Arhitectura Microprocesoarelor

- îndrumar de laborator -

Elena-Diana Șandru

Horia Cucu

Corneliu Burileanu

CUPRINS

1. INTRODUCERE ÎN ARHITECTURA X86 ÎN MODUL REAL ŞI EMULATORUL EMU8086

1.1 Schema bloc a unui microcalculator

Microcalculatorul, structurat ca o maşină "VON NEUMANN", este un sistem programabil de prelucrarea informaţiei care are două componente inseparabile şi definitorii: hardware şi software. Deoarece secvenţa de instrucţiuni poate fi schimbată, microcalculatorul poate să rezolve mai mult de un singur tip de probleme.

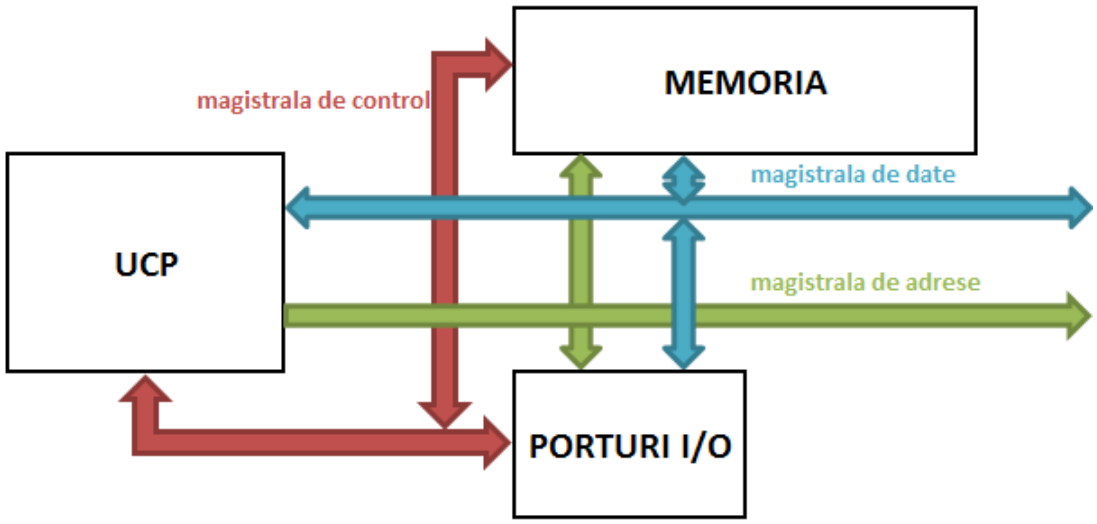


Figura 1. Schema bloc a unui microcalculator

În mod convenţional, un microcalculator conţine cel puţin un element de procesare, unitatea centrală de procesare (UCP), un tip de memorie şi dispozitive de intrare/ieşire, toate interconectate de diferite tipuri de magistrale.

- UCP: blocul hardware care controlează sistemul şi procesează datele;
- memoria: blocul hardware care stochează datele şi instrucţiunile într-o secvenţă de locaţii de memorie. Locaţiile de memorie sunt caracterizate de adrese şi conţinut;

- porturi I/O: blocuri hardware care formează interfaţa dintre microcalculator şi lumea exterioară;
- magistrale: conexiunile dintre cele 3 blocuri hardware.

1.2 Arhitectura x86 în modul real. Componentele UCP

Pe parcursul laboratorului de Arhitectura Microprocesoarelor veţi studia arhitectura Intel x86 în modul real. Această secţiune prezintă registrele şi fanioanele din interiorului UCP.

Un registru este ca o singură locaţie de memorie disponibilă în interiorul UCP. Registrele de uz general se constituie în blocuri hardware cu funcţie de stocare a informaţiei; fiind în interiorul UCP, ele sunt accesate mai rapid decât locaţiile de memorie. Scenariul tipic de folosire a registrelor este următorul: a) UCP încarcă informaţia dintr-o memorie mai mare în registre, b) informaţia din interiorul registrelor este folosită pentru operaţii aritmetico-logice, manipulată sau testată, c) în final, rezultatul este stocat înapoi în memoria de bază, fie prin intermediul aceleiaşi instrucţiuni, fie prin intermediul unei instrucţiuni ulterioare.

Arhitectura x86 pune la dispoziţie mai multe registre de 16 biţi. Patru dintre ele (**AX**, **BX**, **CX**, **DX**) sunt registre de uz general, ceea ce înseamnă că ar putea fi folosite pentru orice operaţie. Totuşi, există nişte restricţii asupra lor: există unele instrucţiuni care folosesc implicit unul sau mai multe registre de uz general. De exemplu:

- numai **AX** şi **DX** pot fi folosite pentru instrucţiunile de înmulţire şi împărţire;
- numai **BX** poate fi folosit pentru stocarea adreselor efective (pentru adresarea indirectă a memoriei);
- numai **CX** poate fi folosit ca şi contor pentru instrucţiunea loop.

Registrele de uz general, **AX**, **BX**, **CX**, **DX**, pot fi accesate ca 2 octeţi separaţi (de exemplu, octetul superior al lui **BX** poate fi accesat ca **BH** şi cel inferior ca **BL**).

Există 2 registre speciale cu rol de indicator ("pointer"): **SP** ("stack pointer") indică spre „vârful” stivei şi **BP** ("base pointer") este deseori folosit pentru a indica spre un alt loc din stivă, de obicei deasupra variabilelor locale. Deoarece aceste registre sunt folosite drept pointeri, informaţiile stocate în ele sunt, de obicei, interpretate drept adrese efective.

Registrele **SI** ("source index") şi **DI** ("destination index") sunt registre de adrese şi pot fi folosite şi pentru indexarea vectorilor. Aceste registre stochează adrese efective. Instrucţiuni speciale, precum instrucţiunile de manipulare a vectorilor, folosesc aceste registre pentru a indica către elementul curent din vectorul sursă (**SI**) şi către elementul curent din vectorul destinaţie (**DI**).

Microprocesoarele x86 în modul real organizează memoria în subdiviziuni logice numite segmente. Pentru definirea acestor segmente, se folosesc 4 registre segment: **CS** - registrul segment de program ("code segment"), **SS** - registrul segment de sivă ("stack segment"), **DS** - registrul segment de date ("data segment"), **ES** - registrul segment de date suplimentar

("extended data segment"). Ele vor stoca mereu adrese segment. Mai multe informaţii despre adresarea memoriei vor fi furnizate în laboratorul 3.

Registrul de fanioane **F** ("**FLAGS**") conţine toate fanioanele UCP:

- **CF** (carry flag): semnalizează un transport sau un împrumut aritmetic (depăşire) pentru *numerele fără semn*; *depăşirea aritmetică* are loc dacă rezultatul operaţiei este în afara gamei de reprezentare a numerelor (de exemplu, -12 si 270 sunt în afara gamei 0…+255 în care pot fi reprezentate numerele pe 8 biţi, fără semn);
- **PF** (parity flag): semnalizează că numărul de biţi egali cu 1 din cel mai puţin semnificativ octet al rezultatului este par;
- **AF** (auxiliary flag): semnalizează un transport sau un împrumut aritmetic peste primul nibble;
- **ZF** (zero flag): semnalizează că rezultatul este 0;
- **SF** (sign flag): semnalizează că cel mai semnificativ bit al rezultatului este setat (acesta este bitul de semn pentru prelucrarea numerelor întregi cu semn);
- **DF** (direction flag): controlează parcurgerea în sens crescător sau descrescător al adreselor elementelor dintr-un vector;
- **OF** (overflow flag): semnalizează o depăşire aritmetică pentru *numerele cu semn*; *depăşirea aritmetică cu semn* are loc dacă rezultatul operaţiei este in afara gamei de reprezentare a numerelor (de exemplu, -130 si +200 sunt in afara gamei -128…+127 in care pot fi reprezentate numerele pe 8 biţi, cu semn).

Registrul **IP** (instruction pointer – indicator de instrucţiuni) stochează adresa efectivă a instrucţiunii curente. Acest registru nu este un atribut de arhitectură.

1.3 Organizarea memoriei

În cazul organizării lineare a memoriei orice bloc de memorie poate fi considerat o **secvenţă de locaţii de memorie**. De obicei, fiecare locaţie de memorie stochează un număr pe 8 biţi, un octet (byte), în acest caz formatul memoriei fiind octetul; însă există și locații de memorie de alte dimensiuni, 2 octeți (bytes), 4 octeți (bytes), etc. Acest număr pe 8 biţi, 16 biți, 32 biți reprezintă conţinutul locaţiei de memorie.

Fiecare locaţie de memorie este identificată printr-un număr unic numit adresă, mai exact, adresa fizică. Dimensiunea hărţii memoriei este dependentă de adresa fizică prin următoarea formulă:

$$Harta\_memoriei\ (locaţii) = 2^{DimensiuneAdresaFizică[biti]} \quad (1)$$

Exemplu 1: folosind o adresă fizică de 2 biţi putem forma 4 adrese fizice diferite: 00, 01, 10 şi 11, corespunzând celor 4 locaţii diferite de memorie. Aşadar, o memorie cu adresa fizică pe 2 biţi este formată din 4 locaţii de memorie (4 octeţi dacă fiecare locație de memorie stochează 8 biți, 8 octeţi dacă fiecare locație de memorie stochează 16 biți, etc).

Exemplu 2: folosind o adresă fizică de 20 biţi putem forma 2^{20} adrese fizice diferite, corespunzând celor 2^{20} locaţii diferite de memorie. Aşadar, o memorie cu adresa fizică pe 20 biţi este formată din 2^{20} locaţii de memorie (1 MB dacă fiecare locație de memorie stochează 8 biți).

Procesoarele x86 în modul real folosesc organizarea segmentată a memoriei care va fi prezentată în Lucrarea de laborator nr. 3.

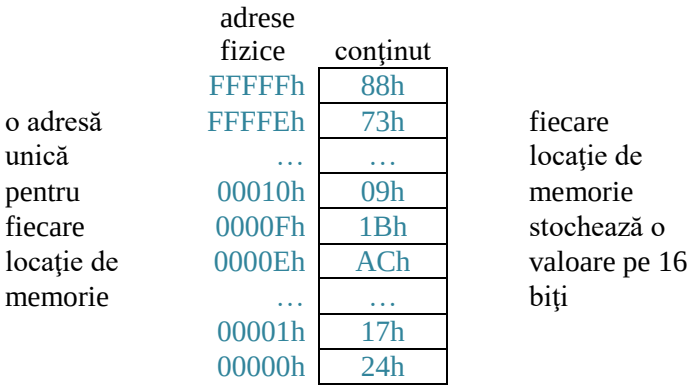


Figura 2. Memoria este o secvenţă de locaţii de memorie cu adrese unice

Notă: numerele din Figura 2 sunt scrise în hexazecimal. Pentru mai multe informaţii cu privire la bazele de numeraţie consultaţi secţiunea 1.7.

1.4 Directive de asamblare

Directivele de asamblare sunt operaţii care sunt executate de asamblor în timpul asamblării, nu de către CPU în timpul rulării. Directivele de asamblare pot realiza asamblarea programului dependent de anumiţi parametri de intrare daţi de programator, astfel încât un program poate fi asamblat în moduri diferite, eventual pentru diferite aplicaţii. Directivele de asamblare pot fi, de asemenea, utilizate pentru a prezenta un program, pentru a fi mai uşor de citit şi depanat. O altă utilizare obişnuită a directivelor de asamblare este să rezerve zone de stocare pentru date în timpul rulării şi opţional să iniţializeze conţinutul lor cu valori cunoscute. Directivele de asamblare pot fi folosite şi pentru a asocia nume arbitrare (etichete sau simboluri) cu locaţii de memorie şi diferite constante. De obicei, fiecărei constante şi variabile îi este asociat un nume pentru ca instrucţiunile să se poată referi la acele locaţii prin nume.

Directiva de asamblare x86 pe care o vom utiliza în platforma curentă este:

- **org** (numărătorul de atribuire a locaţiei)
    - Utilizare: org adresă

- Exemplu: org 100h
- Efect: Următoarea instrucţiune va fi încărcată în memorie la adresa specificată

1.5 Înţelegerea unor instrucţiuni x86

Tipurile de instrucţiuni pentru procesoarele x86 în modul real sunt următoarele:

- instrucţiuni de transfer de date
  - copierea unei constante într-un registru sau într-o locaţie de memorie
  - copierea datelor dintr-o locaţie de memorie într-un registru sau viceversa
  - copierea datelor de la/către dispozitivele I/O
- instrucţiuni de procesare de date
  - operaţii aritmetice (adunare, scădere, înmulţire, împărţire etc.)
  - operaţii logice (şi, sau, sau exclusiv, deplasări, rotaţii etc.)
  - operaţii de comparare
- instrucţiuni de control al execuţiei
  - saltul către o altă locaţie în program şi executarea instrucţiunilor de acolo
  - saltul condiţionat către o altă locaţie dacă o anumită condiţie este îndeplinită
  - saltul către o altă locaţie salvând adresa de reîntoarcere (un apel)

Mai jos sunt prezentate nişte instrucţiuni tipice x86.

MOV – Copiază Date

Mod de utilizare: MOV d, s

Operanzi:

- d – registru de uz general, registru segment (cu excepţia CS) sau locaţie de memorie
- s – constantă cu adresare imediată, registru de uz general, registru segment sau locaţie de memorie

Efecte: Copiază sursa la destinaţie, suprascriind valoarea destinaţiei: **(d) ← (s)**.

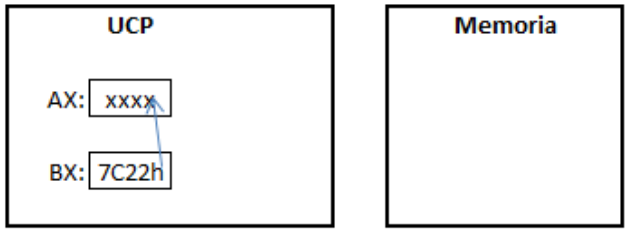
Fanioane: niciunul

Notă: Argumentele trebuie să aibă aceeaşi dimensiune (byte, word).

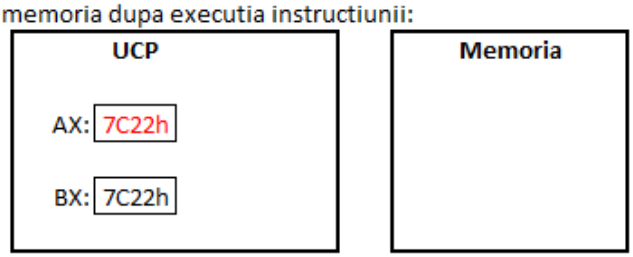
Exemplu

MOV AX, BX

1. UCP si memoria inainte de executia instructiunii:



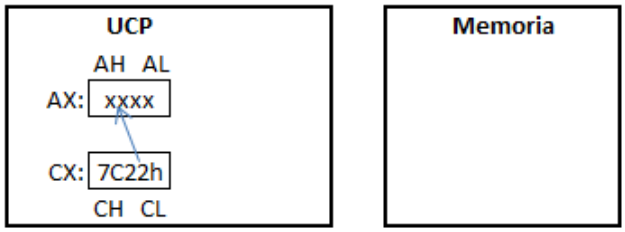
2. Valoarea 7C22h, care este stocata in registrul BX, este copiata in registrul AX.
3. UCP si memoria dupa executia instructiunii:



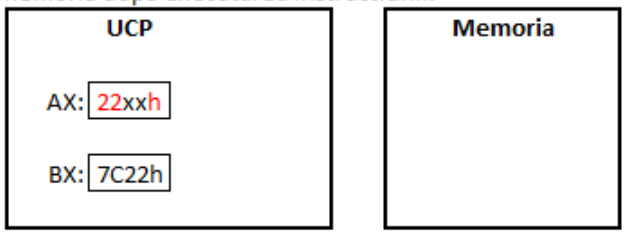
Exemplu

MOV AH, CL

1. UCP si memoria inainte de executarea instructiunii:

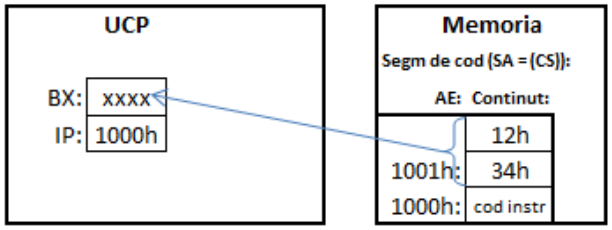


2. Valoarea 22h, care este stocata in registrul CL, este copiata in registrul AH.
3. UCP si memoria dupa executarea instructiunii:



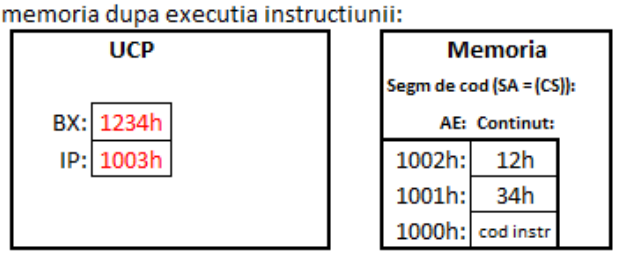
MOV BX, 1234h

1. UCP si memoria inainte de executia instructiunii:



2. Valoarea 1234h, care este stocata in segmentul de cod (imediat dupa codul instructiunii) este copiata in registrul BX

3. UCP si memoria dupa executia instructiunii:



Exemplu

ADD – Adunare de numere întregi

Mod de utilizare: ADD d, s

Operanzi: ADD d, s

- d – registru sau locaţie de memorie
- s – constantă cu adresare imediată, registru sau locaţie de memorie; nu pot fi folosiţi 2 operanzi din memorie

Efecte: Adunare; adună sursa la valoarea din destinaţie: **(d) ← (d) + (s)**.

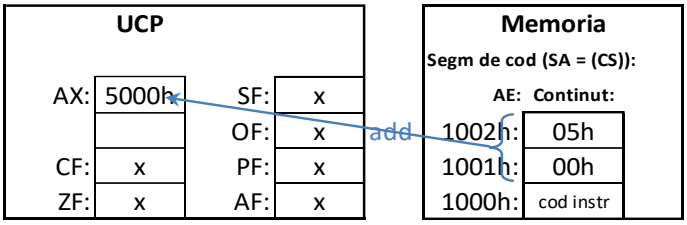
Fanioane: CF, ZF, OF, SF, AF si PF sunt setate conform rezultatului.

Notă: CF si OF ne indică depăşire în cazul valorilor fără semn, respectiv cu semn. SF indică semnul rezultatului în cazul valorilor cu semn.

Exemplu

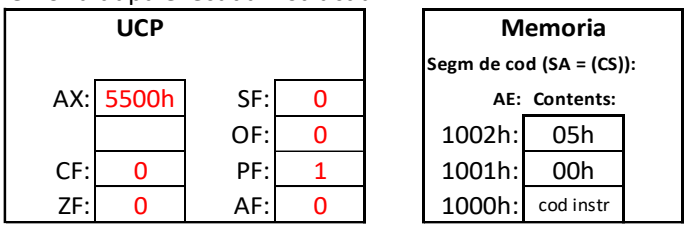
ADD AX, 500h

1. UCP si memoria inainte de executia instructiunii:



2. Valoarea 0500h, care este stocata in segmentul de cod (dupa codul instructiunii), este adunata la valoarea stocata in registrul AX. Fanioanele CF, ZF, OF, SF, AF si PF sunt setate conform rezultatului.

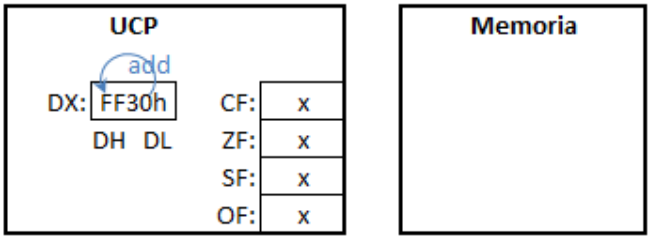
3. UCP si memoria dupa executia instructiunii:



Exemplu

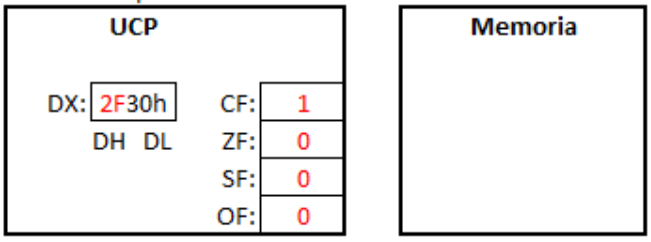
ADD DH, DL

1.UCP si memoria inainte de executia instructiunii:



2. Valoarea 30h, care este stocata in registrul DL, este adunata la valoarea stocata in registrul DH. Fanioanele CF, ZF, OF, SF, AF si PF sunt setate conform rezultatului.

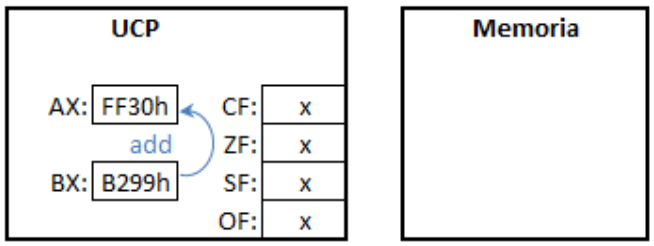
3. UCP si memoria dupa executia instructiunii:



Exemplu

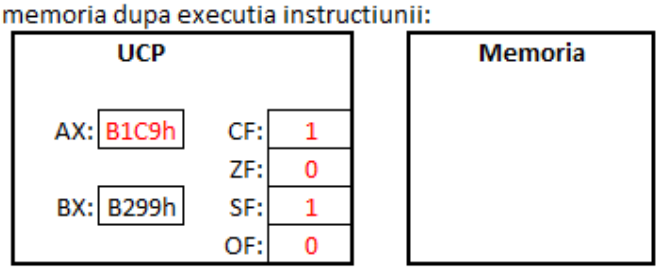
ADD AX, BX

1.UCP si memoria inainte de executia instructiunii:



2. Valoarea B299h, care este stocata in registrul BX, este adunata la valoarea stocata in registrul AX. Fanioanele CF, ZF, OF, SF, AF si PF sunt modificate conform rezultatului.

3. UCP si memoria dupa executia instructiunii:



ADC – Adunare cu transport

Mod de utilizare: ADC d, s

Operanzi:

- d – registru sau locaţie de memorie
- s – constantă cu adresare imediată, registru sau locaţie de memorie; nu pot fi folosiţi 2 operanzi din memorie

Efecte: Adunare cu transport; adună fanionul de transport (CF) şi sursa la destinaţie:

(d) ← (d) + (s) + (CF).

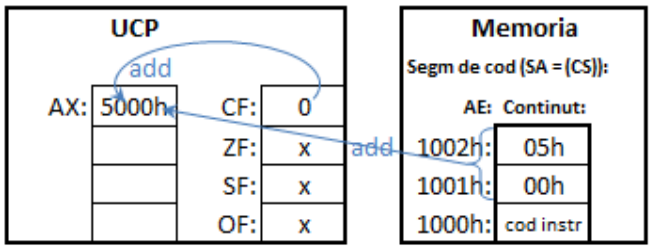
Fanioane: CF, ZF, OF, SF, AF si PF sunt setate conform rezultatului.

Notă: CF şi OF ne indică depăşire în cazul valorilor fără semn, respectiv cu semn. SF indică semnul rezultatului în cazul valorilor cu semn.

Exemplu

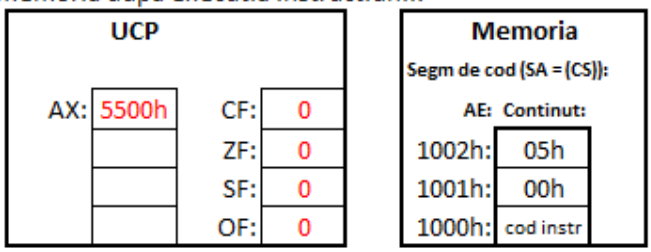
ADC AX, 500h

1.UCP si memoria inainte de executia instructiunii:



2. Fanionul CF si valoarea 0500h, care este stocata in segmentul de cod (dupa codul instructiunii), sunt adunate la valoarea stocata in registrul AX. Fanioanele CF, ZF, OF, SF, AF si PF sunt setate conform rezultatului.

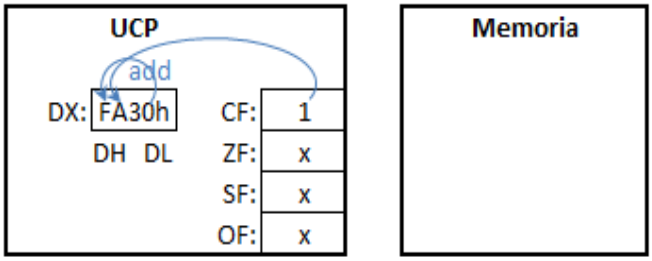
3. UCP si memoria dupa executia instructiunii:



ADC DH, DL

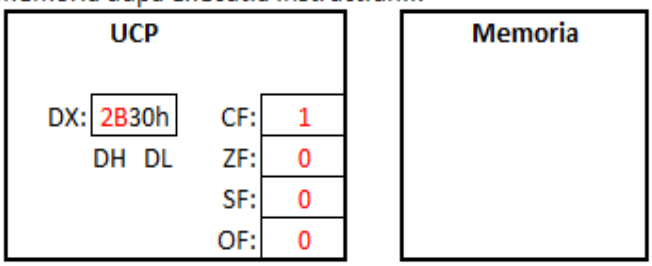
Exemplu

1.UCP si memoria inainte de executia instructiunii:



2. Fanionul CF si valoarea 30h, care este stocata in registrul DL, sunt adunate la valoarea stocata in registrul DH. Fanioanele CF, ZF, OF, SF, AF si PF sunt setate conform rezultatului.

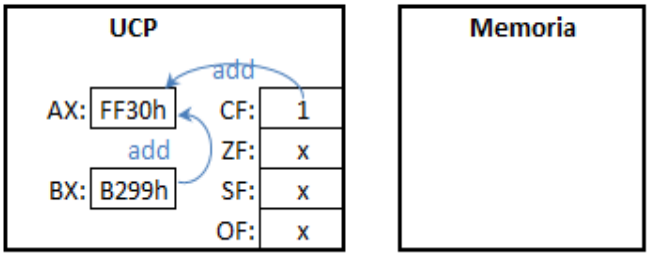
3. UCP si memoria dupa executia instructiunii:



ADC AX, BX

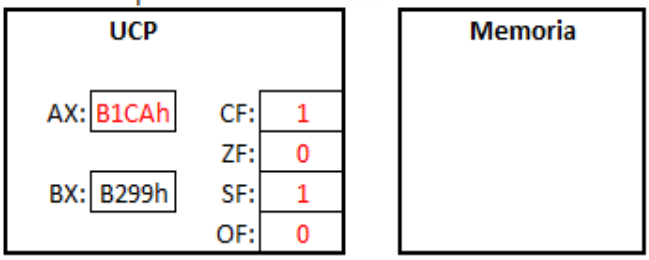
Exemplu

1.UCP si memoria inainte de executia instructiunii:



2. Fanionul CF si valoarea B299h, care este stocata in registrul BX, sunt adunate la valoarea stocata in registrul AX. Fanioanele CF, ZF, OF, SF, AF si PF sunt setate conform rezultatului.

3. UCP si memoria dupa executia instructiunii:



DIV – Împărţire fără semn

Mod de folosire: DIV s

Operanzi:

- s – registru pe 8 sau 16 biţi, locaţie de memorie;

Efecte:

- dacă s este o valoare pe 8 biţi: se împarte valoarea stocată în registrul **AX** la **s**, restul împărţirii este stocat în **AH**, iar câtul împărţirii este stocat in AL:
  (AH) ← (AX) mod (s), **(AL) ← (AX) div (s)**.
- dacă **s** este o valoare pe 16 biţi: se împarte valoarea stocată în **DX** concatenat cu **AX** la **s**, restul împărţirii este stocat în **DX**, iar câtul împărţirii este stocat în **AX**:
  (DX) ← (DX) ↕ (AX) mod (s), **(AX) ← (DX) ↕ (AX) div (s)**.

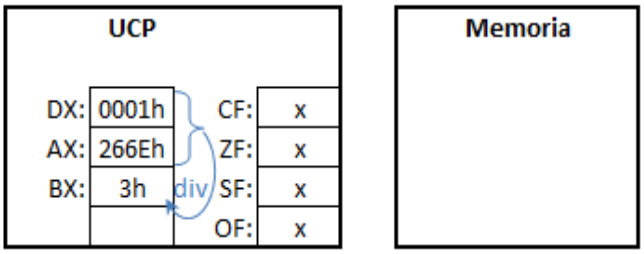
Fanioane: CF, ZF, OF, SF, AF şi PF sunt nedefinite.

Nota: În cazul în care câtul împărţirii este mai mare de 8 biţi respectiv 16 biţi şi nu poate fi stocat în **AL** respectiv **AX,** atunci va apărea o eroare de depăşire la împărţire.

Exemplu

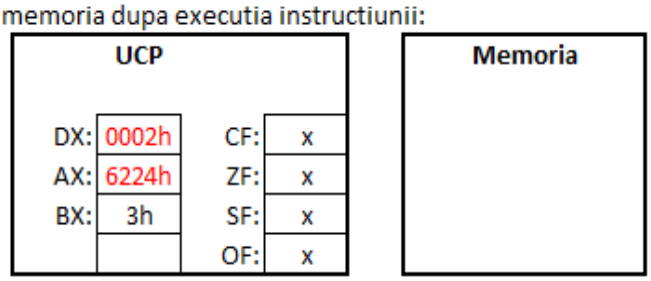
DIV BX

1.UCP si memoria inainte de executia instructiunii:



2. Valoarea 0001266Eh, care este stocata in DX concatenat cu AX, este impartita la valoarea stocata in registrul BX. Restul este stocat in registrul DX si catul este stocat in registrul AX. Fanioanele CF, ZF, OF, SF, AF si PF sunt nedefinite.

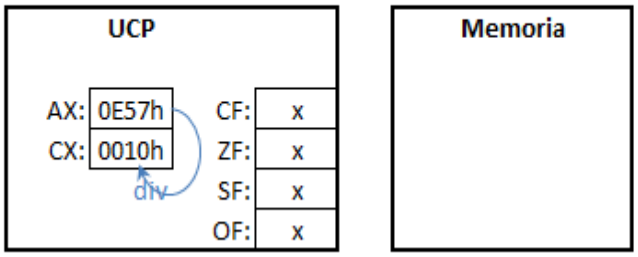
3. UCP si memoria dupa executia instructiunii:



Exemplu

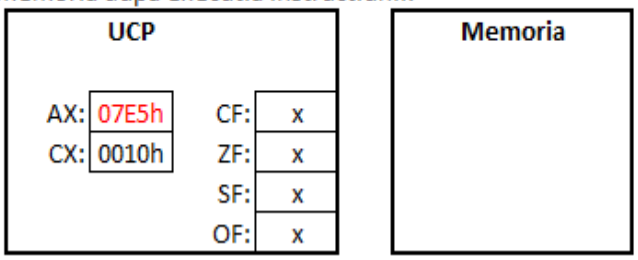
DIV CL

1.UCP si memoria inainte de executia instructiunii:



2. Valoarea 0E57h, care este stocata in AX, este impartita la valoarea stocata in registrul CL. Restul este stocat in registrul AH si catul este stocat in registrul AL. Fanioanele CF, ZF, OF, SF, AF si PF sunt nedefinite.

