

Ciprian OPRIȘA Anca HANGAN Mădălin NEAGU

Emil CEBUC Gheorghe SEBESTYEN

**PROGRAMARE
ÎN
LIMBAJ DE ASAMBLARE**



**Editura UTPRESS
Cluj-Napoca, 2018
ISBN 978-606-737-333-2**

Introducere

Limbaajul de asamblare al unui calculator este forma simbolică de reprezentare a limbajului nativ al acelui calculator, limbaj numit “cod mașină”. Orice program scris pentru calculator indiferent de limbajul de programare folosit se va traduce în cod mașină în vederea executării. Din această perspectivă studiarea limbajului de asamblare al unui calculator ajută la înțelegerea mai profundă a modului în care un procesor reușește să interpreteze și să execute instrucțiunile date printr-un program. Nu întâmplător disciplina de Programare în limbaj de asamblare este studiată în majoritatea programelor de pregătire universitară din domeniul calculatoarelor. Cartea de față conține o serie de lucrări practice prin care autorii doresc să prezinte principiile programării la nivelul limbajului de asamblare dar și elementele specifice ale limbajului de asamblare ISA x86, limbaj utilizat în majoritatea calculatoarelor personale actuale. Prin denumirea generică ISA x86 (Instruction Set Architecture x86) s-a desemnat o arhitectură virtuală necesară pentru înțelegerea limbajului de asamblare al procesoarelor din familia Intel x86 și al procesoarelor cu arhitectură echivalentă (ex. AMD). Această arhitectură este doar o viziune simplificată a arhitecturii fizice a procesoarelor din familia x86, o arhitectură care în decursul timpului (în ultimele 3-4 decenii) a rămas în linii mari nemodificată. Menținerea acestei arhitecturi ISA x86 pe o perioadă atât de lungă a asigurat compatibilitatea generațiilor de procesoare Intel, la nivelul limbajului de asamblare. Modificările care s-au adus la arhitectură, dar și la setul de instrucțiuni ISA x86 au fost relativ reduse, ele fiind impuse de trecerea de la arhitectura inițială pe 16 biți (I8086) la arhitecturile pe 32 și respectiv 64 de biți. Lucrările din cadrul acestei culegeri se concentrează mai ales pe instrucțiunile setului de bază ISA x86, cele care au fost introduse la prima variantă de procesor din familia x86 și au fost menținute până astăzi. Acest îndrumător de laborator conține opt lucrări care acoperă setul de instrucțiuni de bază al Arhitecturii Intel pe 32 de biți și setul de instrucțiuni al coprocesorului matematic. Lucrările se adresează celor care învață programarea în limbaj de asamblare, la nivel începător. Fiecare lucrare conține exemple și probleme propuse cu diferite grade de dificultate, pentru exersarea folosirii diferitelor clase de instrucțiuni, a utilizării macrourilor și procedurilor. Îndrumătorul de față nu conține lucrări referitoare la seturi de instrucțiuni avansate cum ar fi MMX sau SSE.

În prima lucrare de laborator se prezintă modul în care sunt reprezentate în calculator diferitele tipuri de date: întregi cu sau fără semn și numere în virgulă mobilă. De asemenea, se prezintă caracteristicile specifice ale operațiilor aritmetice efectuate în diferite reprezentări.

A doua lucrare prezintă elementele de baza ale Arhitecturii ISA x86 și ale limbajului de asamblare. Sunt descrise structura sistemului de calcul, organizarea memoriei și arhitectura setului de instrucțiuni. Nu în ultimul rând, sunt prezentate uneltele care vor fi utilizate pe mai departe pentru scrierea și depanarea programelor în limbaj de asamblare. Următoarele trei lucrări prezintă principalele clase de instrucțiuni cu detalierea modului de funcționare al instrucțiunilor mai des utilizate. Se prezintă apoi modurile de adresare și modul în care se realizează controlul fluxului de instrucțiuni.

Utilizarea bibliotecilor de funcții din limbaj de asamblare și convențiile de apel sunt prezentate în lucrarea a șasea. Apoi, în a șaptea lucrare se arată cum se pot modulariza programele prin scrierea de macrouri și proceduri. Coprocesorul matematic x87 și setul său de instrucțiuni sunt prezentate în cea de-a opta lucrare. Îndrumătorul de laborator se încheie cu exemple de cod și lista instrucțiunilor uzuale ale limbajului de asamblare. Autorii speră că prin intermediul acestor lucrări cititorii vor avea o viziune mai profundă asupra modului în care calculatoarele actuale realizează execuția programelor. Aceasta

viziune va permite o programare mai eficienta din perspectiva timpului de execuție, chiar și în alte limbaje de programare.

Cuprins

1	Reprezentarea datelor în calculator	9
1.1	Scopul lucrării	9
1.2	Conversii și operații în diverse baze de numerație	9
1.2.1	Considerații teoretice	9
1.2.2	Conversia numerelor din baza 10 într-o bază oarecare	10
1.2.3	Conversia unui număr dintr-o bază oarecare în baza 10	12
1.2.4	Operații simple cu numere scrise în diverse baze	12
1.3	Reprezentarea internă a datelor	13
1.3.1	Reprezentarea numerelor întregi în Mărime și Semn (MS) în Complement față de 1 și de 2 (C1,C2)	13
1.3.2	Reprezentarea numerelor reale în format IEEE	15
1.3.3	Reprezentarea numerelor în BCD împachetat și despachetat	16
1.4	Întrebări recapitulative	17
1.5	Mersul lucrării	17
1.5.1	Conversii	17
1.5.2	Reprezentări	17
2	Arhitectura Intel x86 și elemente de bază ale limbajului de asamblare	19
2.1	Arhitectura Intel x86	19
2.1.1	Structura unui sistem de calcul (considerații generale)	19
2.1.2	Familia Intel x86	19
2.1.3	Arhitectura setului de instrucțiuni	20
2.1.4	Organizarea memoriei	21
2.2	Elementele de bază ale limbajului de asamblare	24
2.2.1	Construcții de bază	24
2.2.2	Structura generală a unui program MASM	25
2.3	Instrumentele de lucru pentru compilare și depanare	26
2.3.1	Asamblorul și linker-ul	26
2.3.2	Depanarea programelor	27
2.4	Întrebări recapitulative	29
2.5	Mersul lucrării	30
3	Setul de instrucțiuni al familiei de procesoare Intel x86: Sintaxa și clasele de instrucțiuni	33
3.1	Scopul lucrării	33
3.2	Prezentarea instrucțiunilor	33
3.2.1	Clase de instrucțiuni	34
3.3	Întrebări recapitulative	42
3.4	Mersul lucrării	43

