Simulator de microprocessor 8086

Introducere despre simulatorul EMU8086

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.

Despre microprocesorul 8086 spunem că 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, 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).

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, …) sau pe 8 biți (AH, AL, BH, …);

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 8 biți (AH, AL, BH, …), 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.

5. Prezentarea simulatorului EMU8086

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**).

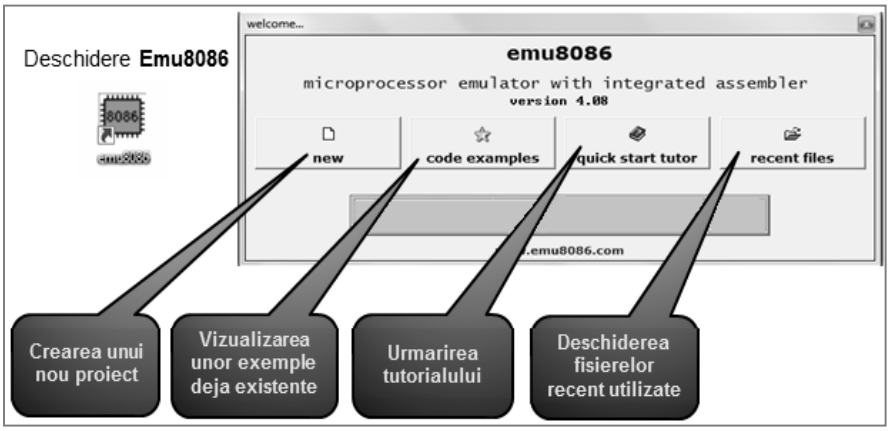

Figura 5.1. Fereastra principală a simulatorului Emu8086

5.1. 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*.

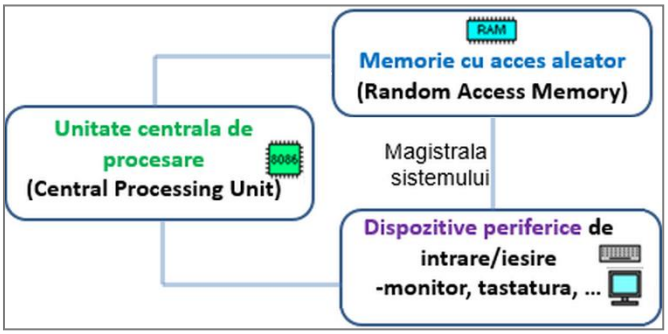

Figura 5.2. Componentele principale ale unui sistem de calcul emulat în EMU8086

(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. Datele pot fi luate din regiştri (sau din memorie), procesate de ALU, iar rezultatele pot fi stocate tot în regiştri (sau pot fi depuse în memorie).

