă de memorie pe 32 biţi

QWORD - în memorie
- întreg cu/fără semn pe 64 biţi
- număr real în dublă precizie

TBYTE - în memoria internă sau în registrele coprocesorului aritmetic
- număr real în precizie extinsă (80 biţi)
- număr întreg ca secvenţă de cifre BCD

Variabilele se definesc folosind directivele: DB, DW, DD, DQ, DT cu sintaxa de mai jos:
Nume **directiva** lista_de_valori

unde în lista_de_valori putem avea :
- constante ex. A DB 0CFh, 21
- simbolul (?) pentru locaţie neiniţializată ex. STIVA DW 256 DUP(?)

- variabilă sau etichetă
- şir ASCII ex. SIR DB 'Exemplu'
- operatorul DUP pentru iniţializare tablou, ex: STRING DB 10 DUP(0).

Operatori. Limbajul de asamblare pune la dispoziţia utilizatorilor mai mulţi operatori pentru modificarea, combinarea, compararea sau analiza operanzilor. Operatorii apar în expresii aritmetice şi logice evaluate la momentul asamblării şi din care rezultă constante numerice.

- *operatori aritmetici şi logici*: +, -, *, /, MOD, SHL, SHR, NOT, AND, OR, XOR

Exemple:
```
MOV BX,6        ; BX=6
SHL BX,3        ; BX=30H
XOR BX,6        ; BX=BX⊕6, BX=36H
```

- *operatori relaţionali* : EQ, NE, LT, LE, GT, GE care au semnificaţii evidente şi returnează valori logice codificate ca 0 sau secvenţe de 1 pe un anumit numar de biti.

Exemple:
```
A  DB 5 EQ 7    ; A=0
C  DW 1 NE 5    ; C=0FFFFh
```

- *operatori de atribuire:* definesc constante simbolice

```
nume_var EQU expresie
```

Exemplu:
```
N EQU 17        ; sau N=17
```

- *operatori care returnează valori*: se aplică unor variabile sau etichete returnând valori ale acestora.
- *operatorul SEG* aplicat variabilelor sau etichetelor returnează adresa de segment asociată variabilei respective

```
MOV AX, SEG V1
```

- *operatorul OFFSET*, similar operatorului SEG returnează offset-ul variabilei/etichetei

```
MOV BX,OFFSET V1
```

- *operatorul THIS* creează un operand care are o adresă de segment şi un offset identic cu contorul de locaţii curent. Sintaxa de utilizare este: THIS tip

```
VAR EQU THIS WORD
```

unde tip poate fi: BYTE, WORD, DWORD, QWORD sau TBYTE pentru variabile sau NEAR/FAR pentru etichete.
- *operatorul TYPE* se aplică variabilelor sau etichetelor şi returnează tipul acestora. Pentru variabile se returnează valorile 1, 2, 4, 8 sau 10 (dacă anterior au fost definite cu directivele: DB, DW, DD, DQ, DT), iar pentru structuri numărul de octeţi pe care este memorată structura respectivă. Pentru etichete returnează tipul NEAR sau FAR.
Exemplu:

```
VAR DW 1,2,10h          ; variabila VAR este de tip cuvânt, cu valorile 1,2 şi 10h
    sub BP, TYPE VAR    ; asamblorul înlocuieşte TYPE VAR cu 2.
```

- *operatorul LENGTH* se aplică variabilelor şi returnează numărul de elemente definite în variabila respectivă

Exemplu:

```
TAB     DW 50 DUP(?)
MOV CX, LENGTH TAB     ; cx=50
```

- *operatorul SIZE* aplicat variabilelor returnează dimensiunea în octeţi a variabilei respective, astfel pentru exemplul anterior expresia SIZE TAB =100.
Pentru o variabilă oarecare "X" avem: SIZE X = (LENGTH X) * (TYPE X).