3.4.1	Probleme rezolvate	43
3.4.2	Probleme propuse	43
4	Modurile de adresare ale procesorului Intel x86	45
4.1	Scopul lucrării	45
4.2	Prezentarea modurilor de adresare	45
4.2.1	Adresarea imediată	45
4.2.2	Adresarea de tip registru	46
4.2.3	Adresarea directă	46
4.2.4	Moduri de adresare indirectă	46
4.2.5	Adresarea pe șiruri	48
4.2.6	Adresarea de tip stivă	48
4.3	Întrebări recapitulative	49
4.4	Mersul lucrării	50
4.4.1	Probleme rezolvate	50
4.4.2	Probleme propuse	50
5	Setul de instrucțiuni al familiei de procesoare Intel x86: Controlul fluxului de instrucțiuni	51
5.1	Scopul lucrării	51
5.2	Considerații teoretice	51
5.2.1	Instrucțiuni de salt	51
5.2.2	Instrucțiuni de ciclare	54
5.2.3	Instrucțiuni pe șiruri	55
5.3	Întrebări recapitulative	56
5.4	Mersul lucrării	57
5.4.1	Probleme rezolvate	57
5.4.2	Probleme propuse	57
6	Utilizarea bibliotecilor de funcții	59
6.1	Scopul lucrării	59
6.2	Rolul sistemului de operare și al bibliotecilor de funcții	59
6.3	Utilizarea funcțiilor externe. Convenții de apel	59
6.3.1	Convenția cdecl	60
6.3.2	Convenția stdcall	61
6.3.3	Convenția fastcall	61
6.4	Funcții standard din msvcrt	62
6.4.1	Afișarea pe ecran și citirea de la tastatură	62
6.4.2	Lucrul cu fișiere text	63
6.5	Întrebări recapitulative	64
6.6	Mersul lucrării	65
6.6.1	Probleme rezolvate	65
6.6.2	Probleme propuse	65
7	Scrierea de macrouri și proceduri	67
7.1	Scopul lucrării	67
7.2	Scrierea și utilizarea macrourilor	67
7.3	Scrierea de proceduri în limbaj de asamblare	68
7.4	Întrebări recapitulative	70
7.5	Mersul lucrării	70

7.5.1	Probleme rezolvate	70
7.5.2	Probleme propuse	70
8	Utilizarea coprocesorului matematic	73
8.1	Scopul lucrării	73
8.2	Considerații generale	73
8.2.1	Principiul de funcționare	73
8.2.2	Tipuri de date cunoscute de Intel 8087	74
8.2.3	Erori de operație (excepții)	74
8.3	Setul de instrucțiuni al coprocesorului	75
8.3.1	Instrucțiuni de transfer de date	76
8.3.2	Instrucțiuni transfer de date intern	77
8.3.3	Instrucțiuni încărcare a constantelor	77
8.3.4	Instrucțiuni aritmetice și de comparare	77
8.3.5	Funcții în virgulă mobilă	79
8.3.6	Instrucțiuni de comandă	79
8.4	Întrebări recapitulative	80
8.5	Mersul lucrării	81
8.5.1	Probleme rezolvate	81
8.5.2	Probleme propuse	81
A	Exemple de cod	83
B	Lista instrucțiunilor uzuale în Limbaj de Asamblare	97

Laborator 1

Reprezentarea datelor în calculator

1.1 Scopul lucrării

Scopul lucrării îl reprezintă înțelegerea modului de reprezentare a datelor în calculator.

În prima parte se va studia modul de conversie a unui număr întreg sau zecimal dintr-o bază în alta. Se va pune accent pe conversia numerelor întregi și zecimale din baza 10 într-o bază oarecare, în special baza 16, 2 și 8, precum și pe conversia inversă, dintr-o bază oarecare în baza 10, mai ales din baza 16, 2 și 8 în baza 10. Se va studia și modul de realizare a conversiei din baza 16 direct în baza 2 sau 8 și invers. Vor fi prezentate operații simple (adunări, scăderi) în diferite baze de numerație.

În a doua parte a lucrării se vor prezenta reprezentările interne ale diverselor tipuri de date. Pentru numerele întregi se vor studia reprezentările prin Mărime și Semn (MS), Complement față de 1 (C1), Complement față de 2 (C2), respectiv binar zecimal împachetat și despachetat (BCD). Pentru numerele reale se va utiliza formatul IEEE scurt, lung și temporar.

1.2 Conversii și operații în diverse baze de numerație

1.2.1 Considerații teoretice

Un sistem de numerație este constituit din totalitatea regulilor de reprezentare a numerelor cu ajutorul anumitor simboluri denumite cifre.

Pentru orice sistem de numerație, numărul semnelor distincte pentru cifrele sistemului este egal cu baza (b). Deci pentru baza $b = 2$ (numere scrise în binar) semnele vor fi cifrele 0 și 1. Pentru baza $b = 16$ (hexazecimal) semnele vor fi 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F . Se observă că pentru numerele scrise într-o bază mai mare decât baza 10 (zecimal) se folosesc și alte simboluri (litere) pe lângă cifrele obișnuite din baza 10. Astfel, în cazul numerelor scrise în hexazecimal, literele A, B, C, D, E, F au ca și valori asociate 10, 11, 12, 13, 14, 15.