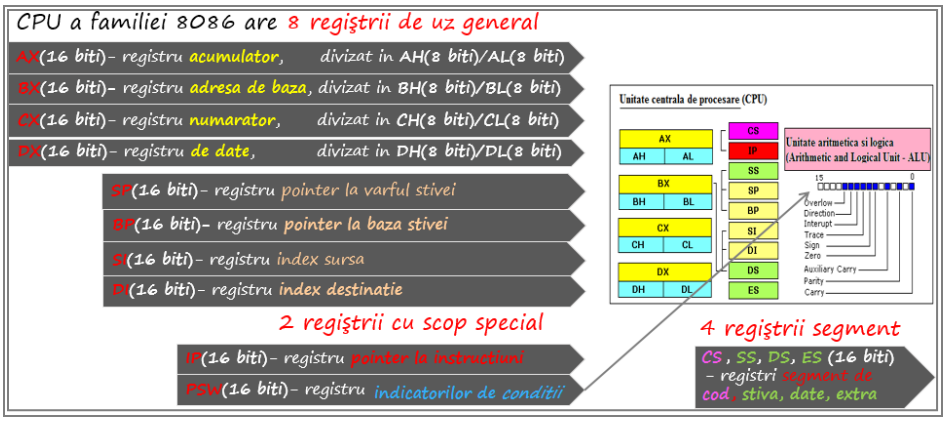

Figura 5.3. Structura Unității Centrale de Procesare din EMU8086

Similar cu un CPU al familiei 8086 (observaţi analogia cu arhitectura pe 16 biţi a microprocesorului 8086), 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 la arhtectura CPU 8086 sunt emulaţi de EMU8086, la fel și registrul de flaguri. Figura 5.4 arată modul cum se prezintă aceștia în simulatorul EMU.

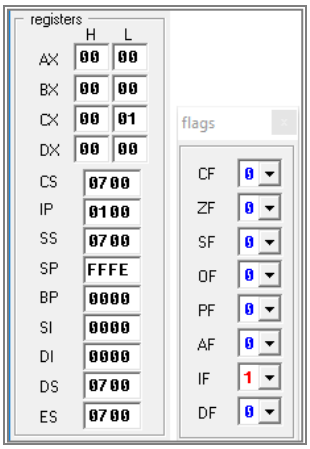

Figura 5.4. Modul de ilustrare al regiștrilor în EMU8086

(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 la formarea adresei (în accesarea memoriei) folosind segmente.

(3) Dispozitivele de intrare/ ieşire sau ***periphericele*** 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. (sistemului de calcul), iar folosind un periferic de ieşire vom genera informaţie în exteriorul S.C. 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 ulterior.

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.

5.2. 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).

5.3. 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 apărea fereastra din Figura 5.5.

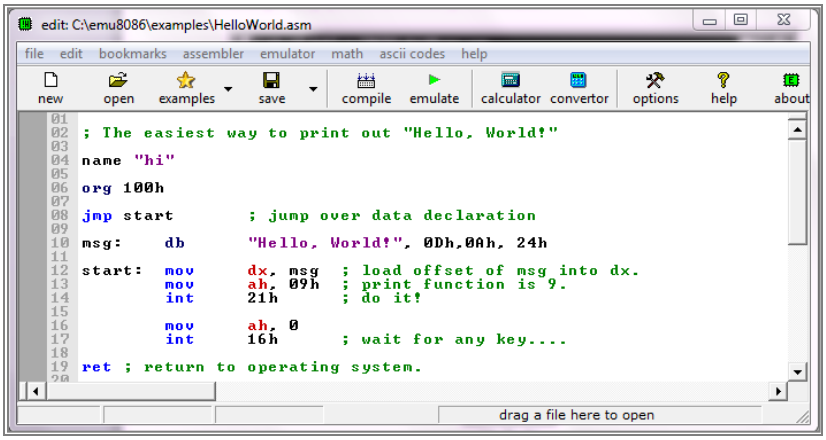

Figura 5.5. Fereastra principală de editare a codului sursă

În cadrul Figurii 5.5 se poate observa zona de editare a programului sursă (fişier de tip asm), iar în partea de sus este 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 şablon de tip *COM* sau *EXE*, aşa cum arată Figura 5.6.