- *operatorul PTR* are ca efect schimbarea tipului variabilei/etichetei şi utilizarea lui este obligatorie în cazul unor referinţe anonime la memorie. Sintaxa de utilizare este:

tip PTR expresie

Exemplu:

```
INC BYTE PTR[BX]; octetul adresat de BX este incrementat

JMP FAR PTR Var ; salt de tip FAR
```

Exemplu:

```
DATA DB 03,04,05,06     ;variabila DATA declarată ca sir
                        ;de octeti
mov AX, WORD PTR DATA   ;asamblorul foloseste elementele
                        ;variabilei DATA ca şi cuvinte
mov WORD PTR DATA+2,DX ;se obţine compatibilitate cu
                        ;tipul cuvânt al registrelor
```

- *operatorii NEAR/FAR/SHORT* poziţionează tipul unei etichete în modul specificat de operator.

Exemple:

```
JMP     NEAR  ET1       ;salt intrasegment
JMP     SHORT ET        ;salt scurt sub 128 octeţi
```

- *operatorii HIGH/LOW* returnează octetul cel mai semnificativ respectiv cel mai puţin semnificativ al unei expresii.

Exemple:
```
FULL    EQU     1234H
MOV AH, HIGH FULL       ;AH=12H
MOV AL, LOW FULL        ;AL=34H.
```

Directiva *LABEL*, având forma generală:
nume* LABEL *tip ,unde ***tip*** poate fi
- NEAR sau FAR dacă ceea ce urmează sunt instrucţiuni si eticheta va fi folosită pentru referirea la instrucţiuni de tip JMP/CALL şi
-BYTE, WORD, DWORD dacă ceea ce urmează reprezintă definiţii de date.

Exemplu:
```
sirb LABEL BYTE   ; s-a definit eticheta cu numele sirb ce
                  ; permite accesul la variabila sirw octet
                  ; cu octet
sirw DW 1234h
 mov AL, sirb     ; AL=34h
```

A4.2 Definirea segmentelor

Un modul dintr-un program poate fi: o porţiune dintr-un segment, un segment, porţiuni din segmente diferite sau mai multe segmente. Modulele obiect se leagă la link-editare. Instrucţiunile şi datele dintr-un program .asm, indiferent de modul de dezvoltare, trebuie să se găsească în interiorul unui segment. Definirea unui segment se face după modelul de mai jos, folosind directivele SEGMENT respectiv ENDS:

nume **SEGMENT** *[tip_aliniere] [tip_combinare] ['clasa']*
.
.
.
nume **ENDS**

unde :
- ***nume*** este numele segmentului
- ***tip_aliniere*** specifică tipul adresei de inceput a segmentului unde va fi relocat segmentul în memorie şi poate fi: PARA (implicit) paragraf – multiplu de 16 bytes, BYTE, WORD, DWORD, PAGE - multiplu de 256 bytes.
- ***tip_combinare*** specifică dacă şi cum se va combina segmentul respectiv cu alte segmente la editarea de legături. Poate fi:
  - PUBLIC : segmentul curent se va concatena cu alte segmente având acelaşi nume şi atributul PUBLIC;
  - COMMON: segmentele cu acelaşi nume şi atribut se vor suprapune, lungimea finală va fi maximul lungimii segmentelor componente;

STACK : specifică segmentul curent ca fiind stiva programului (LIFO) dacă sunt mai multe segmente cu tipul STACK se vor concatena;
AT exp : indică faptul că segmentul curent va fi plasat la o adresă fizică absolută dată de exp în memorie;
Necombinabil - implicit (nu se scrie nimic).

- ***clasa*** indică un nume de clasă pentru segment care poate fi : 'code' , 'data', 'stack'. Dacă segmentul are şi nume de clasă atunci atributele COMMON şi PUBLIC vor acţiona numai asupra segmentelor cu acelaşi nume şi din aceeaşi clasă.

Directiva **ASSUME** reg_seg: nume_seg [, ….] realizează o conexiune logică între definirea datelor şi instrucţiunilor în segmente la asamblare şi accesul, la execuţie, la date şi instrucţiuni prin intermediul regiştrilor segment. Directiva ASSUME nu realizează încărcarea regiştrilor segment cu adresele de segment corespunzătoare.