Pentru a face ușor distincție între numerele scrise într-o anumită bază, la sfârșitul numărului se mai scrie o literă ce simbolizează baza, de exemplu:

- B pentru numerele scrise în binar (baza 2)
- Q pentru numerele scrise în octal (baza 8)
- D pentru numerele scrise în zecimal (baza 10)

Tabel 1.1: Corespondența cifrelor în bazele 10, 16, 2 și 8

Valoarea în zecimal	Valoarea în hexazecimal	Numărul binar coresp. cifrei hexa	Numărul binar coresp. cifrei octal
0	0	0000	000
1	1	0001	001
2	2	0010	010
3	3	0011	011
4	4	0100	100
5	5	0101	101
6	6	0110	110
7	7	0111	111
8	8	1000	
9	9	1001	
10	A	1010	
11	B	1011	
12	C	1100	
13	D	1101	
14	E	1110	
15	F	1111	

Mai trebuie ținut cont la trecerea unui număr prin bazele 2,8,16 că gruparea cifrelor din baza 2 se face "dinspre virgulă spre extremități", adică la numerele întregi de la dreapta la stânga (prin completare cu zerouri la stânga numărului dacă este cazul, deci în partea care nu-i afectează valoarea), iar la numerele zecimale gruparea se face de după virgulă de la stânga la dreapta, prin adăugare de zerouri la dreapta numărului.

În concluzie:

- $347_{10} = 15B_{16} = 1\ 0101\ 1011_2 = 533_8$
- $438_{10} = 1B6_{16} = 1\ 1011\ 0110_2 = 666_8$

Conversia părții zecimale

Pentru a converti un număr subunitar (deci partea fracționară a unui număr) din baza 10 într-o bază oarecare se fac înmulțiri succesive ale părților fracționare până când se ajunge la parte fracționară nulă, sau se ajunge la perioadă sau se depășește capacitatea de reprezentare (se obțin cifre suficiente, deși algoritmul nu s-ar fi terminat). Ceea ce depășește partea zecimală la fiecare înmulțire reprezintă o cifră a numărului în baza spre care se face conversia.

Pentru exemplificare este ușor să se folosească schema următoare, care prin cele două linii separă mai clar cifrele reprezentării precum și indică mai bine poziția virgulei (cifrele de la prima înmulțire în jos adică de sub linie sunt după virgulă). Trebuie remarcat că se înmulțește doar ceea ce este în dreapta virgulei.

0,	47×2
0	94
1	88
1	76
1	52
1	04
0	08
0	16
0	32
0	64
1	28
0	56
1	12
0	24
0	48
0	96
1	...

Figura 1.2: Conversia numărului $0,47_{10}$ în binar

Exemplu: Să se convertească numărul $0,47_{10}$ în binar, octal și hexazecimal.

Conform Figurii 1.2, avem $0,47_{10} \approx 0,0111\ 1000\ 0101\ 0001_2 = 0,7851_{16} \approx 0,3605_8$.

Conversia unui număr care are atât parte întreagă cât și parte zecimală se face convertind pe rând partea întreagă și cea zecimală.

Exemplu: Să se reprezinte în bazele 2 și 16 numărul real 14,75.

Obținem: $14_{10} = 1100_2 = E_{16}$,

iar $0,75_{10} = 0,11_2 = 0, C_{16}$.

Deci, $14,75_{10} = 1110,11_2 = E, C_{16}$.

1.2.3 Conversia unui număr dintr-o bază oarecare în baza 10

Pentru a converti un număr dintr-o bază oarecare în baza 10 se poate folosi formula definită în prima parte a lucrării și anume dacă se dă un număr scris într-o bază oarecare b sub forma parte întreagă și parte zecimală:

$$Nr(b) = C_n C_{n-1} C_{n-2} \dots C_2 C_1 C_0, D_1 D_2 D_3 \dots$$

atunci valoarea sa în baza 10 va fi:

$$\begin{aligned} Nr(10) = & C_n \cdot b^n + C_{n-1} \cdot b^{n-1} + C_{n-2} \cdot b^{n-2} + \dots + C_2 \cdot b^2 + C_1 \cdot b^1 + C_0 \cdot b^0 \\ & + D_1 \cdot b^{-1} + D_2 \cdot b^{-2} + D_3 \cdot b^{-3} + \dots \end{aligned}$$

Exemple:

- Se dă numărul întreg în hexazecimal $3A8_{16}$ și se cere valoarea sa în zecimal:
 $N = 3 \cdot 16^2 + 10 \cdot 16 + 8 = 3 \cdot 256 + 160 + 8 = 936_{10}$
- Se dă numărul fracționar $0,341_8$ scris în octal și se cere valoarea sa în zecimal:
 $N = 3 \cdot 8^{-1} + 4 \cdot 8^{-2} + 1 \cdot 8^{-3} = \frac{3}{8} + \frac{4}{64} + \frac{1}{512} = 0.4394_{10}$
- Se dă numărul în binar $110,11_2$ și se cere valoarea sa în hexazecimal și în zecimal:
 $N = 110,11_2 = 6, C_{16} = 6,75_{10}$

1.2.4 Operații simple cu numere scrise în diverse baze

În continuare vor fi prezentate operațiile de adunare și scădere a numerelor scrise în binar, octal și hexazecimal a numerelor întregi fără semn.

Adunarea

Adunarea se face după aceleași reguli ca în zecimal, cu observația că cifra cea mai mare dintr-o bază b va fi $b - 1$ (adică 9 în zecimal, 7 în octal, 1 în binar și F în hexazecimal). Deci dacă prin adunarea a două cifre de rang i se va obține un rezultat mai mare decât $b - 1$, va apare acel transport spre cifra de rang următor $i + 1$, iar pe poziția de rang i va rămâne restul împărțirii rezultatului adunării cifrelor de rang i la bază. Transportul spre cifra de rang $i + 1$ va deveni o nouă unitate la suma cifrelor de rang $i + 1$, adică se va mai aduna acel transport 1.