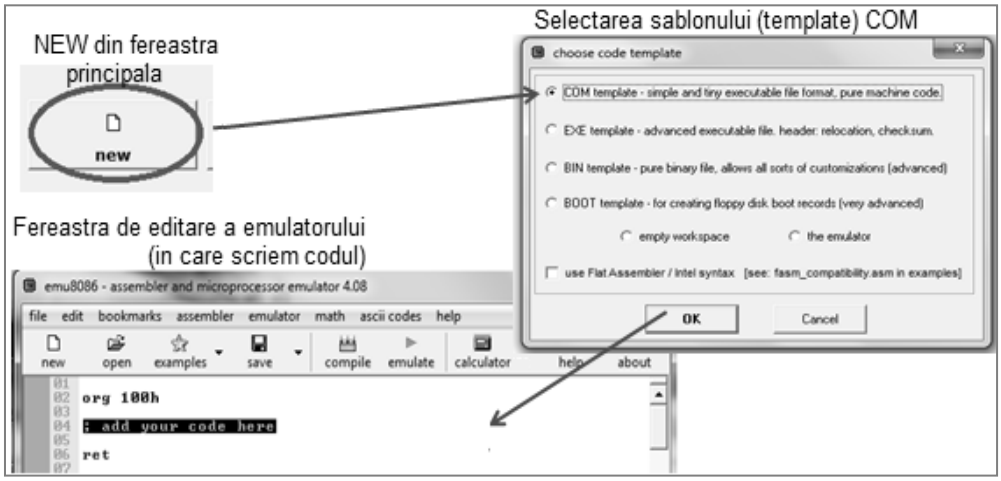

Figura 5.6. Fereastra de editare a codului în cadrul simulatorului

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*.

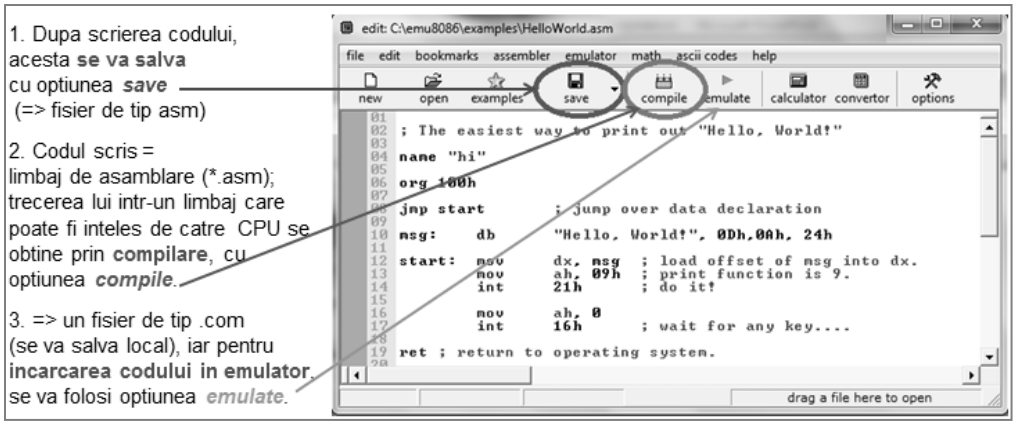

Figura 5.7. Ordinea de efectuare a operaţiilor asupra unui fişier .asm

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 apărea 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.

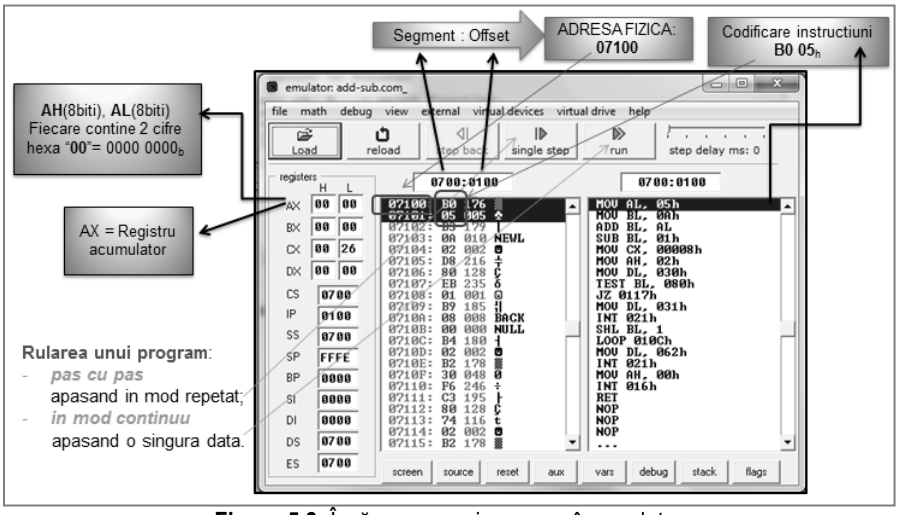

Figura 5.8. Încărcarea unui program în emulator

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 precizează convenţia Little Endian.