3. UCP si memoria dupa executia instructiunii:



1.6 Emulatorul emu8086

Emulatorul emu8086, disponibil gratuit pentru descărcare la http://www.emu8086.com, va fi folosit pe parcursul laboratoarelor de Arhitectura Microprocesoarelor pentru a exemplifica atributele arhitecturale ale arhitecturii x86. Acest emulator va fi utilizat pentru a scrie programe în limbaj de asamblare, a le compila şi a le executa cu scopul de a înţelege cum operează microprocesorul.

Emulatorul emu8086 are numeroase opţiuni, dar scenariul de bază pentru laborator va fi următorul:

1. Deschideţi emulatorul. Fereastra Source Code (fără a conţine nici un cod de asamblare) va fi afişată.
2. Folosiţi fereastra Source Code pentru a scrie un program în limbaj de asamblare. Salvaţi programul selectând opţiunea File menu -> Save As submenu.
3. Opţional, dacă programul vă este dat de îndrumătorul de laborator, încărcaţi fişierul .asm apăsând butonul Open şi selectând fişierul sursă.
4. Compilaţi programul:
   4.1. Daţi click pe butonul Compile pentru a compila programul.
   4.2. Dacă în caseta Assembler Status dialog, afişată după compilare, apar erori de compilare, apăsaţi dublu click pe textul de eroare pentru a vă întoarce în Source Code window, reparaţi eroarea şi compilaţi din nou programul.
   4.3. Dacă nu sunt erori de asamblare continuaţi cu pasul următor.

5. Încărcaţi programul executabil în emulator.

   5.1. In fereastra Assembler Status, afişată după compilare, apăsaţi butonul Run.

6. Executaţi programul pas cu pas, observaţi cum se modifică valorile registrelor, ale locaţiilor de memorie, ale fanioanelor etc. şi notaţi observaţii pe baza lor.

   6.1. Apăsaţi butonul de Reload pentru a relua programul.

   6.2. Observaţi registrele în fereastra Emulator.

   6.3. Vizualizaţi fereastra Source făcând click pe butonul Source.

   6.4. Opţional, dacă este necesar, observaţi şi fereastra fanioanelor făcând click pe butonul Flags.

   6.5. Opţional, dacă este necesar, observaţi şi fereastra memoriei făcând click pe butonul Memory din submeniul View.

   6.6. Opţional, dacă este necesar, observaţi şi fereastra stivei făcând click pe butonul Stack.

   6.7. Apăsaţi butonul Single Step şi observaţi cum instrucţiunea curentă, evidenţiată, este executată. Sesizaţi modul în care se modifică registrele, locaţiile de memorie, fanioanele etc.

   6.8. Repetaţi pasul 6.7 până când apare o fereastră de dialog care afişează: *the program has returned control to the operating system*.

7. Trageţi concluzii referitor la efectul pe care diverse instrucţiuni îl au asupra registrelor, locaţiilor de memorie, fanioanelor etc.

1.7 Exerciţii

1.7.1 Exerciţiul 1

Obiectiv. Scopul acestui exerciţiu este de a vă familiariza cu emulatorul emu8086 şi cu instrucţiunile MOV şi ADD.

Cerinţă: Scrieţi un program care să facă media aritmetică a 3 numere pe 16 biţi, fără semn.

Soluţie.

1. Porniţi emulatorul.

2. Utilizaţi fereastra Source Code pentru a scrie următorul program:

```
org 100h

mov AX, 0h
mov DX, 0h

mov AX, 1234h
add AX, 8017h
adc DX, 0h
add AX, 9423h
adc DX, 0h

mov BX, 3h
div BX
mov CX, AX

int 20h
```

3. Explicaţia programului

   3.1. Prima linie din program (org 100h) nu este o instrucţiune. Aceasta este o directivă de asamblare care specifică faptul că următoarea instrucţiune (şi implicit tot programul) va fi încărcată în memorie (în segmentul de cod) începând cu adresa 100h.

   3.2. Următoarele 2 instrucţiuni iniţializează registrele care vor stoca media. Observaţi că suma a 3 numere pozitive pe 16 biţi rezultă într-un număr mai mare, care s-ar putea să nu încapă pe 16 biţi. De aceea vom folosi 2 registre pe 16 biţi (AX şi DX) pentru a stoca rezultatul. DX va stoca cei mai semnificativi 2 octeţi si AX pe cei mai puţin semnificativi 2 octeţi.

   3.3. Mergând mai departe, primul număr (1234h) este încărcat în AX (mov AX, 1234h). Apoi al doilea număr este adunat la AX (add AX, 8017h).

   3.4. După instrucţiunea anterioară, fanionul de transport (CF) ar putea avea valoarea 1 (dacă suma nu încape pe 16 biţi). Valoarea acestui bit este adunată la DX (adc DX, 0h).

   3.5. În continuare, al treilea număr (9423h) este adunat la AX (add AX, 9423h). Apoi fanionul de transport este din nou adunat la DX (adc DX, 0h).

   3.6. Instrucţiunea mov BX, 3h încarcă valoarea 3h în BX, iar instrucţiunea div BX împarte valoarea pe 32 de biţi stocată în DX ↕ AX la valoarea pe 16 biţi stocată în BX. După diviziune, câtul este stocat în AX şi restul în DX.

   3.7. În final, rezultatul este copiat din AX în CX.

   3.8. Instrucţiunea int 20h este o întrerupere software. Aceasta termină programul curent şi returnează controlul sistemului de operare.

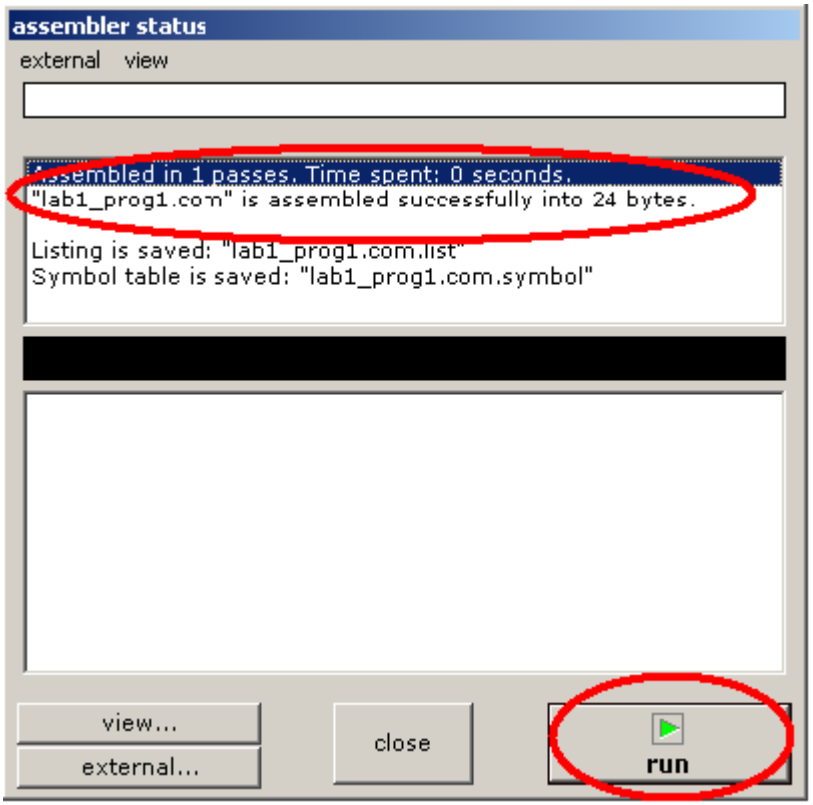
4. Salvaţi programul (File menu -> Save As submenu) cu numele lab1_prog1.asm.

5. Compilaţi programul:

   5.1. Faceţi click pe butonul Compile pentru a compila programul.

5.2. Veţi fi îndrumaţi să salvaţi fişierul executabil. Salvaţi-l cu numele recomandat (lab1_prog1.com).

5.3. Vizualizaţi statusul compilării în caseta Assembler Status. Dacă programul a fost editat corect, mesajul ar trebui sa fie: “lab1_prog1.com is assembled successfully into 24 bytes.”



6. Încărcaţi programul executabil în emulator.

6.1. Faceţi click pe butonul Run pentru a încărca programul în emulator şi a-l executa.

7. Executaţi programul pas cu pas, urmariţi schimbările apărute în cadrul registrelor, locaţiilor de memorie, fanioanelor etc. şi notaţi observaţii legate de acestea.

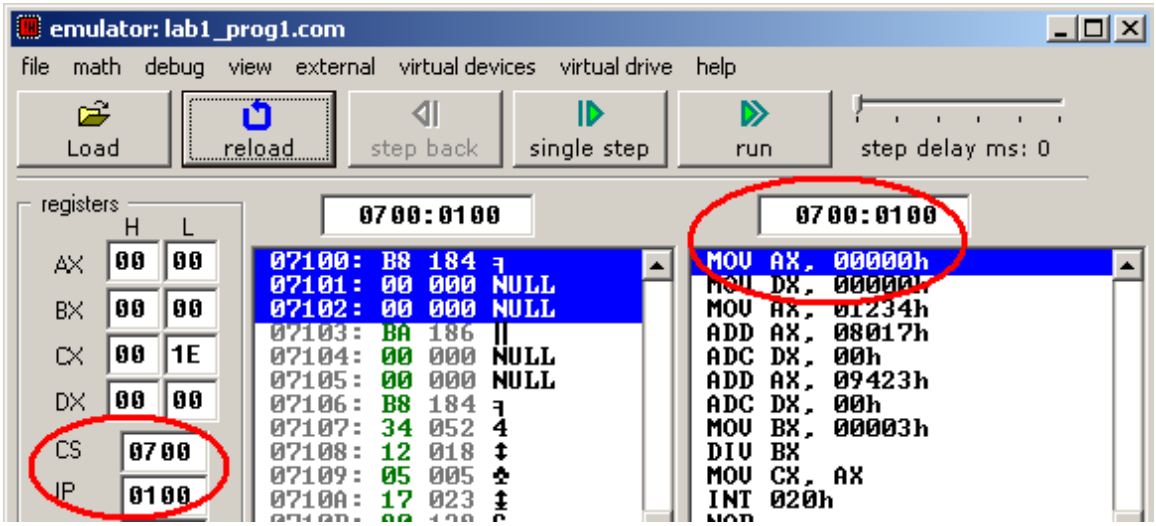
7.1. Faceţi click pe butonul de Reload pentru a reîncărca programul executat.

7.2. Observaţi fereastra Emulator si reţineţi că:

7.2.1. Instrucţiunea curentă (mov AX, 00000h) este evidenţiată. Aceasta este prima instrucţiune din program şi a fost încărcată la adresa logică 0700:0100 (adresa

segment : adresa efectivă). Adresa efectivă a fost impusă de directiva de asamblare org 100h.

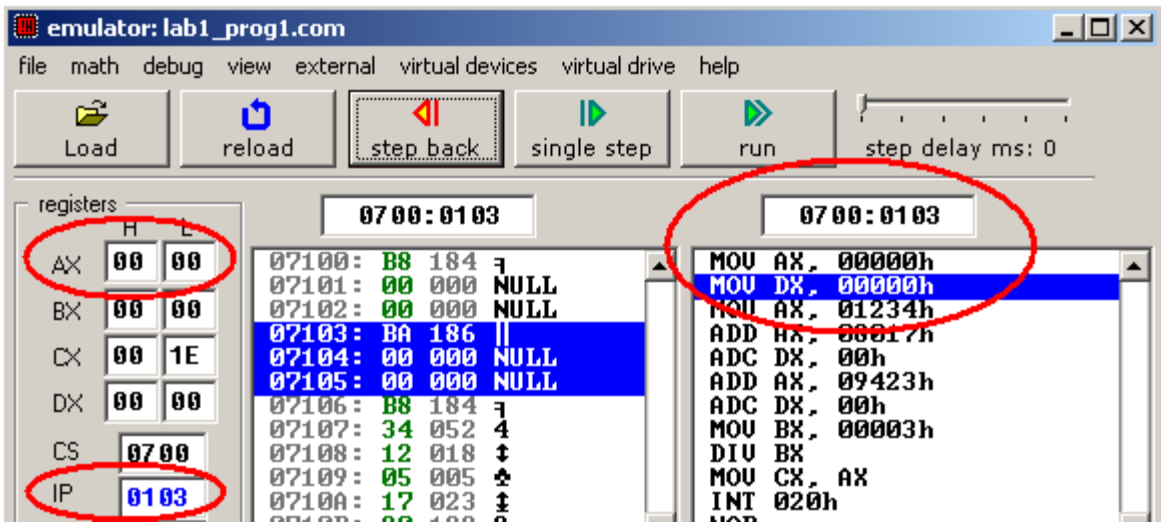
7.2.2. Valoarea din registrul IP (registrul care stochează adresa efectivă a instrucţiunii curente) este 0100h.



7.3. Faceţi click pe butonul Single Step pentru a executa prima instrucţiune. Observaţi că:

7.3.1. Valoarea din registrul IP s-a schimbat (în 103h) deoarece IP punctează acum către a doua instrucţiune (evidenţiată), care este stocată în memorie la adresa 0700:0103.

7.3.2. Valoarea stocată în registrul AX nu s-a modificat, întrucât era deja 0000h.



7.4. Faceţi din nou click pe butonul Single Step pentru a executa a doua instrucţiune. Observaţi că:

7.4.1. Valoarea din registrul IP s-a schimbat (în 0106h) pentru că IP pointează acum către a treia instrucţiune (evidenţiată), care este stocată în memorie la adresa 0700:0106.

7.4.2. Valoarea stocată în registrul DX nu s-a schimbat, deoarece era deja 0000h.

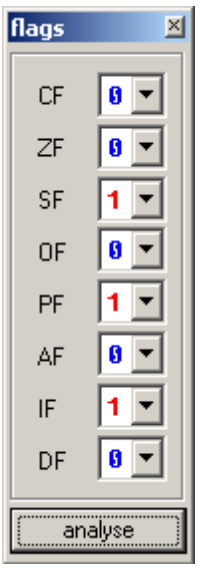
7.5. Executaţi instrucţiunea următoare (mov AX, 1234h) şi observaţi că:

7.5.1. Registrul AX a fost încărcat cu valoarea 1234h. Noua valoare din AX este acum 1234h.

7.5.2. Valoarea din registrul IP s-a schimbat din nou.

7.6. Faceţi click pe butonul Flags şi vizualizaţi statusul fanioanelor după această operaţie aritmetică. Observaţi că pentru moment toate fanioanele au valoarea 0.

7.7. Executaţi următoarea instrucţiune (add AX, 8017h) şi reţineţi că:



7.7.1. Registrul AX a fost încărcat cu valoarea sumei dintre valoarea sa anterioară (1234h) şi valoarea 8017h. Noua valoare din AX este acum 924Bh.

7.7.2. Fanionul de transport (CF) este în continuare 0, deoarece suma dintre cele 2 valori nu a rezultat într-un număr mai mare de 16 biţi.

7.7.3. Fanionul de zero (ZF) este în continuare 0, deoarece rezultatul este o valoare nenulă.

7.7.4. Fanionul de semn (SF) este 1 şi fanionul de overflow (OF) este 0. Acestea vor fi discutate în Laboratorul 2.

7.7.5. Fanionul de paritate (PF) este 1, deoarece suma modulo 2 a biţilor din rezultat este 1.

7.7.6. Valoarea din registrul IP s-a schimbat din nou.

7.8. Executaţi următoarea instrucţiune (adc DX, 0h) şi observaţi că:

7.8.1. Valoarea din registrul DX rămâne neschimbată, deoarece fanionul de transport este 0.

7.8.2. Valoarea din registrul IP s-a schimbat din nou.

7.9. Executaţi următoarea instrucţiune (add AX, 9423h) şi sesizaţi că:

7.9.1. Registrul AX a fost incărcat cu suma dintre valoarea sa anterioară (924Bh) şi valoarea 9423h. Noua valoare din AX este acum 266Eh, care reprezintă doar cei mai puţin semnificativi 16 biţi din rezultatul de 17 biţi. Cel mai semnificativ bit, care este 1, este stocat în fanionul de transport.

7.9.2. Fanionul de transport (CF) este 1, deoarece suma dintre cele 2 valori a rezultat într-o depăşire aritmetică. În acest caz, fanionul de transport stochează cel mai semnificativ bit al rezultatului.

7.9.3. Fanionul de zero (ZF) este în continuare 0, deoarece rezultatul este o valoare nenulă.

7.9.4. Fanionul de semn (SF) este 0 şi cel de overflow (OF) este 1. Acestea vor fi discutate în Laboratorul 2.

7.9.5. Fanionul de paritate (PF) este 0, deoarece suma modulo 2 a biţilor din rezultat este 0.

7.9.6. Valoarea din registrul IP s-a schimbat din nou.

7.10. Executaţi următoarea instrucţiune (adc DX, 0h) şi observaţi că:

7.10.1. Valoarea din registrul DX este incrementată cu 1 (valoarea fanionului de transport a fost adăugată la valoarea anterioară a lui DX). În consecinţă, valoarea din DX este acum 1h.

7.10.2. Toate fanioanele aritmetice sunt actualizate din nou. Dintre acestea, fanionul de transport (CF) devine 0, deoarece suma dintre DX, 0h şi CF nu a produs o depăşire aritmetică.

7.10.3. Valoarea din registrul IP s-a schimbat din nou.

7.11. Executaţi instrucţiunea următoare (mov BX, 3h) şi observaţi că:

7.11.1. Registrul BX a fost încărcat cu valoarea 3h.

7.11.2. Valoarea din registrul IP s-a schimbat din nou.

7.12. Executaţi următoarea instrucţiune (div BX) şi observaţi că:

7.12.1. Valoarea pe 32 de biţi stocată în DX ↕ AX (1266Eh) este împărţită la valoarea pe 16 biţi stocată în BX (3h). După împărţire, câtul (6224h) este stocat în AX şi restul (2h) este stocat în DX.

7.13. Executaţi următoarea instrucţiune (mov CX, AX) şi observaţi că:

7.13.1. Registrul CX este încărcat cu valoarea stocată în registrul AX. Valoarea lui AX nu se modifică.

7.14. Instrucţiunea curentă este int 20h. Faceţi click pe butonul Single Step **de două ori** şi observaţi că o fereastră de mesaj este afişată, spunând că programul a returnat controlul sistemului de operare. Faceţi click pe Ok.

8. Scrieţi concluzii referitoare la efectul diferitelor instrucţiuni asupra registrelor şi a locaţiilor de memorie.

1.8 Anexa 1. Bazele de numeraţie: 2, 10, 16

Orice număr poate fi reprezentat ca:

- număr zecimal (număr în baza 10) şi poate fi scris ca o secvenţă de cifre (0,1,…,9),
- număr binar (număr în baza 2) şi poate fi scris ca o secvenţă de cifre binare sau biţi (0 şi 1),
- număr hexazecimal (număr în baza 16) şi poate fi scris ca o secvenţă de simboluri hexazecimale (0, 1, …, 9, A, B, C, D, E şi F). Un număr hexazecimal este identificat folosind sufixul „h” (1A44h) sau prefixul “0x” (0x1A44).

	Definiţie	**Gama binară**	**Gama zecimală fără semn**	**Gama hexazecimală**
byte (B)	secvenţă de 8 biţi	00000000 – 11111111	0 – 255	0x00 – 0xFF
word (w)	secvenţă de 16 biţi	…	0 – 65535	0x0000 – 0xFFFF
double word (dw)	secvenţă de 32 biţi	…	…	0x00000000 – 0xFFFFFFFF

1.8.1 Conversia dintr-o bază în alta pentru numere fără semn

Zecimal -> Binar (Hexazecimal)

- se împarte repetat numărul la 2 (16), reţinând resturile
- se scriu toate resturile împărţirilor în ordine inversă

Binar (Hexazecimal) -> Zecimal

- se înmulţeşte fiecare cifră binară (hexazecimală) cu 2 (16) la puterea indexului respectivei cifre şi se însumează produsele
- exemplu (binar): valoareZecimală(1101_2) = $1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 13_{10}$
- exemplu (hexazecimal): valoareZecimală ($3A_{16}$) = $3*16^1 + A*16^0 = 58_{10}$

Binar <-> Hexazecimal

- se observă că o secvenţă de 4 biţi se converteşte într-un singur simbol hexazecimal şi viceversa:
  - 0000 -> 0h, 0001 -> 1h, ..., 1111 ->Fh
- o secvenţă mai lungă de 4 biţi poate fi convertită în hexazecimal prin convertirea secvenţelor de 4 biţi în hexazecimal (începând de la dreapta spre stânga:
  - 0101101101101011110111 -> 01 0110 1101 1010 1111 0111 -> 1 6 D A F 7 -> 0x16DAF7
- o secvenţă de simboluri hexazecimale va fi convertită simbol cu simbol în secvenţe de 4 biţi:
  - 0x3F9 -> 0011 1111 1001 -> 11111110001

Pentru mai multe informaţii despre bazele de numeraţie şi conversii accesaţi website-ul: http://www.purplemath.com/modules/numbbase.htm.

2. INSTRUCȚIUNI DE TRANSFER ȘI PRELUCRARE DE DATE. INSTRUCȚIUNI DE CONTROL AL PROGRAMULUI

2.1 Reprezentarea informaţiei în sistemele digitale

Tipurile de date cu mai mult de 2 valori posibile sunt stocate folosind secvenţe de biţi:

- byte (octet) (**B**) – o secvenţă de 8 biţi: poate reprezenta maximum 2^8 (256) valori
- word (cuvânt) (**w**) – o secvenţă de 16 biţi: poate reprezenta maximum 2^{16} (65 536) valori
- double word (cuvânt dublu) (**dw**) – o secvenţă de 32 biţi: poate reprezenta maximum 2^{32} valori

Numerele întregi fără semn (pozitive) pot fi reprezentate folosind convenţia "binar natural". Pentru numere întregi cu semn (pozitive şi negative) există câteva convenţii de reprezentare care vor fi descrise şi discutate în continuare.

Notă: valoarea zecimală corespunzătoare unei secvenţe de biţi reprezentând un număr întreg cu semn nu poate fi calculată dacă nu este specificat tipul reprezentării.

2.1.1 Mărime şi semn

În primul mod de scriere, problema reprezentării semnului unui număr poate fi redusă la a utiliza cel mai semnificativ bit (msb) pentru a reprezenta semnul: msb e 0 pentru un număr pozitiv, respectiv 1 pentru un număr negativ. Restul biţilor din număr indică mărimea (sau valoarea absolută). Aşadar, într-un octet, cu doar 7 biţi (în afară de bitul de semn), mărimea poate lua valori între 0000000 (0) şi 1111111 (127). Astfel se pot reprezenta numere de la -127_{10} până la 127_{10}, odată ce se adaugă şi bitul de semn (al optulea bit). O consecinţă a acestei reprezentări este aceea că există 2 moduri de a reprezenta valoarea zero, 00000000 (+0) şi 10000000 (-0).

Pentru a obţine reprezentarea cu mărime şi semn a unei valori zecimale, se aplică algoritmul următor:

- dacă numărul e pozitiv (43_{10}):
    - se transformă numărul in binar natural (00101011)
- dacă numărul e negativ (-43_{10}):
    - se transformă numărul pozitiv în binar natural (00101011)
    - se complementează bitul de semn (10101011)

Pentru a obţine valoarea zecimală a unui număr scris în reprezentarea mărime şi semn, se aplică următorul algoritm:

- se transformă mărimea în valoare zecimală;
- se transformă bitul de semn în + (dacă msb=0) sau în – (dacă msb=1).

2.1.2 Complement faţă de unu

În complement faţă de unu, *numerele pozitive* sunt reprezentate, ca de obicei, în binar natural, în timp ce *numerele negative* sunt reprezentate prin complementul valorii pozitive.

În complement faţă de unu (exact ca şi în mărime şi semn):

- bitul cel mai din stânga indică semnul (1 este negativ, 0 este pozitiv),
- există două reprezentări ale lui 0: 00000000 (+0) şi 11111111 (-0),
- pe un octet pot fi reprezentate numere între -127_{10} şi $+127_{10}$.

Pentru a obţine reprezentarea în complement faţă de unu a unei valori zecimale, se aplică următorul algoritm:

- dacă numărul e pozitiv (43_{10}):
    - se transformă numărul în binar natural (00101011)
- dacă numărul e negativ (-43_{10}):
    - se transformă numărul pozitiv în binar natural (00101011)
    - se complementează toţi biţii săi (11010100)

Pentru a obţine valoarea zecimală a unui număr scris în complement faţă de unu, se urmează algoritmul:

- dacă bitul de semn (msb) este 0 (00101011):
    - se transformă numărul în zecimal (43_{10})

- dacă bitul de semn (msb) este 1 (11010100):
    - se complementează toţi biţii săi pentru a obţine reprezentarea în binar natural (00101011)
    - se transformă numărul în zecimal (43_{10})
    - se plasează semnul în faţa numărului (-43_{10}).

2.1.3 Complement faţă de doi

Nevoia unui algoritm special pentru a aduna numere negative reprezentate în mărime şi semn sau în complement faţă de unu sunt rezolvate de o convenţie numită complement faţă de doi. În complement faţă de doi, *numerele pozitive* sunt reprezentate ca de obicei, în binar natural, în timp ce *numerele negative* sunt obţinute prin complementarea tuturor biţilor numărului pozitiv fără semn, la care se adună 1.

În complement faţă de doi:

- bitul cel mai din stânga indică semnul (1 este negativ, 0 este pozitiv),
- există o singură reprezentare a lui 0: 00000000,
- pe un octet pot fi reprezentate numere între -12810 şi +12710.

Pentru a obţine reprezentarea în complement faţă de doi a unei valori zecimale, se aplică următorul algoritm:

- dacă numărul e pozitiv (43_{10}):
    - se transformă numărul în binar natural (00101011)
- dacă numărul e negativ (-43_{10}):
    - se transformă numărul pozitiv în binar natural (00101011)
    - se complementează toţi biţii săi (11010100)
    - se adaugă 1 (11010101)

Pentru a obţine valoarea zecimală a unui număr scris în complement faţă de doi, se aplică următorul algoritm:

- dacă bitul de semn (msb) este 0 (00101011):
    - se transformă numărul în zecimal (43_{10})
- dacă bitul de semn (msb) este 1 (11010101):
    - se scade 1 (11010100)

- se complementează toţi biţii săi pentru a obţine reprezentarea în binar natural (00101011)
- se transformă numărul în zecimal (43_{10})
- se plasează semnul în faţa numărului (-43_{10}).

2.1.4 Tabel de comparaţie

Tabelul 1 arată întregii pozitivi şi negativi care pot fi reprezentaţi folosind 4 biţi.

Tabelul 1. Reprezentarea pe 4 biţi a numerelor întregi

Zecimal	Fără semn	Mărime şi semn	Complement faţă de unu	Complement faţă de doi
+16	N/A	N/A	N/A	N/A
+15	1111	N/A	N/A	N/A
+14	1110	N/A	N/A	N/A
+13	1101	N/A	N/A	N/A
+12	1100	N/A	N/A	N/A
+11	1011	N/A	N/A	N/A
+10	1010	N/A	N/A	N/A
+9	1001	N/A	N/A	N/A
+8	1000	N/A	N/A	N/A
+7	0111	0111	0111	0111
+6	0110	0110	0110	0110
+5	0101	0101	0101	0101
+4	0100	0100	0100	0100
+3	0011	0011	0011	0011
+2	0010	0010	0010	0010
+1	0001	0001	0001	0001
+0	0000	0000	0000	0000
−1	N/A	1001	1110	1111
−2	N/A	1010	1101	1110
−3	N/A	1011	1100	1101
−4	N/A	1100	1011	1100
−5	N/A	1101	1010	1011
−6	N/A	1110	1001	1010
−7	N/A	1111	1000	1001
−8	N/A	N/A	N/A	1000
−9	N/A	N/A	N/A	N/A

2.2 Instrucţiuni de tip transfer de date pentru x86 în modul real

Instrucţiunile de transfer de date sunt acele instrucţiuni CPU utilizate pentru a:

- copia o constantă într-un registru sau într-o locaţie de memorie;
- copia date dintr-o locaţie de memorie într-un registru sau vice versa;
- copia date de la/către dispozitivele I/O.

Cele mai importante instrucţiuni de transfer de date oferite de setul de instrucţiuni al arhitecturii x86 care pot fi folosite fără a utiliza neapărat memoria sunt listate în Tabelul 2.

Tabelul 2. O parte din instrucţiunile de transfer de date în x86

Instrucţiune	Mod de utilizare	Descriere
MOV – Move Data	MOV d, s	Copiază s la d
XCHG – Exchange Data	XCHG d, s	Schimbă s cu d

2.3 Instrucţiuni de tip procesare de date pentru x86 în modul real

Instrucţiunile de procesare de date sunt instrucţiunile care realizează operaţiile aritmetice (adunare, scădere, înmulţire, împărţire etc.) şi operaţiile logice (şi, sau, sau exclusiv, deplasare, rotaţie etc.). Fiecare microprocesor are setul său specific de instrucţiuni, care include diferite tipuri de instrucţiuni de procesare de date. Cele mai importante instrucţiuni de procesare de date din setul de instrucţiuni x86 sunt enumerate în Tabelul 3. Detalii şi exemple de utilizare a unora dintre instrucţiunile de procesare de date au fost date în Laboratorul 1 şi se află de asemenea şi în secţiunea 2.6.

Majoritatea instrucţiunilor de procesare de date modifică unele fanioane sau pe toate (**CF**, **ZF**, **SF**, **OF**, **PF**, **AF**), în funcţie de rezultatul operaţiei. **CF** şi **ZF** se referă la operaţii cu numere fără semn reprezentate în binar natural, în timp ce **ZF**, **SF** şi **OF** se referă la operaţii cu numere cu semn reprezentate în complement faţă de doi.

Se observă că microprocesorul nu ştie dacă operanzii sau rezultatul sunt cu semn sau fără semn; acesta pur şi simplu modifică toate fanioanele în funcţie de rezultat. În consecinţă, dându-se următorul bloc de instrucţiuni:

```
mov   AL, 80h
add   AL, 90h
```

fanioanele aritmetice vor fi modificate astfel:

- **CF** va fi 1, deoarece *rezultatul fără semn* al operaţiei: 80h (128_{10}) + 90h (144_{10}) = 110h (272_{10}) este mai mare decât numărul maxim reprezentabil pe 8 biţi (255_{10}).

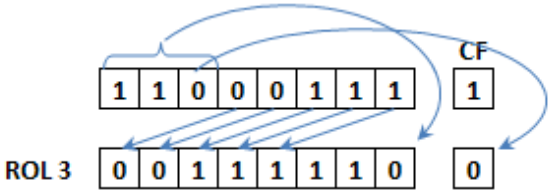
- **ZF** va fi 0, întrucât rezultatul instrucţiunii este nenul.
- **PF** va fi 0, deoarece *cel mai nesemnificativ octet al rezultatului* (10h) conţine un număr impar de cifre de 1.
- **SF** va fi 0, deoarece semnul *rezultatului pe 8 biţi* (10h) este 0 (pozitiv).
- **OF** va fi 1, deoarece *rezultatul cu semn* al operaţiei: 80h (-128_{10}) + 90h (-112_{10}) = 110h (-220_{10}) este mai mic decât numărul minim cu semn reprezentabil pe 8 biţi (-128_{10}).

