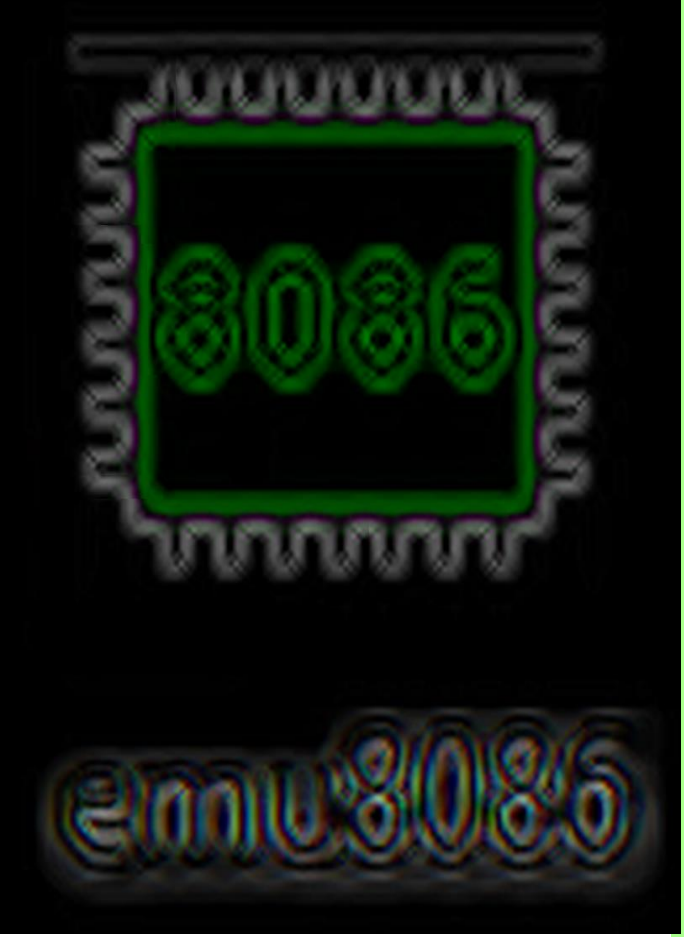
Aspecte de bază în programarea în limbaj de asamblare folosind SIMULATOR DE MICROPROCESOR 8086

2016



ANCA APĂTEAN

UT Press
Cluj-Napoca, 2016
ISBN 978-606-737-216-8

Editura U.T.PRESS
Str. Observatorului nr. 34
C.P. 42, O.P. 2, 400775 Cluj-Napoca
Tel.: 0264-401.999
e-mail: utpress@biblio.utcluj.ro
http://biblioteca.utcluj.ro/editura

Director: Ing. Călin D. Câmpean

Recenzia materialului a fost asigurată de:
Camelia Chira – Şef lucrări - UTCN
Adrian Sterca – Lector - UBB

ISBN: 978-606-737-216-8
Bun de tipar: 23.12.2016

Aspecte de bază în PROGRAMAREA ÎN LIMBAJ DE ASAMBLARE folosind

SIMULATOR DE MICROPROCESOR 8086

ANCA APĂTEAN
2016

ABSTRACT

Cartea prezintă aspecte de bază ale programării în limbajul de asamblare specific procesoarelor din familia x86 folosind un simulator: EMU8086.

Materialul se prezintă modular, începând cu noțiunile de bază necesare înțelegerii aspectelor arhitecturale ale procesorului; continuă cu aspecte despre reprezentarea informației în PC și apoi converge înspre aplicații (atât rezolvate cât și propuse, cu grade de dificultate variate).

Dedicație:

Dedic acest material fetiței mele, care la vârsta de 6 ani dă dovadă de înțelepciune, deșteptăciune, intuiție, curiozitate, empatie, bunătate, harnicie și frumusețe (de toate felurile). O minune de copil căruia îi mulțumesc că a acceptat să iau din timpul pe care ar fi trebuit să-l petrec cu ea și să scriu acest material.

A fost înțeleaptă și a spus că trebuie să-l scriu pentru oricine ar putea avea nevoie de el.

Anca Apătean

Prefaţă:

Materialul cuprins în această carte este structurat pe 3 părţi; acestea au vizat în special:

1. Prezentarea noţiunilor de bază în ceea ce priveşte reprezentarea informaţiei în PC şi arhitectura procesorului 8086, care se consideră o referinţă în domeniu. Înţelegerea acestor aspecte este esenţială în parcurgerea materialului care urmează în celelalte două părţi. Exerciţiile propuse la finalul părţii întâi vor să valideze aceste cunoştinţe.

Partea I:

1 – Introducere
2 – Reprezentarea informaţiei în PC. Conversii între diferite baze de numeraţie
3 – Arhitectura de bază a procesoarelor x86 pe 16 biţi
4 – Avantajele şi dezavantajele folosirii unui simulator de microprocesor
5 – Prezentarea simulatorului
6 – Conversii de numere şi operaţii de bază cu simulatorul

2. Prezentarea tuturor aspectelor specifice simulatorului în ceea ce priveşte dezvoltarea unei aplicaţii simple, indiferent că datele vor fi definite local, în cadrul programului sau că se vor prelua de la tastatură. Se introduc toate noţiunile de bază, inclusiv în ceea ce priveşte afişarea informaţiei pe ecranul simulatorului.

Partea II:

7 – Definirea şi reprezentarea datelor în simulator
8 – Lucrul cu memoria simulatorului
9 – Codificarea instrucţiunilor în simulator
10 – Scrierea unei aplicaţii simple

3. Prezentarea unor programe existente în cadrul simulatorului în scopul parcurgerii cât mai multor tipuri de programe şi deci posibile aplicaţii; pentru început, sunt prezentate aplicaţiile mai vizuale, cele care au o interfaţă definită în cadrul simulatorului prin intermediul unui port de intrare sau/şi de ieşire, iar apoi sunt prezentate aplicaţiile mai complexe, unele dintre ele destul de dificile. Fiecare program prezentat este abordat din prisma unui novice în domeniu, fiind un material cu specific (auto)didactic. Pe măsură ce se parcurge materialul şi se acumulează cunoştinţe şi deprinderi (prin rezolvarea exerciţiilor propuse) se vor putea rezolva inclusiv problemele de la finalul cărţii. Validarea corectitudinii acestora poate fi realizată individual, prin obţinerea efectului cerut în fiecare aplicaţie în simulator.

Partea III:

11 - Acomodarea cu simulatorul – aplicaţii simple
12 – Întreruperi şi macrouri
13 – Elemente de programare în limbaj de asamblare
14 – Exerciţii şi aplicaţii propuse
15 – Concluzii

Exerciţiile propuse sunt organizate în cadrul unui şablon care ajută la parcurgerea materialului, in ideea (auto)evaluării modului de rezolvare. Dacă în cadrul unei probleme există subpuncte *i), ii), iii), iv)* cu formă asemănătoare, se sugerează rezolvarea unui singur punct (*i)* de exemplu), celelalte fiind înrudite.

Fiind o carte adresată începătorilor în domeniul programării în limbaj de asamblare, am preferat execuția programelor pe un simulator, în locul execuției pe un procesor real. Aceasta, din considerente de simplitate în utilizare, dar și din motive de protecție a resurselor sistemului real. EMU8086 este mai mult decât potrivit pentru inițierea în programarea în limbajul de asamblare, fiind foarte ușor de utilizat. Acesta rulează programele ca un microprocesor 8086 real: codul sursă este asamblat și executat de către emulator pas cu pas, existând posibilitatea de a urmări modificările apărute în regiștri, în flag-uri și în memorie (chiar în timpul execuției programelor). În plus, EMU8086 pune la dispoziție o documentație complexă, împreună cu mai multe exemple de cod care pot fi executate fără nici un efort, chiar și de un începător în domeniu. În această carte, am încercat să încurajez învățarea prin descoperire; am prezentat într-o primă fază o posibilă aplicație și abia apoi am prezentat cunoștințele necesare obținerii acesteia (acolo unde a fost posibil acest lucru); astfel, cunoștințele se asimilează pornind de la practic înspre teoretic.

În **Partea I**, sau mai exact *primele 5 capitole* tratează aspecte generale (precum reprezentarea numerelor și conversii între diferite baze de numerație), dar și prezintă ***2 simulatoare (EMU8086 și SMS32v50)*** cu feresterele disponibile, fiind deci o parte de acomodare – atât cu noțiunile teoretice cât și cu cele de lucru cu simulatorul respectiv. În continuare, se recomandă urmărirea aspectele așa cum se prezintă mai jos:
În *Capitolul 6* se sugerează folosirea ambelor simulatoare pentru realizarea de *conversii de numere* între diferite baze de numerație, dar și *operații aritmetice de bază* cu acestea – precum adunări, scăderi, deplasări sau rotiri de biți, etc.

Partea a II-a se indică a fi parcursă doar cu simulatorul EMU8086, astfel: *Capitolul 7* – se concentrează pe *definirea și vizualizarea datelor* în simulator; *Capitolul 8* – se urmărește acomodarea și lucrul cu memoria simulatorului, în special în ceea ce privește *zona de date și de stivă*; *Capitolul 9* – se urmărește *înțelegerea modului cum sunt codificate instrucțiunile* în simulator în segmentul de cod; *Capitolul 10* – prezintă cea mai simplă modalitate de scriere a programelor în EMU8086, în vederea *dezvoltării primelor programe*; acestea vor ajuta la scrierea de secvențe de instrucțiuni care apoi pot fi testate.

Partea a III-a este dedicată ***aplicațiilor cu interfețe***, astfel că în *capitolul 11* se reiau unele aplicații simple și în simulatorul SMS32v50; se indică din nou folosirea ambelor simulatoare. Acestea au fost abordate din considerarea cât mai multor aplicații, cât mai vizuale și încurajarea cu înțelegerea și modificarea acestora. În acest fel, am considerat că instrucțiunile din setul 8086 vor fi mai bine înțelese și se vor reține pe termen mai lung.
În *Capitolul 12* se prezintă o scurtă parte teoretică, necesară explicării unora dintre noțiunile întâlnite în *Capitolul 11*. *Capitolul 13* are ca scop principal prezentarea tuturor aspectelor care ar putea încuraja scrierea de programe fără nici unul dintre simulatoare și deci execuția aplicațiilor pe un procesor real. *Capitolul 14* prezintă câteva probleme rezolvate și apoi mai multe seturi de probleme propuse a fi rezolvate, la unele oferindu-se chiar sugestii de rezolvare.

Partea I

Capitolele 1, 2, 3, 4, 5, 6

Capitolul 1. Introducere

Ce este un microprocesor? Care este rolul lui într-un sistem de calcul (SC)?

Microprocesorul (Unitatea centrală de procesare – UCP sau în engleză CPU de la Central Processing Unit) este elementul de bază al unui sistem de calcul (SC), fiind un cip complex, plasat de obicei pe placa de bază, în unitatea centrală a sistemului. Acesta, nu doar asigură procesarea datelor (interpretarea, prelucrarea şi controlul acestora), ci și supervizează transferurile de informaţii şi controlează activitatea generală a celorlalte componente care alcătuiesc SC.

În general, orice sistem de calcul conţine o parte hardware și una software. În acest volum, am insistat pe ***aspectele software ale UCP***: am abordat problema proiectării nu a cablajului, ci mai degrabă a aplicaţiilor posibile, în special considerând implicarea interacţiunii cu utilizatorul. Atunci când se dorește abordarea de acest gen (cu o implicare minimală a aspectelor hardware), în general se optează pentru folosirea unui simulator. ***Simulatorul EMU8086*** este un emulator de ***microprocesor 8086***, deci se vizează abordarea arhitecturii familiei de procesoare x86 pe 16 biţi. Simulatorul EMU execută *programele* pe o Maşină Virtuală, prin emularea hardware-ului real: ecranul monitorului, memoria şi dispozitivele de intrare/ ieşire, toate acestea pot fi utilizate, accesate sau vizualizate din EMU, ca dispozitive virtuale.

Un emulator este de fapt un software care *emulează* sau realizează o copie fidelă (un duplicat) al funcţiilor și facilităţilor oferite de un anumit hardware. Acest hardware emulat poate fi un întreg sistem de calcul sau o singură componentă, de exemplu un procesor din familia x86, mai exact procesorul 8086, cum este cazul de faţă. Această *emulare* sau asigurare a facilităţilor hardware-ului real prin software se realizează pe o mașină gazdă (adică un sistem de calcul) care poate fi foarte diferit din punct de vedere hardware de cel emulat. În cazul de faţă de exemplu, pe un sistem cu procesor Core i3 (procesor pe 64 biţi), a fost emulat sau simulat un procesor 8086.

Procesorul 8086 este procesor pe 16 biţi (proiectat de Intel) și este considerat baza unei întregi familii de microprocesoare (cea care a luat ulterior numele de *familia de procesoare x86)*. Astfel, pentru înţelegerea arhitecturii unui procesor și scrierea primelor programe într-un limbaj de asamblare, în această carte am ales un simulator pentru procesorul 8086. Aceasta, deoarece procesorul 8086 a constituit o referinţă în domeniu pentru procesoarele care i-au urmat, inclusiv cele din familia Pentium şi Athlon (chiar şi pentru cele actuale pe 64 biţi). Folosind simulatorul EMU8086 pot fi executate majoritatea instrucţiunilor Intel (pentru arhitectura de 16 biţi) şi chiar directive specifice MASM şi TASM.

EMU8086 suportă setul de instrucţiuni specific procesorului 8086 care stă la baza tuturor microprocesoarelor înrudite cu acesta și apărute ulterior (deci care fac parte din ***familia x86***). Scrierea de programe care să folosească astfel de instrucţiuni, obţinerea codului executabil și apoi încărcarea programului spre execuţie este mult facilitată cu ajutorul simulatorului.

Folosind un simulator de microprocesor 8086, se încurajează ***scrierea primelor programe în limbaj de asamblare***, fără riscuri pentru sistemul gazdă și fără a ţine cont de comenzile (uneori greu de stăpânit de începători) ce ar trebui date în vederea obţinerii și apoi execuţiei programului folosind un procesor real.

Ciclul complet de obţinere al unui fișier executabil în mod tradiţional plecând de la cod scris în limbaj de asamblare implică următoarele etape:

asamblare,
linkeditare,
depanare,
execuţie.

Acest ciclu poate fi parcurs:

1) ***prin intermediul unui emulator*** (de ex. *EMU8086*) și atunci programele vor fi adaptate arhitecturii emulate, în cazul de faţă a procesorului 8086; astfel, vor rezulta programe care folosesc regiștri pe 16 biţi (AX, BX, …);

2) ***direct cu procesorul real*** și atunci *p*rogramele pot fi adaptate arhitecturii CPU real: în prezent, de exemplu, se poate exploata la nivel de Core i3, i5, i7; astfel, vor rezulta programe care folosesc regiștrii pe 16 biţi (AX, BX, …), pe 32 biţi (EAX, EBX, …) sau chiar pe 64 biţi (RAX, RBX, …).

Modul cum se realizează programarea în limbaj de asamblare implicând un simulator în locul procesorului real, se realizează în mod nu tocmai asemănător cu ***abordarea curentă***, practicată în prezent. Atunci când vine vorba de folosirea limbajului de asamblare în industria informatică din prezent, în general, dintr-un limbaj de nivel înalt se accesează zone de cod în limbaj de asamblare, sau invers: se pot folosi diverse funcţii scrise în limbajele de nivel înalt (în combinaţie cu cod scris în limbaj de asamblare, așa cum procedează cei care dezvoltă compilatoare, asambloare, sisteme de operare sau care realizează depanare la nivel scăzut, etc.

Toate acestea sunt aspecte ce trebuie considerate atunci când alegem să folosim un simulator în locul procesorului real.

Capitolul 2. Reprezentarea informaţiei în PC

2.1. Aspecte generale de reprezentare a informaţiei

Cum se citește sau interpretează informaţia stocată în PC ?

În general, atunci când se face referire la un computer, calculator, sistem de calcul (SC) sau PC (personal computer) se înţelege un sistem capabil să prelucreze în mod inteligent informaţii. Aceste informaţii trebuie reprezentate (sau codificate) în interiorul PC-ului într-un anumit format, folosind anumite reguli şi convenţii bine stabilite. Totuşi, ***datele din PC pot avea semnificaţie diferită***: de exemplu, 43 poate reprezenta o valoare de temperatură, vârsta, o dimensiune sau un cod (codul ASCİİ al literei "C") – deci interpretarea informaţiei poate fi diferită. De asemenea, o valoare scrisă în binar de forma 10000001b poate desemna numărul 129 sau numărul -127, în funcţie de convenţia de reprezentare a numerelor: *fără semn* sau *cu semn*. Acestea sunt aspectele asupra cărora vom insista în cadrul acestui capitol.

Cum se reprezintă datele din PC? Ce sistem de numeraţie se folosește?

Oamenii folosesc în mod uzual ***sistemul de numeraţie zecimal*** pentru a se referi la date (sistem adoptat în principal datorită celor 10 degete de la mâini).

Analog, pentru folosirea datelor în PC, s-a căutat un sistem de reprezentare a informaţiei mai apropiat de situaţiile/ stările care apar într-un astfel de sistem și anume: ***trece*** sau ***nu trece*** curent printr-o porţiune de circuit. Astfel, datorită apariţiei circuitelor cu 2 stări stabile, s-a decis că cel mai potrivit este ***sistemul binar***, având cele 2 simboluri sau cifre binare: 0 şi 1.

Ce este bitul?

Cea mai mică unitate de informaţie o constituie ***bitul*** (în engleză se scrie şi se pronunţă "bit", la plural "bits" de la **b**inary dig**its**). În timp, s-au cunoscut mai multe forme de implementare practică a biţilor: plecând de la comutatoare, relee şi diode, tranzistoare, iar în prezent circuite integrate, toate s-au bazat pe acelaşi principiu:

- să permită trecerea curentului electric, analogie cu **bit = 1**,
- să blocheze trecerea curentului electric, analogie cu **bit = 0**,

întrucât bitul poate lua doar una din cele 2 valori: 0 sau 1 (nu permite/ permite).

Cu toate acestea, o singură cifră (0 sau 1) s-a constatat că deţine prea puţină informaţie pentru a fi utilă sub această formă singulară. Plecând tot de la sistemul zecimal, a apărut ideea de a reprezenta numerele binare prin *şiruri de biţi*, în analogie cu scrierea poziţională din sistemul zecimal, atât de uzuală, în care numerele sunt şiruri de cifre scrise secvenţial; exemplu: 4321.

Pentru noi, oamenii, este foarte cunoscută această scriere a numerelor folosind sistemul zecimal (am fost "antrenaţi" încă de timpuriu să-l folosim); numărul din exemplu, 4321, îl citim *patru mii trei sute douăzeci şi unu*; în schimb, acelaşi număr scris folosind valori de biţi 1000011100001 este dificil (dacă nu imposibil pentru majoritatea) de reţinut şi manevrat în limbaj uzual. Astfel, datorită numărului mare de cifre necesare reprezentării unui număr în sistemul binar, s-au căutat soluţii care să-i ajute pe oameni să interacţioneze cu sistemul binar și deci indirect cu SC.

Cum se pot grupa biţii pentru a fi mai ușor de manevrat?

În vederea simplificării modului de lucru cu şirurile de biţi, a apărut ideea grupării biţilor. Organismele autorizate au elaborat standarde în vederea dimensionării şirurilor de biţi, apărând astfel noţiunea de ***octet*** (în engleză "byte") care desemnează un grup de 8 biţi. Folosind octeţi, numărul scris în binar sub forma 1000011100001b se va scrie: ***00***010000 11100001b (biţii se grupează în octeţi întotdeauna pornind dinspre dreapta şi se completează – *biţii subliniaţi* - până la umplerea structurii de 16 biţi, cu valoarea bitului din extremitatea stângă, adică **0**).

Cum s-a ajuns de la 4321 la 01000011100001b?

Explicaţia se poate urmări în detaliu în secţiunea următoare; deocamdată reţinem simplu că se aplică o regulă: se efectuează împărţiri succesive la 2 şi se culeg resturile în ordine inversă obţinerii lor, primul bit de **0** apărând datorită semnului pozitiv al numărului scris în zecimal.

Cum se poate trece mai ușor de la sistemul binar la cel zecimal sau invers?

Pentru a face trecerea de la sistemul zecimal la cel binar şi invers, adesea se utilizează sistemul de ***numeraţie hexazecimal*** care e oarecum apropiat de ambele sisteme: permite exprimarea unor valori cu un număr rezonabil de cifre (asemănător sistemului zecimal) şi permite conversia relativ simplă în sistemul binar. Astfel, numărul din zecimal 4321 scris în binar 00**0**1000011100001b devine mult mai simplu de urmărit în hexazecimal: 10E1h, fiind *aproape* la fel de uşor de manevrat (de către noi, oamenii) ca şi cel scris în zecimal.

Cum s-a ajuns de la 0001000011100001b la 10E1h?

Tot în secţiunea următoare veţi putea găsi regulile de transformare între diferite baze de numeraţie. Scopul prezentului capitol este de a furniza noţiunile introductive despre reprezentarea numerelor în calculator, abordând câteva exemple simple la început şi regulile de conversie ale unui număr dintr-o bază în alta, ulterior acestea urmând a fi tratate mai în detaliu. Cel mai cuprinzător material din această primă parte este dedicat reprezentării numerelor și conversiei lor în diferite baze sau la diverse dimensiuni, înţelegerea acestora fiind esenţială pentru materialul din capitolele următoare.

Într-o anumită situaţie, care sistem de numeraţie e mai potrivit?

Nu se poate spune că există situaţii în care doar un anumit sistem de numeraţie e bun, iar altul nu. Este vorba doar de obișnuință: atunci când ne raportăm la conţinutul unor structuri din PC, de ex. regiştri sau zone din memorie, informaţia este prezentată în format hexazecimal, chiar dacă în computer informaţia respectivă este în binar. Aceasta, deoarece aşa cum am precizat mai devreme, este mai uşor pentru noi oamenii să lucrăm cu numere în hexazecimal decât cu numere în binar. Concret, programatorii în limbaj de asamblare preferă forma hexazecimală în locul celei zecimale (fiind mai apropiată de forma de reprezentare din calculator). Raţionamentul este simplu, așa cum arată Figura 2.1: noi oamenii gândim în zecimal datorită structurii corpului nostru, în timp ce SC „gândesc" în sistemul lor, adică în binar. Pentru a putea realiza comunicarea cât mai ușoară între cele 2 entităţi însă, uzual se folosește sistemul hexazecimal.

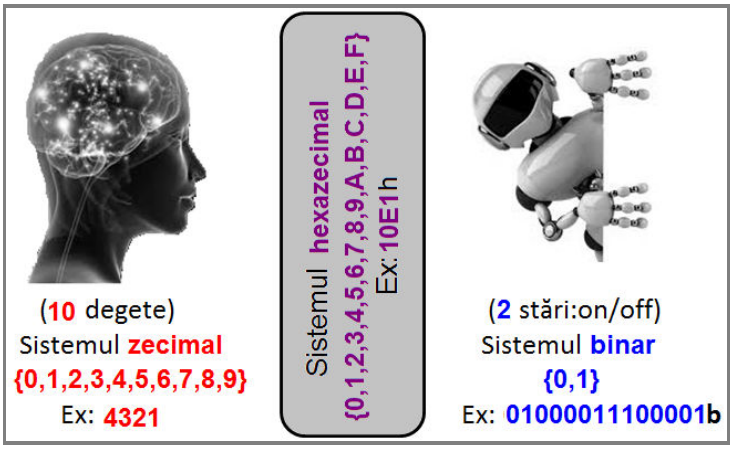


Figura 2.1 Legătura dintre sistemul zecimal, binar şi hexazecimal

Memoria PC-ului conţine doar numere sub forma unor secvenţe de 0 şi 1 şi din acest motiv, se spune că PC-urile stochează informaţia în format binar, nu zecimal. Aceste secvenţe sau înșiruiri de 0 și 1 sunt organizate sub formă de octeţi sau grupuri de octeţi (multiplu de 2, deci 2^x octeţi) în memorie sau în regiștrii interni ai procesorului.

Ce este octetul ?

Octetul (byte) este un şir de 8 biţi şi reprezintă un standard unanim respectat. Orice combinaţie de 8 biţi poate reprezenta un octet. De exemplu, 01010101b, 11110000b sau 00011101, etc.

Câte combinaţii (sau numere binare) putem scrie folosind un octet ?

Este ușor de observat că folosind ***un singur bit*** se pot reprezenta ***2 valori (0 şi 1)***, ***pe 2 biţi*** se pot reprezenta ***4 valori (00, 01, 10, 11)***, ș.a.m.d. Se deduce simplu că ***pe un octet*** se pot reprezenta ***2^8 numere***, adică ***256 valori diferite***, de exemplu numere întregi pozitive (naturale) de la 0 la 255.

Care sunt submultiplii octetului ?

Uneori se foloseşte noţiunea "***nibble***" (Figura 2.2) pentru un grup de doar 4 biţi; nibble-ul sau tetrada în română se folosește în special la exprimarea valorilor hexazecimale de către om, aşa cum se va vedea în continuare. Trebuie menţionat faptul că nu se folosește nibble-ul la accesarea informaţiei din memorie sau din regiștrii procesorului.

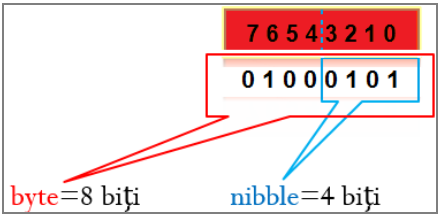


Figura 2.2 Reprezentarea nibble - submultiplu uzual al octetului

Cum sunt ordonaţi/ identificaţi biţii în cadrul unui octet ?

Numerotarea biţilor în oricare din aceste structuri începe de la bitul 0, desemnând bitul de pe poziţia cea mai din dreapta, numit și *cel mai puţin semnificativ bit* (LSb). Această numerotare este fictivă, doar că trebuie ţinut cont de ea.

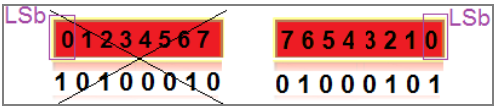


Figura 2.3. Scrierea corectă a biţilor în cadrul unui octet

Cum se reprezintă informaţia din memoria SC ?

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

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

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

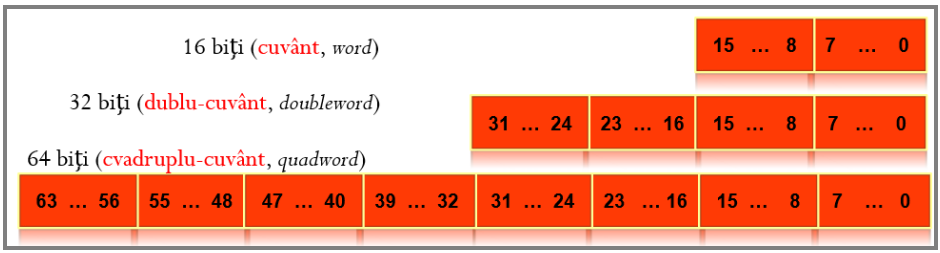


Figura 2.4 Multipli uzuali ai octetului: cuvânt, dublu-cuvânt şi cvadruplu-cuvânt

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

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

Operanzii folosiţi în PC pot avea şi dimensiuni mai mari, dar numai multipli de dimensiunea octetului şi în plus, aceştia sunt în general doar puteri ale lui 2:

2^0 octeţi = 1 octet, 2^1 octeţi = 1 cuvânt, 2^2 octeţi = 1 dublucuvânt, etc

Cum putem accesa informaţia din memorie?

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

Câţi octeţi putem accesa la un moment dat ?

Reprezentarea din Figura 2.5 se referă la modul cum se depune în memorie octetul ***21h***, cuvântul ***43 21h*** sau dublucuvântul ***87 65 43 21h*** începând de la adresa 126 și deci ocupând 1 locaţie, 2 locaţii sau 4 locaţii succesive.

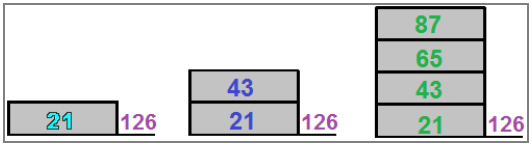


Figura 2.5 Octetul 21h, cuvântul 4321h și dublucuvântul 87654321h stocate în memorie începând de la adresa 126

Cum putem accesa octeţii din memorie folosind adrese ?

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

De exemplu, o instrucţiune de forma:

mov AL, [126] ; *va accesa octetul din memorie* de la *adresa 126* și îl va copia (sau muta după cum sugerează instrucţiunea ***mov***) în *registrul AL*, care așa cum vom vedea ulterior, este un registru pe 8 biţi al procesorului 8086. Similar,

mov AX, [126] ; va accesa *cuvântul* din memorie de la *adresa 126* și îl va copia în *registrul AX*, care este un registru pe 16 biţi al procesorului 8086.

Câţi biţi putem accesa minim din memorie ?

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

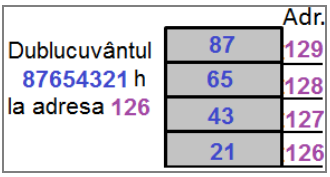


Figura 2.6 Dublucuvântul 87654321h stocat în memorie la adresele 129,...,126

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

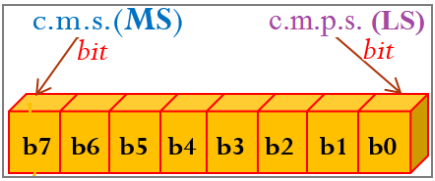


Figura 2.7 Numerotarea biţilor într-un octet: cel mai semnificativ (c.m.s.) şi cel mai puţin semnificativ (c.m.p.s.) *bit dintr-un octet*

Această specificare e importantă atunci când se consideră operaţiile logice la nivel de bit, precum cele de deplasare sau rotire deoarece atunci se dorește acces în interiorul octetului. Reamintesc aici că biţii nu-și modifică niciodată poziţia la accesare, bo rămâne LSb (în extrema dreaptă), iar b7 rămâne MSb (în extrema stângă) – revedeţi Figura 2.3. Această organizare a biţilor într-un octet poate fi generalizată şi asupra octeţilor (aşa cum arată Figura 2.8), a cuvintelor, etc.

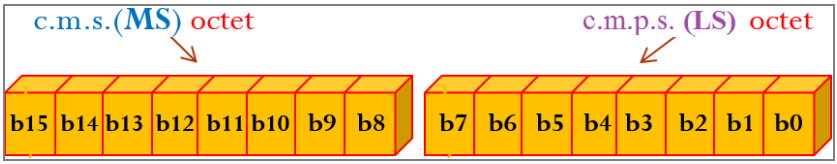


Figura 2.8 Cel mai semnificativ (c.m.s.) şi cel mai puţin semnificativ (c.m.p.s.) *octet dintr-un cuvânt*

În care sertar va ajunge sacoul ?

Revenind la analogia cu dulapul cu sertare, să presupunem că trebuie sa depozităm în acest dulap un costum din 2 piese: sacou şi pantaloni. În care sertar va ajunge sacoul: în cel de deasupra sertarului cu pantaloni, sau în cel de sub acesta? Este importantă această organizare deoarece la un moment dat am putea ruga pe cineva să ne livreze conţinutul sertarului 5 de exemplu (fiind siguri că acolo e sacoul).

Această problemă se pune și în cazul organizării informaţiei în memorie. De exemplu, cuvântul din Figura 2.8, cum se va aranja în memorie? Octetul c.m.s. deasupra celui c.m.p.s. sau invers?

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

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

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

Dublu-cuvântul 87654321h se depune în memorie folosind una din convenţiile:
- Little Endian: octetul **LSB**, adică cel de la sfârşitul structurii ("**END**"-ian) se depune în memorie la locaţia cu **adresa cea mai mică ("Little")** – această convenţie este specifică procesoarelor din familia *Intel* – Figura 2.9a);
- Big Endian: octetul **LSB** se depune în memorie la locaţia cu **adresa cea mai mare ("Big"),** convenţia fiind specifică procesoarelor din familia *Motorola* – Figura 2.9b).

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

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

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

Deşi pare foarte simplu, trebuie acordată o deosebită atenţie la ilustrarea schematică a conţinutului memoriei în acest caz. În Figura 2.10 puteţi observa un *octet la adresa 118*, un *cuvânt la adresele 122-123* şi un *dublucuvânt la adresele 126-129*.

Conţinutul memoriei este identic în cele 2 cazuri prezentate în figură, doar că în partea stângă (Figura 2.10a)) s-a desenat zona de memorie de la adrese mai mari înspre adrese mai mici, iar în partea dreaptă (Figura 2.10b)) este invers.
Desenul este valabil pentru un procesor din familia Intel, de exemplu, deci care respectă formatul ***Little Endian***.
Se recomandă utilizarea desenului cu ***adrese crescătoare*** întrucât şi în cadrul simulatorului EMU8086 acestea sunt prezentate sub această formă. Recomandarea ţine mai mult de practică, pentru a nu greşi în interpretarea datelor din memorie.

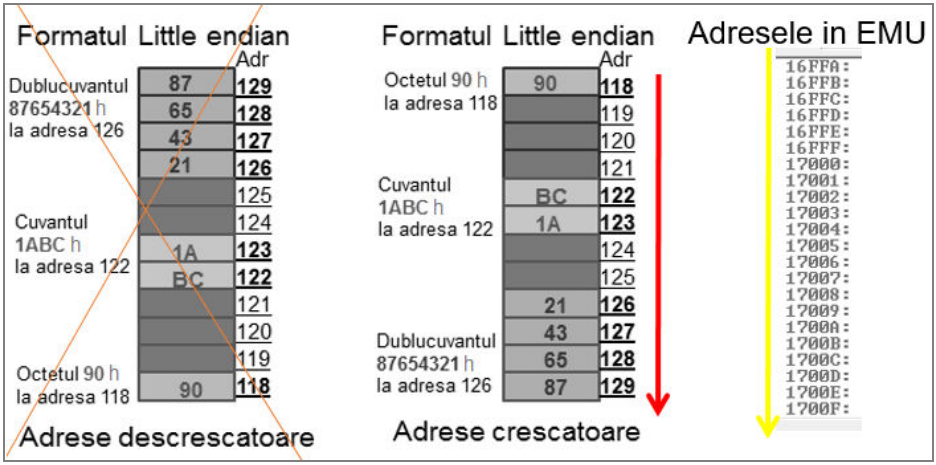


Figura 2.10 Formatul Little Endian ilustrat la adrese
a) descrescătoare sau b) crescătoare

2.2. Reprezentarea informaţiei numerice

2.2.1. Sisteme de numeraţie

Informaţia numerică sau numerele din PC se reprezintă prin intermediul unui ***sistem de numeraţie***, adică un ansamblu de reguli folosit pentru scrierea numerelor cu simboluri (numite *cifre).*

Se numeşte *bază* sau *rădăcină* (*q*) a sistemului de numeraţie numărul de simboluri permise pentru reprezentarea unei cifre. Sistemele de numeraţie pot fi:

- *nepoziţionale*: sistemul roman cu cifrele I, V, X, L, C, D, M
  -> 1, 5, 10, 50, 100, 500, 1000;
- *poziţionale*: sistemul arab (zecimal), binar, octal, hexazecimal
  -> 4321 = 4000 + 300 + 20 + 1.

Ce sistem de numeraţie se foloseşte la scrierea valorilor din PC ?

La scrierea numerelor în contextul reprezentării informaţiei în PC se folosesc **sisteme de numeraţie poziţionale**. Dintre acestea, cele mai uzuale sunt:

- sistemul **zecimal** – folosit de oameni în exprimarea numerelor,
- sistemul **binar** – folosit pentru exprimarea valorilor în calculator,
- sistemul **hexazecimal** – folosit în scrierea mai rapidă a valorilor din calculator,
- sistemul **octal**.

Tabelul 2.1. Baza şi cifrele sistemelor de numeraţie binar, zecimal, hexazecimal şi octal

Sistem de numeraţie	Baza (q)	Cifrele sistemului de numeraţie
Binar	2	{0,1}
Zecimal	10	{0,1,2,3,4,5,6,7,8,9}
Hexazecimal	16	{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}
Octal	8	{0,1,2,3,4,5,6,7}

Sistemul zecimal este cel mai folosit sistem în limbajul uzual. Pentru reprezentarea unui număr în zecimal pot fi folosiţi cei 10 digiţi (cifrele 0-9), fiecare digit al unui număr având asociată o putere a lui 10 în funcţie de poziţia sa.

Similar, ***sistemul binar*** este cel mai potrivit pentru reprezentarea valorilor numerice într-un PC: pentru a reprezenta un număr în binar, se folosesc doar două cifre, 0 şi 1, fiecare cifră a unui număr având asociată o putere a lui 2 în funcţie de poziţia sa.

Sistemul hexazecimal este folosit pentru reprezentarea prescurtată a numerelor binare. Pentru reprezentarea unui număr în hexazecimal se folosesc cifrele de la 0 la 9, iar ca cifre suplimentare se folosesc caracterele A – F, toate acestea desemnând mulţimea resturilor modulo 16. Numerele hexazecimale care încep cu literă vor fi precedate de 0, de exemplu, numărul A2h se va scrie 0A2h (pentru a nu fi confundate cu nume de etichete, detalii în *Capitolul 6*).

Care sistem de numeraţie îl folosim când interacţionăm cu un SC?

La reprezentarea sau utilizarea informaţiei din PC de către oameni, în general se foloseşte sistemul hexazecimal pentru exprimarea valorilor, fiind un sistem mai apropiat de cel zecimal (oamenii s-au obişnuit cu el) decât cel binar.

Relaţia între cele 3 sisteme de numeraţie este una foarte simplă: noi oamenii ne *gândim* la *valorile numerice în sistemul zecimal* (că așa ne-am obișnuit de mici), *scriem* valorile *în hexazecimal* în cadrul programelor care folosesc structuri ale procesorului, dar *PC-ul folosește reprezentarea binară*. Astfel, indiferent că e nevoie să programăm procesorul sau doar să consultăm valorile regiștrilor lui, trebuie să putem trece de la un sistem de numeraţie la altul cât mai ușor și rapid.

După parcurgerea secţiunii 2.2.3 și 2.2.4, reveniţi la Figura 2.11 și marcaţi operaţiile de conversie pe măsură ce vi le-aţi însușit.

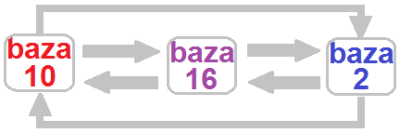


Figura 2.11 Conversii posibile între cele 3 baze de numeraţie

Cum putem trece ușor dintr-un sistem de numeraţie în altul?

Într-o primă fază, până ne acomodăm cu regulile de conversie, putem folosi un tabel (Tabelul 2.2), care arată exemple de numere scrise în cele 4 sisteme de numeraţie.

Tabelul 2.2. Numere scrise în sistemul zecimal, binar, hexazecimal şi octal

Sistemul zecimal q=10	Sistemul binar q=2	Sistemul hexazecimal q=16	Sistemul octal q=8
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

O altă posibilă variantă pentru a trece din hexazecimal în binar este prin folosirea șablonului din Figura 2.12, unde **b3**,**b2**,**b1**,**b0** sunt cifre binare (0 sau 1) care se ponderează cu puterile corespunzătoare ale lui 2, adică **8**, **4**, **2**, respectiv **1**. Acest șablon însă funcţionează doar pentru o cifră hexazecimală, deci numere care se scriu cu maxim 4 biţi.

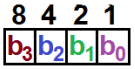


Figura 2.12 Șablon posibil la conversia din zecimal în hexazecimal sau invers

Exemplu: Dacă vrem să transformăm numărul **1010**b în hexazecimal, după ponderarea cu puterile lui 2 corespunzătoare sau aplicarea șablonului, vom obţine un opt plus un doi care este zece și care se scrie A în hexazecimal. Invers, dacă de exemplu vrem să vedem cum se scrie numărul D din hexazecimal în binar, va trebui să-l scriem în zecimal ca 13 și să-l descompunem în **8+4+1**, adică un opt, un patru și un unu care se scrie în binar: **1101**b.

1010b = **1*8** + **1*2** = **8** + **2**=10=Ah; Dh=13 = **8** + **4** + **1** = 1101b

Ce fel de numere vom întâlni în simulator ?

Atunci când lucrăm cu valori de adrese (deci care desemnează locaţii din memorie), acestea nu pot fi decât numere naturale pozitive (nu putem merge în memorie la adresa -1). La numerele naturale ne mai referim simplu ca "numere fără semn". În schimb, valorile care pot să apară în regiștrii procesorului 8086 pot fi și numere întregi cu semn + sau - și de aceea ele se vor numi "numere cu semn". Simulatorul EMU8086 suportă toate aceste tipuri de numere, atât fără semn (N) cât și cu semn (Z).

După cum știm din clasele primare, există și numere fracţionare; acestea însă nu sunt suportate în simulator și de aceea vom parcurge doar noţiunile de bază ale lor.

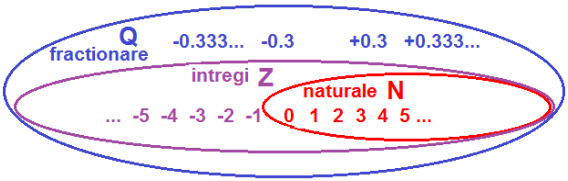


Figura 2.13. Ilustrarea domeniului numerelor suportate în simulator

2.2.2. Clasificarea numerelor fără semn vs. cu semn

Sistemul zecimal are cifrele 0...9, iar numerele pot fi scrise fără semn (se consideră implicit pozitive) sau cu semn având semnul **+** sau **–** specificat înaintea numărului (pentru a desemna numere pozitive, respectiv negative).

Exemplu**:* Numărul scris sub forma **4** se va considera ***fără semn, în timp ce numărul scris sub forma **+4** va fi considerat pozitiv, iar cel **-4** negativ, ultimele două fiind deci considerate implicit în convenţia de reprezentare a numerelor ***cu semn***.

Similar sistemului zecimal, numerele folosite în PC pot fi considerate/ **scrise *în binar fără semn*** sau ***cu semn***. În cazul numerelor *cu semn*, se utilizează în general, o cifră suplimentară pentru a indica semnul, aceasta fiind reprezentată pe poziţia c.m.s. a numărului, în analogie cu scrierea folosită de oameni pentru marcarea semnului unui număr scris în zecimal. Convenţional, la scrierea numărului *cu semn,* **în binar** se atribuie ***cifra 0 pentru semnul plus*** şi ***cifra 1 pentru semnul minus***. Trebuie subliniat faptul că în nici un alt sistem de numeraţie (binar, hexazecimal, octal) înafară de cel zecimal nu există simbolul „-" sau „+" care să desemneze semnul unui număr.

La ce se referă "gama numerelor" ?

Pentru a se evita ambiguităţile, la reprezentarea numerelor *cu semn* este obligatoriu să se ţină cont de ***gama numerelor***. Reamintesc aici că folosind un număr de 8 biţi se pot reprezenta 2^8=256 valori diferite. În Figura 2.14 s-a ilustrat grafic *gama numerelor* pe sistemul de axe, cu poziţionarea celor 256 valori diferite ***fără semn*** vs. ***cu semn*** care se pot scrie ***folosind un octet***.

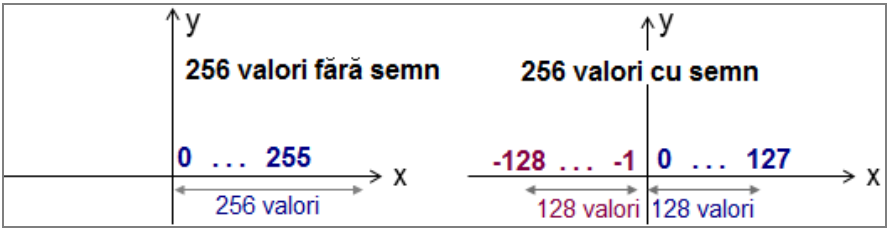


Figura 2.14. Reprezentarea a 256 valori fără semn vs. cu semn

Gama numerelor se referă la domeniul numerelor ce se pot reprezenta pe un număr fix de biţi; astfel, se poate deduce că ***folosind n biţi***,

numerele ***fără semn*** care se pot reprezenta sunt în **gama [0;2^n-1],** iar
numerele ***cu semn*** care se pot reprezenta sunt în **gama [-2^{n-1};+2^{n-1}-1]**.

Exemplu*:* Cu 16 biţi se pot reprezenta 65.536 valori diferite, cuprinse în gama de valori [0; 65.535] pentru numere fără semn sau [-32.768; +32.767] pt numere cu semn.

Presupunând că se poate deduce gama numerelor, cum se poate găsi numărul de biţi necesar pentru acea gamă? Este uşor de remarcat că nu trebuie decât să facem legătura cu puterea lui 2 corespunzătoare unui capăt sau celuilalt.

Exemplu*:* Aflaţi numărul minim de biţi necesar reprezentării corecte a numărului 3. Răspuns: Fiind un număr fără semn, se va utiliza relaţia corespunzătoare, încadrându-se numărul 3 în gama [0;3]; astfel, din relaţia 2^n-1=3 va rezulta că numărul minim de biţi necesar scrierii corecte a lui 3 este n=2. Într-adevăr, 3 se va scrie corect în binar ca 11b. În schimb, dacă enunţul s-ar fi referit la numărul +3, atunci acesta ar fi trebuit încadrat în gama [-4,+3] şi s-ar fi folosit relaţia $+2^{n-1}$-1=+3 sau -2^{n-1}= -4 de unde rezultă n=3. Astfel, avem nevoie de un bit suplimentar pentru a reprezenta corect numărul cu semn +3, iar acest bit suplimentar va fi 0 deoarece numărul este pozitiv. Nu la fel de uşor stau lucrurile pentru reprezentarea numerelor negative, însă se vor da explicaţiile necesare ulterior, în secţiunile următoare.

Cum ne dăm seama ce semn are un număr ?

Exemple de numere cu semn pe 8 biţi, ***pozitive*** sunt: 05h, 0Ah, 10h, 2Eh, 7Fh și
negative: 80h, 9Dh, A7h, FFh, etc.

Astfel, dacă un număr scris în hexazecimal are **cifra c.m.s. între 0 și 7**, atunci el este **pozitiv**, iar dacă această cifră este **între 8 și Fh**, atunci numărul este **negativ**.
Regula similară pentru un număr scris în binar: dacă un număr scris în binar are **MSb în 0**, atunci el este **pozitiv**, iar dacă **MSb al lui este în 1**, atunci numărul este **negativ**.

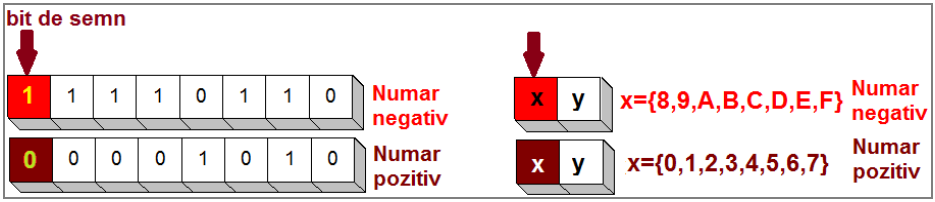


Figura 2.15. Reprezentare sugestivă a numerelor pozitive și negative
a) în binar și b) în hexazecimal (pe 8 biţi)

Câte numere pozitive/ negative putem scrie folosind un anumit număr de biţi ?

Așa cum reiese și din Figura 2.14, folosind acelaşi număr de biţi, în reprezentarea numerelor fără semn se pot scrie de două ori mai multe numere pozitive decât în cea cu semn. Problemele apar atunci când valorile depăşesc domeniul de reprezentare, de exemplu dacă adunăm 1111+1=0000b în loc de **1**0000b, dar aceste situaţii sunt uşor detectate de către programatori şi pot fi gestionate în vederea evitării posibilelor erori.

2.2.3. Conversii simple de numere dintr-o bază în alta

În secţiunile următoare vor fi tratate pe larg conversiile de numere, atât *fără semn* cât şi *cu semn*, şi în plus vor fi aplicate atât numerelor întregi cât şi celor fracţionare; în secţiunea de faţă, pentru simplitate, mă voi rezuma la câteva exemple simple de ***conversii de numere întregi fără semn***. Se vor parcurge exemple pentru fiecare situaţie posibilă, astfel încât să se poată converti un număr în orice bază dorită.

Cum distingem baza în care s-a reprezentat un număr ?

Așa cum am văzut deja, pentru a distinge numerele scrise în diferite baze de numeraţie, la sfârşitul numărului se adaugă o literă ce simbolizează baza, astfel:

- B sau b pentru numerele scrise în binar (baza 2),
- Q, O, q sau o pentru numerele scrise în octal (baza 8),
- D sau d pentru numerele scrise în zecimal (baza 10) sau nimic,
- H sau h pentru numerele scrise în hexazecimal (baza 16).

De regulă, numerele scrise în baza 10 nu trebuie marcate deoarece această bază se consideră implicită. Există şi alte moduri de notare, cum ar fi scrierea la sfârşitul numărului în paranteză a bazei: 10100001(2), 123AB(16) sau cu indice jos 10100001_2, $123AB_{16}$ ultima modalitate nefiind recomandată.

Care sunt regulile de conversie dintr-o bază în alta ?

(h->b) Conversii din hexazecimal în binar

Exemplu: Utilizând Tabelul 2.2, este uşor de observat că numărul **9A**h scris în hexazecimal se va transforma în binar în şirul de cifre binare: **1001 1010**b. Astfel, orice cifră hexazecimală va fi înlocuită cu un şir de 4 cifre binare corespunzătoare.

9Ah = **9 A** h = **1001 1010**b = 10011010b

(b->h) Conversii din binar în hexazecimal

Exemplu: La conversia în sens invers, un număr scris în binar ce se doreşte a fi transformat în hexazecimal, se procedează în felul următor: se formează grupuri de câte 4 cifre binare pornind dinspre c.m.p.s. bit, deci din dreapta spre stânga, înlocuind fiecare grup cu cifra hexazecimală corespunzătoare. Astfel, 11010b se va scrie 1Ah.

11010b =**000**11010b = **0001 1010** b = **1 A** h,

unde s-au adăugat 3 biţi de 0 pentru a forma un grup complet de 4 biţi.

(d->b) Conversii din zecimal în binar

Metoda 1: Se împarte în mod repetat numărul din zecimal la baza 2 până la obţinerea câtului 0 şi se culeg resturile obţinute în ordine inversă.

Exemplu: Conversia numărului 37 din zecimal în binar folosind metoda 1 (baza=2)
Sunt posibile 2 metode de scriere:

Forma de scriere I :

37 : 2 cât 18 rest **1**
18 : 2 cât 9 rest **0**
9 : 2 cât 4 rest **1**
4 : 2 cât 2 rest **0**
2 : 2 cât 1 rest **0**
1 : 2 cât 0 rest **1**

=> 37 = **1 0 0 1 0 1** $_b$

Forma de scriere II :

37| 2
36| 18| 2
= **1**| 18| 9| 2
= **0**| 8| 4| 2
=**1**| 4| 2| 2
=**0**| 2| 1| 2
=**0**| 0| 0
=**1**|

Exemplu: 4321:2 = 2160 (reţinem restul **1** - va fi bitul b0), apoi 2160:2=1080 (reţinem restul **0** - va fi bitul b1), apoi 1080:2=540 (reţinem restul **0** - va fi bitul b2), ș.a.m.d. iar când se obţine câtul 0 ne oprim și scriem aceste resturi în ordine inversă:

4321=0001000011100**001**b

Metoda 2: Se va căuta puterea cea mai mare a bazei care se poate scădea din numărul de reprezentat şi se efectuează scăderea, notând puterea respectivă. Există posibilitatea de a fi necesare mai multe scăderi ale aceleiaşi puteri, iar astfel se va nota multiplul puterii. Operaţia se repetă până la obţinerea diferenţei 0, noua valoare a descăzutului fiind numărul rămas după efectuarea scăderii.

Exemplu: Reprezentarea numărului 37 în binar folosind metoda 2 (baza=2):

Avem puterile lui 2 <=37: $\{2^5=32, 2^4=16, 2^3=8, 2^2=4, 2^1=2, 2^0=1\}$

37 - **32** = 5 (se notează $\mathbf{2^5}$ o dată)

5 - **4** = 1 (se notează $\mathbf{2^2}$ o dată)

1 - **1** = 0 (se notează $\mathbf{2^0}$ o dată) =>

2^5	2^4	2^3	2^2	2^1	2^0
1	0	0	**1**	0	**1**

pe poziţiile 5, 2, 0 se va pune (în general multiplul puterii) cifra 1, iar în rest cifra 0.

(b->d) Conversii din binar în zecimal

Dacă se consideră numărul reprezentat ca un număr fără semn, algoritmul e simplu: se înmulţesc cifrele cu puterile corespunzătoare ale bazei, în acest caz ale lui 2.

Exemplu: Reprezentarea numărului 4321 în binar, ca număr fără semn este:

	12	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1000011100001b =	**1**	0	0	0	0	**1**	**1**	**1**	0	0	0	0	**1**b=

$= 1*\mathbf{2^{12}}+1*\mathbf{2^7}+1*\mathbf{2^6}+1*\mathbf{2^5}+1*\mathbf{2^0} = 4096+128+64+32+1 = 4321$

Exemplu: Reprezentarea numărului 100101b din binar în zecimal, știind că este scris ca număr fără semn:

$100101_b = 1*2^5 + 0*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0$

$= 32 + 4 + 1 = 37$

În schimb, dacă se consideră că este vorba despre reprezentarea numărului + 37 în binar, știind că +37 este scris ca număr cu semn, atunci corect s-ar fi scris în binar:

$0100101_b = \mathbf{0*2^6} + 1*2^5 + 0*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0$

$= \mathbf{0} + 32 + 4 + 1 = + 37$

Dacă se dorea obţinerea numărului -37 atunci numărul în binar s-ar fi scris:

$1011011_b = \mathbf{-1*2^6} + 0*2^5 + 1*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0$

$= \mathbf{-64} + 16 + 8 + 2 + 1 = - 37$

Aceste aspecte vor fi reluate în secţiunea următoare și acolo se vor da mai multe detalii legate de conversia numerelor *cu semn*.

(h->d) Conversii din hexazecimal în zecimal

La conversia numerelor fără semn, se procedează asemănător conversiei din binar în zecimal, doar că puterile considerate vor fi ale bazei 16:

Exemplu: $\mathbf{10E1}h=\mathbf{1*16^3}+\mathbf{0*16^2}+\mathbf{14*16^1}+\mathbf{1*16^0}=\mathbf{4096}+\mathbf{0}+\mathbf{224}+\mathbf{1}=4321$

iar pentru reprezentarea numărului 37 din hexazecimal în zecimal vom avea:

$\mathbf{25}h = \mathbf{2*16^1} + \mathbf{5*16^0} = \mathbf{32} + \mathbf{5} = 37$

Specificarea unui număr (în reprezentarea cu semn) ca fiind pozitiv sau negativ este ușor de observat dacă acesta este scris în binar (Figura 2.10). Astfel, dacă numărul este scris în hexazecimal, se recomandă trecerea prin binar înainte de a decide dacă este pozitiv sau negativ.

(d->h) Conversii din zecimal în hexazecimal

Se pot adapta cele 2 metode de conversie aplicate la conversia din zecimal în binar, dar în general se alege metoda 1: se împarte numărul în mod repetat la 16 şi se culeg resturile; o altă metodă, a III-a ce se poate folosi în acest caz, considerată în general și cea mai simplă, este cea cu trecere prin binar: se transformă numărul din zecimal în binar şi apoi formând grupuri de câte 4 biţi se scriu cifrele hexa corespunzătoare.

Metoda 1. Folosind metoda cu împărţire la bază vom avea:

Exemplu: Conversia numărului 37 din zecimal în hexazecimal (baza=16)
37: 16 -> cât 2 rest **5**
2: 16 -> cât 0 rest **2** -> 37 = 32 + 5 = **2***16^1 + **5***16^0 = **2 5** $_h$

Exemplu: pentru un număr mai mare se va proceda similar:
4321:16= 270 (reţinem restul **1** - va fi cifra hexa de ordin 0), apoi 270:16=16 (reţinem restul **14=E** - va fi cifra hexa de ordin 1), apoi 16:16=1 (reţinem restul **0** - va fi cifra hexa de ordin 2), și în final 1:16=0 rest **1** – aceasta va fi cifra hexa de ordin 3; s-a obţinut câtul 0, deci se scriu aceste resturi în ordine inversă: 4321=**10E1**h

Metoda 2. Folosind metoda cu scădere a bazei la diferite puteri vom avea:

Exemplu: Reprezentarea numărului 37 în hexazecimal (baza=16).
37 -16 = 21 (**16^1**)
21 -16 = 5 (**16^1**)
5 -1 = 4 (**16^0**)
4 -1 = 3 (**16^0**)
3 -1 = 2 (**16^0**)
2 -1 = 1 (**16^0**)
1 -1 = 0 (**16^0**) => **2 5** h

Metoda 3: Cu trecere prin binar vom avea:
Exemplu: 4321=0001000011100001b = **0001 0000 1110 0001**b = **10E1**h

(nr <-> q) Conversia numerelor din/ în octal

Se procedează în mod similar conversiei din/ în hexazecimal, singura diferenţă fiind la trecerea din binar în octal sau din octal în binar, unde în locul formării de grupuri de 4 biţi, se vor forma grupuri de câte 3 biţi şi se va folosi cifra în octal corespunzătoare; astfel, în loc de tetrade se folosesc triade. Se va ţine cont de asemenea de baza 8 a sistemului şi de cifrele octale asociate.

Exemplu: 4321:8= 540 (reţinem restul **1** - va fi cifra octală de ordin 0), apoi 540:8=67 (reţinem restul **4** - va fi cifra octală de ordin 1), apoi 67:8=8 (reţinem restul **3** - va fi cifra octală de ordin 2), ș.a.m.d. iar când se obţine câtul 0 se scriu aceste resturi în ordine inversă: 4321=**0**10341q sau dacă se dorește folosirea metodei *cu trecere prin binar*:
4321=0001000011100001b =**0 001 000 011 100 001**b = **0**10341q

2.2.4. Conversii de numere cu semn

Reamintesc din secţiunea 2.2.2. că folosind **n biţi**, dacă se consideră numerele *fără semn*, gama numerelor este: $\mathbf{0 \div 2^n-1}$, iar dacă se consideră numerele *cu semn*, atunci gama numerelor este: $\mathbf{-2^{n-1} \div +2^{n-1} -1}$. Cele 2^n *numere reprezentabile pe n biți* formează *inelul claselor de echivalență a resturilor modulo* 2^n. Ca regulă generală, este mai mult decât indicată verificarea numărului de biţi necesar scrierii corecte a unui număr, întotdeauna înainte de efectuarea operaţiei efective de conversie din zecimal.

Exemplu: Așa cum am văzut în secţiunea anterioară, reprezentarea numărului întreg **37** ca număr fără semn, scris în binar este:

$37= 1*2^5 + 0*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = \mathbf{1\ 0\ 0\ 1\ 0\ 1}_b$

Dacă este vorba despre numărul cu semn **+37**, atunci reprezentarea corectă se va obţine prin adăugarea unui bit suplimentar de 0, astfel:

$+37= \mathbf{0}*2^6 + 1*2^5 + 0*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = \mathbf{0\ 1\ 0\ 0\ 1\ 0\ 1}_b$

iar pentru a se obţine numărul **-37**, numărul în binar se va scrie[1]:

$-37 = \mathbf{-1}*2^6 + 0*2^5 + 1*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = \mathbf{1\ 0\ 1\ 1\ 0\ 1\ 1}_b$

Care este MSb la numerele reprezentate în convenţia cu semn ?

La scrierea numerelor cu **semn pozitiv/ negativ**, se mai adaugă un bit suplimentar pt specificarea semnului. Astfel, bitul de semn va fi bitul c.m.s. (MSb) al reprezentării, având valoarea "0" dacă numărul este pozitiv sau "1" dacă numărul este negativ.

Care sunt pașii pentru conversia corectă în binar a unui număr cu semn ?

Pentru a evita ambiguităţile, la reprezentarea numerelor *cu semn* este obligatoriu să ţinem cont de gama numerelor.

Astfel, la conversia unui număr *cu semn* din zecimal în binar,

I. prima dată se va determina numărul minim de biţi necesar scrierii corecte a numărului (folosind gama numerelor) și abia apoi

II. se va trece la aplicarea algoritmului de conversie a numărului în noua bază.

Exemplu: Numărul *fără semn* 125 se încadrează în gama [0,+127] ce foloseşte un număr de minim 7 biţi:

$[0,+2^n-1] \Rightarrow 2^n-1=127 \Rightarrow 2^n=128= 2^7 \Rightarrow n=7 \Rightarrow 125 = 111\ 1101b$

Exemplu: Numărul *cu semn* +125 se încadrează corect în gama [-128,+127] ce foloseşte un număr de minim 8 biţi:

$[-128,+127] = [-2^{n-1},+2^{n-1}-1] \Rightarrow 2^{n-1}=128=2^7 \Rightarrow n-1=7 \Rightarrow n=8$

[1] pentru conversia numărului negativ s-a folosit codul complementar (convenţia complement faţă de 2)

Numărul cu semn + 125 se va reprezenta corect pe n=8 biţi, şi nu pe doar 7 biţi; deci +125 se va scrie corect **0**111 1101b, și nu doar 111 1101b cum s-a scris 125 ca număr fără semn (și care ar fi -3 ca număr cu semn).

Tabelul 2.3. Gama numerelor fără semn

Reprezentare	Gama numerelor fără semn	Puterea lui 2
octet	0 … 255	0 … 2^8-1
cuvânt	0 … 65 535	0 …2^{16}-1
dublucuvânt	0 … 4 294 967 295	0 …2^{32}-1

Tabelul 2.4. Gama numerelor cu semn

Reprezentare	Gama numerelor cu semn	Puterea lui 2
octet	-128 … +127	-2^7 … $+2^7$-1
cuvânt	-32 768 … +32 767	-2^{15} …$+2^{15}$-1
dublucuvânt	-2 147 483 648 … +2 147 483 647	-2^{31} …$+2^{31}$-1

Există vreo diferenţă la reprezentarea numerelor pozitive în convenţia fără semn versus cea cu semn ?

La exprimarea ***valorilor pozitive*** în ***convenţia cu semn***, se observă că reprezentarea *e aproape identică* cu cea corespunzătoare în ***convenţia fără semn***, dar ***se mai adaugă un bit (MSb) de 0***.

Care sunt convenţiile ce se pot folosi la reprezentarea unui număr negativ în convenţia cu semn ?

Pentru a reprezenta ***numerele cu semn negativ*** se pot utiliza 3 convenţii:

1. Modul şi semn (MS, numit şi "cod direct"): bit de semn urmat de modul (numit și valoare absolută). Cea mai mare, resp. cea mai mică valoare reprezentată pe octet este +127=**0**1111111, respectiv -127=**1**1111111. Se reprezintă valoarea absolută a numărului pe n-1 biţi în binar, n fiind numărul minim de biţi necesar scrierii corecte a numărului şi se adaugă un bit de semn pe poziţia bitului MSb care va fi 0 (dacă numărul e pozitiv) sau 1 (dacă numărul e negativ).

Exemplu: numărul +37 = **010 0101b**, de unde -37 se va scrie: -37 = **110 0101b** în MS

Codificarea în MS pentru numere negative se realizează foarte simplu, doar prin modificarea bitului de semn. Deşi operaţiile de înmulţire şi împărţire se realizează simplu, la operaţiile de adunare şi scădere unitatea aritmetică ar trebui să ţină cont de semnul operanzilor, deci ar rezulta circuite hardware mai complexe. Această metodă are dezavantaje în organizarea logică a unităţii centrale, deoarece există 2 reprezentări pentru 0: reprezentările 1000 0000b şi 0000 0000b care sunt echivalente (plus şi minus zero), această reprezentare nefiind deci eficientă.

2. Complement faţă de 1 (C1) (numit şi "cod invers"): se calculează prin complementarea fiecărui bit din reprezentarea ***modul şi semn***.

Exemplu: numărul +37 = **010 0101**b, deci -37 = **101 1010**b în C1

Reprezentarea în C1 este uşor de realizat de către partea hardware, dar algoritmii de înmulţire şi împărţire sunt mai complecşi decât la reprezentarea MS. În plus, ca şi în cazul convenţiei MS există 2 reprezentări pentru 0.

3. Complement faţă de 2 (C2) (numit şi "*cod complementar*"): la reprezentarea în C1 se mai adaugă un bit de 1 prin adunare cu transport.

Exemplu: numărul -37 = **101 1010**b + **1** = **101 1011**b în C2

O ***modalitate rapidă de conversie în C2*** este următoarea: dacă se dorește obţinerea unui număr -x pe n biţi (determinat cu gama numerelor) se poate aduna la 2^n acel număr (negativ) și se scrie reprezentarea rezultatului obţinut ca număr fără semn.

Exemplu: pentru numărul -37 scris pe 7 biţi, vom avea: 128 -37 = 91 = 1011011b.

Care dintre convenţiile MS, C1, C2 s-a ales pentru reprezentarea numerelor negative în PC ?

Deşi algoritmul pentru înmulţire şi împărţire pentru ***operanzi reprezentaţi în C2*** este mai complex decât cel corespunzător codului MS, aceasta este convenţia care s-a generalizat la procesoarele actuale, datorită următoarelor avantaje:

- la reprezentarea numerelor în C2 se implementează doar operaţia de adunare, deoarece scăderea unui număr din alt număr în C2 este echivalentă matematic cu adunarea complementului faţă de 2 a scăzătorului la descăzut; astfel, circuitele electronice pentru adunare şi scădere nu trebuie să examineze semnul operanzilor, vor efectua întotdeauna doar adunări;
- codificarea în C2 printr-un circuit electronic este uşor de realizat;
- convenţia C2 are o singură reprezentare pentru zero (00...0, deci oferă un cod în plus faţă de convenţiile MS şi C1);
- un întreg în reprezentarea C2 poate fi uşor extins la un format mai mare (pe un număr mai mare de biţi) fără a i se altera valoarea.

Cum se poate realiza conversia unui număr din pozitiv în negativ sau invers folosind instrucţiuni ale CPU 8086?

Pentru realizarea conversiei, procesorul 8086 are implementată instrucţiunea NEG. De exemplu, dacă vom avea valoarea +37, o vom putea obţine pe -37 sau invers.

Extensia se va realiza cu bitul de semn; astfel, în faţa MSb se vor adăuga biţi de 0 sau de 1 corespunzător MSb, până la atingerea dimensiunii dorite. Un număr scris pe *n* biţi în C2 se poate extinde la orice număr de biţi, mai mare decât *n*, cu condiţia extinderii corespunzătoare a semnului.

Care e semnificaţia pe care o poate lua un număr scris pe n biţi în reprezentarea fără semn versus cea cu semn (în oricare din convenţiile MS, C1 și C2) ?

Este important de subliniat că un număr scris pe n biţi poate avea semnificaţie diferită, în funcţie de convenţia folosită pentru reprezentare; exemple pentru numere scrise pe doar 3 biţi sunt prezentate în Tabelul 2.5.

Exemplu: numărul 7 din zecimal, scris doar pe 3 biţi, poate fi considerat 7 în reprezentarea fără semn, -3 în MS, -0 în C1 și -1 în C2.

Tabelul 2.5. Reprezentarea unui număr pe 3 biţi în diferite convenţii

Numărul în zecimal baza 10	Reprezentare în binar baza 2	Semnificaţia numărului în convenţie			
		fără semn	cu semn		
			MS	C1	C2
0	000	0	0	0	0
1	001	1	1	1	1
2	010	2	2	2	2
3	011	3	3	3	3
4	100	4	-0	-3	-4
5	101	5	-1	-2	-3
6	110	6	-2	-1	-2
7	111	7	-3	-0	-1

2.2.5. Extensia şi contractarea numerelor

Cum putem transforma o valoare reprezentată pe 8 biţi, la una pe 16 biţi ?

Pentru numerele reprezentate în calculator se pot aplica operaţii de extensie la un număr mai mare de biţi (de exemplu de la 8 la 16 biţi, de la 16 biţi la 32 biţi) sau operaţii inverse, de contracţie sau contractare (de exemplu după realizarea unei operaţii de înmulţire la care se asigură automat dimensiune dublă pentru stocarea rezultatului, când rezultatul obţinut este mic și s-ar putea scrie într-un registru de dimensiune mai mică).

Cum putem transforma o valoare reprezentată pe 16 biţi, la una pe 8 biţi ? Care sunt situaţiile care permit acest lucru ?

Dacă operaţiile de extensie a numerelor se pot realiza prin anumite instrucţiuni specifice în limbajul procesoarelor x86 (vom vedea ulterior de exemplu instrucţiunea CBW care convertește un operand de tip byte la unul de tip word), nu se poate afirma același lucru și despre operaţiile de contracţie. Aceste aspecte ţin mai mult de manevrarea valorilor în regiștri (scalarea corectă a operanzilor în regiștri) decât de folosirea lor în cadrul instrucţiunilor, aspect deosebit de important în programarea în limbaj de asamblare.

Sunt posibile 4 situaţii diferite: *extensia* sau *contractarea* unui *nr fără semn/ cu semn*.

Există vreo diferenţă la extensia fără semn versus cea cu semn a unei valori ?

- Dacă se doreşte **extensia unui număr fără semn**, atunci extensia se realizează ***întotdeauna folosind valoarea 0***.

Exemplu: extinderea numărului 80h pe 16 biţi va da 0080h, iar pe 32 biţi 0000 0080h. Operaţia de extensie fără semn a unui număr poate fi realizată folosind procesorul 8086 foarte simplu, întrucât nu implică decât adăugarea de zerouri în faţa numărului; se poate folosi instrucţiunea MOV, prin care se va zeroriza partea superioară a operandului destinaţie. De exemplu, dacă se va folosi accumulatorul AL ca având stocată o valoare ce se dorește a fi extinsă fără semn, registrul AH va trebui încărcat cu 0 (folosind de exemplu instrucţiunea mov AH,0) și atunci se va putea considera că registrul AX conţine numărul iniţial extins de la 8 biţi la 16 biţi.

- **Extensia cu semn a unui număr** este esenţială atunci când se operează două valori *cu semn,* dar *de dimensiuni diferite*.

Exemple:

a) Dacă se consideră numărul 80h, acesta se extinde ca FF80h sau FFFF FF80h;
b) Numărul 40h se extinde ca 0040h pe 16 biţi sau 0000 0040h pe 32 biţi;
Operaţiile de extensie cu semn a unui număr pot fi realizate folosind procesorul 8086 prin instrucţiunile CBW, CWD care sunt implementate și în cadrul simulatorului.

Există vreo diferenţă la contractarea fără semn vs cea cu semn a unei valori?

Contractarea numerelor cu semn este posibilă pentru acele numere care au un număr de mai mulţi biţi pornind de la c.m.s. bit având aceeași valoare (ori toţi 0, ori toţi 1).

- **Contractarea numerelor cu semn:**

Exemple: a) FF80h poate fi contractat cu semn la 80h;
b) FF40h nu poate fi contractat cu semn la doar 8 biţi, pentru că scrierea corectă se realizează pe un număr de minim 9 biţi[2];
c) 0040h poate fi contractat cu semn la 40h, sau chiar mai mult, e posibil inclusiv la un număr de doar 7 biţi[3];

- **Contractarea numerelor fără semn**

Această operaţie este foarte sensibilă întrucât în interpretarea numerelor fără semn, orice bit de 1 reprezintă o cantitate ce nu se poate neglija sau elimina pur și simplu.

Exemple:

a) FF80h nu poate fi contractat; FF40h nu poate fi contractat;
b) 0040h poate fi contractat la 40h, sau chiar mai mult, e posibil inclusiv la un număr de doar 6 biţi[4];

---

[2] Trebuie ţinut cont totuși de faptul că în arhitectura x86 nu există registru de 9 biţi
[3] nu există registru de 7 biţi

Care este regula pentru verificarea faptului că o extensie sau o contractare s-a realizat corect?

Ca o concluzie în ceea ce privește reprezentarea numerelor, trebuie subliniat că pentru transformarea numărului din binar în zecimal, primul termen din descompunerea binară este considerat cu semnul + sau – după cum numărul este pozitiv sau negativ, toţi ceilalţi termeni fiind consideraţi pozitivi.

Astfel, regula care se aplică la verificare este:

- primul termen *este nul* doar pentru ***numere pozitive sau fără semn*** deoarece cifra binară este 0, dar
- pentru ***numere negative*** primul termen trebuie considerat cu semn negativ deoarece acesta va fi o *cantitate nenulă* întotdeauna.

Această regulă poate fi folosită în special atunci când dorim să verificăm dacă o extensie s-a realizat corect sau nu.

Exemplu: Numărul +37 scris pe 7 biţi este 010 0101b şi poate fi extins pe 8 biţi astfel: **00100101**b; transformat înapoi în zecimal (pentru verificare), acesta se va scrie: $0+0+2^5+0+0+2^2+0+2^1$=+37, deci este corect chiar şi după extensie.

Exemplu: Numărul -37 în C2 a fost extins de la 7 biţi la 8 biţi: -37 = **11011011**b; transformat înapoi în zecimal, va fi: $-2^7+2^6+0+2^4+2^3+0+2^1+2^0$= **-128+64+16+8+2+1**=-37 deci este corect.

2.2.6. Interpretarea valorilor numerice

Convenţiile de reprezentare a numerelor ilustrează faptul că datele din memoria PC-ului se pot interpreta diferit. Unitatea centrală de prelucrare nu ştie semnificaţia octetului (sau cuvântului, etc).

Există instrucţiuni ale CPU care ţin cont de convenţia de reprezentare a numărului (fără semn sau cu semn) ?

Instrucţiunea folosită *poate sugera* modul de reprezentare al datelor (de ex. există instrucţiunile înrudite MUL şi IMUL, una pentru înmulţirea numerelor *fără semn*, iar cealaltă pentru înmulţirea numerelor *cu semn*); la programarea în limbaj de asamblare, ***programatorul este cel care trebuie să se asigure de folosirea corectă a acestor instrucţiuni***, în funcţie de tipul datelor.

Definirea datelor este diferită faţă de limbajele de nivel înalt, deoarece în asamblare nu există posibilitatea de a defini tipuri de date. De exemplu, valoarea C0h poate fi interpretată ca numărul fără semn 192 sau numărul cu semn -64 în C2.

---

[4] nu există registru de 6 biţi

Reamintesc aici că ***numerele negative*** se pot obţine după una din regulile ***C2***, C1, MS, iar ***numerele pozitive*** se obţin ***ca în reprezentarea fără semn***, precedate de 0.

Astăzi, toate calculatoarele folosesc reprezentarea numerelor în convenţia complement faţă de 2, celelalte două forme nemaifiind utilizate. Astfel, în continuare nu voi mai preciza acest lucru, se va considera (dacă nu se specifică altfel) că implicit, dacă e vorba de un număr negativ, acesta a fost reprezentat în C2.

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

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

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

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

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

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

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

Ce trebuie să reţinem din paragrafele anterioare este că există o posibilitate de a observa diverse situaţii excepţionale care pot să apară după efectuarea operaţiilor în

SC; pentru a ajuta la gestionarea corectă a acestor situaţii excepţionale, se pot interpreta așa numitele ***flaguri aritmetice***. Acestea pot arăta una din următ. situaţii:

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

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

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

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

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

Exemple: Se vor considera operaţiile realizate pe 4 biţi, deci și rezultatul va fi pe 4 biţi:

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

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

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

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

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

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

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

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

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

2.2.7. Numere fracţionare

O altă posibilă clasificare a informaţiei numerice este în ***numere fracţionare*** şi ***numere întregi*** (a căror parte fracţionară este deci nulă).

Cum se scrie matematic, cu puteri ale lui 2, un număr fracţionar ?

Orice număr poate fi scris ca având **o parte întreagă** şi **o parte fracţionară**, separate prin virgula binară (sau punct în sistemul de scriere englezesc şi în tehnica de calcul) (a se vedea relaţia 2.1).

Într-un sistem poziţional, un număr *N* având parte întreagă şi parte fracţionară, se poate scrie sub diferite forme, în funcţie de necesitate:

- pentru reprezentare: $\mathbf{N}=a_{n-1}\ a_{n-2}\ \ldots a_1\ a_0,\ a_{-1}\ a_{-2}\ \ldots\ a_{-(m-1)}\ a_{-m}$ (2.1)

- pt calculul valorii: $\mathbf{N}=a_{n-1}*q^{n-1} + a_{n-2}*q^{n-2} +\ldots+a_1*q^1 + a_0 + a_{-1}*q^{-1} + a_{-2}*q^{-2} + \ldots a_{-m}*q^{-m}$ (2.2)

unde: q este baza sistemului de numeraţie (q=număr întreg pozitiv);
a_i reprezintă cifrele sistemului: $0 \le a_i < q$, $i = n\text{-}1,\ldots,1,0,-1,\ldots,-m$;
n este numărul de cifre ce determină partea întreagă a numărului;
m este numărul de cifre ce determină partea fracţionară a numărului.

Cifrele a_i reprezintă coeficienţii de înmulţire a bazei la diferite puteri (pozitive sau negative), cu cifra a_{n-1} fiind c.m.s., iar cifra a_{-m} fiind c.m.p.s.
Dacă *m* = 0 numărul *N* este întreg,
dacă *n* = 0 numărul *N* este fracţionar şi subunitar, iar
dacă *m* şi *n* sunt numere întregi şi diferite de zero, atunci numărul *N* este mixt.

Câte forme de reprezentare a numerelor există
în funcţie de modul de amplasare al virgulei binare ?

Deşi virgula nu se reprezintă fizic, trebuie cunoscută localizarea ei. După modul de amplasare al virgulei binare, există două forme de reprezentare a numerelor:
(1) ***cu virgulă fixă (VF)*** sau (2) ***cu virgulă mobilă (VM)*** sau ***FP – Floating Point***.
În *forma cu virgulă fixă*, se cunoaşte a priori (este stabilită prin proiectare şi nu mai poate fi schimbată) poziţia virgulei care separă partea întreagă de cea fracţionară, existând două posibilităţi extreme de poziţionare:

- dacă virgula este aşezată după cifra de semn (adică c.m.s. bit), se operează cu numere fracţionare, subunitare;
- dacă virgula este aşezată după cifra c.m.p.s., se operează cu numere întregi.

Se poate utiliza aritmetica pentru numere întregi în ambele situaţii, iar apoi, după obţinerea rezultatului, se plasează virgula binară în poziţia predefinită; de exemplu, dacă e primul caz, virgula va fi după bitul de semn.

2.2.7.1. Conversii de numere fără semn

În această secţiune nu voi prezenta decât regulile de conversie a numerelor fracţionare pentru numere fără semn, regulile pentru conversia numerelor fracţionare cu semn considerându-se material suplimentar.

(d->b) Conversia din zecimal în binar: Se înmulţeşte numărul (partea fracţionară) cu baza 2, obţinând partea fracţionară *F*1 şi partea întreagă a_{-1} și se repetă până când se obţine partea fracţionară *Fm* = 0 sau se ajunge la precizia dorită. Cifrele întregi obţinute reprezintă cifrele numărului în baza *q*, a_{-1} fiind cifra c.m.s., iar a_{-m} cifra c.m.p.s.