Exemplu: `ASSUME CS:COD_SEG, DS:D_SEG, SS:STIVA, ES:NOTHING.`
Deci la instrucţiunea : `MOV BX,CEVA ; BX=(DS:CEVA).`

Directiva **GROUP** se utilizează la gruparea mai multor segmente sub acelaşi nume (chiar având nume şi atribute diferite) şi a căror lungime nu depăşeşte 64Ko. Sintaxa directivei este :

```
NUME_GRUP       GROUP   NUME_SEG1, NUME_SEG2, …
```

Exemplu:

```
DATA1   SEGMENT
Var1   db   10h
DATA1   ENDS
DATA2   SEGMENT
Var2   dw   20h
DATA2   ENDS
DATA   GROUP   DATA1, DATA2
       ; după această directivă GROUP, putem avea
       ; în segmentul de cod:        mov   ax, data
                                     mov   ds, ax
                                     mov   ax, OFFSET Var1
       ; echivalentă cu instrucţiunea
       ; mov ax, OFFSET data:Var1
       ; sau directive de forma:  ASSUME   ds : data
```

Utilizarea acestei directive reprezintă o altă modalitate de combinare a mai multor segmente pe lângă cea oferită de atributul PUBLIC.

Directiva **END** marchează sfârşitul logic al unui program. Ea apare obligatoriu în toate modulele. Eticheta este punctul de start după încărcarea programului : `END eticheta`

Contorul de locaţii indică offsetul curent, fiind accesibil prin $.

```
mesaj db "un mesaj"
lung_mesaj EQU $-mesaj
```

Exemplu:

```
sirAscii db "12345"           ;sir de octeţi: coduri Ascii ale
                              ;carac. "1","2","3","4","5".
lung_sir EQU $-sirAscii       ; valoarea $-nume variabila
               ;indică lungimea (asem. length) variabilei
```

Directiva **ORG** modifică explicit contorul de locaţii.

Exemplu: `ORG 100H.  ;asamblează la offsetul absolut 100h=256`

Structura unui program .COM :

```
PR_COM SEGMENT PARA PUBLIC 'CODE'
ORG 100h
ASSUME CS:PR_COM,DS: PR_COM, SS: PR_COM, ES: PR_COM,
Start: JMP inceput
       ; datele
inceput:
       ; proceduri
       ..
       MOV AX,4C00H
       INT 21H

PR_COM ENDS
       END Start
```

Obţinerea programului executabil (.com) pentru aplicaţii (medii) Microsoft şi Borland se face prin secvenţa de prelucrări de mai jos.

MASM PR_COM.ASM	**TASM PR_COM.ASM**
LINK PR_COM.OBJ	**TLINK /t PR_COM.OBJ**
EXE2BIN PR_COM.EXE	**PR_COM.COM**

Structura unui program .EXE :

```
STIVA  SEGMENT PARA STACK 'STACK'
       DW 512 DUP(?)
STIVA  ENDS
```

```
DATA   SEGMENT       PARA   PUBLIC 'DATA'
       ;
       ;DATE
       ;
DATA   ENDS
COD    SEGMENT PARA PUBLIC 'CODE'

       MAIN PROC FAR

       ASSUME CS:COD, DS:DATA, SS:STIVA,ES: NOTHING
       PUSH   DS
       XOR    AX,AX
       PUSH   AX
       MOV    AX,DATA
       MOV    DS,AX
       ...
       RET
       ;proceduri
MAIN   ENDP
       COD    ENDS
       END    MAIN
```

Obţinerea programului executabil (.exe) se face astfel:

MASM PROG_COM.ASM TASM PROG_COM.ASM
LINK PROG_COM.OBJ TLINK PROG_COM.OBJ

A4.3 Exerciţii şi teme

1. Cum se defineşte o variabilă “x” pe cuvânt, iniţializată cu 5?

2. Cum se defineşte un şir de 100 octeţi neiniţializaţi?

3. Cum se încarcă într-un registru adresa variabilei x? Dar a şirului definit anterior? Ce registru se foloseşte pentru adresare?

4. Se consideră o variabilă definită astfel: SIR db 1,2,3,4,5,6,7,8,9,0. Comentaţi instrucţiunea mov ax, word ptr SIR[2]. Ce se va încărca în registrul AX ?