Tabelul 3. Instrucţiuni de procesare de date în x86

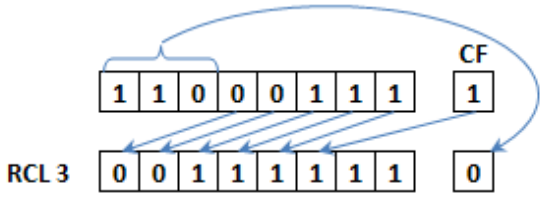
Instrucţiune	Mod de utilizare	Descriere
INC – Increment	INC s	Incrementează s
DEC – Decrement	DEC s	Decrementează s
ADD – Add	ADD d, s	Adună s la d
ADC – Add with Carry	ADC d, s	Adună s şi CF la d
SUB – Subtract	SUB d, s	Scade s din d
SBB – Subtract with Borrow	SBB d, s	Scade s şi CF din d
MUL – Multiply	MUL s	Înmulţeşte acumulatorul cu s
DIV – Divide	DIV s	Împarte acumulatorul la s
CMP – Compare	CMP s1, s2	Scade s2 din s1 fără modificarea operanzilor
NOT – Complement	NOT s	Complementează s
AND – Logic AND	AND d, s	Se realizează ŞI logic între s şi d şi se stochează rezultatul în d
OR – Logic OR	OR d, s	Se realizează SAU logic între s şi d şi se stochează rezultatul în d
XOR – Exclusive OR	XOR d, s	Se realizează SAU EXCLUSIV logic între s şi d şi se stochează rezultatul în d
SHL – Shift Left	SHL s, num	Se deplasează s la stânga cu num poziţii
SHR – Shift Right	SHR s, num	Se deplasează s la dreapta cu num poziţii
ROL – Rotate Left	ROL s, num	Se roteşte s la stânga cu num poziţii
ROR – Rotate Right	ROR s, num	Se roteşte s la dreapta cu num poziţii
RCL – Rotate Left with Carry	RCL s, num	Se roteşte CF□s la stânga cu num poziţii
RCR – Rotate Right with Carry	RCR s, num	Se roteşte CF□s la dreapta cu num poziţii
TEST – Compare using AND	TEST s1, s2	Se realizează ŞI logic între s1 şi s2 fără a modifica operanzii

Operaţiile de deplasări și rotaţii sunt exemplificate în următoarele figuri:

Deplasarea la stânga cu 3 poziții din exemplul de mai sus elimină cei mai semnificativi 3 biți din număr, introducând 3 zerouri pe cele mai nesemnificative poziții ale acestuia. Ultimul bit deplasat din număr suprascrie valoarea anterioară din **CF**.



Operațiile de tip **ROL** sau **ROR** rotesc numerele cu un număr de poziții, independent de valoarea anterioară stocată în **CF**. Rotația la stânga cu 3 poziții din exemplul de mai sus mută cei mai semnificativi 3 biți ai numărului pe cele mai nesemnificate poziții, deplasându-i cu 3 poziții la stânga pe ceilalți 5 biți. Ultimul bit rotit din număr suprascrie valoarea anterioară din **CF**.



În cazul operațiilor de tip **RCL** sau **RCR**, **CF** poate fi considerat ca făcând parte din numărul rotit. În exemplul de mai sus sunt rotiți 3 biți la stânga; ultimul dintre aceștia suprascrie valoarea anterioară din **CF**.

2.4 Instrucţiuni de tip salt pentru x86 în modul real

Instrucţiunile de salt sunt instrucţiuni de control al programului. Ele sunt excepţii în *execuţia secvenţială normală a instrucţiunilor*, spunându-i microprocesorului să continue execuţia instrucţiunilor de la o adresă specificată în memorie (şi nu de la instrucţiunea următoare).

Arhitectura x86 oferă o instrucţiune de salt necondiţionat (**JMP**) şi câteva instrucţiuni de salt condiţionat (**Jxx**), prezentate în Tabelul 4.

Instrucţiunea de salt necondiţionat execută întotdeauna saltul la destinaţia specificată. Destinaţia poate fi specificată ca o etichetă, o adresă stocată într-un registru sau o adresă stocată într-o locaţie de memorie. În cazul salturilor condiţionate, o condiţie booleană (referitoare la fanioane) este mai întâi verificată şi, dacă aceasta este îndeplinită, se execută saltul la instrucţiunea specificată.

Notă importantă. Unele dintre instrucţiunile de salt condiţionat (cele care utilizează cuvintele "above" şi "below") se referă strict la numere fără semn (condiţiile lor implică valoarea lui **CF**). Alte instrucţiuni de salt condiţionat (cele care utilizează cuvintele "greater" şi "lower") se referă strict la numere cu semn (condiţiile lor implică valorile lui **SF** şi **OF**). Programatorul este responsabil pentru folosirea corectă a instrucţiunilor de salt condiţionat (după o instrucţiune aritmetică sau logică), deoarece numai el cunoaşte interpretarea numerelor pe care le foloseşte.

Tabelul 4. Instrucţiuni de salt în x86

Instrucţiune	Mod de utilizare	Condiţie	Descriere
JMP	JMP d	N/A	Salt la destinaţie (necondiţionat)
JA | JNBE	JA label	(CF)=0 AND (ZF)=0	Salt la etichetă dacă above | not below
JAE | JNB | JNC	JAE label	(CF)=0	Salt la etichetă dacă above or equal | not below | not carry
JB | JNAE | JC	JB label	(CF)=1	Salt la etichetă dacă below | not above or equal | carry
JBE | JNA	JBE label	(CF)=1 OR (ZF)=1	Salt la etichetă dacă below or equal | not above
JE | JZ	JE label	(ZF)=1	Salt la etichetă dacă equal | zero
JG | JNLE	JG label	(SF)=(OF) AND (ZF)=0	Salt la etichetă dacă greater | not lower or equal
JGE | JNL	JGE label	(SF)=(OF)	Salt la etichetă dacă greater or equal | not lower
JL | JNGE	JL label	(SF)!=(OF)	Salt la etichetă dacă lower | not greater or equal
JLE | JNG	JLE label	(SF)!=(OF) OR (ZF)=1	Salt la etichetă dacă lower or equal | not greater
JNE | JNZ	JNE label	(ZF)=0	Salt la etichetă dacă not equal | not zero
JNO	JNO label	(OF)=0	Salt la etichetă dacă not overflow
JNP | JPO	JNP label	(PF)=0	Salt la etichetă dacă not parity | parity odd
JNS	JNS label	(SF)=0	Salt la etichetă dacă not signed | positive
JO	JO label	(OF)=1	Salt la etichetă dacă overflow
JP | JPE	JP label	(PF)=1	Salt la etichetă dacă parity | parity even
JS	JS label	(SF)=1	Salt la etichetă dacă signed | negative

2.4.1 Folosirea salturilor condiţionate pentru a crea structuri de decizie

Structurile de decizie cu care sunteţi familiari din limbajele de programare de nivel înalt (if-then-else, switch-case etc.) pot fi implementate în limbaj de asamblare folosind salturi condiţionate şi necondiţionate. În Tabelul 5 sunt enumerate câteva exemple de blocuri de instrucţiuni în limbaj de asamblare echivalente cu structuri decizionale if-then-else.

Tabelul 5. Exemple de structuri decizionale

Pseudo-cod	Echivalent în asamblare	Note

if (AL > 13h){
  BX = 1234h;
}else{
  BX = 4321h;
}
	sub AL, 13h
	jbe else
	then: mov BX, 1234h
	jmp endif
	else: mov BX, 4321h
	endif: …
		În acest exemplu se consideră numerele ca fiind fără semn. Dacă numerele ar fi fost cu semn, atunci JLE ar fi fost folosit în loc de JBE.

if (AL == 0h){
  BX = 100h
}else if (AL == 1h){
  BX = 200h
}else{
  BX = 300h
}
	cmp AL, 0h
	jne else1
	then: mov BX, 100h
	jmp endif
	else1: cmp AL, 1h
	jne else2
	mov BX, 200h
	jmp endif
	else2: mov BX, 300h
	endif: …	

2.5 Instrucţiuni de tip ciclu pentru x86 în modul real

Ciclurile sunt de asemenea instrucţiuni de control. Ele sunt similare salturilor condiţionate, întrucât execuţia lor implică următoarele: a) o condiţie booleană (referitoare la fanioane) este verificată şi b) dacă această condiţie este îndeplinită, saltul la instrucţiunea specificată este executat. Totuşi, în afara celor menţionate mai sus, toate ciclurile decrementează CX, numărătorul implicit al arhitecturii x86. Ciclurile sunt enumerate în Tabelul 6.

2.5.1 Folosirea ciclurilor pentru crearea de structuri repetitive

Structurile repetitive cu care sunteţi familiari din limbajele de programare de nivel înalt (bucle for, while etc.) pot fi implementate în limbaj de asamblare folosind cicluri. În Tabelul 7 sunt enumerate câteva exemple de blocuri de instrucţiuni în limbaj de asamblare echivalente cu structuri repetitive din limbajele de programare de nivel înalt.

Tabelul 6. Ciclurile în x86

Instrucţiune	Mod de folosire	Condiţie testată	Descriere
LOOP	LOOP label	(CX) != 0	Decrementează CX (fără a modifica fanioanele) şi execută salt la etichetă dacă CX nu este zero
LOOPE | LOOPZ	LOOPE label	(CX) != 0 AND (ZF)=1	Decrementează CX (fără a modifica fanioanele) şi execută salt la etichetă dacă CX nu este zero şi ZF este 1.
LOOPNE | LOOPNZ	LOOPNE label	(CX) != 0 AND (ZF)=0	Decrementează CX (fără a modifica fanioanele) şi execută salt la etichetă dacă CX nu este zero şi ZF este zero.

Tabelul 7. Exemple de structuri repetitive

Pseudo-cod	Echivalent în asamblare

for (int index=9; index>0; index--){
alpha = alpha*2+beta;
}
	mov CX, 9h
	for: shl AX, 1
	add AX, BX
	loop for

count = 10h;
result = 15h;
while ((count > 0) && (result != 21h)){
result = result+2;
count = count-1;
}
	mov CX, 10h
	mov DX, 15h
	while: add DX, 2h
	cmp DX, 21h
	loopne while

2.6 Exerciţii

2.6.1 Exerciţiul 1

Obiectiv. Acest exerciţiu prezintă diferitele metode de conversie din zecimal în binar (şi vice-versa) pentru întregi cu semn, folosind convenţiile de reprezentare a numerelor cu semn (semn şi mărime, complement faţă de unu şi complement faţă de doi). De asemenea, se exemplifică şi câteva operaţii de adunare binară.

Cerinţă. Să se transforme numerele +/-5 şi +/-12 în binar (utilizând diferitele metode de reprezentare a numerelor) şi să se calculeze următoarele sume: (5 + 12), (-5 + 12), (-12 + 5) şi (12 + -12), folosind algoritmul de adunare normală pentru numere binare. Să se transforme rezultatul înapoi în zecimal.

Soluţie.

Se transformă numerele din zecimal în binar.

Tabelul 8. Transformări zecimal - binar

Zecimal	Mărime şi semn	Complement faţă de unu	Complement faţă de doi
+5	00000101	00000101	00000101
-5	10000101	11111010	11111011
+12	00001100	00001100	00001100
-12	10001100	11110011	11110100

Regulile de reprezentare a numerelor în aceste trei convenţii sunt următoarele:

- Numerele pozitive se reprezintă identic, indiferent de convenţie.
- În „mărime şi semn”, numerele negative diferă de cele pozitive numai prin bitul de semn.
- În „complement faţă de 1”, mărimea numărului negativ se obţine din reprezentarea precedentă prin complementare bit cu bit; convenţia pentru bitul de semn se păstrează.
- În „complement faţă de 2”, mărimea numărului negativ se obţine din reprezentarea precedentă prin adunarea unei cifre binare 1 la LSB; convenţia pentru bitul de semn se păstrează.

Se adună numerele binare în mărime şi semn folosind algoritmul de adunare normală şi se transformă rezultatul înapoi în zecimal.

5 + 12	-5 + 12	-12 + -5	12 + -12
00000101	10000101	10001100	00001100
00001100	00001100	10000101	10001100
_			
00010001	10010001	00010001	10011000
17	-17	17	-24

Se adună numerele binare în complement faţă de unu folosind algoritmul de adunare normală şi se transformă rezultatul înapoi în zecimal.

5 + 12	-5 + 12	-12 + -5	12 + -12
00000101	11111010	11110011	00001100
00001100	00001100	11111010	11110011
00010001	00000110	11101101	11111111
17	6	-18	0

Se adună numerele binare în complement faţă de doi folosind algoritmul de adunare normală şi se transformă rezultatul înapoi în zecimal.

5 + 12	-5 + 12	-12 + -5	12 + -12
00000101	11111011	11110100	00001100
00001100	00001100	11111011	11110100
00010001	00000110	11101101	00000000
17	7	-19	0

Notă: algoritmul de adunare normală a numerelor binare poate fi utilizat cu succes (se obţin rezultate corecte) numai dacă numerele zecimale cu semn sunt reprezentate în binar în complement faţă de doi. Acesta este motivul pentru care reprezentarea în complement faţă de doi este cel mai des folosită în sistemele computaţionale.

2.6.2 Exerciţiul 2

Cerinţă. Să se transforme în zecimal următoarele secvenţe de biţi: 00110011, 10110011, 01010101, 11010101. Se consideră că secvenţele de biţi de mai sus sunt reprezentate în binar folosind: a) reprezentarea în semn si mărime; b) reprezentarea în complement faţă de unu; c) reprezentarea în complement faţă de doi.

2.6.3 Exerciţiul 3

Obiectiv. Înţelegerea efectului executării instrucţiunilor ADD, ADC, SUB şi SBB şi a rolului fanionului de transport (CF).

Cerinţă. Să se scrie un program care adună/scade două numere fără semn pe 16 biţi, stocate iniţial în CX şi DX. Cele două operaţii pe 16 biţi trebuie făcute în doi paşi folosind operaţii pe 8 biţi.

Soluţie.

1. Se porneşte emulatorul.

2. Se scrie următorul program, folosind fereastra Source Code:

```
                org     100h
init:           mov     CX, 6234h
                mov     DX, 49D0h

sum:            mov     AL, CL
                add     AL, DL
                mov     AH, CH
                adc     AH, DH

dif:            mov     AX, 0h
                mov     AL, CL
                sub     AL, DL
                mov     AH, CH
                sbb     AH, DH

                int     20h
```

3. Explicaţia programului

3.1. Prima linie a acestui program (org 100h) nu este o instrucţiune. Este o directivă de asamblare care specifică faptul ca următoarea instrucţiune (şi în consecinţă întregul program) va fi încărcată în memorie (în segmentul de cod) începând cu adresa 100h.

3.2. A doua instrucţiune este precedată de o etichetă (init) care poate fi folosită pentru a face referire la această instrucţiune dintr-un loc diferit din program. În acest caz eticheta este utilizată doar pentru a face codul mai uşor de înţeles.

3.3. Blocul de instrucţiuni etichetat init iniţializează registrele CX şi DX cu cele două numere pe 16 biţi.

3.4. Blocul de instrucţiuni etichetat sum realizează suma celor două numere în doi paşi: a) adună cei mai puţin semnificativi octeţi în AL şi b) adună cei mai semnificativi octeţi şi fanionul de transport (CF) în AH.

3.5. Blocul de instrucţiuni etichetat dif realizează diferenţa celor două numere în doi paşi: a) scade cel mai puţin semnificativ octet al celui de-al doilea număr (din DL) din cel mai puţin semnificativ octet al primului număr (din CL) şi stochează rezultatul în AL şi b) scade cel mai semnificativ octet al celui de-al doilea număr (din DH) şi fanionul de transport (CF) din cel mai semnificativ octet al primului număr (din CH) şi stochează rezultatul în AH.

3.6. Instrucţiunea int 20h este o întrerupere software. Aceasta opreşte programul curent şi returnează controlul sistemului de operare.

4. Se salvează programul (File menu -> Save As submenu) cu numele lab2_prog1.asm.

5. Se compilează programul:

5.1. Se face click pe butonul Compile pentru a compila programul.

5.2. Veţi fi îndrumaţi să salvaţi fişierul executabil. Se salvează cu numele recomandat (lab2_prog1.com).

5.3. Se vizualizează statusul compilării în caseta Assembler Status dialog. Dacă programul a fost editat corect, mesajul ar trebui sa fie "lab2_prog1.com is assembled successfully into 27 bytes."

6. Se încarcă programul executabil în emulator.

6.1. Se face click pe butonul Run pentru a încărca programul în emulator şi a-l executa.

7. Se execută programul pas cu pas, se observă modificările apărute în registre, locaţiile de memorie, fanioane etc. şi se notează observaţii.

7.1. Se face click pe butonul Reload pentru a reîncărca programul executat.

7.2. Se execută toate instrucţiunile pas cu pas, făcând click succesiv pe butonul Single Step.

8. Se scriu concluzii referitoare la efectul diferitelor instrucţiuni asupra registrelor, fanioanelor şi locaţiilor de memorie.

2.6.4 Exerciţiul 4

Obiectiv. Înţelegerea efectului executării instrucţiunilor AND, OR, NOT şi SHL.

Cerinţă. Să se scrie un program care implementează funcţia logică prezentată mai jos, folosind ca intrări următoarele numere pe 8 biţi: alpha=26h, beta=3Fh, gamma=99h, theta=8Dh.

NOT ((alpha << 8 OR gamma) AND (beta << 8 OR theta))

Notă. OR, AND, NOT sunt operatorii logici, iar <<, >> sunt operatorii de deplasare la stânga, respectiv la dreapta.

Soluţie.

1. Se porneşte emulatorul.

2. Se utilizează fereastra Source Code pentru a scrie următorul program:

```
        org     100h

init:   mov     AX, 0h
        mov     BX, 0h

part1:  mov     AL, 26h
        shl     AX, 8h
        or      AL, 99h

part2:  mov     BL, 3Fh
        shl     BX, 8h
        or      BL, 8Dh

part3:  and     AX, BX
        not     AX
        int     20h
```

3. Explicaţia programului

3.1. Prima linie a acestui program (org 100h) nu este o instrucţiune. Este o directivă de asamblare care specifică faptul ca următoarea instrucţiune (şi în consecinţă întregul program) va fi încărcată în memorie (în segmentul de cod) începând cu adresa 100h.

3.2. Blocul de instrucţiuni etichetat init iniţializează registrele AX şi BX cu zero.

3.3. Blocul de instrucţiuni etichetat part1 realizează prima parte a funcţiei logice (alpha << 8 OR gamma), folosind registrul AX. alpha=26h este încărcat în AL, apoi AX este deplasat la stânga cu 8 poziţii şi în final este executată operaţia SAU logic între AL şi gamma=99h.

3.4. Blocul de instrucţiuni etichetat part2 realizează a doua parte a funcţiei logice (beta << 8 OR theta), folosind registrul BX. beta=3Fh este încărcat în BL, apoi BX este deplasat la stânga cu 8 poziţii şi în final este executată operaţia SAU logic între BL şi theta=8Dh.

3.5. Blocul de instrucţiuni etichetat part3 realizează ultima parte a funcţiei logice: execută operaţia ŞI logic între AX şi BX şi apoi complementează rezultatul în AX.

3.6. În final, instrucţiunea int 20h termină programul şi returnează controlul sistemului de operare.

4. Se continuă acest exerciţiu, urmând aceiaşi paşi ca în exerciţiul anterior.

2.6.5 Exerciţiul 5

Obiectiv. Înţelegerea efectului executării instrucţiunilor XOR şi SHR.

Cerinţă. Să se scrie un program care implementează funcţia logică prezentată mai jos, utilizând ca intrări următoarele numere pe 16 biţi: alpha=1A26h, beta=553Fh.

(alpha >> 3) XOR (beta >> 5)

Notă. XOR este operatorul obişnuit şi >> este operatorul de deplasare la dreapta.

2.6.6. Exerciţiul 6

Obiectiv. Înţelegerea efectului executării instrucţiunilor ROR, ROL, RCR şi RCL şi observarea diferenţei între rotirile cu carry şi rotirile fără carry.

Cerinţă. Să se scrie un program care implementează funcţiile logice prezentate mai jos, folosind ca intrări următoarele numere pe 8 biţi: alpha=26h, beta=3Fh.

(alpha ROR 2) AND (beta ROL 5)

(alpha RCR 2) AND (beta RCL 5)

Note. ROR, ROL, RCR şi RCL sunt operatorii de rotire la dreapta/stânga şi dreapta/stânga cu carry.

2.6.7 Exerciţiul 7

Obiectiv. Înţelegerea efectului executării instrucţiunilor CMP şi JA. Înţelegerea etichetelor şi a etichetelor care fac referinţă la instrucţiuni.

Cerinţă. Să se scrie un program care găseşte maximul dintre trei numere pe 16 biţi fără semn (alpha=1234h, beta=8004h, gamma=072Fh), stocate iniţial în AX, BX şi CX.

Soluţie.

1. Se porneşte emulatorul.

2. Se utilizează fereastra Source Code pentru a scrie următorul program:

```
                org     100h

init:           mov     AX, 1234h
                mov     BX, 8004h
                mov     CX, 072Fh

                mov     DX, AX

compareBX:      cmp     DX, BX
                ja      compareCX
                mov     DX, BX

compareCX:      cmp     DX, CX
                ja      exit
                mov     DX, CX

exit:           int     20h
```

3. Explicaţia programului

3.1. Blocul de instrucţiuni etichetat init iniţializează registrele AX, BX şi CX cu numerele pe 16 biţi.

3.2. Mergând mai departe, registrul DX, care va stoca maximul, este încărcat cu valoarea primului număr (presupunem că primul element este maximul).

3.3. Instrucţiunea cmp DX, BX compară primele două numere, scăzând valoarea stocată în BX din valoarea stocată în DX. Rezultatul nu este stocat nicăieri, dar fanioanele sunt modificate în consecinţă. De exemplu, dacă valoarea fără semn din BX este mai mare decât numărul fără semn din DX, atunci CF va fi 1.

3.4. Instrucţiunea ja compareCX utilizează fanionul de transport pentru a lua o decizie: se face saltul la eticheta compareCX sau se continuă cu următoarea instrucţiune? Se foloseşte această instrucţiune (JA) şi nu JG deoarece numerele pe care le comparăm sunt fără semn. Dacă numărul fără semn din BX a fost mai mare decât numărul fără semn din DX, atunci microprocesorul continuă cu următoarea instrucţiune, care înlocuieşte maximul (din DX) cu noul maxim (din BX). Altfel, microprocesorul ignoră valoarea din BX şi sare la eticheta compareCX.

3.5. Instrucţiunea cmp DX, CX compară maximul cu al treilea număr, scăzând valoarea stocată în CX din valoarea stocată în DX. Rezultatul nu este stocat nicăieri, însă fanioanele sunt modificate în consecinţă. De exemplu, dacă valoarea fără semn din CX este mai mare decât valoarea fără semn din DX, atunci CF va fi 1.

3.6. Instrucţiunea ja exit utilizează fanionul de transport pentru a lua o decizie: se face saltul la eticheta exit sau se continuă cu următoarea instrucţiune? Se foloseşte această instrucţiune (JA) şi nu JG deoarece numerele pe care le comparăm sunt fără semn. Dacă numărul fără semn din CX a fost mai mare decât numărul fără semn din DX, atunci microprocesorul continuă cu următoarea instrucţiune, care înlocuieşte maximul (din DX) cu noul maxim (din CX). Altfel, microprocesorul ignoră valoarea din CX şi sare la eticheta exit.

3.7. În final, instrucţiunea int 20h termină programul şi returnează controlul sistemului de operare.

3.8. Se decide care dintre cele trei numere fără semn este mai mare şi care dintre cele două salturi va fi executat (şi care nu) înainte de a executa programul!

4. Se salvează programul (File menu -> Save As submenu) cu numele lab2_ex7.asm.

5. Se compilează programul:

5.1. Se face click pe butonul Compile pentru a compila programul.

5.2. Veţi fi îndrumaţi să salvaţi fişierul executabil. Se salvează cu numele recomandat (lab2_ex7.com).

5.3. Se vizualizează statusul compilării în caseta Assembler Status dialog. Dacă programul a fost editat corect, mesajul ar trebui să fie “lab2_ex7.com is assembled successfully into 25 bytes.”

5.4. Se face click pe View button -> Symbol Table pentru a vizualiza tabelul de simboluri asociat programului. Se observă ca etichetele init, compareBX, compareCX şi exit sunt asociate cu nişte offseturi (100h, 10Bh, 110h, 117h), reprezentând adresele de memorie unde sunt stocate instrucţiunile corespunzătoare.

6. Se încarcă programul executabil în emulator.

6.1. Se face click pe butonul Run pentru a încărca programul în emulator şi a-l executa.

7. Se execută programul pas cu pas, se observă modificările apărute în registre, locaţiile de memorie, fanioane etc. şi se notează observaţii.

7.1. Se face click pe butonul Reload pentru a reîncărca programul executat.

7.2. Se observă ca instrucţiunile ja compareCX şi ja exit sunt înlocuite în emulator de instrucţiunile jnbe 0111h şi jnbe 0117h. Reţineţi (a se vedea Tabelul 3) că instrucţiunile JA şi JNBE sunt echivalente şi că în lista de simboluri etichetele compareCX şi exit au fost asociate cu adresele 110h şi 117h.

7.3. Se execută toate instrucţiunile pas cu pas, făcând click succesiv pe butonul Single Step.

8. Se scriu concluzii referitoare la efectul diferitelor instrucţiuni asupra registrelor, fanioanelor şi locaţiilor de memorie.

2.6.8 Exerciţiul 8

Obiectiv. Înţelegerea diferenţei dintre procesarea numerelor cu semn şi procesarea numerelor fără semn.

Cerinţă. Să se modifice programul anterior pentru a se găsi maximul dintre trei *numere pe 16 biţi cu semn* (alpha=1234h, beta=8004h, gamma=072Fh), stocate iniţial în AX, BX şi CX.

Indicaţie. Înainte de a scrie şi executa programul, decideţi care dintre cele trei numere cu semn este mai mare. În timpul execuţiei programului, înainte de orice instrucţiune de salt, analizaţi fanioanele făcând click pe butonul Analyze din fereastra Flags.

2.6.9 Exerciţiul 9

Obiectiv. Practicarea folosirii instrucţiunilor de salt condiţionat.

Cerinţă. Să se modifice programul de la Exerciţiul 7 pentru a se găsi minimul dintre aceleaşi trei *numere pe 16 biţi fără semn* (alpha=1234h, beta=8004h, gamma=072Fh), stocate iniţial în AX, BX şi CX.

2.6.10 Exerciţiul 10

Obiectiv. Practicarea folosirii instrucţiunilor de salt condiţionat.

Cerinţă. Să se modifice programul de la Exerciţiul 8 pentru a se găsi minimul dintre aceleaşi trei *numere pe 16 biţi cu semn* (alpha=1234h, beta=8004h, gamma=072Fh), stocate iniţial în AX, BX şi CX.

2.7 Anexa 1. Exemple de instrucţiuni de transfer și procesare de date

MOV – Copiază Date

Mod de utilizare: MOV d, s

Operanzi:

- d – registru de uz general, registru de segment (cu excepţia CS) sau locaţie de memorie
- s – constantă cu adresare imediată, registru de uz general, registru segment sau locaţie de memorie

Efecte: Copiază sursa la destinaţie, suprascriind valoarea destinaţiei: **(d) ← (s)**.

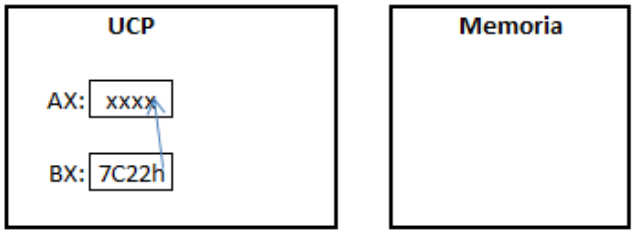
Fanioane: niciunul

Notă: Argumentele trebuie să aibă aceeaşi dimensiune (byte, word).

Exemplu

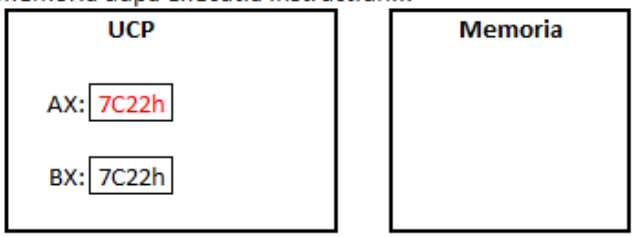
MOV AX, BX

1. UCP si memoria inainte de executia instructiunii:



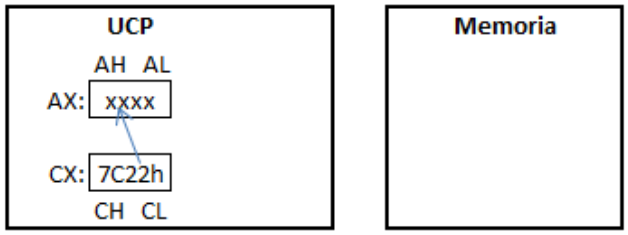
2. Valoarea 7C22h, care este stocata in registrul BX, este copiata in registrul AX.

3. UCP si memoria dupa executia instructiunii:

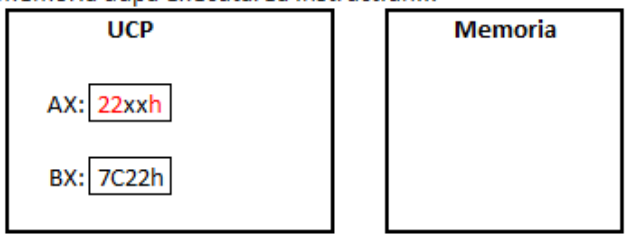


MOV AH, CL

1. UCP si memoria inainte de executarea instructiunii:



2. Valoarea 22h, care este stocata in registrul CL, este copiata in registrul AH.
3. UCP si memoria dupa executarea instructiunii:

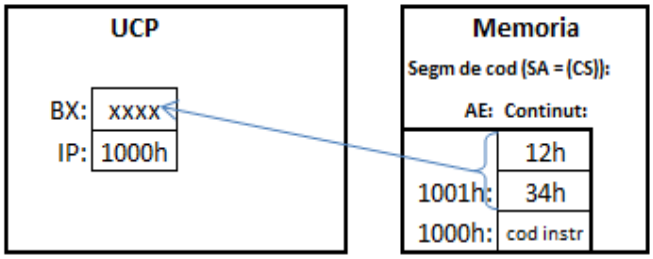


Exemplu

Exemplu

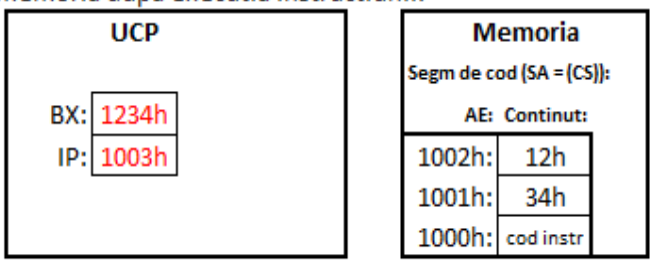
MOV BX, 1234h

1. UCP si memoria inainte de executia instructiunii:



2. Valoarea 1234h, care este stocata in segmentul de cod (imediat dupa codul instructiunii) este copiata in registrul BX

3. UCP si memoria dupa executia instructiunii:



CMP – Compararea a doi operanzi

Mod de utilizare: CMP s1, s2

Argumente:

- s1, s2 – constantă adresată imediat sau registru pe 8 biţi sau 16 biţi sau locaţie de memorie;

Efecte: Se scade s2 din s1: **(s1) – (s2)**. Fanioanele sunt setate în acelaşi mod ca în cazul instrucţiunii SUB, dar nu este salvat rezultatul scăderii.