Exemple:

$$\begin{array}{r}
 \overset{1}{0} \overset{1}{1} \overset{1}{0} \overset{1}{1} \overset{1}{0} \overset{1}{1} \overset{1}{0} \overset{1}{2} + \\
 \hline
 10110101_2 \\
 \hline
 100001011_2
 \end{array}
 \quad
 \begin{array}{r}
 \overset{1}{1} \overset{1}{3} \overset{1}{6} \overset{1}{4} \overset{1}{8} + \\
 \hline
 3721_8 \\
 \hline
 5305_8
 \end{array}
 \quad
 \begin{array}{r}
 \overset{1}{6} \overset{1}{D} \overset{1}{8} \overset{1}{A} \overset{1}{3} \overset{1}{2} \overset{1}{16} + \\
 \hline
 33E4C8_{16} \\
 \hline
 A16EFA_{16}
 \end{array}$$

S-a marcat transportul de o unitate la cifra de rang superior prin scrierea unui 1 deasupra cifrei de rang superior la care s-a făcut transportul. Operația de adunare în binar este utilă la reprezentarea numerelor în complement față de 2 când se alege varianta adunării valorii 1 la reprezentarea din complement față de 1 (vezi partea a doua a lucrării).

Exemplu: Să se adune cele 2 numere întregi 347_{10} și 438_{10} convertite mai sus în lucrare în bazele 16 și 8 și să se verifice rezultatul prin conversia lui în baza 10.

$$347_{10} + 438_{10} = 785_{10}$$

$$15B_{16} + 1B6_{16} = 311_{16}. \text{ Verificare: } 311_{16} = 3 \cdot 256 + 1 \cdot 16 + 1 = 785$$

$$533_8 + 666_8 = 1421_8. \text{ Verificare: } 1421_8 = 1 \cdot 512 + 4 \cdot 64 + 2 \cdot 8 + 1 = 785$$

Scăderea

Și pentru scădere sunt valabile regulile de la scăderea din zecimal și anume: dacă nu se pot scădea două cifre de rang i (adică cifra descăzutului este mai mică decât a scăzătorului) se face "împrumut" o unitate din cifra de rang următor ($i+1$). În cazul în care cifra din care se dorește realizarea "împrumutului" este 0, se face "împrumutul" mai departe la cifra de rang următor.

Exemple:

$$\begin{array}{r}
 \overset{1}{0} \overset{1}{1} \overset{1}{1} \overset{1}{0} \overset{1}{1} \overset{1}{0} \overset{1}{2} - \\
 \hline
 01001100_2 \\
 \hline
 00001110_2
 \end{array}
 \quad
 \begin{array}{r}
 \overset{1}{A} \overset{1}{3} \overset{1}{D} \overset{1}{4} \overset{1}{16} - \\
 \hline
 751B_{16} \\
 \hline
 2EB9_{16}
 \end{array}$$

Să se scadă cele două numere întregi 438_{10} și 347_{10} convertite mai sus în lucrare în bazele de numerație 16 și 8 și să se verifice rezultatul prin conversia lui în zecimal.

$$438_{10} - 347_{10} = 91_{10}$$

$$1B6_{16} - 15B_{16} = 5B_{16}. \text{ Verificare: } 5B_{16} = 5 \cdot 16 + 11 = 91$$

$$666_8 - 533_8 = 133_8. \text{ Verificare: } 133_8 = 1 \cdot 64 + 3 \cdot 8 + 3 = 91$$

Operația de scădere este utilă când se dorește reprezentarea numerelor în Complement față de 2 și se efectuează scăderea din $2^{\text{nr. biti-reprez}} + 1$ a numărului reprezentat în modul.

1.3 Reprezentarea internă a datelor

1.3.1 Reprezentarea numerelor întregi în Mărime și Semn (MS) în Complement față de 1 și de 2 (C1, C2)

În general, constantele și variabilele întregi se reprezintă pe un octet, pe un cuvânt (2 octeți), două cuvinte (dublu cuvânt), sau patru octeți. La toate reprezentările bitul cel mai semnificativ reprezintă semnul, iar restul reprezentării (ceilalți biți) se folosesc pentru reprezentarea în binar a numărului (numerele negative au o reprezentare diferită a modului în cele trei tipuri de reprezentări).

Există deci două câmpuri în reprezentarea numerelor întregi: semnul și modulul. La toate cele trei moduri de reprezentare semnul este 0 pentru numerele pozitive și 1 pentru numerele negative.

Câmpul pentru modul se reprezintă astfel:

- La reprezentarea în mărime și semn (MS) se reprezintă modulul numărului, deci reprezentarea unui număr se va face punând 0 sau 1 pe bitul semn, în funcție de

valoarea pozitivă, respectiv negativă a numărului, iar pe restul reprezentării se va pune valoarea modulului său în baza 2.

- La reprezentarea în complement față de 1 (C1) dacă numărul este pozitiv, reprezentarea este la fel ca la mărime și semn, adică se reprezintă modulul numărului, iar bitul semn este implicit 0. Dacă numărul este negativ, atunci se completează toți biții reprezentării numărului în modul, adică biții 1 devin 0 iar cei cu valori 0 devin 1, astfel ca bitul semn va fi pe 1. Trebuie reținut faptul că se face complementarea reprezentării numărului în modul, adică se reprezintă mai întâi numărul pozitiv, după care se completează toți biții.
- La reprezentarea în complement față de 2, dacă numărul este pozitiv se reprezintă la fel ca la mărime și semn, respectiv ca la complement față de 1, adică se reprezintă numărul în modul, iar bitul de semn este 0. Dacă numărul este negativ, atunci se completează față de 2 reprezentarea numărului în modul și anume se scade din valoarea 2^{n+1} (unde n este numărul de biți pe care se face reprezentarea, bitul de semn devenind automat 1) reprezentarea în modul; o altă cale de a obține reprezentarea în complement față de 2 a numerelor negative este adăugând valoarea 1 la reprezentarea numărului negativ în complement față de 1.