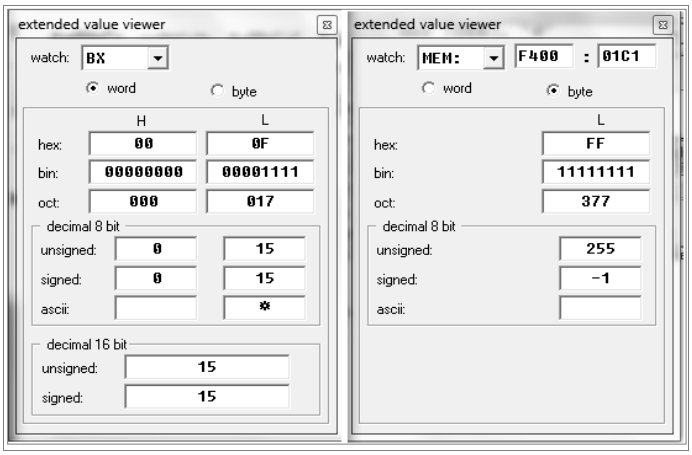

Figura 5.9. Fereastra de vizualizare şi modificare a regiştrilor (stânga) sau a unei zone din memorie (dreapta)

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.

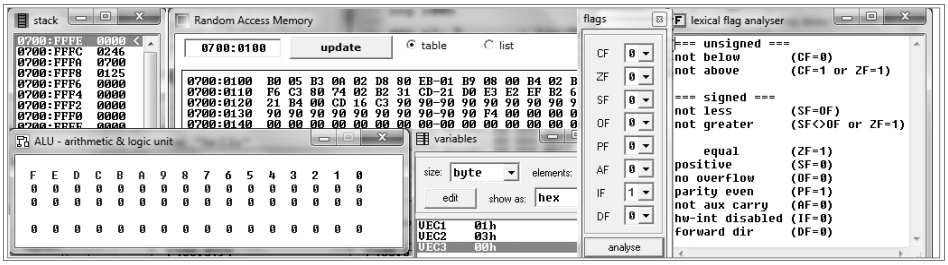

Figura 5.10. Ferestre de vizualizare a conţinutului stivei, memoriei, al ALU, al variabilelor definite în memorie şi al flag-urilor

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.

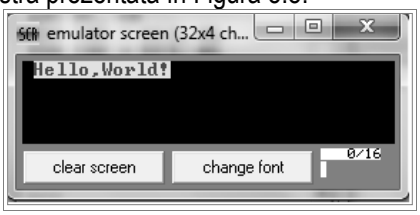

Figura 5.11. Fereastra de vizualizare a ecranului (emulator screen)

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.

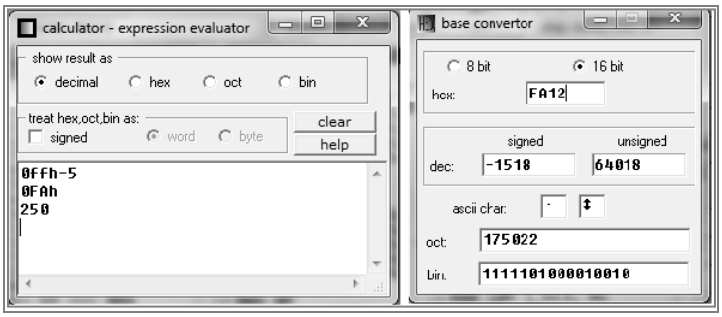

Figura 5.12. Ferestrele calculatorului şi convertorului

6. Folosirea simulatorului EMU8086

În continuare, 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 această instrucțiune este ***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.