Fanioane: CF, ZF, OF, SF, AF, şi PF sunt modificate în acord cu rezultatul.

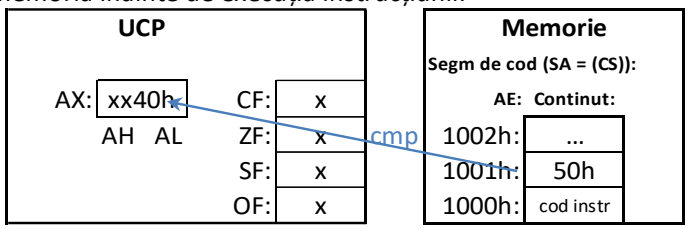
Notă:

- de obicei următoarea operaţie va fi un salt condiţionat spre executarea unei operaţii în acord cu rezultatul comparaţiei;
- doar un singur argument de memorie este permis, iar ambele argumente trebuie să aibă aceeaşi dimensiune.

Exemplu

CMP AL, 50h

1. UCP și memoria înainte de execuția instrucțiunii:



2. Valoarea 50h este scăzută din valoarea 40h. Rezultatul pe 8 biți (F0h) nu este stocat nicăieri dar fanioanele sunt modificate după cum urmează:

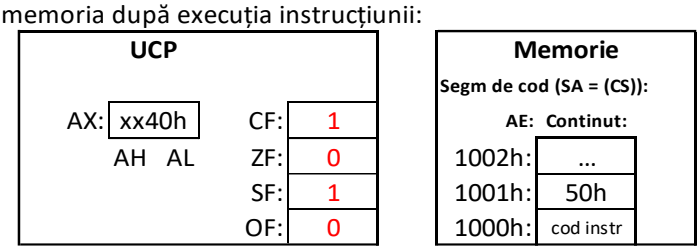
a) CF este 1, deoarece operația: 40h (64_{10}) - 50h (80_{10}) = F0h (240_{10}) a produs un împrumut;

b) ZF este 0, deoarece rezultatul este nenul;

c) SF este 1, deoarece semnul rezultatului pe 8 biți (F0h) este 1 (negativ);

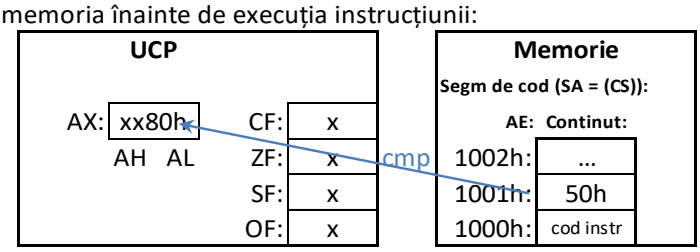
d) OF este 0, deoarece rezultatul cu semn al operației: 40h (64_{10}) - 50h (80_{10}) = F0h (-16_{10}), este în gama de valori a numerelor reprezentabile pe 8 biți: -128_{10} ... $+127_{10}$.

3. UCP și memoria după execuția instrucțiunii:



CMP AL, 50h

1. UCP și memoria înainte de execuția instrucțiunii:



2. Valoarea 50h este scăzută din valoarea 80h. Rezultatul pe 8 biți (30h) nu este stocat nicăieri dar fanioanele sunt modificate după cum urmează:

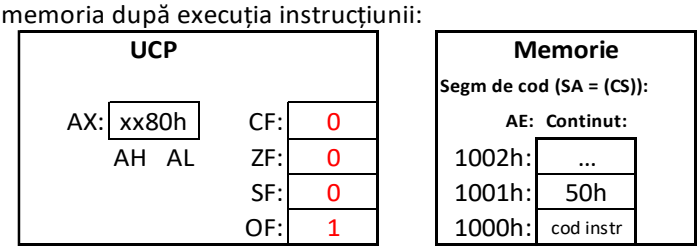
a) CF este 0, deoarece rezultatul fără semn al operației: 80h (128_{10}) - 50h (80_{10}) = 30h (48_{10}) est în gama de valori a numerelor fără reprezentabile pe 8 biți: 0_{10} ... $+255_{10}$.

b) ZF este 0, deoarece rezultatul este nenul;

c) SF este 0, deoarece semnul rezultatului pe 8 biți (30h) este 0 (pozitiv);

d) OF este 1, deoarece operația: 80h (-128_{10}) - 50h (80_{10}) = 30h (-208_{10}), a produs o depășire.

3. UCP și memoria după execuția instrucțiunii:



Exemplu

SBB – Scădere între întregi cu împrumut

Mod de folosire: SBB d, s

Argumente:

- d – registru pe 8 sau 16 biţi sau locaţie de memorie
- s – constantă adresată imediat sau registru pe 8 sau 16 biţi sau locaţie de memorie;

Efecte: Scade fanionul de transport şi s din d: **(d) ← (d) – (s) – (CF)**.

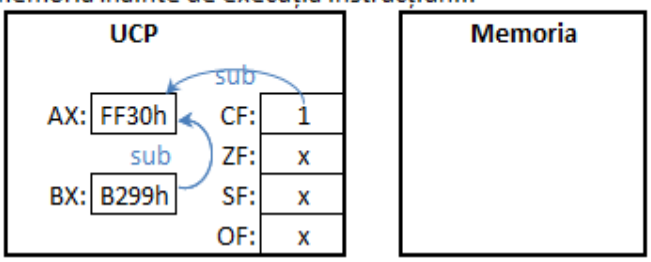
Fanioane: CF, ZF, OF, SF, AF, şi PF sunt modificate corespunzător cu rezultatul.

Notă: doar un singur argument de memorie este permis, iar ambele argumente trebuie sa aibă aceeaşi dimensiune.

Exemplu

SBB AX, BX

1.UCP și memoria înainte de execuția instrucțiunii:



2. Fanionul CF și valoarea B299h, care este stocată în registrul BX, sunt scăzute din valoarea stocată în registrul AX. Fanioanele se modifică după cum urmează:

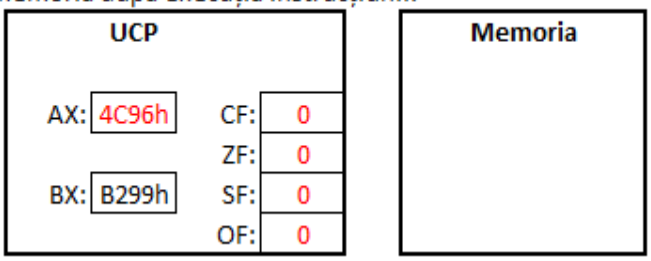
a) CF este 0, deoarece rezultatul operației fără semn FF30h(6532_{10}) - B299h(45721_{10}) -1 = 4C96h(19606_{10}) este în gama numerelor fără semn, reprezentabile pe 16 biți: 0...+65536_{10}

b)ZF este 0, deoarece rezultatul este nenul

c)SF este 0, deoarece semnul rezultatului pe 16 biți (4C96h) este 0 (pozitiv)

d)OF este1, deoarece rezultatul cu semn al operației FF30h(-208_{10}) - B299h(-19815_{10}) -1 = 4C96h(19606_{10}) este în gama de valori a numerelor reprezentabile pe 16 biți: -32768_{10}...+32767_{10},

3.UCP și memoria după execuția instrucțiunii:



MUL – Înmulţire fără semn a lui AL sau AX

Mod de folosire: MUL s

Argumente:

- s – registru pe 8 sau 16 biţi sau locaţie de memorie.

Efecte:

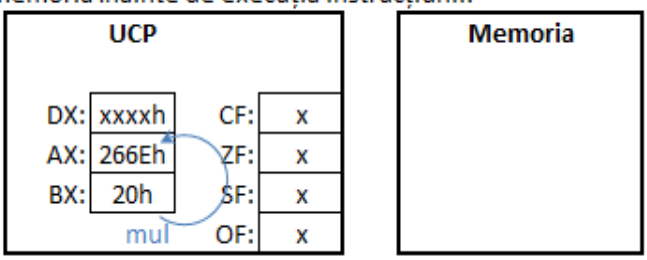
- dacă s este o valoare pe 8 biţi: se înmulţeşte valoarea stocată în **AL** cu **s** şi se stochează rezultatul în AX: **(AX) ← (AL) * (s)**. CF şi OF sunt resetate la 0 dacă **AH** e 0, altfel sunt setate la 1.
- dacă s este o valoare pe 16 biţi: se înmulţeşte valoarea stocată în **AX** cu **s** şi se stochează rezultatul în **DX** concatenat cu **AX**: **(DX) ↕ (AX) ← (AX) * (s)**. CF şi OF sunt resetate la 0 dacă **DX** e 0, altfel sunt setate la 1.

Fanioane: CF şi OF sunt modificate aşa cum s-a menţionat mai sus. Restul fanioanelor sunt nedefinite.

Exemplu

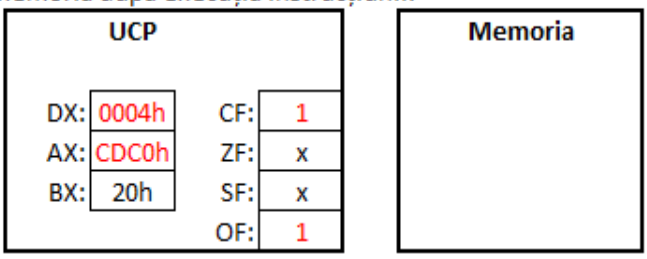
MUL BX

1.UCP și memoria înainte de execuția instrucțiunii:



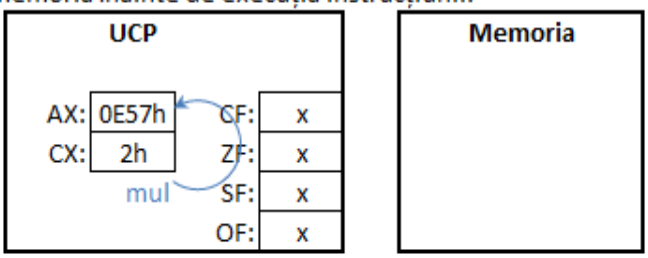
2. Valoarea 266Eh, care este stocată în AX, este înmulțită cu valoarea stocată în registrul BX. Rezultatul este stocat în DX concatenat cu AX. Fanioanele CF și OF sunt 1, deoarece valoarea din DX nu este 0, iar SF, ZF și PF sunt nedefinite.

3.UCP și memoria după execuția instrucțiunii:



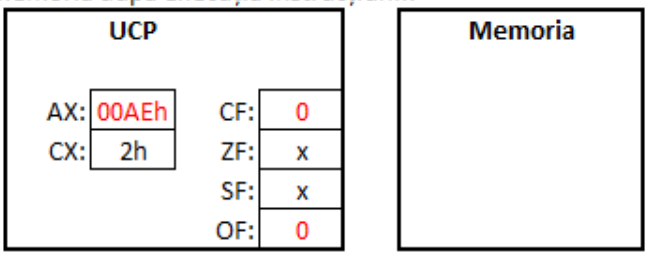
MUL CL

1.UCP și memoria înainte de execuția instrucțiunii:



2. Valoarea 57h, care este stocată în AL, este înmulțită cu valoarea stocată în registrul CL. Rezultatul este stocat în AX. Fanioanele CF și OF sunt 0, deoarece valoarea din AH este 0, iar SF, ZF și PF sunt nedefinite.

3.UCP și memoria după execuția instrucțiunii:



Exemplu

3. ACCESAREA MEMORIEI. INSTRUCȚIUNI DE TRANSFER ȘI PRELUCRARE DE DATE UTILIZÂND MEMORIA

3.1 Organizarea memoriei

3.1.1 Modelul de memorie x86 în modul real (segmentat)

Procesorul x86 in modul real are o magistrală de adrese de 20 de biţi, ceea ce înseamnă că poate accesa memorii cu adrese fizice de maxim 20 de biţi. x86 în modul real organizează memoria în segmente mai mici folosind o adresă pe 16 biţi, numită adresă segment (**AS**), pentru a identifica un segment în memorie, şi o altă adresă pe 16 biţi, numită adresă efectivă (**AE**) sau offset, pentru a identifica locaţia de memorie în interiorul segmentului. Ca o consecinţă a faptului că aceste adrese sunt numere pe 16 biţi şi fiecare locaţie de memorie stochează un număr pe 8 biţi, memoria este organizată în segmente ce cuprind fiecare 2^{16}=64kB.

Adresa fizică (**AF**), ce este folosită de microprocesor pentru a accesa fizic memoria, este obţinută prin sumarea adresei segment concatenată cu 0h (deplasată la stânga cu 4 biţi) şi a adresei efective: **AF = AS ↕0h + AE**.

De exemplu, presupunând că dorim să accesăm locaţia de memorie cu adresa efectivă 0001h din segmentul cu adresa segment 0001h, adresa fizică va fi 0001 ↕0h + 0001h = 00011h.

Modul de organizare a memoriei folosit de microprocesoarele x86 care lucrează în modul real este numit *organizarea segmentata a memoriei*.

3.1.2 Organizarea informaţiei în memorie

Există 3 tipuri de informaţii stocate de obicei în memorie:

- programul curent (instrucţiunile),
- datele pentru programul curent (operanzi, rezultate, variabile, şiruri de date) şi
- stiva programului curent.

Arhitectura x86 organizează aceste informaţii în 4 segmente separate: segmentul de cod, segmentul de date, segmentul extins de date şi segmentul de stivă.

Segmentul de cod stochează programul curent (instrucţiunile). Adresa segment a segmentului de cod curent este stocată în **CS** (registrul segment de cod). În cadrul segmentului de cod, informaţia poate fi accesată utilizând adresa efectivă stocată într-un registru cu funcţie specială, numit **IP** (registrul indicator de instrucţiuni). **IP** stochează *întotdeauna* adresa efectivă a instrucţiunii curente.

Segmentul de date stochează operanzii, rezultatele, variabilele, şirurile de date pentru a fi utilizate în program. Adresa segment a segmentului curent de date este stocată în **DS** (registrul segment de date). În cadrul segmentului de date, informaţia poate fi accesată folosind adresele efective stocate în **BX** ("base index register") şi **SI** ("source index register"). **BX** stochează *de obicei* adresa efectivă a variabilelor elementare sau adresa efectivă a primului element dintr-un şir de date. SI stochează *de obicei* adresa efectivă a elementului curent într-un şir special de date, numit şir sursă.

Segmentul suplimentar de date este utilizat similar cu segmentul de date pentru a stoca operanzi, rezultate, variabile, şiruri de date. Adresa segment a segmentului suplimentar de date curent este stocată în **ES** (registrul segment suplimentar de date). În cadrul segmentului suplimentar de date informaţia poate fi accesată folosind adresa efectivă stocată în **DI** ("destination index register"). **DI** stochează *de obicei* adresa efectivă a elementului curent într-un şir special de date, numit şir destinaţie.

Segmentul de stivă stochează stiva utilizată de program. Adresa segment a segmentului curent de stivă este stocată în **SS** (registrul segment de stivă). În cadrul segmentului de stivă, informaţia poate fi accesată folosind adresa efectivă stocată în **SP** (registrul indicator în stivă) şi **BP** (registrul indicator al bazei în stivă). **SP** stochează *întotdeauna* adresa efectivă a elementului situat în vârful stivei primare. **BP** stochează *de obicei* adresa efectivă a altui element din stiva primară sau adresa efectivă a elementului situat în vârful stivei alternative.

Deşi registrele de adrese (**BX**, **SI**, **DI**, **SP**, **BP** şi **IP**) sunt asociate în mod implicit cu registre de segment specifice (**IP** cu **CS**, **BX** şi **SI** cu **DS**, **DI** cu **ES**, **SP** şi **BP** cu **SS**) pentru a forma adrese complete pentru informaţii specifice în memorie, arhitectura x86 permite folosirea unor registre de adrese şi cu alte registre decât cele din oficiu (aceasta se numeşte redirecţionarea segmentelor). **BX**, **SI** şi **DI** pot, de asemenea, fi utilizate pentru a accesa date din segmentul de cod, segmentul de date, segmentul extins de date si segmentul de stivă. BP poate fi utilizat şi pentru a accesa date din segmentul de cod, segmentul de date şi segmentul extins de date.

CS, **DS**, **ES** şi **SS** pot stoca aceeaşi adresă segment şi nu există nicio restricţie în ceea ce priveşte conţinutul unui astfel de registru în raport cu celalalt. De aici rezultă ca segmentele astfel formate în memorie pot să se suprapună total, parţial sau deloc. În cazul în care avem aceeaşi adresă segment în toate registrele segment, rezultă că toată informaţia necesară programului curent se găseşte într-un singur segment de memorie. Acesta este în general scenariul pe care îl vom folosi mai departe în laborator.

Sumar pentru organizarea memoriei:

- memoria poate fi privită ca o secvenţă de locaţii de memorie;

- fiecare locaţie de memorie stochează o informaţie pe 8 biţi şi are o adresă unică pe 20 de biţi, numită adresă fizică;
- procesoarele de tipul x86 în modul real organizează memoria sub formă de segmente de 64kB;
- x86 foloseşte o adresă segment (**AS**) pe 16 biţi pentru a selecta segmentul şi o adresă efectivă (**AE**) pe 16 biţi pentru a identifica locaţia de memorie în interiorul segmentului;
- adresele segment (**AS**) pot fi stocate într-unul dintre următoarele registre de segment: **CS**, **DS**, **ES**, **SS;**
- adresele efective (**AE**) pot fi stocate într-unul dintre următoarele registre: **BX**, **SI**, **DI**, **SP**, **BP** şi **IP;**
- translatarea între organizarea logică a memoriei în segmente şi adresa fizică este realizată astfel: : **AF = AS ↕0h + AE**;
- Segmentele organizate în memorie se pot suprapune total, parţial sau deloc.

3.2 Moduri de adresare x86 în modul real pentru microprocesoare pe 16 biţi

Modurile de adresare sunt tehnicile utilizate pentru a specifica (în formatul instrucţiunii) locaţia operanzilor şi/sau a rezultatelor, precum şi adresa unei noi instrucţiuni în cazul salturilor. Modurile de adresare ale x86 în modul real sunt sintetizate în Tabelul 9. Se observă că toate modurile de adresare prezentate în Tabelul 9 se referă la memorie. Modul de adresare care specifică faptul ca datele se găsesc într-un registru se numeşte **adresare în registru**.

3.3 Alte directive de asamblare

Directivele de asamblare au fost introduse în secțiunea 1; aici găsiți mai multe dintre directivele pe care le vom utiliza în cadrul acestor platforme:

- **db** (define byte)
    - Utilizare: [simbol] db [date, [date]]
    - Exemple:
        - array db 10h, 23h, 29h
        - charX db ‘a’
        - inputString db ‘ana are mere’
    - Efect: Alocă unul sau mai mulţi octeţi şi îi iniţializează cu valori din expresie (dacă există). Simbolul va fi asociat cu adresa primului octet alocat.

Tabelul 9. Moduri de adresare x86 în modul real

	Mod de adresare	Exemplu	Descriere
adresare simplă	imediată	mov AX, **1234h**	datele se găsesc în memorie (în segmentul de cod) imediat după codul instrucţiunii
	directă	mov AX, **[1234h]**	datele se găsesc în memorie la adresa efectivă specificată în instrucţiunea curentă
	indexată	mov AX, **[SI+4h]**	datele se găsesc în memorie la adresa specificată de conţinutul lui SI sau DI + un offset ce se găseşte în instrucţiunea curentă
	indirectă	mov AX, **[SI]**	datele se găsesc în memorie la adresa specificată de conţinutul lui BX, SI sau DI
adresare relativă la bază	directă	mov AX, **[BX+13h]**	datele se găsesc în memorie la adresa efectivă obţinută ca sumă între conţinutul lui BX şi offsetul specificat în instrucţiunea curentă
	indexată	mov AX, **[BX+DI+13h]**	datele se găsesc în memorie la adresa efectivă obţinută ca sumă între conţinutul lui BX, conţinutul lui SI sau DI şi offsetul specificat în instrucţiunea curentă
	implicită	mov AX, **[BX+SI]**	datele se găsesc în memorie la adresa efectivă obţinută ca sumă între conţinutul lui BX şi conţinutul lui SI sau DI
adresare relativă în stivă	directă	mov AX, **[BP+13h]**	datele se găsesc în memorie la adresa efectivă obţinută ca sumă între conţinutul lui BP şi offsetul specificat în instrucţiunea curentă
	indexată	mov AX, **[BP+SI+13h]**	datele se găsesc în memorie la adresa efectivă obţinută ca sumă între conţinutul lui BP, conţinutul lui SI sau DI şi offsetul specificat în instrucţiunea curentă
	implicită	mov AX, **[BP+DI]**	datele se găsesc în memorie la adresa efectivă obţinută ca sumă între conţinutul lui BP şi conţinutul lui SI sau DI

- **DW** (define word)
  - Utilizare: [simbol] dw [date, [date]]
  - Exemple: numbers dw 8933h, 1240h, 0328h, 99A0h, 0F422h, 0101h
  - Efect: Alocă unul sau mai multe cuvinte şi le iniţializează cu valori din expresie (dacă există). Simbolul va fi asociat cu adresa primului cuvânt alocat.

- **equ** (create symbol)
  - Utilizare: simbol equ expresie
  - Exemplu: number equ 4000h
  - Efect: Simbolul va avea, pe parcursul întregului program, valoarea obţinută după evaluarea expresiei.

- **offset** (offsetul unei expresii)
    - Utilizare: offset simbol
    - Exemplu: offset array
    - Efect: Returnează adresa asociată simbolului.

- **byte ptr** (pointer de octet)
    - Utilizare: byte ptr simbol
    - Exemplu: byte ptr [1234h]
    - Efect: converteşte simbolul în octet.

- **word ptr** (pointer de cuvânt)
    - Utilizare: word ptr simbol
    - Exemplu: word ptr [1234h]
    - Efect: Converteşte simbolul în cuvânt.

Se pot declara şi zone statice de date (analog variabilelor globale) în arhitectura x86 folosind directivele de asamblare **db** şi **dw**. Locaţiile pot fi etichetate cu nume pentru o referinţă ulterioară – acest lucru este similar declarării variabilelor după nume. Variabilele declarate într-un şir se vor găsi în memorie una lângă cealaltă.

Exemple de declarare:

```
var     db 64  ; Declară un byte, la care se face referinţă cu locaţia  var, care conţin valoarea 64
var2    db ?   ; Declară un byte neiniţializat, la care se face referinţă cu locaţia var2
        db 10  ;  Declară un byte fara etichetă, conţinând valoarea 10.  Adresa  lui este var2 + 1.
var3    dw ?   ; Declară o valoare neiniţializată de 2 octeţi, la care se face referinţă cu locaţia var3.
```

Spre deosebire de limbajele de programare de nivel înalt unde matricile pot avea mai multe dimensiuni şi pot fi accesate cu ajutorul indicilor, matricile în limbajul de asamblare x86 sunt structuri unidimensionale de tip şir. Un şir poate fi declarat prin simpla enumerare a valorilor pe care le conţine ca în primul exemplu de mai jos. Alte două metode pentru declararea unui şir de valori sunt directiva dup şi folosirea literelor în string. Directiva dup îi spune asamblorului să multiplice o expresie de un anumit număr de ori. De exemplu, 4 dup(2) este echivalent cu 2, 2, 2, 2.

Exemple de declarare:

array1 dw 1, 2, 3 ;Declară un şir cu 3 valori, iniţializate cu 1, 2 şi 3
bytes db 10 dup(?) ;Declară 10 octeţi neiniţializaţi începând de la locaţia bytes
array dw 100 dup(0);Declară 100 cuvinte (2 octeţi) începând de la locaţia array, toate iniţializate cu 0
string db 'hello', 0 ;Declară 6 octeţi începând de la adresa string, iniţializaţi cu valorile ASCII corespunzătoare caracterelor cuvântului hello şi octetul nul (0).

3.4 Instrucţiuni de transfer de date cu memoria pentru procesoarele x86

Instrucţiunile de transfer de date sunt acele instrucţiuni CPU utilizate pentru a:

- copia o constantă într-un registru sau într-o locaţie de memorie;
- copia date dintr-o locaţie de memorie într-un registru sau vice versa;
- copia date de la/către dispozitivele I/O.

Fiecare microprocesor are setul său specific de instrucţiuni, inclusiv instrucţiuni specifice de transfer de date. Cele mai importante instrucţiuni de transfer de date oferite de setul de instrucţiuni al arhitecturii x86 sunt listate în Tabelul 10.

Prima particularitate a stivei x86 este că toate elementele stocate în stivă au dimensiunea de 16 biţi. În consecinţă, instrucţiunile care inserează sau scot implicit valori din stivă, inclusiv **push** şi **pop**, incrementează sau decrementează indicatorul stivei (**SP**) cu 2.

A doua particularitate a stivei este faptul că ea creşte în jos (înspre adrese mai mici). În consecinţă, instrucţiunile care inserează valori în stivă, cum ar fi **push**, decrementează indicatorul stivei (**SP**), în timp ce instrucţiunile care scot valori din stivă, cum ar fi pop, incrementează indicatorul stivei.

Detalii şi exemple de utilizare privind instrucţiunile de transfer de date sunt oferite în Secţiunea 3.6.

Tabelul 10. Instrucţiuni de transfer de date în x86

Instrucţiune	Utilizare	Descriere
MOV – Move Data	MOV d, s	Copiază s la d
XCHG – Exchange Data	XCHG d, s	Schimbă s cu d
LEA – Load Effective Address	LEA d, s	Încarcă adresa efectivă a s în d
PUSH – Push in the Stack	PUSH s	Scrie s în stivă
POP – Pop out of the Stack	POP d	Scoate un cuvânt din stivă şi îl stochează în d

3.4.1 Instrucţiuni speciale de transfer de date: instrucţiunile pentru şiruri ("string operations")

Arhitectura x86 defineşte două zone implicite de memorie care stochează două şiruri de numere pe 8 sau 16 biţi, numite şirul sursă şi şirul destinaţie. Șirul sursă este stocat implicit în segmentul de date (segmentul cu adresa în **DS**). Elementul curent din şirul sursă este implicit la adresa efectivă specificată în **SI**. Șirul destinaţie este stocat implicit în segmentul extins de date (segmentul cu adresa în **ES**). Elementul curent din şirul destinaţie este implicit la adresa efectivă specificată în **DI**.

Arhitectura setului de instrucţiuni x86 include câteva instrucţiuni specifice şirurilor (vezi Tabelul 3) pentru a încărca, stoca, muta, scana şi compara elementele acestor două şiruri implicite de date. Toate aceste instrucţiuni au o versiune pe 8 biţi (lodsb, stosb, movsb, scasb and cmpsb) şi o versiune pe 16 biţi (**lodsw, stosw, movsw, scasw, cmpsw**). Pentru operaţiile de încărcare, stocare şi scanare, registrul implicit pentru ţintă/sursă/comparaţie este **AL** (pentru versiunea pe 8 biţi) şi **AX** (pentru versiunea pe 16 biţi).

Toate instrucţiunile specifice şirurilor folosesc implicit fanionul de direcţie (**DF**), care controlează parcurgerea în sens crescător sau descrescător al adreselor elementelor dintr-un şir. Aşadar, toate instrucţiunile specifice şirurilor realizează 2 operaţii: a) procesarea elementului curent din şir (pe un octet sau pe 2 octeti) şi b) trecerea la următorul element din şir.

Tabelul 11. x86 instrucţiuni specifice şirurilor

Instrucţiune	**Utilizare**	**Descriere**
MOVS – Move String	MOVSB / MOVSW	Copiază elementul curent din şirul sursă în şirul destinaţie
LODS – Load String	LODSB / LODSW	Încarcă elementul curent din şirul sursă în acumulator
STOS – Store String	STOSB / STOSW	Stochează valoarea din acumulator în elementul curent din şirul destinaţie
SCAS – Scan String	SCASB / SCASW	Compară valoarea din acumulator cu elementul curent din şirul destinaţie
CMPS – Compare String	CMPSB / CMPSW	Compară elementul curent din şirul sursă cu elementul curent din şirul destinaţie
STD – Set DF	STD	Setează fanionul de direcţie: (DF) <- 1
CLD – Clear DF	CLD	Resetează fanionul de direcţie: (DF) <- 0

Instrucţiunile specifice şirurilor pot fi precedate de unul dintre următoarele prefixe de repetabilitate: **rep**, **repe/repz**, **repne/repnz**. Aceste prefixe spun procesorului să repete instrucţiunea referitoare la şir de un număr de ori specificat de numărătorul implicit (**CX**) şi să decrementeze numărătorul la fiecare repetare. În cazul prefixelor **repe/repz, repne/repnz**, care sunt utilizate de obicei cu **scas** sau **cmps**, fanionul de zero (rezultat din comparaţie) este de

asemenea verificat şi repetiţia este continuată doar în cazul în care condiţia este îndeplinită (**ZF**=1 pentru **repe/repz** şi **ZF**=0 pentru **repne/repnz**).

Detalii şi exemple de utilizare privitoare la instrucţiunile specifice şirurilor sunt oferite în Secţiunea 3.7.

3.5 Exerciţii

3.5.1 Exerciţiul 1

Obiectiv. Înţelegerea efectului executării instrucţiunilor LEA, MOV, CMP, JE, INC, JMP şi LOOP şi directivele de asamblare org şi db.

Cerinţă. Să se scrie un program care înlocuieşte toate apariţiile unui caracter (denumit charX) dintr-un şir de caractere (numit inputString) cu un alt caracter (denumit charY). Să se reţină numărul înlocuirilor realizate folosind registrul DX.

Soluţie.

1. Se porneşte emulatorul.

2. Se scrie următorul program, folosind fereastra Source Code:

```
                org 100h

init:           lea BX, inputString
                mov SI, 0h
                mov CX, charX-inputString
                mov DX, 0h
                mov AH, [charY]

compare:        mov AL, [BX+SI]
                cmp AL, [charX]
                je replace
nextElem:       inc SI
                loop compare

                int 20h

replace:        mov [BX+SI], AH
                inc DX
                jmp nextElem

inputString     db      'alpha beta'
charX           db      'a'
charY           db      'A'
```

3. Explicaţia programului

3.1. Prima linie a acestui program (org 100h) nu este o instrucţiune. Este o directivă de asamblare care specifică faptul ca următoarea instrucţiune (şi în consecinţă întregul program) va fi încărcată în memorie (în segmentul de cod) începând cu adresa 100h.