5. Definiţi o constantă cu numele CR, având valoarea 0Dh.

6. Specificaţi ce va conţine registrul AL, în urma execuţiei instrucţiunii: mov al,Var1.

7. Care va fi valoarea lui B din expresia: B dw 6 LT 7 ?

8. Fie variabila: SIR dd 2,3,4,5,6,7. Specificaţi care vor fi valorile pentru length, size şi type.

9. Scrieţi instrucţiunile corespunzătoare următoarelor operaţii:

- copiere dată imediată 1200h în registrul segment DS
- continutul locatiei de memorie 10h să fie mutat în locaţia de memorie 20h
- conţinutul registrului segment DS să fie mutat în registrul segment ES.

10. Scrieţi un program complet care adună două numere, urmărind paşii:
    a. Definiţi o variabilă “a” pe un octet iniţializată cu 2;
    b. Definiţi o variabilă “b” pe un octet iniţializată cu 3;
    c. Definiţi o variabilă “sum” pe un octet neiniţializată;
    d. Scrieţi secvenţa care adună cele două numere şi depune rezultatul în “sum”;
    e. Completaţi toate aceste fragmente în modelul de structură a programului .exe şi salvaţi cu extensia .asm;
    f. Asamblare: ***tasm*** program.asm. Studiaţi opţiunile de asamblare: ***tasm ?***; creaţi fişier listing pentru programul scris;
    g. Linkeditare: ***tlink*** program.obj;
    h. Execuţie asistată: ***td*** program.exe.

11. Scrieţi un program complet care determină maximul dintr-un şir.
    a. Definiţi un şir de 10 octeţi iniţializaţi cu numere aleatoare;
    b. Definiţi o variabilă “maxim” pe octet;
    c. Scrieţi o secvenţă de program care determină maximul dintr-un şir;
    d. Scrieţi sursa (nume.asm);
    e. Asamblaţi aplicaţia; generaţi şi listingul;
    f. Linkeditaţi aplicaţia;
    g. Executaţi programul cu td.exe.

12. După modelul prezentat, scrieţi un program care determină minimul dintr-un şir de 10 numere.

13. Scrieţi un program care calculează suma elementelor unui şir de numere. Parcurgeţi şirul în două moduri: prin adresare bazată indexată şi folosind instrucţiuni specifice şirurilor.

A5. Dezvoltarea programelor în limbaj de asamblare (II)

A5.1 Definirea simplificată a segmentelor

Variantele mai noi ale asambloarelor au introdus o modalitate de definire simplificată a segmentelor, cu avantajul că se respectă acelaşi format (structura programului obiect) cu programele scrise în limbaje de nivel înalt. Utilizând definirea simplificată se vor genera segmente cu nume şi atribute identice cu cele obţinute cu compilatoarele pentru limbaje de nivel înalt.

Pentru specificarea tipului de memorie folosit se va folosi directiva **.MODEL** cu sintaxa dată mai jos:

. MODEL tip

unde tip poate fi: *tiny, small, medium, large, huge*. Semnificaţia lor este următoarea :

- *tiny* -toate segmentele (date, cod, stiva) se pot genera într-o zonă de 64Ko şi alcătuiesc un singur grup de segmente;
  -salturile, apelurile şi definiţiile de proceduri sunt toate NEAR
  -este utilizat la programele .COM;
- *small* -datele+stiva formează un segment iar codul un alt segment. Segmentele sunt <64Ko;
  -salturile, apelurile şi definiţiile de proceduri sunt toate NEAR
- *medium* -datele+stiva formează un segment <64Ko, iar codul poate fi în mai multe segmente separate (negrupate), deci >64Ko;
  -salturile şi apelurile şi definiţiile de proceduri sunt implicit de tip FAR;
- *compact* -datele şi stiva se găsesc în segmente separate (deci > 64Ko) iar codul grupat <64Ko;
  -salturile şi apelurile sunt de tip NEAR, iar datele sunt cu referinţă de tip FAR;
- *large* -datele şi codul >64Ko; nici o structură de date nu va depăşi 64Ko ;
- *huge* -datele şi codul >64Ko; nu sunt restricţii asupra structurilor de date;
  -datele, codul şi pointerii vor fi cu referinţă îndepărtată;