Exemplu: Reprezentarea numărului fracţionar 0,35 din zecimal în binar:
0,35 *2
0,7 *2 a_{-1} = **0**
1,4 *2 a_{-2} = **1**
0,8 *2 a_{-3} = **0**
1,6 *2 a_{-4} = **1**
1,2 *2 a_{-5} = **1**
0,4 a_{-6} = **0**
Astfel, numărul din zecimal 0,35 se va scrie în binar: 0,**0**1**0**1**1**0...b

Operaţia se continuă fără a se putea ajunge la *rezultat* = 0. Deci un număr fracţionar finit într-un sistem de numeraţie poate avea ca reprezentare un număr infinit într-un alt sistem de numeraţie. Analogia cea mai simplă este cu numărul 1 împărţit la diferite baze: la împărţirea cu 2 rezultă un număr finit de cifre zecimale 1/2 = 0,5, în timp ce la împărţirea cu 3 numărul de cifre zecimale continuă la infinit 1/3 = 0,3333... .

(d->h) La **conversia din zecimal în hexazecimal** se aplică aceeași regulă ca în cazul conversiei din zecimal în binar, doar că se folosește baza 16.

Exemplu: Reprezentarea numărului fracţionar 0,25 din zecimal în hexazecimal:

0,25 *16

4,**0** a_{-1} = **4** => 0,25 = 0,**4**h

2.2.8. Reprezentarea numerelor mixte

Reprezentarea numărului 37,75 (trei zeci şi şapte virgulă şapte zecimi şi cinci sutimi) în zecimal: **37,75** = $3*10^1 + 7*10^0 + 7*10^{-1} + 5*10^{-2}$

Cum se realizează conversia unui număr fracţionar > 1 ?

Pentru numere fracţionare mai mari decât 1, partea întreagă şi cea fracţionară se obţin separat după regulile enunţate în secţiunile anterioare.

(d->b) Conversia din zecimal în binar:

Exemplu: Reprezentarea numărului fracţionar 6,35 din zecimal în binar:

6,35 = **110,0101 10...**b

(b->d) Conversia din binar în zecimal:

Exemplu: reprezentarea numărului scris în binar și transformat în zecimal ca 37,75:

100101,**11** = $1*2^5 + 0*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0 + 1*2^{-1} + 1*2^{-2}$

= **32** + **4** + **1** + **0,5** + **0,25**= **37,75**

(h->d) Conversia din hexazecimal în zecimal:

Exemplu: reprezentarea numărului 37,75 în hexazecimal și transformat în zecimal:

25,C$_h$ = $2*16^1 + 5*16^0 + 12*16^{-1}$ = **32** + **5** + **0,75** = **37,75**

(b->h) La conversia din binar în hexazecimal a unui nr mixt, având atât parte întreagă cât şi parte fracţionară, se realizează grupuri de câte 4 biţi pornind de la virgula zecimală înspre extremităţi, în ambele direcţii şi completând cu 0 dacă este cazul.

Exemplu: Numărul fracţionar 1010 0011,01b din binar în hexazecimal se scrie:

1010 0011,01b = **1010 0011,0100**b = **A3,4**h= **0A3,4**h

(h->b) Invers, conversia din hexazecimal în binar se efectuează prin înlocuirea cifrelor hexazecimale, prin grupul de 4 cifre binare corespunzător.

Exemplu:. Reprezentarea numărului fracţionar 0A3,4h din hexazecimal în binar:

0A3,4h = **A3,4**h = **1010 0011,0100**b = **1010 0011,01**b

La ce se referă noţiunea x87 ?

Procesoarele nu știu să stocheze virgula zecimală, dar pot fi programate astfel încât să considere că un anumit număr de biţi reprezintă partea zecimală. Deși EMU8086 nu știe să lucreze cu numere FP, înţelegerea materialului de mai sus – și anume lucrul cu numere fracţionare – este esenţial atunci când vorbim de arhitectura procesorului 8086. Intel a proiectat cipul 8087 în 1980, pentru a fi folosit împreună cu linia de procesoare 8086, în vederea îmbunătăţirii performanţei calculelor aritmetice în virgulă mobilă (cu până la 500%). Aspectele implementate în acest cip au devenit baza standardului IEEE754 pentru numere floating-point, prima dată acest standard fiind implementat în procesorul 80387. Cipul 8087 a implementat două tipuri de date FP, pe 32 și 64 biţi, iar intern avea inclusiv un format pentru date temporare pe 80 biţi. După 8087 au urmat co-procesoare matematice pentru 80186, 80286, 80386 și 80386SX, iar de la 80486 în sus, procesoarele din seria x86 aveau aceste funcţii implementate în interiorul cipului (aparţinând familiei de cipuri x87), ca o unitate de calcul separată (nu a mai fost un cip separat).

2.2.9. Operaţii de bază în diverse sisteme de numeraţie

2.2.9.1. Operaţii aritmetice: Așa cum am văzut, atunci când se foloseşte un număr finit de biţi pentru a reprezenta numerele, există riscul ca rezultatul obţinut să depăşească domeniul de valori posibile (pentru acel număr de biţi). Astfel, se va spune că a apărut o depăşire – *overflow*; aceste situaţii nu pot fi prevenite, însă pot fi detectate. *Adunarea* a două numere poate produce transport (carry) înafara numărului, iar acest transport nu se va reflecta în valoarea sumei, întrucât toate datele sunt reprezentate pe o lungime fixă (ca număr de biţi).

Există vreo asemănare între realizarea operaţiilor în zecimal versus în binar?

Analogie cu operaţiile realizate în zecimal: Adunarea în binar, hexazecimal, respectiv octal se va realiza similar modului de adunare a valorilor în zecimal, cu transportarea unei unităţi în caz de depășire (reamintesc aici că cifrele sistemului de reprezentare aparţin unui sistem modulo n, unde n este baza de reprezentare).

Tabelul 2.6. Reguli de obţinere a biţilor la adunarea și scăderea în binar

Adunarea: adunarea a 2 cifre binare

a	b	transport	Suma a+b
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Scăderea : scăderea a 2 cifre binare

a	b	împrumut	Diferenţa a-b
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

Exemple: în binar

		1	1	1		
0	1	0	1	0	1 b	+
1	0	0	1	1	1 b	
1	1	1	1	0	0 b	

în zecimal

1		
4	3	+
2	7	
7	0	

în hexazecimal

		1		
0	A	2	B h	+
1	2	3	6 h	
1	C	6	1 h	

Exemple: în binar

*						
1	0	0	1	1	1 b	-
0	1	0	1	0	1 b	
0	1	0	0	1	0 b	

în zecimal

6	3	-
2	2	
4	1	

în hexazecimal

*	*			
6	B	3	4 h	-
4	D	F	1 h	
1	D	4	3 h	

Cum realizăm adunarea a două numere în baza q?

Adunarea a două numere în baza q este o operaţie modulo q, deci cifra cu valoarea cea mai mare va fi q-1. Dacă rezultatul adunării depășește această valoare, va apărea un transport. Biţii de transport au fost indicaţi deasupra, cu o culoare diferită, așa cum se notau aceștia în clasele primare, când am învăţat să adunăm numerele în zecimal. De exemplu, la adunarea în hexazecimal, Bh+6h=11+6=17 care se va scrie 1h, adică 17-***16*** (*baza*) și o unitate merge mai departe, cu transport la cifra de rang mai mare.

La ***scăderea a două numere*** de rang i, dacă cifra descăzutului este mai mică decât cifra scăzătorului, apare un împrumut (unitatea este 1, adică BAZA) de la rangul i+1.

Exemple: Să se *însumeze* numerele de mai jos:

1 1 1	1	1	1	1 1
010101b+	21_{10} +	23_{10} = 10111b +	0A2Bh+	$6B34_{16}$ +
100111b	39_{10}	19_{10} = 10011b	1236h	$4DF1_{16}$
111100b	60_{10}	42_{10}=101010b	1C61h	$B925_{16}$

Exemple: Să se *scadă* numerele. Vom avea:

100111b-	39_{10} -	23_{10} = 10111b -	2A3Bh-	$6B34_{16}$ -
010101b	21_{10}	18_{10} = 10010b	154Dh	$4DF1_{16}$
010010b	18_{10}	5_{10} = 00101b	14EEh	$1D43_{16}$

Cum realizăm interpretarea corectă a rezultatelor obţinute ?

Operarea numerelor poate conduce uneori la rezultate greșit interpretate datorită numărului limitat de biţi în reprezentarea rezultatului. Totuși, dacă se consideră posibilele flaguri, rezultatul poate fi interpretat corect. De exemplu, adunarea a 2 numere pe n biţi va produce un rezultat tot pe n biţi, însă e posibil ca acesta să fie eronat dacă nu se consideră posibilele situaţii arătate de Carry și Overflow.

```
111
0111b+      7 +          +7 +
0101b       5            +5
1100b       12 (fără semn)   - 4 (cu semn)
```

Din calculele de mai sus scrise în zecimal, se poate observa că dacă numerele sunt interpretate în reprezentarea cu semn, rezultatul este eronat, întrucât +12 ar fi avut nevoie de 5 biţi în reprezentare, al 5-lea bit fiind 0; numărul corect s-ar fi scris 01100b (Overflow=1, Carry=0).

```
11
 1110b+     14 +         - 2 +
 0101b       5           +5
 0011b       3 (fără semn)   +3 (cu semn)
```

Din calculele de mai sus scrise în zecimal, se poate observa că dacă numerele sunt interpretate ca numere fără semn, de data aceasta, rezultatul este eronat, întrucât 17 ar fi avut nevoie de 5 biţi în reprezentare, al 5-lea bit fiind 1; numărul corect s-ar fi scris 10011b (Overflow=0, Carry=1).

Cum realizăm înmulţirea a două numere în binar ?

Înmulţirea a două numere în binar se efectuează de obicei prin adunarea repetată a unor produse parţiale. În Tabelul 2.7 se prezintă regula de înmulţire a două cifre binare a şi b: produsul este 1 doar dacă atât deînmulţitul cât şi înmulţitorul sunt 1. În general, înmulţirea a 2 numere pe n biţi va produce un rezultat pe 2n biţi.

Tabelul 2.7. Reguli și exemple de obţinere a valorilor la înmulţirea în binar

a	*b*	*produsul*
0	*0*	*0*
0	*1*	*0*
1	*0*	*0*
1	*1*	*1*

Exemplu: 12 * 6 = 72

```
12 = 1100b x
 6 = 0110
     0000
    1100
   1100
  0000
```

1001000b = $2^6 + 2^3 = 64 + 8 = 72$

Cum realizăm împărţirea a două numere în binar ?

Împărţirea a două numere în binar se efectuează invers operaţiei de înmulţire: va fi nevoie de scrierea deîmpărţitului ca un număr pe 2n biţi, de exemplu 73 = 01001001b; împărţitorul se va scrie folosind doar n biţi, de exemplu 6=0110b. Realizând operaţia de împărţire se va putea obţine câtul 12=1100b și restul 0001b, care așa cum se poate observa se vor scrie tot pe n biţi fiecare.

Exemplu: se dorește împărţirea lui 73 la 6, unde 73 = 01001001b și 6=0110b
73:6 = cât 12, rest 1 -> scrise în binar 01001001b: 0110b => cât 1100b, rest 0001b

Abordarea folosită în acest material este de a aduce informaţia necesară treptat; pe măsură ce se presupune că s-a realizat acomodarea cu anumite reguli și moduri de lucru se introduc altele noi. Astfel, din punct de vedere al operaţiilor prezentate în cadrul acestei secţiuni, abordarea este următoarea:

- se prezintă ***operaţiile de bază*** (precum cele aritmetice, cele logice la nivel de bit și cele de deplasare și rotire) dându-se *regulile* de realizare a acestor operaţii și *exemple simple*, la nivel de numere binare; aceasta pentru a deprinde cât mai ușor modul de operare și de interpretare al rezultatelor acestor operaţii în SC, așa cum sunt ele intern, în binar sau hexazecimal;

- după prezentarea câtorva ***exemple la nivel ipotetic***, de operaţie *generală*, se prezintă exemple de operaţii ***folosind operatorii*** care așa cum vom vedea în *Capitolul 6* sunt **disponibili în simulator** (de exemplu + și - pentru a realiza adunarea, respectiv scăderea a două numere, *, / și % pentru a obţine produsul, câtul și respectiv restul împărţii a două numere, ș.a.m.d.).

Deși în *Capitolul 3* se prezintă sub formă tabelară instrucţiunile suportate de procesorul 8086, folosirea intensă a acestora în cadrul programelor, cu sintaxa corespunzătoare lor se va realiza abia în *Capitolul 11*, prin intermediul secvenţelor care se vor executa pe simulator. Prin vizualizarea rezultatelor obţinute se fixează aceste cunoștinţe. Realizarea ulterioară de exerciţii propuse asigură înţelegerea aprofundată și reţinerea pe termen lung a cunoștinţelor dobândite.

Pe ce lungime trebuie stocat rezultatul unei anumite operaţii ?

Concret, la realizarea operaţiilor aritmetice de până acum, vom putea lucra astfel:

- ***se pot aduna 2 numere* scrise pe n biţi**, rezultatul obţinut va fi tot pe n biţi, însă trebuie interpretate flagurile întrucât ar putea apărea depășire de capacitate (practic unele rezultate ar putea avea nevoie de scrierea pe n+1 biţi); pentru realizarea operaţiei de adunare în simulator se va folosi operatorul +, iar pe procesorul 8086 operaţia a fost implementată prin instrucţiunea **ADD**;
- ***se pot scădea 2 numere* scrise pe n biţi**, rezultatul obţinut va fi tot pe n biţi, însă trebuie interpretate flagurile întrucât la scădere ar putea apărea depășire de capacitate în sensul nevoii de împrumut - borrow (practic unele rezultate ar putea avea nevoie de scrierea pe n+1 biţi pentru a marca faptul că a fost nevoie de un împrumut pentru a putea realiza scăderea); pentru realizarea operaţiei de scădere în simulator se va folosi operatorul -, iar pe procesorul 8086 operaţia a fost implementată prin instrucţiunea **SUB**;

- ***se pot înmulţi 2 numere*** **scrise pe n biţi**, rezultatul obţinut va fi pe un număr dublu de biţi, adică 2n biţi; aceasta înseamnă că va trebui să ţinem cont de acest aspect atunci când vom citit rezultatul obţinut - trebuie interpretat corect tot rezultatul, așa cum apare el scris pe toţi cei 2n biţi; pentru realizarea operaţiei de înmulţire în simulator se va folosi operatorul *, iar pe procesorul 8086 operaţia a fost implementată în 2 forme, pentru considerarea valorilor ca numere fără semn, respectiv cu semn, prin instrucţiunile **MUL** și **IMUL**;

- invers, ***se poate realiza împărţirea a 2 numere***, dacă deîmpărţitul se consideră scris pe 2n biţi, iar împărţitorul pe n biţi. Rezultatele obţinute (scrise tot pe n biţi fiecare) ce se pot considera vor fi câtul și restul împărţirii, iar operatorii corespunzători sunt /, respectiv %. Pentru implementarea operaţiilor corespunzătoare pe procesorul 8086, se folosește doar o singură instrucţiune având 2 forme: **DIV** și **IDIV** (pentru considerarea valorilor ca numere fără semn, respectiv cu semn); citirea câtului sau a restului se va realiza diferit, adică rezultatele obţinute se vor furniza în structuri diferite ale procesorului. Prin cunoașterea regulilor prezentate mai sus (de către programator), se va putea lucra ușor cu oricare din aceste operaţii la nivel de programe scrise în limbaj de asamblare.

2.2.9.2. Operaţii logice pe şiruri de biţi

Există ***4 operaţii logice*** majore pe biţi: **NOT**, **AND**, **OR** şi **XOR**, iar Tabelul 2.8 furnizează tabelele de adevăr corespunzătoare. Operaţiile logice se realizează asupra şirurilor de biţi, deci se va aplica o funcţie logică fiecărui bit în parte (din reprezentarea numărului), la acest tip de instrucţiuni neexistând transport.

Tabelul 2.8. Reguli de obţinere a valorilor la diferite operaţii logice efectuate în binar

NOT a unei cifre binare

NOT	0	1
	1	0

AND între 2 cifre binare

AND	0	1
0	0	0
1	0	1

OR între 2 cifre binare

OR	0	1
0	0	1
1	1	1

XOR între 2 cifre binare

XOR	0	1
0	0	1
1	1	0

Operaţia **NOT** (Negare logică bit cu bit) are ca efect negarea tuturor biţilor numărului din binar (de fapt calculează complementul faţă de 1 al acestuia).

Exemplu: NOT 1234h ;

1234h= 0001 0010 0011 0100b => not 1234h =1110 1101 1100 1011b =0EDCBh

Operaţiile **AND** (ŞI logic bit cu bit), **OR** (SAU logic bit cu bit) şi **XOR** (SAU-exclusiv bit cu bit) realizează operaţiile corespunzătoare, pe procesorul 8086 instrucţiunile având exact același nume, iar în simulator fiind disponibili și următorii operatori:

operatorul **~** pentru operaţia de inversare – **NOT** – inversează valoarea tuturor biţilor;
operatorul **&** pentru operaţia logică **AND** la nivel de bit;
operatorul **^** pentru operaţia logică **XOR** la nivel de bit;
operatorul **|** pentru operaţia logică **OR** la nivel de bit.
operaţiile AND și OR se utilizează în special atunci când se doreşte **mascarea** (se folosesc biţi de 0, respectiv de 1 pentru mască) anumitor biţi, în timp ce instrucţiunea XOR se foloseşte când se doreşte **complementarea** anumitor biţi.

Tabelul 2.9. Mascarea biţilor folosind operaţiile AND, OR și XOR

AND	OR		XOR
xxxx xxxx operand 0000 **1111** **masca** 0000 xxxx **rezultat**	xxxx xxxx operand 0000 **1111** **masca** xxxx 1111 **rezultat**		xxxx xxxx operand 0000 **1111** **masca** xxxx xxxx **rezultat**

Exemple: Presupunând valorile 0110b și 0011b, operaţiile logice pot fi realizate astfel:
Operaţia not: ~ 0110b = 1001b
~0011b = 1100b
Operaţia and: 0110b & 0011b = 0010b
Operaţia xor: 0110b ^ 0011b = 0101b
Operaţia or: 0110b | 0011b = 0111b

2.2.9.3. Operaţii de deplasare a şirurilor de biţi

Operaţiile logice de deplasare şi rotire sunt foarte utile programatorilor în limbaj de asamblare. De exemplu, operaţia de deplasare spre stânga (în binar) mută fiecare bit din şir cu o poziţie spre stânga, aşa cum arată Figura 2.16 (din stânga), iar rezultatul unei astfel de operaţii este echivalent cu o înmulţire cu 2 a acelui număr. În figură sunt reprezentate valorile în binar pe structuri de câte 8 biţi, deoarece așa cum vom vedea în capitolul următor aceasta este dimensiunea cea mai mică (regiștri de 8 biţi) la care se pot realiza operaţiile în cadrul procesorului 8086; o altă posibilitate ar fi folosirea regiștrilor de 16 biţi. Totuși, pentru efectuarea câtorva exerciţii simple, la mod ipotetic, se pot folosi structuri de orice dimensiune (în cazul nostru 4 biţi de exemplu).

Deplasarea spre stânga este echivalentă cu o înmulţire.

Exemplu: Numărul 0011b care este 3 în zecimal, deplasat înspre stânga cu o poziţie, va produce numărul 0110b, care este 6, iar deplasat cu 2 poziţii va produce 1100b care este 12, interpretat ca număr fără semn.

La mod general, dacă se consideră numărul într-o altă bază și prin analogie s-ar muta cifrele spre stânga cu o poziţie, s-ar obţine ca rezultat numărul înmulţit cu acea bază.

Exemplu: Numărul 1234 deplasat stânga în zecimal cu o poziţie va produce numărul: 12340, cu 2 poziţii: 123400, cu 3 poziţii: 1234000, ș.a.m.d.

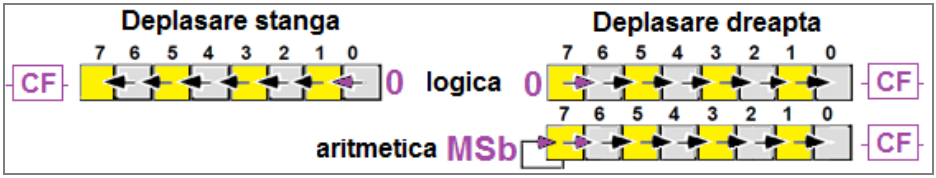


Figura 2.16. Reprezentarea operaţiilor de deplasare spre stânga și spre dreapta

La o operaţie de deplasare (logică) spre stânga, pe locul bitului LSb, adică bitul b0, se va introduce un 0, iar bitul MSb, și anume b7 va ajunge în flagul Carry. Operaţia de deplasare spre stânga cu o poziţie este echivalentă cu înmulţirea valorii cu 2^1. În general se foloseşte mnemonica **SHL** (shift left) pentru a desemna o astfel de operaţie în cadrul procesorului 8086, dar în simulator se poate folosi și operatorul **<<**.

Exemplu: numărul 6 scris în binar şi deplasat la stânga cu o poziţie va furniza numărul 12, considerat ca număr fără semn: 0110b << 1 = 1100b.

Deplasarea spre dreapta este echivalentă cu o împărţire.

Exemplu: Numărul 0110b care este 6 în zecimal, deplasat înspre dreapta cu o poziţie, va produce numărul 0011b, care este 3, iar deplasat cu 2 poziţii va produce 0001b care este 1.

În general, dacă se consideră numărul într-o altă bază și prin analogie s-ar muta cifrele spre dreapta cu o poziţie, s-ar obţine ca rezultat câtul împărţirii cu acea bază.

Exemplu: Numărul 123400 deplasat spre dreapta în zecimal cu o poziţie va produce numărul: 12340, iar cu 2 poziţii: 1234, cu 3 poziţii: 123, ș.a.m.d.

O operaţie de deplasare (logică) spre dreapta funcţionează în mod asemănător celei spre stânga, doar că datele se deplasează în sens opus, spre dreapta, aşa cum arată Figura 2.16 (din dreapta). Totuși, aici trebuie menţionat că există 2 posibilităţi, așa cum reiese și din figură: pe locul bitului MSb, adică bitul b7, se va introduce:

- ori **un 0**, caz în care se spune că s-a realizat o ***deplasare logică spre dreapta***,

- ori **un bit identic cu bitul MSb**, caz în care se spune că s-a realizat o ***deplasare aritmetică spre dreapta***.

În general, se foloseşte mnemonica **SHR** (shift right), pentru a desemna o operaţie de ***deplasare logică*** spre dreapta în cadrul procesorului 8086, dar în simulator se poate folosi și operatorul **>>**. Operaţia de ***deplasare aritmetică*** spre dreapta se poate obţine folosind mnemonica **SAR** (shift aritmetic to right) în cadrul procesorului 8086, dar în simulator nu are atribuit vreun operator.

Operaţia de deplasare spre dreapta rotunjeşte rezultatul înspre întregul cel mai apropiat, care e mai mic sau egal cu rezultatul. În oricare din cazurile de deplasare spre dreapta, bitul LSb, și anume b0 va ajunge în flagul Carry (CF).

Pentru efectuarea câtorva exerciţii simple, la mod ipotetic, se vor folosi structuri de 4 biţi pentru acomodare.

Exemplu: Numărul 0101b care este 5 în zecimal, deplasat înspre dreapta cu o poziţie, va produce numărul 0010b, care este 2 (bitul de 1 care s-a pierdut poate fi găsit în flagul Carry), iar deplasat cu 2 poziţii va produce 0001b care este 1 (flagul Carry va conţine acum un bit de 0).

La modul general, o înmulţire cu 2^n a numărului, înseamnă o deplasare spre stânga cu n biţi, iar o împărţire cu 2^n a numărului, înseamnă o deplasare spre dreapta cu n biţi.

Există situaţii când sunt necesare operaţii de deplasare a numerelor fără semn, iar aceste deplasări trebuie realizate prin operaţii de deplasare logică; în situaţiile când se dorește deplasarea numerelor cu semn, se vor folosi operaţii de deplasare aritmetică, întrucât acestea nu vor modifica semnul numerelor, ci doar valoarea lor. Similar, la deplasarea spre stânga trebuie ţinut cont de semnul numărului și de posibilele alterări ale acestuia prin operaţia de deplasare (pentru a nu obţine un rezultat eronat).

Exemple: Numere fără semn: 0011b << 2 = 1100b adică 3 x 4 = 12

Numere cu semn: 1010b >> 1 = 1101b adică – 6 : 2= - 3

2.2.9.4. Operaţii de rotire a şirurilor de biţi

Aceste operaţii de rotire la nivel de șiruri de biţi se comportă asemănător cu cele de deplasare, cu diferenţa că bitul care iese înafara reprezentării este cel care completează din cealaltă direcţie rezultatul, așa cum arată Figura 2.17.

Figura 2.17. Reprezentarea operaţiilor de rotire spre stânga și spre dreapta

Operaţiile de rotire mai au o variantă disponibilă și anume prin implicarea flagului Carry în cadrul operaţiei de rotire. Acesta acţionează ca o celulă suplimentară, așa cum reiese din Figura 2.18.

Figura 2.18. Reprezentarea operaţiilor de rotire spre stânga și spre dreapta cu CF

Aceste operaţii de rotire a șirurilor de biţi, indiferent că folosesc sau nu CF în operaţia de rotire, nu au atribuit vreun operator în simulator, dar pe procesor acestea există prin instrucţiunile specifice: **ROL** și **ROR**, resp. cu implicarea lui Carry Flag: **RCL** și **RCR**.

Exemple: Valoarea 0011b=3, rotită spre stânga cu 2 poziţii va furniza valoarea 1100b; ca operaţie, aceasta nu are definit un operator în cadrul simulatorului. Aceeași valoare, dar deplasată cu 4 poziţii, va furniza tot 0011b.

Exemple: Dacă se presupune că valoarea din flagul Carry este 1, deci CF=1 și se repetă prima operaţie din exemplul anterior (dar prin rotire spre stânga cu participarea flagului Carry), atunci se va obţine 0111b după prima rotire, și apoi 1110b după cea de-a doua. În cadrul celei de-a doua operaţii, rezultatul final, deci după 4 rotiri va fi: 1001b și CF=1.

Exemple: Fie valoarea 101100b care se dorește a fi rotită spre dreapta, fără participarea CF cu 2 și respectiv 4 poziţii; în primul caz rezultatul va fi 110010b, iar în cel de-al doilea caz rezultatul va fi 001011b.

Exemple: Același exemplu, pentru valoarea 101100b, dar acum rotită spre dreapta cu participarea CF=0, va furniza rezultatele 001011 și CF=0, respectiv în cel de-al doilea caz rezultatul va fi 100010b și CF=1.

2.3. Reprezentarea informaţiei nenumerice

Informaţia nenumerică (alfanumerică sau de tip caracter) este cea care apare sub formă de text. Termenul “caracter” se referă la orice simbol pe care oamenii sau sistemele de calcul ştiu să-l interpreteze: orice e ***tastabil*** (se poate prelua de la tastatură, chiar şi folosind mai multe combinaţii de taste) sau ***tipăribil*** (pe un sistem de afişare) reprezintă un *caracter*. Nu trebuie echivalat termenul ***caracter*** cu literele alfabetice, deoarece acesta se poate referi la: litere, numere, caractere speciale, spaţii, enter şi alte caractere de control, semne de punctuaţie, simboluri, caractere matematice, etc. Acestea sunt reprezentate în PC cu ajutorul unor sisteme de codificare (coduri), precum *ASCİİ (*pronunţat “askey”*)*, *EBCDİC* şi *Unicode.* Astfel, în timp ce informaţia numerică este reprezentată cu ajutorul sistemului de numeraţie binar, informaţia nenumerică este reprezentată prin sisteme de codificare special dezvoltate pentru informaţii de tip text. Aceste informaţii sunt reprezentate în PC tot ca numere binare, însă au o semnificaţie diferită: ele codifică un caracter.

Codul Morse a fost printre primele metode de a codifica text (anii 1840), fiind folosit la sistemul de telegraf. Pe măsură ce au evoluat calculatoarele, au evoluat şi codurile de reprezentare a caracterelor, pe un spaţiu (memorie) mai mare existând posibilitatea de a codifica mai multe caractere sau simboluri sau având posibilitatea de a reprezenta aceleaşi caractere dar cu fineţe mai mare.

BCD (Binary Coded Decimal) a fost unul dintre primele coduri folosite în sistemele de calcul; inventat de IBM, a fost folosit în calculatoarele lor încă din anii '50-'60 (sistemele 704, 7040, 709, 7090). Ceva mai târziu a fost extins la 8 biţi şi s-a redenumit ***EBCDİC (Extended Binary Coded Decimal İnterchange Code)***, incluzând caractere speciale, semne de punctuaţie şi caractere de control. IBM a folosit (până nu demult) setul de caractere EBCDİC în multe dintre sistemele lor de tip mainframe.

Alţi producători de calculatoare au optat pentru folosirea codului ***ASCİİ (American Standard Coding for İnformation İnterchange)*** care a apărut în anii 1960. În timp ce BCD şi EBCDİC erau concentrate pe coduri folosite la perforarea cartelelor sau afişaje, ASCİİ era des utilizat în telecomunicaţii. Multe dintre sistemele actuale folosesc însă standardul ***Unicode*** pe 16 biţi, o continuare a lui Ascii, acesta având posibilitatea de a codifica toate caracterele speciale (diacritice) specifice diferitelor limbi ale lumii.

2.3.1. Standardul BCD

În anumite aplicaţii (precum afişaje LCD, reprezentări pe digiţi), în general în sisteme embedded (autovehicule, cuptoare cu afişaj electronic, ceasuri cu alarmă, etc) datele numerice se folosesc ***în formă zecimală*** şi se preferă utilizarea codului (codificării) *zecimal codificat în binar* ***BCD*** (Binary Coded Decimal). Astfel, pot fi evitate conversiile repetate din zecimal în binar şi invers. De câte ori se menţionează termenul de ***valori BCD*** în cadrul instrucţiunilor implementate în simulator (și desigur cu referire la cele originale ale 8086) trebuie folosite doar cifrele zecimale, de la 0 la 9.

Codificarea BCD folosește **4 biţi** pentru a reprezenta **cele 10 cifre zecimale** {0, 1, ... 9}, asemănător cu reprezentarea hexazecimală, dar se folosesc numai primele 10 combinaţii de biţi (câte cifre sau numere zecimale există). Acestea sunt: 0_{10} -> 0000_2, 1_{10} -> 0001_2, 2_{10} -> 0010_2, ... 9_{10} -> 1001_2

celelalte combinaţii fiind nepermise (precum 1010, ... 1111).

În formatul BCD se folosesc doar 10 combinaţii în loc de 16 cât ar fi posibile pe un nibble (un digit), ducând astfel la o stocare ineficientă; un alt dezavantaj este că toate calculele în format BCD sunt mai lente decât cele în binar.

2.3.2. Standardul ASCİİ

Pentru reprezentarea sau codificarea informaţiilor alfanumerice din PC s-a folosit îndelung codificarea standard ***ASCİİ (American Standard Coding for İnformation İnterchange),*** prima utilizare comercială fiind în anul 1963 în telecomunicaţii, iar în 1968 guvernul american a adoptat pe scară largă formatul ASCİİ, atât pentru computere cât şi pentru echipamentele înrudinte.

ASCİİ este un sistem bazat pe alfabetul englez; varianta standard, codificată pe 7 biţi, cuprinde 128 de caractere text (coduri/ simboluri alfanumerice codificate ca valori întregi fără semn, între 0 şi 127 sau [0h;7Fh]): ***33 neimprimabile*** (majoritatea fiind caractere de control învechite) şi ***95 imprimabile***.

Codificarea ASCII foloseşte ***8 biţi*** (sau 7 biţi varianta standard) în varianta extinsă (numit şi ***ASCİİ-8***) pentru a reprezenta un cod Ascii şi poate cuprinde: litere (mari, mici) – cele 26 litere din alfabetul englezesc, cifre zecimale {0,...,9}, simboluri matematice (+ - =), semne de punctuaţie (. , ; !), coduri de editare şi formatare a textului (SP - Space, BS - BackSpace, CR - CarriageReturn, LF - LineFeed), coduri de control al transferului de date/text (STX - start of text, ETX - end of text).

Varianta extinsă a codului are în plus al 8-lea bit (MSb) care de-a lungul timpului a avut mai multe semnificaţii (sau interpretări):

- întotdeauna este 0;
- este bit de paritate pentru asigurarea protecţiei;
- e folosit la extinderea alfabetului ASCİİ de la 128 la 256 simboluri.

Setul de caractere ASCİİ, aşa cum se poate urmări şi în Tabelul 2.10, se poate diviza în 4 grupuri mari a câte 32 caractere (fiecare caracter este reprezentat de un număr):

1) *primul grup* cuprinde caractere speciale (între 0 şi 1Fh sau 31), neprintabile, numite şi „de control" pentru că realizează diferite operaţii de control asupra unor periferice. De exemplu, *carriage return* – poziţionează cursorul în partea stângă a liniei curente, *line feed* – mută cursorul în jos, *back space* –mută cursorul înapoi o poziţie (spre stânga). Din păcate, diferitele caractere de control au roluri diferite în cadrul unor periferice diferite, existând foarte puţine standardizări între dispozitive în acest sens;

2) *al doilea grup* include semne de punctuaţie, caractere speciale şi cifre zecimale, dar şi caracterul spaţiu - *space* (cod ASCİİ 20h);

3) *al treilea grup* de 32 caractere ASCİİ este în mare măsură dedicat caracterelor majuscule din alfabetul englezesc: "A"..."Z" pentru 41h...5Ah (65...90), în total 26 caractere diferite; celelalte 6 caractere sunt dedicate unor simboluri speciale;

4) *ultimul grup* cuprinde caracterele minuscule ale alfabetului englezesc între 61h...7Ah, încă 5 simboluri speciale şi un caracter de control (delete).

În Tabelul 2.10 se pot urmări grupurile de caractere, aşa cum se prezintă mai jos:
Litera A – se reprezintă ca numărul 65=41h (coloana 4 şi linia 1),
Litera M – se reprezintă ca nr. 77=4Dh (coloana 4 şi linia 13 - coresp. 0Dh),
Cifra 5 - se reprezintă ca numărul 35h (coloana 3 şi linia 5),
Spaţiu – se reprezintă ca numărul 20h (coloana 2 şi linia 0).

Tabelul 2.10. Standardul ASCİİ pe 7 biţi

		$b_6b_5b_4$	000	001	010	011	100	101	110	111
Baza 10	*Baza 16*	Binar $b_3b_2b_1b_0$	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0	*0*	**0000**	(ctrl@)NUL	(ctrlP)DLE	**SP**	0	@	P	‘	p
1	*1*	**0001**	(ctrlA)SOH	(ctrlQ)DC1	!	1	**A**	Q	a	q
2	*2*	**0010**	(ctrlB)**STX**	(ctrlR)DC2	“	2	B	R	b	r
3	*3*	**0011**	(ctrlC)**ETX**	(ctrlS)DC3	#	3	C	S	c	s
4	*4*	**0100**	(ctrlD)EOT	(ctrlT)DC4	$	4	D	T	d	t
5	*5*	**0101**	(ctrlE)ENQ	(ctrlU)NAK	%	**5**	E	U	e	u
6	*6*	**0110**	(ctrlF)ACK	(ctrlV)SYN	&	6	F	V	f	v
7	*7*	**0111**	(ctrlG)BEL	(ctrlW)ETB	`	7	G	W	g	w
8	*8*	**1000**	(ctrlH)**BS**	(ctrlX)CAN	(	8	H	X	h	x
9	*9*	**1001**	(ctrlİ)(tab)HT	(ctrlY)EM	)	9	İ	Y	i	y
10	*A*	**1010**	(ctrlJ)**LF**	(ctrlZ)SUB	*	:	J	Z	j	z
11	*B*	**1011**	(ctrlK)VT	(ctrl[)**ESC**	+	;	K	[	k	{
12	*C*	**1100**	(ctrlL)FF	(ctrl\)FS	,	<	L	\	l	|
13	*D*	**1101**	(ctrlM)**CR**	(ctrl])GS	-	=	**M**	]	m	}
14	*E*	**1110**	(ctrlN)SO	(ctrl^)RS	.	>	N	^	n	~
15	*F*	**1111**	(ctrlO)Sİ	(ctrl_)US	/	?	O	_	o	DEL

În concluzie, se pot specifica şi reţine următoarele: literele mari încep de la 41h, literele mici încep de la 61h, iar cifrele din domeniul [0;9] sunt în gama [30h;39h].

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.

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 2.19 (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 e echivalentă valorii 10 (la adresa 07107h), pe când 'A' este caracterul având codul Ascii 41h (la adresa 07108h).

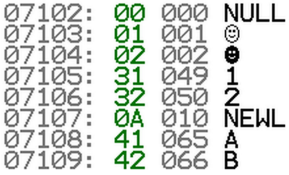


Figura 2.19. Exemple de valori interpretate ca numere binare sau coduri Ascii

2.3.3. Interacţiunea dintre tastatură, programul sursă şi ecran

Atunci când se apasă pe tasta cu numărul 1, de la tastatură se va înregistra/ prelua codul Ascii al lui ‚1' și va fi disponibil în cadrul programului (intern, în PC). Invers, atunci când dorim să se afișeze pe ecran cifra 1, va trebui să formăm codul Ascii al acestei cifre și apoi să găsim o modalitate (folosind întreruperi specifice ecranului) de a trimite acest cod înspre ecran din cadrul programului (sau din interiorul PC).

Exemplu: Figura 2.20 ilustrează operaţiile ce trebuie realizate atunci când se dorește preluarea de la tastatură a 2 cifre zecimale (considerate numere fără semn), adunarea lor și afișarea sumei acestora pe ecran. Înainte de realizarea sumei e nevoie de obţinerea valorilor 2 și 5 din ‚2' și ‚5'; astfel, se va scădea 30h sau ‚0' din fiecare.

După obţinerea sumei ca valoarea 7, aceasta trebuie la rândul ei transformată în ‚7' pentru a putea fi tipărită pe ecran. Problemele se pot complica destul de mult prin faptul că funcţiile de afișare prezentate până acum nu pot afișa un număr format din mai mulţi digiţi, deci valori mai mari decât 9. Pentru a putea opera cu astfel de valori (>9), rezultatul va trebui considerat un șir de cifre, exact ca în scrierea poziţională și procedeul de afișare va trebui repetat pentru fiecare caracter în parte.

Exemplu: Se dorește afișarea numărului 123: se va considera că prima dată trebuie să apară pe ecran 1, apoi cu o poziţie a cursorului mai spre dreapta 2, iar în final, cu încă o poziţie a cursorului spre dreapta cifra 3. Inclusiv la preluarea valorilor de la tastatură se va considera acest aspect în sens invers, de exemplu dacă se introduce 123 va trebui compus numărul 123 (o sută două zeci și trei) ca fiind 1*100+2*10+3. În plus, pentru simplitate, se presupune că aceste numere sunt doar pozitive.

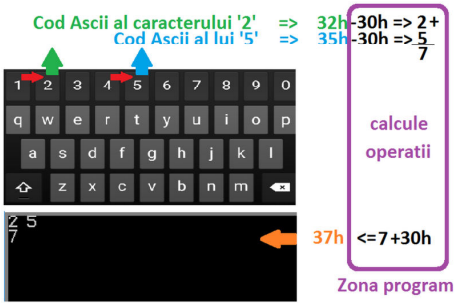


Figura 2.20. Transformări necesare la preluarea și afișarea numerelor

În procesor nu există implementate funcţii speciale pentru a lucra cu valori ASCII. Toate caracterele sunt prelucrate la fel, noi ca programatori trebuie să avem grijă la interpretarea lor ca și caractere speciale.

În cadrul programelor de editare şi procesare text, atunci când tastăm caractere, acestea sunt stocate sub formă de coduri ASCİİ într-o zonă din memorie asignată acelui document. Procesorul Word va adăuga noi caractere pe măsură ce le tastăm (în memoria RAM) folosindu-se de un pointer. Când documentul se scrie pe (hard)disc, secţiunile separate de text sunt sortate şi scrise în fişier într-un flux continuu, ca șir de caractere (în engleză *string)* Ascii.

Există o legătură importantă între ASCİİ-8 şi Unicode, deoarece cel din urmă îl cuprinde pe cel dintâi, între codurile 0000h-007Fh. Totuşi, dacă c.m.s. 9 biţi din cei 16 sunt diferiţi de 0, atunci ceilalţi 7 biţi c.m.p.s. vor avea altă semnificaţie, nu ASCİİ. Toate versiunile de Windows folosesc intern formatul Unicode pe care îl convertesc la ASCİİ dacă e necesar. Unicode este actualizat în mod regulat pentru a adăuga noi caractere şi noi simboluri pentru limbi care nu au fost codificate în varianta originală.

Unele programe, de exemplu Microsoft Office Word, pot afişa caractere şi alte simboluri folosind codificarea Unicode: apăsând pe caseta de dialog *Symbol* în meniul *Insert* apare o fereastră asemănătoare celei din Figura 2.21 în care se poate remarca reprezentarea Unicode a caracterelor, dar şi cea ASCİİ în zecimal sau hexazecimal.

Exemplu: Codul Unicode al caracterului @ se observă de pe figură că este 0040h. Se poate folosi acest cod al caracterului urmat de combinaţia de taste Alt+X pentru a obţine direct caracterul; de exemplu, apăsarea tastelor 0040Alt+X va produce apariţia caracterului @.

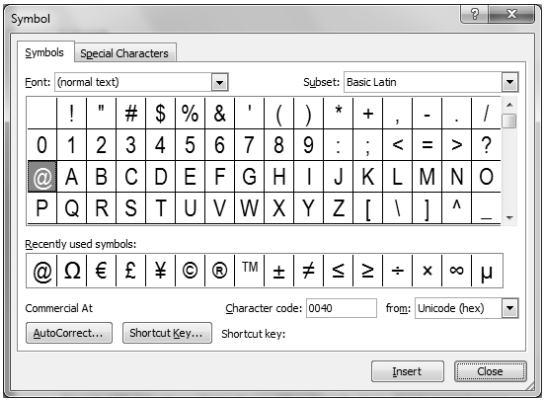


Figura 2.21. Reprezentarea codurilor Unicode în Microsoft Office Word

Există 3 forme generale de codificare disponibile în Unicode, UTF 8/16/32, iar acestea permit transmiterea datelor pe octet, cuvânt sau dublucuvânt.

Astfel, stocarea caracterelor în memoria PC-ului se poate realiza prin mai multe seturi de caractere: standard ASCİİ (0–127), ASCİİ extins (0–255), ANSİ (0–255), Unicode (0–65,535), dar e important să existe o codificare standard a acestora deoarece programele pot ajunge pe alte PC-uri sau pot utiliza alte dispozitive periferice (e esenţial ca acestea să „vorbească aceeaşi limbă").

Deşi ASCİİ este cod standardizat, prin simpla codificare a caracterelor nu se garantează compatibilitatea între diferite sisteme. De exemplu, chiar dacă valoarea 48h reprezintă litera H pe 2 sisteme diferite, există posibilitatea ca un cod de control să aibă semnificaţie diferită pe cele 2 sisteme, deoarece standardizarea nu s-a realizat şi la nivelul acestora. Un anumit cod de control ar putea avea efecte diferite pe 2 PC-uri diferite. De exemplu, sfârşitul unei linii (*End of line*) este marcat în unele S.O. cu 2 caractere: CR, urmat de LF (Windows, MS-DOS), sau cu un singur caracter CR (Apple Macintosh), sau cu un singur caracter LF (Linux). Interschimbarea de documente între 2 astfel de sisteme trebuie realizată cu precauţie.

2.3.4. Operaţii cu valori BCD

În sistemele electronice care realizează prelucrări de numere, este des întâlnită codificarea *binary-coded decimal* (BCD) adică ***zecimal codificat ca binar***. Aceasta este un tip de codificare a numerelor zecimale în formă binară, în care fiecare digit zecimal se reprezintă pe un număr fix de biţi: 4 (forma împachetată) sau 8 (forma despachetată). BCD a fost utilizat pe scară largă în trecut, dar sistemele mai noi nu au mai implementat instrucţiunile specifice (de exemplu cele cu CPU de tip ARM).

Familia de procesoare x86 are implementate încă aceste instrucţiuni, deși nu au mai fost optimizate pentru viteză. Utilizarea cea mai des întâlnită este în aplicaţiile din domeniul financiar, comercial și industrial, unde sunt necesare diverse calcule. De exemplu, într-un sistem electronic unde trebuie afișată o valoare numerică, manipularea datelor numerice este mult simplificată (versus exploatarea valorilor în binar) prin tratarea fiecărui digit ca un subcircuit electronic separat, implementat de exemplu cu afișaje de tip 7-segmente; această situaţie este mai apropiată de realitatea fizică sau hardware-ul sistemului. Astfel, în SC unde calculele sunt relativ simple (adunări, scăderi, etc), lucrul cu valori BCD poate simplifica mecanismul de implementare al întregului sistem (vs. conversia din zecimal în binar, efectuarea de calcule și apoi conversia înapoi în zecimal pentru afișare). Calculatorul de buzunar e un exemplu sugestiv în acest sens.

De exemplu, un inginer proiectant sau un programator va lucra cu memoria internă, cu CPU, deci va reprezenta valorile în regiștri pentru a le opera; deci intern, în interiorul SC vom lucra cu valori binare, deși utilizatorul va vedea aceste valori în zecimal; problema majora care se pune este dacă putem simplifica modul de lucru și de implementare al sistemului, pentru a realiza cât mai puţine din conversiile necesare (cele prezentate anterior în Figura 2.20). Multiplele astfel de conversii din formatul extern SC (de tip Ascii) și cel intern SC (de tip binar) pot fi evitate folosind instrucţiuni BCD. Aceste instrucţiuni pot ajuta mult atunci când reprezentăm aceste valori pe digiţi zecimali – din interiorul SC, pe un sistem care reprezintă doar valori zecimale sau doar preia valori zecimale pe care apoi intern le prelucrează.

Procesoarele Intel din familia x86 au instrucţiuni în limbaj de asamblare care suportă operaţii aritmetice în reprezentarea BCD, adică valorile pot fi considerate numerele cu care ne-am obișnuit noi oamenii din școala primară: pe un digit: 0…9, pe 2 digiţi: 00..99, și așa mai departe.

Valorile cifrelor zecimale pot fi codificate BCD:

- **individual - fiecare cifră pe câte un octet** (forma ***despachetată***)
  Exemplu: nr **53**h se va scrie **0000 0101 0000 0011**b
- **împreună, câte 2 cifre pe un octet** (forma ***împachetată***)
  Exemplu: nr **53**h se va scrie **0101 0011**b

Aritmetica BCD: operaţiile realizate în aritmetică BCD se efectuează pentru valori exprimate în forma împachetată, adică 2 digiţi BCD pe un octet. Astfel, se poate realiza corecţia zecimală („***Decimal adjust***”) după ***adunarea***, respectiv după ***scăderea*** a două valori exprimate pe octet în formă BCD împachetată; operaţiile corespunzătoare sunt implementate în EMU folosind instrucţiunile **DA*A*** și **DA*S***.

Tabelul 2.11. Exemplificarea operaţiilor de *ajustare Zecimală*

Operaţia	**Adunare** BCD împachetat	**Scădere** BCD împachetat
Exemplificare	54h+ 19h 6Dh	54h- 19h 3Bh
După corecţie	73h	35h

Aritmetica Ascii: operaţiile în aritmetică Ascii se realizează pentru valori exprimate în forma despachetată, adică 1 singur digit BCD se va exprima pe un octet (doar nibble-ul c.m.p.s.). Este posibilă corecţia Ascii („**Ascii adjust**") după **adunarea**, **scăderea**, **înmulţirea** și respectiv înainte de **împărţirea** valorilor exprimate pe octet în forma BCD împachetată; operaţiile corespunzătoare sunt implementate în EMU folosind instrucţiunile **AAA**, **AAS**, **AAM** și **AAD**.

Tabelul 2.12. Exemplificarea operaţiilor de *ajustare Ascii*

Operaţia	**Adunare** BCD despachetat	**Scădere** BCD despachetat	**Înmulţire** BCD despachetat	**Împărţire** BCD despachetat
Exemplificare	0504h+ 0109h 060Dh	0504h- 0109h 03FBh	06h* 07h 42	0208h pregătește pt împărţire
După corecţie	0703h	0205h+0100h= 0305h	0402h	28 = 001Ch

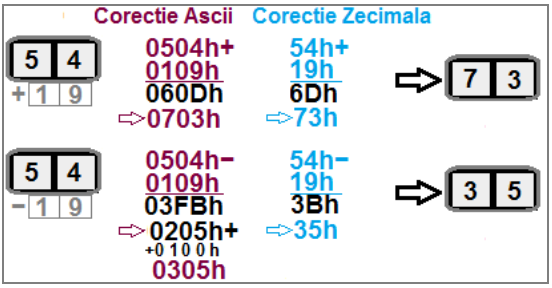


Figura 2.22. Exemplificarea operaţiilor de corecţie Ascii și Zecimală

Dacă, de exemplu la scădere, realizăm operaţia 28h-19h, după corecţia zecimală se va obţine 09h, dar dacă se va scădea 18h-19h, după corecţia zecimală se va obţine 99h; este ca și cum ar fi avut loc un împrumut din exteriorul domeniului de reprezentare al valorii – Borrow; în loc de FFh, în zecimal avem 99.

Folosirea operaţiilor de corecţie Ascii:

Exemplu: Să presupunem că a fost efectuată o adunare care a dus la obţinerea rezultatului 15 în zecimal care se scrie 0Fh sau 00F0h; de exemplu s-au adunat două valori ca format împachetat 9+6 și am depășit cifra BCD; ca să putem reprezenta tot cu BCD, vom folosi o operaţie de *corecţie Ascii după adunare*, astfel:

000**F**h = **15**d (prin operaţia de corecţie Ascii după adunare)-> **0105**h

Exemplu: Se presupune că am preluat de la tastatură 2 valori zecimale, de exemplu am preluat tastele 1 și 5, adică vom avea ‚1' și ‚5' și dacă scădem din ele ‚0', obţinem 1 și 5 -> scrise în hexazecimal acestea vor furniza valoarea 0105h (care se va scrie pe 16 biţi); pentru a forma această valoare ca numărul 15d =0Fh (adică scris pe 8 biţi numărul cincisprezece), putem proceda în următorul mod:

0105h (prin operaţie de *ajustare Ascii înainte de împărţire*)->se obţine **15** = **0F**h adică l-am transformat într-o valoare ce se poate scrie intern pe un singur octet.

Exemplu: **0205**h (prin operaţie de *ajustare Ascii înainte de împărţire*)-> **25** = **19**h

Acest tip de prelucrări ne poate folosi atunci când preluăm valori zecimale de la tastatură și apoi vrem să le prelucrăm; de exemplu, l-am întrebat pe utilizator vârsta sa și vrem să o verificăm; în funcţie de aceasta, îi cerem sau nu informaţii suplimentare

Exemplu: Invers decât în exemplele anterioare, dacă de exemplu, în funcţie de datele de la angajator am calculat vârsta la care un utilizator poate intra la pensie și vrem să afișăm aceste informaţii pe ecran, atunci intern avem valoarea 0041h =65 și ne pregătim să apelăm o funcţie de afișare care să scrie pe ecran un 6 urmat de un 5, adică vom folosi codurile Ascii ale lor, mai exact: ‚6','5', fiecare scris pe câte 8 biţi, deci ca 36h 35h. Astfel: 0041h=**65** (prin operaţie de *corecţie Ascii după înmulţire*) -> **0605**h și apoi vom aduna 3030h, obţinând 3635h, adică ‚65' scris pe 16 biţi; nu recomand această scriere întrucât în general se comit erori la interpretarea și scrierea valorilor în memoria sistemului ca și coduri Ascii.

2.4. Alte tipuri de date

Până acum, s-au prezentat informaţiile numerice în secţiunea 2.2, iar cele alfanumerice în secţiunea 2.3. Totuși, de multe ori avem impresia că în PC ar mai exista și alte tipuri de date, precum cele multimedia (desene grafice, fișiere audio sau video, etc). De fapt, toate acestea se reprezintă intern în PC tot în format binar, așa cum am arătat în secţiunea 2.2.

Capacitatea mare de stocare a SC actuale, viteza ridicată de prelucrare dar şi cea de transmitere (prin Internet) a acestor tipuri de date au dus la o explozie a popularităţii lor în SC. De exemplu, vocea umană ar putea fi înregistrată și convertită în semnal digital pentru a fi stocată pe un (hard)disc.

După digitizare, semnalul se va reprezenta în memorie sub forma unor eșantioane, fiecare eșantion (reprezintă amplitudinea semnalului) fiind codificat pe un anumit număr de biţi, de exemplu 8 biţi. Astfel, eșantioanele vor putea reprezenta maxim 256 valori de amplitudine, care pot fi considerate ca numere fără semn (între [0;255]) sau ca numere cu semn (între [-128;+127]). Cât de multă zonă va fi ocupată în memorie de acest semnal depinde de durata iniţială a fișierului sau a înregistrării audio; în final, fiecare eșantion stocat în memorie va fi tratat ca orice dată binară.

Ca și numerele, textul sau datele multimedia, tot așa și programele software trebuie reprezentate în memoria PC-ului, deoarece tot ceea ce se folosește de către procesor este preluat de acesta din memorie sau din regiștrii săi (unde pentru a ajunge, tot din memorie sau de la porturi vine). Înainte ca un procesor să execute instrucţiuni dintr-un program (de exemplu să ceară o informaţie de la utilizator prin intermediul tastaturii sau să mute un fișier de pe un dispozitiv de stocare pe un altul) va trebui să convertească instrucţiunile într-un cod binar numit ***limbaj mașină***, care în final devine doar șir de biţi: **10100001 00000000 00000000 11110111 00100110 00000010 00000000 00000011 00000110 00000100 00000000 11101000 00000001 00000000**. Un astfel de șir de biţi, ar putea părea că nu are nici o semnificaţie, însă acesta reprezintă de fapt anumite instrucţiuni și operanzi. Biţii au fost grupaţi în octeţi, astfel că ei pot fi înlocuiţi cu cifrele hexazecimale: **A10000 F7260200 03060400 E80100h** care după operaţia de decodificare a instrucţiunilor se transformă în **mov ax,a; mul b; add AX,c; call WriteResult;** această secvenţă de instrucţiuni în limbaj de asamblare putea să provină ca echivalent al unei instrucţiuni scrise în limbajul C: ***cout<<(a*b+c)***.

Primele calculatoare trebuiau programate direct în limbaj mașină, dar în prezent putem programa într-un *limbaj de programare* (de nivel scăzut precum limbajul de asamblare sau de nivel ridicat precum C++, Java, etc); acest limbaj este apoi translatat de către computer într-un limbaj mașină pe care procesorul să-l poată înţelege și executa. Procesorul nu poate distinge intern semnificaţii diferite ale octeţilor, acesta lasă interpretarea în sarcina aplicaţiilor sau programelor care folosesc acele date. Este sarcina programatorilor de a scrie programe în vederea interpretării corecte a tipului datelor găsite în memorie. Cei care scriu aplicaţiile, de exemplu cei de la Microsoft care au realizat Word-ul din suita Office, au în grijă modul de interpretare a valorilor tastate de utilizatorul aplicaţiei. Una dintre cele mai importante lucruri pe care le realizează *limbajele de nivel înalt* este că furnizează o asociere între tipul datelor și un șir de biţi din memorie. Aceasta permite compilatorului să interpreteze datele, în locul programatorului. Totuși, la scrierea programelor în *limbaj de asamblare*, programatorul este cel care trebuie să gestioneze corect datele, adică să le și interpreteze sensul.

2.5. Exerciţii propuse

Exerciţii PRACTICE: (se vor rezolva în şablon)

Set 1 (secţiunea 2.1)

1. Specificaţi numărul de locaţii ocupate în memorie de: i) un cvadruplucuvânt; ii) două cuvinte; iii) două cvadruplucuvinte; iv) patru cuvinte; acest număr este: _____locaţii.

2. În reprezentarea numărului 12345678h, octetul de rang i) 1; ii) 0; iii) 2; iv) 3 este: ____h

3. În reprezentarea valorii 0001 0010 0011 0100 0101 0110 0111 1000b bitul c.m.p.s. din octetul de rang i) 0; ii) 1; iii) 2; iv) 3 este: ____b

4. În reprezentarea valorii 0001 0010 0011 0100 0101 0110 0111 1000b bitul c.m.s. din octetul de rang i) 2; ii) 3; iii) 0; iv) 1 este: ____b

5. În reprezentarea valorii 0001 0010 0011 0100 0101 0110 0111 1000b tetrada (nibble-ul) de rang i) 0; ii) 2; iii) 4; iv) 6 este: _____b

6. În reprezentarea valorii 0001 0010 0011 0100 0101 0110 0111 1000b tetrada (nibble-ul) de rang i) 5; ii) 7; iii) 1; iv) 3 este: _____b

7. În reprezentarea cvadruplucuvântului 0123456789135764h, octetul de rang i) 0; ii) 2; iii) 4; iv) 6 este: _____h

8. În reprezentarea cvadruplucuvântului 0123456789135764h, octetul de rang i) 5; ii) 7; iii) 1; iv) 3 este: _____h

9. În reprezentarea cvadruplucuvântului 0123456789135764h, cuvântul de rang i) 1; ii) 0; iii) 3; iv) 2 este: _____h

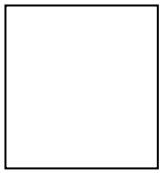
10. În reprezentarea cvadruplucuvântului 0123456789135764h, tetrada (nibble-ul) c.m.p.s. din octetul de rang i) 2; ii) 3; iii) 0; iv) 1 este: _____h

11. În reprezentarea cvadruplucuvântului 0123456789135764h, tetrada (nibble-ul) c.m.s. din cuvântul de rang i) 2; ii) 3; iii) 0; iv) 1 este: _____h

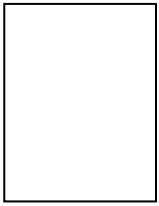
12. În reprezentarea cvadruplucuvântului 0123456789135764h, ordinul bitului c.m.p.s. din cuvântul de rang i) 2; ii) 3; iii) 0; iv) 1 este: _____

13. În reprezentarea cvadruplucuvântului 0123456789135764h, ordinul bitului c.m.s. din octetul de rang i) 2; ii) 3; iii) 0; iv) 1 este: _____

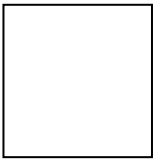
14. După modelul din Figura 2.4., realizaţi mai jos un desen care să ilustreze conţinutul memoriei dacă se va depune (după convenţia Little End-ian)
i) dublucuvântul 12345678h în memorie la adresele 103h ... 100h;
ii) dublucuvântul 56781234h în memorie la adresele 103h ... 100h;
iii) dublucuvântul 78563412h în memorie la adresele 103h ... 100h;
iv) dublucuvântul 34127856h în memorie la adresele 103h ... 100h;
Specificaţi în dreptul fiecărei locaţii și adresa în hexazecimal.



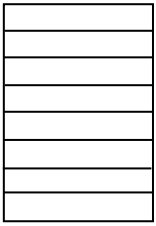
15. După modelul exerciţiului anterior, realizaţi mai jos un desen care să ilustreze conţinutul memoriei dacă se vor depune mai multe entităţi în memorie (se consideră că acestea se vor depune înspre adrese crescătoare) astfel:
i) cuvântul 1234h, urmat de cuvântul 5678h și apoi octetul 11h;
ii) cuvântul 5678h, urmat de cuvântul 1234h și apoi octetul 22h;
iii) cuvântul 7856h, urmat de cuvântul 3412h și apoi octetul 33h;
iv) cuvântul 3412h, urmat de cuvântul 7856h și apoi octetul 44h;
începând de la adresa 100h;
Specificaţi în dreptul fiecărei locaţii și adresa în hexazecimal.



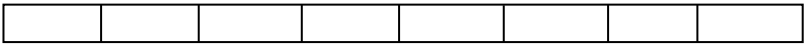
16. i), ii) Explicaţi (desen) cum se depune în memorie dublucuvântul 12345678h începând de la adresa 1234h după convenţia Little Endian.
iii), iv) Explicaţi (desen) cum se depune în memorie dublucuvântul 12345678h începând de la adresa 1234h după convenţia Big Endian.



17. i), ii) Explicaţi (desen) cum se depun în memorie următoarele:
un octet 12h, un cuvânt 1234h, un octet 56h, un dublucuvânt 12345678h începând de la adresa 1234h după convenţia Big Endian.
iii), iv) Explicaţi (desen) cum se depun în memorie următoarele:
un octet 12h, un cuvânt 1234h, un octet 56h, un dublucuvânt 12345678h începând de la adresa 1234h după convenţia Little Endian.



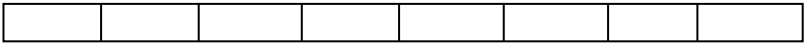
18. Repetaţi exerciţiul anterior, desenând zona de memorie pe orizontală:



Adresa: ____1234h_____

Specificaţi în dreptul fiecărei locaţii și adresa în hexazecimal.

19. Repetaţi exerciţiul 17, desenând zona de memorie pe orizontală descrescător:



Adresa: _____1234h _

Specificaţi în dreptul fiecărei locaţii și adresa în hexazecimal.

20. Care este valoarea bitului specificat, ştiind că biţii au fost numerotaţi de la dreapta spre stânga, deci LSb este b0:
i) a) b5=___; b) b10=___; c) b14=___; în cadrul numărului 7ABCh;
ii) a) b6=___; b) b11=___; c) b12=___; în cadrul numărului CBA9h;
iii) a) b4=___; b) b7=___; c) b13=___; în cadrul numărului 9876h;
iv) a) b7=___; b) b12=___; c) b15=___; în cadrul numărului 789Ah;

Set 2 (secţiunea 2.2.1.)

1. Folosind Tabelul 2.2, scrieţi următoarele valori așa cum se sugerează:
i) 0 = _____b = _____h = ____q; ii) 10 = _____b = _____h = ____q;
iii) 9 = _____b = _____h = ____q; iv) 15 = _____b = _____h = ____q;

2. Următoarele numere au fost scrise în grabă și s-a pierdut sufixul. În ce bază pot fi scrise aceste numere? (tăiaţi cu x varianta greșită)

i) 2759 d, b, h, q; ii) 02A4 d, b, h, q;
iii) 1000 d, b, h, q; iv) 1028 d, b, h, q;

3. Completaţi valorile lipsă din tabel astfel încât să număraţi în baza indicată:

i), iii)	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
ii), iv)	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
Binar:														
Hexa:														
Octal:														

4. Completaţi valorile lipsă din tabel astfel încât să număraţi în baza indicată:

i), iii)	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**
ii), iv)	**20**	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**
Binar:														
Hexa:														
Octal:														

5. Completaţi valorile lipsă din tabel astfel încât să număraţi în baza indicată:

i), iii)	**0**	**2**	**4**	**8**	**10**	**12**	**14**	**16**	**18**	**20**	**22**	**24**	**26**	**28**
ii), iv)	**0**	**4**	**8**	**12**	**16**	**20**	**24**	**28**	**32**	**36**	**40**	**44**	**48**	**52**
Binar:														
Hexa:														

6. Număraţi din 8 în 8 în binar și hexa, după cum se sugerează:

i) 3 = 0011 b = ___h ii) 5 = 0101b = ___h iii) 1 = 0001b = ___h iv) 7 = 0111b = ___h
11 = _____b = ___h 13 = _____b = ___h 9 = _____b = ___h 15 = _____b = ___h
__ = _____b = ___h __ = _____b = ___h __ = _____b = ___h __ = _____b = ___h
__ = _____b = ___h __ = _____b = ___h __ = _____b = ___h __ = _____b = ___h
__ = _____b = ___h __ = _____b = ___h __ = _____b = ___h __ = _____b = ___h

7. Scrieţi toate valorile aparţinând domeniului indicat (în baza sugerată):

i) [0101b;1001b] _____
ii) [0011b;0110b] _____
iii) [0111b;1010b] _____
iv) [1001b;1110b] _____

8. Scrieţi toate valorile aparţinând domeniului indicat în baza sugerată:

i) [12h;23h] _____
ii) [18h;29h] _____
iii) [36h;45h] _____
iv) [55h;64h] _____

9. Specificaţi numărul de elemente cuprins în interval, considerând valorile în baza indicată: i) [55h; 68h]; _____elemente ; ii) [36h;48h]; _____elemente ;
iii) [12h; 26h]; _____elemente ; iv) [18h;29h]; _____elemente ;

10. Având ca model reprezentarea din Figura 2.4 și știind că dimensiunea locaţiei de memorie la procesoarele x86 nu se modifică (este tot pe 8 biţi), sugeraţi o metodă de a reprezenta în memorie începând de la adresa 126:
i) valoarea 1234h scrisă în hexazecimal;
ii) valoarea 8765h scrisă în hexazecimal;
iii) valoarea 1234h scrisă în binar;
iv) valoarea 8765h scrisă în binar;

11. Având ca model reprezentarea din Figura 2.4 și știind că dimensiunea locaţiei de memorie la procesoarele x86 nu se modifică (este tot pe 8 biţi), sugeraţi o metodă de a reprezenta în memorie începând de la adresa 126:
i) valoarea 4321 scrisă în zecimal;
ii) valoarea 5678 scrisă în zecimal;
iii) valoarea 3456 scrisă în zecimal;
iv) valoarea 6543 scrisă în zecimal;

12. Având ca model reprezentarea din Figura 2.4 și știind că dimensiunea locaţiei de memorie la procesoarele x86 nu se modifică (este tot pe 8 biţi), sugeraţi o metodă de a reprezenta în memorie la nivel de dublucuvânt, începând de la adresa 126 următoarele valori:
i) valoarea 4321 scrisă în hexazecimal;
ii) valoarea 5678 scrisă în hexazecimal;
iii) valoarea 3456 scrisă în hexazecimal;
iv) valoarea 6543 scrisă în hexazecimal;

Set 3 (secţiunea 2.2.2.)

1. Specificaţi și prin încercuire care din următoarele valori (în convenţia de reprezentare cu semn) sunt pozitive[5]:
i) 0010b, 82h, 64h, 1000b, 73q, 42q; **ii) 0011b, 0A7h, 1010b, 75h, 71q, 32q;**
iii) 0110b, 94h, 5Ah, 1001b, 22q, 83q; **iv) 0011b, 71q, 32q, 0C7h, 1111b, 75h;**
Pozitive: _____

2. Specificaţi și prin încercuire care din următoarele valori (în convenţia de reprezentare cu semn) sunt negative[5]:
i) 0110b, 94h, 5Ah, 1001b, 22q, 83q; **ii) 0011b, 71q, 32q, 0C7h, 1111b, 75h;**
iii) 0010b, 82h, 64h, 1000b, 73q, 42q; **iv) 0011b, 0A7h, 1010b, 75h, 71q, 32q;**
Negative: _____

3. Ordonaţi crescător următoarele numere, acestea considerându-se numere cu semn[5]:
i) 0010b, 82h, 64h, 1000b, 73q, 42q; **ii) 0011b, 0A7h, 1010b, 75h, 71q, 32q;**
iii) 0110b, 94h, 5Ah, 1001b, 22q, 83q; **iv) 0011b, 71q, 32q, 0C7h, 1111b, 75h;**
Ordonate crescător: _____

[5] Numărul de biţi se consideră în funcţie de numărul de cifre specificate

4. Ordonaţi descrescător următoarele numere, considerându-le numere fără semn[5]:
i) 0110b, 94h, 5Ah, 1001b, 22q, 83q; ii) 0011b, 71q, 32q, 0C7h, 1111b, 75h;
iii) 0010b, 82h, 64h, 1000b, 73q, 42q; iv) 0011b, 0A7h, 1010b, 75h, 71q, 32q;
Ordonate descrescător: _____
5. Numărul minim de biţi necesar scrierii corecte a numărului fără semn
i) 25; ii) 58; iii) 120; iv) 130 este: ____ biţi
6. Numărul minim de biţi necesar scrierii corecte a numărului cu semn
i) +130; ii) +120; iii) +58; iv) +25 este: ____ biţi
7. Numărul minim de biţi necesar scrierii corecte a numărului cu semn
i) -25; ii) -58; iii) -120; iv) -130 este: ____ biţi
8. Analizaţi numărul de biţi necesari pentru a reprezenta numere întregi în intervalele:
i) [0;100]; ii) [0;1000]; iii) [100;200]; iv) [100; 300]
Justificaţi răspunsul: _____
_____

9. Presupunând că se dorește implementarea unui SC la care se vor folosi valori întregi i) între -1000;+1000, ii) între -100;+100, iii) între -500;+500, iv) între -200;+200, specificaţi numărul de biţi necesari la implementare și justificaţi răspunsul.
10. Specificaţi câte numere fără semn se pot scrie folosind i) 5; ii) 6; iii) 7; iv) 9 biţi. Dar dacă se consideră numere cu semn, câte vor fi pozitive / câte vor fi negative ?
____valori fără semn; ____ valori pozitive; ____valori negative

11. Completaţi capătul care lipsește din gama numerelor:
i) a) [_____; 127]; ii) a) [_____; 255]; iii) a) [_____; 2^8-1] ; iv) a) [_____; 2^7-1] ;
i) b) [-2^9; _____]; ii) b) [-2^{10}; _____]; iii) b) [-2^8; _____]; iv) b) [-2^7; _____];

12. Specificaţi numărul minim de biţi necesari scrierii corecte a numerelor fără semn: ___ biţi și respectiv cu semn pozitiv: ___ biţi și cu semn negativ: ___ biţi.
i) fără semn 4, resp. cu semn +4 și - 4;
ii) fără semn 8, resp. cu semn +8 și - 8;
iii) fără semn 16, resp. cu semn +16 și - 16;
iv) fără semn 32, resp. cu semn +32 și - 32.

Set 4 (secţiunea 2.2.3.)

1. Transformaţi din baza specificată în baza 2:

i) a) 43h=	**b) 101h=**	**c) 56q=**	**d) 7654q=**
ii) a) 75h=	**b) 110h=**	**c) 67q=**	**d) 4567q=**
iii) a) B5h=	**b) 100h=**	**c) 73q=**	**d) 1234q=**
iv) a) E8h=	**b) 111h=**	**c) 43q=**	**d) 4321q=**

2. Transformaţi din baza 2 în baza specificată (considerând numerele fără semn):

i) a) 11101000b= ____h;	**b) 0001101000b=_____h;**	**c) 1100011b=_____q;**
ii) a) 10110101b=____h;	**b) 0100100011b=_____h;**	**c) 1111011b=_____q;**
iii) a) 1110101b=____h;	**b) 0101100111b=_____h;**	**c) 1110111b=_____q;**
iv) a) 1000011b=____h;	**b) 0100110100b=_____h;**	**c) 1101110b=_____q;**

3. Transformaţi în baza 10 (din baza indicată) considerând numerele fără semn:

i) a) 3A5Dh= b) 12345q= c) 1010101b=
ii) a) 4B4Ch= b) 13245q= c) 1011100b=
iii) a) 5C3Bh= b) 12435q= c) 1001101b=
iv) a) 6D2Ah= b) 12354q= c) 1011011b=

4. Transformaţi în baza 2 (folosiţi metoda 2 de conversie):

i) a) 67= b) 1001= c) 2000=
ii) a) 64= b) 1010= c) 2010=
iii) a) 65= b) 1000= c) 2100=
iv) a) 68= b) 1100= c) 2001=

5. Transformaţi din baza 10 în baza 16 (folosind fiecare din cele 3 metode):

i) a) 43210= b) 3210= c) 456=
ii) a) 41230= b) 3012= c) 457=
iii) a) 42310= b) 3120= c) 458=
iv) a) 41320= b) 3201= c) 453=

6. Transformaţi din baza 10 în baza 8 (folosind fiecare din cele 3 metode):

i) a) 2468= b) 28= c) 16=
ii) a) 2466= b) 26= c) 14=
iii) a) 2464= b) 24= c) 15=
iv) a) 2462= b) 22= c) 17=

7. Să se convertească următoarele numere din baza 10 în baza indicată, ţinând cont de faptul că sunt numere fără semn: i) 249; ii) 251; iii) 254; iv) 247;
Numărul: _____d = _____b = _____h = _____q.
Specificaţi cifra hexa corespunzătoare lui 16^0 : ___ şi lui 16^2: ___

8. Scrieţi următoarele numere în binar folosind minim 8 biţi:

i) a) 1= .....b; ii) a) 2= .....b; iii) a) 7= .....b; iv) a) 8= .....b;
i) b) 2516= .....b; iii) b) 1250=.....b;
ii) b) 258= .....b; iv) b) 550=.....b;

9. Scrieţi următoarele numere în hexazecimal folosind minim 2 cifre hexazecimale:

i) a) 1= .....h; ii) a) 2= .....h; iii) a) 7= .....h; iv) a) 8= .....h;
i) b) 2516= .....h; ii) b) 258= .....h; iii) b) 1250=.....h; iv) b) 550=.....h;

10. Scrieţi următoarele numere din binar în zecimal:

i) 0000 0010b = ... ; ii) 000000000001b = ...;
1111110000b = ...; 111110000001b = ...;
iii) 000000000000111b = ...; iv) 0000000000111b = ...;
10000000b = ...; 1010101010b = ...;

11. Scrieţi următoarele numere din binar în hexazecimal:

i) 0000 0010b = ... ; ii) 000000000001b = ...;
1111110000b = ...; 111110000001b = ...;
iii) 000000000000111b = ...; iv) 0000000000111b = ...;
10000000b = ...; 1010101010b = ...;

12. Specificaţi cea mai mare valoare și cea mai mică valoare (considerate numere fără semn) reprezentabilă pe:

i) octet: _____b= _____d și _____b= _____d;
ii) cuvânt: _____b= _____d și _____b= _____d;
iii) 10 biţi: _____b= _____d și _____b= _____d;
iv) 12 biţi: _____b= _____d și _____b= _____d;

Set 5 (secţiunea 2.2.4.)

1. Urmărind regulile, transformaţi numerele în convenţiile specificate:

i) -131 = _____b (MS); -131 = _____b (C1); -131 = _____b (C2);
ii) -134 = _____b (MS); -134 = _____b (C1); -134 = _____b (C2);
iii) -137 = _____b (MS); -137 = _____b (C1); -137 = _____b (C2);
iv) -141 = _____b (MS); -141 = _____b (C1); -141 = _____b (C2);

2. Transformaţi din baza 2 în baza specificată (considerând numerele cu semn):

i) a) 11101000b= ____h; b) 100001101000b=_____h; c) 100011b=_____q;
ii) a) 10110101b=____h; b) 111100100011b=_____h; c) 111011b=_____q;
iii) a) 11110101b=____h; b) 110101100111b=_____h; c) 110111b=_____q;
iv) a) 11000011b=____h; b) 101100110100b=_____h; c) 101110b=_____q;

3. Transformaţi în baza 10 (din baza indicată) considerând numerele cu semn în C2:

i) a) 8A5Dh= b) 12345q= c) 1010101b=
ii) a) 9B4Ch= b) 13245q= c) 1011100b=
iii) a) AC3Bh= b) 12435q= c) 1001101b=
iv) a) BD2Ah= b) 12354q= c) 1011011b=

4. Transformaţi în baza 2 (după regula Complement faţă de 2). Comparaţi rezultatele cu cele obţinute la exerciţiul 4 din secţiunea anterioară:

i) a) -67= b) -1001= c) -2000=
ii) a) -64= b) -1010= c) -2010=
iii) a) -65= b) -1000= c) -2100=
iv) a) -68= b) -1100= c) -2001=

5. Transformaţi din baza 10 în baza 16 (după regula Complement faţă de 2). Comparaţi rezultatele cu cele obţinute la exerciţiul 5 din secţiunea anterioară:

i) a) -43210= b) -3210= c) -456=
ii) a) -41230= b) -3012= c) -457=
iii) a) -42310= b) -3120= c) -458=
iv) a) -41320= b) -3201= c) -453=

6. Transformaţi din baza 10 în baza 8 (după regula Complement faţă de 2). Comparaţi rezultatele cu cele obţinute la exerciţiul 6 din secţiunea anterioară:

i) a) -2468= b) -28= c) -16=
ii) a) -2466= b) -26= c) -14=
iii) a) -2464= b) -24= c) -15=
iv) a) -2462= b) -22= c) -17=

7. Să se convertească următoarele numere din baza 10 în baza indicată (după regula Complement faţă de 2), ţinând cont de faptul că sunt numere cu semn. Comparaţi rezultatele cu cele obţinute la exerciţiul 7 din secţiunea anterioară:

i) -249; ii) -251; iii) -254; iv) -247;

Numărul: _____d = _____b = _____h = _____q.

Numărul de biţi necesar scrierii corecte a numărului este: ____biţi.

Specificaţi cifra hexa corespunzătoare lui 16^0 : ___ și lui 16^2: ___

Specificaţi cifra octală corespunzătoare lui 8^1 : ___ și lui 8^3: ___

Scrieţi numărul corespunzător în zecimal fără semn: _____d

8. Transformaţi din baza 10 în baza 2 (folosind metoda rapidă de conversie în Complement faţă de 2). Comparaţi rezultatele cu cele obţinute la exerciţiul 4:

i) a) -67= b) -1001= c) -2000=

ii) a) -64= b) -1010= c) -2010=

iii) a) -65= b) -1000= c) -2100=

iv) a) -68= b) -1100= c) -2001=

9. Specificaţi cea mai mare valoare și cea mai mică valoare (considerate numere cu semn) reprezentabilă pe:

i) octet: _____b= _____d și _____b= _____d;

ii) cuvânt: _____b= _____d și _____b= _____d;

iii) 10 biţi: _____b= _____d și _____b= _____d;

iv) 12 biţi: _____b= _____d și _____b= _____d;

10. Completaţi în tabel în locurile corespunzătoare:

Numărul în zecimal baza 10	Reprezentare în binar baza 2	Semnificaţia în zecimal a numărului în convenţie			
		fără semn	cu semn		
			MS	C1	C2
10					
	1010				
-1					
	1110001				
-10					
1010					
					-100
				-100	
			-100		
-100					
255					
256					

Set 6 (secţiunea 2.2.5.)