3.2. Ultimele trei linii de cod ale programului sunt folosite pentru a defini şi iniţializa trei variabile: şirul de caractere inputString, caracterul înlocuit charX şi caracterul folosit pentru înlocuire charY. Elementele şirului sunt octeţi (numere pe 8 biţi) şi cele două variabile charX şi charY sunt octeţi, de asemenea. Din această cauză ele sunt definite folosind directiva de asamblare db (define byte). Variabilele sunt plasate în memorie la nişte adrese efective ce vor fi dezbătute ulterior.

3.3. Observaţi că elementele din inputString şi valorile celorlalte două variabile sunt caractere numai din punctul de vedere al programatorului (deoarece el doreşte să le interpreteze drept caractere). Din punctul de vedere al procesorului memoria conţine doar numere (codurile ASCII ale acestor caractere, vezi Anexa 3. Tabelul ASCII).

3.4. Blocul de instrucţiuni init are rolul de a iniţializa registrele. Instrucţiunea lea BX, inputString încarcă adresa efectivă a lui inputString în BX. Instrucţiunea mov SI, 0h iniţializează registrul SI (ce va fi folosit pentru a itera în interiorul şirului) cu valoarea 0h. Instrucţiunea mov CX, charX-inputString calculează diferenţa dintre adresa lui charX şi adresa primului element al şirului inputString (această diferenţă reprezintă numarul de elemente din şir) şi stochează rezultatul în registrul contor CX. Instrucţiunea mov DX, 0h iniţializează registrul DX (care va fi folosit pentru a număra înlocuirile efectuate) cu valoarea 0h. Ultima instrucţiune a acestui bloc, mov AX, charY, copiază codul ASCII al caracterului charY din memorie în registrul AH.

3.5. Blocul de instrucţiuni compare este folosit pentru a parcurge şirul de caractere şi a compara fiecare element cu caracterul charX (făcând înlocuirile când este nevoie). Instrucţiunea mov AL, [BX+SI] copiază valoarea stocată în memorie, la adresa efectivă BX+SI în registrul AL (date fiind iniţializările de mai sus această valoare va fi primul caracter in inputString). Instrucţiunea cmp AL, [charX] compară (prin scădere) valoarea din registrul AL cu valoarea stocată în memorie la adresa charX. Această instrucţiune nu modifică valoarea stocată în AL. Scopul ei este de a modifica valorile fanioanelor (OF, SF, ZF, AF, PF, CF). Următoarea instrucţiune (je replace) face un salt la eticheta replace dacă cele două numere comparate de instrucţiunea precedentă sunt egale. Instrucţiunea je (jump if equal) are acces la rezultatul instrucţiunii precedente prin intermediul fanionului zero flag (ZF). Daca ZF este 1 înseamnă că scăderea s-a terminat cu rezultatul nul şi se face saltul. Daca ZF este 0 înseamnă că saltul nu se face si procesorul continuă cu următoarea instrucţiune (inc SI). Instrucţiunea inc SI incrementează valoarea stocată în registrul SI. Aşadar, la următoarea iteraţie se va procesa următorul element al şirului. Ultima instrucţiune a acestui bloc este loop compare. Această instrucţiune decrementează in mod implicit contorul (CX) şi, daca rezultatul este o valoare diferită de 0, face salt înapoi la eticheta compare (pentru a procesa următorul element din inputString). Daca valoarea lui CX este 0 înseamnă că toate elementele şirului au fost procesate şi nu se mai execută saltul la eticheta compare, ci se trece mai departe, la următoarea instrucţiune (int 20h). Această ultimă instrucţiune termină programul curent.

3.6. Blocul de instrucţiuni replace se execută numai când elementul curent din şirul inputString este egal cu charX. Prima instrucţiune din acest bloc (mov [BX+SI], AH) suprascrie valoarea stocată în memorie la adresa BX+SI (elementul curent) cu valoarea stocată în registrul AH (caracterul charY). A doua instrucţiune din acest bloc (inc DX)

incrementează numărul de înlocuiri efectuate, iar ultima instrucţiune face un salt necondiţionat înapoi la eticheta nextElem din bucla compare.

4. Se salvează programul (File menu -> Save As submenu) cu numele lab2_prog1.asm.

5. Se compilează programul:

5.1. Se face click pe butonul Compile pentru a compila programul.
5.2. Veţi fi îndrumaţi să salvaţi fişierul executabil. Se salvează cu numele recomandat.
5.3. Se vizualizează statusul compilării în caseta Assembler Status dialog. Dacă programul a fost editat corect, mesajul ar trebui sa fie "<numele programului> is assembled successfully into … bytes."
5.4. Apăsaţi butonul View şi apoi Symbol Table pentru a vizualiza tabela de simboluri asociate acestui program. Informaţia prezentată în listă ar trebui interpretată în felul următor:

- simbolurile charX, charY şi inputString sunt variabile de un octet stocate în memorie la adresele 012Ch, 012Dh, 0122h. Observaţi că, deşi inputString defineşte un şir de numere simbolul inputString reprezintă doar adresa de început a acestui şir. Aceste simboluri pot fi asociate cu pointerii din C.

- simbolurile compare, init, nextElem şi replace sunt etichete ale unor instrucţiuni în program şi sunt asociate cu adresele acestor instrucţiuni (0110h, 0100h, 0118h, 011Dh).

5.5. Lista de simboluri ne va ajuta să găsim datele cu care lucrăm (în memorie).

6. Se încarcă programul executabil în emulator.

6.1. Se apasă butonul Run pentru a încărca programul în emulator şi a-l executa.

7. Se execută programul pas cu pas, se observă modificările apărute în registre, locaţiile de memorie, fanioane etc. şi se notează observaţii.

7.1. Se apasă butonul Reload pentru a reîncărca programul executat.
7.2. Se execută toate instrucţiunile pas cu pas, făcând click succesiv pe butonul Single Step.
7.2.1. Instrucţiunea curentă (mov BX, 0122h) este evidenţiată. Aceasta este prima instrucţiune din program şi a fost încărcată la adresa logică 0700:0100 (adresa segment : adresa efectivă). Adresa efectivă a fost impusă de directiva de asamblare org 100h. Această instrucţiune este echivalentă cu instrucţiunea scrisă în codul sursă (lea BX, inputString), deoarece simbolul inputString a fost înlocuit cu adresa care îi este asociată.
7.2.2. Valoarea din registrul IP (registrul care stochează adresa efectivă a instrucţiunii curente) este 0100h.
7.3. Se face click pe meniul View -> Memory pentru a vizualiza starea curentă a memoriei. În caseta Address se va scrie adresa variabilei inputString: se lasă neschimbată adresa segment (0700) şi se va înlocui adresa efectivă (0100) cu adresa efectivă a lui inputString (0122). Se face click pe butonul Update şi se observă următoarele:

7.3.1. Adresa de început este acum 0700:0122 şi valorile stocate în memorie sunt 61h, 6Eh, 61h, … În partea dreaptă a ferestrei Memory aceste numere sunt interpretate drept coduri ASCII şi sunt reprezentate cu caractere (“alpha beta”). Observaţi că aceste caractere sunt cele din inputString. În continuarea lor se află caracterele pentru charX (“a”) şi charY (“A”).

7.4. Se face click pe butonul Single Step pentru a executa prima instrucţiune (mov BX, 0122h) şi observaţi că registrul BX a fost încărcat cu adresa de început a şirului inputString (0122h).

7.5. Se execută următoarele trei instrucţiuni şi se observă iniţializarea registrelor SI,CX şi DX cu valorea 0000h.

7.6. Observaţi în fereastra Memory că locaţia de memorie cu adresa 012Dh (adresa lui charY) stochează valoarea 41h (codul ASCII al lui “A”). Executaţi instrucţiunea mov AH, [012D] şi observaţi că în registrul AH (partea superioară a lui AX) a fost încărcată valoarea stocată în memorie la adresa 012Dh (41h).

7.7. Calculaţi BX+SI (la prima iteraţie rezultatul este 0122h + 0h = 0122h). Observaţi în fereastra Memory că locaţia de memorie cu adresa 0122h (adresa lui inputString) stochează valoarea 61h (codul ASCII al lui “a”). Executaţi instrucţiunea mov AL, [BX+SI] şi observaţi că în registrul AL (partea inferioară a lui AX) a fost încărcată valoarea stocată în memorie la adresa 0122h (61h).

7.8. Observaţi că locaţia de memorie cu adresa 012Ch (adresa lui charX) stochează valoarea 61h (codul ASCII al lui “a”). Executaţi instrucţiunea cmp AL, [012C], faceţi click pe butonul Flags pentru a vedea starea registrelor şi observaţi că fanionul zero flag (ZF) a fost setat (la valoarea 1).

7.9. Următoarea instrucţiune este jz 011Dh. Această instrucţiune este echivalentă cu instrucţiunea scrisă în program (je replace), deoarece simbolul replace a fost înlocuit cu adresa care îi era asociată. Zero flag este acum 1, ceea ce înseamnă că execuţia instrucţiunii ar trebui să rezulte într-un salt la eticheta replace. Executaţi instrucţiunea curentă şi observaţi că saltul a fost făcut către prima instrucţiune din blocul de instrucţiuni replace. De asemenea, valoarea din registrul IP este 011Dh.

7.10. Primul Element din inputString era egal cu “a” (aceeaşi variabilă ca şi charX). Prin urmare, s-a efectuat un salt la blocul de instrucţiuni replace. Calculaţi BX+SI (la prima iteraţie rezultatul este 0122h + 0h = 0122h). Executaţi instrucţiunea curentă (mov [BX+SI], AH) şi observaţi că valoarea din registrul AH (41h) a fost copiată în locaţia de memorie cu adresa 122h. Observaţi şi modificarea în reprezentarea caracterelor din fereastra Memory (inputString este acum “Alpha beta”).

7.11. Executaţi următoarea instrucţiune (inc DX) şi observaţi ca valoarea lui DX a fost incrementată.

7.12. Executaţi următoarea instrucţiune (jmp 0118h) şi observaţi că s-a efectuat un salt la instrucţiunea cu adresa efectivă 0118h (următoarea instrucţiune ce va fi executată este cea cu adresa efectivă 0118h). Amintiţi-vă că această adresă este, de fapt, asociată cu eticheta nextElem (instrucţiunea pe care aţi scris-o în codul sursă este jmp nextElem).

8. Se scriu concluzii referitoare la efectul diferitelor instrucţiuni asupra registrelor, fanioanelor şi locaţiilor de memorie.

3.5.2 Exerciţiul 2

Obiectiv. Înţelegerea efectului executării instrucţiunilor de string (cld, std, movs, lods şi stos), câteva instrucţiuni de salt condiţionat (ja, jne) şi prefixul de repetabilitate rep.

Cerinţă. Să se scrie un program care înlocuieşte toate apariţiile unui caracter (denumit charX) dintr-un şir de caractere (numit inputString) cu un alt caracter (denumit charY), prin copierea şirului de caractere într-o zonă auxiliară de memorie (unde şirul va fi numit outputString).

Soluţie.

1. Se porneşte emulatorul.

2. Se încarcă programul numit lab3_prog2.asm, folosind butonul Open. Fereastra Source Code trebuie să afişeze următorul program:

```
                org 100h

init:           lea SI, inputString
                lea DI, outputString
                mov CX, charX-inputString
                cld

compare:        lodsb
                cmp AL, [charX]
                jne copyCurrent
                je copyCharY

nextElem:       loop compare

copyBack:       dec SI
                dec DI
                xchg SI, DI
                mov CX, charx-inputString
                std
                repnz movsb

end:            int 20h

copyCurrent:  stosb
                jmp nextElem

copyCharY:    mov AL, [charY]
                stosb
                jmp nextElem
```

```
inputString     db 'alpha beta'
charX           db 'a'
charY           db 'A'
outputString    db 20 dup('X')
```

3. Explicaţia programului

3.1. Ultimele 4 linii ale acestui program sunt folosite pentru a defini şi iniţializa variabilele. În plus faţă de exerciţiul anterior, în acest program linia outputString db 20 dup('X') alocă un şir de 20 de locaţii de memorie şi iniţializează toate valorile şirului cu caracterul „X". Acesta va fi şirul de ieşire.

3.2. Spre deosebire de exerciţiul anterior, inputString va fi acum iterat folosind un singur pointer (SI). OutputString va fi iterat folosind DI. În consecinţă, în blocul de instrucţiuni etichetat init aceste două registre sunt iniţializate cu adresa primului element din cele 2 şiruri. Fanionul de direcţie este iniţializat cu 0 (cld), însemnând că şirurile vor fi iterate de la stânga la dreapta. Numărătorul CX este iniţializat exact ca în exerciţiul anterior (cu numărul de elemente din inputString).

3.3. Blocul de instrucţiuni etichetat compare iterează prin şirul inputString şi compară elementul curent cu charX. Dacă elementul curent este charX, atunci programul copiază charY în outputString, altfel copiază elementul curentdin inputString în outputString. Instrucţiunea lodsb copiază elementul curent din şirul sursă în acumulator şi incrementează SI. Instrucţiunea cmp AL, [charX] compară elementul curent cu charX. Dacă cele 2 valori nu sunt egale, programul execută un salt la copyCurrent (jne copyCurrent), altfel sare la copyCharY (je copyCharY). În final, instrucţiunea loop decrementează numărătorul CX şi sare înapoi la compare dacă CX nu este zero. Instrucţiunile din copyCurrent: a)stochează valoarea din acumulator în outputString (decrementând de asemenea şi DI) şi b) execută saltul înapoi în bucla compare.

3.4. După ce bucla compare este terminată, valorile din outputString trebuie copiate înapoi în inputString. Această operaţie este executată de blocul de instrucţiuni numit copyBack. SI şi DI sunt decrementate, astfel încât ele stochează adresele efective ale ultimelor elemente din inputString şi respectiv outputString. Apoi valorile lor sunt interschimbate (xchg SI, DI), deoarece pentru această operaţie inputString este şirul destinaţie, iar outputString este şirul sursă. CX este din nou iniţializat cu numărul de elemente din inputString (numărul de elemente care trebuie copiate înapoi). Fanionul de direcţie este setat la 1 (std), deoarece şirurile vor fi iterate de la stânga la dreapta. Prefixul de repetabilitate rep plasat în faţa lui movsb (rep movsb) repetă instrucţiunea movsb de un număr de ori specificat în CX. CX este decrementat după fiecare execuţie a lui movsb. În consecinţă, instrucţiunea rep movsb copiază toate elementele din outputString înapoi în inputString, începând de la dreapta la stânga.

4. Se compilează programul şi se vizualizează lista de simboluri.

5. Se încarcă programul executabil în emulator.

6. Se execută programul pas cu pas, urmărind schimbările din registre, locaţii de memorie, fanioane etc. şi se scriu observaţii. În particular, în timpul executării acestui program ar trebui să urmăriţi schimbările din memorie şi să observaţi cum elementele din cele 2 şiruri de caractere sunt modificate.

7. Se scriu observaţii privitoare la efectul diferitelor instrucţiuni asupra registrelor, fanioanelor şi locaţiilor de memorie.

3.5.3 Exerciţiul 3

Obiectiv. Înţelegerea efectului executării instrucţiunilor push, pop, lods, stos, a directivelor de asamblare equ şi dw şi al folosirii stivei. Înţelegerea convenţiei de stocare de date little endian.

Cerinţă. Să se scrie un program care inserează în stivă toate valorile mai mari decât 8000h dintr-un şir de numere pe 16 biţi fără semn (numit numbers), apoi extrage valorile din stivă copiindu-le la începutul şirului.

Soluţie.

1. Se porneşte emulatorul.

2. Se încarcă programul numit lab3_prog3.asm, folosind butonul Open. Fereastra Source Code trebuie să afişeze următorul program:

```
                org 100h
                jmp init

value           equ 8000h
numbers         dw 8933h, 1240h, 0328h, 99A0h, 0F422h, 0101h

init:           lea SI, numbers
                mov CX, (init-numbers)/2
                mov DX, 0h
                cld

compare:        lodsw
                cmp AX, value
                ja insert

nextElem:       loop compare
                lea DI, numbers
                mov CX, DX

extract:        pop AX
                stosw
                loop extract
```

```
end:        int 20h

insert:     push AX
            inc DX
            jmp nextElem
```

3. Explicaţia programului

3.1. Prima instrucţiune din program (jmp init) sare peste secţiunea de definire a variabilelor.

3.2. Următoarele 2 linii ale acestui program sunt folosite pentru a defini şi iniţializa o valoare constantă, numită value, şi o variabilă: şirul de numere numit numbers. Amintiţi-vă că directiva equ este utilizată pentru a defini valori constante, în timp ce directiva dw este utilozată pentru a defini variabile de tip cuvânt. În consecinţă, elementele şirului sunt cuvinte (numere pe 16 biţi). Șirul este plasat în memorie la o adresă efectivă care va fi discutată mai târziu. Reţineţi că numerele hexazecimale pe 16 biţi care încep cu o literă trebuie să fie precedate de un zero.

3.3. Blocul de instrucţiuni etichetat init este folosit pentru a iniţializa registrele şi fanioanele care vor fi utilizate în program. Instrucţiunea lea SI, numbers încarcă adresa efectivă a şirului numbers în SI. Instrucţiunea mov CX, (init-numbers)/2 calculează diferenţa dintre adresa etichetei init şi adresa primului element din şirul numbers (această diferenţă reprezintă numărul de locaţii de memorie alocate pentru şir), o împarte la 2 (pentru a obţine numărul de elemente ale şirului) şi stochează rezultatul în registrul numărător CX. Instrucţiunea mov DX, 0h iniţializează registrul DX (care va fi folosit pentru a număra câte valori sunt introduse în stivă) cu valoarea 0h. Ultima instrucţiune din acest bloc (cld) resetează valoarea fanionului de direcţie (DF), însemnând că instrucţiunile de string vor itera prin şiruri de la stânga la dreapta.

3.4. Blocul de instrucţiuni etichetat compare iterează prin şirul de numere şi stochează în stivă toate valorile mai mari decât 8000h. Prima instrucţiune din acest bloc (lodsw) copiază numărul curent pe 16 biţi din şirul sursă (valoarea din memorie de la adresa efectivă stocată în SI) în acumulator (AX). Apoi această valoare este comparată cu 8000h. După comparaţie, un salt condiţionat (ja insert) este folosit pentru a sări la eticheta insert dacă elementul curent din şir este mai mare decât 8000h. Instrucţiunea loop decrementează numărătorul CX şi sare înapoi la eticheta compare dacă CX nu este zero.

3.5. Blocul de instrucţiuni etichetat insert împinge în stivă valoarea din AX şi decrementează DX (care numără câte numere au fost stocate în stivă). În final sare înapoi în bucla compare.

3.6. După bucla compare, DI este iniţializat cu adresa efectivă a primului element din şirul numbers şi CX este reiniţializat cu numărul de valori care trebuie extrase din stivă. Bucla extract extrage valorile din stivă, le stochează în acumulator şi din acumulator în şirul destinaţie (care este acelaşi şir numbers).

3.7. În final, instrucţiunea int 20h încheie programul curent.

4. Se compilează programul şi se vizualizează lista de simboluri.

5. Se încarcă programul executabil în emulator.

6. Se execută programul pas cu pas, se urmăresc schimbările din registre, locaţii de memorie, fanioane etc. şi se scriu observaţii. În particular, în timpul executării acestui program ar trebui urmărite schimbările din stivă şi ar trebui observat cum sunt stocate numerele pe 16 biţi în memorie folosind 2 locaţii de memorie (cel mai puţin semnificativ byte la adresa mai mică şi cel mai semnificativ byte la adresa mai mare).

7. Se scriu concluzii referitoare la efectul diferitelor instrucţiuni asupra registrelor, fanioanelor şi locaţiilor de memorie.

3.5.4 Exerciţiul 4

Obiectiv. Sublinierea importanţei semnificaţiei numerelor procesate într-un program de asamblare.

Cerinţă. Apelaţi la exerciţiul anterior şi modificaţi codul sursă astfel încât valorile din şirul numbers să fie privite ca numere cu semn.

Soluţie.

1. Se reface exerciţiul anterior şi se observă că după execuţia programului valorile 8933h, 99A0h şi 0F422h au fost selectate ca fiind mai mari decât 8000h şi au fost copiate la începutul şirului.

2. Se modifică codul sursă, înlocuind instrucţiunea ja insert cu instrucţiunea jg insert. Aceasta este unica schimbare necesară, deoarece:

2.1. Instrucţiunea cmp compară elementul curent din şir cu constanta value indiferent de semnificaţia datelor: aceasta schimbă valorile lui CF şi ZF ca şi cum numerele comparate sunt fără semn şi valorile lui SF şi OF ca şi cum valorile comparate sunt cu semn.

2.2. Valorile din şirul numbers şi constanta value pot fi privite ca valori cu sau fără semn; interpretarea este la alegerea programatorului: programatorul decide dacă foloseşte un salt condiţionat pentru numere fără semn sau unul pentru numere cu semn după comparaţie.

2.3. Restul logicii programului rămâne neschimbată.

3. Se compilează programul şi se vizualizează lista de simboluri.

4. Se execută programul pas cu pas, se urmăresc schimbările din registre, locaţii de memorie, fanioane etc. şi se scriu observaţii.

5. Se scriu concluzii referitoare la efectul diferitelor instrucţiuni asupra registrelor, fanioanelor şi locaţiilor de memorie.

3.5.5 Exerciţiul 5

Cerinţă. Să se scrie un program care copiază toate valorile mai mici decât 1000h dintr-un şir de numere pe 16 biţi cu semn (numit inputArray) într-un alt şir (numit outputArray). În acest exerciţiu trebuie să folosiţi instrucţiunile de string lods şi stos.

3.6 Anexa 1. Exemple de instrucţiuni de transfer de date

MOV – Copiază date

Mod de întrebuinţare: MOV d, s

Argumente:

- d – registru de uz general, registru segment (cu excepţia CS) sau locaţie de memorie.
- s – valoare imediată, registru de uz general, registru segment sau locaţie de memorie.

Efecte: copiază sursa la destinaţie suprascriind valoarea destinaţiei: **(d) ← (s).**

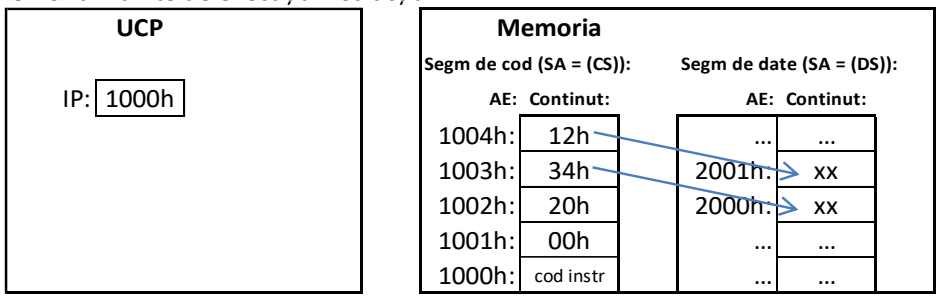
Fanioane: niciunul

Notă: argumentele trebuie să fie de aceeaşi dimensiune (octet, cuvânt).

Exemplu

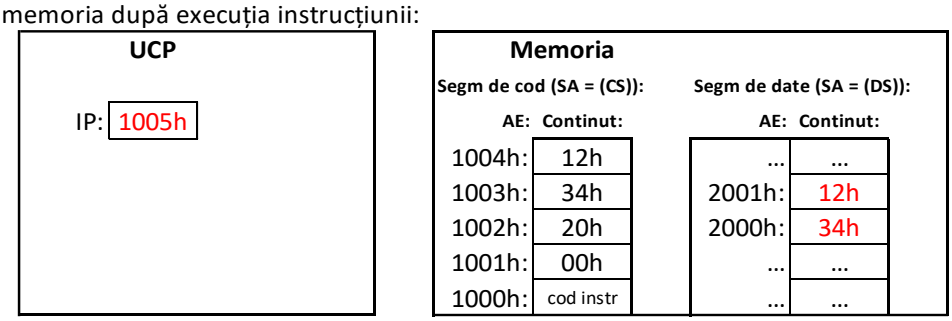
MOV [2000h], 1234h

1. UCP și memoria înainte de execuția instrucțiunii:



2. Valoarea 1234h, care este stocată în segmentul de cod (imediat după codul instrucțiunii), este copiată în segmentul de date (la adresa efectivă specificată în instrucțiunea curentă).

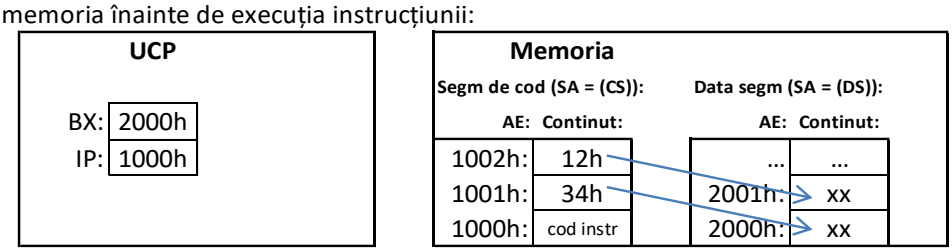
3. UCP și memoria după execuția instrucțiunii:



Exemplu

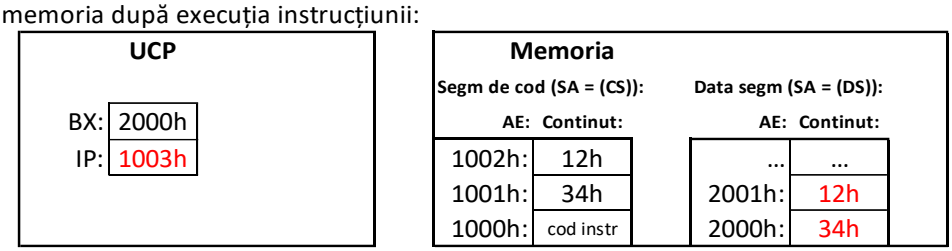
MOV [BX], 1234h

1. UCP și memoria înainte de execuția instrucțiunii:



2. Valoarea 1234h, care este stocată în segmentul de cod (imediat după codul instrucțiunii), este copiată în segmentul de date (la adresa efectivă stocată în registrul BX).

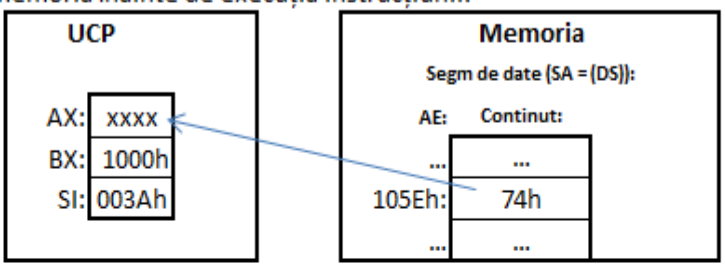
3. UCP și memoria după execuția instrucțiunii:



Exemplu

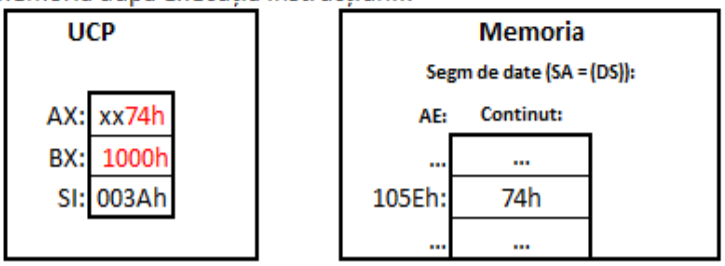
MOV AL, [BX+SI+24h]

1.UCP și memoria înainte de execuția instrucțiunii:



2. Valoarea 74h, care este stocată în segmentul de date (la adresa efectivă obţinută ca suma dintre conţinutul lui BX, conţinutul lui SI şi 24h) este copiată în registrul AL.

3.UCP și memoria după execuția instrucțiunii:



XCHG – Inter-schimbă date

Mod de întrebuinţare: XCHG d, s

Argumente:

- d – registru sau locaţie de memorie.
- s – registru sau locaţie de memorie.

Efecte: inter-schimbă sursa cu destinaţia: **(d) ← (s)**.

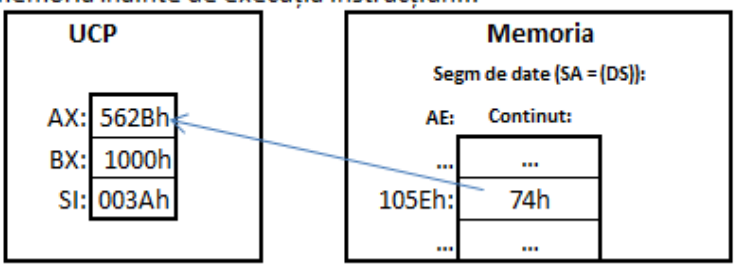
Fanioane: niciunul

Notă: argumentele trebuie să fie de aceeaşi dimensiune (octet, cuvânt). Două locaţii de memorie nu pot fi folosite în aceeaşi instrucţiune.

Exemplu

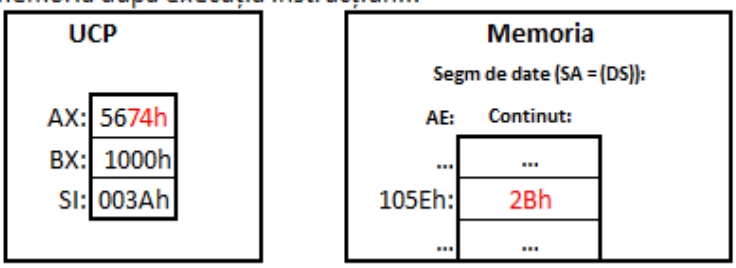
XCHG AL, [BX+SI+24h]

1.UCP și memoria înainte de execuția instrucțiunii:



2. Valoarea 74h, care este stocată în segmentul de date (la adresa efectivă obţinută ca suma dintre conţinutul lui BX, conţinutul lui SI şi 24h) este interschimbată cu valoarea din registrul AL.

3.UCP și memoria după execuția instrucțiunii:



PUSH – Introduce un operand în stivă

Mod de întrebuinţare: PUSH s

Argumente: s – valoare imediată pe 16 biţi, registru sau locaţie de memorie

Efecte: decrementează **SP** cu 2 unităţi şi copiază **s** în vârful stivei:

(SP) ← (SP) – 2

((SS)↕0H+(SP)+1) ← (s)$_{high}$

((SS) ↕0H+(SP)) ← (s)$_{low}$.

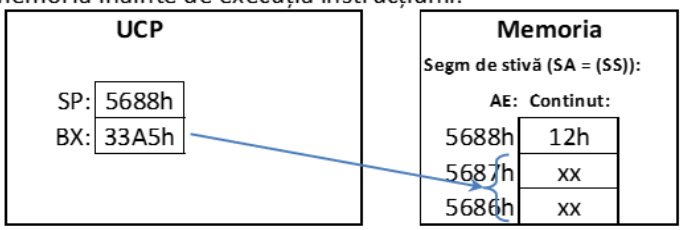
Fanioane: niciunul

Notă: **s** trebuie să fie o valoare pe 16 biţi.