Sintaxele directivelor simplificate de definire a segmentelor sunt prezentate mai jos:

```
. stack dimensiune     ; implicit 512 octeţi pentru TASM
. code [nume]
. data
. data?          ; date neiniţializate în limbaj de nivel înalt
```

```
. fardata [nume] ; seg. de date utilizate prin adrese complete
                 ; in LNI
. fardata [nume]
. const          ; definire constante în LNI (HLL)
```

Dacă parametrul [nume] lipseşte se vor atribui nume implicite segmentelor generate:

```
TEXT                  pentru  .code (modele de cod mic)
Nume_fis_sursa_TEXT   pentru  (modele de cod mare)
DATA                  pentru  .data
_BSS                  pentru  .data?
CONST                 pentru  .const
STACK                 pentru  .stack
_FAR_DATA             pentru  .fardata
_FAR_BSS              pentru  .fardata?
```

O parte din segmente se grupează în mod implicit într-un grup numit DGROUP, după cum urmează:
- la modelul tiny intră toate segmentele
- la restul modelelor intră segmentele DATA, DATA?, CONST, STACK.

Există o directivă ASSUME implicită, de forma:

- la modele small/compact:

```
ASSUME cs:TEXT, ds:DGROUP, ss:DGROUP
```

- la modele large/huge:

```
ASSUME cs:nume_TEXT, ds:DGROUP, ss:DGROUP
;nume apare în directiva .code sau numele fis_sursa.asm
```

- la modelul tiny:

```
ASSUME cs:DGROUP, ds:DGROUP, ss:DGROUP.
```

Folosirea directivelor .fardata/ .fardata? cere utilizarea explicită a directivei ASSUME.

Accesul la adresele de început ale segmentelor sau grupurilor de segmente se face prin simbolurile globale: @data, @data?, @fardata, @fardata?, dgroup.

Iniţializarea regiştrilor segment DS şi ES la începutul unui program se poate face cu simbolul @data sau dgroup.

```
MOV AX,DGROUP ; SAU @DATA
MOV DS,AX
MOV ES,AX.
```

A5.2 Aplicaţie

Folosind definirea simplificată a segmentelor, în aplicaţia următoare este determinată suma a două numere

```
.MODEL SMALL
.STACK 200H
.DATA
        A DB 3
        B DB 5
        SUMA DB ?
.CODE
MAIN LABEL
        MOV AX,@DATA
        MOV DS,AX
        MOV BX, OFFSET A
        MOV AL, [BX]
        MOV BX, OFFSET B
        ADD AL,[BX]
        MOV BX, OFFSET SUMA
        MOV [BX],AL
        MOV AX,4C00h
        INT 21h
END MAIN
```

Listingul aplicaţiei:

```
Turbo Assembler         Version 4.1       04/22/04 10:14:02
Page 1
first1.ASM
      1 0000                        .MODEL SMALL
      2 0000                        .STACK 200H
      3 0000                        .DATA
      4 0000  03                            A DB 3
      5 0001  05                            B DB 5
      6 0002  ??                            SUMA DB ?
      7 0003                        .CODE
      8 0000                        MAIN LABEL
      9 0000  B8 0000S                      MOV AX,@DATA
     10 0003  8E D8                         MOV DS,AX
     11 0005  BB 0000R                      MOV BX, OFFSET A
     12 0008  8A 07                         MOV AL, [BX]
     13 000A  BB 0001R                      MOV BX, OFFSET B
     14 000D  02 07                         ADD AL,[BX]
     15 000F  BB 0002R                      MOV BX, OFFSET SUMA
     16 0012  88 07                         MOV [BX],AL
     17
     18                             END MAIN

Turbo Assembler         Version 4.1       04/22/04 10:14:02
Page 2
Symbol Table
```

```
Symbol Name                     Type   Value

??DATE                          Text   "04/22/04"
??FILENAME                      Text   "first1  "
??TIME                          Text   "10:14:02"
??VERSION                       Number 040A
@32BIT                          Text   0
@CODE                           Text   _TEXT
@CODESIZE                       Text   0
@CPU                            Text   0101H
@CURSEG                         Text   _TEXT
@DATA                           Text   DGROUP
@DATASIZE                       Text   0
@FILENAME                       Text   FIRST1
@INTERFACE                      Text   000H
@MODEL                          Text   2
@STACK                          Text   DGROUP
@WORDSIZE                       Text   2
A                               Byte   DGROUP:0000
B                               Byte   DGROUP:0001
MAIN                            Word   _TEXT:0000
SUMA                            Byte   DGROUP:0002

Groups & Segments      Bit Size Align  Combine Class

DGROUP                 GROUP
STACK                  16  0200 PARA    STACK   STACK
_DATA                  16  0003 WORD    PUBLIC  DATA
_TEXT                  16  0014 WORD    PUBLIC  CODE
```