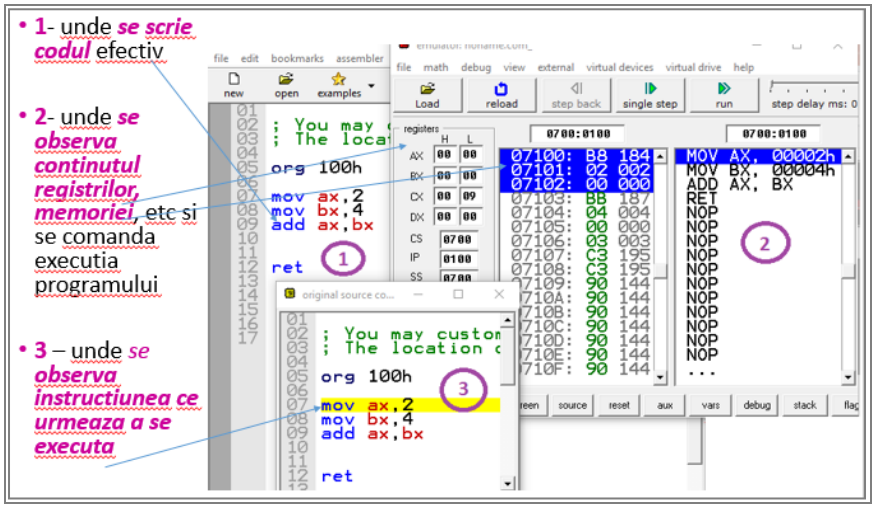

Figura 6.1. EMU8086 – execuţia primelor instrucţiuni cu emulatorul

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. Folosirea simulatorului pentru Conversii de numere

Conversia automată a numerelor dintr-o bază în alta poate fi realizată în cadrul simulatorului, așa cum am prezentat 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.

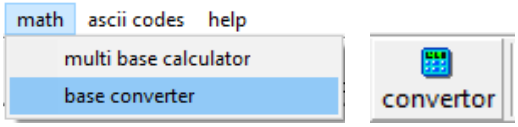

Figura 6.2. Emu8086 – opţiuni disponibile pentru accesarea convertorului

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 inițial, 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.

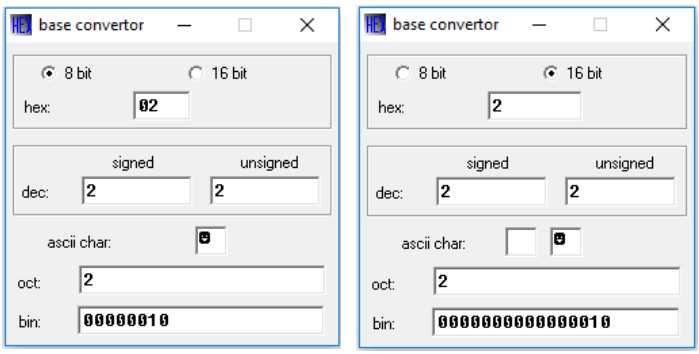

Figura 6.3. Emu8086 – conversia numărului 2 pe 8 sau 16 biţi

Exemplu: 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.

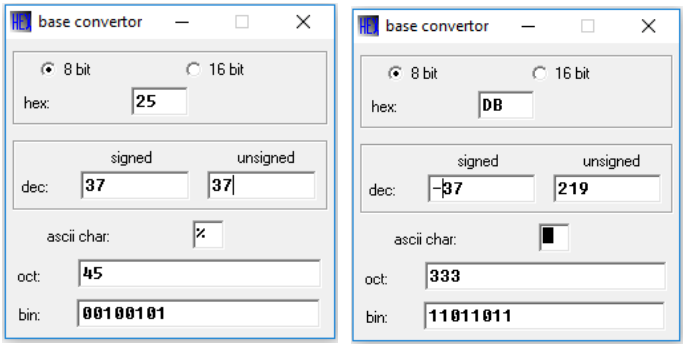

Figura 6.4. Emu8086 – conversia numărului 37 și -37 pe 8 biţi

6.2. Folosirea simulatorului pentru 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. Exemplele considerate aici folosi valorile pe 8 sau 16 biţi (cât au dimensiunea regiștrii interni ai procesorului 8086).