Din modul de reprezentare a numerelor în cele trei forme rezultă că numerele pozitive au aceeași reprezentare atât în mărime și semn cât și în complement față de 1 și în complement față de 2.

O atenție sporită trebuie acordată spațiului minim (număr minim de octeți) pe care se poate reprezenta un număr în cele trei moduri de reprezentare. De exemplu când se dorește aflarea numărului minim de octeți pe care se poate reprezenta numărul 155 trebuie să se țină cont că pentru reprezentarea modulului este la dispoziție mai puțin cu un bit (cel de semn) din spațiul ales pentru reprezentare. În acest caz deși valoarea modulului său încapă pe un octet ($155 = 9B_{16}$), numărul nu se poate reprezenta pe un octet în nici un mod de reprezentare, deoarece bitul semn trebuie reprezentat separat, altfel la interpretarea reprezentării $9B_{16}$, primul bit fiind 1, reprezentarea va fi a unui număr negativ în loc de numărul dorit. În concluzie, pentru reprezentarea numărului 155 va fi nevoie de minim 2 octeți (reprezentarea se face pe multiplu de octet), iar numărul va fi reprezentat astfel: $009B_{16}$ în toate cele trei moduri de reprezentare, fiind pozitiv.

Exemple: Să se reprezinte pe 4 octeți următoarele numere: 157, 169, -157 , -169 .

$157_{10} = 1001\ 1101_2 = 9D_{16}$. Deci reprezentarea în MS. C1 și C2 va fi $00\ 00\ 00\ 9D_{16}$.

$169_{10} = 1010\ 1001_2 = A9_{16}$. Deci reprezentarea în MS. C1 și C2 va fi $00\ 00\ 00\ A9_{16}$.

Pentru -157 , se reprezintă mai întâi în modul (este calculat mai sus) și se obține:

MS: $1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 1101_2 = 80\ 00\ 00\ 9D_{16}$

C1: $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 0110\ 0010_2 = FF\ FF\ FF\ 62_{16}$

C2: $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 0110\ 0011_2 = FF\ FF\ FF\ 63_{16}$

Pentru -163 analog:

MS: $80\ 00\ 00\ A9_{16}$

C1: $FF\ FF\ FF\ 56_{16}$

C2: $FF\ FF\ FF\ 57_{16}$

1.3.2 Reprezentarea numerelor reale în format IEEE

Standardul IEEE de reprezentarea numerelor reale propune trei moduri de reprezentare pentru numerele reale:

- Formatul scurt pe 4 octeți
- Formatul lung pe 8 octeți
- Formatul temporar pe 10 octeți

Numerele reale se reprezintă în formatele scurte și lungi în memoria calculatorului, iar formatul temporar se găsește la încărcarea numerelor reale în coprocesorul matematic.

Toate cele trei formate conțin trei părți:

Semn	Caracteristică	Mantisă
------	----------------	---------

- Bitul de semn - S
- Caracteristica - C (pe 8, 11, respectiv 15 biți, la formatul scurt, lung, temporar)
- Mantisă - M (pe 23, 52, respectiv 64 biți)

Pentru fiecare reprezentare: S este 0 dacă numărul este pozitiv și 1 dacă numărul este negativ.

Caracteristica $C = E + 7F_{16}$ (respectiv $3FF_{16}$ la IEEE lung și $3FFF_{16}$ la formatul temporar), unde E este exponentul.

Pentru găsirea mantisei mai întâi se normalizează numărul scris în baza 2, adică se scrie numărul sub forma: $NR = 1. < \text{alte cifre binare} > \cdot 2^E$.

La reprezentarea în format IEEE scurt și lung, mantisa este formată din cifrele de după virgulă, deci primul 1 dinaintea virgulei nu se mai reprezintă în mantisă, iar la formatul temporar se reprezintă toate cifrele din număr.

Exemple:

Să se reprezinte în format IEEE scurt numărul $17,6_{10}$.

Se va converti separat partea întreagă și cea zecimală și se obține:

Partea întreagă: $17_{10} = 1116 = 0001\ 0001_2$

Partea zecimală: $0,6_{10} = 0,(1001)_2$ (se observă că numărul este periodic)

Deci $17,6_{10} = 10001,(1001)_2$.

Se normalizează numărul: $17,6_{10} = 10001,(1001)_2 = 1,0001(1001) \cdot 2^4$ (deși era mai corect în loc de 2^4 să se scrie 10^{100}_2 pentru că notarea era în baza 2, faptul că se calculează caracteristica mai ușor în hexa decât în binar poate fi o scuză motivată).

Din această reprezentare se poate deduce mantisa (cea ce este după virgulă, deci fără acel 1 dinaintea virgulei care prin convenție nu se mai reprezintă) și anume: $M = 0001(1001)_2$.

În continuare se calculează caracteristica: $C = E + 7F_{16} = 4 + 7F_{16} = 83_{16} = 1000\ 0011_2$.

Se va scrie bitul semn 0 și deja se poate trece la scrierea reprezentării:

0	1000011	00011001100110011001100
Semn	Caracteristică	Mantisă

Pentru a scrie reprezentarea în hexa se vor grupa câte 4 cifre binare. Atenție însă la faptul că gruparea a câte 4 cifre nu va corespunde caracteristicii, datorită bitului de semn care decalează o poziție. Deci cifrele hexa ale caracteristicii nu se vor regăsi în reprezentarea notată în hexa.

Rezultatul final al reprezentării este: $41\ 8C\ CC\ CC_{16}$.

În cazul practic, în memoria calculatorului, datorită unei rotunjiri care se face la ultimul bit al reprezentării, se poate observa că în mantisă ar mai urma un 1 după cei 23 de biți, iar calculatorul va face rotunjire superioară, de aceea pe ultimul bit (cel mai puțin semnificativ) va apare 1, iar reprezentarea va fi: $41\ 8C\ CC\ CD_{16}$.