1. Dintre octeţii următori, scriși ca numere cu semn reprezintă o valoare pozitivă:

i) 12h; 89h; 4h; 9h;

ii) 34h; 98h; 8h; 6h;

iii) a2h; 28h; ah; 6h;

iv) b2h; 48h; bh; 7h;

2. Extindeţi fără semn valorile de la exerciţiul 1, a.î. să fie depuse în regiștri de 16 biţi.

i) 12h= 89h= 4h= 9h= iii) a2h= 28h= ah= 6h=
ii) 34h= 98h= 8h= 6h= iv) b2h= 48h= bh= 7h=

3. Extindeţi cu semn valorile de la exerciţiul 1, a.î. să fie depuse în regiștri de 16 biţi.

i) 12h= 89h= 4h= 9h= iii) a2h= 28h= ah= 6h=
ii) 34h= 98h= 8h= 6h= iv) b2h= 48h= bh= 7h=

4. Următoarele numere scrise în convenţia cu semn trebuie depuse în regiștri de 8 biţi. Specificaţi dacă numerele sunt pozitive sau negative:

i) 12q; 67q; 3q; 6q; iii) 23q; 47q; 2q; 7q;
ii) 34q; 56q; 7q; 2q; iv) 35q; 56q; 5q; 1q;

5. Extindeţi fără semn valorile de la exerciţiul 4, la o dimensiune de 15 biţi (exprimaţi valorile rezultate tot în baza 8).

i) 12q= 67q= 3q= 6q= iii) 23q= 47q= 2q= 7q=
ii) 34q= 56q= 7q= 2q= iv) 35qh= 56q= 5q= 1q=

6. Extindeţi cu semn valorile de la exerciţiul 4, la o dimensiune de 15 biţi (exprimaţi valorile rezultate tot în baza 8).

i) 12q= 67q= 3q= 6q= iii) 23q= 47q= 2q= 7q=
ii) 34q= 56q= 7q= 2q= iv) 35qh= 56q= 5q= 1q=

7. Completaţi în tabel în locurile corespunzătoare:

Numărul în zecimal baza 10	Reprezentare în binar (baza 2) pe nr min de biţi	Numărul de biţi pentru reprezentare			
		Extins fără semn	Extins cu semn C2		
		La 8 biţi	La 8 biţi	La 16 biţi	La 32 biţi
20					
-20					
1					
-1					
-100					

8. Contractaţi cu semn următoarele valori de la 16 biţi la 8 biţi. Dacă operaţia nu este posibilă, precizaţi acest lucru și justificaţi.

i) FF91h = _____; FF01h=_____; 8080h=_____; 123h=_____;
ii) FF82h = _____; FF20h=_____; 9090h=_____; 234h=_____;
iii) FF00h = _____; FFABh=_____; 80FFh=_____; 567h=_____;
iv) FF30h = _____; FFBCh=_____; 90EFh=_____; 789h=_____;

9. Care dintre următoarele valori poate fi contractată la 8 biţi dacă se consideră valorile fără semn?

i) FE10h = _____; 050Fh=_____; 00FFh=_____; 000Ah=_____;
ii) FC10h = _____; 010Eh=_____; 00F2h=_____; 000Bh=_____;
iii) FE80h = _____; 030Fh=_____; 00FCh=_____; 000Ch=_____;
iv) FC90h = _____; 070Eh=_____; 00FEh=_____; 000Eh=_____;

10. Specificaţi nr minim de biţi care permite contractarea valorilor următoare dacă se consideră numere *fără semn*. i) FE23h; ii) FC17h; iii) FA78h; iv) EA29h;
Dar dacă se consideră numerele *cu semn*?
i) FE23h; ii) FC17h; iii) FA78h; iv) EA29h;
11. Ce valoare poate avea x pentru a putea fi contractat cu semn la nr de biţi specificat?
i) xx80h la doar 8 biţi; ii) FFx1h la doar 6 biţi;
iii) FFx0h la doar 7 biţi; iv) FFx2h la doar 5 biţi;
12. Extindeţi cu semn următoarele numere astfel încât să se dubleze dimensiunea lor în biţi: i) FBh; ii) 7Ah; iii) 80h; iv) 94h
13. Contractaţi cu semn următoarele numere la octet:
i) FF8Ah; ii) 0012h; iii) FF9Fh; iv) FF9Dh;
14. Să se scrie numerele de mai jos folosind un nibble, un octet, un cuvânt şi un dublucuvânt. Dacă nu este posibil, specificaţi şi justificaţi acest lucru.
i) +7 şi numărul -9 (in C2); iii) +8 şi numărul -7 (in C2);
ii) +9 şi numărul -10 (in C2); iv) +10 şi numărul -8 (in C2);

Set 7 (secţiunea 2.2.6.)

1. Realizaţi următoarele operaţii la nivel de 4 biţi. Scrieţi valorile în zecimal considerându-le numere *fără semn* și analizaţi rezultatul obţinut pe cei 4 biţi. Specificaţi și valoarea flagurilor aritmetice.
i) 2h+7h=___d+___d= ___h, C=__; Z=__; S=__; O=__;
ii) 8h+Ah=___d+___d= ___h, C=__; Z=__; S=__; O=__;
iii) 6h+Dh=___d+___d= ___h, C=__; Z=__; S=__; O=__;
iv) 4h+7h=___d+___d= ___h, C=__; Z=__; S=__; O=__;
2. Realizaţi următoarele operaţii la nivel de 4 biţi. Scrieţi valorile în zecimal considerându-le numere *cu semn* și analizaţi rezultatul obţinut pe cei 4 biţi. Specificaţi și valoarea flagurilor aritmetice.
i) 2h+7h=___d+___d= ___h, C=__; Z=__; S=__; O=__;
ii) 8h+Ah=___d+___d= ___h, C=__; Z=__; S=__; O=__;
iii) 6h+Dh=___d+___d= ___h, C=__; Z=__; S=__; O=__;
iv) 4h+7h=___d+___d= ___h, C=__; Z=__; S=__; O=__;
3. Realizaţi următoarele operaţii la nivel de 4 biţi. Scrieţi valorile în zecimal considerându-le numere *fără semn* și analizaţi rezultatul obţinut pe cei 4 biţi. Specificaţi și valoarea flagurilor aritmetice (simbolul * se referă la operaţia de înmulţire).
i) 2h*7h=___d*___d= ___h, C=__; Z=__; S=__; O=__;
ii) 8h*Ah=___d*___d= ___h, C=__; Z=__; S=__; O=__;
iii) 6h*Dh=___d*___d= ___h, C=__; Z=__; S=__; O=__;
iv) 4h*7h=___d*___d= ___h, C=__; Z=__; S=__; O=__;
4. Realizaţi următoarele operaţii la nivel de 4 biţi. Scrieţi valorile în zecimal considerându-le numere *cu semn* și analizaţi rezultatul obţinut pe cei 4 biţi. Specificaţi și valoarea flagurilor aritmetice (simbolul * se referă la operaţia de înmulţire).

i) 2h*7h=___d*___d= ___h, C=__; Z=__; S=__; O=__;
ii) 8h*Ah=___d*___d= ___h, C=__; Z=__; S=__; O=__;
iii) 6h*Dh=___d*___d= ___h, C=__; Z=__; S=__; O=__;
iv) 4h*7h=___d*___d= ___h, C=__; Z=__; S=__; O=__;
5. Presupunând că se foloseşte convenţia C2 pentru a reprezenta numerele cu semn pe 8 biţi, consideraţi următoarele operaţii şi precizaţi dacă sunt corecte:
i), ii) 46h+70h= 01000110b + 01110000b = 10110110b
iii), iv) 7Dh+17h=01111101b + 00011101b = 10010100b

Set 8 (secţiunea 2.2.7.)

1. Realizaţi conversia numerelor din zecimal în binar:
i) 0,25 = _____b; iii) 0,75 = _____b;
ii) 0,125 = _____b; iv) 0,375 = _____b;
2. Realizaţi conversia numerelor din zecimal în binar (cu un număr de maxim 6 zecimale):
i) 0,225 = _____b; iii) 0,425 = _____b;
ii) 0,325 = _____b; iv) 0,275 = _____b;
3. Realizaţi conversia numerelor din zecimal în hexazecimal (cu maxim 2 zecimale):
i) 0,225 = _____h; ii) 0,325 = _____h; iii) 0,425 = _____h; iv) 0,275 = = _____h;
4. Realizaţi conversia numerelor din binar în zecimal:
i) 0,001110b = _____d; ii) 0,010100b = _____d;
iii) 0,011011b = _____d; iv) 0,010001b = _____d;
5. Realizaţi conversia numerelor din hexazecimal în zecimal (cu sau fără trecere prin binar, după cum doriţi):
i) 0,39h = _____d; ii) 0,53h = _____d; iii) 0,6Ch = _____d; iv) 0,46h = _____d;
6. Specificaţi dacă s-a realizat corect conversia, completaţi cu încă un pas și notaţi valoarea finală a numărului:

i) 0,3125 *8	i) 0,4125 *8	i) 0,2125 *8	i) 0,5125 *8
2,5 *8 a-1 = 2	3,3 *8 a-1 = 3	1,7 *8 a-1 = 1	4,1 *8 a-1 = 4
4,0 a-2 = 4	2,4 a-2 = 2	5,6 a-2 = 5	0,8 a-2 = 0
_____ a-3 = ___	_____ a-3 = ___	_____ a-3 = ___	_____ a-3 = ___
=> 0,3125d = ____q	=> 0,4125d = ____q	=> 0,2125d = ____q	=> 0,5125d = ____q

Set 9 (secţiunea 2.2.8.)

1. Reprezentaţi cu puteri ale lui 10 următoarele numerele:
i) 70034,103; ii) 506,2004; iii) 1234,00001; iv) 4324,001.
2. Reprezentaţi cu puteri ale lui 2 numerele și apoi scrieţi corespondentul în zecimal:
i) 110,11; ii) 110101,1; iii) 101010,101; iv) 1011,0011;
3. Reprezentaţi cu puteri ale lui 16 numerele și apoi scrieţi corespondentul în zecimal:
i) 12,34h; ii) 14,72h; iii) 21,56h; iv) 31,27h;
4. Realizaţi conversia numerelor din zecimal în binar:
i) 20,25 = _____b; iii) 35,75 = _____b;
ii) 17,125 = _____b; iv) 25,375 = _____b;

5. Realizaţi conversia numerelor din zecimal în binar (cu un număr de maxim 6 zecimale):
i) 119,225 = _____b; iii) 125,425 = _____b;
ii) 121,325 = _____b; iv) 117,275 = _____b;

6. Realizaţi conversia numerelor din zecimal în hexazecimal (cu maxim 2 zecimale):
i) 119,225 = _____h; ii) 121,325 = _____h; iii) 125,425 = _____h; iv) 117,275 = = _____h;

7. Realizaţi conversia numerelor din binar în zecimal (numere *fără semn*):
i) 1110111,001110b = _____d; ii) 1111000,010100b = _____d;
iii) 1111101,011011b = _____d; iv) 1110101,010001b = _____d;

8. Realizaţi conversia numerelor din hexazecimal în binar. Se consideră numerele în convenţia *fără semn*:
i) 77,39h = _____b; ii) 79,53h = _____b;
iii) 7D,6Ch = _____b; iv) 75,46h = _____b;

9. Realizaţi conversia numerelor din hexazecimal în zecimal (cu sau fără trecere prin binar, după cum doriţi). Se consideră numerele în convenţia *fără semn*:
i) 77,39h = _____d; ii) 79,53h = _____d; iii) 7D,6Ch = _____d; iv) 75,46h = _____d;

10. Convertiţi valorile de mai jos, din hexazecimal în octal (cu sau fără trecere prin binar, după cum doriţi). Se consideră numerele în convenţia *fără semn*:
i) 77,39h = _____q; ii) 79,53h = _____q; iii) 7D,6Ch = _____q; iv) 75,46h = _____q;

11. Realizaţi conversia numerelor din zecimal în octal:
i) 20,25 = _____q; iii) 35,75 = _____q;
ii) 17,125 = _____q; iv) 25,375 = _____q;

Set 10 (secţiunea 2.2.9.)

1. Realizaţi următoarele operaţii la nivel de 8 biţi; a) scrieţi valorile din zecimal în binar și hexazecimal; b) efectuaţi operaţia în zecimal și binar separat, considerând numerele *fără semn*. c) Verificaţi apoi dacă rezultatul din binar corespunde celui care s-ar fi obţinut în zecimal (prin conversie); d) realizaţi operaţia și în hexazecimal și comparaţi apoi cu valorile obţinute la b); e) Reconsideraţi apoi din nou tot exerciţiul dar folosind operaţii la nivel de 9 biţi. Ce aţi observat din compararea celor 2 situaţii ?

i) 25+ b + h+ ii) 17+ b + h+
120 _____b ___h 125 _____b ___h

iii) 35+ b + h+ iv) 19+ b + h+
115 _____b ___h 123 _____b ___h

2. Realizaţi următoarele operaţii la nivel de 8 biţi; a) scrieţi valorile din hexazecimal în binar și zecimal; b) efectuaţi operaţia în hexazecimal și zecimal separat, considerând numerele *cu semn*. c) Verificaţi apoi dacă rezultatul din hexazecimal corespunde celui

care s-ar fi obţinut în zecimal (prin conversie); d) realizaţi operaţia și în binar și comparaţi apoi cu valorile obţinute la b); e) Reconsideraţi apoi din nou tot exerciţiul dar folosind operaţii la nivel de 9 biţi. Ce aţi observat din compararea celor 2 situaţii ?

i) 19h+ d+ b +
78h d b

ii) 17h+ d+ b +
7Dh d b

iii) 23h+ d+ b +
73h d b

iv) 13h+ d+ b +
7Bh d b

3. Realizaţi următoarele operaţii la nivel de 8 biţi; a) scrieţi valorile din zecimal în binar și hexazecimal; b) efectuaţi operaţia în zecimal și binar separat, considerând numerele *fără semn*. c) Verificaţi apoi dacă rezultatul din binar corespunde celui care s-ar fi obţinut în zecimal (prin conversie); d) realizaţi operaţia și în hexazecimal și comparaţi apoi cu valorile obţinute la b); e) Reconsideraţi apoi din nou tot exerciţiul dar folosind operaţii la nivel de 9 biţi. Ce aţi observat din compararea celor 2 situaţii ?

i) 25- b + h+
30 b h

ii) 17- b + h+
25 b h

iii) 35- b + h+
45 b h

iv) 19- b + h+
23 b h

4. Realizaţi următoarele operaţii în octal:

i) 123q+765q=
ii) 234q+657q=
iii) 432q+567q=
iv) 675q+257q=

5. Realizaţi calculul direct în hexazecimal:

i) A234h-9876h=
iii) ABCDh-8CDEh=
iii) 0FFFFh-0FEDCh =
iv) 1000h-123h =

6. Realizaţi următoarele operaţii în binar:

i) 1100bx101b =
10111bx11b =
iii) 11010bx1100b =
11010bx1011b =

ii) 10101b x111b =
10011b x1011b =
iv) 10101b x10b =
1011b x101b =

7. Realizaţi operaţia AND între următoarele perechi de numere (convertiţi valorile în binar, realizaţi operaţia AND bit cu bit și apoi convertiţi rezultatul în hexazecimal):

i) 1111h AND 5678h: h ii) 0ABCDh AND 2345h: h
iii) 3211h AND 8765h: h iv) 0ABCDh AND 6543h: h

8. Realizaţi operaţia OR între următoarele perechi de numere (convertiţi valorile în binar, realizaţi operaţia OR bit cu bit și apoi convertiţi rezultatul în hexazecimal):

i) 1111h OR 5678h: h ii) 0ABCDh OR 2345h: h
iii) 3211h OR 8765h: h iv) 0ABCDh OR 6543h: h

9. Realizaţi operaţia XOR între următoarele perechi de numere (convertiţi valorile în binar, realizaţi operaţia XOR bit cu bit și apoi convertiţi rezultatul în hexazecimal):
i) 1111h XOR 5678h: _____h ii) 0ABCDh XOR 2345h: _____h
iii) 3211h XOR 8765h: _____h iv) 0ABCDh XOR 6543h: _____h
10. Realizaţi operaţia NOT pentru fiecare dintre următoarele numere (convertiţi valorile în binar, realizaţi operaţia NOT bit cu bit și apoi convertiţi rezultatul în hexazecimal):
i) NOT 1111h: _____h, NOT 5678h: _____h;
ii) NOT ABCDh: _____h, NOT 2345h: _____h;
iii) NOT 3211h: _____h, NOT 8765h: _____h;
iv) NOT ABCDh: _ _____h, NOT 6543h: _____h.
11. Presupunând că valorile de la punctul 10 sunt scrise pe 16 biţi, realizaţi operaţia de deplasare logică la dreapta cu 4 poziţii:

12. Presupunând că valorile de la punctul 10 sunt scrise pe 16 biţi, realizaţi operaţia de deplasare aritmetică la dreapta cu 8 poziţii:

13. Pentru valorile de la punctul 10 realizaţi operaţia de deplasare la stânga cu 2 poziţii:

14. Pentru valorile de la punctul 10 realizaţi operaţia de rotire la dreapta cu 2 poziţii:

15. Pentru valorile de la punctul 10 realizaţi operaţia de rotire la stânga cu 3 poziţii:

16. i) Specificaţi valoarea lui -1 deplasat la stânga cu 1 poziţie:
ii) Specificaţi valoarea lui -1 deplasat la stânga cu 3 poziţii:
iii) Specificaţi valoarea lui -1 deplasat la stânga cu 5 poziţii:
iv) Specificaţi valoarea lui -1 deplasat la stânga cu 7 poziţii:
17. i) Specificaţi valoarea lui -1 deplasat logic spre dreapta cu 1 poziţie:
ii) Specificaţi valoarea lui -1 deplasat logic spre dreapta cu 3 poziţii:
iii) Specificaţi valoarea lui -1 deplasat logic spre dreapta cu 5 poziţii:
iv) Specificaţi valoarea lui -1 deplasat logic spre dreapta cu 7 poziţii:
18. Obţineţi rezultatul, dar fără a realiza vreo operaţie de înmulţire sau împărţire:
Model: Numere cu semn: 0001 0111b x 2^2= 0101 1100b adică 23 x 4 = 92
Numere cu semn: 1010 1010b : 2^1 = 1101 0101b adica -86:2=-43
i) 0101 0101b x 2^2= _____b = ____h; 0101 0101b : 2^2= _____b = ____h;
ii) 0111 0101b x 2^3= _____b = ____h; 0111 0101b : 2^3= _____b = ____h;
iii) 0101 1101b x 2^4= _____b = ____h; 0101 1101b : 2^4= _____b = ____h;
iv) 0111 1101b x 2^5= _____b = ____h; 0111 1101b : 2^5= _____b = ____h;
19. Calculaţi pe 8 biţi: i) –(56h)= ; ii) –(12h)= ; iii) –(23h)= iv) –(47h)=
20. Propuneţi o metodă de a codifica toate valorile întregi aflate în domeniul specificat. De câţi biţi e nevoie?
i) [100;200] _____; ii) [0;300] _____; iii) [10;200] _____; iv) [1;400] _____;
i) [-100;+200] _____; ii) [-200;+300] _____; iii) [-10;+200] _____; iv) [-1;+400] _____;

21. Se presupune că s-a realizat operaţia specificată în hexazecimal. Care va fi rezultatul dacă se aplică o corecţie Ascii după adunare?
i) 04h+0Ch= **iii) 07h+0Dh=**
ii) 08h+0Bh= **iv) 06h+0Ah=**
22. Propuneţi o metodă de a aduna un număr pe 8 biţi cu un număr pe 16 biţi
i), ii) reprezentate în convenţia cu semn; iii), iv) reprezentate în convenţia fără semn.

Secţiunea 2.3

1. Completaţi cu codurile ASCII pentru:
i) cifra 7: _____ cifra 3: _____
ii) litera a: _____ litera C: _____
iii) litera M: _____ space (SP): _____
iv) escape (ESC): _____ retur (CR): _____
2. Presupunând că litera A are codul ASCII 1000001b, specificaţi care va fi codul ASCII al literei i) G = _____h; ii) H = _____h; iii) K = _____h; iv) L = _____h;
3. Care va fi codificarea numărului specificat pe un calculator care foloseşte:
a) codarea ASCII pe 7 biţi şi fără paritate;
b) codarea ASCII pe 8 biţi cu paritate pară (adică există 1 bit suplimentar pt paritate);
i) 246 a: _____; b: _____
ii) 244 a: _____; b: _____
iii) 248 a: _____; b: _____
iv) 242 a: _____; b: _____
4. Decodificaţi următorul mesaj scris cu coduri ASCII pe 7 biţi şi fără paritate:
i) 1101100 1100001 1100010 0100000 0110000 0110001b;
ii) 1001100 1000001 1000010 0100000 0110000 0110001b;
iii) 1001100 1100001 1100010 0110001 1000001 1000011b;
iv) 1101100 1100001 1100010 0110011 1100001 1000011b;

5. Propuneţi o formulă de transformare
i), ii) a codului Ascii al literelor mari în cod Ascii al literelor mici corespunzătoare;
iii) iv) a codului Ascii al literelor mici în cod Ascii al literelor mari corespunzătoare.

6. Propuneţi un algoritm de verificare al unui cod Ascii dacă este corespunzător unei cifre sau nu.

7. Convertiţi numărul codat în BCD în forma sa echivalentă în zecimal:
1000011101100101.01000011

8. Pe câţi i) octeţi; ii) nibble; iii) biţi; iv) cuvinte se codifică informaţia în memorie: ‚Ana are mere'? Dar dacă aceeași valoare este scrisă în format BCD împachetat?

Capitolul 3. Arhitectura de bază a procesoarelor x86 pe 16 biţi

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

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

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

Scurt istoric al familiei x86:

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

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

3.1. Sisteme von Neumann şi non-von Neumann

J.W. Mauchly și J.P. Eckert au propus înainte de a termina ENIAC, primul calculator electronic de uz general, o nouă metodă pentru a schimba comportamentul mașinii lor de calcul. Ideea respectivă a fost propusă (top secret în timpul războiului) ca fundament pentru noul lor proiect EDVAC (primul calculator care conţinea circuite electronice).

Matematicianul John von Neumann (se pronunţă noy-mann) a participat în calitate de colaborator la realizarea calculatorului EDVAC și a fost cel care a publicat pentru prima dată ideea respectivă. Astfel, computerele cu program memorat au primit în timp numele de „***sisteme von Neumann***”, iar arhitectura respectivă s-a numit „*arhitectură von Neumann*”. Acest model de bază pentru arhitectura unui SC cu program memorat a fost introdus în anii 1944-1945 și până nu demult, majoritatea computerelor au respectat criteriile din „modelul von Neumann”.

Criteriile von Neumann

Cele ***5 caracteristici principale ale calculatorului cu program memorat***, criterii enunţate de von Neumann, sunt:

1. **intrare (I)** - prin intermediul căreia să se poată introduce un număr nelimitat de operanzi şi instrucţiuni în SC;
2. o **memorie (MEM)** - din care să se citească ***instrucţiunile*** şi ***operanzii*** şi în care să se poată memora rezultatele obţinute;
3. o **unitate de calcul (UAL)** - pentru a efectua operaţii aritmetice şi logice asupra operanzilor din memorie;
4. **ieşire (O)** - prin intermediul căreia un număr nelimitat de rezultate să poată fi redate în afara sistemului (utilizatorul să poată avea acces la ele);
5. o **unitate de comandă (UC)** - capabilă să interpreteze instrucţiunile preluate/ obţinute din memorie şi să selecteze diferite moduri de desfăşurare a activităţii viitoare a SC pe baza rezultatelor calculelor.

Principiul von Neumann

Pe baza celor menţionate anterior, s-ar mai putea enunţa principiul von Neumann astfel: un SC *cu program memorat* trebuie să posede 3 ***componente hardware principale: UCP (în care se regăsesc UAL și UC), MEM și I/O (periferice)***.

Cele 3 componente sunt legate printr-un **bus** ce transportă informaţia. **Busul** sau **magistrala** sistemului se definește ca un grup de semnale electrice sau fire folosite pentru a transfera informaţie dintr-o parte în alta a sistemului. *Viteza busului* afectează major performanţa întregului SC, la fel ca viteza UCP sau capacitatea memoriei.

Îmbunătăţiri aduse (în timp) modelului convenţional von Neumann

Dintre *îmbunătăţirile aduse (în timp) modelului convenţional von Neumann* se pot menţiona următoarele ca fiind cele mai semnificative (s-au adăugat la modelul de bază): modelul busului sistem, principiul ierarhiei de memorie, folosirea regiștrilor *index* la adresare, adăugarea de unităţi (și nu cipuri) pentru realizarea de operaţii în virgulă mobilă x87, etc.

1) modelul busului sistem, adică busuri specializate, de date, adrese și control:

Busul de date (BD) folosit pentru a muta datele din memoria principală în registrele UCP (și invers), *busul de adrese* (BA) pentru a păstra adresa datelor pe care BD le accesează la momentul curent și *busul de control* (BC) care se folosește pentru a vehicula semnalele de control (acestea specifică modul cum va avea loc transferul informaţiei - înspre CPU/ dinspre CPU, cu intervenţia unui port sau a memoriei, etc);

2) **principiul ierarhiei de memorie**: programele și datele conţinute în medii de stocare cu viteză scăzută la accesare (ex: HDD) s-au propus a fi *copiate într-o memorie volatilă*, rapid de accesat (precum RAM) înainte de a fi executate; tot ca parte a ierarhiei de memorie, s-a adăugat memorie cache și memorie virtuală;

3) **folosirea regiștrilor *index* la adresare**, iar de la 386 în sus și a regiștrilor generali;

4) adăugarea de unităţi (și nu cipuri) pentru realizarea de operaţii în virgulă mobilă; s-a adăugat intern un **coprocesor matematic**, specializat pe realizarea de operaţii în virgulă mobilă de la 486 în sus; aceste *cipuri* (până la 487) sau *unităţi separate* (după 487) au fost denumite ca aparţinând seriei sau liniei **x87**;

5) folosirea **întreruperilor** și a operaţiilor de **intrare/ ieșire (I/O) asincrone**;

6) s-au adăugat **mai multe procesoare** și mai multe **nuclee (core)** în cadrul aceluiași procesor.

Sisteme cu procesare distribuită

Astfel, odată cu îmbunătăţirea sistemelor, au apărut așa-numitele ***sisteme din categoria non-von Neumann***: computere care pot procesa informaţie ADN, pot prelucra informaţii din domeniul cuantic, cu prelucrare de fluxuri de informaţie, de tip vectorial, etc, toate acestea în general exploatând paralelismul în diverse feluri.

Din punct de vedere al evoluţiei, se poate menţiona faptul că încă de la sfârșitul anilor 1960 au existat computere de performanţă ridicată – procesoare duale;

- în anii ‘70 – existau supercomputere cu 32 procesoare;
- în anii ‘80 – au apărut supercomputere cu 1,000 procesoare;
- în 1999, IBM anunţa sistemul Blue Gene cu peste 1 million procesoare.

Toate acestea s-au folosit în domenii de vârf ale tehnologiei, din spectrul aplicaţiilor destinate armatei, domeniului medical de ultimă generaţie, etc. Ce au toate acestea în comun este faptul că **procesarea este distribuită între *mai multe unităţi de procesare ce lucrează în paralel*.**

3.2. Arhitectura uniprocesor

Microcalculatoarele tipice folosesc (după *modelul von Neumann*), în plus, pe lângă unitatea centrală de prelucrare UCP și interfeţele cu memoria şi cu dispozitivele externe de intrare/ ieşire (adaptoare de interfaţă) și un **generator de ceas (clock)**.

Unităţile funcţionale sunt interconectate prin busuri (magistrale) specializate care transferă informaţiile între acestea, așa cum reiese din Figura 3.1.

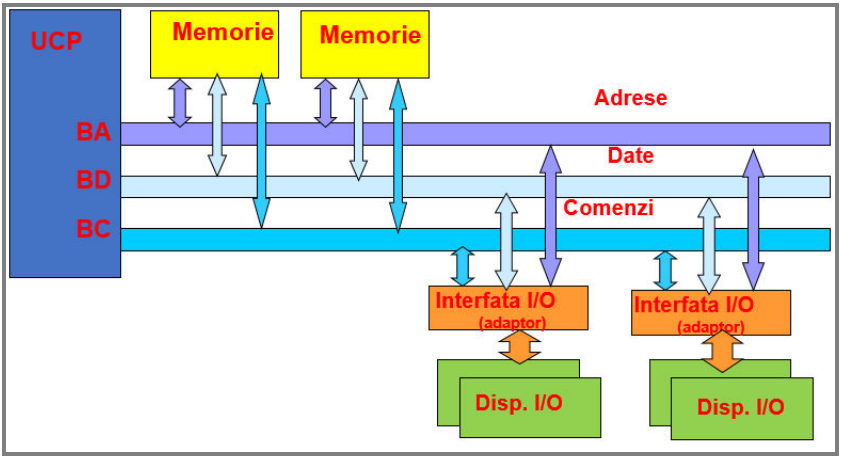


Figura 3.1. Arhitectura simplificată a unui SC

Procesorul acţionează ca un controler al tuturor acţiunilor/ serviciilor furnizate de SC, acţiunile lui fiind sincronizate cu un semnal de ceas. Astfel, fiecare SC conţine un ceas intern care specifică *intervalul de timp dintre 2 operaţii consecutive* și ajută la sincronizarea tuturor dispozitivelor din sistem.

Când scriem programe (indiferent ca folosim HLL sau LLL[6]) acestea cuprind *secvenţe de instrucţiuni* necesare îndeplinirii unei anumite sarcini. Aceste *instrucţiuni* sunt apoi "traduse" în *secvenţe echivalente de instrucţiuni* în limbaj mașină (de către asamblor) pe care procesorul le înţelege; ulterior, S.O. încarcă programul în memoria principală (cu ajutorul unui program încărcător, numit loader), îi indică procesorului locaţia respectivă și "îl ghidează" spre execuţia lui.

Codificarea/ decodificarea instrucţiunilor

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

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

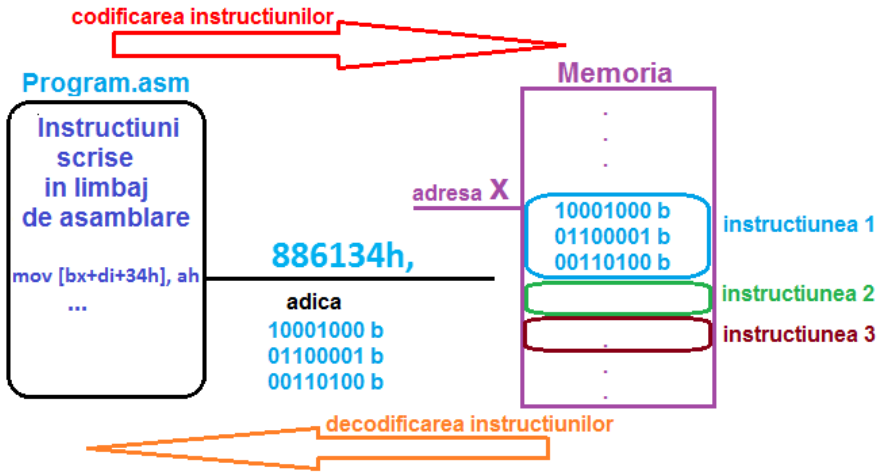


Figura 3.2. Codificarea și decodificarea instrucțiunilor

Ciclii mașină și tacți

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

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

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

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

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

Procesare secvenţială de tip von Neumann

Ideea principală sugerată de von Neumann a fost că atât instrucţiunile cât și datele sunt păstrate în aceeași memorie. Realizarea instrucţiunilor se efectuează astfel prin ***procesare secvenţială****: există* o singură cale de date (fizică sau logică) între UCP și memoria principală, forţând astfel *alternarea* de date (instrucţiuni) sau comenzi (de control); această limitare a dus în timp la ceea ce a devenit cunoscut sub numele de gâtuire ("bottleneck") de tip *von Neumann*. În modelul von Neumann NU există distincţie între instrucţiuni și date (operanzi), ele implicit fiind executate secvenţial (pe măsură ce apar în program), cu excepţia situaţiilor când apar diverse apeluri de proceduri, acestea ducând la salturi în program într-o altă zonă de memorie.

La *sistemele von Neumann* este implementat conceptul memoriei singulare (amestecate date și instructiuni), pe când la *sistemele Harvard* există memorii (*cache*) separate pentru instrucţiuni și date. Așa cum se prezintă în Figura 3.3, UCP conţine:

- **Unitate de calcul** (UAL – unitate aritmetică şi logică sau unitate de execuţie a operaţiilor) care execută operaţii aritmetice şi logice asupra operanzilor;
- **Unitate de comandă** (control) (UC) care interpretează instrucţiunile extrase din memorie şi alege diferite acţiuni pe baza rezultatelor obţinute;
- **Regiștri** – pentru stocarea temporară a datelor și instrucţiunilor; *accesarea* datelor stocate în regiștri este *mai rapidă* decât cea a datelor din memorie și de aceea în cadrul instrucţiunilor e de preferat să folosim ca locaţii regiștri, și nu zone din memorie; aceasta nu e ușor de realizat întrucât numărul și uneori și rolul regiștrilor este limitat. *Numărul de regiștri* variază de la un tip de procesor la altul: de exemplu, procesorul Pentium, un succesor mult mai avansat al lui 8086, are 8 regiștri de date și alţi 8 regiștri diferiţi, în timp ce procesorul Itanium are 128 regiștri doar pentru operarea cu date de tip întreg.

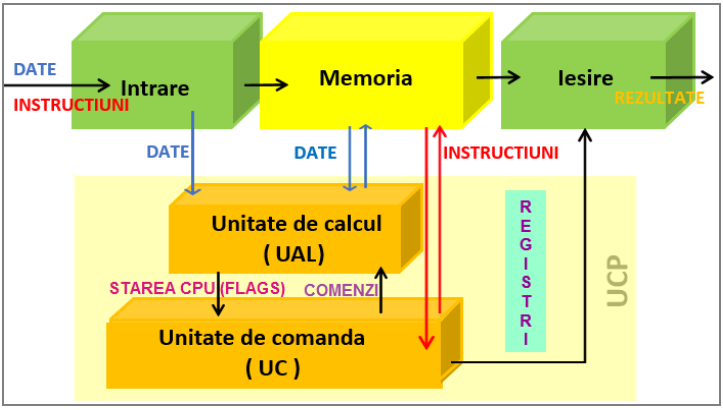


Figura 3.3. Schema bloc simplificată a UCP

3.3. Arhitectura software a microprocesorului I8086

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

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

3.3.1. Ce înseamnă procesor pe 16 biţi

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

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

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

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

Exemplu:

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

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

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

3.3.2. Schema bloc internă

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

- Unitatea de execuţie (Execution Unit - EU)
- Unitatea de interfaţă cu magistrala (Bus Interface Unit - BIU)

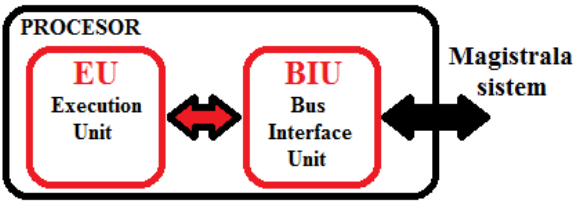


Figura 3.4. Cele două componente principale ale procesorului

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

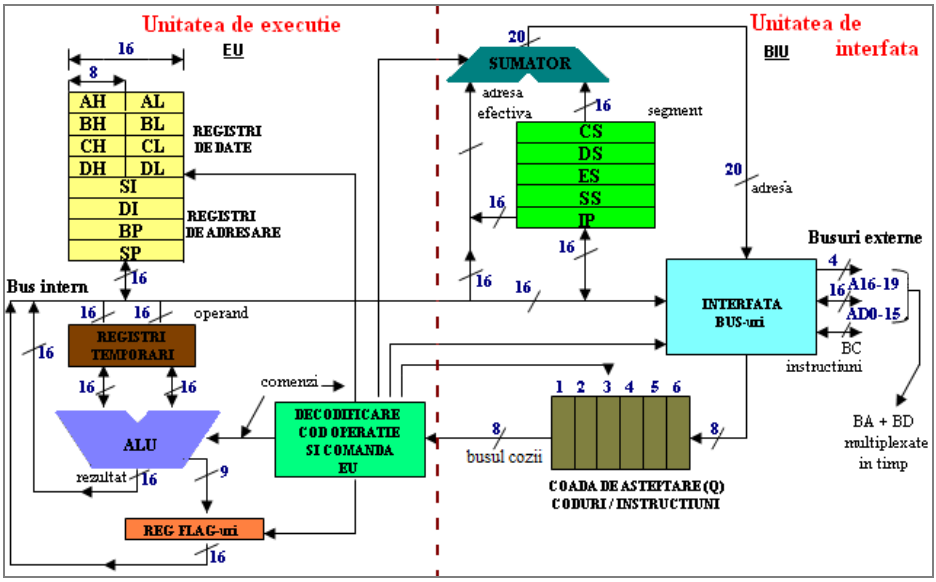


Figura 3.5 Schema bloc internă a microprocesorului I8086

Pipeline cu EU și BIU

ALU (Arithmetic and Logic Unit în engleză sau UAL în română) și **UC** (în engleză Command Unit) prezentate în Figura 3.3 – se regăsesc în cadrul EU în Figura 3.5 cu **albastru** și **verde**. Cele două unităţi EU și BIU sunt legate între ele cu o conductă (*pipeline*) prin care sunt transferate instrucţiunile extrase (din program) de către BIU spre EU; unitatea de execuţie EU are doar rolul de a executa instrucţiunile extrase de

BIU, neavând nici o legătură cu magistrala sistemului. În timp ce EU îşi îndeplineşte sarcina de a executa instrucţiuni, BIU extrage noi instrucţiuni pe care le organizează într-o coadă de aşteptare (*queue*). BIU pregăteşte execuţia fiecărei instrucţiuni astfel: extrage o instrucţiune din memorie, o depune în coada de instrucţiuni şi calculează adresa din memorie a unui eventual operand. La terminarea execuţiei unei instrucţiuni, EU are deja la dispoziţie o nouă instrucţiune din coada de aşteptare construită de BIU. Cele două unităţi, EU şi BIU, lucrează astfel în paralel, existând momente de sincronizare şi aşteptare între ele, funcţionarea paralelă a celor două unităţi fiind transparentă utilizatorului. Această arhitectură se mai numeşte şi arhitectură cu *prelucrare secvenţial – paralelă* de tip ***pipeline cu 2 etaje***.

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

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

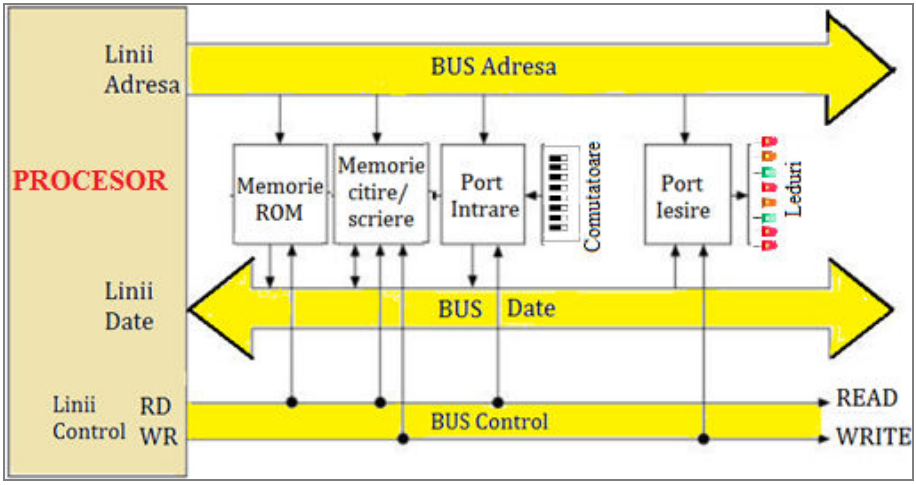


Figura 3.6 Interfaţarea cu exteriorul prin periferice de intrare și ieșire

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

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

Procesorul 8086 are o magistrală de 20 biţi, așa cum se poate observa în Figura 3.5, din care cei mai puţin semnificativi 16 biţi ***AD_{0-15} sunt partajaţi*** – termenul folosit este „multiplexaţi în timp”, adică sunt folosiţi în comun, dar nu în același timp, de către ***busul de date*** și ***cel de adresă***.

Deoarece sunt 16 biţi care circulă pe BD, operaţiile se pot realiza doar la nivel de 16 biţi; faptul că sunt 20 biţi pt adresă oferă posibilitatea ca ***dimensiunea memoriei (la adresare) să fie de 1MB*** (1 Mega Octet – adică 10^6 octeţi sau mai bine, echivalent în puteri binare cu 2^{20}).

Termenul corect ar fi fost MebiB și nu MegaB, întrucât diferenţa între 10^6 și 2^{20} nu este neglijabilă (48.576 octeţi).

3.3.3. Setul de regiştri

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

I. regiștri de UZ GENERAL (care pot fi de date, pointer sau index)

- regiștrii de date în general sunt identificaţi prin litera X de la sfârșit, regiștrii pointer - *litera P, și regiștrii* index - *litera I*, toţi aceștia fiind numiţi și GPR (General Purpose Registers în engleză);

II. regiștri SEGMENT (identificabili ușor prin *litera S*);

III. regiștri SPECIALI: cuprinde registrul pointer la instrucţiune (IP) și registrul indicatorilor de condiţii (PSW).

Setul de regiștri al fiecărui procesor este de fapt un *superset* al procesoarelor apărute anterior; de exemplu, procesorul 386 are în componenţă toţi regiștrii pe care îi avea 8086 la momentul respectiv și în plus, mai are și acei regiștri specifici lui.

Tabelul 3.1. Regiștrii procesorului 8086

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

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

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

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

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

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

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

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

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

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

Registrul IP conţine adresa instrucţiunii curente care se execută și se mai numeşte şi PC (Program Counter). După ce s-a executat instrucţiunea curentă, IP este 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 (care nu se cunoaște în general decât atunci când se ajunge în situaţia de a realiza saltul; în plus, nu se poate prezice dinainte dacă se execută sau nu saltul respectiv).

Tabelul 3.2. Adrese specifice în 8086

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

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

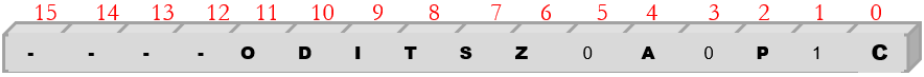


Figura 3.7 Conţinutul registrului PSW (Flags)

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

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

Indicatorii „de stare" și „de control"

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

CF (Carry Flag) este flagul care arată un posibil transport: acesta are valoarea 1 în cazul în care în cadrul ultimei operaţii efectuate a apărut un transport (carry) sau un împrumut (borrow) în/ din afara domeniului de reprezentare al rezultatului și valoarea 0 în caz contrar; a fost deja menţionat la operaţiile aritmetice în *secţiunea 2.2.6*;

PF (Parity Flag) are valoarea astfel încât împreună cu numărul de biţi de 1 din reprezentarea rezultatului, să existe un număr impar de cifre 1. Altfel spus, flagul se setează dacă numărul de biţi de 1 ai rezultatului este un număr par;

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

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

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

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

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

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

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

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

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

D=0 pentru adrese crescătoare,

D=1 pentru adrese descrescătoare.

Trap și Interrupt se mai numesc și flaguri *de sistem*.

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

Exemple: după modelul exerciţiilor prezentate în *Capitolul 2*, analizaţi următoarele operaţii și efectul lor; realizaţi calculele în binar după modelul prezentat:

Model: operaţiile se consideră pe 8 biţi; calculul 02h+02h=04h, în binar se scrie:

0000 0010b +

<u>0000 0010b</u>

0000 0100b, unde biţii se adună poziţie cu poziţie, iar prin însumarea 1+1 a rezultat un bit de 0 și o unitate (ca bază) s-a transportat mai departe, înspre stânga, exact ca la algoritmul aplicat în clasele primare când adunam în zecimal și transportam un 1 (adică o unitate, și se spunea „1 ducem mai departe"). Similar se procedează și în cazul scăderii în binar, doar că în acest caz se va realiza un împrumut de la cifra dinspre stânga înspre cea din dreapta („aducem un 1", adică o unitate care va fi 2 aici).

a) 02h+02h=04h -> C=0, Z=0, S=0, O=0

b) 03h+7Ch=7Fh -> C=0, Z=0, S=0, O=0

c) 04h+7Ch=80h -> C=0, Z=0, S=1, O=1

unde $O=C \oplus transport_{MSb-1,\ MSb}=0+1=1$

d) 80h+80h=00h -> C=1, Z=1, S=0, O=1

unde $O=C \oplus transport_{MSb-1,\ MSb}=1+0=1$

Pentru numerele reprezentate în convenţia fără semn, indicatorul Carry joacă un rol foarte important în cadrul operaţiilor aritmetice; acesta este folosit ca Borrow flag la operaţii de scădere. De exemplu, când se vor compara doi operanzi folosind instrucţiunea ***cmp opStânga, opDreapta*** se realizează de fapt o scădere a celor doi operanzi și se analizează rezultatul obţinut:

dacă opStânga – opDreapta = 0, atunci se setează ZF=1 și CF este lăsat pe 0, CF=0.

dacă opStânga – opDreapta > 0 atunci ZF=0 și CF=0, iar

dacă opStânga – opDreapta < 0, atunci ZF=0 și CF=1 (a fost împrumut - borrow).

Pentru numerele reprezentate în convenţia cu semn, indicatorii Carry și Overflow au un rol important; de asemenea, SF va interveni pentru a arăta semnul rezultatului. Ca în analiza precedentă,

dacă opStânga – opDreapta = 0, atunci ZF=1 și CF este lăsat pe 0;

dacă opStânga – opDreapta > 0 atunci ZF=0, CF=0, iar OF=SF;

dacă opStânga – opDreapta < 0, atunci ZF=0, CF=1 și OF=not SF.

3.4. Setul de instrucţiuni

Îmbunătăţirile tehnologice care s-au adus de la un procesor la altul de-a lungul timpului, s-au concretizat și în evoluţia sau creșterea *setului de instrucţiuni*.

Termenul x86 denotă *o familie* pentru **arhitectura setului de instrucţiuni** – în engleză ISA (Instruction Set Architecture), în care fiecare arhitectură a fost compatibilă cu generaţia apărută anterior (termenul folosit fiind "backward compatible"); toate acestea au avut la bază procesorul Intel 8086. Acest termen "x86" a apărut datorită numelui diferiţilor succesori ai procesorului Intel 8086, incluzând procesoarele 80186, 80286, 80386 și 80486 (numele care se termină cu "86"). Astfel, de-a lungul timpului, la setul de instrucţiuni x86 au fost adăugate multiple instrucţiuni sau forme noi de execuţie ale instrucţiunilor existente (extensii).

Arhitectura ISA x86 a fost implementată fizic în special în procesoarele Intel, AMD, Cyrix, VIA, dar și altele. În domeniul calculatoarelor de tip computer personal (PC) și laptop, majoritatea din cele vândute pe piaţă au la bază arhitectura x86 (inclusiv familia Apple MacBook, staţiile de lucru cu calcul intensiv – de tip workstation, cloud computing, etc). În schimb, alte categorii de sisteme, precum cele din categoria telefoanelor smart sau tabletelor (în special cele pentru care e nevoie de consum mai redus și dimensiune mai mică) sunt dominate de ***arhitectura ARM*** [sursa: https://en.wikipedia.org/wiki/X86].

Arhitectura ISA **x86** (arhitectura iniţial introdusă de Intel pe 16 biţi și apoi extinsă la 32 biţi) a continuat cu ***x64*** sau ***x86-64*** desemnând arhitectura pe 64 biţi. Aceasta din urmă este o extensie a setului de instrucțiuni x86. S-a adăugat suport pentru registre generale pe 64 de biți, mai multe adrese de memorie virtuală și numeroase alte îmbunătățiri. Specificația originală a fost creată de compania AMD (la procesorul AMD-K8) și a fost implementată în procesoare AMD, Intel, VIA și altele. Când compania AMD a scos pe piaţă primă dată acest tip de arhitectură l-a promovat sub numele „AMD64"; ulterior, Intel a adoptat aceeași arhitectură, dar a folosit numele „IA-32e" și „EM64T", iar mai târziu „Intel 64". În final, termenul adoptat a fost „x86-64" sau simplu **x64** și este folosit ca un termen generic neutru pentru a denumi ***arhitectura pe 64 biţi***.

Avantajul execuţiei unui program scris în asamblare direct pe procesorul real versus execuţia pe un simulator, este accesul la hardware-ul real, adică la tehnologia reală, în mod foarte probabil mult îmbunătăţită faţă de cea simulată. De exemplu, dacă se folosește un procesor din generaţia V care are implementată tehnologia AVX512, unele instrucțiuni (cele specializate pentru acea tehnologie) se vor executa pe 512 biţi deodată, lucru care e greu de imaginat ca fiind posibil folosind un CPU pe 16 biţi.

Figura 3.9. arată evoluţia setului de instrucţiuni în funcţie de procesor. O listă completă a acestora poate fi consultată aici: https://en.wikipedia.org/wiki/X86_instruction_listings

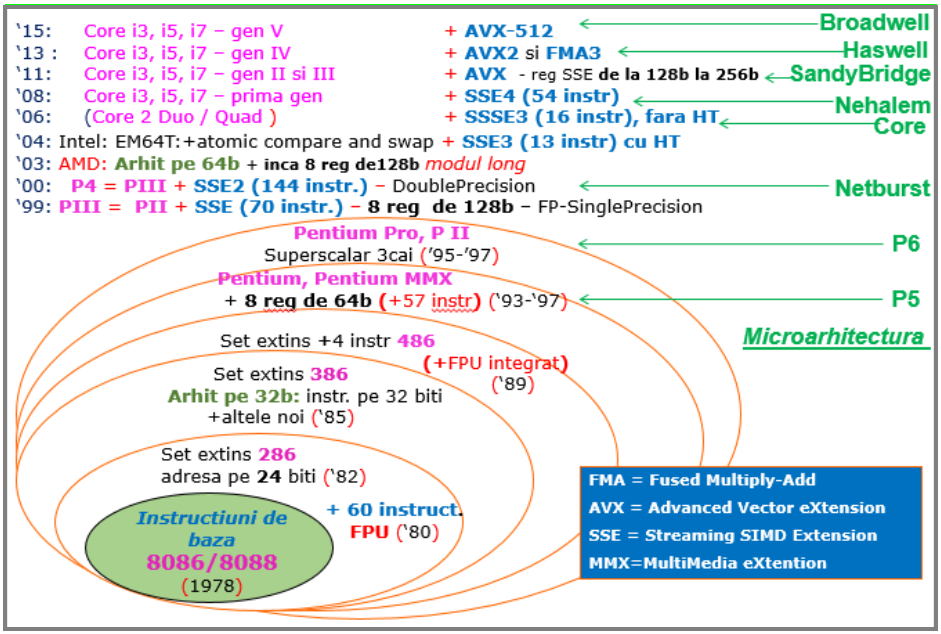


Figura 3.8. Evoluţia microarhitecturii pentru procesoarele din familia x86 și x64

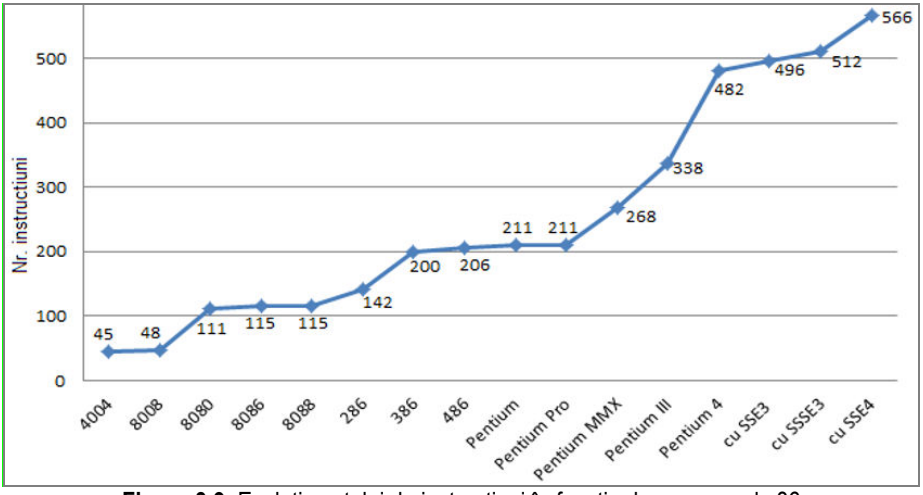


Figura 3.9. Evoluţia setului de instrucţiuni în funcţie de procesorul x86

Procesorul 8086 putea executa un număr de 115 instrucţiuni. Acestea sunt grupate pe diferite categorii, așa cum se poate observa în Tabelul 3.3.

Tabelul 3.3. Setul de instrucţiuni generale ale procesoarelor din familia x86

Instrucţiuni			Observaţii
de transfer	**generale**	MOV, MOVSX, MOVZX, XCHG	MOVSX, MOVZX - 80386↑
	cu stiva	PUSH, POP, PUSHA,POPA, PUSHAD, POPAD	PUSH, POP - 80186↑ PUSHAD, POPAD - 80386↑
	cu acumulatorul	XLAT, IN, OUT	
	pt adrese	LEA, LDS, LES, LFS, LGS, LSS	LFS, LGS, LSS - 80386↑
	cu flaguri	LAHF,SAHF,PUSHF,POPF, PUSHFD,POPFD	PUSHFD, POPFD - 80386↑
aritmetice	**pt adunare**	ADD, XADD, ADC, INC	XADD - 80486↑
	pt scădere	SUB, SBB, DEC	
	pt negare	NEG	
	pt înmulţire	MUL, IMUL	IMUL - diverse forme - 80286↑
	pt împărţire	DIV, IDIV	
	pt comparare	CMP, CMPXCHG, CMPXCHG8B	CMPXCHG - 80486↑, CMPXCHG8B - Pentium↑
	pt corecţia ACC	DAA, AAA, DAS, AAS, AAM, AAD	
	pt extinderea ACC	CBW, CWD, CWDE, CDQ	CWDE , CDQ– 80386↑
pe biţi	**logice**	NOT, AND, OR, XOR	
	de testare/comparare	TEST, BT,BT[S/R/C],BS[F/R],SETcc	80386↑
	de deplasare (shift)	SHL/SAL, SHR, SAR, SHLD,SHRD	SHLD,SHRD – 80386↑
	de rotire (rotate)	ROL, RCL, ROR, RCR	
de manipulare şiruri	**pt operaţii primitive**	MOVSB, MOVSW, MOVSD, movs CMPSB, CMPSW, CMPSD, cmps LODSB, LODSW, LODSD, lods STOSB, STOSW, STOSD, stos SCASB, SCASW, SCASD, scas	MOVSD – 80386↑ CMPSD – 80386↑ LODSD – 80386↑ STOSD – 80386↑ SCASD – 80386↑
	cu şiruri pe port	INSB, INSW, INSD, OUTSB, OUTSW, OUTSD	[IN/ OUT]SB/W – 80186↑ [IN/OUT]SD - 80386↑
	prefixe de repetare	REP,REPE/REPZ, REPNE/REPNZ	
Specifice pt proceduri întreruperi şi salt	**de apel şi revenire din procedură/ întrerupere**	CALL, RET INT, IRET	
	de salt (ne)condiţionat	JMP, J[condiţie]	
	pt controlul buclelor de program	JCXZ, LOOP,LOOPZ/LOOPE,LOOPNZ/LOOPNE	
pt control procesor	**pt controlul explicit al unor indicatori (flaguri)**	CMC, CL[flag], ST[flag], flag=C,D,I	
	pt controlul procesorului	HALT, LOCK, WAIT, NOP	

3.5. Organizarea şi adresarea memoriei

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

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

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

3.5.1. Moduri de adresare a memoriei

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

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

3.5.2. Adresarea memoriei la 8086. Adresarea segmentată

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

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

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

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

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

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

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

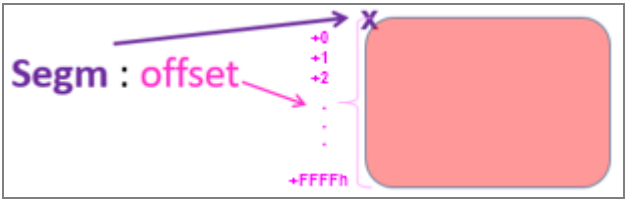


Figura 3.10 Ilustrarea noţiunilor de segment și offset în cadrul memoriei la 8086

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

Adresa Logică = RS:offset (1)

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

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

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

Adresa Fizică $_{20 biţi}$ **=16•RS**$_{16 biţi}$ **+offset**$_{16 biţi}$ (2)

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

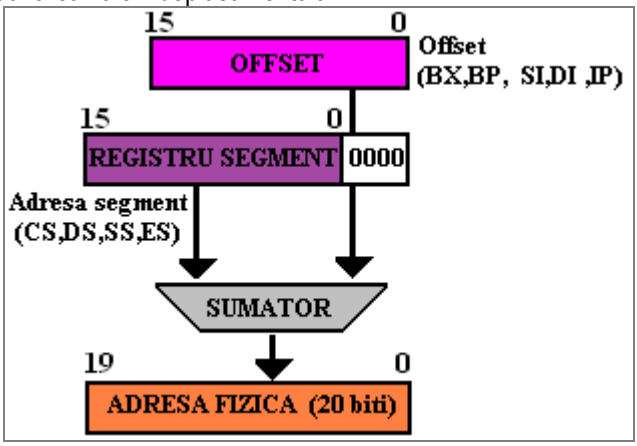


Figura 3.11. Calculul adresei fizice de către BIU

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

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

Mai multe adrese logice pot corespunde aceleiaşi locaţii fizice dacă se află în segmente diferite, ceea ce poate crea anumite probleme de inconsistenţă a datelor.

Altfel spus, o locaţie fizică poate să aparţină unuia sau mai multor segmente, deoarece mai multe adrese logice (folosind segmente diferite) pot conduce înspre aceeaşi adresă fizică.

Exemplu:

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

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

3.6. Moduri de adresare

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

Figura 3.12. Principalele moduri de adresare la 8086

Instrucţiunile care au doi sau mai mulţi operanzi operează întotdeauna de la dreapta spre stânga. Astfel, operandul din dreapta se numeşte operand sursă (specifică datele care vor fi folosite, dar nu şi modificate), iar operandul din stânga se numeşte operand

destinaţie (specifică datele care vor fi folosite şi modificate de către o anumită instrucţiune). Datele imediate (constantele) nu sunt admise ca operand destinaţie.

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

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

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

3.6.1. Adresarea imediată, cu regiștrii și cu porturile

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

Adresare directă

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

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

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

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

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

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

Adresare indirectă

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

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

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

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

3i. **Adresarea indirectă a porturilor prin registrul DX**
Operandul se află în portul de la adresa specificată în registrul DX.
out dx, al ;operandul destinaţie este portul de ieşire a cărui adresă se află în DX
Observaţie: spaţiul de adrese al porturilor este 0...FFFFh (64kocteţi).

3.6.2. Adresarea operanzilor din memorie

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

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

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

Exemplu: Calculaţi adresa fizică şi conţinutul locaţiei respective de memorie după execuţia următoarelor instrucţiuni, presupunând DS=1470h:
mov al, 50h ; registrul AL se încarcă cu valoarea 50h
mov [4320h], al ; adresa fizică va fi AF = 18A20h, deci [18A20h] = 50h.
Observaţie: Nu sunt admise operaţiile pentru care atât sursa cât şi destinaţia sunt operanzi din memorie.
b) Operand cu adresare indirectă la memorie - operanzii cu adresare indirectă utilizează regiştri pentru a indica adrese din memorie. Acest mod de adresare este folosit în manipularea dinamică a datelor (pt că valorile din regiştri se pot modifica).

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

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

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

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

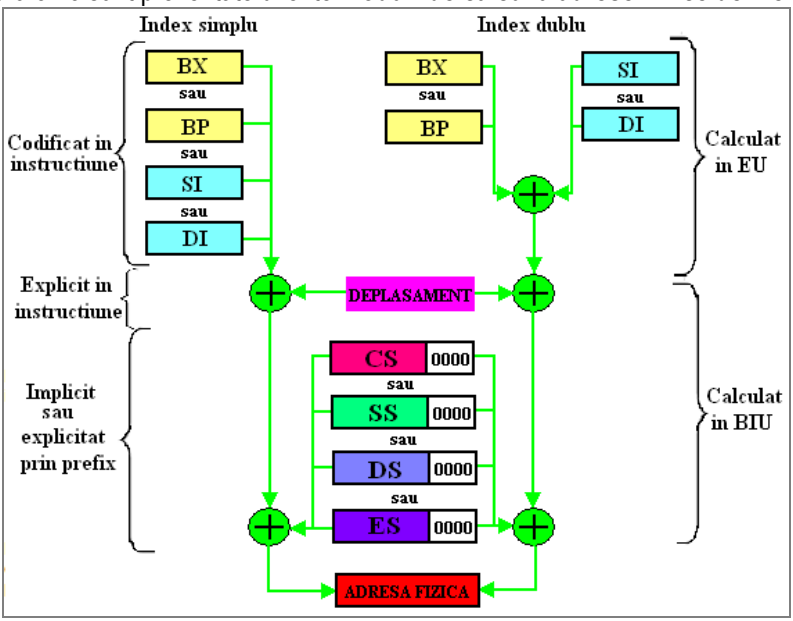


Figura 3.13. Variante de calcul a adresei fizice de memorie

Există 2 posibilităţi:

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

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

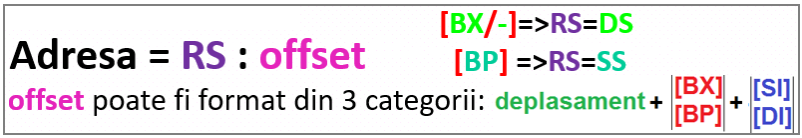


Figura 3.14. Regula de obţinere a **registrului segment** şi a **offsetului** la calculul adresei operandului din memorie

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

Exemplu: Următoarele moduri de specificare sunt echivalente:

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

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

Exemple: Următoarele instrucţiuni sunt **incorecte:**

```
mov ax, sir [bx] [bp]      ; doi regiştri de bază
mov ax, sir [si] [di]      ; doi regiştri index
```

Există mai multe tipuri de adresare:

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

Exemple: mov ax,[bx+5]
mov ax,[si+4]

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

Exemplu: mov ax,[bx+si+7]

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

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

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

3.7. Exerciţii propuse

Exerciţii PRACTICE: (se vor rezolva în şablon)

Set 1 (secţiunea 3.3)

1. Precizaţi dacă următoarele instrucţiuni sunt corecte (legale) şi justificaţi răspunsul:

i) mov AL, 1024	**mov SP, 2**	**mov CF, 1**
ii) mov BL, 5678h	**mov BP, 5**	**mov CF, 0**
iii) mov CL, 0123h	**mov IP, 7**	**stc**
iv) mov DL, 987h	**mov PSW, 12**	**clc**

2. Realizaţi următoarele operaţii la nivel de 8 biţi. Scrieţi valorile în zecimal considerându-le numere *fără semn* şi analizaţi rezultatul obţinut pe cei 8 biţi. Specificaţi şi valoarea flagurilor aritmetice.

i) 25h+79h=_____d+_____d= _____h, C=__; Z=__; S=__; O=__;
ii) 86h+A8h=_____d+_____d= _____h, C=__; Z=__; S=__; O=__;
iii) 68h+D5h=_____d+_____d= _____h, C=__; Z=__; S=__; O=__;
iv) 4Ch+7Dh=_____d+_____d= _____h, C=__; Z=__; S=__; O=__;

3. Realizaţi următoarele operaţii la nivel de 8 biţi. Scrieţi valorile în zecimal considerându-le numere *cu semn* şi analizaţi rezultatul obţinut pe cei 8 biţi. Specificaţi şi valoarea flagurilor aritmetice.

i) 25h+79h=_____d+_____d= _____h, C=__; Z=__; S=__; O=__;
ii) 86h+A8h=_____d+_____d= _____h, C=__; Z=__; S=__; O=__;
iii) 68h+D5h=_____d+_____d= _____h, C=__; Z=__; S=__; O=__;
iv) 4Ch+7Dh=_____d+_____d= _____h, C=__; Z=__; S=__; O=__;

Set 2 (secţiunea 3.5)

1. Daţi exemplu de 3 adrese logice ce se pot scrie pentru adresa fizică următoare:
i) 23456h; ii) 65432h, iii) 87654h, iv) 45678h.

---

2. Să se calculeze adresa fizică ce corespunde adresei logice
i) 89ABh:89ABh, ii) 56CDh:56CDh, iii) 43EFh:43EFh, iv) 12ABh:12ABh; AF=_____

3. Să se calculeze componenta offset corespunzătoare adresei fizice menţionate dacă se cunoaşte componenta segment.

i) AF=10000h, segm.=1000h; offset=_____h; Dar pt AF= 1FFFFh ? offset=_____h
ii) AF=20000h, segm.=2000h; offset=_____h; Dar pt AF= 2FFFFh ? offset=_____h
iii) AF=30000h,segm.=3000h; offset=_____h; Dar pt AF= 3FFFFh ? offset=_____h
iv) AF=40000h,segm.=4000h; offset=_____h; Dar pt AF= 4FFFFh ? offset=_____h

4. Să se calculeze componenta segment corespunzătoare adresei fizice 0ABC10h dacă se precizează componenta offset:

i) offset = 600h; segment= _____h; ii) offset= 800h; segment= _____h;
iii) offset = 0A00h; segment= _____h; iv) offset= 900h; segment= _____h;

5. Verificaţi dacă adresa fizică menţionată aparţine segmentului pentru care se știe componenta segment:

i) AF=31FFFh, segment= 2200h. R:_____ Dar adresa fizică 21000h? R:_____
ii) AF=41FFFh, segment= 3200h. R:_____ Dar adresa fizică 31000h? R:_____
iii) AF=51FFFh, segment= 4200h. R:_____ Dar adresa fizică 41000h? R:_____
iv) AF=61FFFh, segment= 5200h. R:_____ Dar adresa fizică 51000h? R:_____

Set 3 (secţiunea 3.6)

1. Pentru DS=1200h, SI=1234h, DI=4321h, AX=2ABCh şi BX=1A3Bh specificaţi adresa locaţiei de memorie accesată de fiecare dintre instrucţiunile următoare:

i) mov [di], ax ; AF=_____
ii) mov ax, [si+bx] ; AF=_____
iii) mov ax, [si+bx+2] ; AF=_____
iv) mov [di+5], ax ; AF=_____

2. 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]

3. Pentru SS=3F00h, CS=5A00h, ES=1400h, DS=1200h, SI=1234h, DI=340Ch, AX=2ABCh şi BX=1A3Bh, BP=1300h, menţionaţi tipul de adresare folosit şi specificaţi adresa locaţiei de memorie accesată de fiecare dintre instrucţiunile următoare, folosind modelul:

mov AX, [BX] ; s-a folosit adresarea bazată, primul operand este registrul AX, iar al doilea operand este perechea de octeţi (cuvântul) din memoria principală, din segmentul curent de date (specificat de DS), aflat la offset-ul conţinut în registrul BX (adică operandul este cuvântul din memorie,
de la adresa 1200h : 1A3Bh (partea LOW) şi 1200h : 1A3Ch partea HIGH)

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

4. Urmăriţi modele :

ADD AX, [BX] ; al doilea operand este ~~octetul~~ / perechea de octeţi (cuvântul) din memoria principală, din segmentul curent de date (specificat de DS), aflat la offset-ul dat de conţinutul din registrul BX.

ADD [SI], AL ; primul operand este octetul / ~~perechea de octeti~~ din memoria principală, din segmentul de date curent (specificat de DS), aflat la offsetul conţinut în registrul SI.

ADD AX, DS: [BP+2] ; al doilea operand este cuvântul din memorie, din segmentul de date curent (specificat explicit de DS), aflat la offset-ul egal cu suma dintre conţinutul registrului BX și deplasarea 2 (de tip imediat).

ADD TAB [DI], AL ; primul operand este octetul din memorie, din segmentul de date curent (specificat de DS), aflat la offset-ul egal cu suma dintre conţinutul registrului DI și deplasarea TAB (de tip imediat).

ADD AX, [BX] [SI] ; al doilea operand este cuvântul din memorie, din segmentul de date curent, aflat la offset-ul egal cu suma conţinuturilor registrelor BX și SI.
MOV MATR [BX] [SI], AX ; primul operand este cuvântul din memorie, din segmentul de date curent, aflat la offset-ul egal cu suma conţinuturilor registrelor BX și SI cu deplasarea MATR (de tip offset).
5. Comentaţi destinaţia din următoarele instrucţiuni după modelul exercitiului 4 și apoi precizaţi dacă sunt corecte aceste instrucţiuni şi justificaţi răspunsul:

i1) mov ax, sir [BP+BX]	**i2) mov AX, 7[BP]**
ii1) mov AX, var [BX]	**ii2) mov AX, [SI+DI]**
iii1) mov AX, sir [BP+SI]	**iii2) mov AX,[BP+5]**
iv1) mov AX, sir [SI+DI]	**iv2) mov AX, [SI+BP]**

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

i1) mov AX, [bx+3],	**i2) mov AX, 4[bx+si]**
ii1) mov AX, [si]+10h,	**ii2) mov AX, 5[bp]**
iii1) mov AX, [di+8],	**iii2) mov AX,[bx][si]**
iv1) mov AX, 5[bp+si],	**iv2) mov AX, [5][di]**

7. i), ii) Ştiind că CS=1234h, SS=2F00h, IP=2300h şi SP=3456h, specificaţi adresele logice şi fizice pentru elementul din vârful stivei şi instrucţiunea curentă.

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

8. Scrieţi instrucţiunile corespunzătoare următoarelor operaţii:
i) conţinutul locaţiei de memorie 50h să fie mutat în locaţia de memorie 60h:

ii) conţinutul locaţiei de memorie 20h să fie mutat în locaţia de memorie 40h:

iii) în registrul AL să fie mutat conţinutul din memorie de la adresa 23h, să se adune 7 la acest conţinut și rezultatul să fie apoi mutat în memorie la adresa 21h:

iv) în registrul AX să fie mutat conţinutul din memorie de la adresa 10h, să se adune 5 la acest conţinut și rezultatul să fie apoi mutat în memorie la adresa 12h:

Capitolul 4. Avantajele și dezavantajele folosirii unui simulator de microprocesor

După ce în capitolele anterioare am prezentat noţiunile de bază necesare reprezentării informaţiei într-un SC, am arătat aspectele arhitecturale ale procesorului 8086 și am subliniat cele mai importante aspecte ce trebuie considerate atunci când lucrăm cu acesta, în capitolul de faţă vom vedea ce tipuri de simulatoare există pentru procesorul 8086 și care e justificarea alegerii lui EMU8086.

4.1. Simulatoare disponibile pentru arhitectura 8086

Dintre simulatoarele existente ce pot ajuta la înţelegerea funcţionării procesorului 8086 sau la modul general al procesoarelor din familia x86, cele mai des întâlnite sunt:

EMU8086 – unul dintre cele mai puternice simulatoare 8086, fiind disponibil atât pentru Windows Vista, XP, cât și 7, 8, 10 la adresa http://www.emu8086.com/;

SMS32v50 – e un simulator aproape la fel de bun ca EMU8086, dar nu s-a mai actualizat de ceva vreme http://www.softwareforeducation.com/sms32v50/ și nici nu dispune de toate facilităţile implementate în EMU8086; e un predecesor al lui EMU8086;

iEmu8086 – este un simulator foarte asemănător cu EMU8086, chiar și din punct de vedere al interfeţei utilizator http://i8086emu.sourceforge.net/, fiind puţin mai mult înclinat înspre aria microcontrolerelor (folosește PIT, PIC, afișaj cu 7 segmente, butoane, leduri); poate fi instalat sub Windows dar și sub Linux;

Simulator 8086 pe Android – disponibil pentru telefoane/ dispozitive smart la adresa https://play.google.com/store/apps/details?id=com.nextedge.microProcesorSimulator. Este de fapt o aplicaţie Android în care se poate urmări documentaţie despre instrucţiunile 8086 și se pot simula programe simple; există chiar și câteva programe model care pot fi consultate și executate;

Simulator 8086 online – se găsește la adresa http://carlosrafaelgn.com.br/Asm86/ și este chiar mai mult decât un simulator 8086, deoarece arhitectura simulată este pe 32 biţi; ca și în simulatoarele prezentate mai devreme, se poate consulta memoria, flagurile, regiștrii după execuţia fiecărei instrucţiuni;

OllyDbg –încarcă un cod executabil care se poate depana/ testa, dar nu este potrivit pentru scrierea codului sursă și obţinerea lui în formă de fișier executabil;

(din aceeași categorie ar mai putea fi menţionat ***WinDbg***).

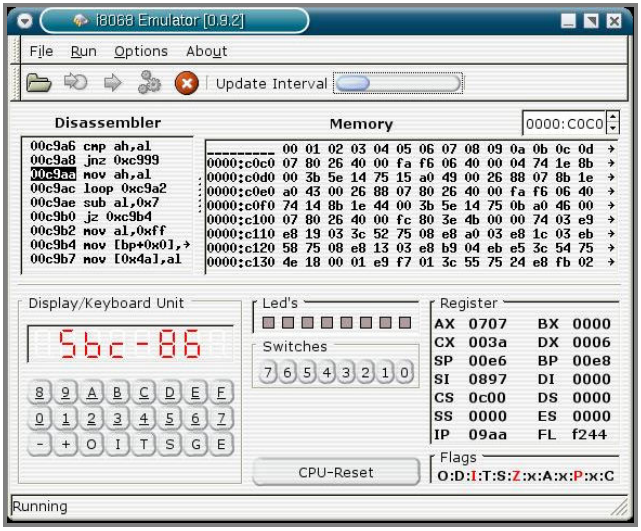


Figura 4.1. Simulator iEmu8086 pe 16 biţi

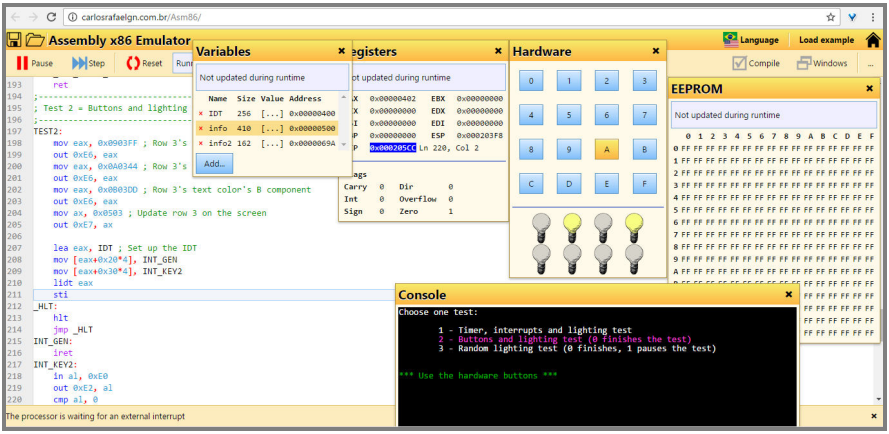


Figura 4.2. Simulator x86 pe 32 biţi online

Dintre toate acestea, pentru execuţia programelor din acest material, la nivel de începător, voi recomanda EMU8086 – pentru instalare ***EMU8086***, vă rog să urmăriţi informaţiile de la adresa http://www.emu8086.com/.

EMU8086 e ușor de folosit, foarte vizual, cu multe avantaje atunci când se dorește doar inițiere în programarea în limbajul de asamblare al procesoarelor din familia x86 și deși este simplu de utilizat, este în același timp complex: permite o mulţime de facilităţi, inclusiv lucrul cu întreruperi, cu periferice, cu PSP, cu memoria video.

4.2. Instalarea simulatorului

Pentru execuţia și înţelegerea programelor, nu sunt necesare cunoștinţe avansate despre arhitectura sau modul de funcţionare al procesorului: EMU8086 este foarte intuitiv. În plus, există o mulţime de tutoriale disponibile online cu acest simulator, de exemplu cele găsite la adresa:http://www.yecd.com/os/8086 %20assembler%20tutorial %20for%20beginners%20(part%201).htm. Totuși, cea mai indicată modalitate este prin urmărirea documentaţiei cu care vine EMU8086 la pachet atunci când se instalează:

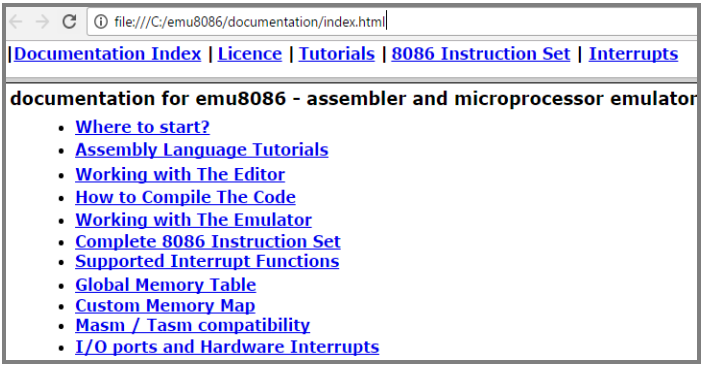


Figura 4.3. Documentaţie EMU disponibilă la instalare

O altă facilitate oferită de EMU8086 este că, odată instalat pe sistem, se instalează o mulţime de programe (*examples*), care pot fi folosite ca model pentru dezvoltarea de noi aplicaţii. Acest software este disponibil în variantă trial, poate fi achiziţionat fără costuri și poate fi utilizat pentru o perioadă de graţie de 4 săptămâni.

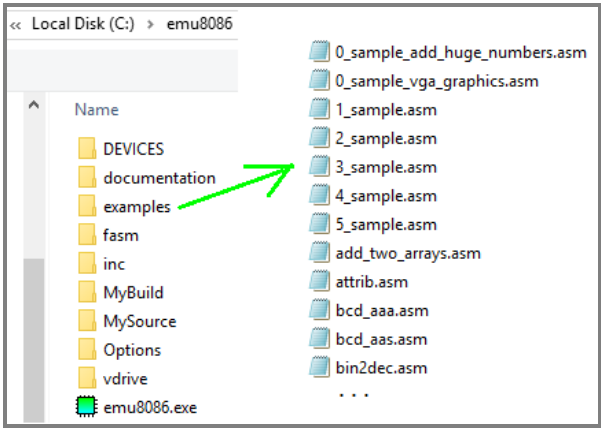


Figura 4.4. Programe model disponibile la instalarea EMU8086

La expirarea acestei perioade, pentru suma de 5$ pentru utilizator individual se poate obţine acces nelimitat. Varianta trial ar trebui să fie suficientă pentru parcurgerea materialului din acest volum; totuși, recomand înscrierea pe site pentru cei 5$ și obţinerea accesului nelimitat la facilităţile oferite de acest simulator, fiind de un real ajutor în procesul de asimilare și validare al cunoștinţelor de programare în limbajul de asamblare al procesoarelor din familia x86.

Pentru a obţine un program executabil, trebuie să avem acces la codul sursă (ce conţine instrucţiunile pe care vrem să le execute procesorul) – de exemplu unul din programele din Figura 4.4, scris ca fișier de tip .asm. Obţinerea acestui fișier poate fi realizată și cu un simplu editor de text (gen Notepad++), scrierea instrucţiunilor dorite și salvarea lui ulterioară ca fișier cu extensia .asm. Acest fișier de tip asm trebuie apoi "asamblat" (adică trebuie folosită o unealtă numită "asamblor" care va ști să traducă instrucţiunile scrise de către programator cu ajutorul mnemonicilor - de exemplu se folosește *mul* pentru operaţia de înmulţire, *multiply* în engleză), într-un limbaj pe care procesorul să-l înţeleagă. Aceasta se poate realiza în EMU simplu, cu o comandă numită "Emulate". În urma operaţiei de emulare, în EMU se obţine un fișier de tip com sau exe, acestea fiind fișiere executabile.

Un posibil dezavantaj al folosirii unui simulator este constituit de perioada în care acesta poate fi considerat un produs robust la erori, fiabil; de asemenea, se pune problema credibilităţii modului de implementare al mecanismelor interne. Este asigurată actualizarea produsului? Cum există garanţia că e un produs suficient de bine testat, fără probleme (bug-uri)? Nu există o astfel de garanţie, în general, un simulator fiind validat în timp, ca orice aplicaţie complexă. EMU8086 este deja destul de bine validat și actualizat, bucurându-se de un real succes.

Totuși, datorită faptului că este foarte ușor de folosit, atât de vizual, mulţi riscă să trateze aspectele programării în mod superficial; problema principală este că mulţi dintre cei care îl utilizează nu învaţă regulile de programare; scriu programul în mod haotic, uneori merge, alteori nu mai merge și rezultă astfel o inconsistenţă a învăţării. De exemplu: dacă datele nu sunt definite în cadrul zonei de date, pot apărea probleme (în special la definirea șirurilor în memorie).

Folosirea unui simulator ar trebui să fie doar primul pas, de acomodare cu facilităţile oferite de acel procesor; indicată ar fi apoi trecerea la scrierea de programe în editor de text, cu folosirea mai multor tipuri de asambloare, precum Turbo Assembler (TASM), Microsoft Macro Assembler (MASM), Netwide Assembler (NASM), YASM (care este de fapt o reactualizarea a NASM), Open Watcom Assembler (WASM), etc; după aceea, indicat ar fi studiul arhitecturii recente existente în procesoarele x64 la care capabilităţile oferite sunt mult îmbunătăţite (de exemplu setul MMX, SSE, AVX-2, FMA3, etc). De considerat că toate acestea înseamnă 38 de ani de avans tehnologic.