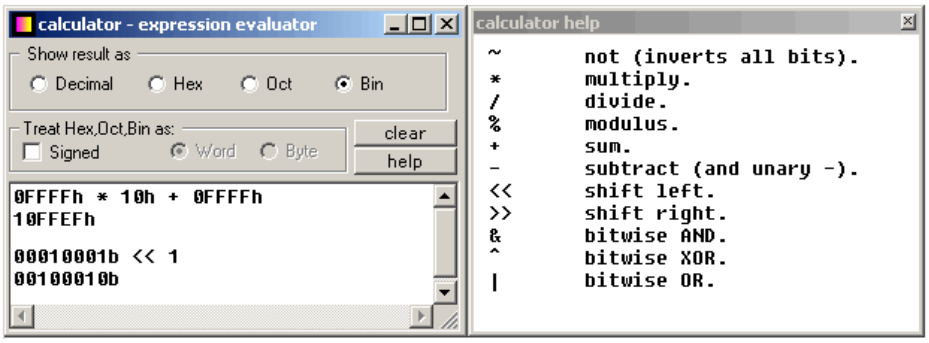

Figura 6.5. Fereastra Calculatorului (stânga) și operatorii disponibili (dreapta)

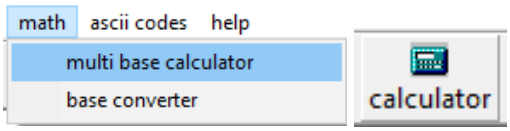

Figura 6.6. Emu8086 – opţiuni disponibile pentru accesarea calculatorului

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.

6.3. Folosirea simulatorului pentru realizarea Operaţiilor de adunare și scădere

În această secţiune, operaţiile menţionate pot fi realizate și folosind operatorii care sunt disponibili în simulator, precum:

+ și **-** pentru a realiza adunarea, respectiv scăderea a două numere,

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. Odată ce s-a obţinut rezultatul 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.

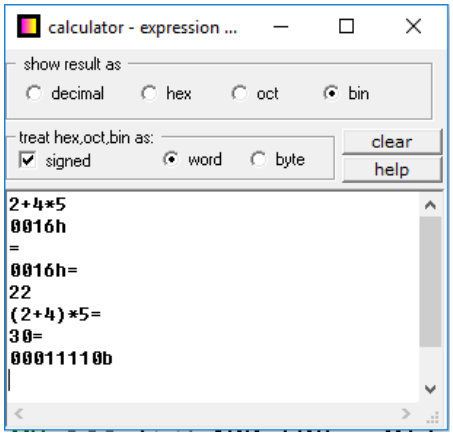

Figura 6.7. Exemple de calcule aritmetice

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).

Exemplu: Pentru adunarea sau scăderea a două numere, vom scrie și primul program în limbaj de asamblare:

```
mov AL, 100      ; depune în registrul AL de 8 biți valoarea 100
```

mov BH, 90 ; depune în registrul BH de 8 biți valoarea 90

add AL, BH ; adună cele 2 valori și depune rezultatul în registrul AL

Este rolul nostru ca programatori să asigurăm stocarea corectă a valorilor de operat (precum și a rezultatelor obținute în urma operației) în regiștrii de dimensiune potrivită. Tot aici va fi esențiala și interpretarea flagurilor aritmetice.