ANEXA 6

Model Probleme Examen

1. Se dă un şir de 15 elemente, definite pe octet, în segmentul de date, format din litere mari majuscule) şi cifre zecimale. Să se genereze două şiruri: unul format din literele mari şi unul din cifrele şirului initial.

Se cere:

a. Definiţi, în formă prescurtată, segmentul de date care sa conţină: **1p**
-şirul initial (SIR_INIT) format din elementele:
2,‘F’,‘A’,3,9,4,‘T’,‘E’,‘P’,8,5,7,’W’,1,’J’

-cele doua şiruri destinaţie denumite CIFRE respectiv LITERE

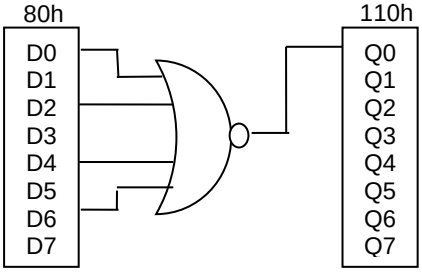
b. Să se scrie secventa de generare a celor două şiruri. **2p**

c. Să se afiseze pe randuri diferite, şrurile obţinute, folosind funcţii ale întreruperii BIOS int 10h. **2p**

Obs:
- pentru elementele definite sub forma ‘X’, se returnează codul ascii corespunzător caracterului;

- codurile ASCII ale cifrelor: 0….9 sunt 30h….39h
literelor mari: A....Z sunt 41h….5Ah

2. Să se scrie o secvenţă de program care emulează circuitul logic combinaţional prezentat în următoarea figură: **3p**



Soluţie probleme:

```
1.a)
;definirea sirului initial si a celor doua siruri destinatie
.data
    SIR_INIT   db     2,'F','A',3,9,4,'T','E','P',8,5,7,'W',1,'J'
    CIFRE      db    15 dup (?)
    LITERE     db    15 dup (?)

b)      mov cx, 15                      ;contor elemente sir
        mov si, offset SIR_INIT         ;adresa initiale siruri
        mov di, offset CIFRE
        mov bx, offset LITERE

et1:    mov al, [si]            ;pentru cifrele din SIR_INIT se
                                ;returneaza valoarea efectiva iar
                                ;pt litere codul ascii coresp.
        cmp al, 40h             ;fiecare element din sir va fi
                                ;comparat cu valoarea 40h.
        jb et2                  ;daca valoarea e mai mica decat
                                ;40h elementul repr. o cifra
        mov [bx] , al           ;daca valoarea e mai mare decat
                                ;40h elementul repr. o litera
        inc bx                  ;se depune majuscula in sirul
                                ;LITERE
        jmp et3
et2:    mov [di] , al           ;se depune cifra in sirul CIFRE
        inc di
        jmp et3
et3:    inc si                  ;se trece la urmatorul element
        dec cx                  ;procedeul se repeta pentru toate
                                ;elementele sirului
        jnz et1

c)                              ;pozitie cursor cifre:
        mov ah, 02h             ;se util functia 02h a int 10h
        mov bh, 0h
        mov dh,6                ;se alege randul si coloana
        mov dl,4
        int 10h
                                ; afisare cifre
        mov cx, length CIFRE
        mov di, offset CIFRE
        mov ah, 0eh             ;pt afisare functia 0eh a int 10h
afis1:  mov al, [di]
        add al,30h              ;se determina codul ascii
                                ;corespunzator cifrelor
        mov bh,0
        int 10h
        inc di
        dec cx
        jnz afis1
```

```
                                ;pozitie cursor litere:
        mov ah, 02h
        mov bh, 0h
        mov dh,7                ;se alege randul si coloana
        mov dl,4
        int 10h
                                ;afisare litere
        mov cx, length LITERE
        mov bx, offset LITERE
        mov ah, oeh
        afis2:  mov al, [bx]    ;elementele sirului LITERE sunt
                                ;reprezentate de codul ascii – nu
                                ;mai e necesara nici o conversie
        mov bh,0
        int 10h
        inc di
        dec cx
        jnz afis2

2.      mov dx, 80h             ;se citeste portul de intrare 80h
                                ;in registrul al
        in al,dx                ;se real functia logica and intre
                                ;octetul  citit  si  valoarea  55h
        and al,55h              ;55h=0101.0101b
        jz et                   ;daca valoarea obtinuta este 0
                                ;salt la eticheta et
        mov al,0000.0000b       ;daca valoarea obtinuta este 1
                                ;inseamna ca cel putin unul din
                                ;bitii D0,D2,D4,D6 a fost 1,
                                ;rezultatul fuctiei SAU negat e 0
        mov dx,110h             ;valoare ce se trimite la portul
        out dx,al               ;de iesire 110h
        jmp go_to
et:     mov al,0000.0001b       ;inseamna ca bitii D0,D2,D4,D6 au
                                ;fost toti 0, rezultatul fuctiei
                                ;SAU negat este 1
        mov dx,110h             ;valoare trimisa la portul 110h
        out dx,al
go_to:
        ……
```