Capitolul 5. Prezentarea simulatorului

În acest capitol am urmărit înţelegerea principalelor aspecte arhitecturale ale procesoarelor din familia 80x86 prin lucrul cu simulatoarele **EMU8086** și **SMS32v50**. Simulatorul EMU este cel asupra căruia se va insista în această carte, deoarece se dorește acomodarea cu arhitectura pe 16 biţi, însă SMS a fost abordat din prisma comparării unui procesor pe 16 biţi cu unul pe 8 biţi. Astfel, atunci când se face referire simplu la termenul „simulator”, ne referim în mod implicit la EMU8086.

Dacă nu aţi instalat până acum EMU8086, este indicat să realizaţi aceasta acum, urmărind pagina http://www.emu8086.com/.

5.1. Prezentarea simulatorului EMU8086

Se reamintește că simulatorul execută *programele* pe o Maşină Virtuală, emulând deci hardware-ul real: ecranul monitorului, memoria şi dispozitivele de intrare/ ieşire; toate acestea pot fi utilizate, accesate sau vizualizate în EMU.

Programele pot fi executate ca un microprocesor 8086 real: codul sursă este asamblat şi executat de către emulator pas cu pas sau în mod continuu. Execuţia de tip pas cu pas oferă posibilitatea de a urmări modificările apărute imediat după execuţia unei instrucţiuni în regiştri, flag-uri şi memorie. De asemenea, variabilele folosite în program pot fi vizualizate în diverse forme.

După cum se poate urmări în Figura 5.1, 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**).

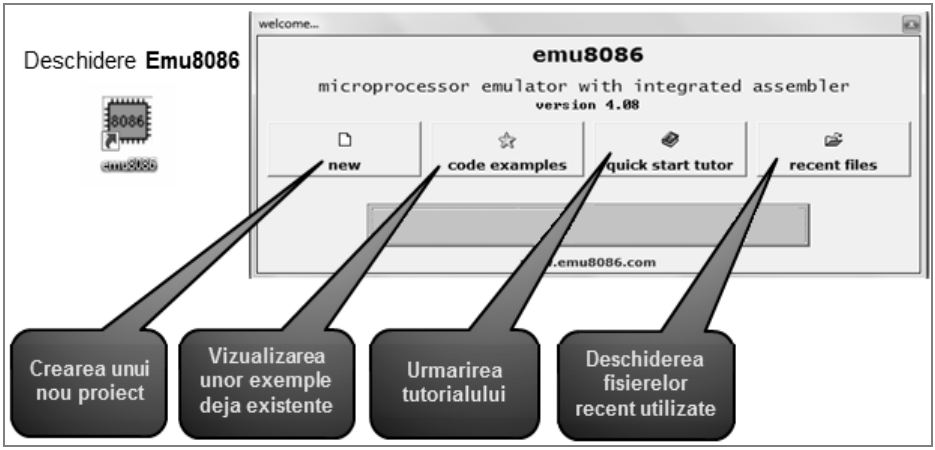


Figura 5.1. Fereastra principală a simulatorului Emu8086

5.1.1. Arhitectura emulată

Aşa cum se poate urmări în Figura 5.2., un *sistem de calcul* cuprinde în general cele 3 tipuri de componente:

(1) una sau mai multe *unităţi centrale de procesare* (UCP), având regiştrii şi ALU,

(2) *memorie* şi

(3) *dispozitive de intrare-ieşire*.

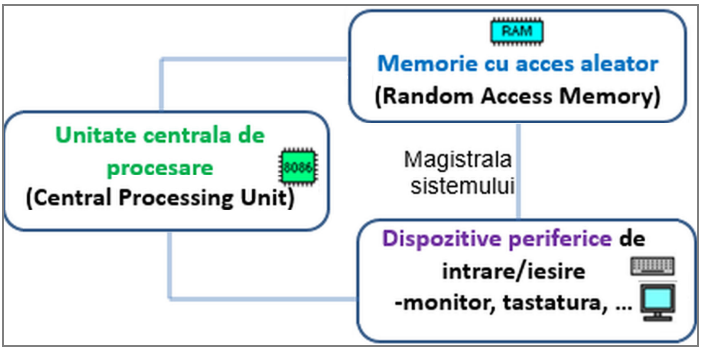


Figura 5.2. Componentele principale ale unui sistem de calcul emulat în EMU8086

(1) Unitatea centrală de procesare (CPU) din EMU este "creierul" sistemului, având structura prezentată în Figura 5.3. Toate calculele, deciziile şi mutările de date se realizează aici, deoarece CPU are în componenţă locaţii de stocare specifice numite regiştri şi o unitate aritmetică şi logică (ALU), similar cu arhitectura reală a 8086 (aşa cum am prezentat în *Capitolul 3*). Datele sunt luate din regiştri, procesate de ALU, iar rezultatele pot fi stocate tot în regiştri (sau pot fi depuse în memorie).

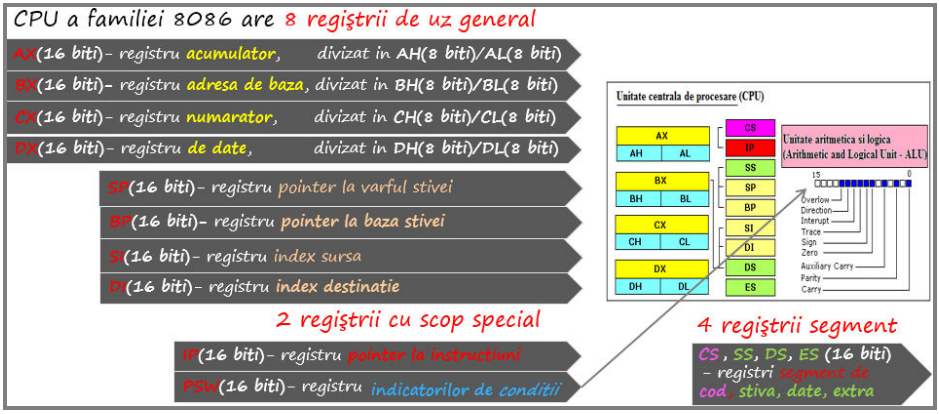


Figura 5.3. Structura Unităţii Centrale de Procesare din EMU8086

Similar cu un CPU al familiei 8086 (observaţi analogia cu arhitectura pe 16 biţi prezentată în *Capitolul 3*), EMU8086 prezintă **8 *regiştri de uz general*** (AX, BX, CX, DX, SI, DI, BP, SP); mărimea lor fiind de 16 biţi, ei pot păstra numere fără semn în domeniul 0..65535 sau numere cu semn în domeniul –32768...+32767, **4 *regiştri segment*** (CS, DS, SS, ES) și **2 *regiştri cu scop special*** (IP, PSW). Toti regiștrii prezentaţi în *Capitolul 3* sunt emulaţi de EMU8086, la fel și registrul de flaguri. Figura 5.4 arată modul cum se prezintă aceștia în simulatorul EMU.

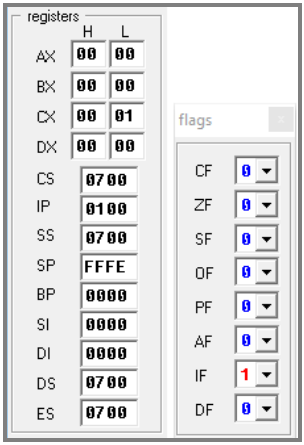


Figura 5.4. Modul de ilustrare al regiștrilor în EMU8086

(2) Memoria RAM: CPU poate accesa până la 1 MB de memorie RAM, exact ca 8086 real, adresele RAM fiind date uzual între paranteze drepte; de exemplu, [7Ch] se citeşte “datele din memorie de la *locaţia* sau *adresa* 7Ch”. Vom vedea că de fapt, 7Ch este doar o parte a adresei, și anume offsetul, așa cum am prezentat în *Capitolul 3* la formarea adresei (în accesarea memoriei) folosind segmente.

(3) Dispozitivele de intrare/ ieşire sau ***perifericele*** sunt diverse, în funcţie de necesităţile utilizatorului şi ale sistemului de calcul, dar în general putem spune că utilizând un periferic de intrare vom achiziţiona informaţie în interiorul S.C., iar folosind un periferic de ieşire vom genera informaţie în exteriorul S.C. (sistemului de calcul). EMU8086 include câteva dispozitive externe virtuale (acestea pot fi modificate sau clonate, codul lor sursă fiind disponibil) prin care se pot realiza diverse experimente, aşa cum vom vedea în *Capitolul 11*.

Magistralele: Simulatorul are un BD de 16 biţi și un BA de 20 biţi, exact ca 8086; biţii de date sunt multiplexaţi împreună cu cei mai puţin semnificativi 16 biţi ai adresei.

Ceasul sistem constă în impulsuri periodice generate astfel încât componentele să se sincronizeze între ele, simulatorul lucrând cu o viteză de câteva instrucţiuni pe secundă, ajustabilă în limite mici prin utilizarea unui slider.

5.1.2. Caracteristicile simulatorului

- CPU de 16 biţi,
- Asamblor;
- Help on-line;
- se pot adresa 2^{16} porturi I/O- periferice simulate pe unele dintre aceste porturi;
- rulare pas cu pas sau rulare continuă a programului;
- posibilitatea de modificare a ceasului procesor (nu cel real, binenţeles).

5.1.3. Utilizarea simulatorului

Dacă în fereastra principală (Figura 5.1) se va selecta ***code examples,*** apoi opţiunea *more examples* şi se alege fişierul *HelloWorld.asm*, va apare fereastra din Figura 5.5.

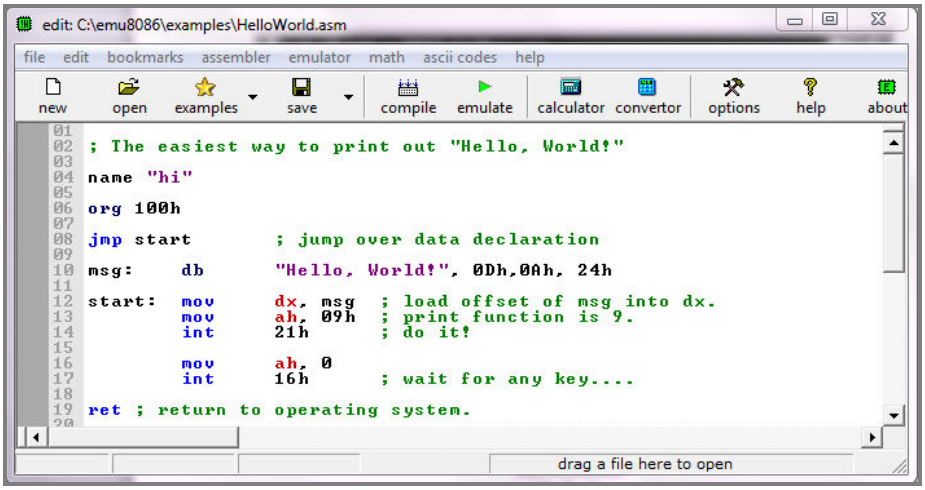


Figura 5.5. Fereastra principală de editare a codului sursă

În cadrul Figurii 5.5 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*. De asemenea, sunt disponibile opţiunile *new, open, examples, save, compile, emulate, calculator, convertor, options, help*. Dacă se alege opţiunea *examples* -> ***More examples*** se vor deschide fişierele existente în directorul c:\emu8086***examples*** (sau unde s-a instalat) şi de unde s-a ales mai devreme fişierul *HelloWorld.asm*. Tot din fereastra din Figura 5.5, din meniu, dar cu opţiunea *open* se vor deschide fişierele existente în directorul c:\emu8086***MySource***.

Pentru a scrie şi a rula un ***program nou***, se va folosi opţiunea ***new*** în fereastra din Figura 5.1 şi se va selecta un șablon de tip *COM* sau *EXE*, aşa cum arată Figura 5.6.

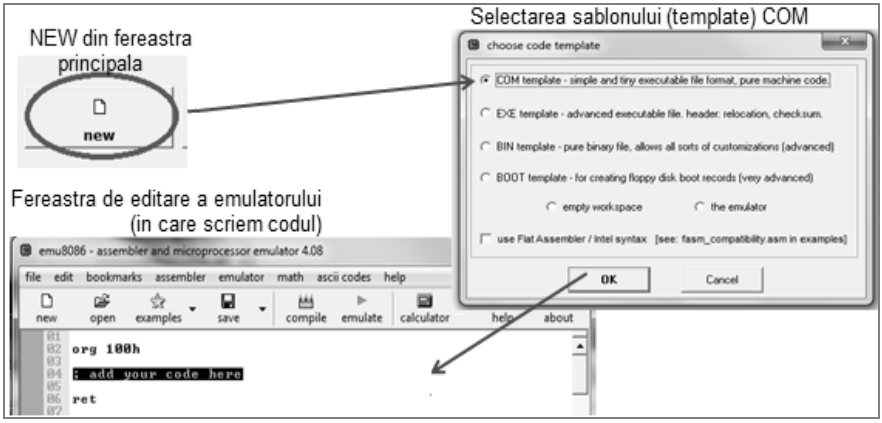


Figura 5.6. Fereastra de editare a codului în cadrul simulatorului

Codul scris se numeşte limbaj de asamblare (*.asm), iar trecerea lui într-un limbaj care să fie înţeles de către CPU în cazul EMU8086, se obţine prin compilare, cu opţiunea *compile*, ca în Figura 5.7. În urma acestei operaţii, va rezulta un fişier de tip .com, iar dacă se doreşte încărcarea codului în emulator, se va folosi opţiunea *emulate*.

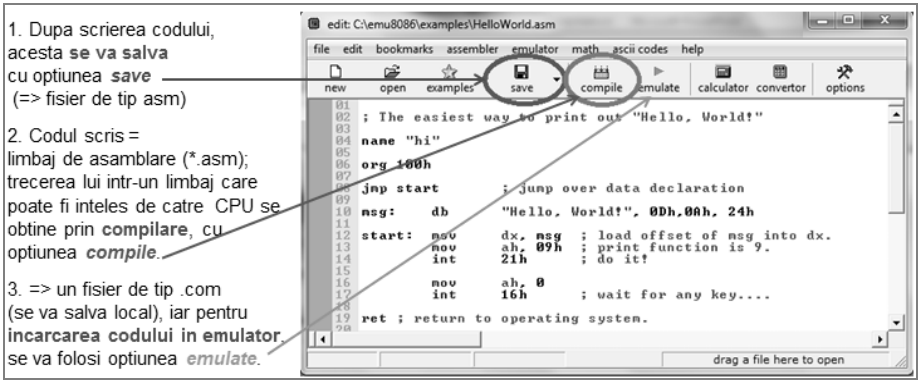


Figura 5.7. Ordinea de efectuare a operaţiilor asupra unui fişier .asm

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

Dacă se încarcă programul în emulator, va apare fereastra din Figura 5.8 în care se pot urmări în partea de jos, dinspre stânga spre dreapta: în prima subfereastră - regiştrii emulatorului, în următoarea subfereastră - adresa fizică: valoarea ei în hexa, în zecimal şi codul Ascii corespunzător, iar în ultima subfereastră: codul sursă al programului în limbaj de asamblare, înainte de a fi asamblat. Viteza de execuţie (ceasul CPU) poate fi selectată cu ajutorul cursorului (de tip slider) din colţul dreapta sus.

Cu un dubluclick în caseta corespunzătoare regiştrilor, se va deschide fereastra *Extended value viewer* (Figura 5.9 stânga) ce conţine valoarea din registru în cele 3 sisteme de reprezentare: hexazecimal, binar şi zecimal. Prin intermediul acestei ferestre, valoarea din registru poate fi modificată direct, în timpul rulării. Cu dubluclick asupra unei zone din memorie, se va lista cuvântul din memorie aflat la locaţia selectată (Figura 5.9 dreapta). Trebuie subliniat faptul că octetul mai puţin semnificativ se găseşte la adrese mai mici așa cum arată convenţia Little Endian.

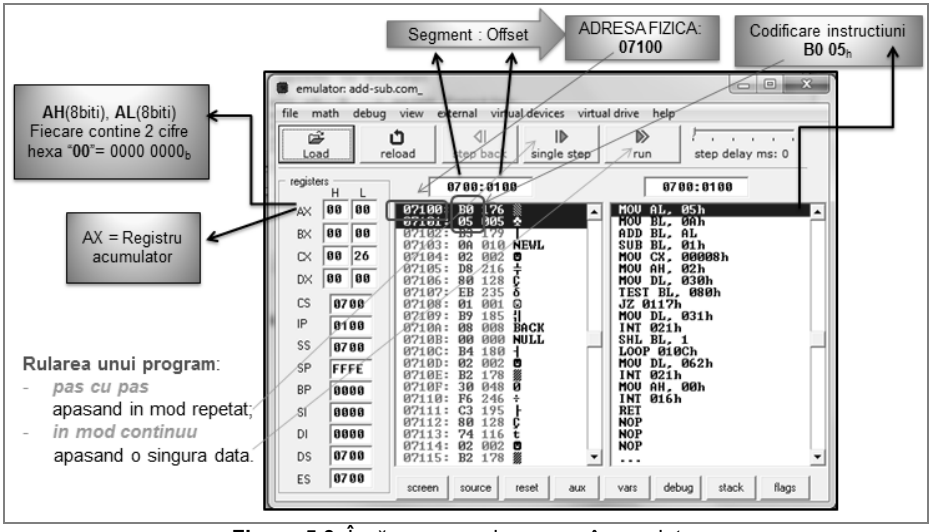


Figura 5.8. Încărcarea unui program în emulator

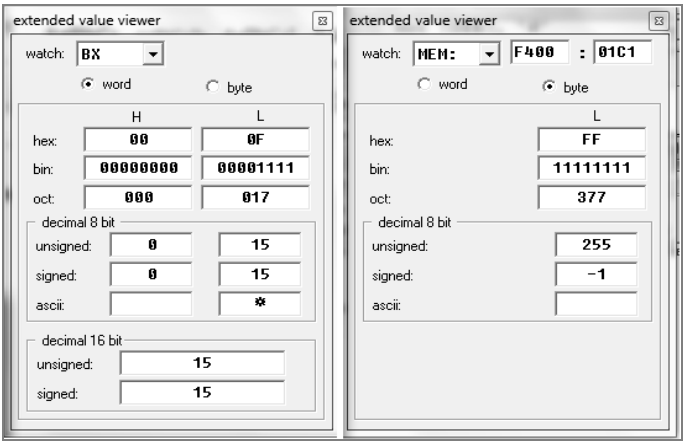


Figura 5.9. Fereastra de vizualizare şi modificare a regiştrilor (stânga) sau a unei zone din memorie (dreapta)

În urma execuţiei fiecărei instrucţiuni, se poate urmări modificarea conţinutului regiştrilor din CPU. De asemenea, există posibilitatea de a vizualiza conţinutul memoriei, al ALU, starea flag-urilor, aşa cum se poate urmări în Figura 5.10. Selectarea informaţiei dorite se realizează din meniul *View* al emulatorului.

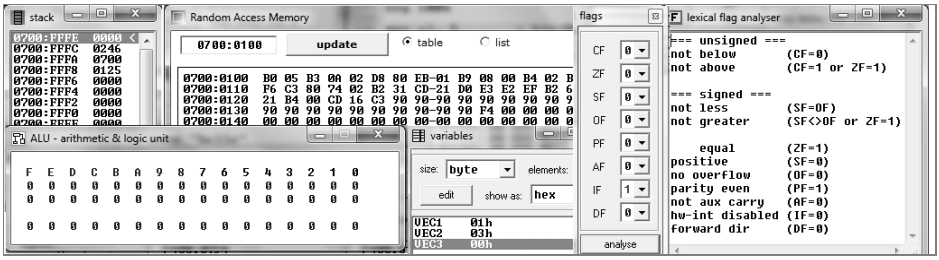


Figura 5.10. Ferestre de vizualizare a conţinutului stivei, memoriei, al ALU, al variabilelor definite în memorie şi al flag-urilor

Ecranul emulatorului (ilustrat în Figura 5.11) poate fi folosit pentru datele de ieşire (modul color este şi el suportat) şi se obţine tot din meniul View, din fereastra prezentată în Figura 5.8.

Figura 5.11. Fereastra de vizualizare a ecranului (emulator screen)

Din meniul Math, se pot deschide ferestrele ilustrate în Figura 5.12, corespunzătoare unui **calculator** (*expression evaluator*) ce poate fi folosit pentru operaţii logice şi aritmetice cu valori hexazecimale, octale, binare şi zecimale şi respectiv unui **convertor**, prin care numerele pot fi convertite dintr-o bază de numeraţie în alta.

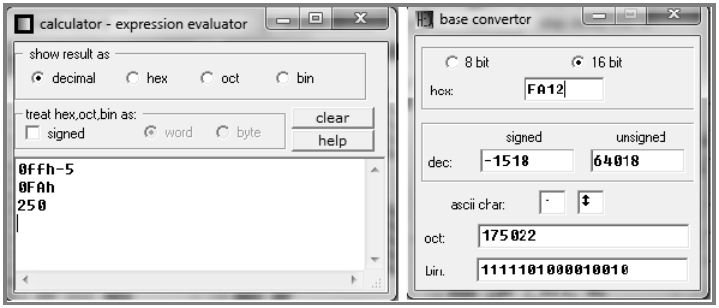


Figura 5.12. Ferestrele calculatorului şi convertorului

5.2. Prezentarea simulatorului SMS32v50

Simulatorul SMS32V50 emulează funcţionarea unui ***procesor de 8 biţi***, analog octetului low al chipurilor din familia de procesoare Intel 8086 sau similar unui procesor pe 8 biţi, de exemplu procesorul 8008.

5.2.1. Caracteristicile principale ale simulatorului

- CPU de 8 biţi
- Un număr de 16 porturi I/O, dar nu sunt folosite toate
- Existenţa unor periferice simulate pe porturile 1...5
- Asamblor
- Rulare pas cu pas sau continuă a programului
- Întreruperea 02 controlată de un circuit timer (simulat)
- Posibilitatea de modificare a ceasului procesor

Simulatorul se compune din: CPU, doar 256 octeţi (bytes) de memorie RAM şi 16 porturi de intrare/ieşire.

Regiştrii de uz general – *sunt doar pe 8 biţi;* regiştrii ***AL, BL, CL, DL*** au dimensiunea de 8 biţi şi pot păstra 256 valori diferite: numere fără semn în domeniul [0;+255] sau numere cu semn în domeniul [–128;+127].

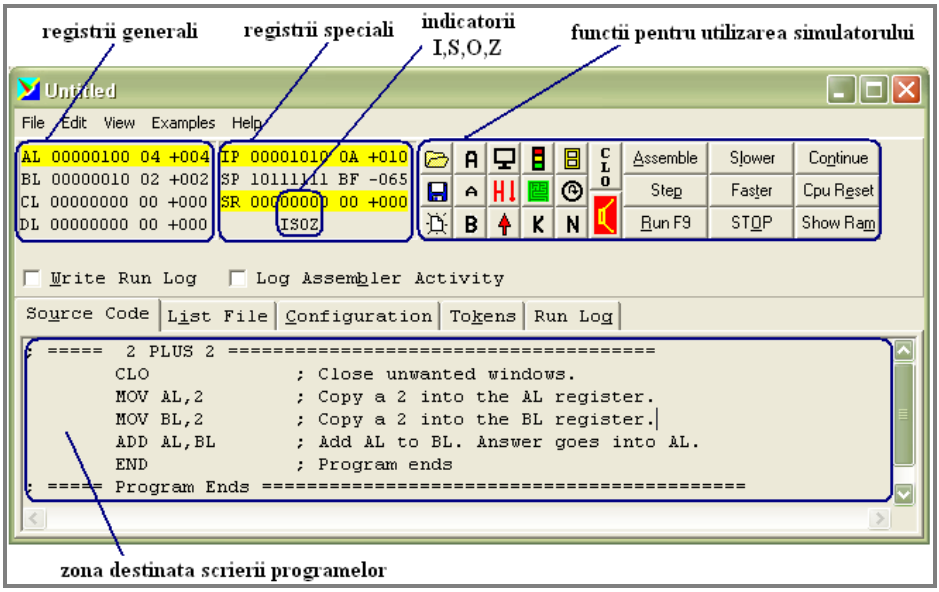


Figura 5.13. Interfaţa simulatorului SMS32v50

Regiştrii speciali: Similar cu EMU8086, ***IP (Instruction Pointer)*** conţine adresa instrucţiunii ce urmează să se execute, iar după ce s-a executat, IP este incrementat automat pentru a pointa instrucţiunea următoare. Registrul ***PSW (Program Status Word)*** conţine şi aici flagurile ce raportează starea procesorului, dar în cazul simulatorului SMS32v50, acest registru se numeşte ***SR (Status Register)*** şi conţine (a se urmări Figura 5.13) flagurile I, S, O şi Z. Aceştia indică starea procesorului după execuţia fiecărei instrucţiuni. Registrul ***SP (Stack Pointer)*** este tot pointer la zona de stivă, acesta arătând următoarea locaţie liberă de pe stivă.

Memoria RAM a simulatorului are o dimensiune de doar 256 octeţi: adresele sunt de la 0 la 255 în zecimal sau de la 00 la FF în hexazecimal, aşa cum se prezintă în Figura 5.14. Trebuie subliniat faptul că valorile în simulatorul SMS sunt considerate implicit în hexazecimal și nu trebuie specificat sufixul h.

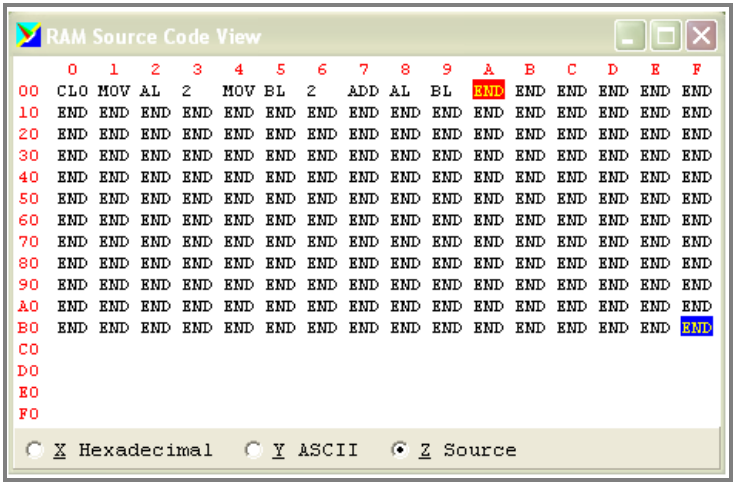


Figura 5.14. Memoria RAM în cazul SMS32v50

5.2.2. Utilizarea simulatorului

La pornirea simulatorului, va apărea o fereastră asemănătoare celei din Figura 5.13 unde se observă conţinutul regiştrilor generali, a regiştrilor speciali, a indicatorilor I, S, O, Z după executarea unui program aritmetic simplu ce adună două numere (stocate în regiştrii AL şi BL), rezultatul depunându-l în registrul AL. În aceeaşi figură se mai poate observa zona destinată scrierii programelor, precum şi funcţiile pentru utilizarea simulatorului. Pentru a scrie un program nou, se va deschide o nouă fereastră din meniul ***File***, folosind comanda ***New***, sau se va apăsa butonul


.

Scrierea programelor (pentru început, este indicată urmărirea exemplelor din tutorial) se va realiza în zona corespunzătoare, aşa cum se poate urmări în Figura 5.13. Se menţionează că în această fereastră codul scris nu este Case Sensitive, deci este posibilă scrierea instrucţiunilor cu litere mari sau mici. Trecerea programului din limbaj de asamblare într-un limbaj care să fie înţeles de către CPU în cazul SMS se obţine prin asamblare, apăsând butonul

(ALT+A)

Execuţia unui program se poate realiza, similar cu EMU, *pas cu pas:* chiar este indicată această urmărire la rularea primelor programe pentru a observa şi înţelege funcţionarea microprocesorului: încărcarea diferiţilor regiştri, modificarea în consecinţă a flag-urilor, etc. Aceasta se obţine apăsând în mod repetat butonul

(ALT+P)

sau se poate realiza *în mod continuu* (în general se foloseşte pentru urmărirea unor efecte), apăsând

(ALT+R sau F9)

Asamblarea se va face automat în momentul apăsării Run sau Step.

Instrucţiunea ce urmează a fi executată este semnalizată utilizatorului de către simulator prin marcarea ei, asemănător cum se realizează aceasta în EMU8086 și aşa cum se poate urmări în Figura 5.15.

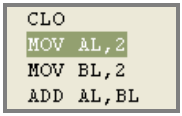


Figura 5.15. Marcarea instrucţiunii ce urmează a fi executată

5.3. Posibile aplicaţii cu EMU şi SMS

Dintre dispozitivele externe virtuale implementate de EMU8086 și SMS32v50, amintesc doar câteva: termostat, motor pas cu pas, sistem de 2 sau 4 semafoare controlate de la UCP, robot, ecran virtual, afişaj de tip 7 segmente cu 5 digiţi sau 2 digiţi, imprimantă, labirint, toate având interfeţe - aşa cum veţi putea urmări în *Cap. 11*.

Probleme mai complexe vor fi apoi abordate în *Capitolul 14*; totuși, pentru început, înainte de a trece la dezvoltarea de aplicaţii, fie acestea chiar și simple, sunt necesare anumite deprinderi în ceea ce privește lucrul cu simulatorul. Astfel, într-o primă fază, se vor parcurge multiple exerciţii pentru acomodarea cu scrierea numerelor într-o bază sau alta și operaţii de bază cu aceste numere folosind ambele simulatoare.

Capitolul 6. Conversii de numere şi operaţii de bază

În acest capitol, ne vom concentra pe lucrul efectiv cu simulatorul. Pentru început, reamintesc faptul că în simulator, la execuţia unui program, sunt disponibile 3 ferestre separate, așa cum se poate urmări în Figura 6.1 (ferestrele sunt numerotate).
Dacă se va dori execuţia rapidă a unor instrucţiuni, așa cum apare în secvenţa de 3 instrucţiuni din Figura 6.1, atunci se poate folosi un șablon minimal, în care e necesară doar directiva ***org 100h*** la început și apoi ***ret*** la sfârșit. Între cele două directive se vor specifica instrucţiunile dorite a fi executate; în fereastra 1 se va scrie codul sursă original. La apăsarea butonului ***Emulate***, vor apărea și ferestrele 2 și 3. În fereastra 2 se va putea urmări ***instrucţiunea ce urmează a fi executată*** colorată în albastru (în Figura 6.1 aceasta fiind *mov ax,2* care a fost transformată de asamblor în *mov ax,0002h*) și care se poate observa că a fost codificată pe 3 octeţi și depusă în memorie la adresele 07100h, 07101h și 07102h. În cea de-a treia fereastră se poate urmări colorată în galben ***instrucţiunea ce urmează să se execute***; imediat după apăsarea Single Step, aceasta se va executa.

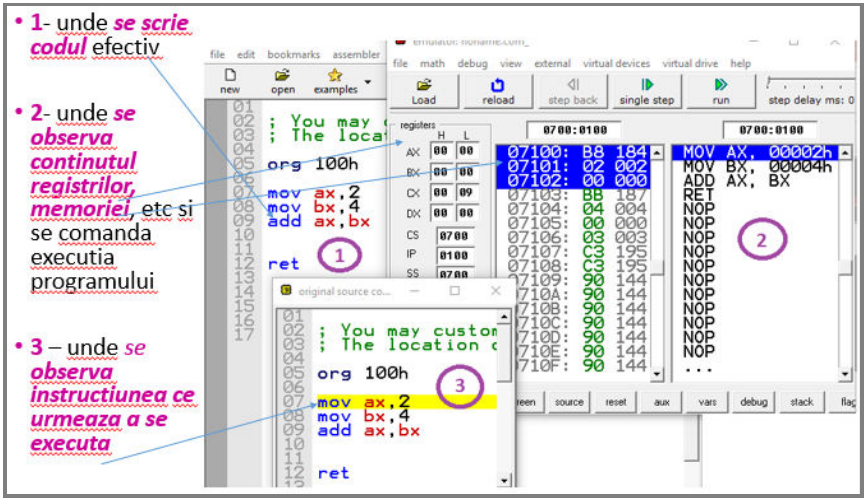


Figura 6.1. EMU8086 – execuţia primelor instrucţiuni

Pentru început, ne vom concentra pe acomodarea cu utilizarea valorilor binare, hexazecimale și zecimale în cadrul simulatorului și operaţii de bază cu numere în aceste baze de numeraţie. Este de dorit ca utilizatorul să realizeze multiple din aceste conversii într-un mod cât mai rapid și cât mai direct posibil. Ulterior, se va trece la scrierea și execuţia de secvenţe simple de instrucţiuni pentru acomodarea cu setul de instrucţiuni disponibil în cadrul simulatorului.

6.1. Conversii de numere

Conversia automată a numerelor dintr-o bază în alta poate fi realizată în cadrul simulatorului, așa cum am prezentat în capitolul anterior, folosind opţiunea ***base converter*** din meniul ***math*** sau apăsând butonul corespunzător (Figura 6.2). Aceste opţiuni sunt disponibile așa cum am arătat în Figura 6.1, în ferestra numerotată cu 1.

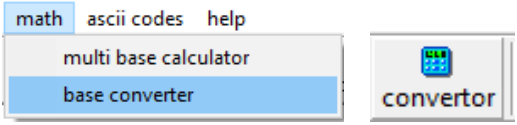


Figura 6.2. Emu8086 – opţiuni disponibile pentru accesarea convertorului

Exemplu: Se dorește conversia numărului 2 din baza 10 în baza 2. De fapt, așa cum se poate urmări în Figura 6.3, după utilizarea convertorului se obţine valoarea nu doar în binar cum s-a dorit iniţial, ci și în octal și hexazecimal. În plus, valoarea în zecimal este specificată atât ca număr cu semn cât și ca număr fără semn. Se poate observa această diferenţă în Figura 6.4 asupra numărului 37 și respectiv -37.

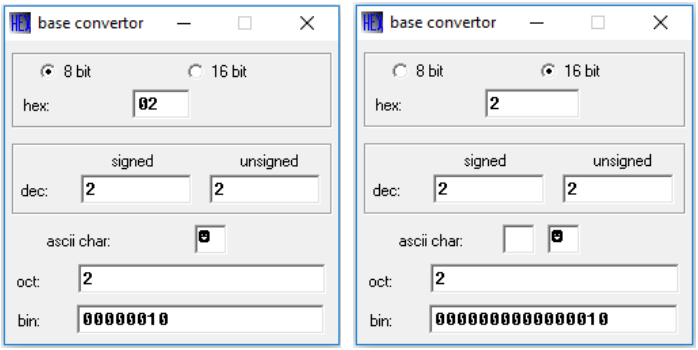


Figura 6.3. Emu8086 – conversia numărului 2 pe 8 sau 16 biţi

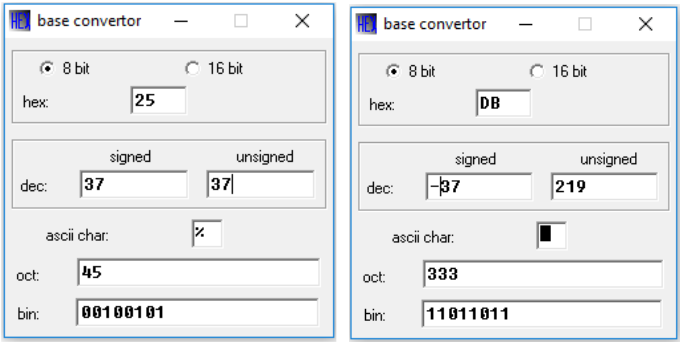


Figura 6.4. Emu8086 – conversia numărului 37 și -37 pe 8 biţi

Exemplu: Așa cum am văzut în *Capitolul 2*, reprezentarea numărului întreg **37** și respectiv **+ 37** sau **- 37** este:

37= $1*2^5 +0*2^4 +0*2^3 +1*2^2 +0*2^1 +1*2^0 =$ **1 0 0 1 0 1** $_b$ = **0010 0101**$_b$= 25h

+ 37= **0*2^6** $+1*2^5 +0*2^4 +0*2^3 +1*2^2 +0*2^1 +1*2^0 =$ **0 1 0 0 1 0 1** $_b$ = **0010 0101**$_b$= 25h

- 37 =**-1*2^6** $+0*2^5 +1*2^4 +1*2^3 +0*2^2 +1*2^1 +1*2^0 =$ **1 0 1 1 0 1 1** $_b$ = **1101 1011**$_b$ = 0DBh

Se observă astfel din Figura 6.4 că un număr scris în hexazecimal, de exemplu 0DBh poate fi numărul - 37 în zecimal, reprezentat ca număr cu semn, dar reprezentat ca număr fără semn ore o cu totul altă valoare: 219.

- 37 = 0DBh= **1101 1011**$_b$ =**-1*2^7 +1*2^6** $+0*2^5 +1*2^4 +1*2^3 +0*2^2 +1*2^1 +1*2^0 =$

$= +128 + 64 + 16 + 8 + 2 + 1 = 219$

În acest exemplu s-a ţinut cont de extensia cu semn sau fără semn corespunzătoare, exact cum e indicat a se proceda în orice caz s-ar folosi convertorul.

6.2. Calcule în diverse baze de numeraţie cu operatori

În cadrul simulatorului se pot realiza diverse operaţii, în vederea acomodării cu operaţiile disponibile în simulator, în oricare din bazele de numeraţie suportate de acesta. Figura 6.5 din dreapta arată care sunt aceste operaţii, iar în stânga se observă fereastra de efectuare a calculelor. Această fereastră este disponibilă folosind opţiunea ***multi base calculator*** din meniul ***math*** sau apăsând butonul corespunzător, așa cum se arată în Figura 6.6. Aceste opţiuni sunt disponibile după cum am arătat în Figura 6.1, în fereastra numerotată cu 1.

Reamintesc aici că diversele operaţii simple suportate (prezentate sugestiv în Figura 6.5 dreapta) au fost deja abordate în cadrul *Capitolului 2*, la secţiunea 2.2.9, cu exemple în binar. În schimb, aici exemplele vor considera valorile pe 8 sau 16 biţi (cât au dimensiunea regiștrii interni ai procesorului 8086).

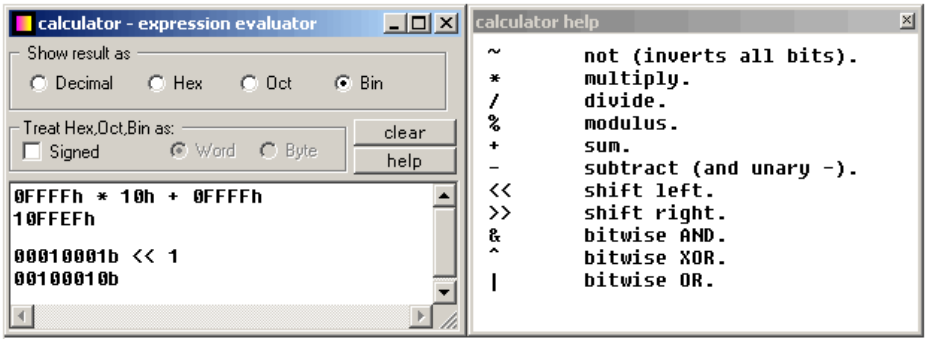


Figura 6.5. Fereastra Calculatorului (stânga) și operatorii disponibili (dreapta)

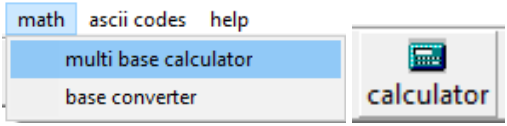


Figura 6.6. Emu8086 – opţiuni disponibile pentru accesarea calculatorului

Realizarea operaţiilor, așa cum arată Figura 6.5 din stânga, trebuie efectuată cu mare atenţie, în special la tratarea numerelor care nu prezintă semnul **+** sau **–** implicit, așa cum sunt valorile binare, hexazecimale sau octale (în special trebuie acordată atenţie sporită celor care au bitul MSb în 1). Dacă se va dori tratarea acestor numere ca fiind numere cu semn, va trebui bifată căsuţa corespunzătoare „Signed”.

Reamintesc aici modul de interpretare al ***numerelor cu semn*** în diverse baze:

dacă un număr scris ***în hexazecimal*** are cifra c.m.s. (MSnibble de rang maxim)

între 0 și 7, atunci el este ***pozitiv***,

între 8 și Fh, atunci numărul este ***negativ***.

dacă un număr scris ***în binar*** are cifra c.m.s. (MSbit, cel de rang maxim)

în 0, atunci el este ***pozitiv***,

în 1, atunci numărul este ***negativ***.

De asemenea, se poate specifica modul cum se dorește prezentarea rezultatului obţinut în urma calculului: sunt disponibile cele 4 sisteme de numeraţie zecimal, hexazecimal, octal și binar. Așa cum se arată în figură, e posibilă realizarea și de operaţii multiple: de exemplu, o înmulţire și apoi o adunare, iar abia la apăsarea tastei <Enter> se va realiza calculul.

6.2.1. Operaţii de adunare, scădere, înmulţire şi împărţire

În această secţiune, operaţiile menţionate vor fi realizate folosind operatorii care sunt disponibili în simulator, precum:

+ și **-** pentru a realiza adunarea, respectiv scăderea a două numere,

*****, **/** și **%** pentru a obţine produsul, și respectiv câtul și restul împărţirii a două numere care pot fi interpretate ca numere cu semn sau fără semn.

Operaţiile se vor realiza în cadrul simulatorului cu respectarea regulilor de prioritate cunoscute din clasele primare; în caz că se folosește paranteza, se consideră operaţia specificată în interiorul parantezei ca fiind mai prioritară. Exemple sunt prezentate în Figura 6.7. Odată ce s-a obţinut rezutatul operaţiei, în caz că se dorește conversia acestuia într-o altă bază, se poate scrie simplu „=”, se bifează noua bază dorită și se apasă <Enter>. Valorile dorite la operare pot fi specificate în zecimal, hexazecimal, etc.

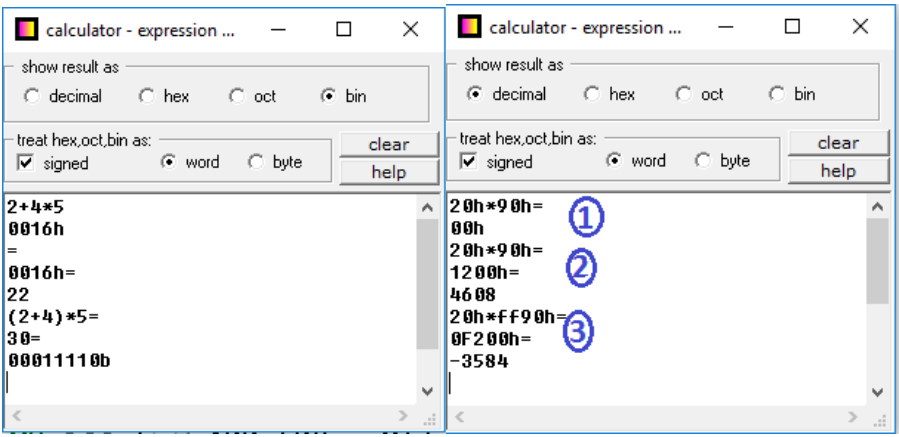


Figura 6.7. Exemple de calcule aritmetice

Exemple: Dacă se dorește înmulţirea a două valori scrise în hexazecimal, de exemplu, și se consideră aceste valori ca fiind numere cu semn, atunci trebuie considerată inclusiv extensia corespunzătoare a numerelor (pentru a garanta scrierea corectă a rezultatului). În fereastra din dreapta din Figura 6.7 s-a considerat înmulţirea a 2 numere 20h și 90h, numerele fiind considerate cu semn: 20h=+32, iar 90h=-112; rezultatul scontat este 32* (-112)= - 3584.

Cele 3 cazuri prezentate în figură sunt: (1) –operaţiile au fost realizate la nivel de octet (byte) și rezultatul a fost 00h, ceea ce este eronat; (2) - operaţiile au fost realizate la nivel de cuvânt (word), iar rezultatul a fost 1200h, ceea ce este eronat, 1200h=4608; (3) - dacă operaţiile au fost realizate la nivel de cuvânt (word), cu scrierea corespunzătoare cel puţin a operandului considerat a fi număr negativ – adică 90h trebuie scris pe 16 biţi, ca FF90h, rezultatul furnizat a fost F200h, ceea ce este corect, F200h= - 3584.

Dacă se dorește realizarea operaţiei de împărţire a două numere, se vor considera aceleași aspecte ca mai devreme, doar că în acest caz e posibilă obţinerea atât a câtului cât și a restului împărţirii, folosind operatorii corespunzători **/** și respectiv **%**.

Exemplu: Operaţia scrisă în calculator sub forma: F200h / 20h= FF90h, transformată în zecimal are semnificaţia: -3584 / 32 = -112, deci este corect.

Exemplu: Operaţia -3564 / 32 furnizează rezultatele : cât = - 111, iar rest = 12
care se vor obţine prin aplicarea operatorilor:

F214h / 20h= FF91h, adică -111
F214h % 20h= FFF4h adică -12

6.2.2. Operaţii logice pe şiruri de biţi

În această secţiune, operaţiile menţionate vor fi realizate folosind operatorii specifici operaţiilor logice suportate în simulator, dar la dimensiunea regiștrilor existenţi în structura procesorului emulat; acești operatori sunt disponibili sub următoarele forme: **~** pentru NOT, **&** pentru AND, **|** pentru OR și **^** pentru XOR**.** Pentru realizarea acestor operaţii, este indicată scrierea valorilor în binar pentru o urmărire mai ușoară a rezultatului obţinut; reamintesc aici că aceste operaţii se realizează bit cu bit și că nu există transport între poziţiile binare.

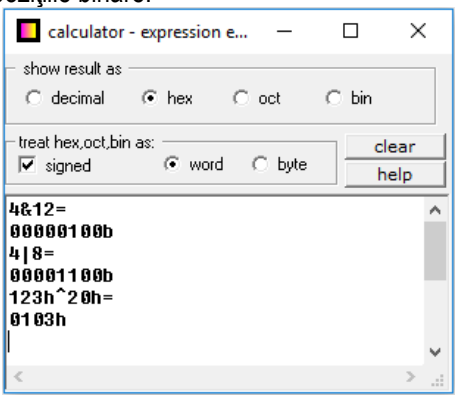


Figura 6.8. Exemple de operaţii logice pe biţi

După cum se poate observa din Figura 6.8, operaţiile logice suportă operanzii scriși în oricare din bazele disponibile în simulator, dar acesta consideră operarea tot la nivelul biţilor. Chiar și operarea cu 2 operanzi de dimensiune diferită scriși în hexazecimal (precum în ultimul exemplu 123h și 20h) este suportată. Vom vedea ca procesorul emulat este sensibil la astfel de situaţii și în unele cazuri furnizează mesaj de eroare.

În primul exemplu ilustrat în Figura 6.8, chiar dacă operanzii 4 și 12 au fost specificaţi în zecimal, simulatorul a convertit rezultatul obţinut la dimensiunea de octet. Similar, în ultimul exemplu din figură, 123h și 20h au furnizat un rezultat pe 16 biţi.

6.2.3. Operaţii de deplasare a şirurilor de biţi

În această secţiune, operaţiile de deplasare la nivel de șiruri de biţi vor fi realizate folosind operatorii specifici, precum: operatorul **<<** pentru operaţia de deplasare spre stânga și operatorul **>>** pentru operaţia de deplasare spre dreapta.

Trebuie acordată o deosebită atenţie la modul cum interpretează simulatorul valorile introduse în *calculator*, și anume: chiar dacă valorile cu care dorim să operăm sunt specificate într-o altă bază decât cea binară (în Figura 6.9 se observă folosirea valorii 123 în zecimal și a celei 7Bh în hexazecimal), valorile sunt întotdeauna considerate în binar. Astfel, 123<<1 este echivalent cu 01111011b<<1=11110110b și nu cu 1230 în

zecimal. Dacă în urma deplasării rezultă un număr mai mare de biţi decât suportă operandul sursă, atunci se folosește operarea la dimensiunea superioară; de exemplu s-a deplasat 123 cu 3 poziţii spre stânga, cu 123 putând fi scris pe 8 biţi, iar rezultatul deplasării s-a scris pe 16 biţi.

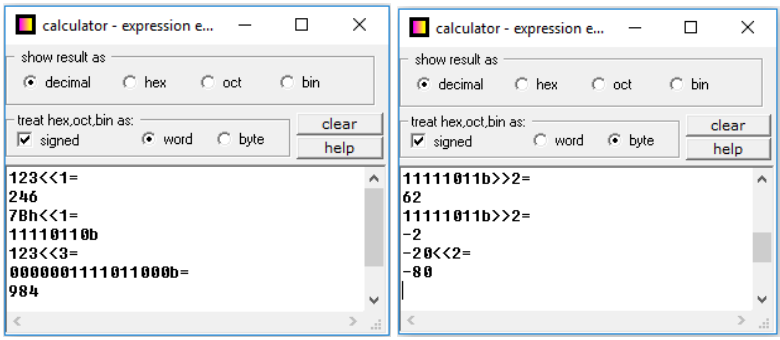


Figura 6.9. Exemple de operaţii de deplasare pe biţi

Un alt aspect deosebit de importat reiese din Figura 6.9 din dreapta, unde s-au introdus atât operanzi în zecimal cu semn, cât și operanzi în zecimal fără semn (ca valoare scrisă pe 8 biţi, numărul putea fi interpretat ca un număr negativ dacă ar fi fost interpretat cu semn).

Valoarea FBh=11111011b a fost deplasată înspre dreapta cu 2 poziţii, ceea ce înseamnă o împărţire cu 4. În primul caz, a fost bifată opţiunea operandului ca fiind de tip word și de aceea, rezultatul s-a considerat a fi numărul 00FBh=251 care împărţit cu 4 furnizează 62. În cel de-al doilea caz, s-a bifat căsuţa corespunzătoare operandului byte și atunci valoarea FBh=11111011b a fost interpretată ca număr cu semn (fiind bifată căsuţa corespunzătoare opţiunii *Signed*) și atunci s-a operat astfel: 11111011b>>2=>11111110b=-2, de unde se observă rotunjirea în jos.

Dacă operandul este specificat în mod explicit cu semnul -, deci în zecimal, de exemplu -20<<2, atunci indiferent pe câţi biţi este selectat operandul (8 sau 16), rezultatul va fi considerat număr negativ (dacă este bifată opţiunea *Signed*).

6.2.4. Operaţii de rotire a şirurilor de biţi

Deoarece aceste operaţii nu au vreun operator corespunzător, nu se vor putea testa aceste operaţii cu calculatorul disponibil în simulator. Scopul acestor exemple și exerciţii era de acomodare cu operaţiile specifice la nivelul operanzilor suportaţi pe procesorul 8086. Totuși, datorită faptului că aceste operaţii se aseamănă oarecum cu cele din secţiunea anterioară, ele nu vor fi dezavantajate la folosirea instrucţiunilor corespunzătoare pe procesor. În general, operatorii efectuează calculele (sau operaţiile respective) cu valori constante determinabile la momentul asamblării. În EMU, acești operatori pot fi folosiţi și la definirea datelor și ca înlocuitori ai instrucţiunilor.

6.3. Folosirea instrucţiunilor lui 8086 în simulator

După ce ne-am acomodat cu facilităţile oferite de *Convertorul* și *Calculatorul* disponibile în simulator, deci ne-am antrenat suficient, putem trece efectiv la scrierea de secvenţe de instrucţiuni în simulator; astfel, vom începe să folosim instrucţiunile specifice operaţiilor pe care le-am realizat până acum și nu vom mai folosi operatori, ci efectiv instrucţiunile implementate în EMU. Instrucţiunile vor folosi regiștrii procesorului 8086 pentru a stoca operanzii care până acum i-am văzut la nivel intuitiv (ca număr aleator de biţi, în general 4 biţi – în *capitolul 2*) și respectiv ca 8 sau 16 biţi când am folosit operatorii din *Calculator*, în *secţiunea 6.2*. În continuare, vom lucra cu instrucţiunile procesorului, așa cum au fost ele disponibile prin mnemonici la un procesor 8086 real. Pentru aceasta, este indicat ca înainte de utilizarea unei instrucţiuni să se consulte Help-ul disponibil în simulator (în fereastra corespunzătoare emulatorului, opţiunea *Help*), așa cum se arată în Figura 6.10:

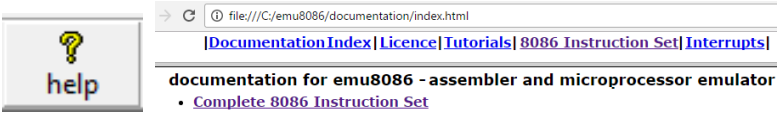


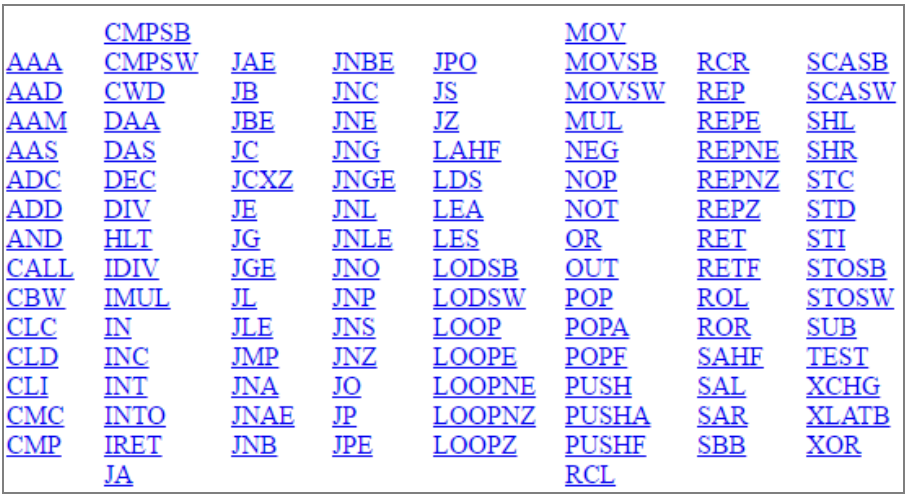
Figura 6.10. Consultarea sintaxei instrucţiunilor disponibile în EMU8086

EMU8086 pune la dispoziţie un Tutorial complex împreună cu mai multe exemple de cod, dar și explicaţii pentru fiecare din instrucţiunile care apar enumerate în Figura 6.11. Mai multe dintre acestea vor fi abordate în continuare, iar aspectele esenţiale care au fost urmărite vor fi subliniate pe parcurs. Exemplele prezentate în tutorial reprezintă o introducere pas cu pas în comenzile şi tehnicile de programare la nivel jos (low level). Fiecare program are unul sau mai multe exerciţii asociate, unele dintre ele simple, altele mai complexe. În continuare, se vor parcurge și programe cu instrucţiuni simple, urmărind în principal sintaxa şi efectul acestora. Pentru aceasta, este esenţială *rularea pas cu pas* a programelor.

Exerciţiu: Pe măsură ce folosiţi o instrucţiune ce apare listată în Figura 6.11, trasaţi o bifă în partea dreaptă a instrucţiunii respective.

Instrucţiunile pe care le vom aborda pentru început, care de altfel intuitiv le-am văzut deja în capitolele anterioare, sunt următoarele:

MOV, XCHG, CBW, CWD;
ADD, SUB, INC, DEC, NEG, ADC, SBB;
MUL, IMUL, DIV, IDIV;
NOT, AND, OR, XOR;
SHL, SHR, SAL, SAR, RCL, RCR, ROL, ROR;
AAA, AAD, AAM, AAS, DAA, DAS.



	CMPSB				MOV		
AAA	CMPSW	JAE	JNBE	JPO	MOVSB	RCR	SCASB
AAD	CWD	JB	JNC	JS	MOVSW	REP	SCASW
AAM	DAA	JBE	JNE	JZ	MUL	REPE	SHL
AAS	DAS	JC	JNG	LAHF	NEG	REPNE	SHR
ADC	DEC	JCXZ	JNGE	LDS	NOP	REPNZ	STC
ADD	DIV	JE	JNL	LEA	NOT	REPZ	STD
AND	HLT	JG	JNLE	LES	OR	RET	STI
CALL	IDIV	JGE	JNO	LODSB	OUT	RETF	STOSB
CBW	IMUL	JL	JNP	LODSW	POP	ROL	STOSW
CLC	IN	JLE	JNS	LOOP	POPA	ROR	SUB
CLD	INC	JMP	JNZ	LOOPE	POPF	SAHF	TEST
CLI	INT	JNA	JO	LOOPNE	PUSH	SAL	XCHG
CMC	INTO	JNAE	JP	LOOPNZ	PUSHA	SAR	XLATB
CMP	IRET	JNB	JPE	LOOPZ	PUSHF	SBB	XOR
	JA				RCL		

Figura 6.11. Instrucţiunile implementate în EMU8086

Deși în Figura 6.1. am arătat un exemplu de scriere a unei secvenţe de instrucţiuni în EMU8086, vom relua acest aspect aici și vom explica mai în detaliu cum trebuie procedat când vrem să scriem programe în EMU8086 și vrem să vizualizăm rezultatele obţinute. Instrucţiunile scrise în cadrul programelor vor trebui compilate, adică traduse în limbaj mașină, pe care procesorul să-l înţeleagă și să-l poată executa.

În Figura 6.12 se prezintă modul cum se execută o secvenţă de instrucţiuni în cadrul simulatorului; această secvenţă este explicată mai jos:

```
org 100h        ; directivă pentru compilator: îi spune cum să gestioneze codul sursă
.data           ; directivă care specifică adresa de început a zonei unde
                ; se depun datele definite în cadrul programului
                ; la începutul programului, se înlocuiește directiva .data cu o
                ; instrucţiune de salt la adresa unde încep instrucţiunile programului
                ; (așa se explică apariţia instrucţiunii jmp 0104h), deci salt peste
                ; locaţiile cu offset 102h și 103h – acolo este variabila de tip word
                ; definită mai jos, cu directiva x dw 3
x dw 3          ; se definește o variabilă cu numele x,
                ; ocupând un cuvânt în memorie și fiind iniţializată cu valoarea 3
doi equ 2       ; doi este o constantă care nu ocupă loc în memorie
.code           ; directivă care arată zona unde încep instrucţiunile programului
mov ax, offset x ; încarcă în registrul AX adresa de început a variabilei x, adică 0102h
```

Directiva ***ORG 100h*** este foarte importantă atunci când lucrăm cu variabile, deoarece aceasta va spune compilatorului că fișierul executabil trebuie încărcat la adresa cu offsetul 100h (după primii 256 octeţi din segmentul curent); astfel, compilatorul va trebui să calculeze adresa relativă a tuturor variabilelor atunci când va înlocui numele variabilelor cu adrese de offset specifice lor. Directivele nu au corespondent cod mașină, deci ele nu sunt transformate în cod, așa cum sunt transformate instrucţiunile.

Directiva *org 100h* este întâlnită doar la programele de tip COM; justificarea acestei adrese constă în faptul că sistemul de operare păstrează în acești primi 100h octeţi informaţii despre execuţia programului (informaţii de mediu, de context în care rulează programul, inclusiv parametrii din linia de comandă sunt păstraţi aici).

La ***programele executabile de tip COM***, adresele de început ale segmentelor sunt toate egale: CS=DS=SS=ES. La cele de ***tip EXE***, aceasta nu se mai respectă, ele fiind încărcate de la offsetul 0000h în cadrul segmentului; nici adresele de început ale segmentelor nu mai sunt identice, ca în cazul fișierelor de tip COM.

Pe Figura 6.12 au fost numerotate cele mai importante aspecte care intervin la execuţia unui program de tip COM în EMU, acestea fiind explicate în continuare:

1. Reamintesc din *capitolul 3* că ***adresa logică*** se scrie ca o pereche de valori pe 16 biţi fiecare, sub forma *segment: offset*, aceasta se va scrie în cadrul simulatorului EMU: 0700h:0100h, adică CS:IP;

2. ***Adresa fizică*** pe 20 biţi se calculează din adresa logică folosind următoarea relaţie: *segment *10h+offset*; deci 07000h+0100h=07100h;

3. În EMU, ***programele de tip COM*** au CS=DS=SS=ES=0700h.

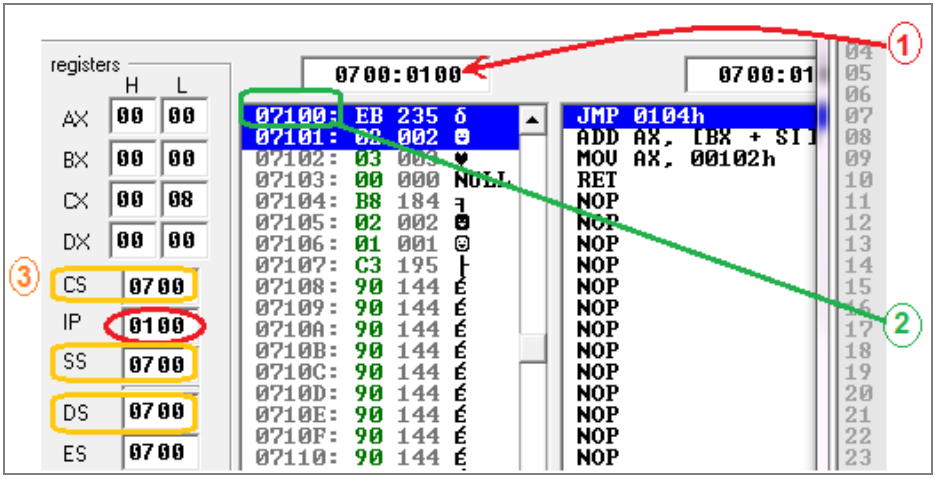


Figura 6.12. Explicarea noţiunilor esenţiale legate de execuţia programelor

6.4. Folosirea unui program de referinţă în EMU

(EMU) 2_sample.asm

Programul ***2_sample.asm*** se deschide din cadrul directorului implicit al simulatorului EMU8086 (*examples*) şi foloseşte instrucţiunile ***mov***, ***add***, ***sub***. Pentru a executa programul, apăsaţi Single Step până la execuţia instrucţiunii *sub bl,1* și nu 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** (este uşor de urmărit vizual orice modificare a valorilor din regiştri).

Figura 6.13. Încărcarea programului *2_sample.asm* în simulator

Observaţii:

Comentarii – orice text aflat după simbolul ";" nu este parte a programului, fiind ignorat de către asamblor. Comentariile se folosesc pentru a explica acţiunile folosite. Un bun programator foloseşte multe comentarii, însă acestea nu trebuie să repete pur şi simplu codul, ci să explice *de ce* se realizează lucrurile în modul respectiv.

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 (100h este 256 în zecimal), fiind specifică programelor de tip *.com*.

mov – este prescurtarea pentru Move. Instrucţiunea ***mov*** copiază datele dintr-o locaţie în alta, datele de la locaţia iniţială rămânând nemodificate. În acest exemplu, numerele sunt încărcate în regiştri pentru a putea realiza operaţia aritmetică.

Instrucţiunea *mov AL, 5*; va încărca în registrul AL valoarea zecimală 5.

Aritmetica – aici *add* se folosește pentru a aduna doi operanzi, rezultatul adunării depunându-se în operandul destinaţie (cel din stânga). Cu instrucţiunea *add BL,AL*; se va aduna conţinutul lui AL la cel din BL și rezultatul final se va depune în BL. Instrucţiunea *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. De exemplu, instrucţiunea *sub BL,1*; scade din BL valoarea 1 și rezultatul va fi în BL.

Regiştrii (precum AX, BX, CX, DX) sunt locaţii de stocare a datelor (așa cum am prezentat în *capitolul 3*), având dimensiunea maximă de 16 biţi în cazul simulatorului EMU8086; astfel, dispunem de regiştrii pe 16 biţi sau de părţile lor pe 8 biţi (care pot fi folosiţi, cu mici excepţii, în orice scop).

Numerele hexazecimale – în comanda *mov AL, 2;* valoarea 2 e un număr zecimal care în hexazecimal pe 8 biţi se scrie 02h. În programarea la nivel scăzut (LLL) se foloseşte sistemul hexazecimal, fiind mai convenabil (sau mai apropiat de binar).

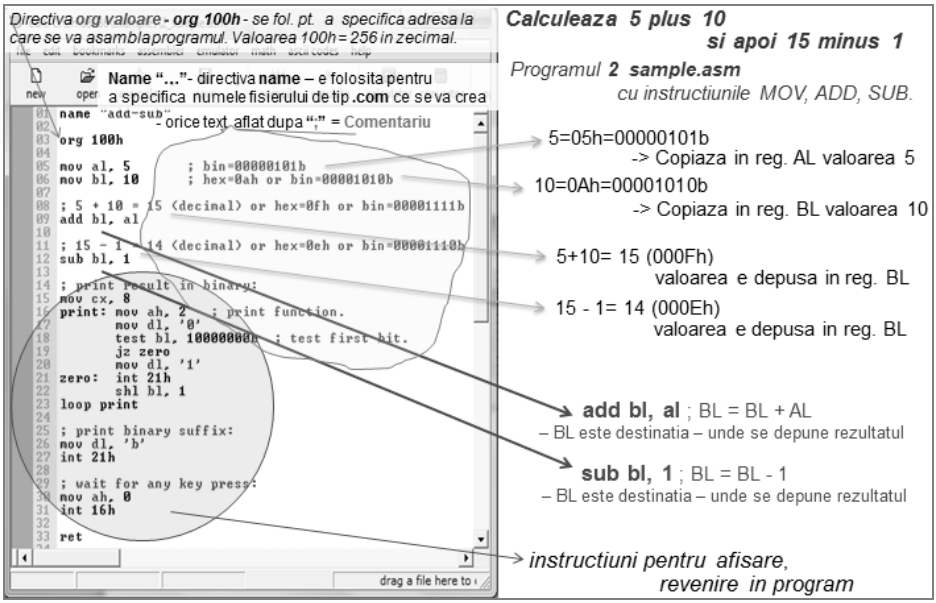


Figura 6.14. Explicarea instrucţiunilor programului *2_sample.asm* pe simulator

Pentru a studia aritmetica, e indicată consultarea valorilor regiştrilor aşa cum am sugerat anterior, în *capitolul 5*. Pentru a urmări rezultatele obţinute, puteţi consulta conţinutul ALU şi starea flagurilor, aşa cum prezintă Figura 6.15. De asemenea, se pot folosi ***calculatorul*** şi ***convertorul*** pentru verificarea corectitudinii calculelor. În Figura 6.16 am detaliat operaţiile necesare obţinerii programului executabil pe simulator.

Instrucţiunea ce urmează a fi executată (în Figura 6.15, aceasta este ***mov BL,10*** care se transformă în ***mov BL,0Ah***) poate fi regăsită în memorie, codificată pe 2 octeţi, începând cu adresa 07102h. Această adresă se scrie folosind 5 cifre hexazecimale, deci cu 20 biţi. Valoarea pe 20 biţi se obţine printr-o formulă de adunare decalată a valorii segmentului de cod (CS) şi a instrucţiunii curente (IP), așa cum am prezentat în *capitolul 3*. Reamintesc aici că din adresa scrisă sub formă logică **0700h:0102h** se obţine 0700**0**h+0102h, adică valoarea ***07102h*** care reprezintă adresa în forma fizică.

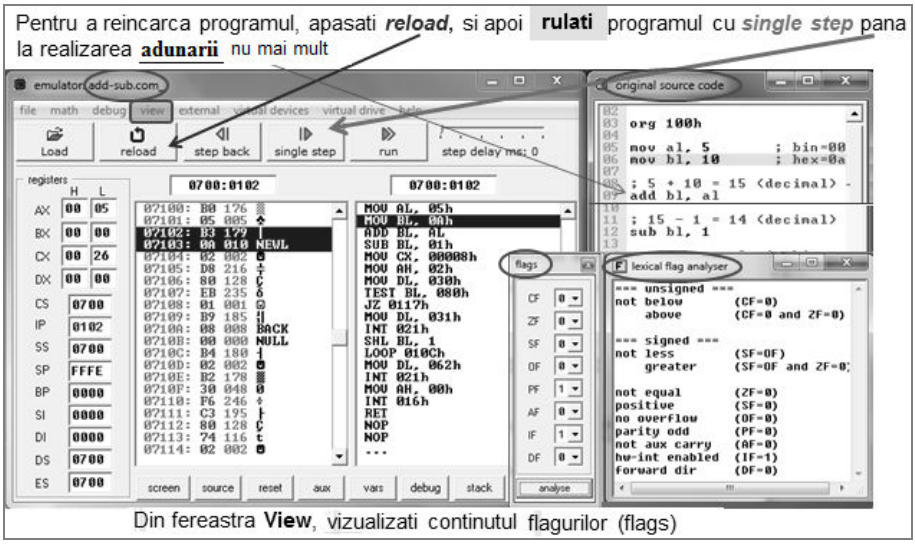


Figura 6.15. Vizualizarea rezultatelor în regiştri, ALU şi flaguri pe simulator

Exerciţiu:

1. Încărcaţi programul *2_sample.asm* în simulator, apăsaţi pe *emulate*, rulaţi-l o dată cu *run* și analizaţi rezultatul afișat pe ecran; daţi apoi *reload* și rulaţi-l cu *single step* analizând în paralel și explicațiile din Figura 6.14.

2. Creaţi apoi un nou program de tip .com plecând de la șablonul furnizat de simulator (Figura 6.13, cu opţiunea *new*) în care să scrieţi secvenţa din chenarul de mai jos. Modificaţi numele programului de tip .com și salvaţi-l local. Încercaţi apoi să-l deschideţi de pe harddisc, localizând directorul în care s-a salvat.

O secvenţă scurtă din programul 2_sample.asm este prezentată în continuare:

```
org 100h
;_____  CALCULEAZĂ 5 PLUS 10 ŞI 15 MINUS 1  _____________________
mov AL, 5        ; copiază în registrul AL valoarea  5=00000101b
mov BL, 10       ; copiază în registrul 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 *2_sample.asm* din simulator continuă cu instrucţiuni necesare afişării rezultatului în binar; acestea vor fi ignorate în capitolul de faţă, se vor studia ulterior.

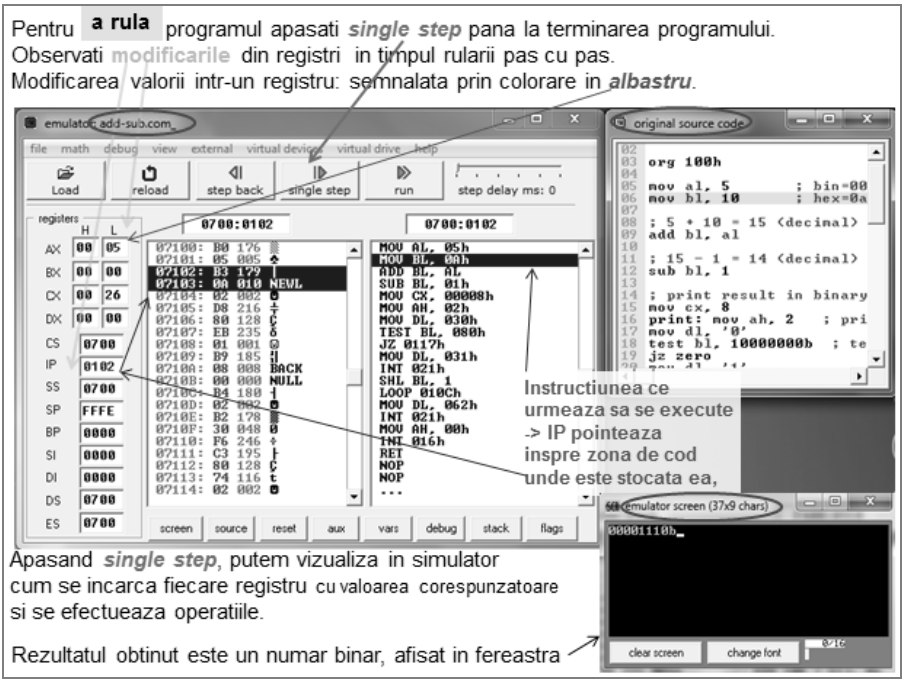


Figura 6.16. Explicarea operaţiilor necesare obţinerii programului executabil pe simulator

Majoritatea programelor prezentate în continuare includ o parte de exerciţii pentru învăţare, pentru a facilita înţelegerea modului de utilizare al instrucţiunilor procesorului. În general, va trebui să folosiţi regiştrii de uz general şi instrucţiunile sugerate. Consultaţi documentaţia din EMU pentru a înţelege modul de lucru al instrucţiunilor şi abia apoi includeţi-le în programe. Verificaţi și valorile flagurilor înainte și după execuţia instrucţiunilor aritmetice.

6.4.1. Acomodarea cu instrucţiunile de transfer

Analizaţi instrucţiunile ***mov***, ***xchg***, ***cbw*** și ***cwd*** din Help-ul simulatorului, Figura 6.10 și apoi încercaţi să scrieţi câte un program cu fiecare dintre exemplele indicate în Help.

Exerciţiu:

1. La consultarea Help-ului pentru instrucţiunea mov, veţi putea observa următoarea secvenţă de program care folosește memoria video pentru a scrie un caracter pe ecran. Observaţi comentariile și completaţi cu instrucţiunile corecte:

```
org 100h                ; directivă specifică fișierului de tip .com, DS=CS=SS=0700h în EMU
```

_____ ; depuneţi în AX valoarea B800h (ce corespunde memoriei VGA
_____ ; copiaţi valoarea din AX în registrul segment DS, astfel DS=0B800h.
_____ ; scrieţi în registrul CH codul ASCII al caracterului pe care doriţi să-l afişaţi
_____ ; depuneţi în CL atributul de culoare după regula bRGB(fond)|bRGB(text)
_____ ; scrieţi în BX poziţia în care doriţi să apară caracterul pe ecran = un număr par (la adresele impare se află atributele de culoare, iar la cele pare sunt codurile Ascii ale caracterului)
MOV [BX], CX ; în memorie la adresa fizică [0B800h:adresa pară] se va incepe copierea valorii din registrul CX, adică codul Ascii și atributul de culoare specificat de voi mai sus.
RET ; revine în sistemul de operare

2. Procedaţi similar ca în exerciţiul 1 și urmăriţi exemplele din Help pentru celelalte instrucţiuni sugerate în această secţiune. Rulaţi aceste exemple și încercaţi să le înţelegeţi instrucţiune cu instrucţiune. Consultaţi de asemenea și flagurile acolo unde se precizează că instrucţiunea respectivă afectează flagurile.

6.4.2. Acomodarea cu instrucţiunile de adunare şi scădere

Analizaţi instrucţiunile ***add***, ***sub***, ***inc***, ***dec***, ***neg***, ***adc*** și ***sbb*** din Help-ul simulatorului, Figura 6.10 și apoi încercaţi să scrieţi câte un program cu fiecare dintre exemplele sugerate în Help-ul simulatorului, așa cum s-a procedat în secţiunea 6.4.1.

6.4.3. Acomodarea cu instrucţiunile de înmulţire și împărţire

Analizaţi instrucţiunile ***mul***, ***imul***, ***div*** și ***idiv*** din Help-ul simulatorului, Figura 6.10 și apoi încercaţi să scrieţi câte un program cu fiecare dintre exemplele indicate, așa cum s-a procedat în secţiunea 6.4.1.

6.4.4. Acomodarea cu instrucţiunile logice pe biţi

Analizaţi instrucţiunile ***not, and, or*** și ***xor*** din Help-ul simulatorului, Figura 6.10 și apoi încercaţi să scrieţi câte un program cu fiecare dintre exemplele indicate, așa cum s-a procedat în secţiunea 6.4.1.

6.4.5. Acomodarea cu instrucţiunile de deplasare şi rotire

Analizaţi instrucţiunile ***shl, shr, sal, sar, rcl, rcr, rol*** și ***ror*** din Help-ul simulatorului, Figura 6.10 și apoi încercaţi să scrieţi câte un program cu fiecare dintre exemplele sugerate în Help-ul simulatorului, așa cum s-a procedat în secţiunea 6.4.1.

6.4.6. Acomodarea cu instrucţiunile de corecţie Ascii şi BCD

Analizaţi instrucţiunile ***aaa, aad, aam, aas, daa*** și ***das*** din Help-ul simulatorului, Figura 6.10 și apoi încercaţi să scrieţi câte un program cu fiecare dintre exemplele indicate, așa cum s-a procedat în secţiunea 6.4.1.

6.5. Folosirea unui program de referinţă în SMS

(SMS) aritmetica.asm

În cadrul unui nou fișier creat în simulatorul SMS, se va scrie următorul program care foloseşte instrucţiunile ***clo***, ***mov***, ***add*** şi ***end***. Pentru a rula programul apăsaţi *Step* până la terminarea sa pentru a observa modificările din regiştri. În momentul modificării valorii într-un registru, acest lucru este semnalat prin ***colorarea celulelor respective în galben***.