Exemplu

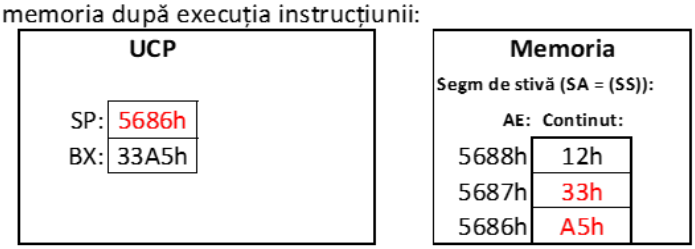
PUSH BX

1. UCP și memoria înainte de execuția instrucțiunii:



2. Valoarea din SP este decrementată cu 2 unități și valoarea 33A5h, care este stocată în BX, este împinsă în stivă (copiată în memorie, în segmentul de stivă, la adresa efectivă stocată în SP).

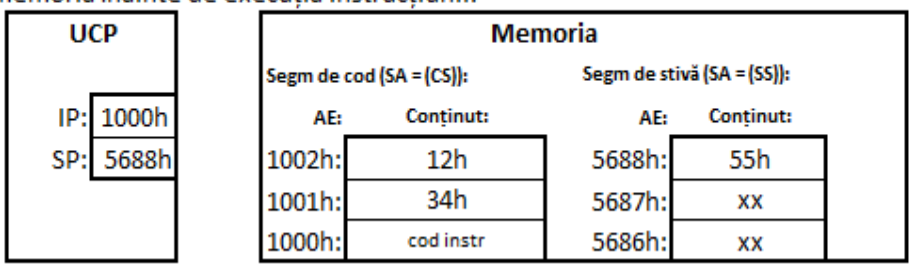
3. UCP și memoria după execuția instrucțiunii:



Exemplu

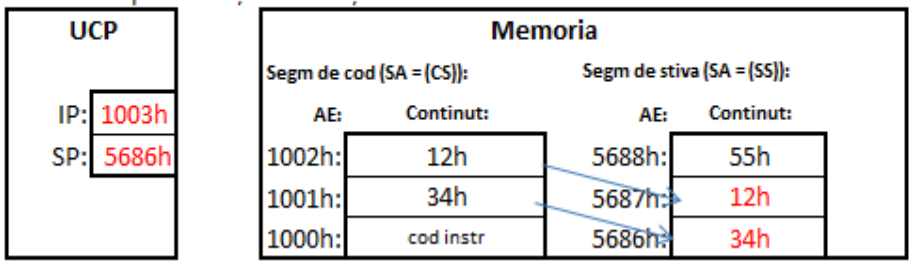
PUSH 1234h

1.UCP și memoria înainte de execuția instrucțiunii:



2. Valoarea lui SP este decrementată cu 2 unități și valoarea 1234h, care este stocată în segmentul de cod (după codul instrucțiunii), este împinsă în stivă.

3.UCP și memoria după execuția instrucțiunii:



Exemplu

POP – Extragerea unui operand din stivă

Mod de întrebuinţare: POP d

Argumente: **d** – registru pe 16 biţi, registru segment sau locaţie de memorie

Efecte: copiază elementul de 16 biţi din vârful stivei în destinaţie şi incrementează **SP** cu 2 unităţi:

$(d)_{high} \leftarrow ((SS)\updownarrow 0H+(SP)+1)$

$(d)_{low} \leftarrow ((SS)\updownarrow 0H+(SP))$

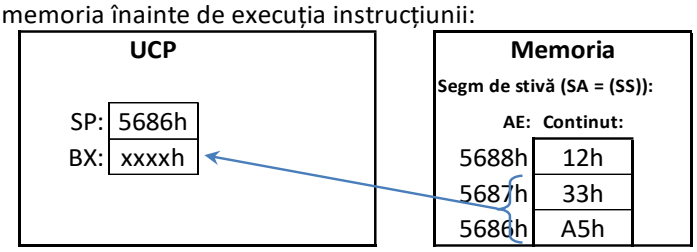
(SP) ← (SP) + 2.

Fanioane: niciunul

Exemplu

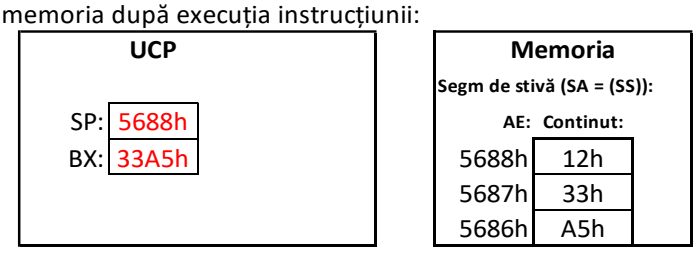
POP BX

1. UCP și memoria înainte de execuția instrucțiunii:



2. Valoarea 33A5h, stocată în vârful stivei (în memorie, în segmentul de stivă, la adresa efectivă stocată în SP), este copiată în BX și SP este incrementat cu 2 unități.

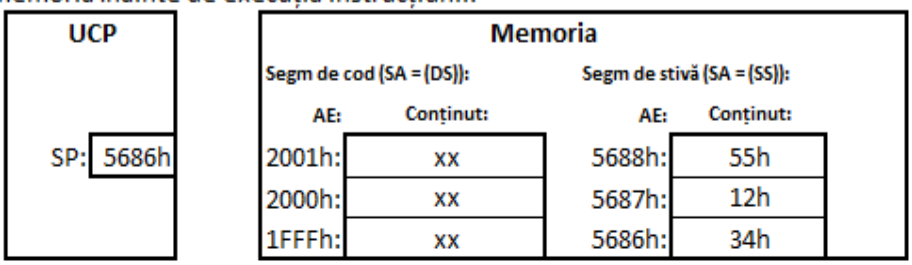
3. UCP și memoria după execuția instrucțiunii:



Exemplu

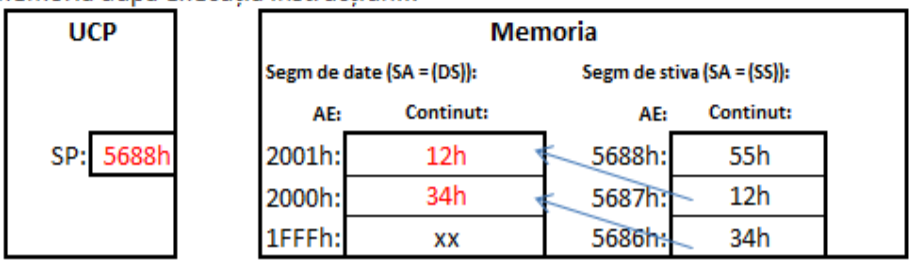
POP [2000h]

1.UCP și memoria înainte de execuția instrucțiunii:



2. Valoarea 1234h, care este stocată în vârful stivei (în memorie, în segmentul de stivă, la adresa efectivă stocată în SP) este copiată în memorie, în segmentul de date, la adresa efectivă 2000h (specificată în instrucțiunea curentă).

3.UCP și memoria după execuția instrucțiunii:



Exemplu

3.7 Anexa 2. Exemple de instrucţiuni pe şiruri/vectori

MOVS – Mută şir

Mod de întrebuinţare: MOVSB / MOVSW
Argumente: niciunul
Efecte:

- movsb: copiază elementul curent pe 8 biţi din şirul sursă peste elementul curent din şirul destinaţie şi incrementează (dacă DF=0) sau decrementează (dacă DF=1) valorile din **SI** şi **DI** cu 1:

((ES)↕0H+(DI)) ← ((DS)↕0H+(SI))
(SI) ← (SI) ± 1
(DI) ← (DI) ± 1.

- movsw: copiază elementul curent pe 16 biţi din şirul sursă peste elementul curent din şirul destinaţie şi incrementează (dacă DF=0) sau decrementează (dacă DF=1) valorile din **SI** şi **DI** cu 2:

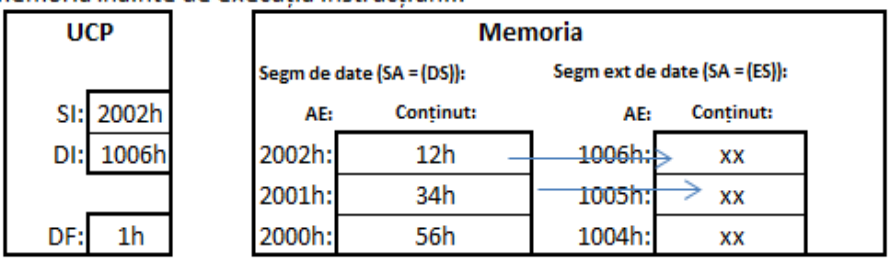
((ES)↕0H+(DI)) ← ((DS)↕0H+(SI))
((ES)↕0H+(DI)+1) ← ((DS)↕0H+(SI)+1)
(SI) ← (SI) ± 2
(DI) ← (DI) ± 2.

Fanioane: niciunul
Notă: poate fi prefixată de rep, repe/repz, repne/repnz

Exemplu

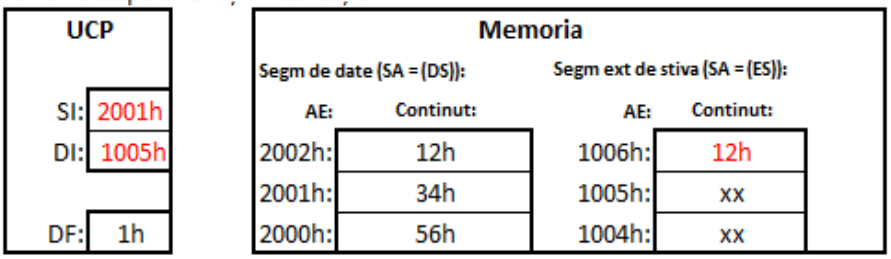
MOVSB

1.UCP și memoria înainte de execuția instrucțiunii:



2. Elementul curent pe 8 biți din șirul sursă (12h), stocat în segmentul de date, la adresa efectivă specificată în SI este copiat în șirul destinație, stocat în segmentul extins de date, la adresa efectivă specificată în DI. SI și DI sunt decrementate (DF=1) cu o unitate.

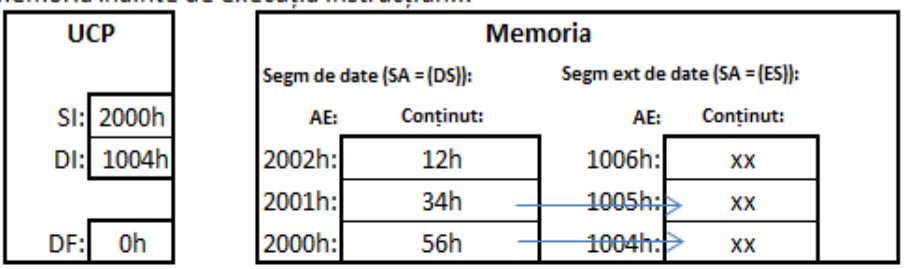
3.UCP și memoria după execuția instrucțiunii:



Exemplu

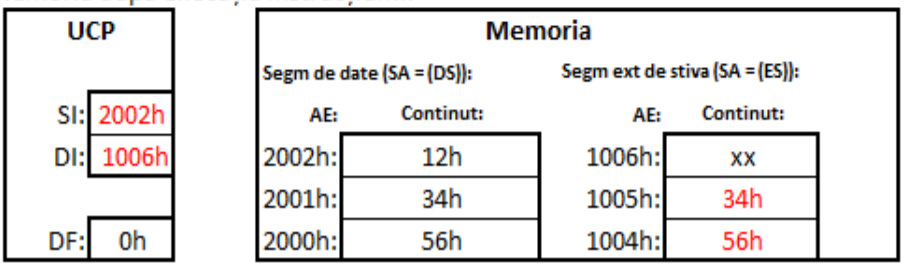
MOVSW

1.UCP și memoria înainte de execuția instrucțiunii:



2. Elementul curent pe 16 biți din șirul sursă (5634h), stocat în segmentul de date, la adresa efectivă specificată în SI este copiat în șirul destinație, stocat în segmentul extins de date, la adresa efectivă specificată în DI. SI și DI sunt incrementate (DF=0) cu 2 unități.

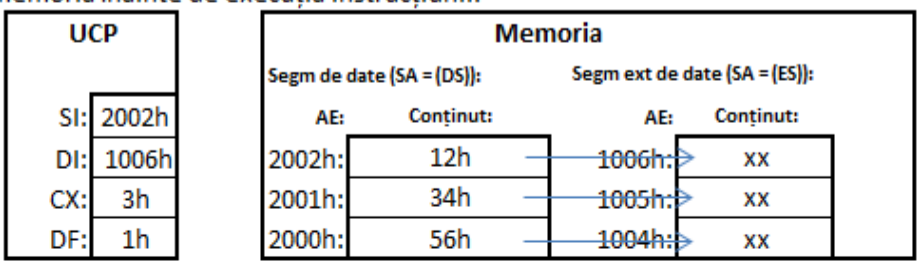
3.UCP și memoria după execuția instrucțiunii:



Exemplu

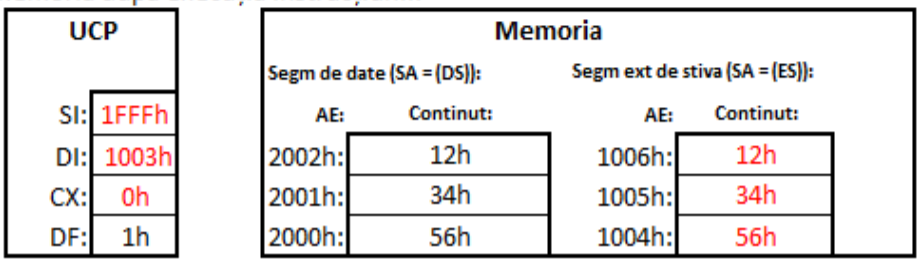
REP MOVSB

1.UCP și memoria înainte de execuția instrucțiunii:



2. 3 elemente pe 8 biți fiecare din șirul sursă, stocate în segmentul de date, începând cu adresa efectivă specificată în SI sunt copiate în șirul destinație, stocat în segmentul extins de date, începând cu adresa efectivă specificată în DI. SI și DI sunt decrementate de 3 ori cu o unitate.

3.UCP și memoria după execuția instrucțiunii:



LODS – Încarcă şir

Mod de întrebuinţare: LODSB / LODSW

Argumente: niciunul

Efecte:

- lodsb: copiază elementul curent pe 8 biţi din şirul sursă în acumulator şi incrementează (dacă DF=0) sau decrementează (dacă DF=1) valoarea lui **SI**:

(AL) ← ((DS)↕0H+(SI))

(SI) ← (SI) ± 1.

- lodsw: copiază elementul curent pe 16 biţi din şirul sursă în acumulator şi incrementează (dacă DF=0) sau decrementează (dacă DF=1) valorea lui **SI** cu 2:

(AL) ← ((DS)↕0H+(SI))

(AH) ← ((DS)↕0H+(SI)+1)

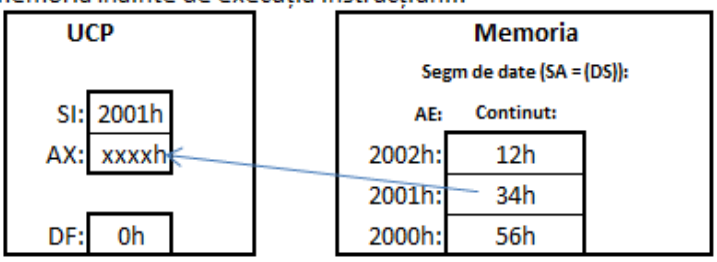
(SI) ← (SI) ± 2.

Fanioane: niciunul

Exemplu

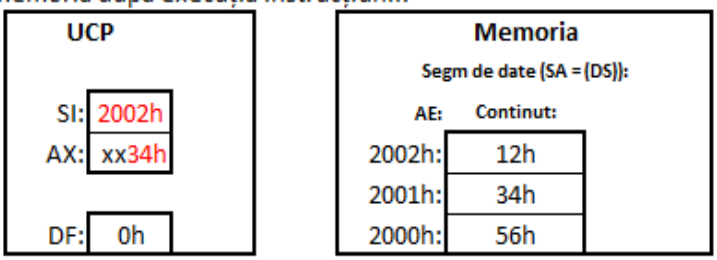
LODSB

1.UCP și memoria înainte de execuția instrucțiunii:



2. Elementul curent pe 8 biți din șirul sursă (stocat în memorie, în segmentul de date, la adresa efectivă specificată în SI) este copiat în acumulator. SI este incrementat (DF=0) cu 1.

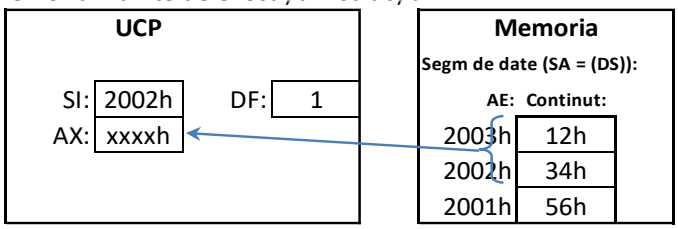
3.UCP și memoria după execuția instrucțiunii:



Exemplu

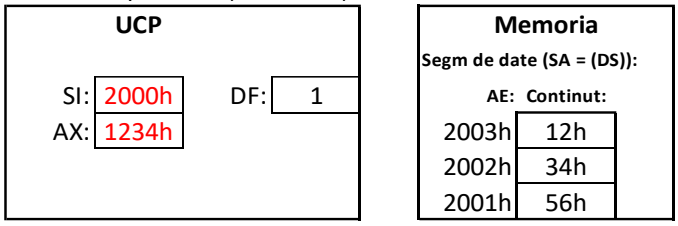
LODSW

1. UCP și memoria înainte de execuția instrucțiunii:



2. Elementul curent pe 16 biți din șirul sursă (stocat în memorie, în segmentul de date, la adresa efectivă specificată în SI) este copiat în acumulator. SI este decrementat (DF=1) cu 2.

3. UCP și memoria după execuția instrucțiunii:



STOS – Stochează şir

Mod de întrebuinţare: STOSB / STOSW

Argumente: niciunul

Efecte:

- stosb: copiază valoarea pe 8 biţi din acumulator în şirul destinaţie şi incrementează (dacă DF=0) sau decrementează (dacă DF=1) valoarea lui **DI**:

((ES)↕0H+(DI)) ← (AL)
(DI) ← (DI) ± 1.

- stosw: copiază valoarea pe 16 biţi din acumulator în şirul destinaţie şi incrementează (dacă DF=0) sau decrementează (dacă DF=1) valorea lui **DI** cu 2:

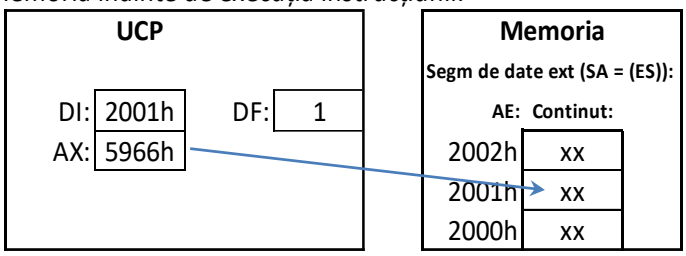
((ES)↕0H+(DI)) ← (AL)
((ES)↕0H+(DI)+1) ← (AH)
(DI) ← (DI) ± 2.

Fanioane: niciunul

Exemplu

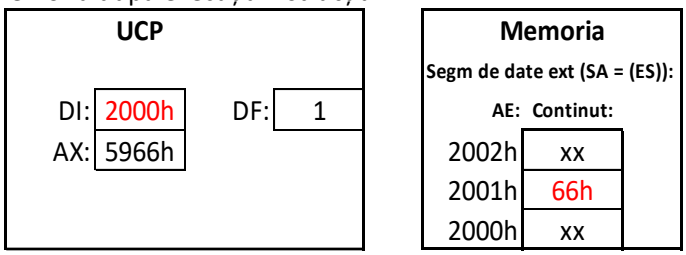
STOSB

1. UCP și memoria înainte de execuția instrucțiunii:



2. Valoarea pe 8 biți din acumulator este copiată în șirul destinație (stocat în memorie, în segmentul extins de date, la adresa efectivă specificată în DI). DI este decrementat (DF=1) cu 1.

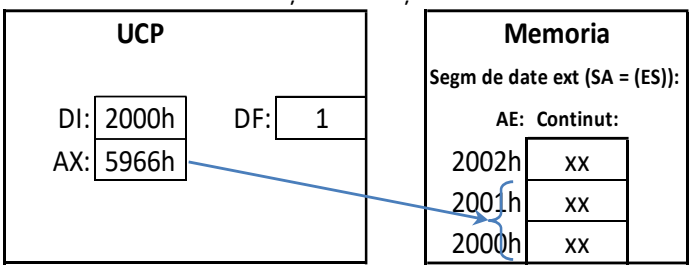
3. UCP și memoria după execuția instrucțiunii:



Exemplu

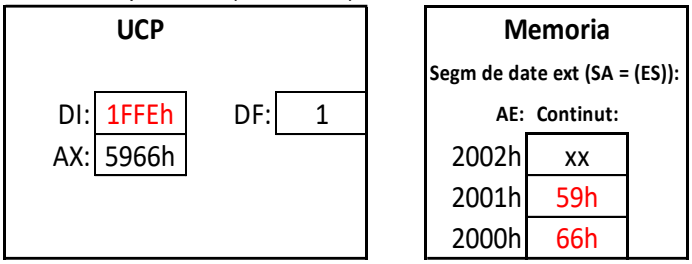
STOSW

1. UCP și memoria înainte de execuția instrucțiunii:



2. Valoarea pe 16 biți din acumulator este copiată în șirul destinație (stocat în memorie, în segmentul extins de date, la adresa efectivă specificată în DI). DI este decrementat (DF=1) cu 2.

3. UCP și memoria după execuția instrucțiunii:



3.8 Anexa 3. Tabelul ASCII

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

4. ALTE INSTRUCȚIUNI DE CONTROL AL PROGRAMULUI

4.1 Instrucţiunile call şi ret pentru procesoarele x86

În afara instrucţiunilor **jump** şi **loop**, arhitectura x86 oferă încă două instrucţiuni de control al programului, folosite pentru a chema (apela) subprograme şi a se întoarce în programul principal, numite **call** şi **ret** (vezi Tabelul 12).

Tabelul 12. Instrucţiunile call/ret în x86

Instrucţiunea	Sintaxă	Descriere
CALL	CALL d	Salvează adresa de întoarcere în stivă şi execută un salt (necondiţionat) la adresa destinaţie (d)
RET - Return	RET	Extrage adresa stocată în vârful stivei şi execută un salt (necondiţionat) la acea adresă

Instrucţiunea **call** scrie adresa următoarei instrucţiuni în stivă (vezi instrucţiunea push pentru detalii), iar apoi execută un salt necondiţionat la adresa indicată de operand. Spre deosebire de instrucţiunile simple de salt, instrucţiunea **call** salvează adresa de întoarcere în stivă; această adresă va fi utilizată pentru întoarcerea în programul principal atunci când se încheie subprogramul.

Instrucţiunea **ret** implementează un mecanism de întoarcere din subprogram. Aceasta scoate din stivă adresa de întoarcere (vezi instrucţiunea pop pentru detalii), iar apoi execută un salt necondiţionat la acea adresă.

Se observă că aceste două instrucţiuni folosesc implicit stiva şi modifică indicatorul stivei (**SP**).

Detalii şi exemple de utilizare privitor la instrucţiunile **call**/**ret** sunt oferite în Secţiunea 4.2.

4.2 Utilizarea instrucţiunilor call şi ret pentru procesoarele x86

Instrucţiunile **call** şi **ret** nu specifică nimic în legătură cu parametrii de intrare/ieşire. Aşa cum este arătat în Tabelul 1, parametrii I/O nu sunt trecuţi în/din subprogram ca operanzi ai instrucţiunii. Există câteva convenţii de programare care specifică modul în care aceste date sunt transferate în/din subprograme. În acest laborator vom folosi regulile explicate mai jos.

4.2.1 Reguli privind programul apelant

Pentru a face un apel la un alt program, programul apelant trebuie:

1. Înainte de a chema un alt program, programul apelant trebuie să salveze conţinutul anumitor registre de care are nevoie şi după apelul de subprogram. De exemplu, în cazul în care programul apelant are nevoie de valorile lui **AX**, **BX** şi **DI** după revenire, programul apelant trebuie să salveze valorile din aceste registre în stivă aşa încât să poată fi restaurate după întoarcerea din celălalt program.

2. Pentru a oferi parametri subprogramului, sunt disponibile mai multe metode:

- folosind valorile înscrise în anumite registre;
- folosind starea anumitor fanioane;
- folosind zone de memorie care vor fi egal accesibile programului apelant şi apelat;
- folosind stiva.

De exemplu, în cazul utilizării stivei: programul apelant trebuie să-i introducă în stivă înainte de apel. Parametrii trebuie introduşi în ordine inversă (i.e. ultimul parametru trebuie introdus primul). Deoarece stiva creşte în jos, primul parametru va fi stocat la adresa cea mai mică.

3. Pentru a chema un subprogram, se foloseşte instrucţiunea **call**. Aceasta plasează adresa de întoarcere în vârful stivei şi realizează saltul la prima instrucţiune a programului apelat.

După întoarcerea din subprogram, pentru a restaura starea maşinii, programul apelant trebuie să:

1. Extragă parametrii din stivă. Astfel, stiva este adusă la starea de dinainte de executarea apelului.
2. Restaureze conţinutul registrelor salvate de apelant, prin extragerea lor din stivă.

4.2.2 Reguli privind programul apelat

Când începe subprogramul:

1. Programul apelat trebuie să încarce **BP** cu valoarea din **SP** (**mov BP**, **SP**). În acest fel, programul apelat va putea accesa parametrii de intrare (din stivă), folosind adresele **BP+2** (primul parametru), **BP+4** (al doilea parametru) etc., chiar dacă **SP** se poate schimba în timpul execuţiei subprogramului.

Atenție la accesarea parametrilor de intrare cu **mov** folosind adrese efective **BP**, **BP+2**, **BP+4**, etc; structura stivei rămâne neschimbată.

Înainte de încheierea subprogramului, programul apelat trebuie:

1. Să extragă tot ce a fost introdus în stivă în timpul subprogramului.

2. Să stocheze parametrul de ieşire într-un registru sau mai multe.

3. Să se întoarcă la apelant prin executarea instrucţiunii **ret**. Această instrucţiune va găsi şi elimina adresa de întoarcere din stivă (dacă pasul 1de mai sus a fost executat corect).

4.2.3 Exemplu de call şi return dintr-un program în altul

Să presupunem că programul principal utilizează registrul **BX** pentru a stoca date importante şi are nevoie să facă apel către un alt program cu doi parametri de intrare (1234h şi valoarea din registrul **DX**) şi să salveze un rezultat în **AX**. Considerăm următorul bloc de instrucţiuni:

```
main:       push   AX      ; salvează valoarea din AX în stivă
            push   BX      ; salvează valoarea din BX în stivă
            push   DX      ; inserează al doilea parametru de intrare în stivă
            push   1234h   ; inserează primul parametru de intrare în stivă
            call   subprogram     ; apelul la subprogram
            add    SP, 4h  ; extrage parametrii de intrare din stivă
            mov    SI, AX  ; copiază rezultatul în SI, deoarece AX va fi restituit
            pop    BX      ; restituie valoarea anterioară a lui BX (din stivă)
            pop    AX      ; restituie valoarea anterioară a lui AX (din stivă)
            …
            …
subprogram: mov    BP, SP  ; plasează BP pentru a indica spre acelaşi element ca şi SP
            …               ; corpul subprogramului, în care parametrii de intrare pot
            …               ; fi accesaţi folosind adresele BP+2, BP+4 şi BP+6
            …               ; aici registrele AX, BX şi CX pot fi modificate
            mov    AX, 98BBh     ; stochează rezultatul în AX
            ret             ; întoarcerea la programul principal
```

În timpul execuţiei acestei părţi de cod, stiva este modificată aşa cum este ilustrat în figura următoare. În această figură am folosit următoarele convenţii:

- toate numerele sunt în hexazecimal;
- valorile anumitor registre sunt scrise în partea stângă;
- valorile din anumite locaţii de memorie sunt scrise în partea dreaptă; adresele efective ale locaţiilor de memorie sunt scrise cu font mai mic;
- celulele colorate fac parte din stivă şi celulele albe nu fac parte din stivă;
- xxxx înseamnă că valoarea stocată în registru/locaţie de memorie este necunoscută.

începutul programului			
			…
SP:	FEE6	FEE6	xxxx
BP:	xxxx	FEE4	xxxx
		FEE2	xxxx
AX:	1122	FEE0	xxxx
BX:	3344	FEDE	xxxx
DX:	5566	FEDC	xxxx
		FEDA	xxxx
			…

după push AX			
			…
SP:	FEE4	FEE6	xxxx
BP:	xxxx	FEE4	1122
		FEE2	xxxx
AX:	1122	FEE0	xxxx
BX:	3344	FEDE	xxxx
DX:	5566	FEDC	xxxx
		FEDA	xxxx
			…

după push BX			
			…
SP:	FEE2	FEE6	xxxx
BP:	xxxx	FEE4	1122
		FEE2	3344
AX:	1122	FEE0	xxxx
BX:	3344	FEDE	xxxx
DX:	5566	FEDC	xxxx
		FEDA	xxxx
			…

după push DX			
			…
SP:	FEE0	FEE6	xxxx
BP:	xxxx	FEE4	1122
		FEE2	3344
AX:	1122	FEE0	5566
BX:	3344	FEDE	xxxx
DX:	5566	FEDC	xxxx
		FEDA	xxxx
			…

după push 1234h			
			…
SP:	FEDE	FEE6	xxxx
BP:	xxxx	FEE4	1122
		FEE2	3344
AX:	1122	FEE0	5566
BX:	3344	FEDE	1234
DX:	5566	FEDC	xxxx
IP:	0500	FEDA	xxxx
			…

după call subprogram			
			…
SP:	FEDC	FEE6	xxxx
BP:	xxxx	FEE4	1122
		FEE2	3344
AX:	1122	FEE0	5566
BX:	3344	FEDE	1234
DX:	5566	FEDC	0502
		FEDA	xxxx
			…

după mov BP, SP			
			…
SP:	FEDC	FEE6	xxxx
BP:	FEDC	FEE4	1122
		FEE2	3344
AX:	1122	FEE0	5566
BX:	3344	FEDE	1234
DX:	5566	FEDC	0502
		FEDA	xxxx
			…

după mov AX, 98BBh			
			…
SP:	FEDC	FEE6	xxxx
BP:	FEDC	FEE4	1122
		FEE2	3344
AX:	98BB	FEE0	5566
BX:	xxxx	FEDE	1234
DX:	xxxx	FEDC	0502
		FEDA	xxxx
			…

după ret			
			…
SP:	FEDE	FEE6	xxxx
BP:	FEDC	FEE4	1122
		FEE2	3344
AX:	98BB	FEE0	5566
BX:	xxxx	FEDE	1234
DX:	xxxx	FEDC	0502
IP:	0502	FEDA	xxxx
			…

după add SP, 4h			
			…
SP:	FEE2	FEE6	xxxx
BP:	FEDC	FEE4	1122
		FEE2	3344
AX:	98BB	FEE0	5566
BX:	xxxx	FEDE	1234
DX:	xxxx	FEDC	0502
		FEDA	xxxx
			…

după pop BX			
			…
SP:	FEE4	FEE6	xxxx
BP:	FEDC	FEE4	1122
		FEE2	3344
AX:	98BB	FEE0	5566
BX:	3344	FEDE	1234
DX:	xxxx	FEDC	0502
		FEDA	xxxx
			…

după pop AX			
			…
SP:	FEE6	FEE6	xxxx
BP:	FEDC	FEE4	1122
		FEE2	3344
AX:	1122	FEE0	5566
BX:	3344	FEDE	1234
DX:	xxxx	FEDC	0502
		FEDA	xxxx
			…

Figura 3. Modificările din stivă în timpul executării de către UCP a codului sursă dat ca exemplu

4.3 Exerciţii

4.3.1 Exerciţiul 1

Obiectiv. Înţelegerea modului în care parametrii de intrare/ieşire sunt trimişi în/primiţi din subprograme prin intermediul stivei/registrelor.