6.4. Scrierea corectă a datelor și a instrucțiunilor în simulator

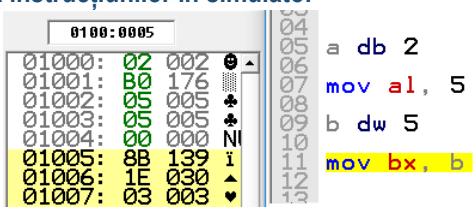

Figura 6.8. Secvenţă de instrucţiuni alternând date cu cod

În Figura 6.8 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. 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 6.8. 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 6.9 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 6.9. Cod scris incorect (stânga) vs cod scris corect (dreapta)

6.5. Folosirea simulatorului pentru Definirea și 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. 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, etc) 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. 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

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

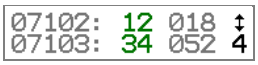

Figura 7.2. Modul de reprezentare al unei variabile de tip octet (a db 12h, 34h) în memorie

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*

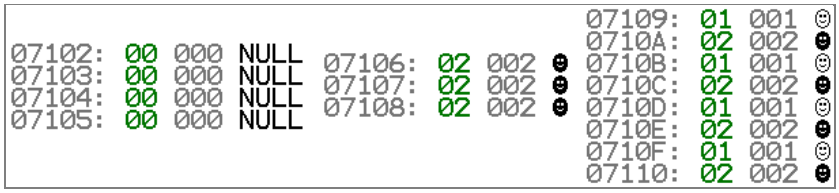

Figura 7.5. Modul de reprezentare a trei variabile de tip șir de octeţi definite cu operatorul DUP în memoria simulatorului

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
```

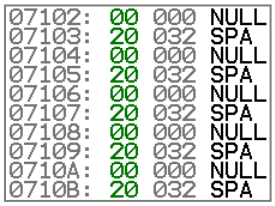

Figura 7.6. Modul de reprezentare al unei variabile de tip cuvânt în memorie

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.

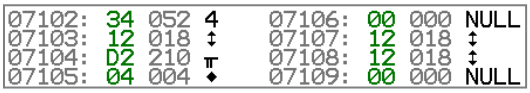

Figura 7.7. Modul de reprezentare a două variabile de tip șir de cuvinte în memorie

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
```

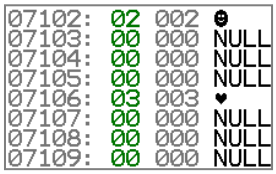

Figura 7.8. Modul de reprezentare a două variabile de tip dublucuvânt în memorie

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.

7.2. 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 înlănţuite.

```
07102:  41  065 A
07103:  6E  110 n
07104:  61  097 a
07105:  61  097 a
07106:  72  114 r
07107:  65  101 e
07108:  6D  109 m
07109:  65  101 e
0710A:  72  114 r
0710B:  65  101 e
```

Figura 7.9. Modul de reprezentare a trei variabile de tip șir de octeţi coduri Ascii în memoria simulatorului

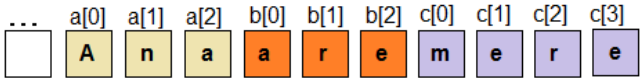

Figura 7.10. Reprezentarea variabilelor din memoria simulatorului pe orizontală

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.

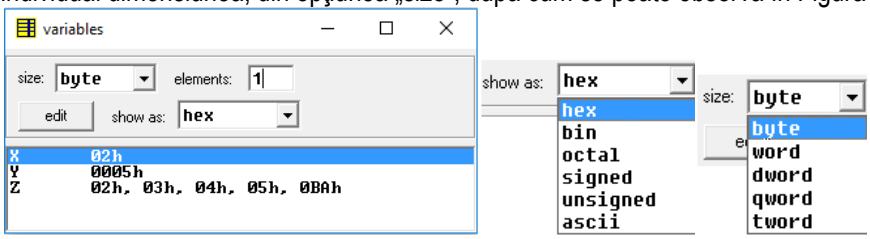

Figura 7.11. Vizualizarea variabilelor în EMU și opţiuni disponibile

7.3. 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.