```
;____CALCULEAZĂ 2 PLUS 2____
CLO                     ; închide ferestrele care nu sunt necesare
        mov AL,2        ; AL = 2
        mov BL,2        ; BL = 2
        add AL,BL       ; AL = AL+BL
END                     ; programul se încheie
```

Comentarii – orice text aflat după simbolul ";" nu este parte a programului şi este ignorat de către asamblor. Comentariile se folosesc pentru a explica acţiunile folosite. Un bun programator foloseşte multe comentarii, însă acestea nu trebuie să repete pur şi simplu codul, ci să explice *de ce* se fac lucrurile în modul respectiv.

CLO – este o comandă aparţinând doar simulatorului; închide orice fereastră care nu este necesară în timpul rulării programului, evitând închiderea manuală a acestora.

MOV – Instrucţiunea *mov AL, 2*; va încărca în registrul AL valoarea hexazecimală 2.

Regiştrii – În cazul simulatorului SMS, dispunem de regiştrii AL, BL, CL, DL având dimensiunea de 8 biţi care pot fi folosiţi, cu mici excepţii, în orice scop.

END – este ultima comandă în orice program, specifică simulatorului SMS; orice text după acestă instrucţiune este ignorat.

6.6. Exerciţii propuse

Set 1 (secţiunea 6.1)

Folosind *convertorul* din Emu, consideraţi următoarele exerciţii din Capitolul 2 pentru (auto) verificare:
Set 4 – secţiunea 2.2.3: exerciţiile 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12;
Set 5 – secţiunea 2.2.4: exerciţiile 1 (doar ultima valoare), 2, 3, 4, 5, 6, 7, 8, 9, 10 (doar C2);

Set 2 (secţiunea 6.2)

Folosind *calculatorul* din Emu și implementând operaţiile prin operatorii disponibili, reluaţi exerciţiile din Capitolul 2 pentru (auto) verificare:
(Furnizaţi o captură cu toate operaţiile implementate)
Operaţii de adunare și scădere (folosind operatorii + și -)
Capitolul 2: Set 7 – secţiunea 2.2.6: exerciţiile 1, 2, 5 (adaptate la 8 biţi);
Capitolul 3: Set 1 – secţiunea 3.3: exerciţiile 2, 3;

Capitolul 2: Set 10 – secţiunea 2.2.9: exerciţiul 1, 2, 3, 4, 5;

Operaţii de înmulţire (folosind operatorul *)

Capitolul 2: Set 7 – secţiunea 2.2.6: exerciţiile 3, 4 (adaptate la 8 biţi);

Capitolul 2: Set 10 – secţiunea 2.2.9: exerciţiul 6 (adaptat la 8 biţi);

Operaţii de împărţire (folosind operatorii / și %): verificaţi în sens invers rezultatele obţinute anterior, la operaţia de înmulţire.

Capitolul 2: Set 7 – secţiunea 2.2.6: exerciţiile 3, 4 (adaptate la 8 biţi);

Capitolul 2: Set 10 – secţiunea 2.2.9: exerciţiul 6 (adaptat la 8 biţi);

Operaţia de inversare a semnului, adică negare (folosind operatorul -)

Capitolul 2: Set 10 – secţiunea 2.2.6: exerciţiul 19; scrieţi valorile și în zecimal;

Operaţiile logice not, and, or, xor (folosind operatorii ~, &, |, ^)

Capitolul 2: Set 10 – secţiunea 2.2.6: exerciţiile 7, 8, 9, 10;

Operaţiile de deplasare spre stânga și spre dreapta (folosind operatorii << și >>)

Capitolul 2: Set 10 – secţiunea 2.2.6: exerciţiile 11, 12, 13, 16, 17, 18;

Set 3 (secţiunea 6.3)

Să se scrie următorul exemplu de program:

```
Org 100h
.data
X dw 3
Y dw 4
.code
mov AX, X       ; _____
mov BX, Y       ; _____
add AX,BX       ; _____
```

Urmărind modelul din secţiunea 6.3, comentaţi acest program, precizaţi rolul lui și care este rezultatul final? (ce operaţie se implementează și unde se va găsi rezultatul final)

_____

Set 4 (secţiunea 6.4)

1. Plecând de la exemplul anterior care este pentru regiștri de 16 biţi, adaptaţi datele și regiștrii la valorile potrivite pentru a implementa operaţiile (pe care anterior le-aţi utilizat prin operatorii corespunzători) din cadrul exerciţiilor de la setul 1 și setul 2.

Pentru fiecare operaţie în parte, specificaţi ce instrucţiune aţi folosit. Verificaţi: rezultatul obţinut în registru trebuie să coincidă cu cel obţinut la folosirea operatorilor.

Operaţii de adunare și scădere (folosind instrucţiunile _____)

Capitolul 2: Set 7 – secţiunea 2.2.6: exerciţiile 1, 2, 5 (adaptate la 8 biţi);

Capitolul 3: Set 1 – secţiunea 3.3: exerciţiile 2, 3;

Capitolul 2: Set 10 – secţiunea 2.2.9: exerciţiul 1, 2, 3, 4, 5;

Operaţii de înmulţire (folosind instrucţiunile _____)

Capitolul 2: Set 7 – secţiunea 2.2.6: exerciţiile 3, 4 (adaptate la 8 biţi);

Capitolul 2: Set 10 – secţiunea 2.2.9: exerciţiul 6 (adaptat la 8 biţi);

Operaţii de împărţire (folosind instrucţiunile _____) verificaţi în sens invers rezultatele obţinute anterior, la operaţia de înmulţire.

Capitolul 2: Set 7 – secţiunea 2.2.6: exerciţiile 3, 4 (adaptate la 8 biţi);

Capitolul 2: Set 10 – secţiunea 2.2.9: exerciţiul 6 (adaptat la 8 biţi);

Operaţia de negare (folosind instrucţiunile _____)

Capitolul 2: Set 10 – secţiunea 2.2.6: exerciţiul 19;

Operaţiile not, and, or, xor (folosind instrucţiunile _____)

Capitolul 2: Set 10 – secţiunea 2.2.6: exerciţiile 7, 8, 9, 10;

Operaţiile de deplasare înspre stânga sau înspre dreapta (folosind instrucţiunile _____)

Capitolul 2: Set 10 – secţiunea 2.2.6: exerciţiile 11, 12, 13, 16, 17, 18;

Operaţiile de rotire înspre stânga sau înspre dreapta (folosind instrucţiunile _____)

Capitolul 2: Set 10 – secţiunea 2.2.6: exerciţiile 14, 15;

Operaţiile de corecţie zecimala sau BCD (folosind instrucţiunile _____)

Capitolul 2: Set 10 – secţiunea 2.2.6: exerciţiul 21;

2. Urmărind exemplele și exerciţiile propuse la secţiunile 6.4 și 6.5., încercaţi să executaţi exerciţiile din secţiunea 6.4 folosind simulatorul SMS (reluaţi toate exemplele din secţiunea anterioară, dar adaptate la arhitectura 8008).

Set 5 (secţiunea 6.4.1)

Acomodarea cu instrucţiunile de transfer

Analizaţi instrucţiunile *mov*, *xchg*, *cbw* și *cwd* din Help-ul simulatorului EMU, Figura 6.10 și apoi încercaţi să realizaţi exerciţiile propuse de mai jos:

1. Scrieţi o secvenţă de program care să conţină următoarele instrucţiuni:

mov AL,5 ; _____
mov BX, 1234h ; _____
mov BH, 98h ; _____
mov BX,5 ; _____

Rulaţi cu single step și analizaţi efectul fiecărei instrucţiuni în parte; comentaţi.

2. Scrieţi o secvenţă de program care să conţină următoarele instrucţiuni:

mov AL,5 ; _____
mov AH, 4h ; _____
xchg AL,AH ; _____

Rulaţi cu single step și analizaţi efectul fiecărei instrucţiuni în parte; comentaţi.

3. Scrieţi o secvenţă de program care să conţină următoarele instrucţiuni:

mov DX,1234h ; _____
mov AX,5678h ; _____
mov AL,5 ; _____
cbw ; _____
cwd ; _____

Rulaţi cu single step și analizaţi efectul fiecărei instrucţiuni în parte; comentaţi.
4. Scrieţi o secvenţă de program care să conţină următoarele instrucţiuni:
mov DX,1234h ; _____
mov AX,5678h ; _____
mov AL,9Ah ; _____
cbw ; _____
cwd ; _____
Rulaţi cu single step și analizaţi efectul fiecărei instrucţiuni în parte; comentaţi.
5. Scrieţi câte un program cu fiecare dintre exemplele sugerate în Help-ul simulatorului la instrucţiunile menţionate în această secţiune și analizaţi-le. Scrieţi mai jos aceste exemple:
mov: xchg: cbw: cwd:

Set 6 (secţiunea 6.4.2)

Acomodarea cu instrucţiunile de adunare și scădere

Analizaţi instrucţiunile *add*, *sub*, *inc*, *dec*, *neg*, *adc* și *sbb* din Help-ul simulatorului, Figura 6.10 și apoi încercaţi să realizaţi exerciţiile propuse de mai jos:
Exerciţii propuse:
1. Scrieţi o secvenţă de program care să conţină următoarele instrucţiuni:
mov AL,5 ; _____
mov BL, 89h ; _____
add AL,BL ; _____
Rulaţi cu single step și analizaţi efectul fiecărei instrucţiuni în parte; comentaţi.
Realizaţi calculul în binar, pe hârtie și apoi specificaţi valoarea flagurilor aritmetice după execuţia instrucţiunii *add* (așa cum am procedat în capitolul 3); scrieţi mai jos valorile obţinute: _____
2. Repetaţi exerciţiul 1, dar de data aceasta, după execuţia instrucţiunii *add*, consultaţi valorile flagurilor din simulator: fereastra emulatorului, meniul View, opţiunea flags: _____
3. Scrieţi o secvenţă de program care să conţină următoarele instrucţiuni:
mov AL,5 ; _____
mov BL, 0FDh ; _____
add AL,BL ; _____
Rulaţi cu single step și analizaţi efectul fiecărei instrucţiuni în parte; comentaţi.
Realizaţi calculul în binar, pe hârtie și apoi specificaţi valoarea flagurilor aritmetice după execuţia instrucţiunii add (așa cum am procedat în capitolul 3); scrieţi mai jos valorile obţinute: _____
4. Repetaţi exerciţiul 3, dar de data aceasta, după execuţia instrucţiunii *add*, consultaţi valorile flagurilor din simulator: fereastra emulatorului, meniul View, opţiunea flags: _____

5. Repetaţi exerciţiile 3 și 4 dar folosiţi regiștrii de 16 biţi în locul celor de 8 biţi. Păstraţi valorile propuse la exerciţiile 3 și 4. Ce observaţi? Comentaţi:

6. Repetaţi exerciţiile 1 și 2 dar înlocuind instrucţiunea *add* cu *adc*. Ce observaţi? Comentaţi:

7. Repetaţi exerciţiul 6 dar înainte de instrucţiunea *adc*, inseraţi instrucţiunea *clc* sau *stc*. Ce observaţi? Cum se modifică valoarea obţinută în cele 2 cazuri? Comentaţi:

8. Repetaţi exerciţiul 6 dar după instrucţiunea *adc*, scrieţi instrucţiunea *stc* și apoi instrucţiunea *add AL,0*. Ce observaţi? Comentaţi:

9. Scrieţi un program care să scadă două numere, folosind instrucţiunea *sub*;

10. Scrieţi un program care să calculeze opusul unui număr, cu instrucţiunea *neg*; de exemplu, plecând de la 2 obţineţi -2, dar și invers: plecând de la -2 să obţineţi 2.

11. Modificaţi programul astfel încât să se realizeze următoarele operaţii de adunare la dimensiunea cerută. Consultaţi valorile flagurilor, notaţi-le și explicaţi.
pe 8 biţi: i1) 67h + 12h; ii1) 67h + 59h; iii1) 97h + 0A9h; iv1) 84h + 0B5h;
pe 16 biţi: i2) 6767h+1212h; ii2) 6767h+5959h; iii2) 9797h+0A9A9h; iv1) 8484h+0B5B5h
9. Modificaţi exerciţiul 8 a.î. să realizeze operaţia de scădere în locul celei de adunare.

Set 7 (secţiunea 6.4.3)

Acomodarea cu instrucţiunile de înmulţire și împărţire

Studiaţi cu ajutorul Help-ului din EMU setul de instrucţiuni aritmetice pentru efectuarea operaţiilor de înmulţire și împărţire. Explicaţi-le.
1. Scrieţi un program care să înmulţească două numere, de câte 8 biţi fiecare, folosind instrucţiunea *mul*; unde va trebui să consultăm rezultatul? Verificaţi calculul cu Calculatorul. Modificaţi apoi astfel încât să se folosească instrucţiunea *imul*. Există vreo diferenţă?
2. Scrieţi un program care să împartă două numere, folosind instrucţiunea *div*; specificaţi cum trebuie să fie sau ce dimensiune trebuie să aibă operanzii și unde se vor vedea rezultatele obţinute.

3. Scrieţi un program care să realizeze împărţirea la 0. Vizualizaţi rezultatul şi amintiţi-vă să nu faceţi această împărţire altă dată. În cele alte condiţii poate apărea această eroare? Căutaţi explicaţia în diverse surse online.

Set 8 (secţiunea 6.4.4)

Acomodarea cu instrucţiunile logice pe biţi

Analizaţi instrucţiunile *not, and, or, xor* din Help-ul simulatorului, Figura 6.10 și apoi încercaţi să realizaţi exerciţiul propus de mai jos:

1. Presupunând valorile AX=3478h și BX=0FF0Fh, operaţiile logice pot fi realizate la mai multe dimensiuni ale operanzilor, așa cum reiese din secvenţa următoare. Scrieţi un program în EMU care să implementeze aceste operaţii și specificaţi rezultatele obţinute:

```
;  3478h  =  _____b -> AX
; 0FF0Fh  =  _____b -> BX
and AX, BX      ; AX= _____h =  _____b
or AX, BX       ; AX= _____h =  _____b
and AX,BX       ; AX= _____h =  _____b
or AX,BX        ; AX= _____h =  _____b
and AH,BL       ; AH= _____h =  _____b
or AL,69h       ; AL= _____h =  _____b
and AL,69h      ; AL= _____h =  _____b
```

Set 9 (secţiunea 6.4.5)

Acomodarea cu instrucţiunile de deplasare și rotire pe biţi

Analizaţi instrucţiunile *shl, shr, sal, sar* din Help-ul simulatorului, Figura 6.10 și apoi încercaţi să realizaţi exerciţiile propuse de mai jos:

i)	ii)	iii)	iv)
mov AL, 11100000b	mov AL, 10100100b	mov AL, 10111100b	mov AL, 10111010b
shl AL,1	shr AL,1	shl AL,1	shr AL,1
sar AL, 5	sal AL, 4	sar AL, 5	sal AL, 4
; AL= _____	; AL= _____	; AL= _____	; AL= _____
mov AX,1234h	mov AX,5678h	mov AX,4321h	mov AX,8765h
sal AL, 3	sar AL, 3	sal AL, 5	sar AL, 5
shr AH,6	shl AH,6	shr AH,4	shl AH,4
; AX= _____	; AX= _____	; AX= _____	; AX= _____

Analizaţi instrucţiunile *rcl, rcr, rol, ror* din Help-ul simulatorului, Figura 6.10 și apoi încercaţi să realizaţi exerciţiile propuse de mai jos:

i)	ii)	iii)	iv)
mov AL, 11100000b	mov AL, 10100100b	mov AL, 10111100b	mov AL, 10111010b
rol AL,1	ror AL,1	rol AL,1	ror AL,1
rcr AL, 5	rcl AL, 4	rcr AL, 5	rcl AL, 4
; AL= _____	; AL= _____	; AL= _____	; AL= _____

i)
```
mov AX,1234h
rol AL, 3
rcr AH,6
; AX= _____
```

ii)
```
mov AX,5678h
ror AL, 3
rcl AH,6
; AX= _____
```

iii)
```
mov AX,4321h
rol AL, 5
rcr AH,4
; AX= _____
```

iv)
```
mov AX,8765h
ror AL, 5
rcl AH,4
; AX= _____
```

Set 10 (secţiunea 6.4.6)

Acomodarea cu instrucţiunile de corecţie Ascii și BCD

Analizaţi instrucţiunile *aaa, aas, aad, aam, daa, das* din Help-ul simulatorului, Figura 6.10 și apoi încercaţi să realizaţi exerciţiile propuse de mai jos:

i)
```
mov AX, 13
aaa; AX= _____

mov AX, 23
aaa; AX= _____

mov AX,0204h
mov BX,0409h
add AX,BX
aaa; AX=_____

mov AX, 02FFh
aas; AX= _____

mov AL,8
mov Bl,7
mul BL; AX= _____
aam; AX= _____

mov AX, 0105h
aad; AX=_____

mov AL, 0FFh
das; AX=_____
```

ii)
```
mov AX, 15
aaa; AX= _____

mov AX, 25
aaa; AX= _____

mov AX,0305h
mov BX,0409h
add AX,BX
aaa; AX=_____

mov AX, 03FFh
aas; AX= _____

mov AL,7
mov Bl,9
mul BL; AX= _____
aam; AX= _____

mov AX, 0107h
aad; AX=_____

mov AL, 0Fh
daa; AX=_____
```

iii)
```
mov AX, 14
aaa; AX= _____

mov AX, 24
aaa; AX= _____

mov AX,0402h
mov BX,0409h
add AX,BX
aaa; AX=_____

mov AX, 04FFh
aas; AX= _____

mov AL,8
mov Bl,6
mul BL; AX= _____
aam; AX= _____

mov AX, 0108h
aad; AX=_____

mov AL, 0FFh
das; AX=_____
```

iv)
```
mov AX, 12
aaa; AX= _____

mov AX, 22
aaa; AX= _____

mov AX,0503h
mov BX,0308h
add AX,BX
aaa; AX=_____

mov AX, 05FFh
aas; AX= _____

mov AL,7
mov Bl,6
mul BL; AX= _____
aam; AX= _____

mov AX, 0109h
aad; AX=_____

mov AL, 0Fh
daa; AX=_____
```

Partea II

Capitolele 7, 8, 9, 10

Capitolul 7. Definirea şi reprezentarea datelor

În acest capitol ne referim în mod special la datele care se definesc imediat după directiva ***.data*** în cadrul unui program scris în simulator; locul din memorie unde se găsește spaţiu pentru a se stoca aceste date în general este segmentul de date și este pointat de registrul segment DS.

7.1. Lucrul cu datele din memorie

Tipurile de date întâlnite în programe pot fi: *variabile, constante* sau *etichete*. Instrucţiunile de salt sau apel precum JMP, CALL folosesc ca operanzi *etichetele* (desemnând adrese în zona program) în timp ce majoritatea instrucţiunilor (precum cele de transfer, aritmetice, logice - cum ar fi MOV, ADD, XOR) folosesc ca operanzi *variabile* şi *constante*.

Etichetele *identifică o zonă de cod* (din program, la nivel de procedură), iar variabilele *identifică date* (spaţiu de memorie rezervat). Constantele este impropriu să spunem că se găsesc în memorie, întrucât acestea nu rezervă vreo locaţie în memorie.

7.1.1. Definirea variabilelor

Datele sau variabilele se definesc în EMU prin intermediul directivelor:

- ***db*** (define byte) – pentru ***octet***;
- ***dw*** (define word) – pentru ***cuvânt***;
- ***dd*** (define double) – pentru ***dublucuvânt***.

Variabilele identifică datele, formând operanzi pentru instrucţiuni. Acestea sunt definite ca fiind *rezidente la o anumită adresă relativă (offset)* în cadrul unui anumit segment şi sunt caracterizate de tipul datelor. Pentru declararea variabilelor se utilizează directive care alocă şi iniţializează memoria în unităţi de octeţi, cuvinte, ***dublu-cuvinte***, astfel:

a. Directiva **DB (Define Byte)** declară octeţi sau şiruri de octeţi.

Exemplu: a db 1 ; s-a declarat o variabilă a de tip octet, de valoare 1

Adresa fizica	Valoarea in hexa	zecimal	Cod Ascii caracter
07102:	01	001	☺

Figura 7.1. Modul de prezentare al unei variabile de tip octet (a db 1) în memoria simulatorului

Astfel, folosind *a db 1* s-a alocat în memorie la adresa 07102h un spaţiu de un octet de valoare 01h sau 1 în zecimal, al cărui cod Ascii este ☺.

Exemplu: a db 12h,34h ; a este o variabilă şir de octeţi cu 2 valori: 12h şi 34h

07102:	12	018	↕
07103:	34	052	4

Figura 7.2. Modul de reprezentare al unei variabile de tip octet (a db 12h, 34h) în memorie

Exemplu: a db 12h,34h ; a este o variabilă şir de octeţi cu 2 valori: 12h şi 34h
b db 12,34 ; b este variabilă şir de octeţi cu 2 valori: 12=0Ch şi 34=22h

Figura 7.3. Modul de reprezentare a două variabile de tip șir de octeţi în memorie

Exemplu:
nume db 'Ana' ; *nume* este o variabilă şir de octeţi, cu valori codurile Ascii 'A', 'n' și 'a'

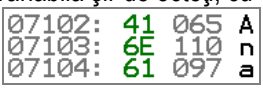


Figura 7.4. Modul de reprezentare al unei variabile de tip șir de octeţi coduri Ascii în memoria simulatorului

Exemplu: REZ DB ? , ? , ? , ? ; declară o variabilă REZ de tip şir de octeţi
; formată din 4 octeţi neiniţializaţi

În cazul variabilelor de dimensiune mare, se poate folosi operatorul DUP. Sintaxa acestuia este: ***număr DUP (valoare_sau_valori)***, unde *număr* trebuie să fie o constantă, iar *valoare_sau_valori* ceea ce se dorește a fi duplicat.

Exemplu: în Figura 7.5 s-au definit 3 variabile astfel:
REZ db 4 DUP (?) - echivalent cu exemplul anterior
a DB 3 DUP(2) - echivalent cu *a DB 2,2,2*
b DB 4 DUP(1, 2) - echivalent cu *b DB 1, 2, 1, 2, 1, 2, 1, 2*

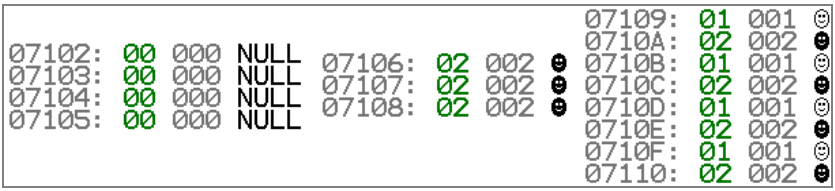


Figura 7.5. Modul de reprezentare a trei variabile de tip șir de octeţi definite cu operatorul DUP în memoria simulatorului

b. Directiva **DW (Define Word)** declară cuvinte sau şiruri de cuvinte.

Exemplu:
SIR DW 5 DUP (2000h) ;declară o variabilă SIR de tip şir de cuvinte (Fig.7.6)
; formată din 5 cuvinte identice iniţializate cu 2000h

Exemplu: în Figura 7.7 s-au definit 2 variabile de tip șir de cuvinte astfel:
SIR1 DW 1234h,1234 ; declară o variabilă SIR1 de tip şir de cuvinte
; formată din 2 cuvinte iniţializate cu 1234h și 1234
SIR2 DW 1200h,12h ; declară o variabilă SIR2 de tip şir de cuvinte
;formată din 2 cuvinte iniţializate cu 1200h și 0012h

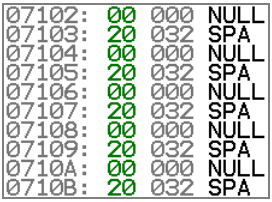


Figura 7.6. Modul de reprezentare al unei variabile de tip cuvânt în memorie

După cum se poate observa în Figura 7.7, pentru variabila sir2 s-a găsit spaţiu în memorie după variabila sir1, deci începând de la adresa cu offsetul 106h.

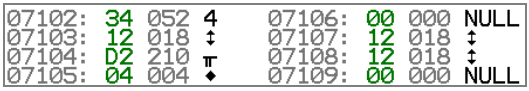


Figura 7.7. Modul de reprezentare a două variabile de tip șir de cuvinte în memorie

c. Directiva **DD** **(Define Double-word)** declară dublu-cuvinte sau şiruri de dublu-cuvinte; deși procesorul 8086 nu știe să prelucreze astfel de entităţi (nu există regiștri de 32 biţi la 8086), EMU permite folosirea acestei directive (doar pentru definirea datelor în memorie, nu și pentru manevrare în cadrul regiștrilor).

Exemplu: a dd 2 ; a este o variabilă de tip dublucuvânt cu valoarea 2=00000002h
b dd 3 ; b este o variabilă de tip dublucuvânt cu valoarea 3=00000003h

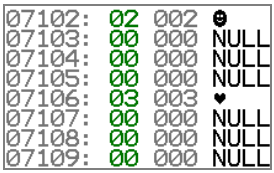


Figura 7.8. Modul de reprezentare a două variabile de tip dublucuvânt în memorie

d. Directiva **DT** **(Define Tera-byte)** declară tera-octeţi, adică un număr real în precizie extinsă (80 biţi); deși în EMU nu e implementată această directivă, vom vedea că se poate folosi la vizualizarea valorilor din memorie.

7.1.2. Adresarea variabilelor din memorie

Adresarea unei variabile care a fost definită în memorie se poate realiza folosind paranteze drepte (există uneori și posibilitatea de a nu le folosi). În următorul exemplu se prezintă 3 cazuri în care se va accesa al 2-lea element al șirului folosind diverse moduri de plasare a parantezei drepte; elementele unui șir sunt indexate de la +0, apoi urmează +1, apoi +2, ș.a.m.d.

Exemplu: Fie următoarea secvenţă scrisă în EMU:

```
org 100h
.data
a db 1,2,3,4        ; se definește variabila a de tip byte, cu 4 octeţi
                    ; iniţializaţi cu valorile 1, 2, 3 și 4
.code
mov al, a+1         ; copiază în AL elementul din șir, indexat cu o poziţie în plus, AL=2
mov bl, [a+1]       ; același element din șir se copiază în BL,deci BL=2
mov cl, a [1]       ; același element din șir se copiază în CL, deci CL=2
 ret
```

În exemplul de mai sus s-au putut urmări multiple moduri de adresare ale aceluiași element din șir cu ajutorul parantezelor drepte sau fără acestea.

Mai mult, trebuie subliniat că adresarea elementelor din memorie nu ţine cont de dimensiunea variabilei din momentul definirii acesteia în memorie. Acest lucru poate fi oarecum dăunător, întrucât se pot suprascrie zone din memorie dacă nu se ţine cont de ordinea și dimensiunea datelor stocate în memorie.

Compilatorul transformă caracterele (atunci când detectează apostroafele din definirea variabilei) automat în octeţi; de exemplu, în memorie se va aloca spaţiu pentru șirul de caractere ‚Anaaremere'.

Exemplu: Fie următoarea secvenţă scrisă în EMU:

```
org 100h
.data
a db 'Ana'
b db 'are'
c db 'm', 65h, 're'
.code
mov al, a+1    ; AL = 6Eh=‚n' rezultat din a[1], adică al 2-lea element al lui a
mov bl, b+2    ;  BL = 65h=‚e'  rezultat din b[2], adică al 3-lea element al lui b
mov cl, a+6    ; CL =6Dh=‚m' rezultat din a[6], adică al 7-lea element al lui a, deși a nu are
decât 3 elemente; astfel, în loc de formularea „al lui a" mai corect se spune „raportat la
adresa de început a lui a".
```

Figura 7.9. arată zona de memorie din cadrul simulatorului, după ce au fost definite cele 3 variabile de tip *șir* din exemplu. Așa cum se poate observa și din figură, șirurile nu sunt altceva decât variabile înlânţuite.

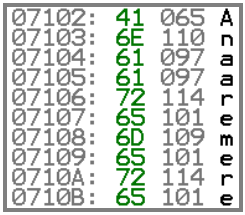


Figura 7.9. Modul de reprezentare a trei variabile de tip șir de octeţi coduri Ascii în memoria simulatorului

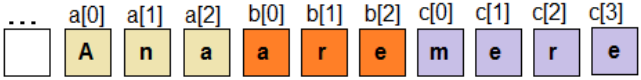


Figura 7.10. Reprezentarea variabilelor din memoria simulatorului pe orizontală

Pentru adresarea elementelor unui șir, în general, așa cum am văzut în *Capitolul 3,* se poate folosi adresarea indirectă prin regiștri; chiar mai mult, se pot folosi doi regiștri la indexare, de exemplu BX și SI, ca în exemplul următor:

Exemplu: pentru același exemplu de mai devreme, se poate folosi următorul mod de adresare indexată dublă:

```
mov bx,1
mov si, 2
mov al, a[bx][si]   ; astfel, se va adresa elementul a[3]
```

Variabilele pot fi vizualizate din simulator, din fereastra emulatorului, meniul *View*, apoi selectând opţiunea *variables*.

Exemplu: Pentru Figura 7.11, variabilele au fost definite ca în exemplul următor:

```
.data
 x db 2
 y dw 5
 z db 2,3,4,5
```

Pentru fiecare variabilă în parte, așa cum se poate observa în Figura 7.11 poate fi selectată dimensiunea ei; de exemplu, pentru variabila ***z*** a fost selectat numărul de 5 elemente din fereastra ilustrată în figură, deși variabila are doar 4 elemente. Aceasta nu înseamnă altceva decât că octetul al 5-lea, de valoare 0BAh este octetul următor variabilei ***z*** (în mod normal nu ar trebui să-l accesăm). În plus, cu opţiunea „show as”, așa cum se arată în figură în partea dreaptă, se poate selecta modul cum să fie prezentată variabila respectivă, fiind disponibilă inclusiv opţiunea de coduri Ascii. Pentru vizualizarea corectă a șirurilor definite în memorie, e necesar ca pentru fiecare variabilă de tip șir să se selecteze în mod individual dimensiunea, din opţiunea „size”, după cum se poate observa în Figura 7.11.

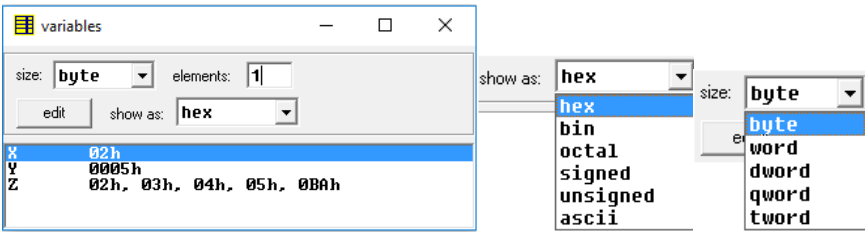


Figura 7.11. Vizualizarea variabilelor în EMU și opţiuni disponibile

7.1.3. Definirea constantelor

Constantele pot fi absolute (numerice) sau simbolice – nume generice asociate unor valori numerice.

Constantele numerice pot fi reprezentate în funcţie de specificatorul utilizat.

Exemple:

Binare	11101010B, 1010b	; secvenţe de 0,1 urmate de b sau B
Zecimale	34[D], 12345	; secvenţe de 0÷9, cu / fără d sau D
Hexazecimale	24h, 0B3H	; secvenţe de 0÷9 şi litere A÷F,cu h sau H
Octale	23o, 23q	; secvenţe de 0÷7 urmate de o,O, q sau Q
Caracter	'alba‘, “alba”, “12345”	; șir de caractere şi va fi de tipul Ascii

Constantele simbolice se pot defini folosind sintaxa: ***nume*** **EQU** ***expresie***

Exemplu:
```
Space EQU 20h
mov AX, Space          ; AX=20h
```

Constantele se comportă asemănător variabilelor, doar că nu ocupă loc în memorie (deoarece nu au dimensiune precum byte, word, etc) și există doar cât timp programul este compilat (adică asamblat). Pentru a defini constante se folosește directiva EQU, având sintaxa: *nume EQU expresie*.

Exemplu:

Space EQU 20h ; s-a definit o constantă cu numele Space de valoare 20h
În programe, aceasta va putea fi folosită în oricare din cele două forme:
mov AX, Space
mov BL, Space
întrucât *Space* nu are o dimensiune asociată.

7.1.4. Etichete

Etichete: Etichetele sunt nume simbolice de adrese, ce se folosesc pentru a identifica instrucţiunile prin raportare la un anumit offset în cadrul unui segment.

Exemplu:
```
        mov ax,0
        mov bx,2
Aduna:  add ax,bx
        inc bx
        call AfisSumaDinAX
        jmp Aduna      ; în acest caz, Aduna este o etichetă de tip NEAR
```

7.1.5. Directive

- directiva LABEL, cu forma generală: ***nume*** **LABEL** ***tip***

va fi folosită pentru referirea la instrucţiuni de tip JMP/CALL

Exemplu:
```
sirb LABEL BYTE      ; s-a definit eticheta cu numele sirb ce permite accesul
                     ; la variabila sirw octet cu octet
sirw DW 1234h        ; o instrucţiune de forma mov AL, sirb ; va face AL=34h
```

- directiva PROC – apare la definirea procedurilor, numele procedurii fiind interpretat ca o etichetă care are tipul derivat din tipul procedurii (NEAR sau FAR). Forma generală a unei proceduri este:

Exemplu:
```
Nume_proc PROC tip
        …            ; corpul procedurii
Nume_proc ENDP
```

7.1.6. Operatori

Se utilizează operatorul **?** pentru rezervarea zonei de memorie pentru variabila neiniţializată; pentru valori multiple ale variabilelor se utilizează operatorul ***DUP*** care precizează numărul replicilor (***DUP***licate).

Operatori: Limbajul de asamblare pune la dispoziţia utilizatorilor mai mulţi operatori pentru modificarea, combinarea, compararea sau analiza operanzilor.

I. Operatori modificatori de valori:

1. Operatori care au corespondent și instrucţiuni

a. ***Operatorii aritmetici*** (pentru operaţii elementare **+, -, *, /, %**) apar în expresii aritmetice evaluate la momentul asamblării şi rezultă constante numerice.

b. ***Operatorii logici*** (pentru operaţiile logice **NOT, AND, OR, XOR)** apar în expresii logice evaluate la momentul asamblării, din care rezultă constante numerice.

c. ***Operatorii pe șiruri de biţi*** (pentru operaţiile de deplasare la stânga sau la dreapta**)** apar în expresii evaluate la momentul asamblării şi rezultă constante numerice.

Toţi aceşti operatori au fost prezentaţi în cadrul *Capitolului 2* și reluaţi apoi prin instrucţiunile corespunzătoare în cadrul *Capitolului 6*.

Exemplu:
mov SIR **+** 2, DX ; asamblorul efectuează operaţia de adunare între
; adresa SIR (componenta de tip offset) şi 2

Exemplu:
6 **shl** 3 ; 0000 0110b SHL 0011 0000b – realizează operaţia coresp.

- Acesta este un exemplu folosind operatorul de deplasare spre stânga (SHL), dar putem folosi și operatorul asociat în EMU (<<)

2. ***Operatorii de atribuire*** definesc constante simbolice după sintaxa:
nume_var **EQU** *expresie*

Exemplu:
ZERO EQU 0 ; asamblorul înlocuieşte constanta ZERO cu valoarea numerică 0
N EQU 17 ; N=17
Directiva **EQU** permite declararea constantelor. Constantele astfel declarate sunt înlocuite cu numere propriu-zise în momentul asamblării. Pentru acestea nu se alocă spaţiu de memorie.

3. ***Operatorii relaţionali*** (**EQ, NE, LT, LE, GT, GE**) au semnificaţii evidente şi returnează valori logice codificate ca 0 sau secvenţe de 1 pe un anumit număr de biţi. Aceștia nu sunt suportaţi în EMU.

Exemplu:
4 EQ 3 ; este fals, întrucât 4 este diferit de 3 și rezultă astfel 00000000b
4 GT 3 ; este adevărat, întrucât 4 este mai mare decât 3 și rezultă 11111111b=-1

II. Operatori generatori de valori:
- se aplică unor variabile sau etichete returnând valori ale acestora, de exemplu size, type, length – nu sunt suportaţi de EMU;

- operatorul **SEG** aplicat variabilelor sau etichetelor returnează adresa de bază a segmentului în care se află variabila / eticheta;
- operatorul **OFFSET** returnează valoarea adresei relative ("offset"-ului) faţă de adresa de început a segmentului în care este declarată variabila/ eticheta căreia i se aplică;
- dimensiunea unei variabile se mai poate obţine folosind operatorul **$**;
- operatorul **THIS** creează un operand care are o adresă de segment şi un offset identic cu contorul de locaţii curent, având sintaxa
THIS ***tip,*** unde ***tip*** poate fi: BYTE, WORD, DWORD, QWORD sau TBYTE pentru variabile sau NEAR/FAR pentru etichete.

Exemplu: VAR EQU **THIS** WORD ; declară variabila VAR de tip cuvânt

```
Exemplu: SIRB EQU   THIS BYTE      ; declară variabila SIRB de tip octet
         SIRW   DW    100   DUP(?) ; SIRW este un şir de cuvinte de
                                    ; lungime 100 cuv., neiniţializat
                                    ; variabila SIRB va avea acelaşi
                                    ; segment şi offset ca SIRW
Exemplu:    REZ DW 100 DUP (1)      ; dacă se declară variabila REZ de 100
                                    ; cuvinte, atunci LENGTH REZ va fi 100,
                                    ; TYPE REZ va avea valoarea 2,
                                    ; iar SIZE REZ va fi 200.
Exemplu:    VAR DW 1,2,10h          ; variabila VAR este definită de tip cuvânt,
                                    ; cu valorile 1, 2 şi 10h
sub BP, TYPE VAR      ; asamblorul înlocuieşte TYPE VAR cu 2.
mov CX, LENGTH VAR    ; asamblorul înlocuieşte LENGTH VAR cu 3
mov BP, SIZE VAR      ; asamblorul înlocuieşte SIZE VAR cu 6
mov SP, OFFSET VAR    ; asamblorul înloc. OFFSET VAR cu offset-ul etichetei VAR
mov AX, SEG VAR       ; asamblorul înlocuieşte SEG VAR cu segmentul etich. VAR
```

Exemplu:

```
mesaj db 'Hello world'     ; se defineşte variabila mesaj ca un şir de octeţi
lungime_mesaj db $-mesaj; lungimea şirului mesaj ($ este contorul locaţiei curente)
```

- operatorii **HIGH** și **LOW** returnează octetul cel mai semnificativ (MSB), respectiv cel mai puţin semnificativ (LSB) al unei expresii.

```
Exemplu:    VAR EQU 3456h       ; VAR=3456h
            mov AH, HIGH VAR    ; AH=34h
            mov AL, LOW VAR     ; AL=56h
```

nu sunt suportaţi în EMU8086.

- operatorii **NEAR/FAR/SHORT** poziţionează tipul unei etichete în modul specificat de operator.

```
Exemplu:    JMP   NEAR  ET      ; salt intrasegment, salt direct
Exemplu:    JMP   SHORT ET      ; salt scurt sub 128 octeţi, IP este relativ
```

III. Operatori modificatori de atribute:

a. operatorul **PTR** modifică tipul unei variabile/ etichete: BYTE, WORD, DWORD, etc. Utilizarea lui este obligatorie în cazul unor referinţe anonime la memorie. Sintaxa de utilizare este: ***tip* PTR *expresie***

Exemplu

```
DATA DB 03                      ; variabila DATA declarată ca octet
mov AX, WORD PTR DATA           ; asamblorul utilizează elementele variabilei DATA
                                ; sub formă de cuvinte
mov WORD PTR DATA + 2 , DX ; se obţine compatibilitate cu tipul cuvânt al registrelor
```

Exemplu: inc BYTE PTR[BX] ; octetul adresat de BX este incrementat
Exemplu: jmp FAR PTR VAR ; salt de tip FAR

- Operatorul ***: (două puncte)*** modifică segmentul implicit al unei variabile/ etichete (utilizat pentru precizarea adresei fizice) într-un segment explicit.

Exemplu: add AX, ES: SIR [BP] ; registrul segment ES este specificat explicit
; în instrucţiune (altfel, era SS)

7.2. Lucrul cu memoria de tip stivă

Această secţiune se referă tot la o zonă de date (tot în memorie se găseste și stiva), dar aceasta e gestionată cu ajutorul registrului segment SS și se folosește într-un mod puţin diferit faţă de cea dinainte. Aici datele nu apar în memorie prin scrierea unor directive (precum db) ci prin folosirea unor instrucţiuni specifice stivei (precum ***push*** și ***pop***, sau cele înrudite cu acestea, ***pusha***, ***popa***, ***pushf***, ***popf***).

Stiva se definește ca ***o zonă din memorie*** de tip listă LIFO (Last In First Out), este de fapt "ca un sac". Această zonă "de tip sac" se află în segmentul de stivă, adresa ei de început fiind pointată de registrul segment SS și offsetul BP – base pointer sau pointer la baza stivei. Pe măsură ce se introduc date în stivă cu una din instrucţiunile suportate de CPU (specifice lucrului cu stiva), se decrementează un pointer numit SP – stack pointer – adică pointerul la ultimul element introdus în stivă; ca și cum am ţine mâna în dreptul ultimului articol introdus în sac; registrul BP este pointer la baza stivei, sau ca și cum am arăta înspre elementul de la baza sacului, cele pe care „stau" (sau mai bine spus *se stivuiesc*) toate celelalte.

La instrucţiunile care folosesc stiva, ***CPU*** *gestionează automat adresele* din memorie, controlând registrul SP (Stack Pointer) care indică vârful stivei (nu și BP).

- De ce e nevoie de stivă?

La ***apelul procedurilor***, *adresa de revenire* în programul principal se depune pe stivă.
La ***apelul subrutinelor***, *parametrii și variabilele locale* se depun pe stivă.

- Cum lucrează stiva?

Iniţial, stiva nu conţine date, este goală, dar pe măsura introducerii elementelor în stivă, se spune că dimensiunea stivei creşte (sacul se umple), întinzându-se ***spre adrese mai mici*** (stiva creşte „**de jos în sus**"), așa cum sugerează Figura 7.12. Astfel, se observă că stiva se umple înspre adrese ce descresc,

⇒ pointerul **SS:SP** conţine întotdeauna
adresa ultimului operand introdus în stivă

(peste care ar urma să se depună un nou element) = "**elementul din vârful stivei**".

Instrucţiunea **PUSH** **(Push Value onto Stack)** ***depune în stivă*** un operand pe 2 octeţi (**PUSH operand**), ceea ce înseamnă că SP urcă în stivă cu 2 (deci descreşte)

Instrucţiunea **POP** **(Pop Value off Stack)** ***extrage din stivă*** ultimul operand pe 2 octeţi introdus (**POP operand**) => SP coboară în stivă cu 2 octeţi.

Sursa şi destinaţia au fost iniţial, la **8086,** operanzi pe **16 biţi** și nu pot avea o dimensiune mai mică (de exemplu, nu pot fi operanzi pe 8 biţi); de la 386 în sus, ei au putut fi și operanzi pe 32 biţi.

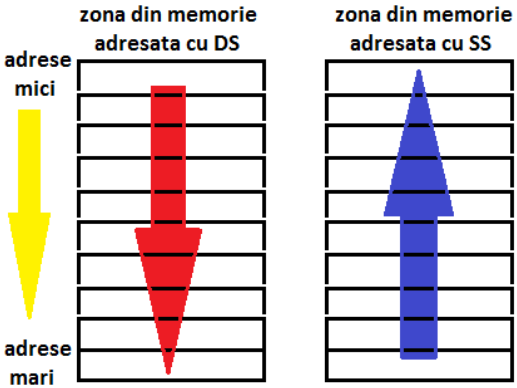


Figura 7.12. Depunerea elementelor **în memorie** (stânga) și **în stivă** (dreapta)

Problemă Rezolvată:

Fie execuţia următoarei secvenţe de instrucţiuni, incluzând comentariile:

```
mov AX, 0123h        ; AX = 0123h
mov BX, 4567h        ; BX = 4567h
mov CX, 89ABh        ; CX = 89ABh
mov DX,0CDEFh        ; DX = CDEFh
push AX              ; SP = SP-2, s-a depus pe stivă 0123h
push BX              ; SP = SP-2, s-a depus pe stivă 4567h
push CX              ; SP = SP-2, s-a depus pe stivă 89ABh
push DX              ; SP = SP-2, s-a depus pe stivă CDEFh
```

Rulaţi în EMU8086 acest exemplu și explicaţi după fiecare instrucţiune starea stivei, vizualizând-o.

Rezolvare:

Iniţial, valorile SS și SP sunt 0700h și FFFEh așa cum reiese și din Figura 7.13 (partea stângă); apoi, la fiecare element depus în stivă, SP descreşte cu 2.
După depunerea celor 4 elemente de tip word (0123h, 4567h, 89ABh, CDEFh) în stivă, SP a devenit pe rând: FFFCh, apoi FFFAh, apoi FFF8h, apoi FFF6h.

Figura 7.13 arată zona de stivă înainte de a depune vreun element în ea (stiva e goală inițial): pentru ca simulatorul să arate zona din memorie corespunzătoare stivei, utilizatorul trebuie să specifice în căsuța corespunzătoare adresei logice (zona de sus în figură) valoarea 0700:FFFEh, adică SS:SP în mod general, sau o valoare puțin mai mică – în figură s-a folosit FFF6h: vom vizualiza cu 8 octeți, sau cu 4 elemente de tip word mai mult din zona unde se află implementată stiva.

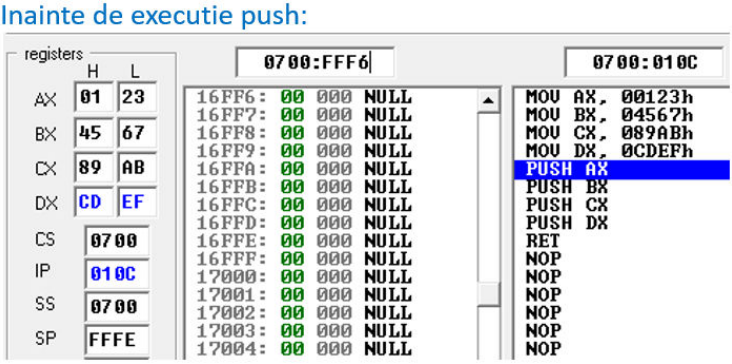


Figura 7.13. Vizualizarea zonei de stivă înainte de execuția vreunei instrucțiuni ***push***

După execuția celor 4 instrucțiuni *push*, dacă se modifică în zona corespunzătoare adresei logice din nou cu valoarea 0700:FFF6h, atunci în stivă vor putea fi vizualizate acum cele 4 elemente, stivuite în ordinea 0123h primul, peste el 4567h, apoi 89ABh și ultimul, în vârf, 0CDEFh (după cum arată Figura 7.14)

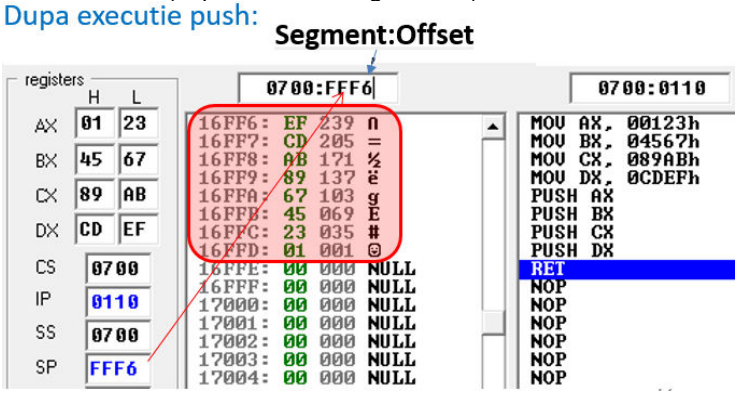


Figura 7.14. Vizualizarea zonei de stivă după execuția celor 4 instrucțiuni ***push***

Figura 7.14 arată rezultatul final, după depunerea celor 4 valori de tip word pe stivă (cele 4 cuvinte s-au stivuit mergând în sus, dinspre bază înspre vârf), iar octeții se vor aranja în memorie după cum precizează convenția Little END-ian.

Există 3 ZONE CURENTE, folosite uzual:

Zona de cod	determinată de	**CS:IP**
Zona de stivă	determinată de	**SS:SP**
Zona de date	determinată de	**DS:offset**

Simulatorul nu ne spune exact unde trebuie să ne uităm (în zona de cod, de date sau de stivă) pentru a interpreta CORECT informaţia. La fiecare execuţie a unei instrucţiuni, în simulator se modifică valorile din zona corespunzătoare adresei logice pentru a actualiza IP (deci simulatorul arată utilizatorului zona de cod).

Dacă din întâmplare, segmentul zonei de cod coincide cu cel de date și stivă (cum este cazul la ***programele de tip .com***, în cazul nostru SS=CS=DS=ES=0700h), atunci în același segment vom vedea și instrucţiunile programului nostru (așa cum sunt ele stocate în memorie la adresa CS:IP, unde IP=0109h de exemplu în Figura 7.15 jos) dar și zona de stivă (care se află la offsetul SP=FFFEh de ex. în Figura 7.15 sus).

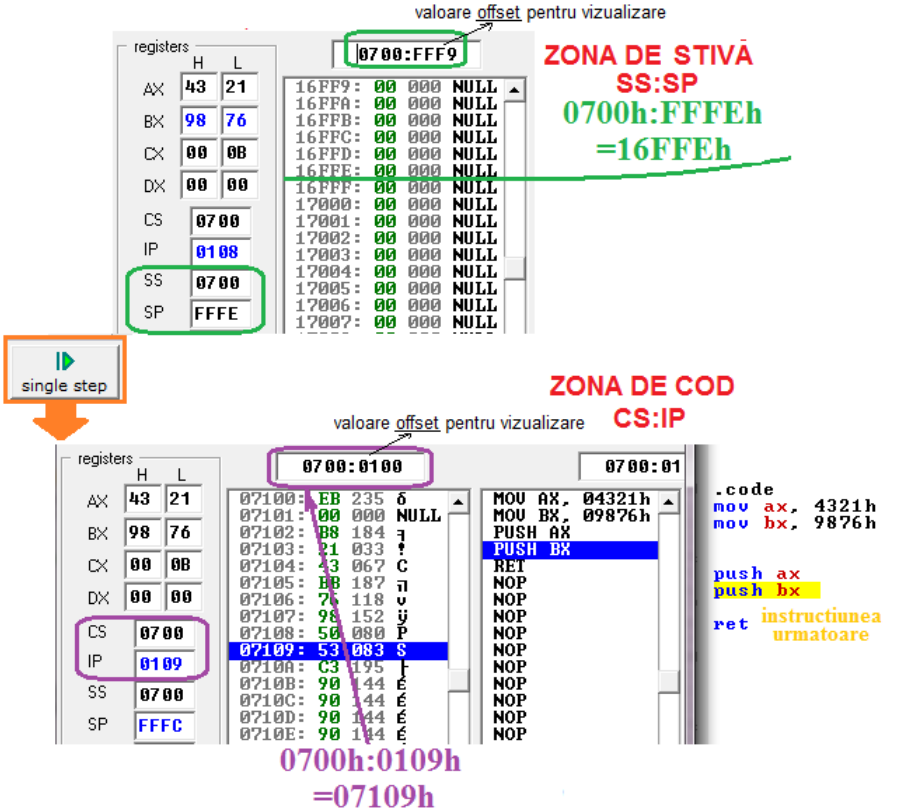


Figura 7.15. Lucrul cu stiva și saltul automat în zona de cod

Figura 7.15 jos, obţinută din Figura 7.15 sus, după ce s-a accesat o dată butonul Single Step, arată că s-a executat instrucţiunea următoare, adică *push AX* care a determinat urcarea pointerului SP cu 2 locaţii, deci SP s-a modificat din FFFEh în FFFCh în vederea depunerii pe stivă a conţinutului registrului AX. Deși această instrucţiune (curentă) executată a afectat stiva, totuși în simulator pentru vizualizare, ***se sare automat la zona de cod***. Astfel, în zona adresei logice vizualizate în partea de sus a simulatorului se va vedea acum următoarea adresă CS:IP și anume 0700h:010Ah=0710Ah și nu zona de stivă cum ne-am fi așteptat 0700h:FFFCh=16FFCh.

Figura 7.16 arată în partea de sus conţinutul stivei după introducerea primului element pe stivă (cu instrucţiunea *push AX*) și după ce utilizatorul a tastat valoarea FFFCh în zona adresei logice. Astfel, se poate formula următoarea regulă: dacă se dorește vizualizarea zonei de stivă, noi trebuie să introducem o valoare apropiată de cea SP (puţin mai mică de preferat) în caseta destinată adresei logice. De exemplu, pentru FFFCh, așa cum reiese și din Figura 7.16, se va putea vizualiza primul element depus pe stivă (și anume conţinutul registrului AX) în formatul Little Endian (capătul/ octetul mai puţin semnificativ la adresa mai mică).

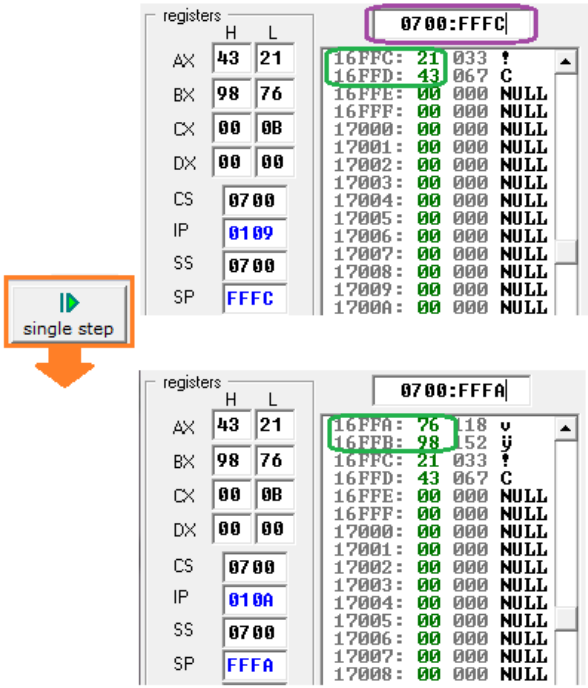


Figura 7.16. Lucrul cu stiva – continuarea Figurii 7.15

În partea de jos a Figurii 7.16 se poate vizualiza structura zonei de stivă după depunerea și celui de-al doilea element pe stivă, și anume conţinutul registrului BX (depus pe stivă folosind instrucţiunea *push BX*).

7.3. Definirea corectă a datelor în simulator

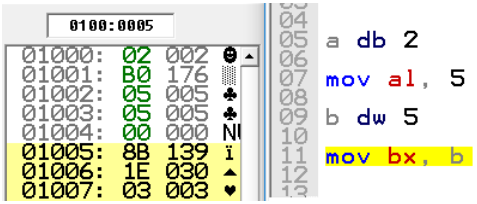


Figura 7.17. Secvenţă de instrucţiuni alternând date cu cod

În Figura 7.17 în cadrul unui program minimal în simulator (template de tip ***.com***) s-au scris alternativ atât definiri de date cât și instrucţiuni, **fără a separa *datele* de *cod;*** din contră, acestea alternează (așa cum se sugerează schematic în Figura 7.19 în partea de sus). În simulator, directiva *a db 2* se codifică în memorie la adresa 01000h, instrucţiunea *mov al,5* la adresele 01001h și 01002h; directiva *b dw 5* la adresa 01003h, iar instrucţiunea *mov bx, b* la adresele 01005h … 01007h așa cum se prezintă în Figura 7.17. O astfel de secvenţă, deși suportată de simulator, nu se va executa corect de către procesor (e posibil să apară erori în plus, pe lângă faptul că nu se depun în regiștri valorile specificate).

Pentru a folosi corect simulatorul, se recomandă scrierea directivelor și a instrucţiunilor în cadrul unor modele sau șabloane de programe. Cel mai simplu astfel de șablon este cel de tip .com, care are specifică directiva ***org 100h*** (apare la începutul secvenţei de program) și instrucţiunea ***ret*** la sfârșit (pentru revenirea în sistemul de operare). În plus, definirea datelor este precedată de directiva ***.data***, iar specificarea instrucţiunilor este precedată de directiva ***.code***.

La programele de tip ***.com***, deși simulatorul poate funcţiona și cu câteva secvenţe precum cele din Figura 7.18 din stânga, Corect se va considera:

Datele* precedate de directiva *.data
Instrucţiunile* precedate de directiva *.code

```
Sir db 1,2,3,4

mov bx, offset sir
mov al, sir[1]

ret
```

```
org 100h

.data
Sir db 1,2,3,4

.code
mov bx, offset sir
mov al, sir[1]

ret
```

Figura 7.18. Cod scris incorect (stânga) vs cod scris corect (dreapta)

Dacă ne-am imagina un scenariu în care *programul de executat ar fi mare* (dar totuși să încapă în cadrul unui segment așa cum sugerează Figura 7.19), deci: ar conţine ***multe instrucţiuni și date*** (pleacă de la offsetul 0h și tot coboară în zona de memorie) și în plus *și stiva ar fi folosită intens*, realizându-se ***depuneri multiple de date în stivă*** (deci pleacă de la offsetul 0FFFFh și tot urcă în zona de memorie) atunci ar putea exista posibilitatea ca *stiva să suprascrie zona de cod și date* și ar putea să apară probleme. De aceea, la programele mari nu se folosește program de tip ***.com***. În plus, pe stivă nu se realizează doar depuneri de date, ci și extrageri de date, acestea din urmă determinând coborârea pointerului SP spre bază. Scenariul imaginat s-a folosit mai mult pentru a înţelege cât mai bine modul cum lucrează simulatorul cu datele, codul și stiva.

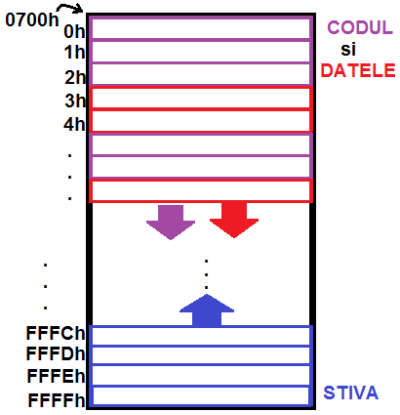


Figura 7.19. Vizualizarea zonei de segment pentru un program de tip .com

7.4. Instrucţiuni ce lucrează cu memoria la nivel de şir

În această secţiune mă voi referi în mod special la operaţiile primitive care consideră mai multe locaţii succesive din memorie ca formând o entitate; aceste entităţi pot fi octet (8 biţi), cuvânt (16 biţi) sau dublucuvânt (32 biţi) și vor folosi sufixele B (byte), W (word), D (doubleword). Operaţiile care se pot realiza asupra acestor entităţi sunt:

- de copiere sau transfer (MOVS),
- de încărcare a elementelor în accumulator (LODS),
- de descărcare a acestora din accumulator în memorie (STOS),
- de scanare (SCAS) și
- de comparare (CMPS).

De exemplu, instrucţiunile de copiere sau transfer ***(MOVe String)*** **MOVSB/W/D** transferă un octet, un cuvânt sau un dublucuvant din şirul sursă (adresat de DS:(E)SI) în şirul destinaţie (adresat de ES:(E)DI), transfer urmat de actualizarea adreselor (adică incrementarea sau decrementarea lui (E)SI sau (E)DI cu 1, cu 2 sau cu 4).

În mod implicit, la instrucţiunile pe şiruri, se consideră următoarele:
- perechea de regiştrii ***DS: (E)SI*** este folosită pentru adresarea ***sursei***;
- perechea de regiştrii ***ES: (E)DI*** este folosită pentru adresarea ***destinaţiei***;

Şirul poate fi stocat în memorie:
- în ***sens crescător*** (de la adrese mai mici -> adrese mai mari) dacă **DF=0**,
- în ***sens descrescător*** (de la adrese mai mari -> adrese mai mici) dacă **DF=1**;

La parcurgerea şirului în sens crescător
=> (E)SI şi (E)DI vor fi actualizaţi prin ***incrementare***,

La parcurgerea şirului în sens descrescător
=> (E)SI şi (E)DI vor fi actualizaţi prin ***decrementare***;

Numărul de octeţi cu care se incrementează/ decrementează regiştrii (E)SI şi (E)DI ***este dat de dimensiunea elementelor şirului:***
d=1 pt **octeţi**, **d=2** pt **cuvinte** şi **d=4** pt **dublucuvinte**, așa cum arată Figura 7.20.

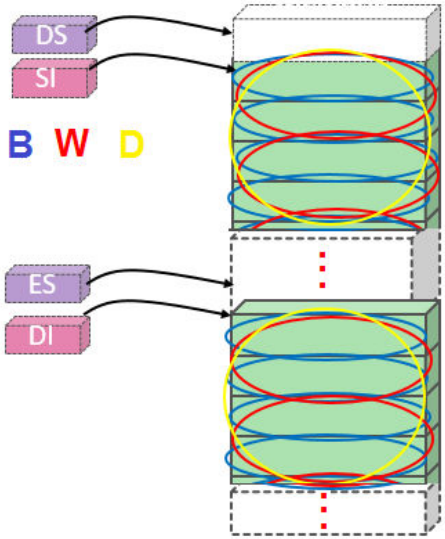


Figura 7.20. Posibilă ilustrare pentru modul de parcurgere al memoriei de către instrucţiunile de manipulare a şirurilor la nivel de **octet**, **cuvânt**, **dublucuvânt**

Aceste instrucţiuni pentru şiruri realizează inclusiv actualizarea adreselor, însă acestea ***nu ştiu să repete operaţia de un anumit număr de ori*** - câte elemente are şirul de prelucrat. Astfel, în general, instrucţiunile pentru lucrul cu şiruri sunt ***combinate cu algoritmi de repetare***, gen:
- prefixe de repetare (precum rep, repe, repz, repne, repnz) sau
- bucle cu salt condiţionat (jump if condiţie, de exemplu jz, jnz, etc).

Problemă Rezolvată:

Să se scrie o secvenţă de program care să folosească instrucţiuni pt manipularea şirurilor în vederea copierii unui şir de octeţi într-o altă zonă din memorie.

Rezolvare:

Secvenţa următoare defineşte 2 şiruri: unul sursă şi unul destinaţie, poziţionează regiştrii index pe zonele de început ale şirurilor (vor pointa spre primul element din fiecare şir), iar apoi execută instrucţiunea MOVSB:

```
SIRs DB 1,2,3,4,5      ; se def. şirul destinaţie cu 5 elem. octet, iniţilizat cu 1,2,3,4,5
SIRd DB 5 DUP(0)       ; se def şirul sursă cu 5 elemente pe octet, iniţilizat cu 0
...
lea SI, SIRs           ; SI=adresa de început a SIRs
lea DI, SIRd           ; DI=adresa de început a SIRd
cld                    ; DF=0, şirurile vor fi parcurse în sens crescător

; dacă se va scrie doar:
movsb                  ; atunci se mută doar primul element din şir şi se poziţionează
                       ; pe cel de-al doillea element din cadrul fiecărui şir, dar nu mai mult

; în schimb, dacă se va scrie instrucţiunea într-o buclă, ca mai jos:
eti:
   movsb
   dec cx
  jnz eti
```

; atunci prin parcurgerea buclei în mod repetat (de 5 ori), se vor copia toate elementele şirului sursă în şirul destinaţie, aşa cum arată Figura 7.21

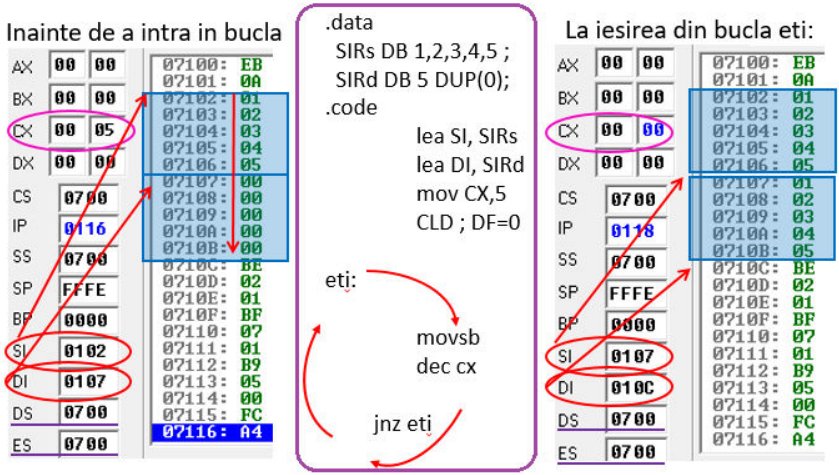


Figura 7.21. Ilustrarea exemplului de copiere a unui şir sursă într-un şir destinaţie

7.5. Organizarea informaţiei de cod în memorie

Această secţiune se referă tot la o zonă de date (tot în memorie se găsește și codul), dar aceasta e gestionată cu ajutorul registrului segment CS, în mod automat. În cadrul zonei de memorie al cărei început este precizată de registrul CS, se realizează salturi în funcţie de valoarea registrului IP. Acesta se actualizează în mod automat de către procesor, utilizatorul nu are acces la IP (acces restricţionat tocmai pentru a nu altera funcţionarea procesorului). Modul cum IP sare de la o valoare la alta depinde de specificul fiecărei instrucţiuni, de modul cum s-a decis codificarea fiecărei instrucţiuni pe procesor (această decizie se ia în etapa de proiectare a procesorului). Unele instrucţiuni, în general cele mai intens utilizate, sunt codificate pe un număr mai mic de octeţi (și deci IP va realiza un salt cu o valoare mai mică), iar cele mai rar utilizate (se mai spune și că nu au fost optimizate pe procesorul respectiv) sunt codificate pe un număr mai mare de octeţi. Mai multe detalii despre codificarea instrucţiunilor vom vedea în *Capitolul 9*.

Problema cea mai mare care poate apărea în cadrul simulatorului și a procesorului în general este întâlnirea unei „instrucţiuni” (așa o vede procesorul, ca pe o instrucţiune de executat deși în zona de memorie poate fi și zonă de date sau stivă, nu neapărat doar cod) care are o codificare invalidă sau eronată, așa cum arată Figura 7.22.

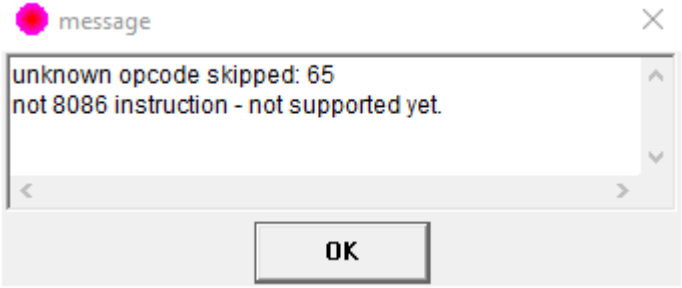


Figura 7.22. Posibilă eroare la execuţia unui cod necunoscut

În programe, apare deseori necesitatea execuţiei unei secvenţe de instrucţiuni, în mod repetat. Secvenţa care se repetă se numeşte buclă (loop) sau iteraţie, instrucţiunile cele mai des utilizate specifice controlului buclelor fiind LOOP, LOOPE, LOOPZ, LOOPNE, LOOPNZ.

De exemplu, ***instrucţiunea LOOP*** va decrementa registrul CX (deci se execută instrucţiunea CX=CX-1) și apoi va verifica dacă (CX≠0); dacă DA, atunci execută saltul la eticheta specificată imediat după LOOP, altfel continuă (nu execută salt). În exemplul din secţiunea anterioară, ultimele 2 instrucţiuni se vor înlocui cu *loop eti*.

```
dec cx
jnz eti
```

Se va înlocui cu:

```
LOOP eti
```

7.6. Exerciţii propuse

Scrieţi în simulator în cadrul unui șablon de tip .com (deci folosind directiva org 100h) următoarele secvenţe și răspundeţi la întrebări:

1. Câţi octeţi se ocupă în zona de date, dacă se folosește următoarea directivă?

i) a db 12h,34h,90h
b dw 56h,78h

ii) a dw 12h,34h,90h
b db 56h,78h

iii) a db 12h,34h
b dw 9A56h,78h

iv) a db 12h,34h
b dw 56h,78h,12

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

2. 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 (în simulator).

3. Înlocuiţi cu directive care să folosească operatorul DUP (duplicate):

i) _____; rezultat db ?,?,?,?,?,?
_____; sir db 2,3,4,2,3,4,2,3,4,2,3,4,2,3,4,
_____; sir dw 2,0,2,0,2,0,2,0,2,0,2,0

ii) _____; rezultat db ?,?,?,?,?,?,?,?
_____; sir db 2,3,4,5,2,3,4,5,2,3,4,5,2,3,4,5,2,3,4,5
_____; sir dw 2,1,0,2,1,0,2,1,0,2,1,0

iii) _____; rezultat db ?,?,?,?,?
_____; sir db 1,2,3,4,5,1,2,3,4,5,1,2,3,4,5,1,2,3,4,5,1,2,3,4,5
_____; sir dw 2,1,0,2,1,0

iv) _____; rezultat db ?,?,?,?,?,?,?,?,?,?
_____; sir db 1,2,1,2,1,2,1,2,1,2,1,2
_____; sir dw 3,2,1,0,3,2,1,0,3,2,1,0

4. i) Definiţi un șir de octeţi (în hexazecimal) care să conţină numerele pare de la 0h la 9h, iar apoi un șir de 5 cuvinte care în convenţia cu semn să denote atât numere pozitive cât și numere negative (alternativ):

ii) Definiţi un șir de octeţi (în hexazecimal) care în convenţia cu semn să conţină numerele de la -5 la 5, iar apoi un șir de 5 cuvinte care în convenţia cu semn să denote numere pozitive și în plus, să fie toate pare:

iii) Definiţi un șir de octeţi (în hexazecimal) care să conţină primele 10 numere pare începând de la 34h, iar apoi un șir de 5 cuvinte care în convenţia cu semn să denote numere negative și în plus, să fie toate impare:

iv) Definiţi un șir de octeţi (în hexazecimal) care să conţină primele 10 numere impare începând de la 12h, iar apoi un șir de 5 cuvinte care în convenţia cu semn să denote numere negative și în plus, să fie toate pare:

5. Fie următoarea secvenţă scrisă în EMU:

```
org 100h
.data
a db 5,4,3,2,1,0
.code
mov AL, a+x       ; AL = _____
mov BL, [a+x]     ; BL = _____
mov CL, a [x]     ; CL = _____
 ret
```

unde x e un număr i) x=2; ii) x=1; iii) x=0; iv) x=3. Comentaţi fiecare instrucţiune, scriind rezultatul obţinut în registrul corespunzător.

6. Fie următoarea secvenţă scrisă în EMU:

```
org 100h
.data
a db 5,4,3,2,1,0
.code
lea bx,  a        ; BX = _____
mov si, _____     ; SI = _____
_____             ; AL = _____
ret
```

a) Comentaţi prima instrucţiune (specificaţi o instrucţiune echivalentă), scriind rezultatul obţinut în registrul corespunzător, dar completaţi și cu valoarea ce trebuie scrisă în registrul SI astfel încât în registrul AL să se încarce elementul de pe poziţia

i) 3; ii) 4; iii) 2; iv) 5 din șir.

b) Înlocuiţi apoi prima instrucţiune cu instrucţiunea *mov bx,0* și ultima instrucţiune cu *mov AL,a[BX][SI]*. Comentaţi noua secvenţă de program obţinută.

*** elementele șirului sunt indexate de la poziţia 0 (prima din șir)**

7. Scrieţi un exemplu asemănător celui ilustrat în Figura 7.9 și explicaţi conţinutul zonei de memorie.

8. Fie următoarea secvenţă scrisă în EMU:

```
org 100h
.data
a db 5,4,3,2,1,0
b dw 9,8,7,6
.code
mov bx, 2                    ; BX = _____
mov si, 3                    ; SI = _____
mov al, a[bx][si]            ; AL = _____
mov ax,a[bx][si]             ; AX = _____
add si, 4                    ; SI = _____
mov al, b [si]               ; AL = _____
mov ax, b [bx]               ; AX = _____
ret
```

Dacă veţi încerca să executaţi această secvenţă în EMU, vă va da erori. Specificaţi care sunt instrucţiunile ilegale, corectaţi-le și specificaţi ce se va găsi în registrul menţionat în instrucţiune.

9. Definiţi un șir de 10 cuvinte și reprezentaţi mai jos zona de memorie. Vă puteţi inspira din Figura 7.11. Folosiţi apoi SI = 0,2,4,... pentru a vă deplasa pe fiecare cuvânt și BX=1 pentru a vă deplasa pe fiecare octet MSB din cadrul fiecărui cuvânt. Executaţi secvenţa de 10 ori astfel încât să obţineţi în registrul AL fiecare din cei 10 octeţi MSB ai fiecărui cuvânt. Notaţi valorile în zona corespunzătoare:

```
org 100h
.data
_____ ; definiţi aici șirul

.code
mov si, _____ ;  SI = ____; ____; ____; ____; ____; ____; ____; ____; ____; ____;
mov bx, _____ ; BX = ____; ____; ____; ____; ____; ____; ____; ____; ____; ____;
mov al, _____; AL =  ____; ____; ____; ____; ____; ____; ____; ____; ____; ____;
```

10. Repetaţi exerciţiul 8 astfel încât în locul directivei *dw* folosită pentru a defini șirul, să folosiţi directiva *dd*. Rulaţi în EMU și comentaţi noul rezultat obţinut.

11. Scrieţi următoarea secvenţă în EMU și urmăriţi valoarea din registrul BX pe măsură ce executaţi bucla (Nu executaţi mai mult de 20 ori bucla cu eticheta Aduna).

```
        mov ax,0
        mov bx,2
Aduna:  add ax,bx
        inc bx      ; BX = ___; ___; ___; ___; ___; ___; ___; ___; ___; ___; ___; ___;
        jmp Aduna
```

12. Scrieţi următoarea secvenţă în EMU și comentaţi:

```
org 100h
.data
unu EQU 1
A db 0          ; la ce adresă va fi A?
B dw 110h       ; la ce adresă va fi B?
.code
mov A, unu      ; A = _____
add A, unu      ; A = _____
add B, 10       ; B = _____
add B, unu      ; B = _____
```

13. Scrieţi următoarea secvenţă în EMU și comentaţi:

```
i), ii)                              iii), iv)
org 100h                             org 100h
.data                                .data
sir dw 1,2,3                         sir db 1,2,3,4
VAR db 5                             VAR db 10
.code                                .code
mov BX, OFFSET VAR  ; BX = _____     mov BX, OFFSET VAR  ; BX = _____
mov AX, SEG VAR     ; AX = _____     mov AX, SEG VAR     ; AX = _____
```

14. Care vor fi valorile registrilor după execuţia următoarei secvenţe de cod în EMU? Desenaţi structura stivei și specificaţi conţinutul registrilor.

```
i) , ii) mov ax, 1234h  ; AX = _____        iii) , iv) mov ax, 4321h  ; AX = _____
mov bx, 2468h           ; BX = _____        mov bx, 8642h             ; BX = _____
mov cx, 3579h           ; CX = _____        mov cx, 9753h             ; CX = _____
mov dx, 4321h           ; DX = _____        mov dx, 1234h             ; DX = _____
push ax                                     push ax
push bx                                     push bx
push cx                                     push cx
push dx                                     push dx
pop bx                  ; BX = _____        pop cx                    ; CX = _____
pop dx                  ; DX = _____        pop bx                    ; BX = _____
pop ax                  ; AX = _____        pop ax                    ; AX = _____
pop cx                  ; CX = _____        pop dx                    ; DX = _____
```

15. Specificaţi câţi octeţi ocupă în zona de stivă următoarea secvenţă de instrucţiuni.