Cerinţă. Să se scrie un subprogram care primeşte ca parametri de intrare trei numere pe 16 biţi, fără semn (trimise prin intermediul stivei) şi returnează media lor (în AX). Să se exemplifice utilizarea acestui subprogram într-un program care înlocuieşte fiecare element dintr-un şir de numere pe 16 biţi cu media dintre elementul respectiv şi vecinii săi.

Soluţie.

Notă: Soluţia acestui exerciţiu implică scrierea subprogramului şi apelarea lui pentru fiecare secvenţă de 3 numere din şirul de intrare.

1. Se porneşte emulatorul.

2. Se încarcă programul numit lab4_prog1.asm, utilizând butonul Open. Fereastra Source Code trebuie să afişeze următorul program:

```
                org 100h
                jmp main

dataset         dw 12h, 18h, 1Ah, 16h, 08h, 08h, 12h
windowSize      dw 3h

main:           lea BX, dataset
                mov SI, 2h
                mov CX, (windowSize-dataset)/2-2

process:        push [BX+SI-2]
                push [BX+SI]
                push [BX+SI+2]
                call average
                mov [BX+SI], AX
                add SP, 6h
                add SI, 2h
                loop process

                int 20h

average:        mov BP, SP
                mov AX, [BP+2]
                mov DX, 0h
                add AX, [BP+4]
```

```
adc DX, 0h
add AX, [BP+6]
adc DX, 0h
div word ptr windowSize
ret
```

3. Explicaţia programului

3.1. Se observă segmentarea codului sursă în 3 zone:

3.1.1. o zonă care defineşte şirul de numere (numită dataset) şi numărul de elemente care vor fi mediate la fiecare pas (windowSize),

3.1.2. o zonă etichetată main, care include şi zona etichetată process, reprezentând programul principal,

3.1.3. o zonă etichetată average, reprezentând subprogramul.

3.2. Prima instrucţiune din acest program (jmp main) sare peste zona de declarare a variabilelor.

3.3. În acest program, BX este utilozat pentru stocarea adresei de început a şirului de date şi este iniţializat de la început (lea BX, dataset). SI stochează indexul elementului curent din şir. Valoarea sa este dublată deoarece fiecare emenent al şirului este stocat în memorie folosind două locaţii de memorie (prin urmare iterând dataset înseamnă incrementarea lui SI cu 2). Primul element care este procesat este 18h, nu 12h, deoarece programul trebuie să înlocuiască fiecare element cu media dintre el şi vecinii lui din stânga-dreapta (şi primul element nu are vecin în stânga). Acesta este motivul pentru care SI este iniţializat cu 2h. CX stochează numărul de elemente care trebuie procesate. CX este iniţializat cu numărul de elemente minus 2, întrucât primul şi ultimul element din şir nu vor fi procesate. Se observă că windowSize-dataset reprezintă numărul de locaţii de memorie alocate pentru şirul de numere şi această valoare trebuie împărţită la doi pentru a obţine numărul real de elemente din dataset (fiecare element ocupă 2 locaţii de memorie).

3.4. Zona etichetată process începe prin împingerea în stivă a parametrilor de intrare pentru subprogram. Aceşti parametri de intrare sunt: vecinul din stânga al elementului curent, elementul curent şi vecinul din dreapta al elementului curent. Cele 3 valori se găsesc în memorie la adresele BX+SI-2, BX+SI şi BX+SI+2. După ce aceste valori sunt împinse în stivă, următoarea instrucţiune (call average) apelează subprogramul. Se ignoră instrucţiunile din subprogram pentru moment, concentrându-ne pe ceea ce se întâmplă după întoarcerea din subprogram. Următoarea instrucţiune după apel (mov [BX+SI], AX) suprascrie elementul curent din şir cu valoarea returnată, ca rezultat, din subprogram. Parametrii de intrare sunt extraşi din stivă prin adunarea cu 6h a lui SP (add SP, 6h): 3 parametri a câte 2 octeţi fiecare. Mergând mai departe, următoarea instrucţiune (add SI, 2h) incrementează indexul pentru a indica spre următorul element. În final, instrucţiunea loop decrementează CX şi, dacă încă mai există numere care trebuie procesate (CX>0), execută saltul înapoi la eticheta process. Când

CX ajunge la valoarea 0, procesorul continuă prin executarea următoarei instrucţiuni (int 20h), care încheie programul curent.

3.5. Subprogramul average realizează media a 3 numere primite ca parametri de intrare prin intermediul stivei. Instrucţiunile sunt foarte asemănătoare cu cele din primul program al primului laborator (lab1_prog1.asm), care executa media a 3 numere constante. Diferenţa este că acum numerele sunt stocate iniţial în stivă. Registrul BP este încărcat cu valoarea curentă a lui SP (mov BP,SP), aşa încât parametrii de intrare pot fi accesaţi folosind adresele BP+2, BP+4 şi BP+6 (pentru a vă aminti de ce, uitaţi-vă la Figura 1). Datorită faptului că suma a trei numere pe 16 biţi poate rezulta într-un număr pe 17 biţi, suma va fi realizată folosind DX□AX. În consecinţă, AX este iniţializat cu primul număr (mov AX, [BP+2]), iar DX este iniţializat cu 0h. După aceea, al doilea şi al treilea număr sunt adunate la AX şi fanionul de transport este adunat la DX de fiecare dată. În final, suma (numărul pe 32 de biţi format ca DX□AX) este împărţit la 3h (valoare pe 16 biţi). Câtul este stocat implicit în AX şi restul este stocat implicit în DX. Ultima instrucţiune a subprogramului (ret) se întoarce la programul apelant, folosind adresa stocată în vârful stivei.

4. Se compilează programul şi se vizualizează lista de simboluri.

4.1. Se face click pe butonul Compile pentru a compila programul.

4.2. Veţi fi îndrumaţi să salvaţi fişierul executabil. Se salvează cu numele recomandat.

4.3. Se vizualizează statusul compilării în caseta de Assembler Status. Dacă programul a fost editat corect, mesajul ar trebui să fie "lab4_prog1.asm is assembled successfully into 75 bytes."

4.4. Se face click pe butonul View şi apoi pe Symbol Table pentru a vizualiza tabela de simboluri asociată programului. Informaţiile din această listă trebuie interpretate după cum urmează:

- simbolurile dataset şi windowSize sunt variabile cuvânt (size=2) stocate în memorie la adresele 0102h şi 0110h. Se observă că în ciuda faptului că dataset defineşte un şir, simbolul dataset reprezintă doar adresa de început a acestui şir. Aceste simboluri pot fi asociate cu indicatorii din C.

- simbolurile main, process şi average sunt etichete ale unor instrucţiuni din program şi sunt asociate cu adresele acestor instrucţiuni (0112h, 011Bh şi 0132h).

4.5. Lista de simboluri vă va ajuta să găsiţi datele cu care lucraţi (în memorie).

5. Se încarcă programul executabil în emulator.

5.1. Se face click pe butonul Run pentru a încărca programul în emulator şi a-l executa.

6. Se execută programul pas cu pas, se observă schimbările produse asupra registrelor, locaţiilor de memorie, fanioanelor etc şi se notează observaţii.

6.1. Se face click pe butonul Reload pentru a reîncărca programul executat.

6.2. Se inspectează fereastra Emulator şi se observă că:

6.2.1. Instrucţiunea curentă (jmp 0112h) este evidenţiată. Aceasta este prima instrucţiune a programului şi a fost încărcată la adresa logică 0700:0100 (adresă segment : adresă efectivă). Adresa efectivă a fost impusă de directiva de asamblare org 100h. Această instrucţiune este echivalentă cu instrucţiunea care a fost scrisă în program (jmp main), deoarece simbolul main a fost înlocuit cu adresa 0112h.

6.2.2. Valoarea din registrul IP (registrul care stochează adresa efectivă a instrucţiunii curente) este 0100h.

6.3. Se face click pe View Menu -> Memory pentru a vedea statusul curent al memoriei. În Address Text Box se scrie adresa variabilei dataset: se lasă adresa segment nemodificată (0700) şi se înlocuieşte adresa efectivă (0100) cu adresa efectivă a lui inputString (0102). Se face click pe butonul Update şi se observă că:

6.3.1. Adresa de început este acum 0700:0102, iar valorile stocate în memorie sunt 12h, 00h, 18h, 00h, 1Ah, 00h, 16h, 00h, … . Se observă că acestea sunt numerele din dataset: fiecare valoare (12h, 18h, 1Ah, 16h) ocupă 2 locaţii de memorie cu cel mai semnificativ byte stocat la adresa mai mare şi cel mai puţin semnificativ byte stocat la adresa mai mică (convenţia little endian).

6.4. Se face click pe butonul Single Step pentru a executa prima instrucţiune (jmp 0112h) şi se observă că registrul IP a fost încărcat cu adresa lui 0112h: a fost executat un salt peste zona de declarare a datelor (la prima instrucţiune din programul main).

6.5. Se execută următoarele 3 instrucţiuni şi se observă că registrele BX, SI şi CX au fost încărcate cu valorile 0102h, 0002h şi 0005h.

6.6. Se face click pe View -> Stack pentru a vizualiza Stack Window. În acelaşi timp se inspectează Fereastra Emulator şi se observă că:

6.6.1. Valorile stocate în registrele SS şi SP sunt 0700h şi FFFEh. În consecinţă, elementul din vârful stivei este stocat în memorie la adresa 0700:FFFE.

6.6.2. Valoarea evidenţiată în Fereastra Stivei (elementul din vârful stivei) este stocată la adresa 0700:FFFE.

6.7. Se execută următoarea instrucţiune (push [BX+SI-2]) şi se observă că:

6.7.1. Valoarea din registrul SP a fost decrementată cu 2 pentru a face loc pentru încă un element în stivă. SP stochează acum valoarea FFFCh.

6.7.2. Numărul pe 16 biţi stocat în memorie la adresa BX+SI-2=102h (0012h) este acum parte a stivei (de fapt este noul vârf al stivei).

6.8. Se execută următoarea instrucţiune (push [BX+SI]) şi se observă că:

6.8.1. Valoarea din registrul SP a fost decrementată cu 2 pentru a face loc pentru încă un element în stivă. SP stochează acum valoarea FFFAh.

6.8.2. Numărul pe 16 biţi stocat în memorie la adresa BX+SI=104h (0018h) este acum parte a stivei (de fapt este noul vârf al stivei).

6.9. Se execută următoarea instrucţiune (push [BX+SI+2]) şi se observă că:

6.9.1. Valoarea din registrul SP a fost decrementată cu 2 pentru a face loc pentru încă un element în stivă. SP stochează acum valoarea FFF8h.

6.9.2. Numărul pe 16 biţi stocat în memorie la adresa BX+SI+2=108h (001Ah) este acum parte a stivei (de fapt este noul vârf al stivei).

6.10. Se inspectează Fereastra Emulator şi Fereastra Stivei şi se observă că:

6.10.1. Instrucţiunea curentă este call 0132h. Amintiţi-vă că 0132h este adresa asociată cu subprogramul average.

6.10.2. Valoarea stocată în registrul IP este 0123h. Aceasta înseamnă că adresa instrucţiunii curente din programul main este 0123h.

6.10.3. Valoarea registrului SP este FFF8h.

6.10.4. Elementul din vârful stivei este 001Ah.

6.11. Se execută următoarea instrucţiune (call 0132h) şi se observă că:

6.11.1. Valoarea din registrul SP a fost decrementată cu 2 pentru a face loc pentru încă un element în stivă. SP stochează acum valoarea FFF6h.

6.11.2. Noua valoare inserată în stivă este 0126h, care este probabil adresa următoarei instrucţiuni după call average.

6.11.3. Valoarea din registrul IP este 0132h (adresa spre care indica instrucţiunea call).

6.12. Se execută următoarea instrucţiune (mov BP, SP) şi se observă că valoarea din registrul SP este copiată în registrul BP.

6.13. Se execută instrucţiunile următoare pas cu pas şi se observă cum parametrii de intrare în stivă (valorile de la adresele BP+2, BP+4 şi BP+6) sunt adunate la AX.

6.14. Când ajungeţi la instrucţiunea div, calculaţi restul şi câtul împărţirii pe o foaie de hârtie. Apoi executaţi instrucţiunea şi observaţi că registrul AX este încărcat cu câtul corect (0016h) şi registrul DX este încărcat cu restul corect (0002h).

6.15. Se inspectează Fereastra Emulator şi Fereastra Stivei şi se observă că:

6.15.1. Stiva nu s-a schimbat în timpul execuţiei subprogramului (valoarea din vârful stivei este tot 0126h).

6.15.2. Valoarea din registrul IP este 0146h.

6.16. Se execută instrucţiunea ret şi se observă că:

6.16.1. Adresa de întoarcere a fost extrasă din stivă (SP a fost incrementat cu 2h).

6.16.2. Adresa de întoarcere a fost utilizată pentru a executa saltul înapoi la programul main: registrul IP a fost încărcat cu adresa extrasă din stivă (0126h).

6.17. Se execută următoarea instrucţiune (mov [BX+SI], AX) şi se observă că valoarea din memorie de la adresa BX+SI (elementul curent din dataset) a fost modificată din 0018h în 0016h.

6.18. Se execută următoarea instrucţiune (add SP, 6h) şi se observă că cei 3 parametri de intrare au fost extraşi din stivă (valoarea evidenţiată din stivă este acum valoarea de la adresa FFFEh).

6.19. Se execută următoarea instrucţiune (add SI, 2h) şi se observă că valoarea din SI a fost incrementată cu 2h. Aceasta este echivalentă cu trecerea la al treilea element din şir.

6.20. Instrucţiunea curentă este loop 011Bh. Verificaţi lista de simboluri şi amintiţi-vă că eticheta process a fost asociată cu adresa 011B. Se ştie că registrul CX stoca valoarea 0005h. Se execută instrucţiunea şi se observă că:

6.20.1. Valoarea din CX a fost decrementată.

6.20.2. Procesorul a executat un salt la instrucţiunea cu adresa 011Bh (prima instrucţiune din bucla process).

6.21. Se continuă executarea instrucţiunilor pas cu pas, observându-se modificările asupra dataset, asupra stivei şi a lui CX (care stochează numărul de elemente neprocesate). Opriţi-vă atunci când CX atinge valoarea 1. Se execută instrucţiunea (loop 011Bh) şi se observă că:

6.21.1. Valoarea din CX a fost decrementată şi acesta stochează acum valoarea 0h.

6.21.2. Procesorul nu execută saltul înapoi la instrucţiunea cu adresa 011Bh (prima instrucţiune din bucla process), ci continuă cu următoarea instrucţiune (int 20h).

6.22. Instrucţiunea curentă este int 20h. Se face click pe butonul Single Step de două ori şi se observă că o casetă de mesaje este afişată, spunând că programul a returnat controlul sistemului de operare. Se face click pe Ok.

7. Se scriu concluzii privitoare la efectul diferitelor instrucţiuni asupra registrelor, fanioanelor şi locaţiilor de memorie.

4.3.2 Exerciţiul 2

Obiectiv. Înţelegerea modului în care parametrii de intrare/ieşire sunt trimişi/primiţi din subprograme prin intermediul stivei/registrelor.

Cerinţă. Să se scrie un subprogram care primeşte ca parametri de intrare adresa de început şi numărul de elemente al unui şir de numere pe 16 biţi cu semn şi găseşte maximul din şir. Subprogramul trebuie să returneze valoarea maximă în registrul AX şi adresa maximului în registrul DI. Să se exemplifice utilizarea acestui subprogram într-un program care îl apelează o dată şi apoi se încheie.

Soluţie.

Notă: Soluţia acestui exerciţiu implică scrierea subprogramului şi apelarea lui o dată dintr-un program principal.

1. Se porneşte emulatorul.

2. Se încarcă programul numit lab4_prog2.asm, utilizând butonul Open. Fereastra Source Code trebuie să afişeze următorul program:

```
                org 100h
                jmp main

numbers         dw 0172h, -218, 2B0h, 16h, -102

main:           push offset numbers
                push (main-numbers)/2
                call getMax
                int 20h

getMax:         mov BP, SP
                mov CX, [BP+2]
                mov BX, [BP+4]
                mov AX, [BX]
                mov DI, BX

findLoop:       cmp AX, [BX]
                jl changeMax
nextElem:       add BX, 2h
                loop findLoop
                ret

changeMax:      mov AX, [BX]
                mov DI, BX
                jmp nextElem
```

3. Se înţelege programul

4. Se compilează programul.

5. Se încarcă programul executabil în emulator.

6. Se execută programul pas cu pas, se observă schimbările produse asupra registrelor, locaţiilor de memorie, fanioanelor etc şi se notează observaţii.

7. Se scriu concluzii privitoare la efectul diferitelor instrucţiuni asupra registrelor, fanioanelor şi locaţiilor de memorie.

4.3.3 Exerciţiul 3

Obiectiv: Înţelegerea modului de transmitere/extragere a parametrilor de intrare/ieşire din subprograme prin intermediul stivei/registrelor.

Cerinţă: Folosiţi subprogramul prezentat în exemplul precedent pentru a crea un program care sortează descrescător un şir de numere pe 16 biţi cu semn.

Soluţie:

Notă: Rezolvarea acestui exerciţiu implică scrierea unui program care apelează în mod succesiv subprogramul din exerciţiul anterior, găsind la fiecare iteraţie maximul dintr-un şir din ce în ce mai scurt şi interschimbând acest maxim cu primul element al şirului. Cu alte cuvinte, subprogramul va fi apelat prima oară folosind adresa de început a şirului şi numărul de elemente = 5. După întoarcerea în programul principal, maximul identificat va fi interschimbat cu primul element al şirului. În continuare, subprogramul va fi apelat din nou, folosind ca adresă de început adresa celui de-al doilea element şi numărul de elemente = 5 – 1 = 4.

1. Porniţi emulatorul.

2. Se încarcă programul numit lab4_prog3.asm, utilizând butonul Open. Fereastra Source Code trebuie să afişeze următorul program:

```
                org 100h
                jmp main

numbers         dw 0172h, -218h, 2B0h, 16h, -102h

main:           lea BX, numbers
                mov CX, (main-numbers)/2

sortLoop:       push BX
                push CX
                call getMax
                pop CX
                pop BX
                xchg [BX], AX
                xchg AX, [DI]
                add BX, 2h
                loop sortLoop

                int 20h

getMax:         mov BP, SP
                mov CX, [BP+2]
                mov BX, [BP+4]
                mov AX, [BX]
                mov DI, BX

findLoop:       cmp AX, [BX]
                jl changeMax
nextElem:       add BX, 2h
                loop findLoop
                ret

changeMax:      mov AX, [BX]
                mov DI, BX
                jmp nextElem
```

3. Se înţelege programul

4. Se compilează programul.

5. Se încarcă programul executabil în emulator.

6. Se execută programul pas cu pas, se observă schimbările produse asupra registrelor, locaţiilor de memorie, fanioanelor etc şi se notează observaţii.

7. Se scriu concluzii privitoare la efectul diferitelor instrucţiuni asupra registrelor, fanioanelor şi locaţiilor de memorie.

4.4. Anexa 1. Exemple pentru instrucţiunile CALL şi RET

CALL – Apelare Subprogram intra-segment

Mod de utilizare: CALL d

Argumente:

- **d** (ţintă) – adresa primei instrucţiuni din programul apelat; poate să fie o valoare imediată, un registru de uz general sau o locaţie de memorie;

Efecte: Adresa următoarei instrucţiuni este salvată în stivă şi **IP** este setat la adresa ţintă (UCP execută un salt la subprogram):

(SP) ← (SP) – 2

((SS)↕0H+(SP)+1) ↕ ((SS)↕0H+(SP)) ← (IP)

(IP) ← (d)

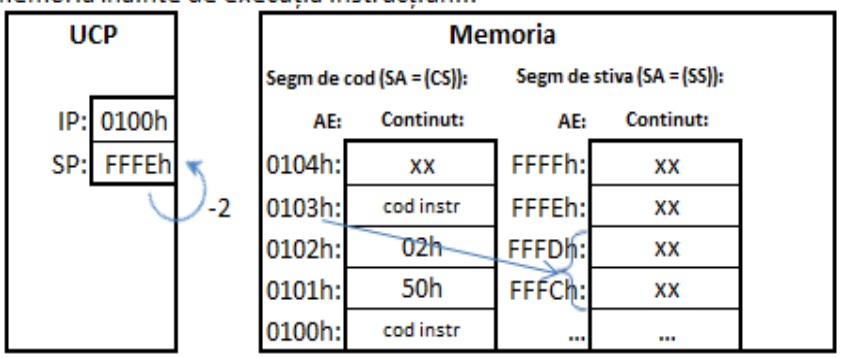
Fanioane: niciunul.

Notă: De obicei există o instrucţiune RET în subprogram pentru a se întoarce la următoarea instrucţiune după CALL.

Exemplu

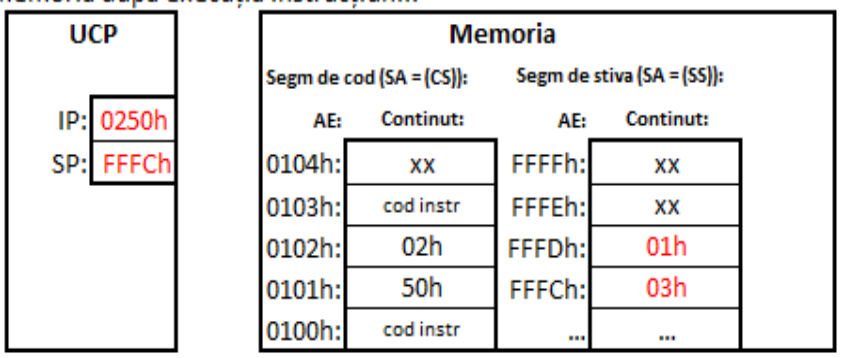
CALL 0250h

1.UCP și memoria înainte de execuția instrucțiunii:



2.UCP stochează adresa următoarei instrucțiuni (0103h) în stivă (pentru aceasta, SP trebuie să fie decrementat cu 2 unități) și sare la adresa trimisă ca operand imediat (stochează în IP valoarea găsită în formatul instrucțiunii imediat după codul instrucțiunii)

3.UCP și memoria după execuția instrucțiunii:



RET – Întoarcere din Subprogram

Mod de utilizare: RET

Argumente: niciunul.

Efecte: UCP extrage ultima valoare din stivă şi o foloseşte pentru a se întoarce la programul apelant:

(IP) ← ((SS)↕0H+(SP)+1) ↕ ((SS)↕0H+(SP))

(SP) ← (SP) + 2.

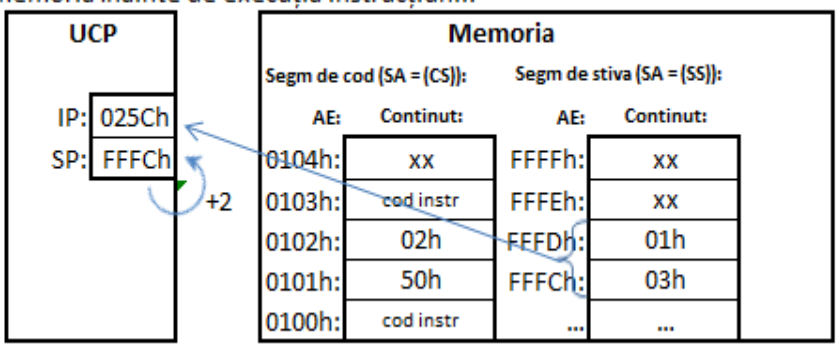
Fanioane: niciunul.

Notă: De obicei adresa a fost plasată în stivă de către o instrucţiune CALL şi întoarcerea în programul principal se face la adresa ce urmează după instrucţiunea CALL.

Exemplu

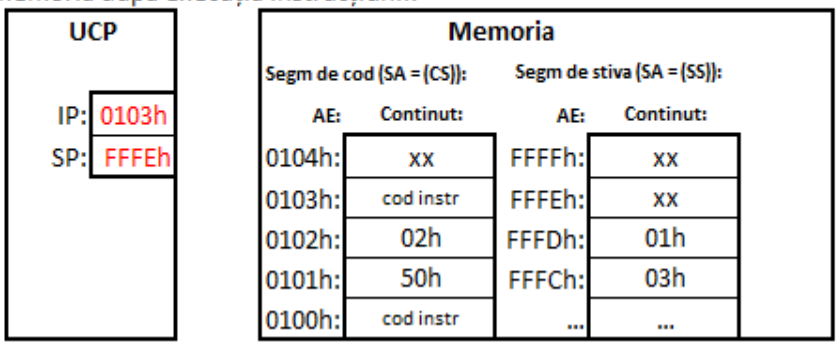
RET

1.UCP și memoria înainte de execuția instrucțiunii:



2.UCP extrage valoarea elementului din vârful stivei (SP este incrementat cu 2 unități) și o folosește pentru a executa un salt necondiționat la programul apelant (stochează această valoare în IP). Valorile din segmentul de stivă rămân neschimbate.

3.UCP și memoria după execuția instrucțiunii:



5. RECAPITULARE ŞI EXERCIŢII

5.1 Exerciţii

5.1.1 Exerciţiul 1

Cerinţă: Scrieţi un subprogram care să primească două numere de 16 biţi, fără semn, ca parametri de intrare (transmişi prin intermediul stivei) şi să returneze maximul celor două numere (prin intermediul registrului AX). Exemplificaţi modul de utilizare al subprogramului într-un program care compară trei numere de 16 biţi, fără semn definite în program ca variabile de tip word (şi denumite alpha, beta şi gamma) în vederea găsirii maximului lor.

Indicaţie: Programul principal va apela subprogramul de două ori: prima dată va compara numerele alpha şi beta, apoi va compara maximul dintre alpha şi beta cu numărul gamma.

5.1.2 Exerciţiul 2

Cerinţă: Scrieţi un subprogram care să primească patru numere de 16 biţi, fără semn, ca parametri de intrare (transmişi prin intermediul stivei) şi să returneze rezultatul următoarei funcţii aritmetice (în registrul DX concatenat cu registrul AX):

(parametrul1 – parametrul2) * (parametrul3 – parametrul4)

Exemplificaţi modul de utilizare al subprogramului într-un program care-l apelează, transmiţând ca parametri de intrare patru numere oarecare, şi stochează rezultatul în memorie într-o variabilă denumită result.

5.1.3 Exerciţiul 3

Cerinţă: Scrieţi un subprogram care să primească patru numere de 8 biţi, cu semn, ca parametri de intrare (transmişi prin intermediul stivei) şi să returneze rezultatul următoarei funcţii logice (în registrul AX):

NOT ((parametrul1 << 2 XOR parametrul2) AND (parametrul3 << 2 XOR parametrul4))

Exemplificaţi modul de utilizare al subprogramului într-un program care-l apelează, transmiţând ca parametri de intrare patru numere oarecare, şi stochează rezultatul în memorie într-o variabilă denumită result.

Notă: XOR, AND, NOT sunt operatorii logici uzuali, iar << este operatorul de deplasare la stânga.

5.1.4 Exerciţiul 4

Cerinţă: Scrieţi un subprogram care să primească ca parametri de intrare adresa de început şi numărul de elemente pentru un şir de numere de 16 biţi, fără semn, şi să returneze suma elementelor şirului (prin intermediul registrelor AX şi DX). Exemplificaţi modul de utilizare al subprogramului într-un program care să-l apeleze o singură dată.

Notă: Pentru a efectua şi returna suma elementelor şirului trebuie să se folosească două registre (AX şi DX) pentru că rezultatul poate fi un număr pe 32 de biţi.

5.1.5 Exerciţiul 5

Cerinţă: Scrieţi un subprogram care să primească ca parametri de intrare adresa de început şi numărul de elemente pentru un şir de caractere ce conţine numai litere şi returneze numărul de litere mari (prin intermediul registrului DX). Exemplificaţi modul de utilizare al subprogramului într-un program care să-l apeleze o singură dată.

Notă: Pentru prelucrarea elementelor şirului este obligatorie copierea elementelor în acumulator utilizând instrucţiunea de transfer de date pentru şiruri lods.

Indicaţie: Subprogramul va verifica dacă elementul curent al şirului este literă mare comparând codul ASCII al elementului curent cu codurile ASCII ale literelor “A” şi “Z”.

5.1.6 Exerciţiul 6

Cerinţă: Scrieţi un subprogram care să primească ca parametri de intrare adresa de început şi numărul de elemente pentru un şir de caractere ce conţine numai litere (denumit în continuare şir sursă), cât şi adresa de început a unui al doilea şir (denumit în continuare şir destinaţie). Subprogramul trebuie să copieze în şirul destinaţie numai literele mici din şirul sursă şi să

returneze numărul de elemente copiate (prin intermediul registrului DX). Exemplificaţi modul de utilizare al subprogramului într-un program care să-l apeleze o singură dată.

Notă: Pentru transferul elementelor şirurilor este obligatorie utilizarea instrucţiunilor de transfer de date pentru şiruri lods şi stos.

Indicaţie: Subprogramul va verifica dacă elementul curent al şirului este literă mică comparând codul ASCII al elementului curent cu codurile ASCII ale literelor “a” şi “z”.

6. ANEXĂ. SETUL DE INSTRUCȚIUNI

Procesoare compatibile Intel x86 – variantele pe 16 biţi

Convenţii:

s: sursă;

d: destinaţie;

AL|AX: acumulatorul implicit de 8 sau de 16 biţi;

mem: conţinutul unei locaţii de memorie sau conţinutul a două locaţii de memorie succesive, adresate cu unul dintre modurile de adresare permise pentru memoria de date, cu excepţia adresării imediate;

mem16: conţinutul a două locaţii de memorie succesive adresate cu unul dintre modurile de adresare permise pentru memoria de date, cu excepţia adresării imediate;

mem32: conţinutul a patru locaţii de memorie succesive adresate cu unul dintre modurile de adresare permise pentru memoria de date, cu excepţia adresării imediate;

r | r_i | r_j: un registru oarecare de 8 sau de 16 biţi, exceptând registrele segment;

r8: un registru de 8 biţi;

r16: un registru de 16 biţi, exceptând registrele segment;

rs: un registru segment (**CS, SS, DS, ES**);

data: un operand de 8 sau 16 biţi care face parte din formatul instrucţiunii (adresare imediată);

data8: un operand de 8 biţi care face parte din formatul instrucţiunii (adresare imediată);

data16: un operand de 16 biţi care face parte din formatul instrucţiunii (adresare imediată);

disp8: deplasament pe 8 biţi (face parte din formatul instrucţiunii);

disp16: deplasament pe 16 biţi (face parte din formatul instrucţiunii);

adr: o adresă completă (pe 16 biţi);

adr8: o adresă scurtă (pe 8 biţi);

adr32: o adresă logică exprimată pe patru octeţi succesivi;

port: adresa (numărul de ordine) unui port de intrare/ieşire, de regulă pe 8 biţi;

tip: un operand de 8 biţi care indică tipul unei întreruperi şi face parte din formatul instrucţiunii (adresare imediată);

nrcel: numărul de celule cu care se poate face deplasarea sau rotaţia unui operand;

AE: adresa efectivă.