În mod analog se va reprezenta $-23,5_{10}$:

$$23_{10} = 17_{16} = 1\ 0111_2$$

$$0,5_{10} = 0,1_2$$

Deci $23,5_{10} = 10111,1 = 1,01111 \cdot 2^4$ de unde rezultă $M = 0111100000000000 \dots$ (23 de biți).

$$\text{Caracteristica } C = 7F_{16} + 4_{16} = 83_{16}.$$

Se pune bitul semn pe 1.

Reprezentarea direct în hexa este $C1\ BC\ 00\ 00_{16}$.

În continuare se pune problema inversă reprezentării: se dă reprezentarea unui număr în format IEEE scurt și se cere aflarea numărului real care este astfel reprezentat.

Exemplu: Se dă reprezentarea $43\ 04\ 33\ 33_{16}$ și se cere valoarea zecimală a numărului real reprezentat.

Se scrie în binar reprezentarea: $0100\ 0011\ 0000\ 0100\ 0011\ 0011\ 0011\ 0011_2$. De aici se deduce că:

- Semnul este 0.
- Caracteristica este $C = 1000\ 0110_2 = 86_{16}$.
Rezultă deci că exponentul este $E = 86_{16} - 7F_{16} = 7_{16}$.
- Mantisa $M = 0000\ 1000\ 0110\ 0110 \dots$

Numărul este:

$$\begin{aligned} Nr &= 1, M \cdot 2^E \\ &= 1,0000\ 1000\ 0110 \dots_2 \cdot 2^7 \\ &= 1000\ 0100,00110011 \dots_2 \\ &= 128 + 4 + 0,125 + 0,0625 + \dots \\ &\approx 132,1875 \end{aligned}$$

Valoarea exactă era 132,2.

1.3.3 Reprezentarea numerelor în BCD împachetat și despachetat

Pe lângă modurile de reprezentare a numerelor întregi în mărime și semn, complement față de 1 și de 2 mai există reprezentarea în BCD împachetat și despachetat.

În reprezentarea BCD împachetat se reprezintă câte o cifră zecimală pe 4 biți, deci câte 2 cifre zecimale pe octet. În reprezentarea BCD despachetat se reprezintă câte o cifră zecimală pe octet (deci pe primii patru biți se pune 0).

Aceste moduri de reprezentare se folosesc și pentru o mai bună lizibilitate a numerelor din punct de vedere al programatorului, chiar dacă aceasta se face prin pierderea unei bune porțiuni din spațiul de reprezentare (la BCD despachetat nu se mai folosesc codurile pe patru biți care trec ca valoare de 9, iar la BCD despachetat se mai pierde încă jumătate de octet în plus).

Pentru a putea realiza operații cu numere reprezentate în BCD există instrucțiuni suplimentare de corecție a rezultatului după adunare, înmulțire care se vor studia în lucrarea cu instrucțiuni pentru operații aritmetice.

Exemplu: Numărul 3912_{10} se va reprezenta în BCD:

- împachetat: 3912_{16} deci pe 2 octeți;
- despachetat 03090102_{16} deci pe 4 octeți.

1.4 Întrebări recapitulative

1. Câte simboluri pentru reprezentarea cifrelor avem într-o bază b ? Care sunt acestea pentru $b = 16$?
2. Cum se poate converti un număr din baza 2 în bazele 8 sau 16 fără a efectua înmulțiri sau împărțiri?
3. Care este formula de conversie a unui număr dintr-o bază oarecare în baza 10?
4. Care este rezultatul adunării $1 + 1$ în baza 2?
5. Cum diferă reprezentarea unui număr pozitiv în reprezentarea C1 față de MS?
6. Având reprezentarea unui număr negativ în C1, cum obținem reprezentarea în C2?
7. Care sunt cele trei componente ale reprezentării unui număr real în format IEEE?

1.5 Mersul lucrării

1.5.1 Conversii

Se vor realiza conversiile din exemplele prezentate:

- conversia numerelor din baza 10 în baza 2, 8 și 16
- conversia unui număr între bazele 2, 8 și 16
- conversii din bazele 2, 8 și 16 în baza 10
- operații de adunare și scădere numere în bazele 2 și 16

1.5.2 Reprezentări

- Se vor reprezenta în MS, C1 și C2 numerele întregi prezentate în exemplele de mai sus.
- Se vor reprezenta în format IEEE scurt numerele reale.
- Se vor realiza două exemple de aflare a numărului real dându-se valoarea reprezentării.

Laborator 2

Arhitectura Intel x86 și elemente de bază ale limbajului de asamblare

2.1 Arhitectura Intel x86

2.1.1 Structura unui sistem de calcul (considerații generale)

Majoritatea sistemelor de calcul din zilele noastre (și nu numai) sunt constituite din 3 tipuri de componente:

- memoria principală
- unitatea centrală de prelucrare (în engleză Central Processing Unit - CPU)
- dispozitive de intrare/ieșire

Memoria principală a sistemului este direct accesibilă de către unitatea centrală de prelucrare, și poate conține date sau cod (vom vedea ulterior că acesta este doar un tip mai special de date). Acest tip de memorie se mai numește și memorie RAM (Random Access Memory), deoarece orice locație aleatoare din aceasta poate fi adresată în timp constant (spre deosebire de discurile magnetice, care favorizează citirea datelor secvențial, sau de memoriile asociative, unde datele sunt accesate prin conținut, nu prin adresă). O altă caracteristică a memoriei principale este volatilitatea. Atunci când alimentarea cu energie a sistemului este întreruptă, conținutul acesteia se pierde.

Unitatea centrală de prelucrare este un circuit care execută instrucțiunile dintr-un program, citind sau scriind date din memorie, efectuând diverse operații asupra acestora, sau accesând dispozitivele de intrare/ieșire.

Dispozitivele de intrare/ieșire fac diferența dintre un sistem de calcul și alte aparate care consumă energie electrică și produc căldură. Cele mai întâlnite astfel de dispozitive sunt tastatura, mouse-ul, monitorul, respectiv mediile de stocare. Prin acestea se asigură interacțiunea cu utilizatorul.