```
i) push ax      ii) push ax     iii) push ax    iv) push ax
   push bx          push bx         push bx         add sp,2
   pop cx           add sp,4        pop bx          push bx
   push dx          push ax         pop cx          add sp,2
```

16. Rescrieţi exemplul (Problema rezolvată) din secţiunea 7.4 în ambele moduri (doar cu o singură instrucţiune movsb, respectiv cu eticheta eti și buclarea lui movsb de 5 ori) și analizaţi zona din memorie înainte și după execuţie.
a) Rulaţi cu Single Step și comentaţi fiecare instrucţiune considerând varianta de program cu buclarea instrucţiunii movsb cu ajutorul etichetei eti;

b) Repetaţi punctul a) dar pentru un număr de 10 elemente. Explicaţi cum aţi procedat.

c) Repetaţi punctul a) înlocuind cu loop, așa cum se sugerează în secţiunea 7.5. Explicaţi rolul lui loop.

d) Analizaţi prefixele de repetare rep, repe, repz, repne, repnz din Help-ul simulatorului EMU și adaptaţi problema de la punctul a) a.î. să folosească un prefix de repetare.

e) Modificaţi programul de la a) a.î. elementele șirurilor să fie cuvinte și nu octeţi.

f) Repetaţi punctul b) dar pentru elemente cuvinte.

g) Repetaţi punctul c) dar pentru elemente cuvinte.

h) Repetaţi punctul d) dar pentru elemente cuvinte.

Capitolul 8. Lucrul cu memoria simulatorului

Așa cum am specificat în capitolele anterioare, programul care se execută se găseşte memorat în *segmentul de cod*. Când se încarcă o instrucţiune din memorie, adresa acesteia este furnizată de regiştrii CS (ca adresă de bază) şi respectiv IP (ca deplasament sau offset). În mod normal, conţinutul registrului IP este incrementat pe măsură ce instrucţiunile se execută (cu excepţia execuţiei unor instrucţiuni de salt în care valoarea IP poate fi modificată în orice mod), astfel ca întotdeauna să fie deja selectată instrucţiunea care urmează a se executa.

Depunerea datelor în stivă:

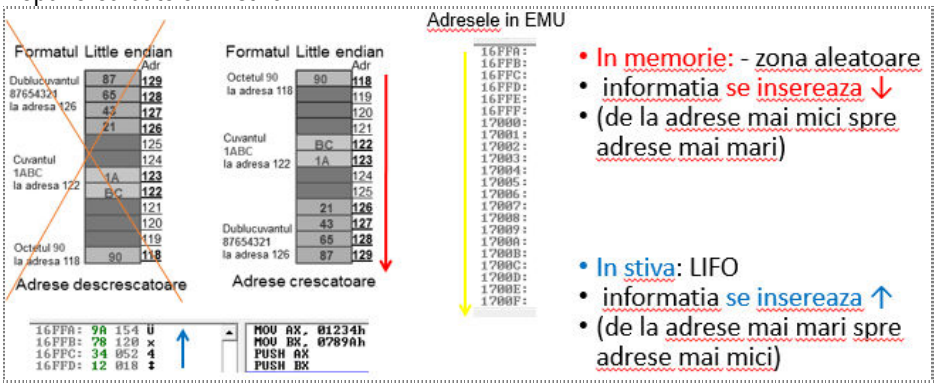


Figura 8.1. Lucrul cu memoria și stiva în simulator

Reamintesc aici regula care se aplică la calculul adreselor fizice:

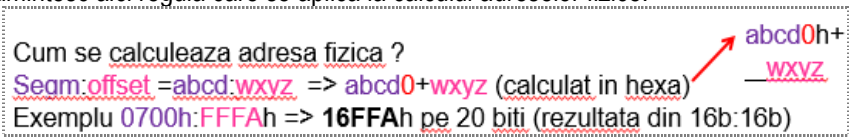


Figura 8.2. Aplicarea regulii de calcul a adresei fizice din segment și offset

8.1. Exemplu de program .COM

În cadrul unui program de tip .com, toate segmentele (de cod, date, stivă, extra) sunt împachetate în cadrul unui singur segment. Astfel, CS=DS=SS=ES, așa cum se poate observa și în cadrul exemplului prezentat în Figura 8.3. Pentru a selecta un program de tip .com în cadrul simulatorului, reamintesc aici că trebuie selectat un nou fișier (opţiunea New), după care se alege un template (șablon) de tip .com.

Cum arată într-un program de tip .com *stiva* și *"memoria"*?

Întrebarea de mai sus ar putea genera o alta: ***Stiva nu e tot în memorie?***

Răspuns: Ba da (în segmentul SS), doar că zona respectivă din memorie se consideră de tip LIFO, și deci lucrul cu ea e diferit (stiva e gestionată altfel decât o zonă "*normală*" din memorie), dar în cadrul programului de tip ***.com***, atât stiva cât și datele (sau memoria clasică) se află în cadrul aceluiași segment. Așa cum arată Figura 8.3, toate sunt în segmentul dat de CS=DS=SS=ES=0700h.

În exemplul ilustrat în Figura 8.3 și Figura 8.4 se prezintă un program în care se folosesc 2 instrucţiuni cu stiva: *push AX* și *push BX*. Figura 8.3 prezintă simulatorul înainte de execuţia programului, iar în Figura 8.4 se poate observa modul în care s-au depus cele 2 elemente pe stivă (deci după execuţia programului).

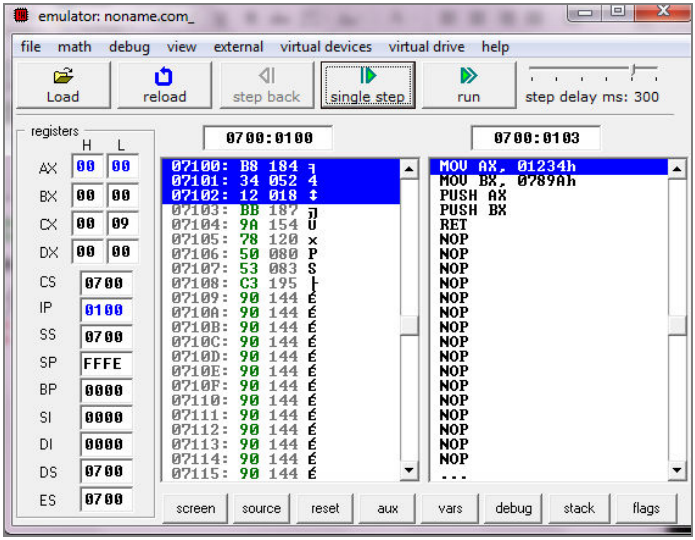


Figura 8.3. Stiva înainte de execuţie (evidenţiată zona de cod)

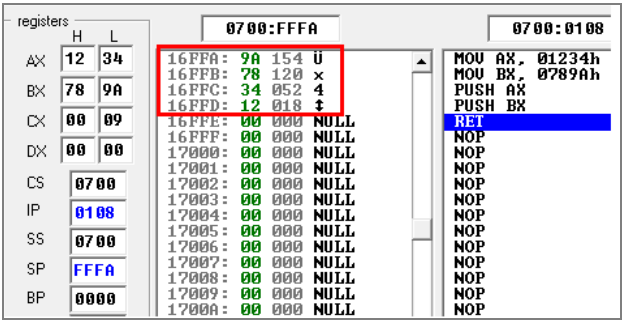


Figura 8.4. Stiva după execuţie (evidenţiată zona de stivă)

Deși se află în cadrul aceluiași segment și datele, și codul și stiva, simulatorul „sare” tot timpul la zona de cod (din cauza actualizării automate a registrului IP spre instrucţiunea ce urmează a se executa). Datele se vor putea vizualiza deasupra zonei de cod (dacă au fost definite corect cu directiva .data), în general prima instrucţiune din program fiind codificată la adresele ce urmează acestei zone în care se află datele. Astfel, datele și codul se pot vizualiza în fereastra simulatorului în mod „automat”. Stiva, în schimb, din cauza gestionării lucrului cu aceasta (se află implementată la celălalt capăt al segmentului), nu este vizualizabilă la fel de ușor: pentru a vedea datele depuse în stivă, utilizatorul va trebui să insereze o valoare mai mică sau egală cu SP în zona adresei logice (în caseta din partea de sus a simulatorului).

8.2. Exemplu de program .EXE

În cadrul programelor de tip .exe, valorile regiștrilor segment DS, SS, CS și ES pot fi diferite. Un posibil exemplu este ilustrat în Figura 8.5.

Exemplu: Selectaţi un program de tip .exe din simulator și comparaţi cu exemplul prezentat. Fie registrele CPU în următoarea stare:
CS=0722h, DS=0700h, SS=0712h, SP=0100h, BP=0000h, BX=5600h, IP=0000h. Specificaţi care va fi adresa următoarei instrucţiuni de executat pe procesor, la ce adresă se află datele accesate cu instrucţiunea mov AX,[BX] și care este locul din memorie unde se află primul și respectiv ultimul element introdus în stivă.
Răspuns: Adresa *instrucţiunii următoare* va fi *CS:IP* adică 0722h:0000h => 07220h
Adresa *unde se află datele* va fi dată de *DS:DI*, *DS:SI* sau *DS:BX* sau o combinaţie a lor, depinde de modul de adresare, așa cum am văzut în *Capitolul 3* (secţiunea 3.6.2). De exemplu, pentru mov AX, [BX]; *data* va fi din *DS:BX*, deci 0700h:5600h, care în urma calculului sugerat în Figura 8.2 va furniza adresa fizică 0C600h.
Adresa *ultimului element introdus în stivă*: *SS:SP*, deci 0712h:0100h => 07220h
Adresa *primului element introdus în stivă*: *SS:BP*, deci 0712h:0000h => 07120h (de aceea, prima oară la lucrul cu stiva e indicată inițializarea lui BP).

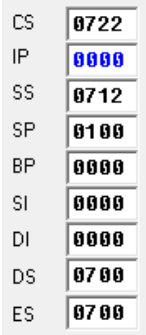


Figura 8.5. Valorile regiștrilor la începerea unui program .exe în simulator

8.3. Exerciţii propuse

1. Analizaţi exemplul de mai jos consultând fiecare instrucţiune pe care nu o cunoaşteţi din Help-ul EMU şi *apoi* executaţi folosind simulatorul; comentaţi efectul fiecărei instrucţiuni cât mai detaliat posibil; completaţi valorile tuturor regiștrilor afectaţi de instrucţiune în zonele indicate.

i), ii)

```
mov AX,0102h  ; _____
mov BX, 0304h ; _____
mov CX, 0506h ; _____
mov DX, 0708h  ; _____
push AX       ; _____
push BX       ; _____
pusha         ; _____
;_____
add CX,0FFh   ; _____
add DX, 0F0Fh ; _____
pop AX        ; _____
popa          ; _____
;_____
```

iii), iv)

```
mov AX,9080h  ; _____
mov BX, 7060h ; _____
mov CX, 5040h ; _____
mov DX, 3020h  ; _____
push AX       ; _____
push BX       ; _____
pusha         ; _____
;_____
add CX,0FFh   ; _____
add DX, 0F0Fh ; _____
pop AX        ; _____
popa          ; _____
;_____
```

2. Analizaţi exemplul de mai jos consultând fiecare instrucţiune pe care nu o cunoaşteţi din Help-ul EMU şi *apoi* executaţi folosind simulatorul; comentaţi efectul fiecărei instrucţiuni cât mai detaliat posibil; completaţi valorile tuturor regiștrilor afectaţi dar notaţi și valoarea flagurilor după fiecare instrucţiune.

i), iii)

```
mov AX,8765h  ; _____
mov BX, 789Bh ; _____
add AX, BX    ; _____
lahf          ; _____
std           ; _____
clz           ; _____
cmc           ; _____
sahf          ;_____
lahf          ;_____
```

ii), iv)

```
mov AX,5678h   ; _____
mov BX, 0A988h ; _____
add AX, BX     ; _____
lahf           ; _____
std            ; _____
clz            ; _____
cmc            ; _____
sahf           ;_____
lahf           ;_____
```

3. Același enunţ ca la punctul 2.

i), ii)

```
mov AX,1234h
xor AX,AX
std
cmc
pushf
cmc
popf
```

iii), iv)

```
mov AX, 8765h
and AX,0
std
cmc
pushf
cmc
popf
```

Capitolul 9. Codificarea instrucţiunilor în simulator

9.1. Depunerea şi extragerea instrucţiunilor din memorie

În momentul când se pregătește programul pentru execuţie, acesta este depus în memorie, instrucţiune cu instrucţiune. Organizarea informaţiei în memorie se realizează prin intermediul segmentelor, folosind adresarea în mod real (segmentată).
Exemplu: **La locaţia 12002h** este stocat ***octetul 48h***, astfel *adresa fizică* 12002h *se va putea scrie* ca și *adresa logică* 1200h:0002h, unde segment va fi **1200h**, iar offset **0002h**. ***Segmentul*** are dimensiunea maximă de 64k, deci atât cât poate fi adresat cu cei 16 biţi (4 cifre hexa): de la 0000h până la FFFFh. Reamintesc aici că un segment nu poate începe decât la adrese multiplu de 16, acestea scriindu-se: abcd**0**h.

De unde din memorie se extrage următoarea instrucţiune?

Dacă exemplul se referă la ***segmentul de cod***, atunci ***octetul 48h*** aflat la adresa 12002h în memorie, ar putea fi octetul obţinut prin codificarea instrucţiunii *dec AX (decrementare)* => *CS=1200h* și *IP=0002h.* Astfel, spunem că din memorie, de la adresa **12002h** s-a extras (operaţia de FETCH) instrucţiunea **48h.** După extragerea acestei instrucţiuni, registrul **IP va fi actualizat în mod automat** astfel încât să se poată apoi extrage următoarea instrucţiune, deci IP va fi incrementat cu 1 în acest caz. Aceste operaţii sunt realizate *automat* de către procesor la execuţia programului.

La modul general: **IP = IP + lungimea instrucţiunii extrase**

Exemple: Instrucţiunea MOV AX, BX -> se va *codifica* sub forma ***8B C3 h*** și deci va ocupa un număr de *2 octeţi în memorie* (octetul de la adresa 07100h și cel de la adresa 07101h în cadrul exemplului ilustrat).

Fără *Deplasament*, fără *Imediat*

```
07100: 8B 139 ï  MOV AX, BX
07101: C3 195 ├
```

Instructiunea MOV AX,1234h -> se va codifica sub forma ***B8 34 12 h*** și deci va ocupa un număr de *3 octeţi în memorie* (în zona de cod).

Fără *Deplasament*, cu *Imediat* pe 2 octeţi (1234h)

```
07100: B8 184 ╕  MOV AX, 01234h
07101: 34 052 4
07102: 12 018 ↕
```

Instrucţiunea MOV [BX+5],1234h -> se va codifica sub forma ***C7 47 05 34 12 h*** și va ocupa un număr de *5 octeţi în zona de cod*.

Cu *Deplasament* pe 1 octet (05h), cu *Imediat* pe 2 octeţi (1234h)

```
07100: C7 199 ╞  MOV w.[BX]+05h, 01234h
07101: 47 071 G
07102: 05 005 ♣
07103: 34 052 4
07104: 12 018 ↕
```

Prin aceste exemple prezentate, doresc să subliniez faptul că există mai multe forme sau moduri de codificare a aceleiași instrucţiuni (*mov* în cazul nostru). Aceste forme depind de numărul și tipul operanzilor folosiţi și vor determina modul de codificare al instrucţiunii (pe un număr mai mic sau mai mare de octeţi). În general, folosirea regiștrilor duce la obţinerea de instrucţiuni mai scurte, în timp ce folosirea memoriei duce la obţinerea de instrucţiuni mai lungi. Alte instrucţiuni, mai rar folosite în general (deci neoptimizate) se pot codifica pe un număr mult mai mare de octeţi.

Figura 9.1. prezintă un exemplu de secvenţă de program de tip ***.com***: mai multe instrucţiuni au fost codificate după cum arată figura, pe un număr de 3, 2 sau chiar 1 octet și depuse în memorie; asamblorul este cel care realizează această codificare (transformarea din *forma cu mnemonică* în *forma de limbaj mașină*, ca valori hexa).

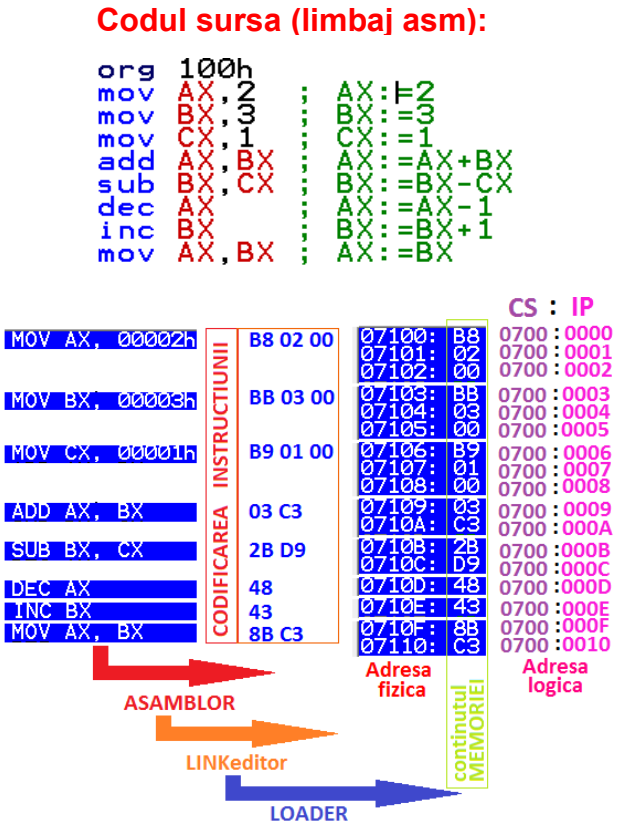


Figura 9.1. De la **cod sursă** (în limbaj de asamblare) la **cod mașină**

Linkeditor-ul și apoi *loader*-ul sunt cele care determină încărcarea programului în memorie la adresele fizice, iar de aici:

- procesorul *va lua rând pe rând câte o instrucţiune* (prin operaţia de prefetch),
- *o va decodifica* (adică o transformă înapoi în cea echivalentă sub formă de mnemonică pentru a ști ce are de executat) și
- în final *o va executa*.

9.2. Codificarea instrucţiunilor

Codificarea instrucţiunilor se realizează în funcţie de modul de adresare folosit.

A. ***Formatul unei instrucţiuni care operează cu memoria*** 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 doi octeţi
11 – adresare registru
reg – indică registrul operand în instrucţiune
r/m – indică modul de calcul al adresei efective (reg. segm implicit)

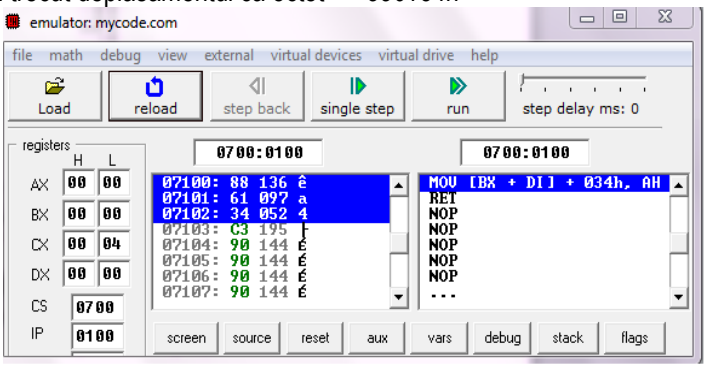
reg	w=0	w=1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

r/m	mod de adresare
000	DS:[BX+SI]
001	DS:[BX+DI]
010	SS:[BP+SI]
011	SS:[BP+DI]
100	DS:[SI]
101	DS:[DI]
110	SS:[BP]
111	DS:[BX]

Exemplu: se dă instrucţiunea:

mov [BX+DI+34h], AH;

arătaţi cum se codifică aceasta (verificaţi și cu EMU)
Răspuns: se transferă *din reg* => d=0, un octet => w=0 ;
se folosește deplasament pe octet => mod=01,
se folosește reg AH=> reg=100,
r/m=001 pentru că se folosește DS :[BX+DI],
va trebui trecut deplasamentul ca octet => *886134h*



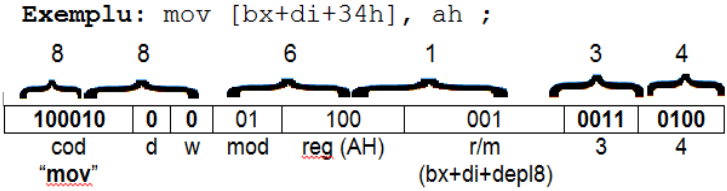


Figura 9.2. Exemplu de codificare a instrucţiunii mov [BX+DI+34h], AH

B. Formatul unei instrucţiuni imediate: în care nu apare operand din memorie, doar registru și valoare imediată, este: **Cod** | **w** | **reg** | depl L | depl H

unde **cod = 1011**, iar câmpurile **w** și **reg** sunt definite ca în cazul precedent.

Exemplu: Din Figura 9.1 se observă că instrucţiunile ***mov AX,2***; ***mov BX,3***; ***mov CX,1*** se codifică diferit: primul octet este ***B8****h* sau ***BB****h* sau ***B9****h*; cel de-al doilea și al treilea octet conţin deplasamentul (***0002****h*, ***0003****h* sau ***0001****h*) sub formă de *word,* acesta fiind depus în memorie după cum precizează convenţia Little End-ian.

1011 1 și **reg** este **000** – pentru AX, **011** pentru BX și **001** pentru CX.

C. Formatul unei instrucţiuni directe: fără operanzi din memorie, fără operanzi de tip imediat – doar regiștri: **Cod | d | w | octet - mod de adresare**

Exemplu:

Instrucţiunea *mov AX,BX* se codifică sub forma **8B C3h** = **100010 11 11 000 011** b,

codOp=100010 (instrucţiunea *mov reg,reg*), **d=1** (în reg), **w=1** (pe cuvânt)

9.3. Exerciţii propuse

1. Analizaţi manual, folosind regulile specificate în material, dar și verificaţi modul de codificare al următoarelor instrucţiuni folosind EMU8086:

i), ii)

```
mov [bx+di+1234h], ah  ; _____
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      ; _____
```

iii), iv)

```
mov [bx+si+5678h], al  ; _____
mov [bx+si+5678h], ax  ; _____
mov ax, [bx+si+5678h]  ; _____
mov cx,dx              ; _____
mov ch,dh              ; _____
mov cl,dl              ; _____
mov sp,bp              ; _____
mov [bx+12],8765h      ; _____
```

2. Folosind EMU8086, scrieţi codificarea pentru următoarele instrucţiuni și comparaţi rezultatele obţinute între ele:

```
add ax,1 ; _____    add bx,1 ; _____    add cx,1 ; _____
sub ax,1 ; _____    sub bx,1 ; _____    sub cx,1 ; _____
dec ax   ; _____    dec bx   ; _____    dec cx   ; _____
inc ax   ; _____    inc bx   ; _____    inc cx   ; _____
```

3. Folosind EMU8086, specificaţi numărul de octeţi și modul cum se codifică următoarele instrucţiuni. Scrieţi o variantă asemănătoare pentru o variabilă definită în memorie pe octet:

```
org 100h
.data
var dw 2
.code
mov var, ax          ; ___; _____
mov var, bx          ; ___; _____
mov byte ptr var, al ; ___; _____
mov byte ptr var, cl ; ___; _____
mov var, 4           ; ___; _____
ret
```

```
org 100h
.data

.code

                     ; ___; _____
                     ; ___; _____
                     ; ___; _____
                     ; ___; _____
                     ; ___; _____
ret
```

4. Folosind EMU8086, scrieţi atât rezultatul obţinut cât și codificarea pentru următoarele instrucţiuni și comparaţi rezultatele obţinute între ele:

```
org 100h
.data
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 dl, sir1 [4] ; _____

mov ax, sir2 [2] ; _____     mov bx, sir2 [3]; _____     mov dx, sir2 [4] ; _____

mov  sir1 [2], ah ; _____    mov sir1 [3], bh; _____     mov sir1 [4], dh; _____

mov sir2 [2], ax ; _____     mov sir2 [3], bx; _____     mov sir2 [4], dx ; _____
```

5. Folosind EMU8086, specificaţi numărul de octeţi și apoi modul cum se codifică următoarele instrucţiuni.

```
xlat  ; ____; _____________________      xchg ax,bx ; ____; _____________________
                                         xchg cx,bx ; ____; _____________________

aaa   ; ____; _____________________      aas ; ____; _____________________
aam   ; ____; _____________________      aad ; ____; _____________________
daa   ; ____; _____________________      das ; ____; _____________________

push ax  ; ____; _____________________   push bx; ____; _____________________
pop ax   ; ____; _____________________   pop bx  ; ____; _____________________

mov bx,1234h ;    urmat apoi de fiecare din următoarele:
mul bl  ; ____; _____________________    imul bl ; ____; _____________________
mul bx ; ____; _____________________     imul bx ; ____; _____________________
```

6. Folosind EMU8086, specificaţi numărul de octeţi și apoi modul cum se codifică următoarele secvenţe de instrucţiuni:

```
mov ax, 1234h                            mov ax, 1234h
rcr ax, 1   ; ___; _____________________ mov cl,4
rcr ax, 4   ; ___; _____________________ rcr ax, cl  ; ___; _____________________
```

7. Folosind EMU8086, analizaţi următoarele instrucţiuni, specificaţi numărul de octeţi și apoi modul cum se codifică următoarele secvenţe de instrucţiuni:

```
movsb  ; ___; _____________________ not ax      ; ___; _____________________
cmpsb  ; ___; _____________________ and ax, 2  ; ___; _____________________
stosb  ; ___; _____________________ and ax, bx ; ___; _____________________
scasb  ; ___; _____________________ and al,bl   ; ___; _____________________

scasw  ; ___; _____________________ xor ax,ax   ; ___; _____________________
                                   mov ax,0    ; ___; _____________________
```

Capitolul 10. Scrierea unei aplicaţii simple

Prin **aplicaţie simplă** înţelegem un program de bază care execută câteva instrucţiuni al căror rezultat poate fi vizualizat într-o formă sau alta în simulator.
Cel mai simplu program este cel de tip COM, generat automat de EMU8086 și care are următorul șablon **(File-> New-> COM template):**

```
org 100h
.data
        ; se definesc variabilele programului
.code
        ; se scriu instrucţiunile
 ret
```

10.1. Exemple de aplicaţii simple folosind memoria

Problema 1.

a) Definiţi o variabilă "x" pe cuvânt, iniţializată cu 3.
b) Definiţi o constantă "doi" de valoare 2.
c) Încărcaţi într-un registru adresa variabilei x.
d) Realizaţi suma celor 2 numere de la punctele a) și b) și depuneţi-o într-un registru.

Urmăriţi etapele indicate pe Figura 10.1, în ordine:
I. Cu directiva org 100h se specifică zona de început a programului să fie 100h; de la acea adresă din memorie se va începe asamblarea programului.

II. începând cu adresa 100h se alocă spaţiu în memorie pentru programul nostru de tip COM (adică DATE și INSTRUCŢIUNI (COD) în cadrul aceluiași segment); se observă că în EMU8086 se alocă CS=DS=SS=ES=0700h. Astfel, prima instrucţiune a programului nostru se va regăsi în memorie la adresa cu offsetul 100h, adică locaţia fizică 07100h; această primă instrucţiune este jmp 104h.

III. directiva .data se folosește pentru a defini variabilele din memorie; se observă că s-au definit 2 variabile: x și doi. Reamintesc din Capitolul 7 că doar pentru x se va aloca spaţiu în memorie, doi fiind o constantă. Astfel, variabila x va ocupa 2 octeţi (fiind o instanţă de tip word) și anume octeţii de la locaţia 07102h și 07103h. Revenind acum la pasul 2, se observă că prin instrucţiunea jmp 104h se realizează saltul peste această zonă unde s-a găsit loc pentru variabilele programului nostru.

Dacă se apasă o singură dată butonul „Single Step" se sare direct la zona de instrucţiuni (definită cu directiva .code) și se colorează instrucţiunea ***mov AX, offset x;*** această instrucţiune a fost codificată pe 3 octeţi și începe la adresa 07104h, adică 0700h:0104h. Se observă că în loc de *offset x* s-a înocuit cu 00102h, adică acesta este offsetul raportat la adresa de început a segmentului (segmentul începe la adresa 0700h). La offsetul 102h s-a găsit loc pentru variabila x, adică 2 octeţi de valoare 3, sau în hexazecimal 0003h.

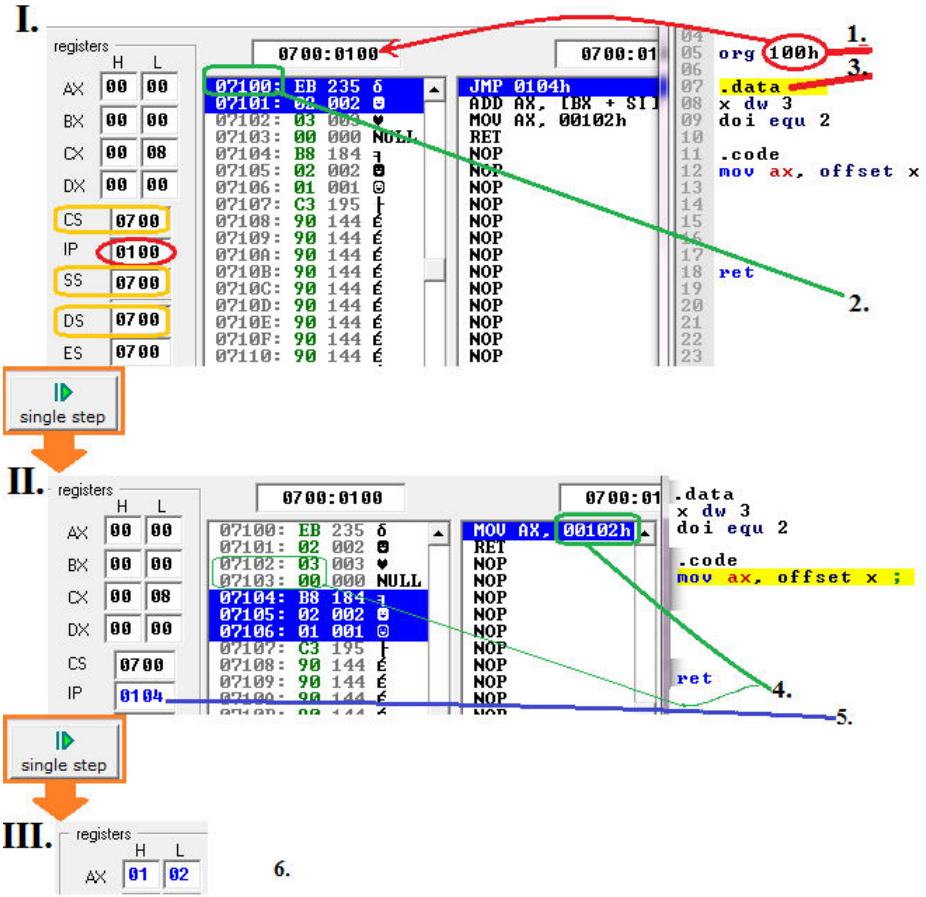


Figura 10.1. Rezolvarea Problemei 1 punctele a), ...,c)

După o nouă apăsare a tastei Single Step, se ajunge în pasul III ilustrat pe Figura 10.1 în partea de jos, în care se poate observa că s-a depus în registrul AX offsetul variabilei x (adică adresa relativă a lui x la începutul segmentului), deci AX=102h.

La punctul d) al problemei s-a cerut calcularea sumei lui x cu doi; o posibilă variantă de rezolvare este ca după încărcarea în AX a adresei lui x, să se încarce în BX valoarea lui x și apoi peste aceasta să se adune valoarea lui doi. Astfel, într-un prim pas, trebuie depusă în BX valoarea lui x.

În Figura 10.2 în partea de sus se observă că instrucţiunea ***mov BX, x*** este codificată pe 4 octeţi și e transformată de asamblor în mov BX, [102h], adică vrea să execute: „mut în BX conţinutul de la adresa având offsetul specificat în [], adică 102h”.

Trebuie acordată atenţie sporită la interpretarea acestei instrucţiuni, întrucât din cauza dimensiunii registrului BX (de 2 octeţi) se va accesa și locaţia următoare din memorie, adică 103h (nu era suficientă o singură locaţie din memorie).

După apăsarea butonului Single Step încă o dată, CPU trece la execuţia instrucţiunii ***add BX,doi*** pe care o transformă în *add BX,02* și pe care o găsim în memorie codificată pe 3 octeţi, așa cum se poate urmări în Figura 10.2 în partea de jos.

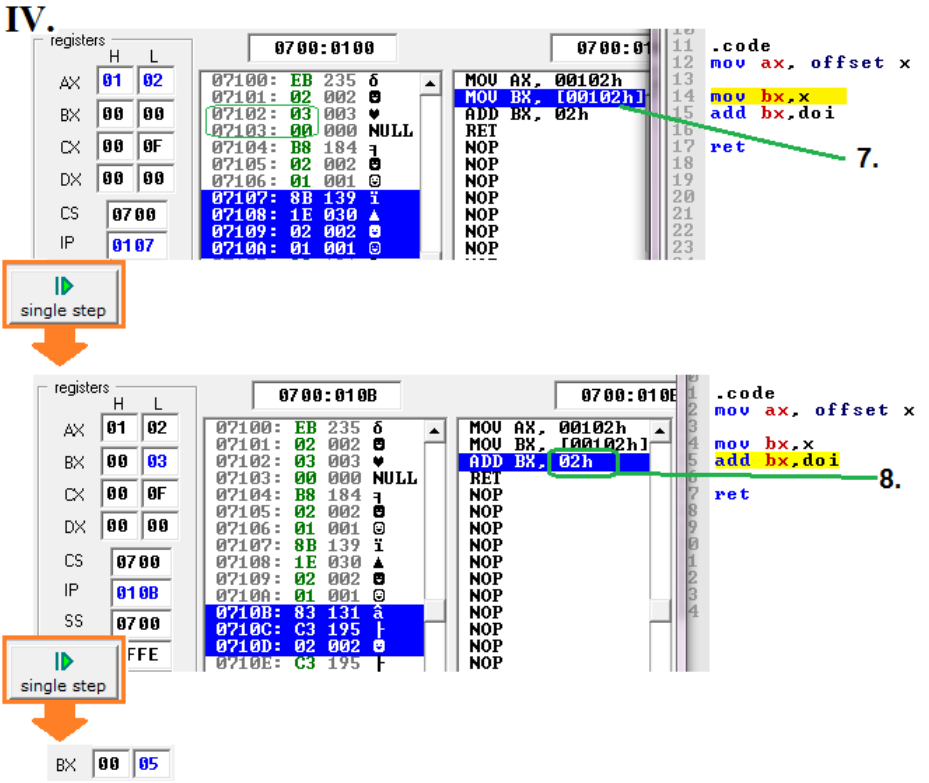


Figura 10.2. Rezolvarea Problemei 1 punctul d)

Trebuie subliniat faptul că în general, accesul la o variabilă stocată în memorie se realizează pe baza adresei, deci cu *mov Registru, offset x;* sau *lea Registru, x;* așa cum am prezentat în exemplele date până acum. În cadrul *problemei 1*, rolul regiștrilor a fost inversat (s-a folosit AX ca *„Registru”* în loc de BX), tocmai pentru a arăta că instrucţiunile sunt flexibile, însă în general, din cauza regulii de adresare a memoriei (relaţia 3 din secţiunea 3.6.2) se va folosi registrul BX ca „Registru” și nu AX.

Problema 2.

În EMU, creaţi un program de tip .com şi scrieţi o secvenţă scurtă de instrucţiuni care să folosească un şir de octeţi; de exemplu:

```
sir db 1,2,3,4
mov BX, offset sir
mov AL, sir[1]
```

a) Explicaţi programul urmărind pas cu pas execuţia acestuia şi interpretând fiecare nouă valoare întâlnită.

b) Modificaţi secvenţa astfel încât valorile şirului să fie definite pe cuvânt. Explicaţi de ce vă dă eroare la execuţie în dreptul instrucţiunii mov AL,sir[1]? Modificaţi astfel încât să fie corect.

Răspuns:

a) se defineşte un şir de octeţi în memorie cu valori 1,2,3,4; în BX se depune offsetul la care începe acest şir (ştim că o adresă fizică se scrie ca adresa logică sub forma segm:offset) şi apoi se încarcă în AL al 2-lea element al şirului (adică de la începutul şirului şi cu 1 în plus, primul fiind cu 0 în plus)

b) sir dw 1,2,3,4 ; s-a modificat directiva de definire şir

mov bx, offset sir

!!! instrucţiunea corectă se scrie: mov AX,sir[1] ; deoarece elementul şirului nu mai încape în AL, e necesar un registru pe 16 biţi.

Problema 3.

a) Pentru variabila *a* s-a găsit loc în memorie începând de la offset 102h (cu un program .com). Realizaţi un desen care să ilustreze zona de memorie (valorile se vor specifica atât în hexazecimal cât şi în binar) şi apoi completaţi în zonele indicate cu valorile corespunzătoare;

b) pentru ultimele 2 instrucţiuni specificaţi şi valoarea flagurilor indicate.

```
org 100h
.data
a dw -127,-1
b db 1,128,255
.code
mov bx, offset b; BX=          _____h=_____b
mov cl,[bx+1]   ; CL= ____d=_____h=_____b
mov dx,[bx+1]   ; DX= ____d=_____h=_____b

add cl,2Fh      ; CL= ____d=_____h=_____b
                ; CF=___; SF=___; OF=___; ZF=___;
add dx,80h      ; DX= ____d=_____h=_____b
                ; CF=___; SF=___; OF=___; ZF=___;
```

Rezolvare:

Zona de memorie va arăta ca mai jos, unde:

-127 = FF81h (la adresa 07102h și 07103h)
-1 = FFFFh (la adresa 07104h și 07105h)
1 = 01h (la adresa 07106h)
1 28 = 80h (la adresa 07107h)
255 = FFh (la adresa 07108h)

```
07102: 81 129
07103: FF 255
07104: FF 255
07105: FF 255
07106: 01 001
07107: 80 128
07108: FF 255
```

```
mov bx, offset b; BX= 106h = 0001 00000110b
mov cl,[bx+1]    ; CL = [107h] = - 128 = 80h=10000000b
mov dx,[bx+1]   ; DX = [108h] = - 128 = FF80h =11111111 10000000b
add cl,2Fh       ; CL= -81=0AFh
add dx,80h       ; DX= 0000h
```

Problema 4.

Creaţi un fișier de tip *.exe* și încercaţi să scrieţi aceeași secvenţă ca la Problema 2 (în zona unde aţi identificat „ *;add your code here* ”) :

```
sir db 1,2,3,4
mov BX, offset sir
mov AL, sir[1]
```

Ce observaţi ? Care este diferenţa faţă de programul de tip *.com*?

Răspuns: Programul rulează ciudat, nu mai execută ca la .com !

Ce s-a întâmplat ?

Directiva de definire a datelor trebuie inserată în locul potrivit, adică în segmentul de date (acolo unde apare „ *;add your data here!* ”), iar

cea cu instrucţiunile să rămână

în zona de cod (acolo unde apare „ *;add your code here* ”).

Rerulaţi și observaţi că după plasarea corectă a datelor în zona de date și a codului în zona de cod, ar trebui să funcţioneze corect.

10.2. Exemple de aplicaţii simple folosind stiva

Problema 5.

Scrieţi o secvenţă care să depună în doi regiștri valorile 4321h și 9876h și apoi plasaţi aceste valori pe stivă. Vizualizaţi-le și apoi interschimbaţi valorile între elefolosind stiva. Explicaţi raţionamentul folosit.

Rezolvare:

Urmăriţi figurile următoare și analizaţi explicaţiile. Dacă nu s-ar fi impus interschimbarea valorilor folosind stiva, operaţia s-ar fi putut realiza foarte simplu, cu instrucţiunea xchg AX,BX (dacă în regiștrii AX și BX s-ar fi găsit cele 2 valori).

În schimb, dacă se dorește interschimbarea valorilor din regiștrii AX și BX folosind stiva, atunci se poate realiza preluarea inversată de pe stivă:

```
push AX        ; se depune pe stivă conţinutul din AX
push BX        ; se depune pe stivă conţinutul din BX
pop AX         ; ceea ce a fost ultima oară depus în stivă se ia în AX,
               ; deci ceea ce a fost în BX va fi acum în AX
pop BX         ; se ia și valoarea de mai devreme în BX,
               ; deci ceea ce a fost în AX va fi acum în BX
```

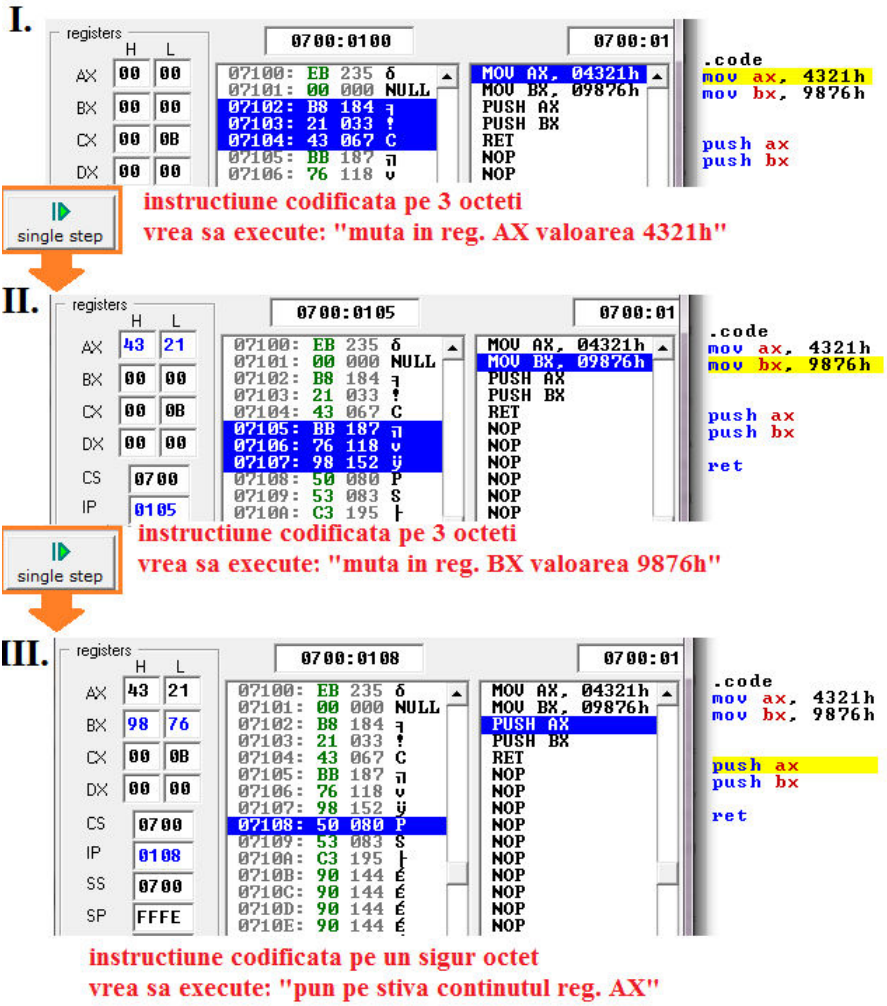


Figura 10.3. Rezolvarea parţială a Problemei 4

10.3. Exemple de aplicaţii simple folosind şiruri

Problema 6

Secvenţa următoare defineşte 2 şiruri: unul sursă şi unul destinaţie, poziţionează regiştrii index pe zonele de început ale şirurilor (vor pointa spre primul element din fiecare şir), iar apoi execută instrucţiunile LODSB şi STOSB, în locul instrucţiunii MOVSB; astfel, elementul din şirul sursă este preluat în registrul acumulator, şi abia apoi depus în şirul destinaţie.

```
SIRs DB 1,2,3,4,5 ;se defineşte şirul destinaţie cu 5 elem. octet, iniţilizat cu 1,2,3,4,5
SIRd DB 5 DUP(0) ;se defineşte şirul sursă cu 5 elemente pe octet, neiniţilizat
...
lea SI, SIRs        ; SI=adr de început a SIRs
lea DI, SIRd        ; DI=adr de început a SIRd
cld                 ; DF=0, şirurile vor fi parcurse în sens crescător
lodsb               ; AL = element curent din SIRs
stosb               ; din AL se depune elementul curent în SIRd
```

Problema 7

Secvenţa următoare verifică în cadrul unui şir cu 10 elemente definite pe octet dacă primul element este egal cu valoarea din registrul AL, prin utilizarea instrucţiunii SCASB:

```
SIRd DB 0,1,2,3,2,4,2,5,2,6 ; se defineşte şirul destinaţie cu 10 elem. pe octet,
                             ; iniţilizat cu 0,1,2,3,2,4,2,5,2,6
...
mov AL, 2           ; numărul (elementul) de căutat
lea DI, SIRd        ; DI=adr de început a SIRd
cld                 ; DF=0, şirul va fi parcurs în sens crescător
scasb               ; se verifică egalitatea elem: cel din şir şi cel căutat
                    ; şi se poziţionează pe următorul elem din şir
```

Problema 8

Exemplul anterior s-ar putea transpune pentru verificarea egalităţii primului element din 2 şiruri prin utilizarea instrucţiunii CMPSB:

```
SIRd DB 0,1,2,3,2,4,2,5,2,6 ; se def şirul destinaţie cu 10 elemente pe octet,
                    ; iniţilizat cu 0,1,2,3,2,4,2,5,2,6
SIRs DB 0,1,2,3,2,4,2,5,2,6 ; se def şirul sursă cu 10 elemente pe octet,
                    ; iniţilizat cu 0,1,2,3,2,4,2,5,2,6
                    ...
lea SI, SIRs        ; SI=adr de început a SIRs
lea DI, SIRd        ; DI=adr de început a SIRd
cld                 ; DF=0, şirul va fi parcurs în sens crescător
cmpsb               ; se verifică egalitatea primului element din cele 2 şiruri
                    ; şi se poziţionează pe următoarele elem din şir
```

10.4. Exerciţii propuse

1. Considerând că simulatorul începe la adresa 0700h:0102h, deci este vorba de un program .com, scrieţi următoarea secvenţă:

i), ii)

```
org 100h

.data
a db 1,2,3,4,5
b dw 10,20,30,40

.code
mov al, a[2]
mov ah, a[3]
push ax
mov bx, b[2]
push bx
push cx
push dx
pop ax
pop bx
```

iii), iv)

```
org 100h

.data
a db 11,22,33,44,55
b dw 50,40,30,20,10

.code
mov al, a[2]
mov ah, a[3]
push ax
mov bx, b[2]
push bx
push cx
push dx
pop ax
pop bx
```

a) realizaţi un desen și ilustraţi zona de memorie, inclusiv cea de stivă;
b) comentaţi fiecare instrucţiune;

2. Modificaţi problemele 6, 7 și 8 astfel încât prelucrările să aibă loc pentru toate elementele șirului, nu doar pentru primul.

3. Repetaţi cerinţa de la 2, astfel încât elementele șirurilor să fie de tip cuvinte.

Partea III

Capitolele 11, 12, 13, 14, 15

Capitolul 11. Acomodarea cu simulatorul.

Aplicaţii simple

Exerciţiile din acest capitol au asociate unul sau mai multe programe exemplu; aceste programe sunt gata scrise în unul din cele două simulatoare și se pot consulta sau executa după nevoie.

Se sugerează studiul programului și al figurii asociate și abia apoi trecerea la exerciţiile propuse de la sfârșitul secţiunii respective.

11.1. Intrări şi ieşiri de date

(EMU) simple_io.asm

Programul ***simple_io.asm*** se exemplifică pentru a arăta cum se pot accesa porturile (virtuale), având adresele posibile de la 0 la 0FFFFh (adică 2^{16}=65536 porturi posibile). Programul foloseşte instrucţiuni specifice porturilor de ieşire (***out***) şi de intrare (***in***), în combinaţie cu date de dimensiuni diferite:

- când intervine ***registrul AL***, datele vehiculate pe port sunt de dimensiune ***octet***,
- când se foloseşte ***registrul AX*** datele sunt de dimensiune **cuvânt**.

Cu instrucţiunea ***in*** se citesc date de la un periferic aflat pe port, iar

cu instrucţiunea ***out*** se trimit/ scriu date înspre un periferic aflat pe port.

Pentru a rula programul *simple_io.asm* este indicat să apăsaţi *Single Step* până la terminarea programului. De asemenea, în rezolvarea primelor exerciţii este indicat să urmăriţi și fereastra corespunzătoare portului virtual, ilustrată în Figura 11.1.

Observaţii:

Instrucţiunea ***out adrPort,AL*** - trimite conţinutul registrului AL (deci un octet) la portul de ieşire având adresa *adrPort*. Circuitul virtual este legat pe ***portul de ieşire 110*** (pentru octet) – așa a fost implementată interfaţa din EMU.

Instrucţiunea ***out adrPort,AX*** - trimite conţinutul registrului AX (deci un cuvânt) la portul de ieşire având adresa *adrPort*. Circuitul virtual este legat pe ***portul de ieşire 112*** (pentru cuvânt) în cadrul simulatorului.

Similar, instrucţiunea ***in AL,adrPort*** sau ***in AX, adrPort*** – asigură o intrare pe 8 sau 16 biţi de la portul cu adresa *adrPort*. Emulatorul aşteaptă introducerea unui octet sau cuvânt şi scrie (eventual şi transformă) valoarea în hexazecimal în registrul AL sau AX.

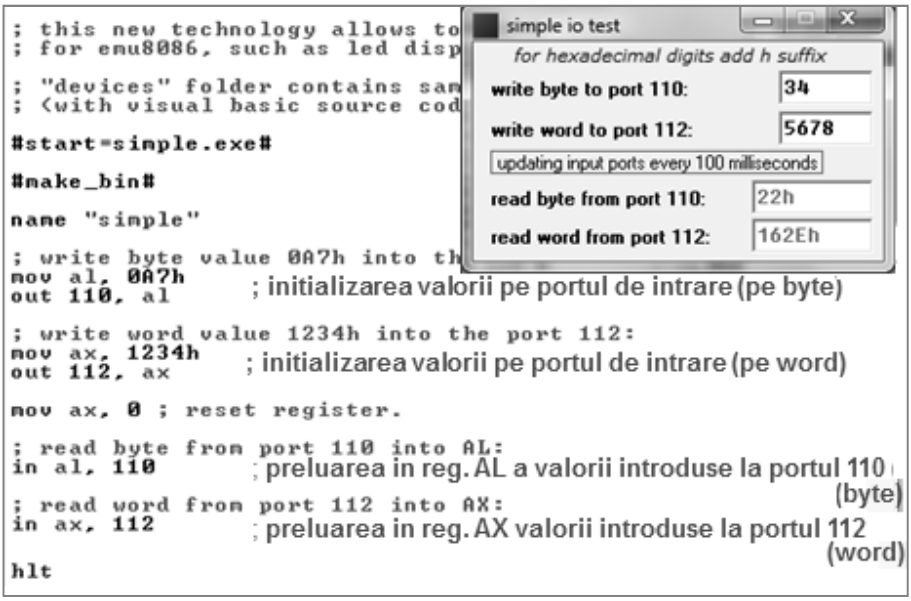


Figura 11.1. Aplicaţia *simple_io.asm* după introducerea datelor de la utilizator

Exerciţii PRACTICE: (se vor rezolva în șablon)

(P) 1. Studiaţi programul *simple_io.asm* din EMU și apoi completaţi în spaţiile goale cu *instrucţiunile adecvate*, după cum se sugerează în comentarii:

_____; încărcaţi valoarea 0A7h în registrul AL
_____; trimiteţi valoarea din registrul AL la portul cu adresa 110
_____; încărcaţi în Accumulator cuvântul de valoare 1234h
_____; trimiteţi valoarea din registrul AX la portul potrivit (aflaţi voi ce adresă)
_____; curăţaţi conţinutul registrului AX – echivalent cu reset
_____; citiţi un octet de la portul cu adresa 110
_____; citiţi un cuvânt de la portul potrivit (aflaţi voi ce adresă)

(P) 2. a) Modificaţi programul de la punctul 1) astfel încât valoarea implicită pe port pentru octet să fie 21h. Scrieţi mai jos modificările propuse:

b) Modificaţi programul de la punctul 1) astfel încât valoarea implicită pe port pentru cuvânt să fie 6543h. Scrieţi mai jos modificările propuse:

c) Modificaţi programul de la punctul 1) astfel încât, cu ajutorul portului de intrare, să preluaţi un octet de valoare 89h. Unde se va regăsi acesta?

Dar dacă se preia un octet de valoare 90?

d) Modificaţi programul de la punctul 1) astfel încât, cu ajutorul portului de intrare, să preluaţi un cuvânt de valoare 6789h. Unde se va regăsi acesta?

Care este valoarea maximă scrisă în zecimal (fără semn) care se poate înscrie pe acest port? Justificaţi răspunsul mai jos:

11.2. Utilizarea numerelor hexazecimale

(SMS) semafor.asm

Aplicaţia propusă simulează prezenţa unui periferic (un semafor) pe ***portul 1*** (de ieşire); astfel, luminile semaforului sunt controlate prin trimiterea datelor la portul 1 al SC. Programul va folosi instrucţiunile ***clo***, ***mov***, ***out***, ***jmp*** şi ***end***. Avem de controlat 6 becuri: roşu, galben, verde pentru 2 semafoare; acest lucru va fi realizat cu ajutorul unui singur octet din care nu se vor folosi ultimii 2 biţi. Prin setarea unui bit la 1, becul corespunzător se va aprinde. Dacă în cazul EMU8086 instrucţiunile de lucru cu portul respectă sintaxa lui 8086, scriindu-se *out adrPort,AL;,* în cazul simulatorului SMS32v50 instrucţiunea e simplificată doar la: ***out adrPort*** și deci implicit se va considera că lucrează cu registrul

AL. Reamintesc aici că SMS este simulator pentru procesor pe 8 biţi, deci nu poate apărea vreo ambiguitate la folosirea accumulatorului.

Controlul becurilor: se poate observa din Figura 11.2 corespondenţa între biţi şi becuri. Prin setarea bitului corespunzător (valoarea 1 în binar) şi transformarea în hexazecimal a întregului număr pe 8 biţi se poate schimba modul de funcţionare al semafoarelor. Un bit de 1 are ca efect aprinderea becului, iar unul de 0 stingerea lui.

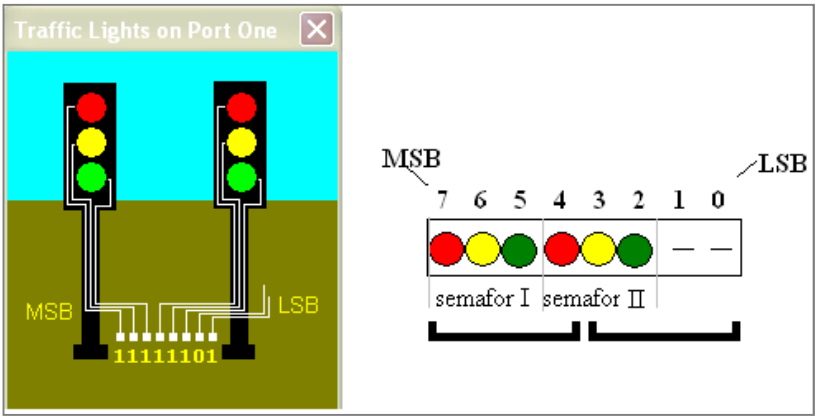


Figura 11.2. Aplicaţia *semafor.asm* pentru controlul becurilor semafoarelor

Etichetele şi instrucţiunea ***jmp*** – etichetele marchează poziţii (adrese) în program care vor fi folosite de comenzile de salt. În acest program toate instrucţiunile vor fi repetate la infinit (în buclă) sau până la apăsarea butonului Stop. Numele unei 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 şi se vor relua instrucţiunile din acel punct. Eticheta destinaţie a saltului se încheie cu ":", de exemplu „***Start:***"

Exerciţii PRACTICE: (se vor rezolva în şablon)

(P) 1. Studiaţi secvenţa de mai jos, completaţi în spaţiile goale cu *instrucţiunile adecvate*, după cum se sugerează în comentarii şi apoi scrieţi secvenţa în SMS şi rulaţi:

```
clo                 ; închide ferestrele care nu sunt necesare
Start:  _____       ; încărcaţi valoarea 0 în registrul AL
        _____       ; trimiteţi valoarea din registrul AL la portul cu adresa 1
        _____       ; încărcaţi o valoare în reg. AL a.î. să se aprindă toate becurile
        _____       ; trimiteţi valoarea la portul potrivit pt a se reflecta în interfaţă
        jmp Start   ; salt înpoi la Start – observaţi implementarea buclei infinite
        end         ; programul se încheie
```

(P) 2. Modificaţi programul de la punctul 1) astfel încât să aprindeţi becurile roşu la ambele semafoare, iar apoi secvenţa de funcţionare să fie în antifază (galben şi roşu la un semafor, verde la celălalt, schimbându-se apoi între ele). Puteţi folosi instrucţiunea *nop* pentru a introduce eventuale întârzieri în realizarea operaţiilor:

(EMU) traffic_lights2.asm

Programul foloseşte instrucţiunile ***mov***, ***out***, ***rol*** şi ***jmp***. Luminile semafoarelor sunt controlate prin trimiterea datelor la ***portul 4***, unde este conectat dispozitivul virtual de control al ledurilor semafoarelor. Prin setarea unui bit la 1, ledul corespunzător se aprinde, legătura fiind ilustrată în Figura 11.3. Instrucţiunea *out 4, AX;* copiază conţinutul registrului AX la portul de ieşire 4 și abia când se execută această instrucţiune (cu portul) se vor aprinde/ stinge ledurile. Se pot controla în total un număr de 12 leduri: roşu(R), galben(Y), verde(G) (în această ordine) pentru cele 4 semafoare. Astfel, controlul ledurilor poate fi realizat prin doi octeţi (16 biţi) din care nu folosim cei mai semnificativi 4 biţi: **xxxx $G_4Y_4R_4G_3$ $Y_3R_3G_2Y_2$ $R_2G_1Y_1R_1$b**.

Implementarea buclei s-a realizat în mod identic cu cea din programul anterior, bucla fiind și aici infinită: toate instrucţiunile sunt repetate la infinit sau până la apăsarea butonului Stop. Directiva ***EQU*** este folosită pentru a defini constante (și nu date în memorie); de exemplu, prin *red EQU 0000_0001b* se defineşte constanta red cu valoarea binară 00000001b. Instrucţiunea ***nop*** (de la *no operation)* introduce o întârziere în prelucrarea datelor. Instrucţiunea ***rol*** (rotate on left) face ca bitul c.m.s. (MSb) să treacă atât în CF (Carry Flag) cât şi în bitul c.m.p.s. (LSb) din operand. Instrucţiunea *rol* este cea care se execută atunci când apare semnul „**<<**” (analogie cu operatorul din C). De ex.: *mov AX, green << 3* va încărca în AX valoarea constantei green, deplasată spre stânga cu 3 poziţii; octetul 0000_0100b va deveni 0010_0000b, deci bitul 5 va fi 1, adică ledul verde de la al doilea semafor va fi aprins (se va controla semaforul 2 în loc de semaforul 1). Astfel, se pot controla 2 sau mai multe semafoare adiacente; când unul e activ, celălalt sau celelalte sunt oprite.

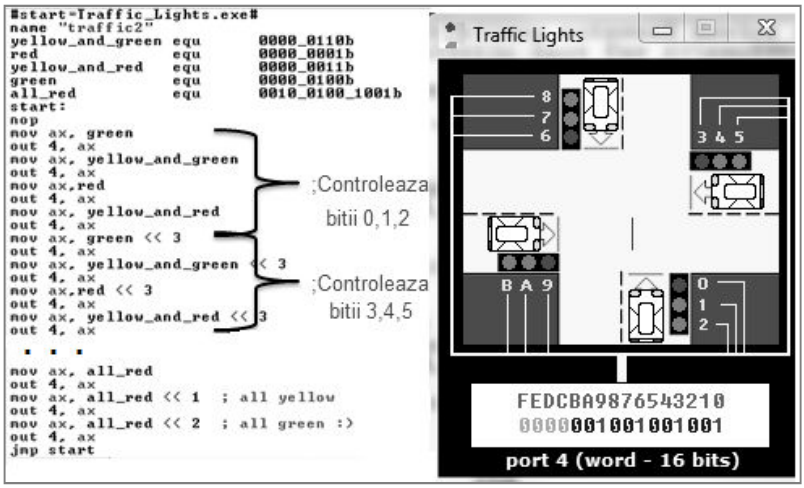


Figura 11.3. Fereastra aplicației *traffic_lights*

Exerciții PRACTICE: (se vor rezolva în șablon)

(P) 1. a) Studiați programul *traffic_lights2.asm* din EMU și apoi completați în spațiile goale cu *directivele adecvate*, după cum se sugerează în comentarii. Scrieți valorile și în binar și în hexazecimal: în instrucțiune în binar și în comentariu în hexazecimal sau invers, cum doriți.

_____ ; definiți o constantă pentru aprinderea ledurilor
; de culoare Verde și Galben (pe semaforul 1)
_____ ; definiți o constantă pentru aprinderea ledului
; de culoare Roșu (pe semaforul 1)
_____ ; definiți o constantă pentru aprinderea ledurilor
; de culoare Galben și Roșu (pe semaforul 1)
_____ ; definiți o constantă pentru aprinderea ledului
; de culoare Verde (pe semaforul 1)
_____ ; definiți o constantă pentru aprinderea tuturor ledurilor
; de culoare Roșu (pe toate semafoarele)

b) Adăugați apoi și *instrucțiunile adecvate*, după cum se sugerează în comentarii. Scrieți valorile și în binar și în hexazecimal, ca în exercițiul anterior.

start: nop ; no operation – introduce o întârziere în prelucrare
\
_____; încărcați în AX o valoare pt a seta culoarea Verde pe semaforul 1
_____; trimiteți această valoare pe portul corespunzător aplicației
_____; încărcați în AX o val. pt a seta culorile Verde și Galben pe semaf. 1
_____; trimiteți această valoare pe portul corespunzător aplicației
_____; încărcați în AX o valoare pt a seta culoarea Roșu pe semaforul 1
_____; trimiteți această valoare pe portul corespunzător aplicației

_____; încărcaţi în AX o val. pt a seta culorile Galben și Roșu pe semaf. 1
_____; trimiteţi această valoare pe portul corespunzător aplicaţiei

c) Adăugaţi apoi și *instrucţiunile adecvate* pentru controlul semaforului din dreapta, după cum se sugerează în comentarii:

_____; încărcaţi în AX o valoare pt a seta culoarea Verde pe semaforul 2
_____; trimiteţi această valoare pe portul corespunzător aplicaţiei
_____; încărcaţi în AX o val. pt a seta culorile Verde și Galben pe semaf. 2
_____; trimiteţi această valoare pe portul corespunzător aplicaţiei
_____; încărcaţi în AX o valoare pt a seta culoarea Roșu pe semaforul 2
_____; trimiteţi această valoare pe portul corespunzător aplicaţiei
_____; încărcaţi în AX o val. pt a seta culorile Galben și Roșu pe semaf. 2
_____; trimiteţi această valoare pe portul corespunzător aplicaţiei

d) Adăugaţi apoi și *instrucţiunile adecvate* pentru controlul semaforului de sus:

_____; încărcaţi în AX o valoare pt a seta culoarea Verde pe semaforul 3
_____; trimiteţi această valoare pe portul corespunzător aplicaţiei
_____; încărcaţi în AX o val. pt a seta culorile Verde și Galben pe semaf. 3
_____; trimiteţi această valoare pe portul corespunzător aplicaţiei
_____; încărcaţi în AX o valoare pt a seta culoarea Roșu pe semaforul 3
_____; trimiteţi această valoare pe portul corespunzător aplicaţiei
_____; încărcaţi în AX o val. pt a seta culorile Galben și Roșu pe semaf. 3
_____; trimiteţi această valoare pe portul corespunzător aplicaţiei

e) Adăugaţi apoi și *instrucţiunile adecvate* pentru controlul semaforului din stânga:

_____; încărcaţi în AX o valoare pt a seta culoarea Verde pe semaforul 4
_____; trimiteţi această valoare pe portul corespunzător aplicaţiei
_____; încărcaţi în AX o val. pt a seta culorile Verde și Galben pe semaf. 4
_____; trimiteţi această valoare pe portul corespunzător aplicaţiei
_____; încărcaţi în AX o valoare pt a seta culoarea Roșu pe semaforul 4
_____; trimiteţi această valoare pe portul corespunzător aplicaţiei
_____; încărcaţi în AX o val. pt a seta culorile Galben și Roșu pe semaf. 4
_____; trimiteţi această valoare pe portul corespunzător aplicaţiei

f) Adăugaţi apoi și *instrucţiunile adecvate* pentru controlul tuturor semafoarelor:

_____; încărcaţi în AX o valoare pt a aprinde toate ledurile Roşii
_____; trimiteţi această valoare pe portul corespunzător aplicaţiei
_____; încărcaţi în AX o val. pt a aprinde toate ledurile Galbene
_____; trimiteţi această valoare pe portul corespunzător aplicaţiei
_____; încărcaţi în AX o valoare pt a aprinde toate ledurile Verzi
_____; trimiteţi această valoare pe portul corespunzător aplicaţiei

g) Adăugaţi *instrucţiunea* necesară pentru reluarea întregii secvenţe de la eticheta Start, în mod automat, necondiţionat: _____

2. Scrieţi o directivă prin care să definiţi o constantă pentru a aprinde toate ledurile Galbene și Verzi de la toate semafoarele:

_____ ;

3. Rulaţi programul cu *run*, setând un timp de întârziere cât mai mare. Încercaţi să observaţi fiecare comandă ce ajunge pe port. Repetaţi rulând *pas cu pas*;

(P) 4. Înlocuiţi operatorul "<<" cu instrucţiunea corespunzătoare pentru a obţine același efect în program:

_____ ;

5. Scrieţi o valoare pe 16 biţi cu ajutorul căreia (trimisă pe port) să se aprindă ledurile Roșii de la semaforul din dreapta și cel din stânga, împreună cu ledurile Verzi de la semafoarele de jos și cel de sus:

_____ ;

6. Scrieţi o valoare pe 16 biţi cu ajutorul căreia (trimisă pe port) să se aprindă ledurile Roșu și Galben de la semaforul din dreapta și cel din stânga, împreună cu ledurile Galbene de la semafoarele de jos și de sus:

_____ ;

7. Scrieţi o valoare pe 16 biţi cu ajutorul căreia să se aprindă ledurile Verzi de la semafoarele 2 şi 4, împreună cu ledurile Roşii de la semafoarele 1 şi 3:

_____ ;

8. Scrieţi o valoare pe 16 biţi cu ajutorul căreia să se aprindă ledurile Galbene de la semafoarele 2 şi 4, împreună cu ledurile Roşii şi Galbene de la semafoarele 1 şi 3:

_____ ;

(P) 9. Introduceţi în mod corespunzător (într-un singur program) secvenţele de la punctele 4 …7 şi rulaţi programul obţinut cu *run*.

10. Indicaţi dimensiunea cea mai potrivită pentru un registru (ţinând cont de aspectele întâlnite până acum la proiectarea procesoarelor din familia x86, precum dimensiunea uzuală a regiștrilor și considerând că un registru de dimensiune mai mare implică automat și costuri mai ridicate de implementare) dacă se dorește realizarea unei aplicaţii prin care să se controleze un dispozitiv de afișaj: afișajul trebuie să primească deodată 8 biţi de date și apoi alţi 2 biţi prin care să se controleze anumite efecte la afișare.

Justificaţi răspunsul: _____

_____

Dar dacă am avea la dispoziţie doar 4 fire pentru a comanda acest afișaj de 8 biţi? Care ar fi cea mai bună soluţie?

Justificaţi răspunsul: _____

_____

(EMU) traffic_lights.asm

O variantă îmbunătăţită a aplicaţiei *traffic_lights2.asm* poate fi urmărită în *traffic_lights.asm* (tot din directorul ***examples*** al EMU8086). Aici, în locul definirii comenzilor ca în programul *traffic_lights2.asm*, cu directiva EQU, s-a optat pentru o altă modalitate: s-a folosit directiva ***dw*** (define word) pentru a crea ***un tabel de date***. Secvenţa de la sfârșitul programului defineşte o zonă de memorie (ilustrată în Figura 11.4), începând de la cuvântul adresabil prin numele „situation”, urmat apoi de cuvântul s1, şi aşa mai departe, încheindu-se cu caracterul „$”.

Parcurgerea instrucţiunilor programului are loc tot în buclă și de această dată, însă datorită folosirii tabelului de date, trebuie introdusă noţiunea de ***pointer la șir***. O instrucţiune de forma *mov AX,[SI]* va folosi registrul SI ca pointer sau index la tabelul de date definit în memorie; astfel, va fi nevoie de:

1) o *etapă de iniţializare* în care pointerul să fie depus corect pe zona din memorie de unde se dorește începerea parcurgerii șirului; în general se folosesc instrucţiuni precum *mov SI,* ***offset*** *sir;* sau ***lea*** *SI, sir*; acestea având același efect asupra lui SI;

2) o *etapă de folosire a pointerului pe elementul curent* – operaţia în sine; în cazul de faţă implementată prin instructiunea ***mov AX,[SI]***; prin care se încarcă în registrul AX cuvântul din memorie adresat de pointerul SI la acel moment;

3) o *etapă de pregătire a pointerului pentru următoarea parcurgere*: în general se folosesc instrucţiuni care ajustează pointerul pentru a pointa spre următorul element din tabela de date/ șir; în cazul de faţă se folosește *add SI,2*; întrucât elementele din șir sunt de tip word.

Programul folosește întârzieri în execuţie prin care CPU aşteaptă 5 secunde. Acestea sunt realizate prin intermediul unei întreruperi: se apelează întreruperea cu tipul 15h, serviciul 86h. Aceste întreruperi, cu serviciile asociate lor, trebuie văzute asemănător funcţiilor din programele HLL: sunt gata construite, trebuie doar să ştim să le utilizăm. Mai multe detalii se pot urmări în *Capitolul 12*.

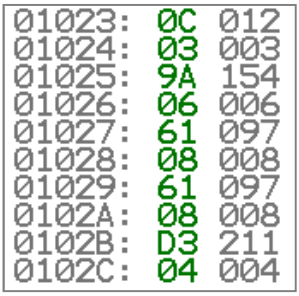


Figura 11.4. Zona de memorie începând cu variabila *situation* din aplicaţia *traffic_lights*

Exerciţii PRACTICE: (se vor rezolva în șablon)

(P) 1 a). Studiaţi programul *traffic_lights.asm* din EMU și apoi completaţi în spaţiile goale cu *directivele adecvate*, după cum se sugerează în comentarii:

```
_____ ; definiţi o variabilă de tip word cu numele situation
                ; având valoarea 030Ch scrisă în binar
_____ ; definiţi o variabilă de tip word cu numele s1
                ; având valoarea 069Ah scrisă în binar
_____ ; definiţi o variabilă de tip word cu numele s2
                ; având valoarea 0861h scrisă în binar
_____ ; definiţi o variabilă de tip word cu numele s3
                ; având valoarea 04D3h scrisă în binar
sit_end = $     ;doar scrieţi această directivă – ea marchează sfârșitul variabilei anterioare
_____; definiţi o constantă cu numele all_red care să aprindă
                ; ledurile Roșii (de la toate semafoarele)
```

b) Adăugaţi apoi și *instrucţiunile adecvate*, după cum se sugerează în comentarii:

```
_____   ; depuneţi în AX constanta care aprinde toate ledurile Roșii
_____   ; reflectaţi această situaţie în interfaţă
```

c) Scrieţi o instrucţiune care să poziţioneze pointerul SI la începutul zonei de memorie unde s-a definit variabila situation. Aceasta este etapa de iniţializare a pointerului:

```
_____   ; poziţionaţi SI ca pointer la zona de memorie unde
        ; începe variabila de tip word situation
```

d) Completaţi în secvenţa de mai jos așa cum se sugerează în comentarii:

```
next:           ; eticheta next este folosită pentru a se implementa o buclă
        _____; încărcaţi în acc. elementul pointat de SI din tabelul de date
        _____; reflectaţi valoarea din accumulator pe interfaţă

        mov CX, 4Ch     ; secvenţa necesară introducerii unei întârzieri: aşteaptă
        mov DX, 4B40h   ; 5 sec (5 milioane microsecunde) 004C4B40h = 5,000,000
        mov AH, 86h     ; dacă se consultă lista întreruperilor din EMU, la instrucţ.
        int 15h         ; int15h cu serv. 86h, valoarea întârzierii e dată în CX:DX
        _____; actualizaţi pointerul SI pentru a trece la elementul următor
                        ; care se află în memorie în tabelul de date situation
        cmp SI, sit_end ; se verifică dacă s-a ajuns la sfârşit
        jb  next        ; dacă încă nu, atunci se sare la eticheta next
        mov SI, offset situation ; dacă DA, se reia de la început – reiniţializare pointer SI
        jmp next        ; salt necondiţionat (automat) la eticheta next
```

2. Vizualizaţi și redaţi mai jos variabilele programului. Explicaţi cum se realizează aceasta:

_____

_____

3. Vizualizaţi zona din memorie unde s-au depus datele definite cu directiva *dw*.

a) Calculaţi adresa pe 20 biţi formată din DS:SI, unde DS este segmentul, iar SI este offsetul variabilei *situation*. Se indică vizualizarea acestor regiştri imediat după execuţia instrucţiunii *mov SI, offset situation*. _____

_____

b) Specificaţi adresa de început și adresa de sfârșit a zonei din memorie în care se depune variabila *situation*:_____

_____

4. Observaţi apariţia instrucţiunii *cmp* şi explicaţi modul cum s-a utilizat aceasta:

_____

_____

_____

Studiaţi instrucţiunea *cmp* din Help și scrieţi mai jos exemplul sugerat acolo:

5. Observaţi modul cum a fost scris programul în cadrul șablonului: la început apare eticheta *Start* și apoi sunt instrucţiunile programului; abia la sfârșit sunt inserate directivele care definesc tabelul de date în memorie și eventualele constante. Rulaţi programul obţinut la punctele anterioare.

(P) 6. Rescrieţi programul, efectuând următoarele modificări în șablon: aduceţi secvenţele de definire a datelor de la sfârșit la început, precedate de directiva *.data*; apoi, exact înainte de a începe zona de instrucţiuni, inseraţi directiva *.code*. Rulaţi acum programul și verificaţi funcţionalitatea sa.