Pentru fanioane:

x : fanionul se schimbă în conformitate cu rezultatul operaţiunii;

1 : fanionul este setat necondiţionat;

0 : fanionul este resetat necondiţionat;

? : fanionul este afectat impredictibil;

blanc: fanionul nu este afectat.

Pentru calculul numărului de stări :

cAE - timpul de calcul al adresei efective, şi anume

- adresare directă: **AE=disp8|disp16** 6 stări;
- adresare indexată: **AE=(SI)|(DI)+ disp8|disp16** 9 stări;
- adresare indirectă implicită: **AE=(SI)|(DI)** 5 stări;
- adresare relativă la bază directă, fără deplasament:

AE=(BX) 5 stări;

- adresare relativă la bază directă, cu deplasament:

AE=(BX)+disp8|disp16 9 stări;

- adresare relativă la bază indexată:

AE=(BX)+(SI)|(DI)+disp8|disp16 12 stări;

- adresare relativă la bază implicită: **AE=(BX)+(SI)|(DI)** 8 stări;
- adresare în stivă directă, fără deplasament: **AE=(BP)** 5 stări;
- adresare în stivă directă, cu deplasament:

  AE=(BP)+disp8|disp16 9 stări;
- adresare în stivă indexată:

  AE=(BP)+(SI)|(DI)+disp8|disp16 12 stări;
- adresare în stivă implicită: **AE=(BP)+(SI)|(DI)** 8 stări;
- pentru redirecţionarea segmentului se mai adaugă 2 stări.

AAA	Ajustare ASCII pentru adunare	OF	DF	IF	TF	SF	ZF	AF	PF	CF
		?				**?**	**?**	**x**	**?**	**x**

Operanzi	Nr. de stări	Octeţi	Exemple
	4	1	AAA

AAD	Ajustare ASCII pentru împărţire (se face înainte de împărţire)	OF	DF	IF	TF	SF	ZF	AF	PF	CF
		?				**x**	**x**	**?**	**x**	**?**

Operanzi	Nr. de stări	Octeţi	Exemple
	60	2	AAD

AAM	Ajustare ASCII pentru înmulţire (se face după înmulţire)	OF	DF	IF	TF	SF	ZF	AF	PF	CF
		?				**x**	**x**	**?**	**x**	**?**

Operanzi	Nr. de stări	Octeţi	Exemple
	83	1	AAM

AAS	Ajustare ASCII pentru scădere	OF	DF	IF	TF	SF	ZF	AF	PF	CF
		?				**?**	**?**	**x**	**?**	**x**

Operanzi	Nr. de stări	Octeţi	Exemple
	4	1	AAS

ADC d,s	Adunare cu transport	OF	DF	IF	TF	SF	ZF	AF	PF	CF
		x				**x**	**x**	**x**	**x**	**x**

Operanzi	Nr. de stări	Octeţi	Exemple
AL|AX, data	4	2-3	ADC AX,9D81H
r, data	4	3-4	ADC CL,36H
mem, data	17+cAE	3-6	ADC [SI],2D31H
r_1, r_2	3	2	ADC BX,SI
r, mem	9+cAE	2-4	ADC AX,[BX]
mem, r	16+cAE	2-4	ADC [BX+SI+64H],DI

ADD d,s	Adunare	OF DF IF TF SF ZF AF PF CF **x** **x** **x** **x** **x**

Operanzi	Nr. de stări	Octeţi	Exemple
AL|AX, data	4	2-3	ADD AX,0F0FH
r, data	4	3-4	ADD DX,0F0F0H
mem, data	17+cAE	3-6	ADD [100H],2H
r_1, r_2	3	2	ADD SI,CX
r, mem	9+cAE	2-4	ADD DI,[BX+22H]
mem, r	16+cAE	2-4	ADD [BX+SI],CL

AND d,s	ŞI logic	OF DF IF TF SF ZF AF PF CF **0** **x** **x** **?** **x** **0**

Operanzi	Nr. de stări	Octeţi	Exemple
AL|AX, data	4	2-3	AND AL,7FH
r, data	4	3-4	AND CX,0F0H
mem, data	17+cAE	3-6	AND [BX],01010000B
r_1, r_2	3	2	AND AL,BL
r, mem	9+cAE	2-4	AND DL,[BX+SI]
mem, r	16+cAE	2-4	AND [DI+10H],AL

CALL adr	Apelarea unui subprogram	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
adr32	28	5	CALL PROC-ALT-SEG
disp16	28	3	CALL SUMA
r16	16	2	CALL BX
mem*	21+cAE	2-4	CALL [BX]
mem**	37+cAE	2-4	CALL [SI]

* apelare de subprogram cu adresare indirectă definită cu directiva de asamblare ca apelare intra-segment;
** apelare de subprogram cu adresare indirectă definită cu directiva de asamblare ca apelare inter-segment.

CBW	Extindere (cu semn) a unui octet la un cuvânt	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
	2	1	CBW

CLC	Resetarea fanionului de transport	OF DF IF TF SF ZF AF PF CF **0**

Operanzi	Nr. de stări	Octeţi	Exemple
	2	1	CLC

CLD	Resetarea fanionului de direcţie	OF DF IF TF SF ZF AF PF CF **0**

Operanzi	Nr. de stări	Octeţi	Exemple
	2	1	CLD

CLI	Resetarea fanionului de validare a întreruperilor	OF DF IF TF SF ZF AF PF CF **0**

Operanzi	Nr. de stări	Octeţi	Exemple
	2	1	CLI

CMC	Complementarea fanionului de transport	OF DF IF TF SF ZF AF PF CF **x**

Operanzi	Nr. de stări	Octeţi	Exemple
	2	1	CMC

CMP s1,s2	Compararea a doi operanzi	OF DF IF TF SF ZF AF PF CF **x x x x x x**

Operanzi	Nr. de stări	Octeţi	Exemple
AL|AX, data	4	2-3	CMP AL,0DH
r, data	4	3-4	CMP SI,200H
mem, data	17+cAE	3-6	CMP [BX+DI+55H],2222H
r_1, r_2	3	2	CMP CL,DH
r, mem	9+cAE	2-4	CMP DH,[4321H]
mem, r	16+cAE	2-4	CMP [BP+2],SI

CMPS	Compararea componentelor din două şiruri	OF DF IF TF SF ZF AF PF CF **x x x x x x**

Operanzi	Nr. de stări	Octeţi	Exemple
	22	1	CMPSB ; pe octeţi
	22	1	CMPSW ; pe cuvinte

Obs.: Primitiva CMPS poate fi însoţită de unul dintre prefixele de repetabilitate **REP**, **REPE**|**REPZ** sau **REPNE**|**REPNZ**; acestea adaugă 9 stări.

CWD	Extindere (cu semn) a unui cuvânt la un cuvânt dublu	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
	5	1	CWD

DAA	Ajustare zecimală pentru adunare	OF	DF	IF	TF	SF	ZF	AF	PF	CF
		?				**x**	**x**	**x**	**x**	**x**

Operanzi	Nr. de stări	Octeţi	Exemple
	4	1	DAA

DAS	Ajustare zecimală pentru scădere	OF	DF	IF	TF	SF	ZF	AF	PF	CF
		?				**x**	**x**	**x**	**x**	**x**

Operanzi	Nr. de stări	Octeţi	Exemple
	4	1	DAS

DEC s	Decrementarea sursei	OF	DF	IF	TF	SF	ZF	AF	PF	CF
		x				**x**	**x**	**x**	**x**	

Operanzi	Nr. de stări	Octeţi	Exemple
r16	2	1	DEC CX
r8	3	2	DEC BH
mem	15+cAE	2-4	DEC [SI+20H]

DIV s	Împărţire	OF	DF	IF	TF	SF	ZF	AF	PF	CF
		?				**?**	**?**	**?**	**?**	**?**

Operanzi	Nr. de stări	Octeţi	Exemple
r8	80÷90	2	DIV BL
r16	144÷162	2	DIV CX
mem8	86÷96+ cAE	2-4	DIV [400H]
mem16	150÷168+ cAE	2-4	DIV [SI+10H]

ESC s	Plasează conţinutul unei locaţii de memorie pe magistrala de date	OF	DF	IF	TF	SF	ZF	AF	PF	CF

Operanzi	Nr. de stări	Octeţi	Exemple
mem	7 + cAE	2-4	ESC [BX+SI]

HLT	Introduce procesorul în starea de „halt”	OF	DF	IF	TF	SF	ZF	AF	PF	CF

Operanzi	Nr. de stări	Octeţi	Exemple
	2	1	HLT

IDIV s	Împărţire cu semn	OF DF IF TF SF ZF AF PF CF **?** **?** **?** **?** **?**

Operanzi	Nr. de stări	Octeţi	Exemple
r8	101÷112	2	IDIV CL
r16	165÷184	2	IDIV BX
mem8	107÷118+cAE	2-4	IDIV [SI+100H]
mem16	171÷190+cAE	2-4	IDIV [BX+1000H]

IMUL s	Înmulţire cu semn	OF DF IF TF SF ZF AF PF CF **x** **?** **?** **?** **?** **x**

Operanzi	Nr. de stări	Octeţi	Exemple
r8	80÷98	2	IMUL CL
r16	128÷154	2	IMUL BX
mem8	86÷104+cAE	2-4	IMUL [BX]
mem16	134÷160+cAE	2-4	IMUL [BP+DI+30H]

IN d,s	Transferul unui octet sau cuvânt de la un port în acumulator	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
AL|AX, port	10	2	IN AL,1FH
AL|AX, DX	8	1	IN AX,DX

INC s	Incrementarea sursei	OF DF IF TF SF ZF AF PF CF **x** **x** **x** **x** **x**

Operanzi	Nr. de stări	Octeţi	Exemple
r16	2	1	INC SI
r8	3	2	INC BL
mem	15+cAE	2-4	INC [BX+SI]

INT [tip]	Cerere de întrerupere software	OF DF IF TF SF ZF AF PF CF **0** **0**

Operanzi	Nr. de stări	Octeţi	Exemple
	52	1	INT ; întrerupere tip 3
tip (tip<>3)	51	2	INT 67

INTO	Cerere de întrerupere de tip 4 (dacă există „depăşire”)		OF DF IF TF SF ZF AF PF CF 0 0
Operanzi	Nr. de stări	Octeţi	Exemple
	53 sau 4	1	INTO

IRET	Întoarcere din rutina de deservire a unei întreruperi		OF DF IF TF SF ZF AF PF CF x x x x x x x x x
Operanzi	Nr. de stări	Octeţi	Exemple
	24	1	IRET

JA | JNBE disp8	Salt dacă „peste” | dacă „nu sub sau egal” (pentru operaţii fără semn)		OF DF IF TF SF ZF AF PF CF
Operanzi	Nr. de stări	Octeţi	Exemple
disp8	16 sau 4	2	JA ET1 ;salt dacă (CF)=0 sau (ZF)=0

JAE|JNB|JNC disp8	Salt dacă „peste sau egal” | Dacă „nu sub” | dacă „nu transport” (pentru operaţii fără semn)		OF DF IF TF SF ZF AF PF CF
Operanzi	Nr. de stări	Octeţi	Exemple
disp8	16 sau 4	2	JAE ET2 ;salt dacă (CF)=0

JB|JNAE|JC disp8	Salt dacă „sub” | dacă „nu peste sau egal” | dacă „există transport” (pentru operaţii fără semn)		OF DF IF TF SF ZF AF PF CF
Operanzi	Nr. de stări	Octeţi	Exemple
disp8	16 sau 4	2	JB ET3 ;salt dacă (CF)=1

JBE | JNA disp8	Salt dacă „sub sau egal” | dacă „nu peste” (pentru operaţii fără semn)		OF DF IF TF SF ZF AF PF CF
Operanzi	Nr. de stări	Octeţi	Exemple
disp8	16 sau 4	2	JNA ET4 ;salt dacă (CF)=1 sau (ZF)=1

JCXZ disp8	Salt dacă (CX) = 0		OF DF IF TF SF ZF AF PF CF
Operanzi	Nr. de stări	Octeţi	Exemple
disp8	16 sau 4	2	JCXZ GATA

JE | JZ disp8	Salt dacă „egal” | dacă " zero" (pentru operaţii cu semn)	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
disp8	16 sau 4	2	JZ ET5 ;salt dacă (ZF)=1

JG | JNLE disp8	Salt dacă „mai mare” | dacă „nu mai mic sau egal” (pentru operaţii cu semn)	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
disp8	16 sau 4	2	JG ET6 ;salt dacă ;(SF)⊗(OF)=0 sau (ZF)=0

JGE | JNL disp8	Salt dacă „mai mare sau egal” | dacă „nu mai mic” (pentru operaţii cu semn)	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
disp8	16 sau 4	2	JGE ET7 ;salt dacă ;(SF)⊗(OF)=0

JL | JNGE disp8	Salt dacă „mai mic” | dacă „nu mai mare sau egal” (pentru operaţii cu semn)	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
disp8	16 sau 4	2	JL ET8 ;salt dacă ;(SF)⊗(OF)=1

JLE | JNG disp8	Salt dacă „mai mic sau egal” | dacă„nu mai mare” (pentru operaţii cu semn)	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
disp8	16 sau 4	2	JNG ET9 ;salt dacă ;(SF)⊗(OF)=1 sau (ZF)=1

JMP adr	Salt propriu-zis, necondiţionat	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
adr32	15	5	JMP ET_ALT_SEG
disp16	15	3	JMP ETICHETA_IN_SEGMENT
disp8	15	2	JMP ET_SALT_SCURT
r16	11	2	JMP BX
mem*	18+cAE	2-4	JMP [BX+100H]
mem**	24+cAE	2-4	JMP [DI]

* salt cu adresare indirectă definit cu directivă de asamblare ca salt intra-segment;

** salt cu adresare indirectă definit cu directivă de asamblare ca salt inter-segment.

JNE | JNZ disp8	Salt dacă „ne-egal” | dacă „non-zero” (operaţii cu semn)	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
disp8	16 sau 4	2	JNE ET10 ;salt dacă (ZF)=0

JNO disp8	Salt dacă „nu există depăşire"	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
disp8	16 sau 4	2	JNO ET11 ;salt dacă (OF)=0

JNP | JPO disp8	Salt dacă „non-paritate” | dacă „impar”	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
disp8	16 sau 4	2	JPO ET12 ;salt dacă (PF)=0

JNS disp8	Salt dacă „non-semn”	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
disp8	16 sau 4	2	JNS ET13 ;salt dacă (SF)=0

JO disp8	Salt dacă „depăşire”	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
disp8	16 sau 4	2	JO ET14 ;salt dacă (OF)=1

JP | JPE disp8	Salt dacă „există paritate” | dacă „par”	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
disp8	16 sau 4	2	JPO ET15 ;salt dacă (PF)=1

JS disp8	Salt dacă „există semn”	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
disp8	16 sau 4	2	JS ET16 ;salt dacă (SF)=1

LAHF	Încarcă **(AH)** cu octetul inferior al registrului **F**	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
	4	1	LAHF

LDS d,s	Încarcă un registru de 16 biţi şi registrul segment de date	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
r16, mem32	16+cAE	2-4	LDS SI,[10H]

LEA d,s	Încarcă un registru de 16 biţi cu o adresă efectivă	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
r16, mem16	2+cAE	2-4	LEA BX,[BX+SI+0FFFH]

LES d,s	Încarcă un registru de 16 biţi şi registrul **ES**	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
r16, mem32	16+cAE	2-4	LES DI,[BX]

LOCK	Pe durata instrucţiunii pe care o prefixează alocă magistrala calculatorului	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
mem	2	1	LOCK XCHG CL,BL

LODS	Încarcă componentele unui şir în acumulator	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
	12	1	LODSB ; pe octeţi
	12	1	LODSW ; pe cuvinte

LOOP disp8	Ciclează necondiţionat	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
disp8	9 sau 5	2	LOOP CICLU_1

LOOPE|LOOPZ disp8	Ciclează cât timp „egal” sau „zero”	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
disp8	11 sau 5	2	LOOPE CICLU_2

LOOPNE | LOOPNZ disp8	Ciclează cât timp „ne-egal” sau „non-zero”	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
disp8	11 sau 5	2	LOOPNE CICLU_3

MOV d,s	Copiază sursa la destinaţie	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
r, data	4	2-3	MOV CX,3168H
mem, data	10+cAE	3-6	MOV [BX],491FH
AL|AX, mem	10	3	MOV AL,[1064H]
mem, AL|AX	10	3	MOV [1064H],AX
r_1, r_2	2	2	MOV AX,CX
r, mem	8+cAE	2-4	MOV BP,[BX+DELTA]
mem, r	9+cAE	2-4	MOV [DI+100H],CX
rs*, r16	2	2	MOV SS,DX
rs*, mem16	8+cAE	2-4	MOV DS,[1000H]
r16, rs	2	2	MOV BP,SS
mem16, rs	9+cAE	2-4	MOV [2000H],CS

* registrul segment nu poate fi **CS**

MOVS	Transferă o componentă a şirului sursă în şirul destinaţie	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
	18	1	MOVSB ; pe octeţi
	18	1	MOVSW ; pe cuvinte

Obs.: Primitiva **MOVS** poate fi însoţită de prefixul de repetabilitate **REP**; acesta adaugă 9 stări.

MUL s	Înmulţire	OF DF IF TF SF ZF AF PF CF **x ? ? ? ? x**

Operanzi	Nr. de stări	Octeţi	Exemple
r8	70÷77	2	MUL BL
r16	118÷133	2	MUL CX
mem8	76÷83+cAE	2-4	MUL [SI+300H]
mem16	124÷139+cAE	2-4	MUL [200H]

NEG s	Complementare faţă de 2 a sursei	OF DF IF TF SF ZF AF PF CF **x** **x** **x** **x** **1***

Operanzi	Nr. de stări	Octeţi	Exemple
r	3	2	NEG AL
mem	16+cAE	2-4	NEG [BX]

* **(CF) = 0** dacă **(s)=0H**

NOT s	Complementare faţă de 1 a sursei	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
r	3	2	NOT BL
mem	16+cAE	2-4	NOT [1000H]

OR d,s	SAU logic	OF DF IF TF SF ZF AF PF CF **0** **x** **x** **?** **x** **0**

Operanzi	Nr. de stări	Octeţi	Exemple
AL|AX, data	4	2-3	OR AX,3030H
r, data	4	3-4	OR CX,001FH
mem, data	17+cAE	3-6	OR [BX+DI],1F1FH
r_1, r_2	3	2	OR AH,BH
r, mem	9+cAE	2-4	OR DX,[DI]
mem, r	16+cAE	2-4	OR [BX],AX

OUT d,s	Transferul unui octet sau cuvânt din acumulator la un port	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
port, AL|AX	10	2	OUT 44,AX
DX, AL|AX	8	1	OUT DX,AL

POP d	Transferă din stivă la destinaţie	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
r16	8	1	POP BX
mem16	17+cAE	2-4	POP [SI]
rs (nu CS)	8	1	POP ES

POPF	Transferă din stivă registrul de fanioane	OF DF IF TF SF ZF AF PF CF **x** **x** **x** **x** **x** **x** **x** **x** **x**

Operanzi	Nr. de stări	Octeţi	Exemple
	8	1	POPF

PUSH s	Transferă în stivă de la sursă	OF DF IF TF SF ZF AF PF CF	
Operanzi	Nr. de stări	Octeţi	Exemple
r16	11	1	PUSH SI
mem16	16+cAE	2-4	PUSH [BX]
rs	10	1	PUSH DS

PUSHF	Transferă în stivă registrul de fanioane	OF DF IF TF SF ZF AF PF CF	
Operanzi	Nr. de stări	Octeţi	Exemple
	10	1	PUSHF

RCL s, nrcel	Rotaţie stânga cu transport	OF DF IF TF SF ZF AF PF CF **x** **x**	
Operanzi	Nr. de stări	Octeţi	Exemple
r, 1	2	2	RCL AX,1
r, CL	8+4/bit	2	RCL BL,CL
mem, 1	15+cAE	2-4	RCL ALFA,1
mem, CL	20+cAE +4/bit	2-4	RCL [DI+ALFA],CL

RCR s, nrcel	Rotaţie dreapta cu transport	OF DF IF TF SF ZF AF PF CF **x** **x**	
Operanzi	Nr. de stări	Octeţi	Exemple
r, 1	2	2	RCR BX,1
r, CL	8+4/bit	2	RCR CX,CL
mem, 1	15+cAE	2-4	RCR [BX+20],1
mem, CL	20+cAE +4/bit	2-4	RCR [SI],CL

REP	Repetă necondiţionat primitiva de operaţie cu şiruri pe care o precede	OF DF IF TF SF ZF AF PF CF	
Operanzi	Nr. de stări	Octeţi	Exemple
	9	1	REP MOVSB

REPE | REPZ	Repetă cât timp „egal” | cât timp „zero” primitiva **CMPS** sau **SCAS**	OF DF IF TF SF ZF AF PF CF	
Operanzi	Nr. de stări	Octeţi	Exemple
	9	1	REPE CMPSB

REPNE|REPNZ	Repetă cât timp „ne-egal” | cât timp „non-zero” primitiva **CMPS** sau **SCAS**	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
	9	1	REPNE SCASW

RET [data16]	Reîntoarcere din subprogram	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
	8	1	RET ;intra-segment
data16	12	3	RET 4 ;intra-segment
	18	1	RET ;inter-segment
data16	17	3	RET 2 ;inter-segment

ROL s, nrcel	Rotaţie stânga	OF DF IF TF SF ZF AF PF CF **x** **x**

Operanzi	Nr. de stări	Octeţi	Exemple
r, 1	2	2	ROL DI,1
r, CL	8+4/bit	2	ROL BX,CL
mem, 1	15+cAE	2-4	ROL [DI+DELTA],1
mem, CL	20+cAE +4/bit	2-4	ROL [BX+DI],CL

ROR s, nrcel	Rotaţie dreapta	OF DF IF TF SF ZF AF PF CF **x** **x**

Operanzi	Nr. de stări	Octeţi	Exemple
r, 1	2	2	ROR CX,1
r, CL	8+4/bit	2	ROR BX,CL
mem, 1	15+cAE	2-4	ROR [SI],1
mem, CL	20+cAE +4/bit	2-4	ROR BETA,CL

SAHF	Încarcă octetul inferior al registrului **F** cu **(AH)**	OF DF IF TF SF ZF AF PF CF **x** **x** **x** **x** **x**

Operanzi	Nr. de stări	Octeţi	Exemple
	4	1	SAHF

SAL | SHL s,nrcel	Deplasare stânga logică sau aritmetică	OF	DF	IF	TF	SF	ZF	AF	PF	CF
		x				**x**	**x**	**?**	**x**	**x**

Operanzi	Nr. de stări	Octeţi	Exemple
r, 1	2	2	SHL AL,1
r, CL	8+4/bit	2	SAL SI,CL
mem, 1	15+cAE	2-4	SHL [DI+100H],1
mem, CL	20+cAE +4/bit	2-4	SHL [BP+SI],CL

SAR s, nrcel	Deplasare dreapta aritmetică	OF	DF	IF	TF	SF	ZF	AF	PF	CF
		x				**x**	**x**	**?**	**x**	**x**

Operanzi	Nr. de stări	Octeţi	Exemple
r, 1	2	2	SAR AH,1
r, CL	8+4/bit	2	SAR DI,CL
mem, 1	15+cAE	2-4	SAR [BP],1
mem, CL	20+cAE +4/bit	2-4	SAR [DI],CL

SBB d,s	Scădere cu împrumut	OF	DF	IF	TF	SF	ZF	AF	PF	CF
		x				**x**	**x**	**x**	**x**	**x**

Operanzi	Nr. de stări	Octeţi	Exemple
AL|AX, data	4	2-3	SBB AX,4D2CH
r, data	4	3-4	SBB CL,1
mem, data	17+cAE	3-6	SBB [BP+SI],3F8H
r_1, r_2	3	2	SBB BL,DL
r, mem	9+cAE	2-4	SBB DI,[BX+ALFA]
mem, r	16+cAE	2-4	SBB [SI+100],AX

SCAS	Compară elementele unui şir cu acumulatorul	OF	DF	IF	TF	SF	ZF	AF	PF	CF
		x				**x**	**x**	**x**	**x**	**x**

Operanzi	Nr. de stări	Octeţi	Exemple
	15	1	SCASB ; pe octeți
	15	1	SCASW ; pe cuvinte

Obs.: Primitiva **SCAS** poate fi însoţită de unul dintre prefixele de repetabilitate **REP**, **REPE**|**REPZ** sau **REPNE**|**REPNZ**; acestea adaugă 9 stări.

SHR s, nrcel	Deplasare dreapta logică	OF	DF	IF	TF	SF	ZF	AF	PF	CF
		x				**x**	**x**	**?**	**x**	**x**

Operanzi	Nr. de stări	Octeţi	Exemple
r, 1	2	2	SHR BL,1
r, CL	8+4/bit	2	SHR SI,CL
mem, 1	15+cAE	2-4	SHR [BP+SI],1
mem, CL	20+cAE +4/bit	2-4	SHR [BX+DI+BETA],CL

STC	Setarea fanionului de transport	OF DF IF TF SF ZF AF PF CF 1

Operanzi	Nr. de stări	Octeţi	Exemple
	2	1	STC

STD	Setarea fanionului de direcţie	OF DF IF TF SF ZF AF PF CF 1

Operanzi	Nr. de stări	Octeţi	Exemple
	2	1	STD

STI	Setarea fanionului de validare a întreruperilor	OF DF IF TF SF ZF AF PF CF 1

Operanzi	Nr. de stări	Octeţi	Exemple
	2	1	STI

STOS	Încarcă elementele unui şir din acumulator	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
	11	1	STOSB ; pe octeţi
	11	1	STOSW ; pe cuvinte

Obs.: Primitiva **STOS** poate fi însoţită de prefixul de repetabilitate **REP**; acesta adaugă 9 stări.

SUB d,s	Scădere	OF DF IF TF SF ZF AF PF CF x x x x x x

Operanzi	Nr. de stări	Octeţi	Exemple
AL|AX, data	4	2-3	SUB AL,65H
r, data	4	3-4	SUB SI,5280
mem, data	17+cAE	3-6	SUB [SI+14H],136H
r_1, r_2	3	2	SUB CX,BX
r, mem	9+cAE	2-4	SUB DH,[BP+4]
mem, r	16+cAE	2-4	SUB [BP+10],CL

TEST s1,s2	ŞI logic nedistructiv	OF DF IF TF SF ZF AF PF CF 0 x x ? x 0

Operanzi	Nr. de stări	Octeţi	Exemple
AL|AX, data	4	2-3	TEST AX,0040H
r, data	5	3-4	TEST SI,0050
mem, data	11+cAE	3-6	TEST [BP],00100000B
r_1, r_2	3	2	TEST SI,DI
r, mem	9+cAE	2-4	TEST AL,[55H]

WAIT	Introduce procesorul în starea de „wait”, până când semnalul **TEST=0**	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
		1	WAIT

XCHG d,s	Transferă sursa la destinaţie şi destinaţia la sursă	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
r16	3	1	XCHG BX ;XCHG AX este NOP
r_1, r_2	4	2	XCHG AL,BL
r, mem	17 + cAE	2-4	XCHG BX,[BP+SI]

XLAT	Translatează	OF DF IF TF SF ZF AF PF CF

Operanzi	Nr. de stări	Octeţi	Exemple
	11	1	XLAT

XOR d,s	SAU exclusiv	OF DF IF TF SF ZF AF PF CF **0 x x ? x 0**

Operanzi	Nr. de stări	Octeţi	Exemple
AL|AX, data	4	2-3	XOR AX,5522H
r, data	4	3-4	XOR SI,00C2H
mem, data	17+cAE	3-6	XOR [BX+DI],2244H
r_1, r_2	3	2	XOR CX,BX
r, mem	9+cAE	2-4	XOR AX,[SI]
mem, r	16+cAE	2-4	XOR [SI+ALFA],DX

7. BIBLIOGRAFIE

[1] C. Burileanu, *Arhitectura Microprocesoarelor*, Editura Denix, Bucureşti, 1994.

[2] *** *MCS-86 User's Manual*, Intel Corporation, 1978.

[3] R. Rector, G. Alexy, *The 8086 Book*, Osborne / McGraw Hill, Inc., Berkeley, Ca, 1980.

[4] S. Evanczuck, editor, *Microprocessor Systems. Software and Hardware Architecture*, McGraw Hill, Inc., 1984.

[5] *** *iAPX 86/88, 186/188 User's Manual. Programmer's Reference*, Intel Corporation, 1985.

[6] *** *iAPX 86/88, 186/188 User's Manual. Hardware Reference*, Intel Corporation, 1985.

[7] S. P. Morse, D. J. Albert, *The 80286 Architecture*, John Wiley & Sons, Inc., New York, 1986.

[8] *** *80286 and 80287 Programmer's Reference Manual*, Intel Corporation, 1987.

[9] *** *80286 Hardware Reference Manual*, Intel Corporation, 1987.

[10] *** *80286 Operating System Writer's Guide*, Intel Corporation, 1987.

[11] J. L. Hennessy, D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed. Morgan Kaufmann Publishers, Inc., San Francisco, Ca., 1996.

[12] J. H. Crawford, P. P. Gelsinger, *Programming the 80386*, Sybex, San Francisco, 1990.

[13] R. P. Nelson, *Microsoft's 80386/80486 Programming Guide*, Microsoft Press, Washington, 1991.

[14] *** *Microprocessors. Data Sheet*, vol I & II, Intel Corporation, 1992.

[15] *** *Pentium Processor User's Manual, Volume 3: Architecture and Programming Manual*, Intel Corp., Mt. Prospect, III., 1993.

[16] H. P. Messmer, *The Indispensable Pentium Book*, Addison-Wesley Publishers Ltd., Wokingham, England, 1995.

[17] M. Bekerman, A. Mendelson, A Performance Analysis of Pentium Processor Systems, IEEE Micro, Vol.15, Nr.5, October 1995, p.72-83.

[18] M. Slater, *The Microprocessor Today*, IEEE Micro, Vol.16, Nr.6, December 1995, p.32-44.

[19] B. Case, *x86 Has Plenty of Performance Headroom*, Microprocessor Report, Aug.22, 1994, p.9-14.

[20] C. Burileanu ş.a., *Arhitectura microprocesoarelor. Îndrumar de laborator*, Universitatea „POLITEHNICA" din Bucureşti, 1997.

[21] C. Burileanu, Mihaela Ioniţă, M. Ioniţă, M. Filotti, *Microprocesoarele x86 – o abordare software*, Editura Albastră, Cluj-Napoca, 1999.

[22] S. Mazor, *Intel's 8086*, IEEE Annals of the History of Computing, Vol. 32, Nr. 1, Ianuarie-Martie 2010, p.75-79.

[23] D. Evans, *x86 Assembly Guide*, CS216: Program and Data Representation, University of Virginia, 2006.

[24] *** *The 8086 Family User's Manual*, Intel Corporation, 1979.

[25] *** *Emu 8086*: http://emu8086.com