2.1.2 Familia Intel x86

Familia de procesoare x86 este, în prezent, cea mai folosită pe piața calculatoarelor desktop (în timp ce procesoarele ARM domină piața dispozitivelor mobile). Deși aceste procesoare au evoluat de-a lungul timpului, suferind diverse modificări arhitecturale, s-a pus un accent

deosebit pe păstrarea compatibilității. Astfel, un program scris pentru generația '386 va rula fără probleme pe un procesor Intel de ultimă generație.

Procesoarele mai vechi au fost procesoare pe 16 bit, ceea ce însemna că registrele acestora (vom vedea ce înseamnă registru în secțiunea următoare) aveau o lungime de 16 bit, iar instrucțiunile erau mai scurte.



Generația a 3-a (80386) a adus procesoare pe 32 bit, în care regiștrii au lungimi pe 32 bit, iar setul de instrucțiuni este îmbogățit. Se păstrează compatibilitatea cu procesoarele pe 16 bit, dar apar instrucțiuni noi, precum și concepte noi, cum ar fi modul virtual de operare.

Unul din principalele dezavantaje ale sistemelor pe 32 bit este faptul că nu pot adresa mai mult de 4GB de memorie RAM (un sistem pe 16 bit putea adresa maxim 1MB de memorie convențională).

Aplicațiile moderne, ce procesează volume mari de date au nevoie de spații de memorie mai largi. Au apărut astfel procesoarele pe 64 bit, care pot adresa până la 256 TB (se folosesc doar 48, din cei 64 bit pentru adrese).

2.1.3 Arhitectura setului de instrucțiuni

În continuare, se vor discuta componentele procesorului, vizibile pentru un programator în limbaj de asamblare. Astfel, un procesor dispune de un set de regiștri interni folosiți pentru păstrarea temporară a datelor, a adreselor sau a instrucțiunilor. Există regiștri generali folosiți în majoritatea operațiilor aritmetico-logice și regiștri speciali, care au o destinație specială.

Procesoarele x86 conțin 8 regiștri de uz general:

EAX - Accumulator	ESI - Source Index
EBX - Base	EDI - Destination Index
ECX - Counter	EBP - Base Pointer
EDX - Data	ESP - Stack Pointer

Fiecare dintre aceștia poate stoca o valoare pe 32 bit. Denumirea acestor regiștri începe cu litera 'E' (de la "Extended"), deoarece sunt extensii ale regiștrilor de uz general pe 16 bit: AX, BX, CX, DX, SI, DI, BP, SP. Dacă se doresc doar cei mai puțin semnificativi 16 bit dintr-un astfel de registru, se pot folosi aceste denumiri pe 16 bit. În plus, pentru primii 4 dintre acești regiștri se pot adresa separat cei mai puțin semnificativi 2 octeți, adică biții 0...7, respectiv 8...15, folosind denumirile *L (low), respectiv *H (high), unde * poate fi una din literele A, B, C sau D. În Figura 2.1 sunt exemplificate părțile adresabile ale registrului EAX.

Un alt registru important este EFLAGS, în care sunt reținuți diverși indicatori de stare sau control ai sistemului.

O parte din aceste flag-uri, ilustrate în Figura 2.2 sunt:

- *CF* - *Carry Flag* - este setat pe 1, dacă la operația anterioară, s-a generat un transport
- *PF* - *Parity Flag* - indică dacă numărul de biți de 1 din ultimul octet al rezultatului anterior este impar.

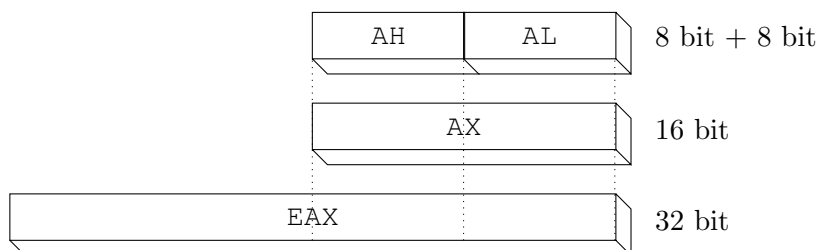


Figura 2.1: Componentele registrului EAX

- *AF* - *Adjust (Auxiliary Carry) Flag* - indică faptul că la operația anterioară, pe cei mai puțin semnificativi 4 biți, s-a generat un transport
- *ZF* - *Zero Flag* - se setează dacă rezultatul operației anterioare este 0
- *SF* - *Sign Flag* - indică semnul rezultatului generat anterior
- *OF* - *Overflow Flag* - indică o depășire de capacitate, la ultima operație aritmetică

Registrul EIP (Instruction Pointer), pe 32 bit, indică locația în memorie a următoarei instrucțiuni ce va fi executată. Acest registru se mai numește uneori și PC (Program Counter). EIP nu poate fi modificat direct, precum regiștrii de uz general, dar se modifică indirect, prin instrucțiuni de control al fluxului de instrucțiuni.

Regiștrii segment (denumiți selectori, în asamblarea pe 32 bit) sunt regiștri pe 16 bit, și participă la formarea unei adrese, sau pointează către o tabelă de descriptori. Acești regiștri sunt: CS (Code Segment), DS (Data Segment), SS (Stack Segment), ES (Extended Segment), FS, respectiv GS.