```
.data
        ; zona de definire a datelor
.code
        ; zona de instrucţiuni
```

11.3. Operaţii de incrementare/decrementare

(SMS): Afişaj cu 7 segmente: afisaj.asm

Prin controlul corespunzător al biţilor, segmentele afişajului implementat la ***portul 2*** în SMS se pot aprinde şi stinge comandat. Bitul cel mai puţin semnificativ (cel din dreapta) este folosit pentru selectarea unui afişaj din cele două (acest bit va specifica afişajul activ) așa cum se observă și în Figura 11.5. Setaţi valoarea acestui bit la “0” pentru a selecta afişajul din stânga sau la “1” pentru a selecta afişajul din dreapta.

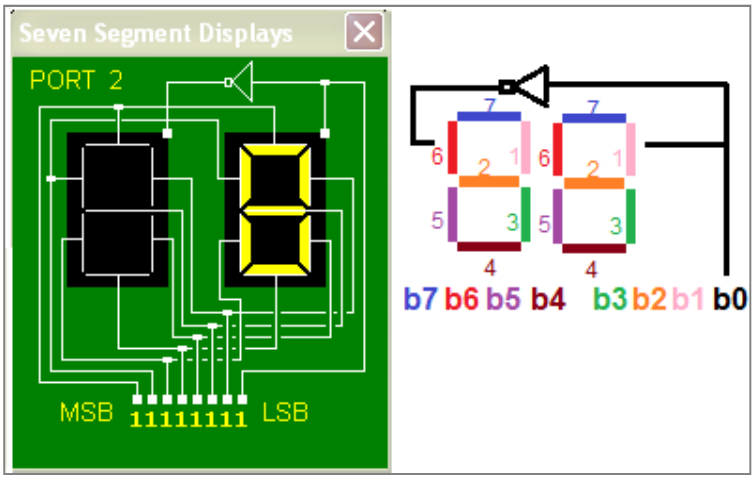


Figura 11.5. Afişaj cu 7 segmente

Exerciţii PRACTICE: (se vor rezolva în şablon)

(P) 1. Studiaţi secvenţa de mai jos, completaţi în spaţiile goale cu *instrucţiunile adecvate*, după cum se sugerează în comentarii și apoi scrieţi secvenţa în SMS și rulaţi:

```
clo                 ; închide ferestrele care nu sunt necesare
Start:  _____       ; încărcaţi o valoare în registrul AL astfel încât să selectaţi
                    ; afișajul din dreapta și acesta să aibă toate segmentele aprinse
        _____       ; trimiteţi valoarea din registrul AL la portul corespunzător
        _____       ; încărcaţi o valoare în registrul AL astfel încât să selectaţi
                    ; afișajul din dreapta și acesta să aibă toate segmentele stinse
        _____       ; trimiteţi valoarea la portul potrivit pt a se reflecta în interfaţă

        _____       ; încărcaţi o valoare în registrul AL astfel încât să selectaţi
                    ; afișajul din stânga și acesta să aibă toate segmentele aprinse
        _____       ; trimiteţi valoarea din registrul AL la portul corespunzător
        _____       ; încărcaţi o valoare în registrul AL astfel încât să selectaţi
                    ; afișajul din stânga și acesta să aibă toate segmentele stinse
        _____       ; trimiteţi valoarea la portul potrivit pt a se reflecta în interfaţă

        jmp Start   ; salt înpoi la Start – observaţi implementarea buclei infinite
        end         ; programul se încheie
```

(P) 2. Modificaţi programul de la punctul 1) astfel încât să aprindeţi un singur segment la alegere (ori cel din stânga, ori cel din dreapta) și să comandaţi afișarea cifrelor zecimale: 0, apoi 1, apoi 2, ș.a.m.d. până la 9:

(P) 3. a) Adăugaţi la secvenţa de la punctul 2) toate instrucţiunile necesare astfel încât afișarea să se reia atunci când se ajunge la 9, din nou de la 0.

b) Repetaţi punctul a) dar în sens descrescător: 9,8,7,...,0,9,8, ...:

a) **b)**

(P) 4. Combinaţi rezolvările de la punctul 3 astfel încât să alternaţi folosirea celor 2 afișaje: în timp ce afișajul din stânga numără crescător, cel din dreapta va număra descrescător. Pentru aceasta, folosiţi un tabel de date: construiţi un șir în memorie cu „comenzile" (valorile) necesare pentru afişarea cifrelor de la 0 la 9 și parcurgeţi șirul într-un sens sau în altul. Scrieţi mai jos programul și explicaţiile necesare:

(P) 5. Scrieţi un nou program astfel încât pe afișaj să apară mesajul H E L L O astfel: prima dată va apărea HE, apoi EL, apoi LL și apoi LO, după care se va relua de la început cu o perioadă de pauză în care se vor stinge toate segmentele. Se sugerează și aici folosirea unui tabel de date în care să se definească valorile:

(EMU) LED_display_test.asm

Aplicaţia este una foarte simplă (apar doar instrucţiuni ***mov***, ***out***, ***inc***, ***jmp***), foloseşte operaţii de incrementare/ decrementare asupra registrului AX şi salturi necondiţionate în program (se asigură revenirea într-un anumit punct al programului în mod automat).

Acest exemplu foloseşte ***portul 199*** pentru a emula existenţa unui dispozitiv virtual precum cel din Figura 11.6. Valorile afişate pe afişaj (display) sunt în zecimal. Executaţi programul cu Run și abia apoi cu Single Step.

Figura 11.6. Fereastra aplicaţiei *LED_display.asm*

Exerciţii PRACTICE: (se vor rezolva în şablon)

(P) 1. a) Studiaţi programul *LED_display_test.asm* din EMU și apoi completaţi în spaţiile goale cu *instrucţiunile adecvate*, după cum se sugerează în comentarii:

_____ ; depuneţi în AX valoarea 1234
_____ ; reflectaţi această situaţie în interfaţă și urmăriţi execuţia
_____ ; depuneţi în AX valoarea -5678
_____ ; reflectaţi această situaţie în interfaţă și urmăriţi execuţia

b) Completaţi în secvenţa de mai jos așa cum se sugerează în comentarii:

_____; încărcaţi în accumulator valoarea 0
x1: ; eticheta x1 este folosită pentru a se implementa o buclă
_____; reflectaţi valoarea din accumulator pe interfaţă
_____; incrementaţi valoarea din accumulator
_____ ; scrieţi instrucţiunea pentru a realiza un salt necondiţionat
; (automat) înapoi la eticheta x1 pentru buclare
hlt ; se suspendă execuţia programului

(P) 2. Modificaţi o instrucţiune în programul de la punctul 2 astfel încât valoarea de la care începe incrementarea să fie 10:_____

(P) 3. Modificaţi o instrucţiune în programul de la punctul 2 astfel încât numărarea să se realizeze din 2 în 2: _____

(P) 4. Studiaţi instrucţiunea *cmp* din help, adăugaţi încă o etichetă x2 la care să sară programul în cazul în care conţinutul registrului AX a depăşit valoarea 50 şi să ajungă la execuţia hlt. Instrucţiunea cmp trebuie urmată de instrucţiunea de salt condiţionat şi abia apoi de instrucţiunea de salt automat - se va furniza un program conţinând rezolvarea.

(P) 5. Modificaţi instrucţiunea de salt necondiţionat *jmp* astfel încât dintr-un salt necondiţionat să apară un salt condiţionat cu flagul ZF. Atenţie! Pentru vizualizare e posibil să fie nevoie să ajustaţi valoarea vitezei de execuţie a instrucţiunilor (cursorul din dreapta sus) - dacă optaţi tot pentru modul Run.

(P) a1) Scrieţi instrucţiunile echivalente astfel încât pe display să apară valorile de la 0 până la 100 (cu incrementare cu 1).

(P) a2) Repetaţi apoi pentru incrementare cu 2;

(P) b1) Scrieţi instrucţiunile echivalente astfel încât pe display să apară valorile de la 100 până la 0 (cu decrementare cu 1).

(P) b2) Repetaţi apoi pentru decrementare cu 2;

(P,P) c) Modificaţi instrucţiunea de salt necondiţionat jmp astfel încât dintr-un salt necondiţionat să apară un salt condiţionat de situaţia mai mic (sau egal) / mai mare (sau egal). Repetaţi a1) și b1) ca c1), c2);

(P) d) Modificaţi programul prin adăugarea unei etichete suplimentare la care să sară programul în cazul în care valoarea din AX a depășit valoarea 50 (începe de la 0 și incrementează cu 1). Când se întâmplă aceasta, scrieţi instrucţiuni astfel încât pe display să apară valoarea 9999.

(SMS) Şarpe în labirint: labirint.asm

Aplicaţia propusă presupune deplasarea şarpelui în labirint astfel încât să fie ocolite obstacolele. Direcţiile sus, jos, stânga, dreapta sunt controlate de cei 4 biţi din stânga (a se urmări Figura 11.7), iar distanţa parcursă (pasul şarpelui) este controlată de cei 4 biţi din dreapta. Pentru a reseta şarpele trimiteţi FF la ***portul 04***.

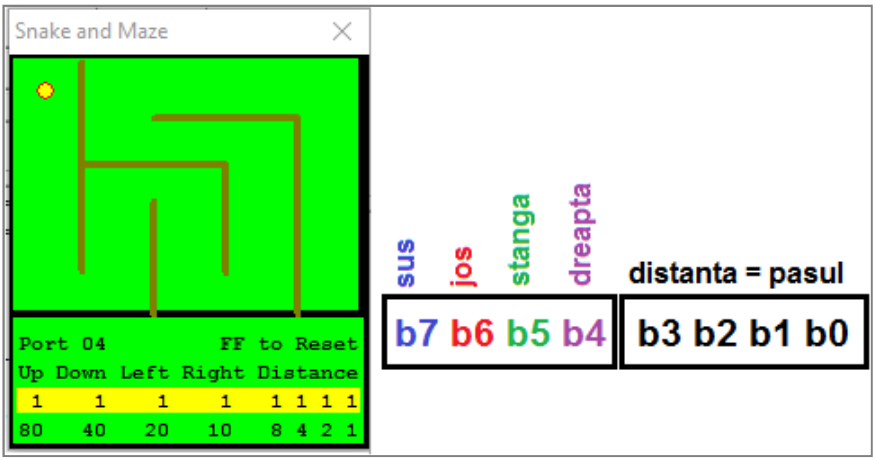


Figura 11.7. Şarpe în labirint

Exerciţii PRACTICE: (se vor rezolva în şablon)

(P) 1. Studiaţi secvenţa de mai jos, completaţi în spaţiile goale cu *instrucţiunile adecvate*, după cum se sugerează în comentarii, scrieţi apoi secvenţa în SMS și rulaţi:

```
clo                  ; închide ferestrele care nu sunt necesare
Start:  _____________; încărcaţi o valoare în registrul AL pt a reseta șarpele
        _____________; trimiteţi valoarea din registrul AL la portul corespunzător
        _____________; încărcaţi o valoare în registrul AL a.î. șarpele
                     ; să se deplaseze la stânga cu 7 unităţi de distanţă
        _____________; trimiteţi valoarea la portul potrivit pt a se reflecta în interfaţă
        _____________; încărcaţi o valoare în registrul AL a.î. șarpele
                     ; să se deplaseze în jos cu 15 unităţi de distanţă
        _____________; trimiteţi valoarea la portul potrivit pt a se reflecta în interfaţă
        _____________; trimiteţi din nou valoarea la port să repetaţi ultima mișcare
        _____________; încărcaţi o valoare în registrul AL a.î. șarpele
                     ; să se deplaseze la dreapta cu 14 unităţi de distanţă
        _____________; trimiteţi valoarea la portul potrivit pt a se reflecta în interfaţă
end                  ; programul se încheie
```

(P) 2. În aplicaţia *şarpe în labirint*, scrieţi în continuare secvenţe de instrucţiuni pentru a implementa mișcări și a dirija şarpele până la capătul labirintului:

(P) 3. O versiune optimă a programului ar trebui să folosească un tabel de date pt a stoca numerele care controlează şarpele. Rescrieţi mai jos noul program:

Observaţi modul cum se execută programul.

11.4. Folosirea întreruperilor

(EMU) 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 de calcul, utilizarea lor într-o aplicaţie realizându-se prin *apelul întreruperii*, cu instrucţiunea **INT**, având sintaxa **INT n**. O întrerupere poate avea mai multe servicii asociate, selectate 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 în *capitolul 12*, în care se vor studia întreruperile mai pe larg.

Aplicaţia, utilă pentru înţelegerea modului de utilizare al întreruperilor, foloseşte întreruperea ***INT 21h cu serviciul 09h***, care *afişează un şir de caractere*, aşa cum se poate urmări în Figura 11.8. Șirul se află la locaţia adresată de DS:DX şi trebuie să se termine cu caracterul ‚$'. Pentru mai multe detalii, consultaţi documentaţia, lista întreruperilor (Tutoriale).

Figura 11.8. Fereastra aplicaţiei *z02.asm*

Exerciţii PRACTICE: (se vor rezolva în șablon)

(P) 1. Studiaţi secvenţa din programul *z02.asm* din EMU și apoi completaţi în spaţiile goale cu *instrucţiunile adecvate*, după cum se sugerează în comentarii:

```
.stack 64h              ; segmentul de stivă de dimensiune 100 octeţi
.data                   ; segmentul de date
_____________________   ; scrieţi o directivă pentru a defini un şir de octeţi cu numele msg ,
                        ; în segmentul de date; acesta este iniţializat cu: codul Ascii ‚H’,
                        ; urmat de codul Ascii ‚e’, ş.a.m.d.(Figura 11.8); șirul se va termina
                        ; cu caracterul având cod Ascii 24h, adică ‚$’.
.code
   mov AX, @data        ; reg. segment DS este încărcat cu adresa de început a
   mov DS, AX           ; segmentului unde s-au definit datele (în segmentul de date)

_____________________   ; poziţionaţi DX pe zona din memorie unde începe msg
_____________________   ; încărcaţi în AH serviciul dorit pentru întrerupere (09h)
_____________________   ; apelaţi întreruperea pentru afișare mesaj pe ecran
.exit
```

(P) 2. a) 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 alt serviciu).

De exemplu, se poate selecta un serviciu care la fiecare apel de întrerupere să afişeze un singur caracter, nu un şir (terminat cu $ ca în cazul prezentat în aplicaţia de la punctul 1). a) Rescrieţi programul de la 1) folosind pentru afișare întreruperea int 10h cu serviciul 0Eh care afișează un singur caracter:

(P) b) Rescrieţi programul de la 1) folosind pentru afișare întreruperea int 10h cu serviciul 09h care afișează un singur caracter:

(P) c) Rescrieţi programul de la 1) folosind pentru afișare întreruperea int 21h cu serviciul 02h care afișează un singur caracter:

(EMU) convert_to_upper_case.asm

Aplicaţia foloseşte întreruperea ***INT 21h cu serviciul 0Ah*** pentru *preluarea unui șir de caractere de la tastatură*; apoi, în vederea afișării acestuia pe ecran folosind INT 21h cu serviciul 09h (care afişează un şir de caractere terminat cu ‚$'), sfârșitul acestuia va trebui marcat cu caracterul ‚$'. Aşa cum se poate urmări în Figura 11.9, aplicaţia va transforma fiecare literă mică în literă mare.

Zona de memorie în 3 etape diferite (iniţial- imediat după depunerea șirului în memorie, după marcarea sfârșitului acestuia cu caracterul ‚$' și după transformarea lui în șir de majuscule) este ilustrată în Figura 11.10. Pentru înţelegerea întreruperilor, consultaţi documentaţia, lista întreruperilor (Tutoriale) - analizaţi cu atenţie int 21h cu serviciul 0Ah.

Întreruperea *int 21h cu serviciul 0Ah* citește de la tastatură un întreg șir pe care îl depune în memorie la adresa dată de perechea de regiștrii DS:DX astfel: la o primă locaţie se depune dimensiunea bufferului (valoarea 20), la o a doua locaţie se depune efectiv numărul de caractere citite de la tastatură, iar abia de la a treia locaţie, în jos în memorie, se depune șirul efectiv; la început zona rezervată este de maxim 20 caractere efective tastabile de către utilizator, otetul de control și tasta Enter de la sfârșit sunt alte 2 locaţii suplimentare, iar valoarea 20 este cea de-a treia locaţie suplimentară. Astfel, variabila string de tip șir va ocupa în memorie o zonă de 23 locaţii, dar din care doar 20 vor fi posibile a fi preluate de la tastatură. Primele două locaţii mai poartă numele și de *octeţi de control*. Pentru ca șirul să poată fi afișat cu întreruperea 21h serviciul 09h, trebuie ca la sfârșitul lui, după ultimul caracter citit de la tastatură, să se insereze un caracter ‚$' ca marcaj de sfârșit al șirului. În vederea poziţionării vreunui registru ca pointer la începutul șirului în memorie, se reamintește că acesta începe efectiv în memorie la adresa DS:(DX+2).

Figura 11.9. Fereastra aplicaţiei *convert_to_upper_case.asm*

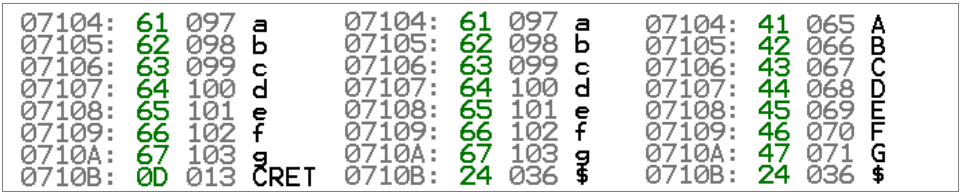


Figura 11.10. Zona de memorie ilustrând depunerea șirului în memorie pentru aplicaţia *convert_to_upper_case.asm*

Exerciţii PRACTICE: (se vor rezolva în șablon)

(P) 1. a) Studiaţi programul *convert_to_upper_case.asm* din EMU și apoi completaţi în spaţiile goale cu *instrucţiunile adecvate*, după cum se sugerează în comentarii:

```
org 100h
jmp start
string db 20, 22 dup('?')          ; se definește variabila string de tip șir de octeţi:
                                   ; primul octet va avea valoarea 20, al doilea va fi octetul
                                   ; de control în care după preluarea șirului de la tastatură
                                   ; se va depune lungimea efectivă a șirului, iar de la
                                   ; următoarea locaţie în jos se rezervă 21 octeţi
                                   ; neiniţializaţi (inclusiv Enter)
new_line db 0Dh,0Ah, '$'           ; șir de valori definit pt a trece la linie nouă, echivalent Enter
start:
        ; preluarea șirului de caractere de la tastatură cu int 21h, serviciul 0Ah
        _____________________      ; depuneţi în DX adresa de început a șirului string
        _____________________      ; depuneţi în AH serviciul întreruperii dorite
        _____________________      ; apelaţi întreruperea pentru a prelua tastele
        ; din octetul de control vom prelua (în AX) nr efectiv de taste introduse de utilizator
        ; în AL se va prelua informaţia din octetul de control, dar vom folosi reg pe 16 biţi
        _____________________      ; depuneţi în BX adresa de început a șirului string
        _____________________      ; depuneţi în AH valoarea 0
        _____________________      ; preluaţi în AL lungimea șirului, folosind adresare cu BX
```

b) Adăugaţi apoi și *instrucţiunea adecvată*, după cum se sugerează în comentarii pentru a plasa caracterul ‚$' la sfârșitul șirului:

```
; vom adăuga la BX lungimea șirului și astfel vom ajunge pe locaţia de după ultimul element citit
_____________________              ; scrieţi instrucţiunea corespunzătoare pt a aduna la BX pe AX
mov byte ptr [bx+2], '$'           ; depune în acea locaţie caracterul ‚$' - pentru a putea ulterior
                                   ; realiza afișarea șirului string cu int 21h, serviciul 09h
```

c) Adăugaţi apoi și *instrucţiunile adecvate*, după cum se sugerează în comentarii pentru a coborâ cursorul pe o linie nouă:

```
_____________________              ; depuneţi în DX adresa de început a șirului new_line
_____________________              ; depuneţi în AH serviciul întreruperii dorite
_____________________              ; apelaţi întreruperea pentru a afișa șirul respectiv pe ecran
```

d) Adăugaţi apoi și *instrucţiunile adecvate*, după cum se sugerează în comentarii pentru a plasa BX pe începutul zonei din memorie unde se găsesc caracterele preluate de la utilizator și pentru a depune în CX numărul efectiv al acestora:

```
_____________________              ; poziţionaţi BX pe începutul șirului string în memorie
; repetaţi raţionamentul de luare din octetul de control a numărului de caractere tastate de
; utilizator folosind de această dată registrul CX ca destinaţie
_____________________
```

```
_____          ; în CX va fi lungimea șirului preluată din octetul de control
jcxz null      ; este CX=0? Dacă DA, sare la eticheta null
add bx, 2      ; adună la BX valoarea 2, deci sare peste octeţii de control
```

e) Adăugaţi apoi și *instrucţiunile adecvate*, după cum se sugerează în comentarii pentru a transforma fiecare caracter literă mică preluat în majusculă:

```
upper_case:    ; etichetă – prima dată se verifică dacă elementul curent este o literă mare
        cmp byte ptr [bx], 'a'          ; deci dacă are codul Ascii >'a'
        _____; scrieţi o instrucţiune prin care dacă e mai mic, să sară la ok
        cmp byte ptr [bx], 'z'          ; și mai mic decât ‚z'
        _____; scrieţi o instrucţiune prin care dacă e mai mare, să sară la ok
        and byte ptr [bx], 11011111b ; analizaţi efectul acestei instrucţiuni
ok:  inc bx                     ; dacă s-a ajuns aici, trece la următorul element din șir
loop upper_case                 ; se reia bucla atât timp cât CX nu e încă nul
```

f) Analizaţi instrucţiunea loop de mai sus și specificaţi cu ce secvenţă de instrucţiuni este echivalentă aceasta: _____

_____

g) Adăugaţi apoi și *instrucţiunile adecvate*, după cum se sugerează în comentarii pentru a afișa pe ecran noul șir string având carcaterele literă mică transformate în majusculă:

```
_____          ; poziţionaţi DX pe adresa din memorie unde încep caracterele
_____          ; depuneţi în AH serviciul întreruperii dorite
_____          ; apelaţi întreruperea pentru a afișa șirul string pe ecran

; se va completa și cu o secvenţă prin care se așteaptă apăsarea unei taste (oricare)
_____          ; depuneţi în AH serviciul întreruperii dorite
_____  ; apelaţi întreruperea pt a prelua o tastă înainte de a ieși din program

null:  ret     ; eticheta null și instrucţiunea ret – de la return – revine în SO
```

2. 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 alt serviciu).

De exemplu, se poate selecta un serviciu care la fiecare apel de întrerupere să afişeze un singur caracter, nu un şir (terminat cu $ ca în cazul prezentat în aplicaţia de la punctul 1).

(P) a) Rescrieţi programul de la 1) folosind pentru afișare întreruperea int 10h cu serviciul 0Eh care afișează un singur caracter:

(P) b) Rescrieţi programul de la 1) folosind pentru afișare întreruperea int 10h cu serviciul 09h care afișează un singur caracter:

(P) c) Rescrieţi programul de la 1) folosind pentru afișare întreruperea int 21h cu serviciul 02h care afișează un singur caracter:

(P) 3. Modificaţi/rescrieţi programul astfel încât să preia două șiruri de la tastatură: primul se va afișa pe ecran nemodificat, dar cel de-al doilea va fi introdus de la tastatură cu majuscule și aplicaţia îl va transforma în litere mici.

(P) 4. Modificaţi/ rescrieţi programul astfel încât să afișeze pe ecran un mesaj sugestiv urmat de numărul de caractere / elemente ale șirului.

11.5. Scrierea datelor în memoria video

(EMU) „Hello, world" (din meniul examples)

O altă posibilitate de a afişa text pe ecran este prin ***scrierea datelor direct în memoria video***. Emulatorul foloseşte 8 pagini de memorie video, cuprinse între adresele ***0B8000h-0C0000h***. Ecranul emulatorului poate fi redimensionat, astfel încât să fie necesară mai puţină memorie pentru fiecare pagină. Folosind acest program se afişează pe ecranul simulatorului mesajul „Hello,World!" prin scriere direct în memoria video (Figura 11.11). O caracteristică a memoriei video este că primul octet are semnificaţie de cod Ascii al caracterului de afişat, iar octetul următor (consecutiv) reprezintă atributul lui de culoare, deci fiecare caracter poate fi controlat independent.

Atributul de culoare (descris în Figura 11.12) 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-roşu, 0111-gri deschis, 1000-gri închis, 1001-albastru deschis, ..., 1110-galben, 1111-alb.

Figura 11.11. Fereastra aplicaţiei *1_sample.asm*

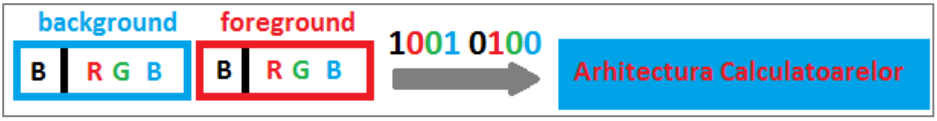


Figura 11.12. Codificarea culorii pentru atributul de culoare necesar aplicaţiei *1_sample.asm*

Exerciţii PRACTICE: (se vor rezolva în şablon)

(P) 1. a) Studiaţi programul *„Hello, world" (în director îl veţi găsi ca fișierul 1_sample.asm)* din EMU și apoi completaţi în spaţiile goale cu *instrucţiunile adecvate*, după cum se sugerează în comentarii:

```
mov ax, 3                ; mod text 80coloane x 25linii, 16 culori, 8 pagini (AH=0, AL=3)
_________                ; apelează întreruperea pentru setarea modului  video

mov ax, 1003h            ; anulează intermitenţa (blinking) şi permite cele 16 culori:
mov bx, 0                ; prin apelul întreruperii de mai jos cu AX=1003h
_________                ; apelează întreruperea pentru setarea modului  video
```

b) Adăugaţi apoi și *instrucţiunile adecvate*, după cum se sugerează în comentarii pentru a poziţiona (prin intermediul registrului AX) registrul segment DS pe zona de memorie video chiar de la început – adresa 0B800h:00000h = 0B8000h:

```
_____________________    ; scrieţi în AX valoarea segmentului pt a obţine adresa menţionată
_____________________    ; scrieţi în registrul segment DS valoarea din AX
```

c) Adăugaţi apoi și *instrucţiunile adecvate*, după cum se sugerează în comentarii pentru a scrie pe ecran mesajul "Hello, World!"

```
mov [02h], 'H'           ; afişează "Hello, World!"
mov [04h], 'e'           ; la adresă pară va fi octet cu cod Ascii de afişat
_____;_____;_____;_____;_____;_____;_____;_____;_____;_____;
mov [18h], '!'           ; la adresă impară va fi octet ca atribut de culoare
```

d) Adăugaţi apoi și *instrucţiunile adecvate*, după cum se sugerează în comentarii pentru a colora toate caracterele:

```
; colorează toate caracterele: în total sunt 12 caractere
_____________________   ; depuneţi în CX numărul de caractere de parcurs
_____________________   ; depuneţi în DI adresa (offset) pentru primul atribut de culoare

c:                      ; etichetă pentru implementarea buclei de colorare a fiecărui caracter
  _____________________ ; depuneţi în memorie la adresa dată de reg DI atributul coresp.
                        ; unei culori roşu aprins(1100) pe galben(1110) - pt caracterul curent
  _____________________ ; actualizaţi pointerul pt a sări la următorul atribut de culoare
  loop c                ; se reia bucla până când CX devine 0
mov ah, 0               ; se aşteaptă apăsarea unei taste (oricare)
int 16h
ret                     ; revenire în SO
```

e) Executaţi aplicaţia pas cu pas și urmăriţi efectul fiecărei instrucţiuni.

2. Modificaţi atributul de culoare; folosiţi pentru fond culoare roşu deschis, iar pentru scrierea caracterului culoarea verde:

_____

_____

(P) 3. a) Scrieţi un nou program, în care să inseraţi următoarele instrucţiuni; comentaţi efectul fiecăreia dar și al întregii secvenţe:

```
mov AX, 0B800h          ; AX= _____________
mov DS, AX              ; DS= _____________
mov CL, 'A'             ; CL= _____________
mov CH, 1101_1111b      ; CH= _____________
mov BX, 15Eh            ; BX= _____________
mov [BX], CX            ; în memorie, la adresa _____________ se depune CX
```

Rulaţi *pas cu pas*, explicaţi codul şi specificaţi efectul programului.

b) Modificaţi secvenţa de la a) astfel încât să se afişeze 3 caractere B de culori diferite: unul roşu, unul verde, unul albastru, toate pe fond gri:

(P) 4. Modificaţi programul astfel încât să apară pe ecran mesajul „Arhitectura CALCULATOARELOR”, fiecare caracter colorat cu o alta culoare (aleator):

(P) 5. Scrieţi un program prin care textul să apară pe linii diferite (fiecare caracter sau fiecare cuvânt); modificaţi şi culoarea, a.î. să fie diferită de la un rând la altul:

(P) 6. Analizaţi instrucţiunea *loop* din Help și testaţi exemplul propus acolo.

11.6. Citirea unor date de la tastatură

(SMS) input.asm

În cazul simulatorului SMS, ***portul de intrare 0*** este legat la tastatură. Programul prezentat în continuare foloseşte instrucţiunile ***clo***, ***in***, ***cmp***, ***jnz***, ***end***. Când se folosește instrucţiunea ***in*** (cu sintaxa *in AdrPort*), simulatorul aşteaptă apăsarea unei taste şi apoi copiază codul Ascii al tastei respective în registrul AL.
Instrucţiunea ***cmp*** *al,0D* – compară conţinutul registrului AL cu codul Ascii al tastei Enter (codul Ascii al tastei Enter este 0Dh). Instrucţiunea ***cmp al, bl***; funcţionează în felul următor: procesorul calculează o scădere de fapt: ***AL-BL*** şi în funcţie de rezultat setează flagurile (indicatorii): Z dacă rezultatul este zero, S dacă rezultatul este negativ. Deci flag-ul ZF=1 dacă AL=BL, SF=1 dacă BL este mai mare decât AL, iar dacă AL este mai mare decât BL nu se setează nici unul. Aici trebuie făcută precizarea că trebuie avută o grijă deosebită când ne referim la relaţii de tipul „mai mic” sau „mai mare”, întrucât:

- dacă se folosește ***ja*** sau ***jb*** numerele vor fi interpretate în ***convenţia fără semn***, iar
- dacă se folosește ***jl*** sau ***jg*** acestea vor fi interpretate în ***convenţia cu semn***.

Astfel, spunem despre numărul 200 că este „above” 199, și nu „greater than” 199.

Instrucţiunea de scădere nu afectează / nu modifică conţinutul regiştrilor AL, BL. Aceleași numere, interpretate ca numere fără semn sunt -56 și respectiv -57, spunând despre ele că sunt: -56 (echivalentul lui 200) este „greater than” -57 (echivalentul lui 199). Unde contează aceste interpretări? Un exemplu relevant ar fi atunci când dorim să găsim maximul dintr-un șir de numere cuprinse între 0 și 200. Toate valorile care sunt

> 127 se vor scrie ca având primul bit în 1 și dacă nu avem grijă la folosirea *ja* și *jb*, ele vor fi tratate ca numere negative (deci dacă folosim *jl* sau *jg*).

JNZ et – instrucţiunea JNZ provine de la *Jump if Not ZeroFlag*, adică salt dacă flagul ZF=0. Programul va face un salt la adresa marcată de eticheta et. O instrucţiune asemănătoare este JZ, adică *Jump if Zero*, în cazul în care flagul ZF=1. În acest program, flag-urile sunt setate de instrucţiunea *cmp*. Instrucţiunile aritmetice setează de asemenea flag-urile. Atât timp cât nu se tastează Enter, instrucţiunea *cmp* nu va modifica flagul ZF, acesta rămânând 0, deci se va efectua saltul la eticheta *et*. La simulator, flagul ZF poate fi urmărit împreună cu flagurile IF, SF, OF în registrul SR, la secţiunea regiştrilor speciali.

Figura 11.13. Citirea unor date de la tastatură

mov [c0],al – va copia AL la adresa [C0]. Unitatea de afişare (***VDU – Video Display Unit***) este implementată în SMS la adresele [C0]...[FF], oferind un afişaj cu 4 linii şi 16 coloane. Adresa [C0] corespunde cu colţul din stânga sus al ecranului.

mov [bl],al – copiază AL la adresa dată de BL. Registrul BL poate fi făcut pointer la prima poziţie în ecranul VDU prin instrucţiunea *mov bl,c0*; apoi, se poate muta poziţie cu poziţie prin incrementare, folosind instrucţiunea *inc bl*.

Figura 11.14. Unitatea de afişare VDU

Exerciţii PRACTICE: (se vor rezolva în şablon)

(P) 1. a) Studiaţi secvenţa de mai jos, completaţi în spaţiile goale cu *instrucţiunile adecvate*, după cum se sugerează în comentarii, scrieţi apoi secvenţa în SMS și rulaţi:

```
clo                 ; închide ferestrele care nu sunt necesare
et:
        _____       ; aşteaptă preluarea unei taste – o depune în AL
        _____       ; s-a tastat Enter? (cod Ascii 0Dh)
        _____       ; dacă nu - salt înapoi la eticheta et
end                 ; dacă DA - programul ajunge aici și se termină
```

b) Modificaţi în loc de tasta Enter să fie tasta Esc (cod Ascii 1Bh) și re-verificaţi:

_____

(P) 2. Modificaţi programul *input.asm* astfel încât să se afişeze fiecare caracter preluat de la tastatură în colţul din stânga sus al VDU:
Sugestie: puteţi realiza aceasta prin copierea codului caracterului la adresa [C0];

(P) 3. Modificaţi / rescrieţi programul folosind BL ca pointer la adresa [C0]. Incrementaţi BL după fiecare tastă apăsată pentru a vedea textul pe măsură ce îl tastaţi:

(P) 4. Modificaţi / rescrieţi programul astfel încât să se depună textul introdus în zona RAM pe măsură ce se tastează caracterele. Abia la apăsarea tastei Enter, să se afișeze pe VDU textul păstrat în RAM:

(P) 5. Modificaţi / rescrieţi programul astfel încât să preluaţi un text şi să-l păstraţi în RAM. La apăsarea tastei Enter afişaţi în ordine inversă pe VDU textul memorat (folosind stiva este mai uşor):

(EMU) keybrd.asm

Programul ***keybrd.asm*** din EMU foloseşte instrucţiunile ***int***, ***cmp***, ***jz***, ***end*** 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, așa cum se poate urmări în Figura 11.15.


```
name "keybrd"

org     100h
  ; INTRAREA DE LA TASTATURA
mov dx, offset msg
mov ah, 9
int 21h          ; afiseaza mesaj de intampinare
;--------------------------
wait_for_key:    ; bucla infinita pt preluarea si
        mov      ah, 1    ;afisarea tastelor
        int      16h
        jz       wait_for_key

mov     ah, 0    ; preia tasta de la tastatura,
int     16h              ;stergand-o din buffer
mov     ah, 0eh  ; afiseaza pe ecran/ printeaza tasta
int     10h

cmp     al, 1bh  ; verifica daca s-a apasat "ESC"
jz      exit                          pt iesire
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
```

Figura 11.15. Fereastra aplicaţiei *keybrd.asm*

Instrucţiunea *JZ et;* unde JZ vine de la *Jump if Zero*, asigură salt dacă flagul Z este setat („Jump if ZeroFlag is set"). Programul va face un salt la adresa marcată de eticheta *et*. O instrucţiune asemănătoare, așa cum am prezentat la programul anterior este JNZ, adică *Jump if Not Zero*, în cazul în care flagul Z nu este setat. În acest program, instrucţiunea *cmp* setează flag-urile.

Exerciţii PRACTICE: (se vor rezolva în şablon)

1. a) (P) Studiaţi programul *keybrd.asm* din EMU și apoi completaţi în spaţiile goale cu *instrucţiunile adecvate*, după cum se sugerează în comentarii:

```
org  100h
_____________________     ; afişaţi mesaj de primire denumit msg cu int 21h, serviciul 09h
_____________________
_____________________
```

```
wait_for_key:          ; buclă infinită pentru preluarea şi afişarea tastelor
      mov ah, 1        ; verifică dacă există tastă nepreluată din buffer
      int 16h          ; apelul întreruperii pentru tastatură
      jz wait_for_key  ; la apăsarea unei taste iese din această buclă şi continuă
      mov AH, 0        ; preia tasta de la tastatură, ştergând-o din buffer
      int 16h
      mov AH, 0Eh      ; afişează pe ecran/printează tasta
      int 10h
_____                  ; verificaţi dacă s-a apăsat 'ESC'
      jz exit          ; dacă DA, se va ieşi din program
      jmp wait_for_key
exit:       ret
msg  db "Type anything...", 0Dh,0Ah          ; mesaj introductiv definit cin mem cu db
   db "[Enter] - carriage return.", 0Dh,0Ah  ; este de fapt un şir de octeţi
   db "[Ctrl]+[Enter] - line feed.", 0Dh,0Ah ; terminat cu caracterul $, marcaj sfârşit mesaj
   db "You may hear a beep", 0Dh,0Ah         ; 0Dh,0Ah - echivalent cu a da ENTER
   db "   when buffer is overflown.", 0Dh,0Ah
   db "Press Esc to exit.", 0Dh,0Ah, "$"
end
```

2. Modificaţi secvenţa de mai sus astfel încât după afișarea mesajului, cursorul să coboare 2 rânduri mai jos: _____

3 (P). Scrieţi pe ecran mesajul "azi e joi" introducând fiecare cuvânt pe o linie nouă.
4. Rulaţi programul *keybrd.asm* setând viteza maximă la execuţie (întârziere de 0 msecunde) dar şi la viteză mai mică (de exemplu întârziere de 100 msecunde) din cursorul pentru timp. Tastati apoi cât mai repede cu putinţă; introduceţi mai mult de 16 caractere. Ce observaţi?_____

11.7. Salturi în program

(EMU) thermometer.asm

Aplicaţia foloseşte un termometru care trebuie menţinut la o temperatură cuprinsă între 60° şi 80° cu ajutorul unui radiator de căldură (Figura 11.16). Se folosesc 2 porturi de intrare: de date la ***adresa 125*** şi de control la ***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 şi

până ce temperatura nu ajunge la o valoare mai mică de 60° nu se mai porneşte. Aplicaţia e utilă pentru înţelegerea *salturilor în program*. Se execută cu Run.

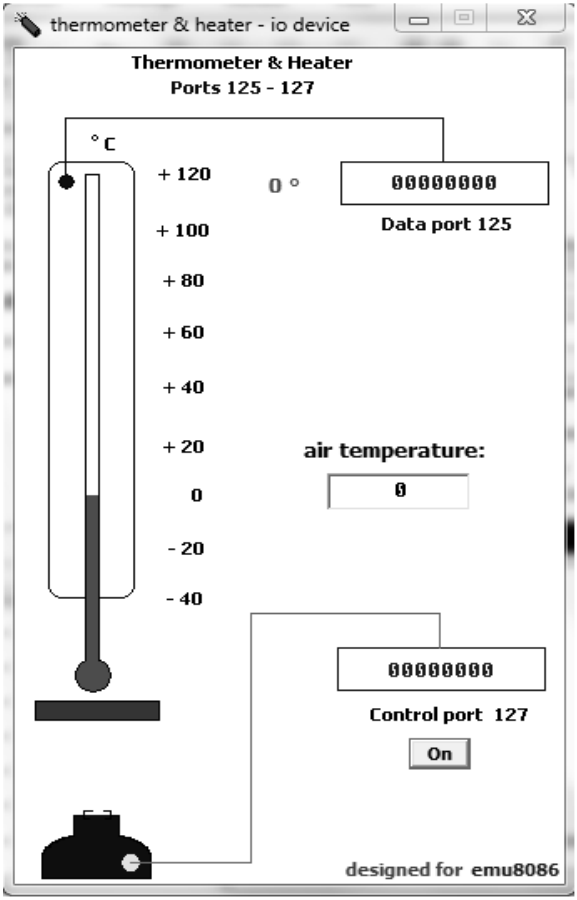


Figura 11.16. Fereastra aplicaţiei *thermometer.asm*

Exerciţii PRACTICE: (se vor rezolva în şablon)

1 a) (P) Studiaţi programul *thermometer.asm* din EMU și apoi completaţi în spaţiile goale cu *instrucţiunile adecvate*, după cum se sugerează în comentarii:

```
mov AX, CS          ; setează adresa segmentului de date înspre segmentul de cod;
mov DS, AX
start:              ; eticheta start:
        _________   ; preluaţi valoarea temperaturii de pe portul de date cu adresa 125
        _________   ; comparaţi valoarea temperaturii curente cu 60°
        jl  low     ; dacă temperatura e mai mică de  60°, sare la eticheta LOW
```

```
        _____       ; comparaţi valoarea temperaturii curente cu 80°
        jle  ok      ; dacă temperatura este ≤ cu 80°, sare la eticheta OK
        jg   high    ; dacă temp. este > de 80°, sare la eticheta HIGH
low:    _____       ; înscrieţi în registrul AL valoarea 1
        _____       ; puneţi radiatorul de căldura pe "on", trimiţând 1 pe port
        jmp ok       ; salt necondiţionat la eticheta OK
high:   _____       ; înscrieţi în registrul AL valoarea 0
        _____       ; puneţi radiatorul de căldura pe "off", trimiţând 0 pe port
ok:     jmp start            ; salt înapoi la start:, deci bucla se reia la infinit.
```

1 (P). Modificaţi limitele între care se încearcă păstrarea temp. la 70° şi 90°.
2. Termostatul funcţionează corect? Imediat la depăşirea valorii maxime se şi opreşte radiatorul, sau există o oarecare întârziere (latenţă)? Cum explicaţi?
3 (P). Modificaţi valoarea ce se încarcă în registrul AL la eticheta *low* să fie 2. Salvaţi şi rulaţi din nou. Ce observaţi ? Cum explicaţi ?
4 (P). Modificaţi programul *LED_display_test.asm* (din secţ. 11.4) a.î. valoarea să se incrementeze până ajunge la 50, apoi să se decrementeze până la 40 şi tot aşa (deci să încerce să menţină o valoare constantă între anumite limite – ca un termostat).

11.8. Folosirea subrutinelor

(EMU) count_key_presses.asm

Aplicaţia foloseşte ***întreruperea 16h, serviciul 0h*** pentru ***preluarea de taste de la tastatură până când utilizatorul apasă tasta ESC*** (cod Ascii 27 sau 1Bh). Apoi, apelează o subrutină cu ajutorul căreia afișează pe ecran numărul de taste apăsate de utilizator (Figura 11.17). Fiecare din cele 2 acţiuni principale menţionate este precedată de afișarea pe ecran a unui mesaj, folosind întreruperea int 21h cu serviciul 09h. Aplicaţia folosește 3 tipuri de întreruperi:

- int 16h, serviciul 0h pentru preluare taste,
- int 10h cu serviciul 0Eh (teletype) pentru afișare caracter pe ecran și
- int 21h cu serviciul 09h pentru afișare mesaj pe ecran.

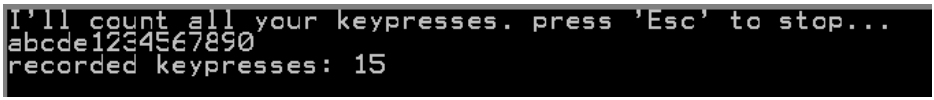


Figura 11.17. Fereastra aplicaţiei *count_key_presses.asm*

Exerciţii PRACTICE: (se vor rezolva în șablon)

(P) 1. a) Studiaţi programul *count_key_presses.asm* din EMU și apoi completaţi în spaţiile goale cu *instrucţiunile adecvate*, după cum se sugerează în comentarii:

```
org  100h
_____________________          ; afișează un mesaj așa ca în Figura 11.17
_____________________
int 21h
```

b) Adăugaţi apoi și *instrucţiunile adecvate*, după cum se sugerează în comentarii pentru a aștepta apăsarea unei taste

```
xor bx, bx                     ; depune 0 în registrul BX
wait: _____________________    ; așteaptă apăsarea unei taste
        _______________
        _______________        ; dacă tasta apăsată este 'ESC' atunci iese afară din program
    je stop                    ; va ieși din buclă doar dacă se îndeplinește condiţia de mai sus
    _______________            ; afișează pe ecran tasta preluată
    int 10h
    inc bx                     ; incrementează registrul BX – aici va contoriza nr. de apăsări
    jmp wait                   ; revine automat în buclă
```

c) Adăugaţi apoi și *instrucţiunile adecvate*, după cum se sugerează în comentarii pentru a afișa pe ecran cel de-al doilea mesaj:

```
stop: _____________________    ; afișează mesajul msg2
      ___________________
      ___________________
```

d) Adăugaţi apoi și *instrucţiunile adecvate*, după cum se sugerează în comentarii pentru a apela subrutina de afișare:

```
mov ax, bx
_____________________          ; apelează subrutina de afișare
```

e) Adăugaţi apoi și *instrucţiunile adecvate*, după cum se sugerează în comentarii pentru a aștepta apăsarea unei taste înainte de a ieși din program:

```
_____________________          ; așteaptă apăsarea unei taste înainte de a ieși din program
_____________________
ret                            ; iese și revine în S.O.
```

f) Analizaţi modul cum au fost scrise subrutinele de mai jos:

```
msg db "I'll count all your keypresses. press 'Esc' to stop...", 0Dh,0Ah, "$"
msg2 db 0Dh,0Ah, "recorded keypresses: $"
print_ax proc      ; subrutină pt afișarea unui număr în zecimal, având mai mult de 1 digit
cmp ax, 0                      ; dacă nr de afișat e 0, atunci se intră în secvenţa de mai jos
jne print_ax_r                 ; altfel, sare la eticheta print_ax_r
   push ax
   mov al, '0'                 ; se va afișa pe ecran 0
   mov ah, 0Eh                 ; folosind modul teletype
   int 10h
```

```
   pop ax
   ret                  ; această instrucţiune ret va asigura ieșirea din subrutină
; dacă ajunge aici, înseamnă ca există cel puţin un digit diferit de 0 care trebuie afișat,
; și se va folosi secvenţa de mai jos
print_ax_r:  pusha      ; salvează toţi regiștrii pe stivă – se realizează atâtea depuneri pe
                        ; stivă câţi digiţi are nr. de afișat; în DX se va vedea acest număr
                        ; de ex., dacă nr de afișat este 123, pe stivă va ajunge primul 3,
                        ; apoi 2, apoi 1 și în ordine inversă se vor prelua de pe stivă
                        ; și se va afișa pe ecran: 123
   mov dx, 0            ; se pregătește perechea DX:AX pt împărţire la 10 (din reg BX)
   cmp ax, 0            ; se compară câtul cu 0,
   je pn_done           ; dacă se obţine egalitate, am terminat
   mov bx, 10           ; altfel, perechea DX:AX se împarte la 10 (din reg BX)
   div bx
   call print_ax_r  ; se scrie cu call, pt a se reveni aici după fiec. instrucţiune popa  de la sfârșit
   mov ax, dx           ; aici se revine după popa, și va muta restul în AX
   add al, 30h          ; îl formează ca și cod Ascii
   mov ah, 0eh          ; îl afișează pe ecran
   int 10h
   jmp pn_done
pn_done:  popa    ; se refac de pe stivă toţi regiștrii - cum au fost înainte de apelul subrutinei;
   ret            ; după fiecare execuţie popa, se reîntoarce după call print_ax_r (la apelant)
endp
```

g) Analizaţi programul, rulând cu Run și urmăriţi modul de execuţie raportat la intervenţia utilizatorului. Specificaţi în ce registru va returna programul numărul de apăsări de tastă. Modificaţi mesajele după cum doriţi.

h) Analizaţi modul cum a fost scrisă procedura de afișare pe ecran a numărului de taste apăsate. Analizaţi fiecare instrucţiune din rutina de afișare a numărului pe ecran, comentând efectul fiecăreia și descrieţi mai jos algoritmul utilizat:

(P) 2. Modificaţi aplicaţia astfel încât programul să înceapă să numere după apăsarea tastei a) Enter; b) Space în locul celei ESC.

a) **b)**

(P) 3. Modificaţi aplicaţia astfel încât secvenţa de afișare a mesajului să fie încadrată într-o subrutină. Apelaţi această subrutină în cele 2 locuri din program pentru a nu altera execuţia aplicaţiei.

(P) 4. In cazul în care se presupune că utilizatorul nu ar apăsa niciodată mai mult de 9 taste, puteţi înlocui secvenţa de afișare cu o alta mult mai simplă (și mai scurtă) care să afișeze un singur digit pe ecran? Explicaţi mai jos:

Modul TEXT și modul GRAFIC:

Ecranul poate fi folosit în general în unul din cele 2 moduri principale:
a) ***mod de lucru grafic*** – în care ecranul e organizat ca o matrice de puncte independente numite pixeli, are dimensiunea 640x480 și pot fi afișate atât caractere (de diferite dimensiuni sau fonturi) cât și alte obiecte grafice (puncte, linii, cercuri, etc);
b) ***mod de lucru text*** – în care ecranul este împărţit în zone de dimensiunea 8x8 pixeli (ca niște căsuţe sau matrici) în care se poate afișa un singur caracter.

11.9. Folosirea ecranului în mod TEXT

(EMU) colors.asm

În toate aplicaţiile de până acum în care s-a afișat ceva pe ecran, s-a folosit modul text, chiar dacă nu s-a specificat acest lucru în mod explicit; folosind acest mod, caracterele sunt afişate pe linii, de la stânga spre dreapta ecranului şi de sus în jos.

Poziţia curentă de afişare este indicată de *cursor*, care dacă ajunge pe ultima poziţie/ coloană de pe un rând, trece pe rândul următor, pe prima coloană. În mod text, pentru fiecare caracter se păstrează în memorie două informaţii: codul Ascii al caracterului care se afişează şi culorile utilizate pentru desenarea caracterului şi respectiv a fondului pe care se face afişarea.

Programul colors din EMU demonstrează cel mai bine lucrul cu ecranul în mod text.

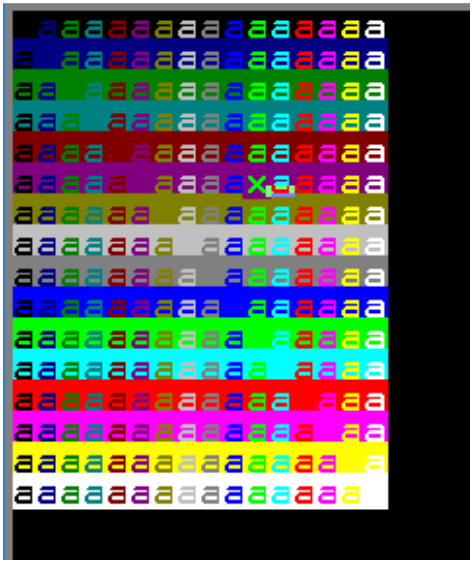


Figura 11.17. Fereastra aplicaţiei *colors.asm*

Exerciţii PRACTICE: (se vor rezolva în șablon)

(P) 1. a) Studiaţi programul *colors.asm* din EMU și apoi completaţi în spaţiile goale cu *instrucţiunile adecvate*, după cum se sugerează în comentarii:

```
org  100h
; setează modul video: ecranul - în mod text, cu 80 coloane și 25 linii, 16 culori, 8 pagini
_____
_____
_____
_____          ; dezactivare serviciu blinking – lumină intermitentă (activat în sisteme DOS)
_____
```

b) Adăugaţi apoi și *instrucţiunile adecvate*, după cum se sugerează în comentarii pentru a respecta comentariile:

```
; partea de iniţializare a unor valori, care se vor modifica în cadrul unei bucle
_____          ; coloana 0 - la început de program
_____          ; linia 0 - la început de program
_____          ; atributul de culoare - la început de program
jmp next_char  ; salt automat la next_char
nrow:  ; etichetă la care se ajunge când trebuie trecut cursorul pe linia următoare
       _____          ; se incrementează linia curentă
       _____          ; se compară dacă s-a ajuns la linia 16
       je stop_print  ; dacă DA, sare la eticheta stop_print
       _____          ; altfel, iniţializează cursorul la începutul liniei – coloana 0
```

```
nchar:    ; etichetă la care se ajunge când trebuie trecut cursorul pe coloana urm.
          mov ah, 02h       ; serviciul de setare a poziţiei cursorului pe ecran
          int 10h
          _____             ; caracterul de afișat pe ecran se depune în registrul AL
          mov bh, 0         ; pagina video 0
          _____             ; numărul de afișări ale caracterului este 1
          _____             ; serviciul de afișare pe ecran
          _____             ; apel întrerupere
          _____             ; se incrementează atributul de culoare
          _____             ; se incrementează coloana
          _____             ; se verifică dacă s-a ajuns pe coloana 16
       je nrow              ; dacă DA, sare la eticheta nrow
       jmp nchar            ; altfel, sare le eticheta nchar
```

c) Adăugaţi apoi și *instrucţiunile adecvate*, după cum se sugerează în comentarii pentru a ....

```
stop_print:
          ; setare poziţie cursor: DH - specifică linia, iar DL - coloana
          _____             ; coloana 10
          _____             ; linia 5
          _____             ; serviciul de poziţionare cursor
          _____
; folosind modul teletype, se afișează pe ecran un caracter folosind atributul de culoare
          _____             ; se dorește afișarea lui ‚x'
          _____             ; serviciul teletype de afișare caracter la poziţia curentă cursor
          _____
```

d) Adăugaţi apoi și *instrucţiunile adecvate*, după cum se sugerează în comentarii pentru a ....

```
; se așteaptă apăsarea unei taste înainte de a ieși din program:
_____               ; serviciul de preluare tastă (oricare) de la utilizator
_____
ret
```

(P) 2. a) Modificaţi programul astfel încât să se afișeze maxim 10 caractere pe o linie; b) Afișaţi 'o' în loc de 'a':

(P) 3. Modificaţi programul astfel încât la sfârșitul secvenţei să se insereze 3 caractere 'x', nu doar unul singur:

(P) 4. Repetaţi cerinţa de la punctul 3, dar să se insereze 20 caractere 'x', nu doar unul singur sau 3. Încercaţi să realizaţi această operaţie folosind o buclă:

11.10. Folosirea ecranului în mod GRAFIC

(EMU) 0_sample_vga_graphics.asm

Programul foloseste ecranul în mod grafic; efectul implementat este de a desena un dreptunghi folosind modul vga.

Exerciţii PRACTICE: (se vor rezolva în șablon)

(P) 1. a) Studiaţi programul *0_sample_vga_graphics.asm* din EMU și apoi completaţi în spaţiile goale cu *instrucţiunile adecvate*, după cum se sugerează în comentarii:

```
org  100h
jmp code          ; salt automat peste zona de definire a datelor
w equ 10          ; constantă pentru dimensiunea dreptunghiului: lăţime – 10 pixeli
h equ 5           ; constantă pentru dimensiunea dreptunghiului: înălţime – 5 pixeli

; se setează modul video: cu 13h este modul grafic, 320x200 pixeli, 256 culori, o pagină.
code:   _________       ; serviciul de setare mod video
        _________       ; se alege modul grafic
        _________

   mov cx, 100+w  ; coloana
   mov dx, 20     ; linia
   mov al, 15     ; culoarea albă: 0Fh = 0000 1111b – fond negru, scris alb
et1:              ; etichetă pt buclarea serviciului de desenare al unui singur pixel:
   mov ah, 0ch    ; în AL se specifică ce culoare se dorește, în CX coloana, iar în DX linia
   int 10h
   _____________  ; se decrementează coloana
   cmp cx, 100    ; se compară dacă s-a ajuns la coloana 100
   jae u1         ; dacă încă NU, sare la eticheta et1 și continuă
   mov cx, 100+w  ; coloana
   _____________  ; repoziţionare pe linia 20
   mov al, 15     ; culoarea albă: 0Fh = 0000 1111b – fond negru, scris alb
```

```
et2:                    ; etichetă pt buclarea serviciului de desenare al unui singur pixel:
   mov ah, 0ch
   int 10h

   dec cx
   _________            ; se compară dacă s-a ajuns la coloana 100
   ja et2               ; dacă încă NU, sare la eticheta et2 și continuă

   mov cx, 100          ; _____________________________________________________
   mov dx, 20+h         ; _____________________________________________________
   mov al, 15           ; _____________________________________________________
et3:
   mov ah, 0ch          ; _____________________________________________________
   int 10h

   dec dx
   cmp dx, 20
   ja et3

   mov cx, 100+w        ; _____________________________________________________
   mov dx, 20+h         ; _____________________________________________________
   mov al, 15           ; _____________________________________________________
et4:
   mov ah, 0ch          ; _____________________________________________________
   int 10h

   dec dx               ; _____________________________________________________
   cmp dx, 20           ; _____________________________________________________
   ja et4               ; _____________________________________________________

; așteaptă apăsarea unei taste înainte de a ieși
___________             ; serviciul pt preluare tastă de la utilizator
  int 16h

; se revine în modul text
  mov ah,0
  _________________     ; modul text cu 80x25 caractere, 16 culori, 8 pagini
  int 10h
ret
```

Figura 11.18. Fereastra aplicaţiei *0_sample_vga_graphics.asm*

1. Analizaţi programul și apoi modificaţi astfel încât să se desenze un dreptunghi a) de 2 ori mai mare; b) cu lăţimea de 20 pixeli și înălţimea de 30 pixeli; c) la alte coordonate: pornind din mijlocul ecranului.

2. Inlocuiţi în program valorile prestabilite cu 2 constante definite la început de program (acestea fiind iniţializate cu valorile respective); astfel, veţi putea apoi modifica aceste valori mult mai ușor și rapid. a) rerulaţi punctul 1 cu alte valori; b) modificaţi valorile astfel încât figura geometrică să fie desenată exact în colţul dreapta jos; c) repetaţi punctul b) dar pentru colţul stânga sus.

3. Modificaţi culoarea de afișare a pixelilor: specificaţi câte o culoare diferită a) pentru fiecare latură a dreptunghiului; b) pentru fiecare pixel desenat.

4. Scrieţi în plus o secvenţă prin care să umpleţi dreptunghiul cu o culoare diferită de cea a muchiilor.

5. Încercaţi să desenaţi o altă figură geometrică: a) un romb; b) un triunghi.

6. Transpuneţi problema chiar și sub formă simplificată în modul text.

Paint.asm

Următorul program nu se găsește nici în EMU, nici în SMS, astfel că se va scrie în cadrul unui șablon de program gol (iniţial de tip COM) din EMU:

Exerciţii PRACTICE: (se vor rezolva în șablon)

(P) 1. Studiaţi programul și apoi scrieţi-l în EMU, comentaţi și rulaţi pas cu pas pentru a observa efectul fiecărei instrucţiuni. Modificaţi astfel încât a) să nu se schimbe atributul de culoare în aplicaţie; b) să se schimbe abia o dată la 3 apăsări;

```
.model small        ; se definește un program de tip .exe, folosind modelul de memorie small
.stack 100h         ; zona de stivă de 256 octeţi
.data               ; zona de date
.code               ; zona de cod
mov ax,@data
mov ds,ax           ; se poziţionează reg segment DS pe zona unde s-au definit datele în memorie
mov ah,0            ;
mov al,13h          ;
int 10h             ; _____________________________________________________________
```

```
mov al,1            ; inițializarea culorii la început
repeta:             ; pentru fiecare pixel afișat se repetă această buclă
        push ax     ; reg AX are rol dublu (se fol. și în cadrul apelului întreruperilor)
        mov ax,3    ; serviciul pentru _____________________________________________
        int 33h
        pop ax      ; se reface AX de pe stivă, pentru a nu se modifica atributul de culoare curent
        mov bh,bl           ; se salvează starea butoanelor
        and bl,1            ; se verifică dacă s-a apăsat butonul stâng al mouse-ului
        jz next             ; dacă DA, sare la eticheta next
        inc al              ; altfel, incrementează atributul de culoare (schimbă culoarea)
next:                       ; etichetă pentru butonul drept al mouse-ului
        mov bl,bh           ; se reface starea butoanelor
        and bl,2            ; se verifică dacă s-a apăsat butonul drept al mouse-ului
        jnz gata            ; dacă DA, _____________________________________
        mov ah,0ch          ; dacă NU, _____________________________________
        mov bh,0
        int 10h
jmp repeta
gata:   mov ah,0            ; _____________________________________________________
        mov al,3
        int 10h
mov ax,4ch                  ; ieșire din program
int 21h
ret
```

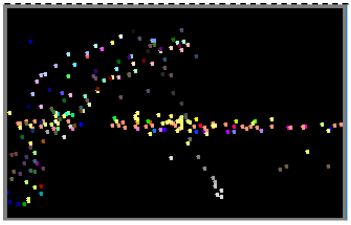


Figura 11.19. Fereastra aplicaţiei *paint.asm*

(EMU) mouse2.asm

Aplicaţia Mouse2.asm din EMU urmărește mouse-ul; dacă se apasă butonul stâng, permite desenarea. Comentaţi programul linie cu linie și completaţi liniile lipsă comparând cu codul sursă original din EMU:

```
org 100h
jmp start           ; salt automat peste zona de definire a datelor
oldX dw -1          ; variabilă definită pentru păstrarea poziţiei cursorului pe coloană
oldY dw 0           ; variabilă definită pentru păstrarea poziţiei cursorului pe linie
start:              ; eticheta de început a programului
        __________  ; setare mod video
        _________   ; modul grafic, 320x200 pixeli, 256 culori, o pagină.
```

```
            _________
            mov ax, 0 ; reset mouse
            int 33h
            cmp ax, 0
verif_ butonMouse:
            _________
            _________
            shr cx, 1
            cmp bx, 1
        jne xor_cursor:
            mov al, 1010b
        jmp deseneaza
xor_cursor:
            cmp oldX, -1
        je etich
            push cx
            push dx
            mov cx, oldX
            mov dx, oldY
            _________
            _________
            xor al, 1111b
            _________
            _________
            pop dx
            pop cx
etich:      _________
            _________
            xor al, 1111b
            mov oldX, cx
            mov oldY, dx
deseneaza:
            _________
_________
verif:      mov dl, 255
            _________
            _________
            cmp al, 27
        jne verif_ butonMouse
stop:   ...
```

Exerciţii: 1. Analizaţi programul întreg așa cum apare în EMU și explicaţi secvenţele de instrucţiuni care apar de la eticheta *stop* în jos. Câte întreruperi sunt apelate? Ce operaţie realizează acestea? Explicaţi.

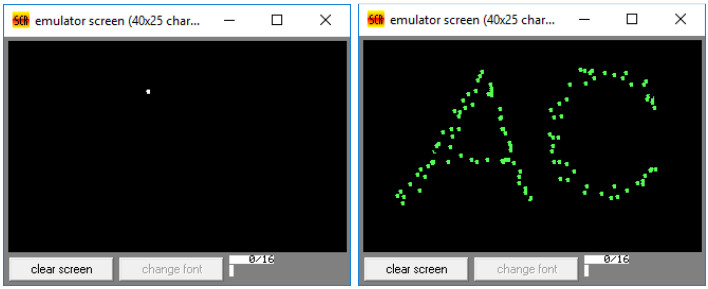


Figura 11.20. Fereastra aplicaţiei *mouse2.asm* la execuţia simplă (stânga), respectiv la execuţia cu butonul stâng apăsat (dreapta)

11.11. Afişarea rezultatelor în EMU în binar

(EMU) 2_sample.asm

Așa cum am prezentat în *Capitolul 6*, programul ***2_sample.asm*** foloseşte instrucţiunile *mov*, *add*, *sub* și poate ajuta la înţelegerea operaţiilor aritmetice și logice întrucât prima parte a programului prezintă astfel de operaţii. Totuși, programul este mult mai complex decât am prezentat în *Capitolul 6*, deoarece acesta include și o parte de afișare a rezultatului obţinut și această afișare este realizată prin intermediul întreruperilor. Folosirea întreruperilor a fost introdusă deja în capitolele anterioare (în secţiunea 11.5); aici folosim întreruperi pentru afișarea valorii stocate într-un registru în binar, pe ecran; Scrieţi următorul program așa cum apare el și explicaţi fiecare instrucţiune:

```
org 100h
mov bl, 74h
mov cx, 8
print: mov ah, 2
      mov dl,'0'
      test bl, 10000000b
      jz zero
      mov dl,'1'
zero:  int 21h
      shl bl,1
loop print
mov dl, 'b'
int 21h
ret
```

a) Comentaţi fiecare instrucţiune; b) Modificaţi (inseraţi cod suplimentar) a. î. să fie afișată valoarea din registrul BX; Inseraţi (cod suplimentar) a. î. să fie afișat un mesaj la început: „în caz că doriţi numărul în interpretarea cu semn, acesta este un număr: ” și scrieţi voi POZITIV/ NEGATIV în funcţie de valoarea primului bit (interactiv); c) Analizaţi toate posibilele situaţii de conversie: din registru <-> binar, hexa, etc

11.12. Lucrul cu fișiere

(EMU) file-operations.asm

Atunci când lucrăm cu fișiere, trebuie acordată mare atenţie: EMU va salva fișierele sau va căuta cele sursă în directorul a cărui cale este setată din opţiuni, la ***assembler-> set output directory***; în Figura 11.20 s-a setat directorul *testFisier* de pe E:, dar poate fi ales orice loc de pe harddisc atât timp cât există drepturi de scriere pe acea partiţie.

Analizaţi programul din EMU și extrageţi secvenţe separate pentru următoarele operaţii:
(1) creare și ștergere fișier;
(2) deshidere, citire/ scriere și închidere fișier;
(3) creare director;
(4) ștergere director.

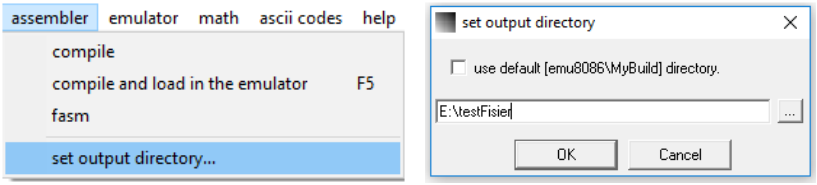


Figura 11.21 Setări pentru lucrul cu fișiere

Capitolul 12. Întreruperi şi macrouri

În cadrul programelor abordate în *Capitolul 11* s-au folosit frecvent instrucţiuni ***int x***. Despre aceste instrucţiuni se spune că sunt „apeluri de întreruperi software". De asemenea, tot în capitolul anterior am văzut programe care au folosit instrucţiunea ***call numeSubr***; acestea desemnează aşa numitele subrutine, un fel de echivalent al întreruperilor însă care pot fi scrise de utilizator. Am văzut chiar şi instrucţiuni care apăreau ciudat, doar ca un simplu ***numeMacr***; acestea sunt de fapt aşa numitele invocări de macrou sau macroinstrucţiuni. Care sunt diferenţele între aceste 3 mari categorii vom vedea mai pe larg în acest capitol.

12.1. Întreruperi

La modul general, întreruperea care poate să apară într-un SC, nu este altceva decât un semnal transmis sistemului prin care acesta este anunţat de apariţia unui eveniment deosebit ce necesită atenţie.

La procesoarele x86 există mai multe tipuri de întreruperi, după cum urmează:

- întreruperi hardware, care la rândul lor pot fi:
  - interne şi externe – se pot clasifica drept dezactivabile / nedezactivabile,
- întreruperi software
  - pentru sistem – de tip BIOS sau DOS, sau întreruperi utilizator.

Întreruperile hardware sunt întreruperi pt controlul programelor bazate pe un eveniment hardware şi în general, cele externe, nu au nimic în comun cu instrucţiunea ce se execută la momentul apariţiei lor (sunt asincrone cu operaţiile microprocesorului).

Întreruperile hardware interne apar ca urmare a unor condiţii speciale de funcţionare a procesorului, de exemplu execuţia pas cu pas a programelor sau întreruperea cu tipul 0 ce apare la tentativa de împărţire cu 0. Acestea mai sunt referite şi sub numele de *excepţii* (în engleză exceptions).

Întreruperile hardware externe (cum ar fi apăsarea unei taste de la tastatură) pot fi provocate de semnale electrice care se aplică la intrările INT (Interrupt) şi NMI (Non Maskable Interrupt) ale procesorului. Acestea pot fi dezactivabile, adică acceptarea lor depinde de starea flagului IF (interrupt) astfel: dacă flagul IF este 1, procesorul poate accepta aceste întreruperi, identificând anterior tipul lor; dacă flagul IF este 0, procesorul nu va lua în considerare aceste întreruperi, spunându-se despre ele că sunt „mascate" sau „dezactivate" (de aici şi denumirea de întreruperi dezactivabile). ***Întreruperile hardware dezactivabile*** sunt controlate de unul sau mai multe circuite specializate (controlere de întreruperi I8259A) care le trimit mai departe pe linia INT a procesorului. ***Întreruperile hardware nedezactivabile*** nu ţin cont de starea flagului IF, deci ele sunt întotdeauna luate în considerare, fiind provocate de semnale electrice aplicate pe linia NMI (non-maskable interrupt) a procesorului.

Întreruperile software apar ca urmare a execuţiei unor instrucţiuni (de exemplu cele ce folosesc instrucţiunea *INT*). Acestea pot fi clasificate drept ***întreruperi sistem***, care sunt gata scrise şi pot fi de tip BIOS sau DOS, sau ***întreruperi utilizator*** care pot fi rescrise pentru a realiza diferite operaţii, după dorinţa programatorului. Întreruperile software mai sunt referite şi sub denumirea de *trapă* (în engleză trap).

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. Instrucţiunea *INT* realizează de fapt patru acţiuni succesive:

- pune în stivă flagurile (PSW);
- pune în stivă adresa *FAR* de revenire (IP şi CS);
- pune 0 în flagurile TF şi IF;
- apelează prin adresare indirectă handlerul asociat întreruperii, adică RTI (Rutina de Tratare a Întreruperii) asociată.

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

Instrucţiunea ***IRET*** provoacă revenirea dintr-o întrerupere, ea fiind ultima instrucţiune executată într-un handler; aceasta are efect complementar instrucţiunii *INT*:

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

Întreruperile software oferă accesul la servicii BIOS şi servicii DOS; sunt foarte mult utilizate datorită facilităţii oferite: ele constituie o bază sau o librărie de programe (rutine) gata scrise, care simplifică mult munca programatorului în limbaj de asamblare. Aceste rutine poartă numele de *servicii*. În cele ce urmează, se vor exemplifica servicii pentru câteva întreruperi folosite mai des în scrierea programelor.

Întreruperi şi servicii BIOS şi DOS

Orice microcalculator dispune de o colecţie de funcţii realizate de către firma producătoare cu scopul de a utiliza resursele hardware ale sistemului. În BIOS sunt scrise o serie de astfel de funcţii legate de echipamentele hardware de intrare/ieşire. Apelarea acestor funcţii BIOS gata scrise într-o aplicaţie se realizează prin întreruperi: ***în cadrul fiecărei întreruperi există mai multe servicii***. Serviciul dorit se selectează prin ***încărcarea în registrul AH a unui număr specific acelui serviciu***, înainte de apelarea întreruperii. De asemenea, serviciul mai poate folosi şi alţi regiştrii (precum AL, BX, DL, DH, DX, DS:DX), care trebuie şi ei încărcaţi după caz.

Funcţii BIOS: BIOS-ul PC-ului IBM foloseşte întreruperi software de tipul *5* şi *10h...1Ah* pentru îndeplinirea unor operaţii utilizând echipamentele instalate în sistem. Astfel, instrucţiunile *INT 5* şi *INT 10h...INT 1Ah* furnizează interfaţa cu BIOS.

În continuare sunt prezentate care sunt aceste funcţii BIOS, cele mai utilizate în lucrarea de faţă fiind *INT 10h* şi *INT 16h*:

INT	*Funcţia*
5h	Operaţia Tipărire Ecran (Print Screen)
10h	**Utilizare terminal grafic (video display)**
11h	Determinare configuraţie sistem (Equipment determination)
12h	Determinare dimensiune memorie (Memory size determination)
13h	Utilizare disc flexibil (Diskette and hard disk)
14h	Utilizare interfaţă serială (Serial I/O)
16h	**Utilizare tastatură (Keyboard)**
17h	Utilizare imprimantă (Printer)
19h	Reîncarcă sistemul de operare (reboot)
1Ah	Utilizare ceas de timp real (Real time clock)

Majoritatea acestor rutine folosesc parametri şi în regiştrii microprocesorului 8086, sau chiar în anumite locaţii de memorie.

Servicii video - INT 10 h

- **Setarea modului video** – se realizează folosind serviciul 00h
  AH = 00 ; modul video dorit
  AL = codul modului video care poate fi în general:
  00h – mod text cu 40x25 caractere, 16 culori, 8 pagini;
  03h – mod text cu 80x25, caractere, 16 culori, 8 pagini;
  13h – mod grafic cu 320x200 pixeli, 256 culori, 1 pagină – se va ţine cont că în acest mod valoarea lui CX e dublată.
- **Setează forma cursorului**
  AH = 01h
  CH = linie de start cursor (biţii 0-4) și opţiunile (biţii 5-7).
  CL = linie de capăt cursor (biţii 0-4).
  când bitul 5 al registrului CH e în 0, cursorul e vizibil; altfel, nu e vizibil.
- **Setarea poziţiei cursorului**
  AH = 02
  BH = numărul paginii video (0 pentru modul grafic)
  DH = rândul
  DL = coloana

- **Scrierea caracterului la cursor** (fără deplasarea cursorului)
  AH = 09
  AL = codul ASCII al caracterului de scris
  BH = pagina video
  BL = atribut de culoare
- **Scrierea unui pixel grafic la coordonate sau schimbarea culorii unui pixel**
  AH = 0Ch
  AL = culoarea
  BH = pagina video
  CX = coloana
  DX = rândul
- **Preia culoarea unui pixel**
  AH = 0Dh
  CX = coloana pixelului dorit
  DX = rândul pixelului dorit
  ⇨ în AL se va obţine culoarea pixelului
- **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

- **Aşteaptă o tastă şi citeşte caracterul tastat**
  AH = 00
  ⇨ AL = (returnat) codul ASCII al caracterului tastat
- **Citeşte starea tastaturii**
  AH = 01
  ⇨ Z = 0 - s-a apăsat o tastă; Z = 1 - nu s-a apăsat o tastă

Servicii mouse - INT 33h

- **Citeşte poziţia mouse-ului și starea butoanelor sale**

AX=0003 - returnează:

⇨ BX = 1 dacă butonul din stânga e apăsat
BX = 2 dacă butonul din dreapta e apăsat
BX = 3 dacă ambele butoane sunt apăsate
CX = x
DX = y

Funcţii DOS

Sistemul de operare DOS conţine o colecţie de funcţii pe care le utilizează pentru gestiunea resurselor calculatorului şi pe care utilizatorul le poate apela din programe scrise în limbaj de asamblare. Astfel, utilizatorul poate avea şi el acces la resursele calculatorului, memorie şi periferice, precum şi la o serie de informaţii care descriu contextul curent în care funcţionează calculatorul. Întreruperea ***INT 21h*** este asociată funcţiilor DOS, sarcinile unora dintre întreruperile amintite mai sus fiind preluate şi uneori extinse de către unele din funcţiile acestei întreruperi.

Similar funcţiilor BIOS, registrul AH trebuie încărcat cu serviciul dorit, iar alţi parametri folosiţi de serviciul respectiv pot fi depuşi în DL, DX, DS:DX, după caz.

- **Citirea unui caracter de la tastatură**
  AH=01
  => AL = (returnat) caracterul citit
- **Scrierea unui şir de caractere terminat cu "$"**
  AH = 09
  DS:DX – pointer la un şir ce se termină cu caracterul "$"
- **Terminarea procesului cu cod de retur** – este o metodă de terminare a unui program; nu este suportată de versiuni DOS sub 2.x.
  AH = 4Ch
  AL = cod de retur

Pentru o listă mai cuprinzătoare a acestor instrucţiuni, se sugerează utilizarea Help-ului din EMU, dar și http://www.ousob.com/ng/asm/.

12.2. Macroinstrucţiuni

Macroinstrucţiunile se mai numesc şi macrouri (în engleză macros) şi ele reprezintă secvenţe de program (instrucţiuni, definiţii de date, directive) asociate unui nume. Macroinstrucţiunile se declară o singură dată, dar pot fi folosite ori de câte ori este nevoie. Prin folosirea numelui macroinstrucţiunii în program, se obţine înlocuirea acestuia cu secvenţa de program asociată (operaţie numită expandare), procesul realizându-se înainte de asamblarea propriu-zisă.

Folosirea macrourilor implică următoarele etape:
- definirea (declararea) lor– macroinstrucţiunile se declară cu directiva MACRO:
NumeMacrou ***MACRO*** *[parametrul1, parametrul2,...]*
- scrierea corpului macroului – reprezintă codul asociat macroinstrucţiunii şi se termină cu ENDM,
- apelul (sau mai corect invocarea) macroinstrucţiunii.

Macroinstrucţiunile pot fi: fără parametri / cu parametri şi repetitive.

12.3. Subrutine

Subrutina, numită simplu și rutină sau procedură este o secvenţă de instrucţiuni scrisă separat, care poate fi apelată din diferite puncte ale unui program. Mecanismul de implementare al subrutinelor este diferit de cel al macrourilor, acesta fiind realizat cu ajutorul instrucţiunilor ***CALL*** şi ***RET***. Instrucţiunea *CALL* are un singur operand, iar instrucţiunea *RET* (return) nu are operanzi.

Pentru a declara o procedură în limbaj de asamblare se folosesc directivele *PROC* şi *ENDP*, care sunt directive asamblor: ele nu generează nici un cod, ci reprezintă doar un mecanism pentru a simplifica munca programatorului. Despre *PROC* şi *ENDP* se mai spune că sunt *separatori logici* ai procedurilor deoarece revenirea din procedură se face prin execuţia unei instrucţiuni *RET* şi nu la întâlnirea unei directive *ENDP*.

O subrutină (procedură) la microprocesoarele din familia 80x86 are forma:

NumeProc ***PROC*** *[NEAR|FAR]*
; corpul procedurii
RET
NumeProc ***ENDP***

unde numele procedurii trebuie să fie unic în program, iar operanzii *NEAR* şi *FAR* pot să lipsească, întrucât sunt opţionali.

Exemplu:

În secvenţa următoare este scrisă o subrutină care adună doi octeţi (stocaţi în regiştrii AH şi AL) şi depune rezultatul în registrul BX.

```
Aduna PROC NEAR      ; declararea procedurii Aduna de tip NEAR
xor BX,BX            ; se goleşte registrul BX, pentru ca aici se va depune
                     ; rezultatul adunării, acesta putând depăşi dimensiunea
                     ; de 8 biţi
mov BL, AH           ; unul dintre octeţii de adunat, cel din AH, se mută
                     ; în registrul BL
mov AH, 00           ; în locul lui, se pune 0 în registrul AH
add BX, AX           ; adună peste primul octet (aflat în BL) şi cel de-al
                     ; doilea octet (din AL), rezultatul fiind în BX
RET                  ; directiva Return marchează revenirea din subrutină
Aduna ENDP           ; sfârşit declarare procedură
```

În programul principal, această subrutină va fi apelată prin instrucţiunea: *call Aduna*.

Dacă se omit parametrii *NEAR* şi *FAR*, tipul procedurii este dedus din directivele de definire a segmentelor, respectiv modelul de memorie folosit. De exemplu, la modelul *LARGE* toate procedurile sunt de tip *FAR*.

Diferenţa între funcţii şi proceduri în limbaj de asamblare este mai mult o problemă de definire a acestora. În general, scopul unei funcţii este de a returna valori explicite, în timp ce scopul unei proceduri este de a executa anumite acţiuni. Cu toate acestea, toate regulile şi tehnicile aplicate procedurilor se pot folosi şi asupra funcţiilor. Astfel, atunci când ne vom referi la proceduri, acestea pot desemna proceduri sau funcţii.

***Exemple* de apeluri de întreruperi pentru interacţionarea cu utilizatorul:**

Exemple:

; secvenţa următoare ***setează poziţia cursorului***

```
mov AH,02       ; selectarea serviciului 02h de setare poziţie cursor cu INT 10h
mov BH,0        ; selectare pagina video 0 pentru modul grafic
mov DH,0Eh      ; rândul pe care se va afişa caracterul este rândul 15
mov DL,12h      ; coloana pe care se va afişa caracterul este coloana 18
INT 10h         ; apelarea întreruperii
```

; secvenţa următoare ***scrie caracterul 'A' la pozitia curentă*** a cursorului

```
mov AH,09h      ; selectarea serviciului 09h de afişare caracter cu INT 10h
mov AL,'A'      ; caracterul de afişat este 'A'
mov BH,0        ; pagina video 0 pentru modul grafic
mov BL, 41h     ; caracter de culoare albastră pe fond roşu
INT 10h         ; apelarea întreruperii
```

; secvenţa următoare ***citeşte un caracter de la tastatură şi apoi îl afişează pe ecran***

```
mov AH,00       ; selectare serviciu de preluare caracter de la tastatură
INT 16h         ; după apelul INT 16h, în AL se va găsi codul Ascii al caract. tastat
mov AH,09h      ; selectare serviciu de afişare caracter
mov BH,0        ; pagina video
INT 10h         ; apel întrerupere
```

; secvenţa următoare ***afişează un mesaj care se termină cu caracterul '$'***

```
.data
mesaj db 'Acest program afiseaza un mesaj', '$'
.code
mov DX, offset mesaj    ; offsetul mesajului se depune în perechea DS:DX
mov AH, 09h             ; selectarea serviciului
INT 21h                 ; apelarea întreruperii
```

***Exemplu* de macrou sau macroinstrucţiune:**

Grupul de caractere 13 și 10, adică CR(carriage return) urmat de LF(line feed) determină apariţia pe ecran a unui <Enter>, sau mai bine spus cursorul va coborâ pe linia următoare, la început de linie. Dacă se va defini un macrou de forma:

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

acesta va putea fi apoi utilizat în programe ori de câte ori se dorește afișarea unei linii noi pe ecran; de exemplu:

```
; grupul de instrucţiuni pentru realizarea operaţiei 1
print_new_line
; grupul de instrucţiuni pentru realizarea operaţiei 2
print_new_line
; grupul de instrucţiuni pentru realizarea operaţiei 3
print_new_line
; grupul de instrucţiuni pentru realizarea operaţiei n
```

Deci macroul se apelează (corect se folosește termenul ***invocă***) în program simplu, doar prin numele lui. Această invocare nu va face altceva decât să înlocuiască în program (în loc de numele macroului) cu tot setul de instrucţiuni ce apare în corpul macroului, deci are loc așa numita expandare a codului. Această operaţie de expandare se realizează înainte de asamblare. Spre deosebire de proceduri, un macrou nu duce la obţinerea unui program mai mic în memorie, ci doar textul din codul sursă al programului devine mai ușor de urmărit și mai scurt.

Macrourile pot fi definite chiar și în fișiere separate și atunci utilizarea lor va implica folosirea directivei INCLUDE.

Exemplu: Analizaţi conţinutul fișierului „include.asm” din directorul EMU:/examples.

Capitolul 13. Elemente de programare în limbaj de asamblare

O primă modalitate de a executa programe a fost prin intermediul simulatorului; o altă posibilitate este execuţia sub DosBox, folosind pachetul de programe TASM pe 16 biţi.

13.1. Pașii de urmat în scrierea unui program în limbaj de asamblare

Mai multe întrebări se pot adresa în această secţiune în vederea scrierii programelor:
Cum se scrie sablonul pentru program .com ? Dar pentru unul .exe ?

Pentru a răspunde acestor întrebări se indică urmărirea Figurii 13.1.

Există un *sablon tip* care trebuie respectat?
Da, este necesară respectarea unor reguli în scrierea programelor, altfel e posibil să apară tot felul de erori "ciudate".
Unde se scrie acest șablon?

Într-un editor de text, gen *Notepad* ++, …

Cum se salvează acest șablon?

Obligatoriu cu extensia .asm, (nu .txt)

Fișierul sursa .asm obţinut va trebui apoi ASAMBLAT, LINKEDITAT și DEPANAT/ sau EXECUTAT; aceste operaţii se realizează în general cu ajutorul unui asamblor; pentru aceasta, se pot folosi utilitarele *TASM.exe, TLINK.exe, TD.exe* (de exemplu).

(Cum se scrie sablonul pentru program .com ?) Cum se scrie sablonul pentru program .exe ?
Exista un sablon tip care trebuie respectat, altfel e posibil sa apara tot felul de erori "ciudate".

	Folosind EMU	Folosind TASM

SABLON program de tip .EXE	; multi-segment executable file template.
	stack segment
	dw 128 dup(0)
	ends ; sau *stack ends*
	data segment
	; add your data here!
	ends ; sau *data ends*
	code segment
	start:
	; set segment registers:
	mov ax, data
	mov ds, ax
	mov es, ax
	; add your code here
	mov ax, 4c00h ; exit to operating system.
	int 21h
	ends ; sau *code ends*
	end start ; set entry point and stop the assembler.
		;sablon folosind definirea prescurtata a segmentelor
		.model small
		.stack 200h
		.data
		;
		; sectiune definire date
		;
		.code
		main label
		; pozitioneaza registrul data segment pe data
		mov ax,@data
		mov ds,ax
		;
		; sectiune linii cod
		;
		mov ax, 4c00h ; exit to operating system.
		int 21h
		end main

La fisierele de tip **.COM**: toate segmentele vor incepe la aceeasi adresa => **DS=CS=SS=ES = 0700h in EMU**
La fisierele de tip **.EXE**: segmentele vor incepe la adrese diferite => **DS=ES = 0700h, dar SS=0712h, CS=0722h in EMU**

Figura 13.1 Șablonul de urmat în scrierea unui program

Cum trebuie să apară în directorul de lucru fișierele folosite?
Vedeţi mai jos sugestii pentru modul de salvare al fișierului sursă: obligatoriu ***cu extensia .asm*** și ***nume fără spaţii***; în cazul prezentat, fișierele s-au salvat în E:/UT/Lab06, dar puteţi să le salvaţi local oriunde, singura condiţie este ca și fișierele *TASM.exe, TLINK.exe, TD.exe* să fie în același director, de exemplu așa cum arată Figura 13.2.

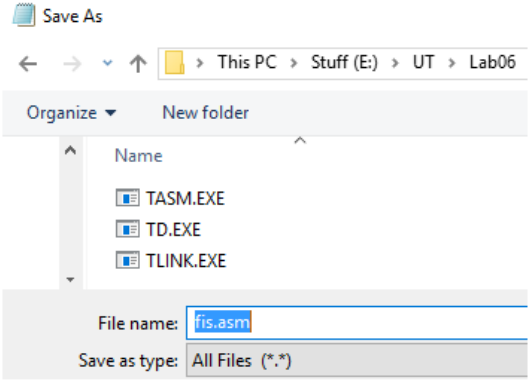


Figura 13.2 Organizarea fișierelor în scrierea unui program

La adresarea în mod real, programele pe 16 biţi pot accesa porturile hardware, vectorii de întrerupere, memoria sistem doar rulând sub MS-DOS, ca la Windows 95, 98, şi Millenium, acest tip de acces nefiind permis sub Windows NT, 2000, XP, Seven, etc.

13.2. Etape în dezvoltarea programelor în limbaj de asamblare

Obţinerea fişierelor executabile:

Un program sursă, scris deci în limbaj de asamblare, nu poate fi executat direct pe calculatorul țintă. Acesta trebuie tradus, translatat sau *asamblat* în cod executabil. De fapt, un asamblor este foarte similar cu un compilator (adică acel tip de „program” ce s-ar folosi pentru a traduce un program din C++ sau Java în cod executabil). Asamblorul produce un fișier care conține limbaj mașină numit ***fișier obiect***. Acest fișier nu este destul de pregătit încă pentru a fi lansat în execuţie: trebuie trecut la un alt program numit linker și acesta va produce un ***fișier executabil.*** Acest fișier este gata acum să se execute de la linia de comandă din MS-DOS / Windows (a se vedea Figura 13.4).
Asambloarele nu sunt altceva decât unelte software care transformă secvenţele de instrucţiuni (adică programele) scrise în limbaj de asamblare în cod executabil. Pentru ca un program scris în limbaj de asamblare să devină cod executabil, este nevoie de parcurgerea etapelor prezentate în Figura 13.3.

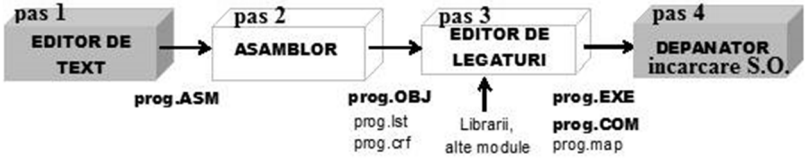


Figura 13.3. Etapele dezvoltării programelor în limbaj de asamblare

Observaţie: Trecerea de la o etapă la următoarea se realizează doar după eliminarea erorilor sintactice din etapa curentă. De exemplu, nu se poate trece la *link-editare* dacă s-au obţinut erori la *asamblare*.

Pas 1. Se scrie programul în limbaj de asamblare cu ajutorul unui **editor de texte**, obţinându-se *fişierul sursă*. Se poate folosi orice editor standard de text ASCII (de exemplu EDIT din DOS sau NOTEPAD din WINDOWS). Fişierul sursă trebuie salvat cu extensia **.asm**, iar la scrierea programului sursă se va ţine cont de convenţiile şi sintaxa impusă de asamblor.

Pas 2. Cu ajutorul unui **asamblor** se citeşte fişierul sursă şi rezultă astfel un fişier obiect, adică programul este tradus în limbajul maşinii (binar). **Asamblorul** este deci un program de conversie a secvenţelor scrise din limbaj de asamblare (.**asm**) în cod maşină. După asamblare se obţine un fişier obiect cu extensia **.obj**, dar mai pot rezulta (opţional) fişiere listing (.lst) sau fişiere de referinţă (.crf). Asamblorul asigură şi prelucrarea etichetelor astfel încât fiecare adresă simbolică este înlocuită cu adresa relativă sau absolută a acesteia.

Pas 3. **Editorul de legături (link-editorul)** leagă mai multe module obiect (.obj) rezultate din asamblări, împreună cu diferite componente de bibliotecă, obţinând un program executabil de tip **.com** (≤64 Kocteţi) sau **.exe** (>64 Kocteţi). Editorul de legături citeşte fişierul obiect (conţin codul obiect dar adresele sunt relative/simbolice), verifică dacă există apeluri de proceduri din alte librării, copiindu-le şi legându-le apoi de fişierul obiect, semnalează eventualele referinţe nerezolvate şi, în cazul în care nu au fost erori, generează fişierul executabil în cod obiect absolut. Modul de lucru este similar cu cel din limbajele de nivel înalt când se folosesc funcţii sau proceduri din bibliotecile externe. Este posibilă obţinerea fişierelor .map care conţin informaţii generate de compilator, de exemplu adresele etichetelor rezultate în urma legării modulelor.

Pas 4. Programul încărcător al S.O. încarcă fişierul executabil în memorie şi apoi direcţionează CPU înspre această adresă în vederea execuţiei lui.

În afară de uneltele (programele) folosite pentru prelucrare, se mai poate folosi şi un program numit **dezasamblor** (disassembler), util în depanare, care are acţiune inversă asamblorului: traduce formatul din cod obiect absolut în textul corespunzător.

Depanatorul (debugger) este utilizat la testarea programului executabil, pentru a înlătura posibilele erori semantice/ logice ale programului. Aceasta se realizează urmărind în paralel rezultatele (reale) obţinute cu cele (dorite) estimate. Ca depanator se poate folosi Turbo Debugger.

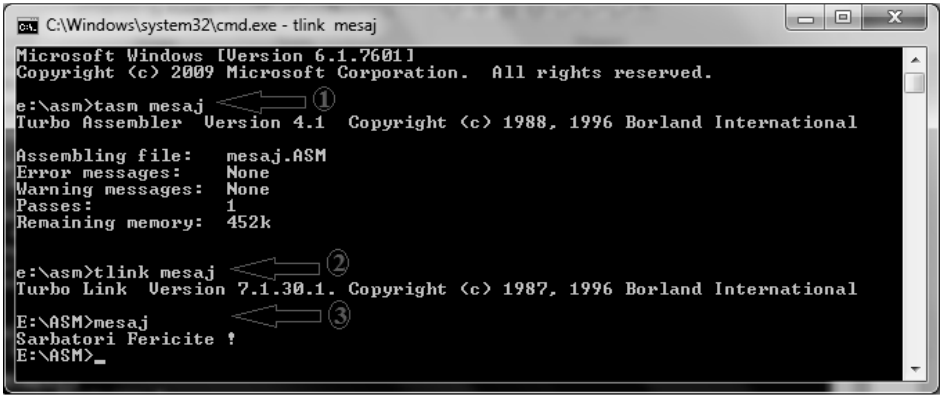


Figura 13.4. Comenzile necesare obţinerii codului executabil

Exemplu:

Mai jos este redat programul sursă care a fost asamblat, linkeditat și executat în cazul prezentat în Figura 13.4. Comentați acest program linie cu linie, în zona indicată:

```
title mesaj.asm
.model small
.stack 200h
.data
    mesaj db 'La Multi Ani !', '$'
.code
main label
      mov ax,@data
      mov ds,ax

afisare:
mov ah,09
mov dx,offset mesaj
int 21h

iesire:
mov ax,4c00h
int 21h

end main
```

13.3. Definirea simplificată a segmentelor

La variantele mai noi ale asambloarelor, memoria poate fi segmentată prin utilizarea unor *modele prestabilite de segmentare a memoriei*, furnizate de către asamblor. Acestea au rolul de a uşura activitatea de programare şi de a asigura compatibilitate d.p.d.v. al structurii programului (segmente cu nume şi atribute identice) cu programele scrise în limbajele de nivel înalt.

Directivele folosite sunt:
.model *tip* - *tip* poate fi:
- *tiny* - lungimea codului + lungimea datelor + lungimea stivei < un segment (64KB)
- *small* - lungimea codului < un segment, lungimea datelor + lungimea stivei < un segment
- *medium* - lungimea codului > un segment, lungimea datelor + lungimea stivei < un segment
- *compact* - lungimea codului < un segment, lungimea datelor + lungimea stivei > un segment
- *large* - lungimea codului > un segment, lungimea datelor + lungimea stivei > un segment
- *huge* - la fel ca modelul anterior, cu diferenţa că referinţele sunt normalizate.

În general: un program are un segment de cod, unul de date şi unul de stivă, însă acestea pot fi grupate în cadrul unui singur segment (modelul tiny), cum se întâmplă în cazul programelor .COM.

Sintaxele directivelor simplificate de definire a segmentelor sunt:
. stack *[dimensiune]* ; implicit dimensiunea este 512 octeţi pentru TASM
. code *[nume]* ; încărcarea registrului segment (CS) se va face automat de către S.O. la execuţie
. data ; utilizatorul va trebui să iniţializeze explicit DS cu adresa segmentului de date

Folosirea directivelor simplificate prezintă şi avantajul că nu mai este necesară scrierea directivei ***ASSUME***, deoarece există câteva forme implicite.

Obţinerea programului executabil (**.COM**) pentru aplicaţii Microsoft, respectiv Borland se realizează prin secvenţa:

```
MASM      PR_COM.ASM          TASM      PR_COM.ASM
LINK      PR_COM.OBJ          TLINK /t  PR_COM.OBJ
EXE2BIN   PR_COM.EXE                    PR_COM.COM
```

Obţinerea programului executabil (**.EXE**) corespunzător se realizează astfel:

```
MASM   PR_ EXE.ASM          TASM    PR_EXE.ASM
LINK   PR_ EXE.OBJ          TLINK   PR_ EXE.OBJ
                                    PR_EXE.EXE
```

Şabloanele corespunzătoare programelor de tip .COM, respectiv .EXE sunt date în continuare:

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
```

Structura unui program .EXE :

```
Stiva     SEGMENT PARA STACK 'STACK'
          DW 512 DUP(?)
Stiva     ENDS
Data      SEGMENT  PARA  PUBLIC 'DATA'
; definire 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
```

13.4. Aspecte de organizare a programelor în asamblare

În general, putem spune că există 3 tipuri de programe, în funcţie de următoarele etape care pot să intervină în interacţiunea cu utilizatorul:

ETAPA I. **Preluarea/ definirea/ iniţializarea datelor**

- **(TIP1) Cu definirea datelor în program (*în data segment*)**
- **(TIP2) Cu preluarea datelor de la tastatură *prin întreruperi***
- **(TIP3) Cu preluarea datelor de la tastatură *prin linia de comandă* (sau vom spune "cu PSP"; PSP vine de la Program Segment Prefix)**

ETAPA II. **Specificul programului** – calcule/ prelucrări/operaţii, în funcţie de cerinţă: se va calcula suma, se va sorta un șir, etc

ETAPA III. **Afișarea rezultatelor** este comună la toate tipurile 1,2,3 de mai sus, și în general poate avea legătură cu:

Afișarea unui singur caracter (poate reprezenta un nr sau un caracter);
Afișarea unui sir de caractere;
Afișarea unui text;
Controlul poziţiei cursorului/ atributului de culoare, etc

Astfel, programele de Tip 1 nu interacţionează DELOC cu utilizatorul,
cele de Tip 2 – interacţionează prin intermediul *întreruperi*lor (“prin întreruperi”), iar
cele de Tip 3 – interacţionează prin interm. parametrilor din *linia de comanda* (“cu PSP”).
Primele două le-am văzut: am definit date preponderent în *Capitolul 7*, iar cu întreruperi am lucrat în *Capitolul 12*. În continuare, vom vedea ce înseamnă lucrul cu PSP.