Model de grila

1. La o operatie(instructiune) cu siruri, valorile registrelor SI si DI sunt incrementate sau decrementate functie de:
a. natura operatiei efectuata
b. valoarea flagului D
c. raportul in care se afla valorile lui SI si DI
d. valoarea flagului Carry

2. Care dintre functiile de mai jos apartin circuitului de ceas 8284?
a. genereaza ceasul procesorului
b. realizeaza sincronizarea RESET-ului
c. genereaza intreruperile
d. realizeaza sincronizarea semnalului READY

3. Care este avantajul folosirii moacro-urilor?
a. programul sursa este mai compact
b. programul executabil este mai scurt
c. programul se executa mai rapid
d. programul este mai inteligibil

4. Când un procesor 8086 execută instrucţiunea
MOV [SI], AL ; ce fel de ciclu de acces are loc?
a. scriere în memorie
b. scriere într-un port de ieşire
c. citire dintr-un port de intrare
d. citire din memorie

5. Care dintre instructiuni nu pot afecta flagul carry?
a. dec cx b. sub ax,dx c. not bx d. add dx,2

6. Care dintre caracteristicile de mai jos corespund PSP-ului (Program Status Prefix)?
a. are lungimea de 256 bytes
b.precede programul din memorie
c.contine rezultatele programului
d.poate contine informatii din linia de comanda

7. Procesorul 8086 in mod maxim :
a. utilizeaza circuitul 8288 BUS controller
b. are 8 semnale diferite fata de modul minim
c. adreseaza memorie mai mare
d. nu necesita semnal de ceas

BIBLIOGRAFIE

1. Irvine, Kip R. *Assembly language for x86 processors,* Prentice Hall, 2015
2. Lupu, E., s.a, *Initiere în limbajul de asamblare x86*, Lucrari practice, teste si probleme, Ed. Gutenberg, 2012
3. Lupu, E., s.a, *Elemente de programare în limbaj de asamblare x86*, Ed. Gutenberg, 2009
4. Abel, P. *IBM PC Assembly language and Programming,* Prentice Hall, 2001
5. Buchanan, W. *PC Interfacing, Communications and Windows Programing* Addison Wesley, 1999
6. https://emu8086.en.lo4d.com/
7. Lungu, V. *Procesoare Intel. Programare în limbaj de asamblare.* Ed. TEORA, 2004
8. Burileanu, C. şi col. *Microprocesorul x86 - o abordare software*, Ed. Albastră, 1999
9. Lupu, E., s.a. *Programare în limbaj de asamblare x86*, Risoprint, 2006
10. [***] www.intel.com
11. [***] www.softwareforeducation.com
12. [***] www.x86.org
13. Carter, P. A., PC assembly language, 2003,www.computer-books.us
14. [***] www.programmersheaven.com
15. [***] www.cpu-world.com