Prefixul unui program executabil (PSP)

Imaginea în memorie a unui program executabil de tip .EXE sau .COM este precedată de un antet numit PSP (Program Segment Prefix). În momentul încărcării programului executabil, imaginea lui în memorie este completată cu acest „tabel” de 256 octeţi. La lansarea în execuţie a programului, adresele DS:0000 şi ES:0000 indică începutul blocului PSP asociat programului. Informaţiile din PSP sunt utilizabile direct de către sistemul de operare DOS şi indirect de către utilizator, structura în memorie a PSP fiind de importanţă majoră pentru linia de comandă, deoarece începând cu adresa 80h, aceasta conţine:

00 ...7Fh	Prima jumătate a PSP, informaţii importante pentru sistem
80h	Lungimea cozii liniei de comandă (maxim 127 octeţi), pe un octet
81h	Coada liniei de comandă (cu spaţiu şi urmată de un caracter CR)
82h	Aici încep efectiv caracterele din linia de comandă

Tabelul 13.2. Structura PSP

La ce se referă mai exact „linia de comandă” ?

O comandă DOS are forma: ***. . . >numecomanda param$_1$, . . . , param$_n$***
Porţiunea *param$_1$, . . . , param$_n$* se numeşte ***coada liniei de comandă*** şi este memorată în a doua jumătate a tabelei PSP (începând cu adresa 81h).

Tabelul 13.3 se referă la un exemplu de program care dispune de un caracter literă mică; acesta ori va fi definit în memorie (tip1), ori va fi preluat de la tastatură prin întreruperi (tip2), ori va fi preluat de la tastatură ca parametru din linia de comandă (tip3). Programul îl va transforma în majusculă și-l va afișa pe ecran cu int 10h, serviciul 0Eh.

Având şablonul unui program de tip .exe (a se consulta *Structura unui program .EXE*), se copiază acest şablon într-o fereastră nouă a unui editor de text (notepad, edit, etc) şi apoi se completează în zonele corespunzătoare cu directivele/ instrucţiunile specifice problemei de rezolvat. După scrierea programului, acesta va fi salvat cu extensia .asm în directorul curent. Pentru a da comenzile specifice obţinerii unui program .exe dintr-un fişier sursă .asm, se va deschide o fereastra DOS, în care se vor da pe rând comenzile:

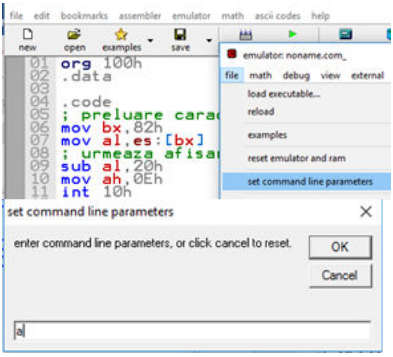
```
TASM progr.asm
TLINK progr.obj
progr            ; comanda pentru a lansa în execuţie programul exe
```

Cum se lansează în execuţie în EMU fiecare din cele 3 tipuri de programe?

Exemplu simplu de program transpus în cele 3 tipuri de variante de execuţie:

Tabelul 13.3. Tipuri de programe și moduri de execuţie

TIP 1	TIP 2	TIP 3
Cu definire de date în segmentul de date	**Cu preluare caracter prin întreruperi**	**Cu preluare de caracter din linia de comanda (cu PSP)**
org 100h .data Val db 'a' .code **mov al, Val** ; urmează afișarea caracterului pe ecran, transformat în majusculă **sub al,20h** **mov ah,0Eh** **int 10h**	org 100h .data .code ; preluare caracter ;fără ecou **mov ah,0h** **int 16h** **;** caracterul va fi in AL ; urmeaza afisarea ;caracterului pe ecran, ;transformat in ;majuscula **sub al,20h** **mov ah,0Eh** **int 10h**	org 100h .data .code ; preluare caracter din PSP **mov bx,82h** **mov al,es:[bx]** ; urmează afișarea caracterului pe ecran, ;transformat în majusculă **sub al,20h** **mov ah,0Eh** **int 10h**
La apel în EMU:		
Emulate-> RUN	**Emulate-> RUN**	**Emulate** -> în fereastra care apare, meniul **File** -> **Set command line parameters** și apoi în caseta care apare se tastează **caracterul dorit**

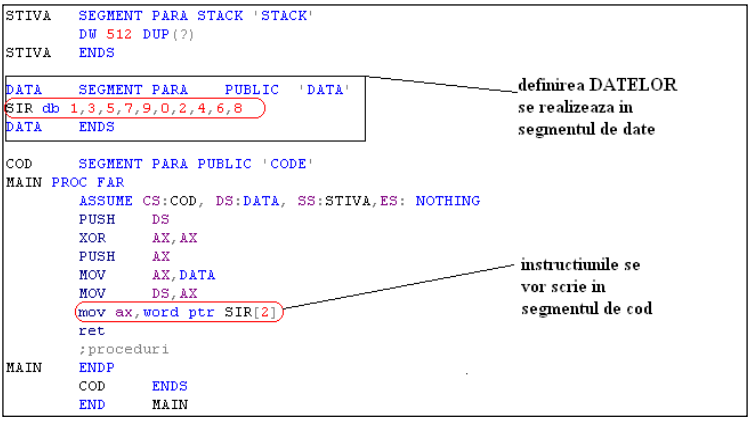


După apăsarea **OK**, **Yes** -> **Reload** -> **Run**

Problema 1:

Se consideră o variabilă definită astfel:
SIR db 1,3,5,7,9,0,2,4,6,8.
Comentaţi instrucţiunea mov ax, word ptr SIR[2]. Ce se va încărca în reg. AX ?

Observaţii: Datele (variabilele) se vor defini în segmentul de date, iar instrucţiunile în segmentul de cod, după cum se poate urmări în Figura 13.5.




```
STIVA   SEGMENT PARA STACK 'STACK'
        DW 512 DUP(?)
STIVA   ENDS

DATA    SEGMENT PARA    PUBLIC  'DATA'
SIR db 1,3,5,7,9,0,2,4,6,8
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
        mov ax,word ptr SIR[2]
        ret
        ;proceduri
MAIN    ENDP
        COD     ENDS
        END     MAIN
```

Figura 13.5. Introducerea datelor şi a instrucţiunilor în cadrul unui şablon cu definire completă a segmentelor

Problema 2:
Studiaţi următoarea aplicaţie (și inseraţi explicaţii în zona marcată) care afişează parametrii din linia de comandă, scrişi cu majuscule:
; afişare mesaj din linia de comandă (PSP), se preiau *** 7 caractere ca litere mici pe care programul le va scrie cu litere mari; aceasta se va obţine prin scăderea din codul ASCII a valorii 20h**
; se face şi verificarea parametrilor din linia 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 ;(8=spaţiu +7 caractere)
mov bx, 80h
          mov al, es:[bx]
          cmp al,8
          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,7
            jnz repeta
iesire: mov ax,4c00h
            int 21h
end main
```

Observaţie! Pentru a executa acest program, se va scrie comanda : *numeFisier abcdefg* adică numele fişierului executabil, urmat de spaţiu şi apoi 7 caractere mici care vor fi transformate de program în majuscule.

Problemă Propusă:

Se preia un număr format din 5 cifre. Să se verifice dacă e scris într-o anumită bază: new d 12345 – nume fisier: new, primul parametru: d, iar șirul de 5 caractere reprezintă un număr care trebuie verificat dacă e în baza 10 sau nu.

Se vor folosi multiple modalităţi de afișare a textului pe ecran:

Varianta 1- Afișarea cu int21h, serviciul 09h – cel cu $

Varianta 2 - Afișarea cu int 10h, serviciul 0eh – mod teletype

Varianta 3 - Afis cu int 10h, serviciul 09 – fără teletype dar cu modificare poziţie cursor cu **int 10h, ah=02h** + controlul atributului de culoare

Care este diferenţa între cele 3 modalităţi de afișare pe ecran a unui șir de caractere?

Cu varianta 1 toate caracterele *apar deodată*, la apelul întreruperii, pe când cu celelalte două metode acestea *apar pe rând*, câte unul. Controlul e mai mare în al doilea și al treilea caz – putem controla fiecare caracter, dar mai ales *atributul de culoare* sau *poziţia* fiecărui caracter înainte de a apărea pe ecran.

Capitolul 14. Exerciţii şi aplicaţii propuse

In continuare se prezintă câteva probleme rezolvate, acestea fiind apoi urmate de probleme propuse spre rezolvare, unele având chiar sugestii sau observaţii în vederea impunerii unui anumit mod de rezolvare. Prescurtări folosite: **PR** provine de la *Problema Rezolvată*, iar **PP** de la *Problema Propusă*. Inseraţi voi comentariile potrivite.

Ca o TEMĂ GENERAL PROPUSĂ: oricare din problemele de mai jos poate fi adaptată în ceea ce privește existenţa datelor iniţiale (de prelucrat) în program prin una din cele 3 modalităţi: cu definirea datelor în segmentul de date, cu întreruperi de la tastatură, cu preluarea parametrilor din linia de comandă (PSP).

Nu uitaţi să inseraţi directiva org 100h la început de program, vom lucra cu fișiere .com.

14.1. Probleme rezolvate ca model

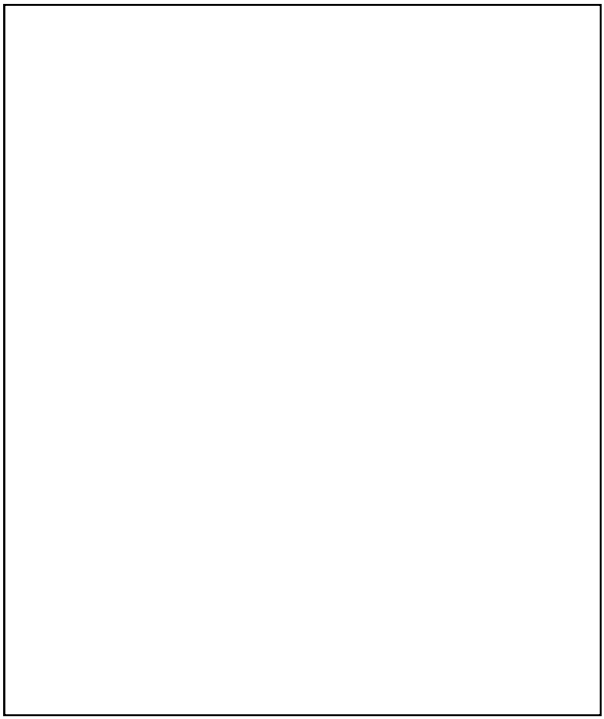
PR1. Să se scrie o secvenţă de program care va prelua într-o primă fază un număr (o cifră) de la tastatură. Apoi, programul preia de la tastatură mai multe taste și le numără. Se va afisa un mesaj corespunzator daca nr preluat iniţial e <,=,> decat nr tastelor apasate ulterior (se vor introduce maxim 20 caractere). Sugestie: folosirea întreruperii int 21h cu serviciul AH=0Ah. Exemplu: se preia un număr 5 și apoi se apasă 7 taste; astfel, se va afișa pe ecran un mesaj: "Ati apăsat un număr > de taste!" Să se afișeze mesajul pe ecran, în mijlocul ecranului. Analizaţi codul și în zonele marcate explicaţi (cât mai detaliat) modul cum s-a implementat acest program.

Rezolvare:

```
org  100h
.data
msg db "Introduceti o cifra de la tastatura: ","$"
msg1 db 0Dh,0Ah,"apasati pe taste si cand doriti sa terminati, apasati 'ESC' ...", 0Dh,0Ah, "$"
msg2 db 0Dh,0Ah, "Ati apasat",20h,"$"
msg3 db " taste, adica",20h,"$"
msg4m db "mai mult decat",20h," $"
msg4p db "mai putin decat ",20h,"$"
msg4e db "exact cat ",20h,"$"
msg5 db "ati declarat! ",20h,"$"
nr db ?
nrTaste db ?
.code
mov dx, offset msg
mov ah, 9
int 21h
mov ah, 1   ; asteapta o tasta
int 21h
sub al, 30h
mov nr, al
```

```
mov dx, offset msg1
mov ah, 9
int 21h
xor bx, bx

wait:  mov ah, 0              ; _____
       int 16h                ; _____
       cmp al, 27             ; _____
       je stop                ; _____
       mov ah, 0eh            ; _____
       int 10h

       inc bx                 ; Care e rolul acestei instructiuni?
                              _____
       mov nrTaste,bl         ; _____
    jmp wait
; afiseaza rezultatul prin mesaje
stop:  mov dx, offset msg2
       mov ah, 9
       int 21h                ; _____
       mov ax, bx
       call print_ax          ; apel functie de afisare numar format din mai multi digiti
       mov dx, offset msg3
       mov ah, 9
       int 21h
       mov al,nrTaste
       cmp al,nr
       jbe eti
       mov dx, offset msg4m
       jmp afis
    eti:
       mov al,nrTaste
       cmp al,nr
       jz etich
       mov dx, offset msg4p
       jmp afis
    etich: mov dx, offset msg4e

 afis: mov ah,9
       int 21h

       mov dx, offset msg5
       mov ah,9
       int 21h
ret                  ; exit
```

```
print_ax proc
        cmp ax, 0
        jne print_ax_r
        push ax
        mov al, '0'
        mov ah, 0eh
        int 10h
        pop ax
        ret
print_ax_r:
        pusha
        mov dx, 0
        cmp ax, 0
        je pn_done
        mov bx, 10
        div bx
        call print_ax_r
        mov ax, dx
        add al, 30h
        mov ah, 0eh
        int 10h
        jmp pn_done
pn_done:
        popa
        ret
endp
```

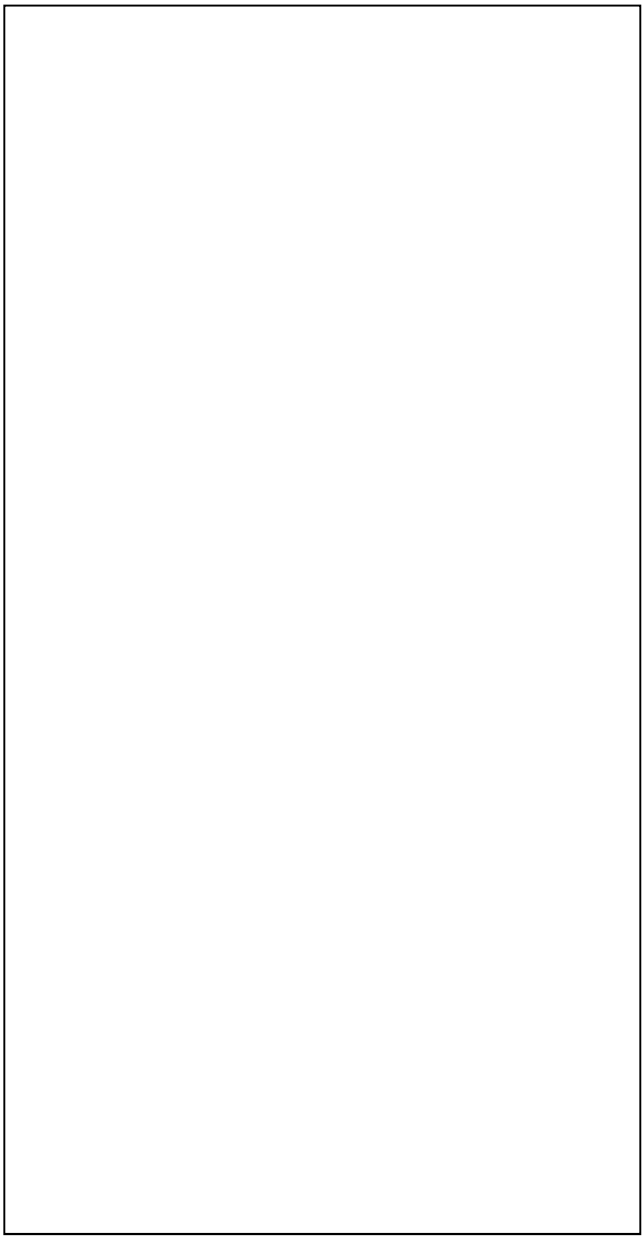


PR2. Să se scrie o secvenţă de program care să afișeze pe ecran toate cifrele, în mod repetat, de la '0' la '9' fiecare cifră cu o culoare diferită; se vor defini 2 numere (de câte o cifră fiecare) și se vor afișa pe un rând atâtea cifre cât arată primul nr, apoi va coborâ pe rândul următor și va repeta până ce ajunge la numărul de rânduri cât specifică cel de-al doilea număr. Exemplu: se preia de la tastatură (din linia de comandă, nu cu întreruperi) 6 și 9-> atunci pe ecran va apărea:
012345
678901
234567 … până ce ajunge la 9 randuri

Rezolvare:

```
org    100h
.data
mesaj db '0123456789'        ; _____
nrlinii  db 5                ; _____
nrCarPeLinie db 7            ; _____

.code
```

```
mov ax, 3
int  10h

mov dl, 0
mov dh, 0
mov bl, 0
mov si,0
jmp next_char
next_row:
   inc     dh
   cmp   dh, nrlinii
   je      stop
   mov     dl, 0
next_char:
   mov ah, 02h
   int 10h
   mov al, mesaj[si]
   inc si
   cmp si,9
   jb et
   push ax
   push bx
   mov ax,si
   mov bl,10
   div bl
   mov al, ah
   xor ah,ah
   mov si, ax
   pop bx
   pop ax
et:
   mov  bh, 0
   mov  cx, 1
   mov  ah, 09h
   int 10h
   inc bl
   inc dl
   cmp  dl, nrCarPeLinie
   je next_row
   jmp next_char
stop:
 ret
```



PR3. Să se scrie o secvenţă de program care să folosească memoria video astfel: Se va defini un șir de maxim 30 caractere (cu spaţii) în segmentul de date. Pe ecran, șirul va apărea afișat pe o singură linie. Prima dată va apărea scris tot șirul pe ecran cu o culoare roșu pe fond negru, iar apoi se va colora. Se indică rezolvarea cerinţei într-o primă variantă cu definirea mesajului direct în memoria video, folosind instructiuni mov;

Problemă propusă: se definește mesajul în segmentul de date.

Rezolvare

```
org 100h
.data
.code
mov ax, 3
int 10h              ; _____________________________________________
mov ax, 0b800h       ; _____________________________________________
mov ds, ax
mov [02h], 'C'
mov [04h], 'o'
mov [06h], 'm'
mov [08h], 'p'
mov [0ah], 'u'
mov [0ch], 't'
mov [0eh], 'e'
mov [10h], 'r'
mov [12h], ' '
mov [14h], 'A'
mov [16h], 'r'
mov [18h], 'c'
mov [1Ah], 'h'
mov [1Ch], 'i'
mov [1Eh], 't'
mov [20h], 'e'
mov [22h], 'c'
mov [24h], 't'
mov [26h], 'u'
mov [28h], 'r'
mov [2Ah], 'e'
mov [2Ch], '!'
mov cx, 22           ; _____________________________________________
mov di, 03h          ; _____________________________________________
mov dh,11001001b     ; _____________________________________________
c:  mov [di], dh     ; _____________________________________________
   add dh,10h        ; _____________________________________________
   add di, 2         ; _____________________________________________
   loop c            ; _____________________________________________
ret
```

PR4. Să se scrie o secvenţă de program în EMU8086 care va prelua un șir de caractere de la tastatură (se vor introduce între 15 și 20 caractere).
Să se adauge o secvenţă care să transforme caracterele literă mică de pe poziţie impară (șirul se consideră că începe cu primul element ca fiind cel de pe poziţia 0) în majusculă.
De exemplu, dacă șirul iniţial este 1*abcf3*, șirul rezultat va fi 1*AbCf3*. Să se afișeze șirul pe ecran, pornind din linia 3, de pe coloana 10. Se sugerează folosirea întreruperii int 21h cu serviciul 0Ah pentru preluarea caracterelor.

Rezolvare

```
org 100h  ; spune asamblorului să înceapă de la adresa 100h
.data
sir db 20, 22 dup('?')          ; alocare spaţiu de maxim 20 caractere în memorie
                                ; va transforma în majusculă literele mici de pe poziţie impară
.code
lea dx, sir          ; DX=offset sir – necesar pentru apel întrerupere int 21h
mov ah, 0ah          ; serviciul 0Ah al întreruperii int 21h
int 21h              ; se apelează întreruperea de preluare caractere de la tastatură
mov bx, dx           ; BX=offset sir – necesar la adresare
mov ah, 0 ; AH=0
mov al, ds:[bx+1]                ; în AX va fi lungimea șirului
mov si, ax                       ; necesar la instrucţiunea de mai jos
mov byte ptr [bx+si+2], '$'      ; pune '$' la sfârșitul șirului fol. adresare bazată indexată


mov cx, ax           ; CX=lungimea șirului
jcxz iesire          ; este șirul nul ? dacă DA (adică ZF=1), sare la sfârșitul programului
add bx, 2            ; sare peste caracterele de control, de la adresa 102h și 103h
```

```
prel:                ; verifică dacă e literă mică, deci dacă e în domeniul ['a';'z']
   cmp byte ptr [bx], 'a'
   jb ok             ; dacă e < 'a', sare la ok, adică nu trebuie prelucrat
   cmp byte ptr [bx], 'z'
   ja ok             ; dacă e > 'z', sare la ok, adică nu trebuie prelucrat
            ; verifică și poziţia – dacă e pe poziţie impară, se transformă în majusculă
   mov ax, bx
   ror ax,1          ; se verifică cu CF dacă poziţia e un număr impar
   jnc ok            ; dacă CF=1 => a fost un caracter de pe o poziţie număr impar
and byte ptr [bx], 11011111b ; transformă în majusculă - operaţia e similară cu scă-
                     ; derea valorii 20h din elementul de tip byte adresat cu registrul BX
ok:    inc bx        ; trece la următorul element din șir, BX e pointer la el
   loop prel         ; se va relua atât timp cât CX nu e încă nul
```

```
afisare:
; controlul cursorului cu serviciul 02h al întreruperii cu tipul 10h
mov ah,02h          ; serviciul 02h al întreruperii int 10h pentru control poziţie cursor
mov dl,10           ; coloana 10
mov dh,3            ; linia 3
mov bx,0  ; pagina video e 0
int 10h             ; apel întrerupere cu tipul 10h
; Afișarea șirului specificat cu ds:dx, terminat cu caracterul '$'
lea dx, sir+2       ; în DX e pointer la șir
mov ah, 09h         ; serviciul 09h al int 21h pentru afișare șir terminat cu '$'
int 21h             ; apel întrerupere cu tipul 21h
iesire:
        ret         ; revenire în S.O.
```

In EMU8086 se poate urmări zona de memorie prin setarea adresei în forma logică 0700h:102h. La execuţia cu EMU8086, se va observa că după adresa fizică 07118h apare codificată instrucţiunea ***lea dx, sir*** (adică prima instrucţiune din program) pe care asamblorul o transformă în ***mov dx,102h*** și o codifică apoi ca BA0201h.
La execuţia programului, cu Run[1] în EMU8086, ecranul va arăta așa ca în Figura **14.1**.

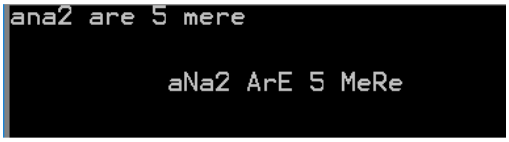


Figura 14.1. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa de la **PR4**

PR5. Să se scrie o secvenţă de instrucţiuni în EMU care va prelua un șir de caractere de la tastatură (se vor introduce între 1 și 20 caractere). Să se adauge apoi o secvenţă care să introducă după fiecare caracter (începând de la poziţia 0) din șirul iniţial, un caracter suplimentar x. De exemplu, dacă șirul iniţial este *abcdef*, șirul rezultat va fi a*xbxcxdxexfx*. Să se afișeze șirul rezultat pe ecran, pornind din linia 3, de pe coloana 10. Se sugerează folosirea întreruperii int 21h cu serviciul 0Ah pentru preluarea caracterelor de la tastatură.

Observaţii: Programul este identic cu cel de la **PR4**, cu observaţiile că:
1) s-au rezervat 40 octeţi în bufferul de la început și
2) zona de program delimitată de eticheta ***prel*** și instrucţiunea ***loop prel*** se va înlocui cu secvenţa de instrucţiuni următoare:

[1] In mod optim, programul din EMU se va executa cu Run, dar pentru a putea vizualiza valorile din memorie sau din regiștri în fiecare pas, este mai potrivită opţiunea Single Step. Se poate folosi și opţiunea Run, dar se va apăsa Step Back până la întâlnirea situaţiei ce se dorește a fi analizată.

```
        ;BX este poziţionat pe primul element din şir, iar în CX este lungimea şirului
; se parcurge şirul dinspre capăt spre început
mov si, bx          ; SI poziţionat pe primul element al şirului
add si, cx          ; SI poziţionat chiar la sfârşitul şirului, pe caracterul '$'

; modificarea octetului de control pentru a stoca noua lungime a şirului
mov ax,cx; AX=CX
add ax,ax ; AX=CX + CX
mov byte ptr ds:[bx-1],al       ; se depune valoarea 2*CX în octetul de control,
                                ; la adresa 103h va fi 30 în loc de 15; lungimea noului şir este dublă
mov di, si                      ; DI poziţionat pe '$'
add di, cx                      ; la DI se mai adună o lungime CX pentru a-l poziţiona
                                ; pe locaţia din destinaţie unde va fi plasat  '$'

intercalare:                    ; algoritmul aplicat pentru dublarea dimensiunii şirului iniţial
        mov al, byte ptr [si]           ; se ia elementul din şirul sursă în AL
        mov byte ptr [di], al           ; se plasează elementul din AL în şirul destinaţie
        dec di                          ; urcăm în destinaţie o poziţie
        mov byte ptr [di], 'x'          ; la acea locaţie depunem 'x'
        dec di                          ; mai urcăm în destinaţie o locaţie
        dec si                          ; urcăm şi în sursă o locaţie
loop intercalare                        ; se repetă pentru numărul de elemente din şir
```

Algoritmul folosit pentru a dubla dimensiunea şirului şi a-l intercala cu un alt caracter ('x' în cazul **PR6**) este prezentat în continuare; algoritmul foloseşte:

CX = nr. de elemente ale sursei;
SI = pointer la sursă, plasat pe ultimul element din sursă;
DI = pointer la destinaţie, plasat pe ultimul element al destinaţiei.

Algoritmul mută elementele din *sursă* în *destinaţie*, rând pe rând, cu ajutorul acumulatorului AL, pornind din capăt spre început (deci „următor" în acest sens).

Paşi: **I.** mută elementul curent (prin AL) din zona pointată de SI în zona pointată de DI;
II. decrementează DI pentru a pointa spre următorul element din destinaţie şi pune ‚x' în zona pointată acum de DI;
III. decrementează DI pentru a pointa spre următorul element din destinaţie, decrementează SI pentru a pointa spre următorul element din sursă şi **repetă** de un număr de ori egal cu numărul de elemente ale sursei (dat în CX).

La execuţia programului, cu Run în EMU, ecranul va arăta aşa ca în Figura **14.2**.

Figura 14.2. Afişarea pe ecran a şirului de intrare şi a celui prelucrat prin secvenţa de la **PR5**

PR6. Să se scrie o secvenţă de instrucţiuni în EMU8086 care va prelua un șir de caractere de la tastatură (se vor introduce între 1 și 20 caractere). Să se adauge apoi o secvenţă care va afișa șirul pe ecran, începând de la linia 4, coloana 10. Șirul se va afișa pe ecran colorat, scris cu roșu pe fond verde.

Diferenţa faţă de problemele anterioare constă în faptul că se cere afișarea cu o culoare anume. Din acest motiv, pentru afișare nu se va mai folosi întreruperea cu tipul 21h, serviciul 09h (pentru afișare șir terminat cu '$'), ci întreruperea cu tipul 10h, serviciul 09h care va realiza o afișare caracter cu caracter. Astfel, există posibilitatea controlării poziţiei și specificului de culoare pentru fiecare caracter în parte. Inconvenientul este că va trebui scrisă și secvenţa care va muta cursorul cu o poziţie spre dreapta după fiecare caracter afișat. Această poziţionare a cursorului se va realiza cu serviciul 02h al întreruperii cu tipul 10h.

Observaţie: Programul este identic cu cel de la problema **PR4**, până la instrucţiunea ***loop prel***. Apoi, se modifică așa cum se prezintă în continuare:

```
; se reiniţializează BX și CX
lea bx, sir+2                   ; BX va pointa spre primul element din șir
mov ch,0            ; CH=0, CL va fi luat din octetul de control
mov cl, byte ptr [bx-1]         ; în CX va fi numărul de elemente ale șirului

mov dl, 10                      ; coloana unde începe afișarea pe ecran
mov dh, 4                       ; linia unde începe Afișarea pe ecran

afisare:
  cmp byte ptr [bx],'$'         ; se verifică dacă s-a ajuns la sfârșit
  jz iesire                     ; dacă DA, salt la sfârșit program
; pentru afișare – nu se mai folosește afișare de șir terminat cu '$',
; ci Afișarea unui caracter (aflat în AL) cu serviciul 09h al întreruperii cu tipul 10h
; (în BL- atributul de culoare, iar în CX – numărul de afișări al caracterului resp.)
; și poziţionare cursor cu serviciul 02h al întreruperii cu tipul 10h
; (DL specifică linia sau rândul pe care se poziţionează cursorul, iar DH coloana)
        mov al, byte ptr [bx]       ; se preia din șir în AL cod Ascii caracter de afișat
        push bx             ; BX va fi folosit în int 10h, serviciul 02h; rol dublu
        mov ah,02h          ; serviciul folosit de int 10h pentru poziţionare cursor
        inc dl      ; deplasează cursor pe următoarea coloană
        mov bx,0            ; în BH-pagina video 0
        int 10h     ; apel întrerupere pentru poziţionare cursor
        push cx             ; similar, CX va fi folosit de int 10h, serviciul 09h; rol dublu
        mov AH,09h          ; serviciul folosit de int 10h pentru afișare caracter la cursor
        mov cx,1            ; altfel, va afișa de mai multe ori acel caracter pe ecran
        mov bl,0A4h         ; atribut de culoare 1010 0100b fond light Green, scris Red
        mov bh,0 ; pagina video 0
        int 10h             ; apel întrerupere pentru afișare caracter din AL la cursor
```

```
        pop cx              ; se reface de pe stivă CX ca nr. de elemente de prelucrat
        pop bx              ; se reface de pe stivă BX ca pointer la șir
        inc bx      ; BX va pointa spre următorul element din șir
loop afisare                ; se reia pentru câte elemente au mai rămas de afișat

iesire:  ret                ; revenire în S.O.
```

La execuţia programului, cu Run în EMU, ecranul va arăta așa ca în Figura **14.3**.

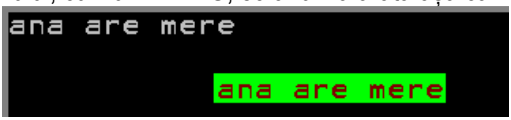


Figura 14.3. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa de la **PR6**

PR7. Să se scrie o secvenţă care să scrie fiecare caracter din șirul introdus de la tastatură în oglindă, cu o culoare roșu pe fond albastru, iar șirul iniţial cu alb pe fond negru. De exemplu, dacă șirul iniţial este *'ana are mere'*, șirul rezultat va fi *erem era ana*, care se va evidenţia în text. Să se afișeze atât șirul iniţial cât și cel rezultat pe ecran, în mijlocul ecranului.

Rezolvare:

Partea de început a programului este identică cu cea de la **PR5**, până la eticheta prel. Nu este necesară nici o prelucrare în program, deoarece se va afișa direct în ordine inversă, pe măsură ce este parcurs șirul dinspre capăt spre început.

La execuţia programului, cu Run în EMU, ecranul va arăta așa ca în Figura **14.4**.

Figura 14.4. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa de la **PR7**

Observaţie: Partea de afișare este aproape identică celei de la problema **PR6**, exceptând faptul că în locul instrucţiunii ***inc bx***, de data aceasta s-a folosit ***dec bx*** întrucât șirul se parcurge acum dinspre capăt spre început. Partea de afișare se va înlocui deci cu secvenţa următoare:

```
add bx, cx   ;la pointerul BX poziţionat pe primul ement din șir se adună lungimea șirului din CX
dec bx       ; se decrementează BX spre a pointa spre ultimul element din șir

mov dl, -1 ; necesar pt coloana 0, întrucât se va incrementa chiar de la prima afișare
mov dh,2             ; necesar pentru linia 2 de unde începe afișarea
```

```
afisare:
        mov al, byte ptr [bx]      ; se preia din șir în AL cod Ascii caracter de afișat
        push bx                    ; BX va fi folosit în int 10h, serviciul 02h; rol dublu

        mov ah,02h      ; serviciul folosit de int 10h pentru poziţionare cursor
        inc dl          ; deplasează cursor pe următoarea coloană
        mov bx,0        ; în BH-pagina video 0
        int 10h         ; apel întrerupere pentru poziţionare cursor

        push cx         ; similar, CX va fi folosit de int 10h, serviciul 09h; rol dublu

        mov AH,09h      ; serviciul folosit de int 10h pentru afișare caracter la cursor
        mov cx,1        ; altfel, va afișa de mai multe ori acel caracter pe ecran
        mov bl,0F4h     ; atribut de culoare 1111 0100b fond alb luminos, scris Red
        mov bh,0        ; pagina video 0
        int 10h         ; apel întrerupere pentru afișare caracter din AL la cursor

        pop cx          ; se reface de pe stivă CX ca nr. de elemente de prelucrat
        pop bx          ; se reface de pe stivă BX ca pointer la șir
        dec bx          ; BX va pointa spre elementul următor de prelucrat
loop afisare            ; se reia pentru câte elemente au mai rămas de afișat

iesire: ret             ; revenire în S.O.
```

14.2. Probleme Propuse

Set 1

Problema 14.1.1. Să se scrie o secvenţă de program care să afișeze un caracter pe ecran. a) Definiţi caracterul direct în registru, iar apoi în memorie (ca o variabilă de tip byte). b) Transformaţi secvenţa de la a) astfel încât caracterul să fie preluat de la tastatură. Testaţi mai multe variante de *preluare a caracterului de la tastatură*. c) Testaţi mai multe variante de *afișare a caracterului pe ecran*.

Sugestie:

a) Pentru afișarea unui caracter pe ecran există mai multe posibilităţi - pentru rezolvarea cerinţei de la punctul a) se poate opta pentru **folosirea întreruperii int 21h cu serviciul 02h**

b) In locul definirii datelor în regiștri sau în memorie, se poate opta pentru preluarea acestora de la tastatură. Pentru aceasta, se pot folosi mai multe variante:

b1) cu int 16h, serviciul 0h

b2) cu int 21h, serviciul 1h

c) pentru afișarea caracterului pe ecran: există mai multe variante:

c1) int 21h cu serviciul 02h;

c2) int 10h cu serviciul 0eh;

c3) int 10h cu serviciul 09h.

Figura 14.5. Rezultatul execuţiei programului a) și b) cu afișarea caracterului 'A' pe ecran (figura din stânga e valabilă și pentru preluare fără ecou)

Figura 14.6. Posibilitatea afișării colorate a caracterului prin utilizarea întreruperii int 10h cu serviciul 09h, cu parametrul a) CX=1; b) CX=5

Problema 14.1.2. Să se scrie o secvenţă de program care să afișeze un mesaj pe ecran. Pentru simplitate, mesajul va fi definit ca șir de caractere în segmentul de date.

Sugestie:Pentru rezolvarea problemei se pot utiliza mai multe variante:

Varianta I) cu int 21h, serviciul 09h;
Varianta II) cu int 21h, serviciul 02h;
Varianta III) cu int 10h, serviciul 0Eh ;
Varianta IV) cu int 10h, serviciul 09h;
Varianta V) folosind lungime șir în CX și oricare din variantele I)...IV).

Figura 14.7. Execuţia programului cu așteptare de tastă și fără coborâre de cursor (stânga), respectiv cu coborâre de cursor (dreapta) folosind CR urmat de LF

Figura 14.8. Posibilitatea afișării colorate a fiecărui caracter prin utilizarea întreruperii int 10h cu serviciul 09h

Figura 14.9. Posibilitatea afișării multiple a fiecărui caracter prin utilizarea întreruperii int 10h cu serviciul 09h

Problema 14.1.3. Să se scrie o secvenţă de program care să preia de la tastatură un șir de caractere, să depună acest șir în memorie și apoi să-l afișeze după modelul prezentat la Problema 14.1.2 variantele II), ..., V).

Sugestie: Pentru rezolvarea problemei se propun mai multe moduri de lucru:

Modul I) cu int 21h, serviciul 0Ah pentru ***preluare șir de caractere;***
Modul II) cu int 16h, serviciul 0h, sau
cu int 21h, serviciul 01h, sau
cu int 21h, serviciul 08h pentru ***preluare caracter.***

Problema 14.1.4. Să se scrie o secvenţă de program care să preia de la tastatură un șir de caractere, să depună acest șir în memorie și apoi să-l afișeze după modelul prezentat la Problema 14.1.2 varianta l).

Problema 14.1.5. a) Să se scrie o secvenţă de program care să preia de la tastatură 2 numere în baza zece fără semn (reprezentate pe câte un singur digit), să depună aceste numere în memorie, să calculeze suma lor și apoi să afișeze această sumă pe ecran. Pentru a se afișa corect, se vor introduce numere astfel încât suma să nu depășească valoarea 9. b) Rescrieţi problema folosind subrutine; c) Rezolvaţi aceeași problemă de mai sus, dar cu macrouri în locul subrutinelor, iar în locul preluării datelor prin întreruperi, folosiţi preluarea din linia de comandă (cu PSP).

Set 2

Următoarea cerinţă este valabilă pentru toate problemele din acest set:
Să se scrie o secvenţă de program care va prelua un șir de caractere de la tastatură (se vor introduce între 15 și 20 caractere). Se sugerează folosirea întreruperii int 21h cu serviciul 0Ah pentru preluarea caracterelor. Elementele șirului se vor modifica în memorie (precum se specifică în cerinţa fiecărei probleme), iar rezultatul obţinut se va afișa folosind serviciul 09h al întreruperii int 21h pentru afișare șir terminat cu caracterul '$'; șirul se va afișa pe ecran pe orizontală, pornind din linia 3, de pe coloana 10.

Problema 14.2.1. Să se scrie o secvenţă care să transforme caracterele literă mică de pe poziţie impară (șirul se consideră că începe cu primul element ca fiind cel de pe poziţia 0) în majusculă. Transformarea se va putea urmări inclusiv în zona din memorie. De exemplu, dacă șirul iniţial este 1*abcf3*, șirul rezultat va fi 1*AbCf3*.
La execuţia programului, în modul Run[2], pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul obţinut va fi:

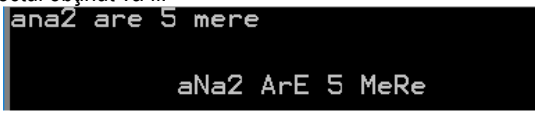


Figura 14.10. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa de la **Problema 14.2.1**

Problema 14.2.2. Să se scrie o secvenţă de program astfel încât transformarea să se realizeze invers celei din Problema 14.2.1: din majusculă să se transforme în literă mică; transformarea se va aplica asupra elementelor din șir de pe orice poziţie număr par. Se presupune că se vor introduce de la tastatură mai multe caractere litere mari (cu tasta Shift apăsată sau folosind Caps Lock).
Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

[2] In mod optim, programul din simulator se va executa cu Run, dar pentru a putea vizualiza valorile din memorie sau din regiștri în fiecare pas, este mai potrivită opţiunea Single Step. Se poate folosi și opţiunea Run, dar se va apăsa Step Back până la întâlnirea situaţiei ce se dorește a fi analizată.

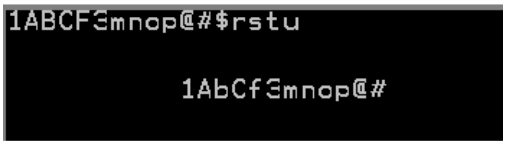


Figura 14.11. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.2.2**

Problema 14.2.3. Să se scrie o secvenţă de program astfel încât elementele din șirul original de pe o poziţie multiplu de 3 să fie înlocuite de un caracter 'x'.
Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

Figura 14.12. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.2.3**

Problema 14.2.4. Să se scrie o secvenţă de program pentru a prelucra cifrele care apar în șir astfel: la fiecare cifră introdusă în șirul iniţial să se adune 1. Prelucrările se vor opera asupra tuturor elementelor din șir, indiferent că sunt pe poziţie număr impar sau număr par. Se presupune că se vor introduce de la tastatură mai multe caractere de tip cifră zecimală (în baza 10, deci 0, ..., 9). Soluţia propusă va implementa un sistem modulo 10.
Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

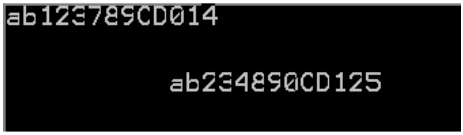


Figura 14.13. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.2.4**

Problema 14.2.5. Să se scrie o secvenţă de program pentru a prelucra cifrele hexazecimale care apar în șir astfel: la fiecare cifră din șirul iniţial să se adune 1. Prelucrările se vor opera asupra tuturor elementelor din șir, indiferent că sunt pe poziţie număr impar sau număr par. Se presupune că se vor introduce de la tastatură mai multe caractere de tip cifre hexazecimale (în baza 16, deci 0, ..., 9, A, B, C, D, E, F). Soluţia propusă va implementa un sistem modulo 16.
Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

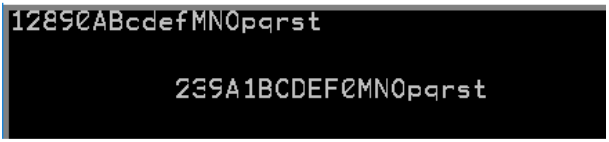


Figura 14.14. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.2.5**

Problema 14.2.6. Să se scrie o secvenţă de program astfel încât literele mici din șirul original să fie transformate în majusculă, iar literele mari în minusculă. Orice alte caractere nu vor fi afectate.

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

Figura 14.15. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.2.6**

Problema 14.2.7. Să se scrie o secvenţă de program astfel încât toate vocalele (‘a’, ‘e’, ‘i’, ‘o’, ‘u’) din șirul original să fie eliminate (înlocuite cu caracterul spaţiu).

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

Figura 14.16. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.2.7**

Problema 14.2.8. Să se scrie o secvenţă de program a.î. transformarea șirului iniţial să se realizeze astfel: din cifre *în cifra cu 2 mai puţin*, din literă mică *în majuscula corespunzătoare codului Ascii cu 1 mai mult*, iar din literă mare *în minuscula corespunzătoare codului Ascii cu 2 mai mult*. Fiecare subsistem va fi implementat într-un buffer circular, iar orice alte caractere nu vor fi afectate. Se presupune că se vor introduce de la tastatură și caractere litere mari (cu tasta Shift apăsată sau folosind Caps Lock).

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

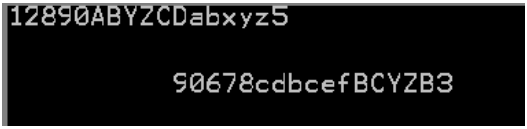


Figura 14.17. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.2.8**

Problema 14.2.9. Să se scrie o secvenţă de program astfel încât caracterele de pe poziţiile extreme din șir să sufere o transformare: primele 3 caractere să fie înlocuite de un caracter 'x', iar ultimele 5 caractere să fie înlocuite de un caracter 'y'.

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

Figura 14.18. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.2.9**

Problema 14.2.10. Să se scrie o secvenţă de program astfel încât caracterele de pe poziţiile mediane din șir să sufere o transformare, și anume: cele 3 caractere (spre capătul din stânga) și respectiv cele 5 caractere (spre capătul din dreapta) raportat la mijloc să sufere aceeași transformare ca la Problema 14.2.9.

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figurile următoare, efectul va fi:

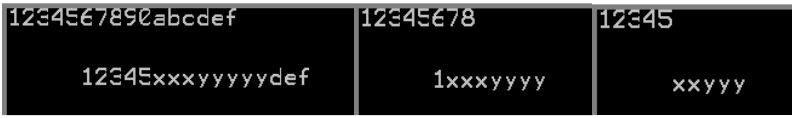


Figura 14.19. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.2.10**

Problema 14.2.11. Să se scrie o secvenţă de program care să scrie fiecare caracter din șirul introdus de la tastatură în oglindă. De exemplu, dacă șirul inițial este *'ana are mere'*, șirul rezultat va fi *erem era ana*.

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figurile următoare, efectul va fi:

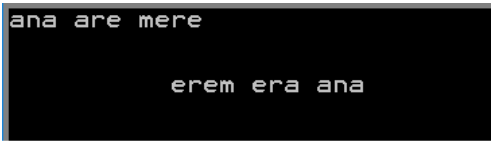


Figura 14.20. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.2.11**

Problema 14.2.12. Să se scrie o secvenţă de program astfel încât dându-se un șir de caractere (fără spaţii), să se construiască din el un posibil palindrom folosind prima jumătate a cuvântului. Prin compararea celor două șiruri (cel original și cel prelucrat) utilizatorul le va putea vizualiza și va putea decide dacă e palindrom sau nu.

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

Figura 14.21. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.2.12**

Problema 14.2.13. Să se scrie o secvenţă de program care să verifice dacă șirul introdus de la tastatură este palindrom sau nu. Programul va afișa 0 în caz că nu este palindrom, respectiv 1 dacă este. De exemplu, dacă șirul iniţial este *'cojac'*, se va afișa 0, iar dacă șirul iniţial este 'cojoc' se va afișa 1.

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figurile următoare, efectul va fi:

Figura 14.22. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.2.13**

Problema 14.2.14. Să se scrie o secvenţă care să verifice dacă în șirul introdus de la tastatură apare un anumit caracter, de exemplu spaţiu (cod Ascii 20h). Programul va elimina toate aceste caractere din șir (operare și în memorie) și va afișa noul șir pe ecran. De exemplu, dacă șirul iniţial este *'ana are mere'*, se va afișa 'anaaremere' și se va putea urmări și în memorie modificat.

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figurile următoare, efectul va fi:

Figura 14.23. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.2.14**

Set 3:

Cerinţă valabilă pentru toate problemele din acest set:

Să se scrie o secvenţă de program care va prelua un șir de caractere de la tastatură (se vor introduce între 15 și 20 caractere).

Se sugerează folosirea întreruperii int 21h cu serviciul 0Ah pentru preluarea caracterelor. Şirul se va modifica în memorie (așa cum se specifică în cerinţa fiecărei probleme), inclusiv ca dimensiune, nu doar ca și conţinut.

Rezultatul obţinut se va afișa folosind serviciul 09h al întreruperii int 21h pentru afișare șir terminat cu '$'; șirul se va afișa pe ecran pornind din linia 3, de pe coloana 10.

Problema 14.3.1. Să se scrie o secvenţă care să introducă după fiecare caracter (începând de la poziţia 0) dintr-un șir preluat de la tastatură, un caracter suplimentar x. De exemplu, dacă șirul iniţial este *abcdef*, șirul rezultat va fi a*x*b*x*c*x*d*x*e*x*f*x*.

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

Figura 14.24. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa de la **Problema 14.3.1**

Problema 14.3.2. Să se scrie o secvenţă de program care să introducă un număr de caractere spaţiu după fiecare caracter din șirul iniţial; acest nr. va depinde de ordinul poziţiei caracterului respectiv. De exemplu, dacă șirul iniţial este *A12Edcba*, șirul rezultat va fi A*x*1*xx*2*xxx*E*xxxx*d*xxxxx*c*xxxxxx*b*xxxxxxx*a*xxxxxxxx* (cu spaţiu în loc de x, și au fost subliniate caracterele nou inserate).

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

Figura 14.25. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.3.2**

Problema 14.3.3. Să se scrie o secvenţă care să citească de la tastatură un număr de maxim 5 cifre zecimale. Să se formeze un șir în care în faţa fiecărei cifre să se depună toate cifrele de la 0 începând până la cifra respectivă. De exemplu, dacă șirul iniţial este 84 șirul rezultat va fi 0123456780123*4* (au fost subliniate caracterele nou inserate).

Observaţie: La execuţia programului, efectul va fi:

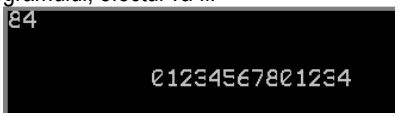


Figura 14.26. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.3.3**

Problema 14.3.4. Să se scrie o secvenţă de program care să introducă după fiecare caracter din șirul iniţial preluat de la tastatură, un caracter nou, și anume caracterul având codul Ascii cu 1 în plus faţă de cel curent. De exemplu, pentru *aBce12fyz*, șirul rezultat va fi a*b*B*C*c*d*e*f*1*2*2*3*f*g*y*z*z*{* (au fost subliniate caracterele nou inserate). *Sugestie:* se va parcurge șirul dinspre capăt spre început: se va prelucra elementul „următor” (în sensul specificat), adică elementul aflat la locaţia cu 1 mai puţin. Se poate spune că „se urcă” în zona din memorie.

Observaţie: La execuţia programului, efectul va fi:

Figura 14.27. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.3.4**

Problema 14.3.5. Să se scrie o secvenţă care să aplice algoritmul de inserare folosind un cod Ascii cu 1 mai mare asupra literelor mici din șir. Toate celelalte caractere vor avea inserat același cod Ascii. De ex., *AB12cdxyz* se va transforma în *AABB1122cddexyyzza*. Se va aplica același principiu de buffer circular ca și la problemele propuse în capitolul anterior (literele mici nu vor putea fi înlocuite decât tot cu litere mici).

Observaţie: La execuţia programului, efectul va fi:

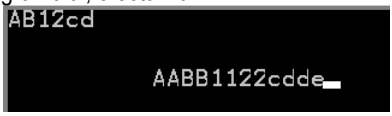


Figura 14.28. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.3.5**

Problema 14.3.6. Să se scrie o secvenţă de program care să aplice algoritmul de inserare folosind un cod Ascii cu 3 mai mare asupra literelor mari din șir. Toate celelalte caractere vor avea inserat același cod Ascii. De exemplu, *AB12cd* se va transforma în *ADBEZC1122ccdd*. Se va aplica același principiu de buffer circular ca și la problemele propuse în capitolul anterior (literele mari vor fi înlocuite tot cu litere mari).

Observaţie: La execuţia programului, efectul va fi:

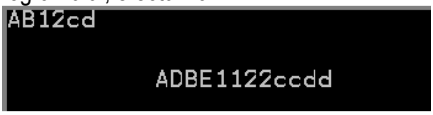


Figura 14.29. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.3.6**

Problema 14.3.7. Să se scrie o secvenţă care să aplice algoritmul de inserare folosind un cod Ascii cu 2 mai mic asupra cifrelor din șir. Toate celelalte caractere vor avea inserat același cod Ascii. De exemplu, *AB12cd* se va transforma în *AABB1920ccdd*. Se va aplica același principiu de buffer circular ca și la problemele propuse în capitolul anterior (cifrele zecimale vor fi înlocuite tot cu cifre în sistem modulo 10).

Observaţie: La execuţia programului, efectul va fi:

Figura 14.30. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.3.7**

Problema 14.3.8. Să se scrie o secvenţă care să insereze pentru fiecare caracter din șir următoarele 2 coduri Ascii. De exemplu, dacă șirul introdus este *AB12cd*, se va transforma în *ABCBCD123234cdedef* (au fost subliniate caracterele nou inserate). Se va aplica același principiu de buffer circular ca și la problemele propuse în capitolul anterior (literele mari vor fi înlocuite tot cu litere mari, cele mici tot cu litere mici, iar cifrele considerate în sistem zecimal tot cu cifre).

Figura 14.31. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.3.8**

Set 4 Cerinţă comună:

În cazul în care se cere preluarea unui șir de caractere de la tastatură, se vor introduce între 1 și 20 caractere și se vor prelua cu int 21h, serviciul 0Ah.

Problema 14.4.1. Să se scrie o secvenţă care va afișa șirul pe ecran, începând de la linia 4, coloana 10. Șirul se va afișa pe ecran colorat, scris cu roșu pe fond verde.

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

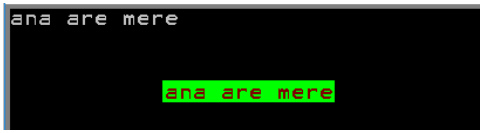


Figura 14.32. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa de la **Problema 14.4.1**

Problema 14.4.2. Să se scrie o secvenţă care să preia de la tastatură un șir de caractere (se va salva în zona de memorie rezervată), iar apoi se va afișa pe ecran astfel: fiecare caracter va fi situat pe o linie nouă, la început de rând, pornind de la coordonatele (1,0). Culoarea la afișare va fi: scris cu roșu pe fond obţinut din combinarea culorii roșu cu albastru. Obţineţi toate cele 4 situaţii rezultate prin setarea luminozităţii pentru fond, respectiv pentru culoarea de scris. Comparaţi cele patru situaţii vizual.

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figurile următoare, efectul va fi:

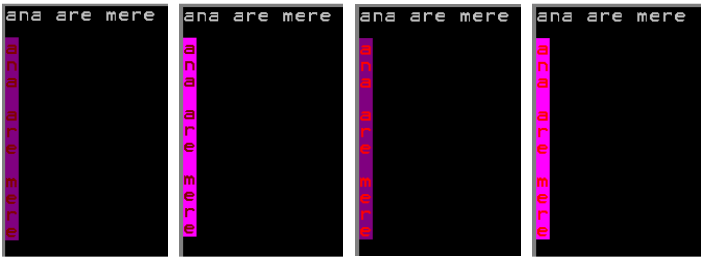


Figura 14.33. Afișarea pe ecran a șirului de intrare și a celui prelucrat la afișare

Problema 14.4.3. Să se scrie o secvenţă care să preia de la tastatură un șir de caractere (se va salva în zona de memorie rezervată), iar apoi se va afișa pe ecran astfel: se va aștepta apăsarea unei taste pentru a se afișa șirul pe ecran, iar apoi - după fiecare apăsare de tastă, poziţia de început pe orizontală se va incrementa cu 10. În total, se va aștepta apăsarea a 5 taste.

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

Figura 14.34. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.4.3**

Problema 14.4.4. Scrieţi o secvenţă de instrucţiuni prin care să se umple pe ecran, pe orizontală, dar până la jumătatea ecranului cu un text preluat de la tastatură. Şirul se va afișa pe ecran cu un spaţiu liber în mod repetat, până la apăsarea unei taste (atunci se va opri din afișare și va reveni în sistemul de operare).

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

Figura 14.35. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.4.4**

Problema 14.4.5. Să se scrie o secvenţă care să preia de la tastatură un șir de caractere (se va salva în zona de memorie rezervată), iar apoi se va afișa pe ecran, pe diagonala principală, deci pe poziţiile (1,1), (2,2), (3,3), ... câte un caracter pe o linie, de culoare verde pe fond mov luminos.

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

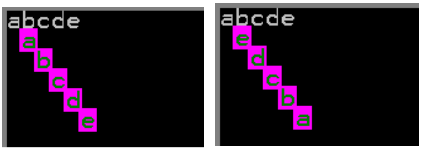


Figura 14.36. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.4.5 (stânga) și Problema 14.4.6 (dreapta)**

Problema 14.4.6. Să se scrie o secvenţă care să preia de la tastatură un șir de caractere (se va salva în zona de memorie rezervată), iar apoi se va afișa pe ecran, pe diagonală, dar de la capăt spre început, deci pe poziţiile ..., (3,3), (2,2), (1,1) câte un caracter pe o linie, de culoare verde pe fond mov luminos.

Problema 14.4.7. Să se scrie o secvenţă care să scrie fiecare caracter din șirul introdus de la tastatură în oglindă, cu o culoare roșu pe fond alb, iar șirul iniţial cu culorile implicite. De exemplu, dacă șirul iniţial este *'ana are mere'*, șirul rezultat va fi *erem era ana,* care se va evidenţia pe ecran. Să se afișeze atât șirul iniţial cât și cel rezultat pe ecran (șirul nu se modifică în memorie, ci doar la afișare).

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

Figura 14.37. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa de la **Problema 14.4.7**

Problema 14.4.8. Să se scrie o secvenţă de instrucţiuni prin care să se formeze pe ecran un pătrat plin: se pornește din centru cu valoarea 0 și apoi în sensul acelor de ceas, începând de sus, se incrementează această valoare până se ajunge la 8. Scrieţi algoritmul într-o subrutină și folosiţi doar instrucţiuni *inc* și *dec* pentru deplasarea cursorului.

Observaţie: La execuţia programului, pentru valorile impuse în cerinţă, efectul va fi:

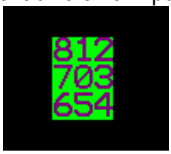


Figura 14.38. Afișarea pe ecran a rezultatului secvenţei propuse la **Problema 14.4.8**

Problema 14.4.9. Să se scrie o secvenţă de program care să scrie un șir de cifre pe ecran după următorul algoritm: la început, X=0; se va poziţiona cursorul în mijlocul ecranului, se afișează caracterul corespunzător valorii lui x și apoi:

```
I. Scade din linie pe x        ; pentru poziţia de sus
Afișează caracterul;
II. Adună la linie pe x        ; pentru poziţia din dreapta
Adună la coloană pe x
Afișează caracterul;
III. Adună la linie pe x       ; pentru poziţia de jos
Scade din coloană pe x
Afișează caracterul;
IV. Scade din linie pe x       ; pentru poziţia din stânga
Scade din coloană pe x
Afișează caracterul;
```

Incrementează pe x și reia algoritmul de 4 ori. Se cere implementarea algoritmului menţionat mai sus într-o subrutină.

Observaţie: La execuţia programului, pentru valorile impuse în cerinţă, efectul va fi:

Figura 14.39. Afișarea pe ecran a rezultatului secvenţei propuse la **Problema 14.4.9**

Problema 14.4.10. Să se scrie o secvenţă de instrucţiuni care să scrie fiecare cuvânt dintr-o propoziţie cu literă mare. De exemplu, dacă șirul inițial este *'ana are mere'*, șirul rezultat va fi *'Ana Are Mere'*. Se va presupune ca primul caracter trebuie sa fie o literă mare, fiind început de propoziţie. Să se afișeze șirul rezultat pe ecran, începând de la linia 4, coloana 5, cu roșu pe fond verde.

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

Figura 14.40. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.4.10**

Problema 14.4.11. Să se scrie o secvenţă de program care să afișeze fiecare cuvânt pe o linie nouă. De exemplu, dacă șirul inițial este *ana are mere*, cuvântul *ana* va fi pe o linie, *are* pe linia următoare, iar *mere* pe ultima linie; afișarea va continua pe linia următoare de la coloana din linia curentă. Să se afișeze șirul rezultat pe ecran, începând de la linia 4, coloana 10, cu roșu pe fond verde.

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

Figura 14.41. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.4.11**

Problema 14.4.12. Să se adapteze Problema 14.4.11 astfel încât să se afișeze șirul rezultat pe ecran, începând de la linia 1, coloana 0. Fiecare cuvânt se va afișa la începutul unei linii, iar spaţiile nu se vor afișa pe ecran.

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

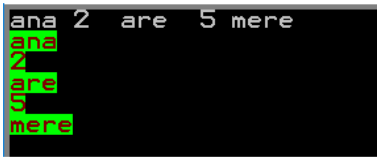


Figura 14.42. Afisarea pe ecran a sirului de intrare si a celui prelucrat prin secventa de la **Problema 14.4.12**

Problema 14.4.13. Să se adapteze Problema 14.4.11 astfel încât să se afișeze șirul pe ecran de la linia 1, coloana 1. Fiecare cuvânt se va afișa pe o linie nouă, spaţiile nu se vor afișa pe ecran, iar cuvintele vor începe pe diagonala principală.

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

Figura 14.43. Afisarea pe ecran a sirului de intrare si a celui prelucrat prin secventa de la **Problema 14.4.13**

Problema 14.4.14. Să se scrie o secvenţă de program care să scrie doar primul cuvânt (al unui șir de caractere preluat de la tastatură) cu litere mari (toate din acel cuvânt) și pe un rând separat.

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

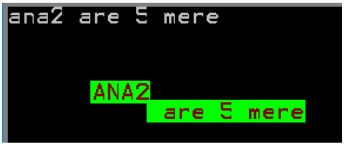


Figura 14.44. Afisarea pe ecran a sirului de intrare si a celui prelucrat prin secventa de la **Problema 14.4.14**

Set 5

Cerinţă comună:

În cazul în care se cere preluarea unui șir de caractere de la tastatură, se vor introduce între 1 și 20 caractere și se vor prelua folosind întreruperea int 21h cu serviciul 0Ah.

Problema 14.5.1. Să se scrie o secvenţă de program care să afișeze pe 2 coloane separate caracterele literă mare și cele literă mică dintr-un șir introdus de la tastatură. De exemplu, dacă șirul inițial este A*NA2 are 5MERErosii,* șirul afișat pe prima coloană va fi *ANAMERE*, iar șirul afișat pe cea de-a doua coloană va fi *arerosii*. „Coloana" va fi interpretat ca distanţiere de 10 caractere pe linie.

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

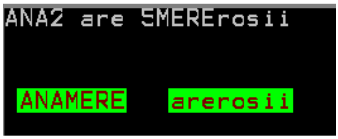


Figura 14.45. Afisarea pe ecran a sirului de intrare si a celui prelucrat prin secventa de la **Problema 14.5.1**

Problema 14.5.2. Să se modifice secvenţa de la Problema 6.1 a.î. să afișeze pe 3 coloane separate caracterele literă mare, cele literă mică și cifrele dintr-un șir introdus de la tastatură. Caracterele vor fi afișate câte unul pe fiecare coloană, în jos, deplasat cu câte o poziţie, în diagonală.

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

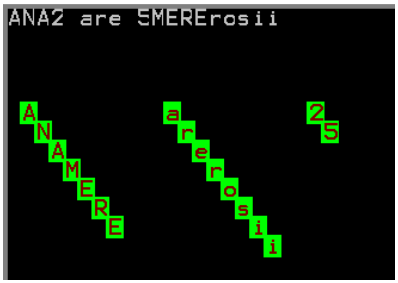


Figura 14.46. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.5.2**

Problema 14.5.3. Adaptaţi Problema 6.2 astfel încât caracterele să fie afișate câte unul pe fiecare coloană, în jos, fără deplasare (unul sub altul).

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

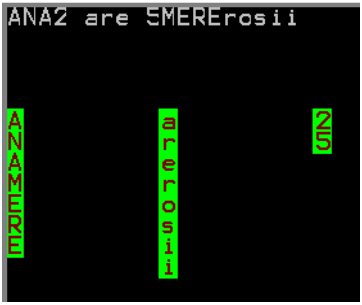


Figura 14.47. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema** 14.5.3

Problema 14.5.4. Să se scrie o secvenţă de instrucţiuni care să afișeze pe ecran un șir preluat de la tastatură, dar să schimbe atributul de culoare la fiecare caracter afișat.
Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

Figura 14.48. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema** 14.5.**4**

Problema 14.5.5 Să se scrie o secvenţă de instrucţiuni care să afișeze pe ecran un șir definit în segmentul de date, tot cu schimbarea atributului de culoare la fiecare caracter afișat, așa ca în Problema 14.4, dar folosind memoria video.
Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

Figura 14.49. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.5.5**

Problema 14.5.6. Să se modifice Problema 14.5 astfel încât să scrie pe ecran șirul câte un caracter pe o linie. În plus, mesajul nu va mai fi definit în segmentul de date, ci se va prelua de la tastatură.
Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

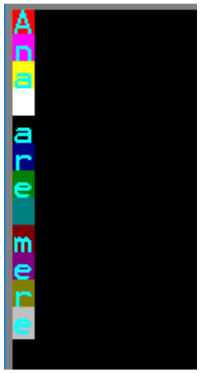


Figura 14.50. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.5.6**

Problema 14.5.7. Să se scrie o secvenţă de instrucţiuni care să scrie fiecare cuvânt al unui mesaj preluat de la tastatură cu o culoare nouă: se va afișa pe aceeași linie, dar primul cuvânt va avea o culoare, cel de-al doilea o altă culoare, ș.a.m.d. Se cere ca spaţiul să nu fie colorat, doar cuvintele (despărţite de spaţii).

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

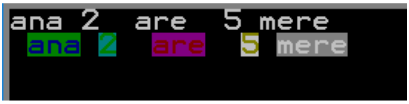


Figura 14.51. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.5.7**

Problema 14.5.8. Să se modifice secvenţa de la Problema 14.7 a.î. fiecare primă literă a fiecărui cuvânt să fie scrisă cu o culoare albastru pe fond roșu luminos (se va evidenţia cu o altă culoare-aceeași primul caracter din cadrul fiecărui cuvânt).

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

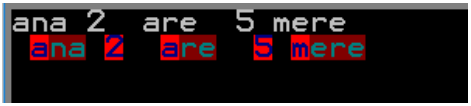


Figura 14.52. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.5.8**

Problema 14.5.9. Să se scrie o secvenţă de program care să scrie cu o culoare distinctă spaţiul din cadrul unui șir de caractere preluat de la tastatură.

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

Figura 14.53. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.5.9**

Problema 14.5.10. Să se scrie o secvenţă de program care să scrie fiecare literă mică dintr-un șir introdus de la tastatură, cu o culoare roșu pe fond albastru; celelalte caractere vor fi scrise cu o altă culoare.

Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

Figura 14.54. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.5.10**

Problema 14.5.11. Să se scrie o secvenţă de program a.î. primul cuvânt al unui șir introdus de la tastatură, să fie scris în oglindă, cu o culoare roșu pe fond albastru.
Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

Figura 14.55. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.5.11**

Problema 14.5.12. Se definesc în segmentul de date 2 numere. Să se scrie o secvenţă care să afișeze pe ecran toate cifrele, urmate de toate literele mici si apoi toate literele mari din alfabet, fiecare caracter cu o culoare diferită; se vor afișa pe un rând atâtea caractere cât arată primul nr și vor scrie atâtea rânduri cât arată cel de-al doilea nr. Dacă se ajunge la sfârșit, se reia din nou.
Observaţie: La execuţia programului, pentru valorile tastate de utilizator așa cum reiese din figura următoare, efectul va fi:

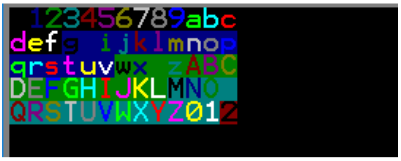


Figura 14.56. Afișarea pe ecran a șirului de intrare și a celui prelucrat prin secvenţa propusă la **Problema 14.5.12**

Capitolul 15. Concluzii

Scopul prezentului material a fost de a înţelege principalele aspecte arhitecturale ale procesoarelor din familia 80x86 prin utilizarea preponderentă a simulatorului EMU8086, dar și prin scrierea de programe direct pe procesor (capitolul 13).

S-au prezentat exemple de aplicaţii, folosind dispozitive externe virtuale gata implementate, precum afişaj cu led-uri, intersecţie gestionată de semafoare, etc.

S-au abordat exemple diverse din tutorial, unele având un grad de dificultate mai ridicat. La finalul fiecarei aplicaţii, s-au propus exerciţii pentru verificarea și validarea informaţiei asimilate.

Programele prezentate au avut rolul:

De a asigura acomodarea cu:
- interpretarea datelor fără semn vs cu semn
- folosirea corectă a regiștrilor dpdv a dimensiunii – la încărcarea datelor
- Interpretarea corectă a flagurilor în urma execuţiei instrucţiunilor cele mai generale – mai ales acolo unde apar depășiri

De a sublinia importanţa:
- *tipului* (in, out sau in/out) și
- *dimensiunii* (8b, 16b, …) porturilor folosite în aplicaţii

De a ilustra importanţa *folosirii codurilor ASCII: se pot folosi 3 forme de scriere a acestora în general: zecimal (52), hexazecimal (34h), cu apostroafe (‘4’).*

De a încuraja scrierea de programe cât mai variate, atât cu simulatorul cât și cu pachetul TASM în vederea interacţionării cu utilizatorul.

ANEXA 1. Alte programe din EMU ca model

print_char_by_char.asm

Afișare simplă a unui șir de caractere preluat de la tastatură, dar pe coloană în jos, așa cum prezintă Figura A1.1.

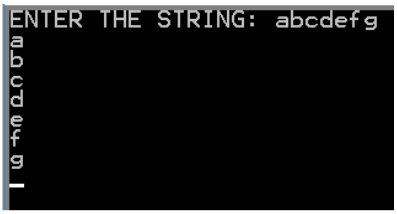


Figura A1.1. Fereastra aplicaţiei *print_char_by_char.asm*

Reverse.asm

Programul inversează un șir de caractere, așa cum arată Figura A1.2.

Figura A1.2. Fereastra aplicaţiei *palindrome.asm*

encrypt_subrotine.asm

Programul folosește o subrutină pentru a cripta/ decripta caracterele literă mică a unui șir de caractere, așa cum prezintă Figura A1.3; se folosește tabel de translatare și instrucţiunea xlat.

Figura A1.3. Fereastra aplicaţiei *encrypt.asm*

Aplicaţii folosite la diverse conversii

print_AL.asm

Aplicaţia foloseşte trei subrutine definite în program, acestea fiind denumite **print_al, print_al_bin** și **print_nl.** Primele două sunt folosite pentru a afișa pe ecran valoarea din registrul AL scrisă în zecimal fără semn, respectiv cea de-a doua pentru a afișa numărul în binar. Subrutina ***print_nl,*** care în programul principal este apelată imediat după ce s-a afișat numărul în zecimal pe ecran este utilizată pentru a coborâ pe linie nouă pe ecran. Pentru a afișa valoarea în zecimal, programul folosește o funcţie recursivă care se oprește atunci când în registrul AL este 0, adică se împarte în mod succesiv numărul la 10, noul cât devenind deîmpărţit, iar dacă se obţine câtul 0, s-a terminat bucla. Restul obţinut este plasat pe stivă și de acolo este luat apoi la afișare (întrucât era nevoie de o inversare a ordinii resturilor obţinute).

O problemă asemănătoare este ***print_AX.asm*** care adaptează subrutinele la valori mai mari ale regiștrilor.

print_AL.asm, *respectiv* ***print_AX.asm***

Figura A1.4. Fereastra aplicaţiei *print_AL.asm, respectiv print_AX.asm*

print_hex_digit.asm

Aplicaţia foloseşte o tabelă de translatare (de forma table db '0123456789abcdef') pentru a afișa valoarea din registrul DL în hexazecimal pe ecran.

Figura A1.5. Fereastra aplicaţiei *print_hex_digit.asm*

bin2dec.asm

Aplicaţia preia de la tastatura un număr scris în binar, deci preia 8 valori binare, convertește acest număr în zecimal și apoi îl afișează pe ecran.

Figura A1.6. Fereastra aplicaţiei *bin2dec.asm*

convert_string_number_to_binary.asm

Aplicaţia convertește un număr preluat de la tastatură ca număr în zecimal, îl transformă în binar și îl afișează pe ecran; este un program inspirat din macroul "scan_num" din c:\emu8086\inc\emu8086.inc; (codul original "scan_num" realizează mai multe operaţii). Numărul nu poate depăși 4 digiţi sau dimensiunea unui word – 16 biţi; sunt permise de asemenea și valori negative.

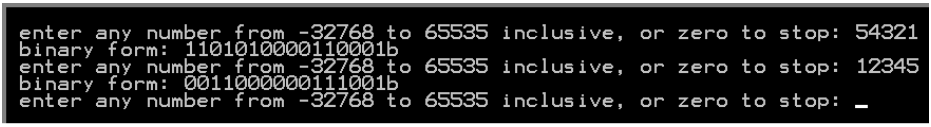


Figura A1.7. Fereastra aplicaţiei *convert_string_number_to_binary.asm*

HexConvertor.asm

Aplicaţia convertește un număr scris în hexazecimal pe 2 digiţi într-o valoare numerică, sau îl transformă în cod Ascii sau în binar și afișează valoarea rezultată pe ecran.

Figura A1.8. Fereastra aplicaţiei *HexConvertor.asm*

Bibliografie :

Sursa biblografică principală: tutorialele EMU puse la dispoziţie odată cu instalarea simulatorului

[alte surse: https://en.wikipedia.org/wiki/].

*Mostafa Abd-El-Barr, Hesham El-Rewini, "**Fundamentals of Computer Organization and Architecture**", 2005*

*Zoltan Baruch, "**Arhitectura calculatoarelor**", Editura Todesco, 2000*

*Barry B. Brey, "**The Intel Microprocessors",** 4th edition**,** 1997*

*John Hennesy, David Patterson, "**Computer Architecture – A quantitative Approach**" , 2007*

*Randall Hide, "**The Art of Assembly Language**", beta edition*

*Eugen Lupu, Simina Emerich , Anca Apatean, "**Initiere in Limbaj de Asamblare x86. Lucrari practice, teste si probleme**", Editura Galaxia Gutenberg, 2012*

*Scott Mueller, "**Upgrading and Repairing PCs**", 20th edition, 2012*

*Linda Null, Julia Lobur, "**The essentials of Computer Organization and Architecture**", 2003*

*David Patterson, John Hennesy, "**Computer Organization and Design – the hardware/software interface**", 4th edition, 2009*

*David Tarnoff, "**Computer Organization and Design Fundamentals",** editia intai revizuita, 2007*