Procesorul știe să execute diverse tipuri de instrucțiuni, având ca operanzi regiștri, adrese sau date din memoria principală, sau date imediate:

- instrucțiuni aritmetice și logice
Spre exemplu, `add EAX, EBX` adună valoarea din registrul EBX, la cea din registrul EAX, registrul EAX conținând rezultatul obținut. `sub AH, 5` va scădea 5 din valoarea conținută în registrul AH.
- instrucțiuni de transfer a datelor
Sunt folosite pentru a copia date dintr-un registru în altul, sau din/în memorie. `mov EDX, EAX` va copia conținutul registrului EAX în EDX. `mov ECX, [123456H]` va copia în registrul ECX valoarea aflată în memoria principală la adresa 123456H.
- instrucțiuni de control al fluxului de instrucțiuni
Se folosesc atunci când se dorește ca următoarea instrucțiune executată să fie alta decât următoarea instrucțiune din program. De exemplu `jz 100` (Jump If Zero) va sări peste următorii 100 de octeți din program, dacă rezultatul operației anterioare a fost 0, altfel va executa instrucțiunea următoare.

2.1.4 Organizarea memoriei

La organizarea memoriei, se întâlnesc 2 moduri de adresare: modul real și modul protejat.

În modul real spațiul maxim de adresare al memoriei este de 1MB. Această memorie este împărțită în segmente de lungime fixă de 64KB. Adresa de început a unui segment se păstrează în unul dintre regiștrii segment (CS, DS, SS, ...). Deoarece un registru segment

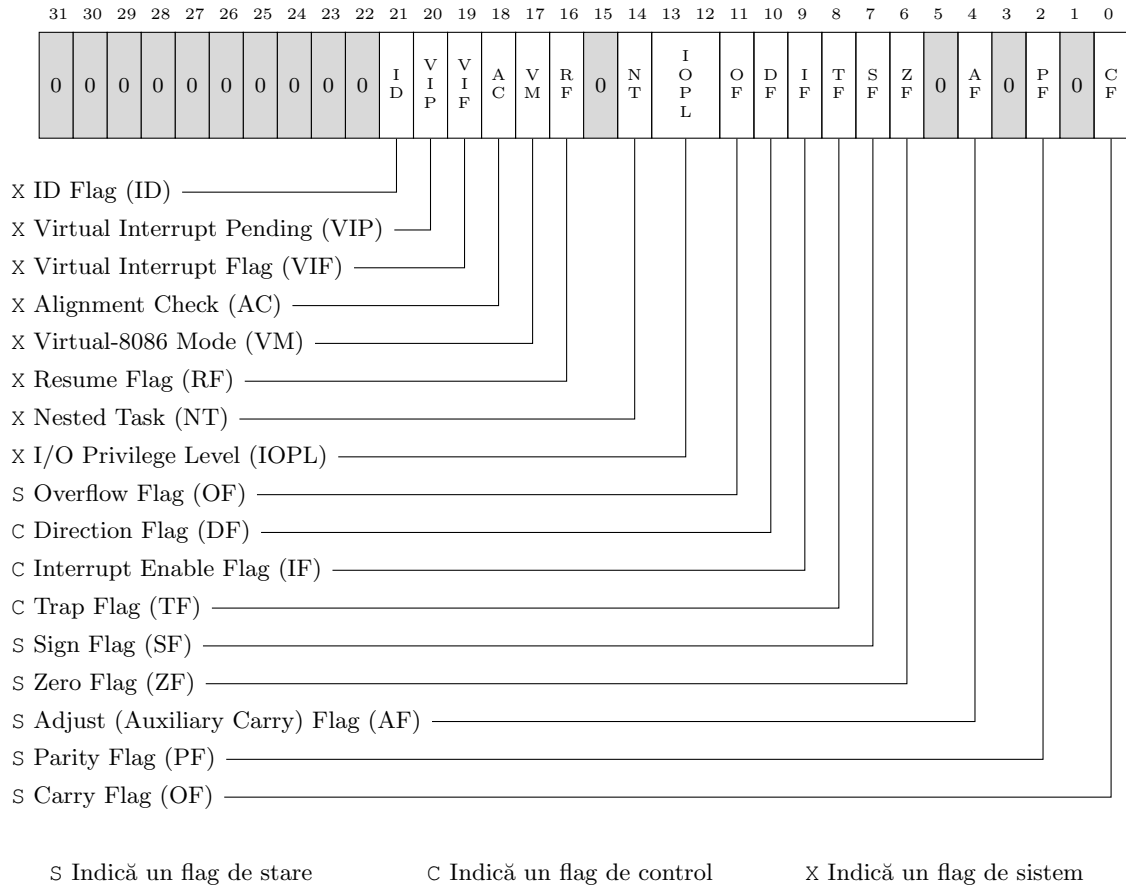


Figura 2.2: Registrul EFLAGS

are doar 16 bit, în el se păstrează doar partea mai semnificativă a adresei de segment, ultimii 4 bit considerându-se în mod implicit 0. Adresa unei locații de memorie se calculează ca o sumă între adresa de segment și o adresă de offset. Adresa de segment se obține prin multiplicarea conținutului registrului segment cu 16 (deplasarea la stânga cu 4 poziții binare). Adresa de offset se calculează pe baza modului de adresare și eventual a adresei conținute în codul de instrucțiune. Prin adunare se obține o adresă fizică pe 20 bit, suficientă pentru adresarea unui spațiu de 1 MB ($1M = 2^{20}$). În exemplul de mai jos, pentru claritate, valorile de adrese sunt exprimate în hexazecimal.

1	2	3	4	0	Adresa de segment
	5	6	7	8	Adresa de offset
<hr/>					
1	7	9	B	8	Adresa fizică

Acest mod de calcul a adresei fizice are câteva consecințe:

- spațiul maxim de adresare este 1MB
- un segment trebuie să înceapă la o adresa multiplu de 16
- un segment are maxim 64KB
- segmentele se pot suprapune parțial sau total

- aceeași locație fizică se poate exprima prin mai multe variante de perechi de adrese (segment:offset)
- există puține posibilități de protejare a zonelor de memorie
- orice program poate adresa orice locație de memorie, neputându-se impune restricții (lucru nedorit într-un sistem multitasking)

Modul protejat s-a introdus odată cu procesorul '386 și apoi s-a perfecționat la procesorul '486. Acest mod a fost necesar pentru a soluționa limitările modului real, în special în ceea ce privește spațiul limitat de adresare și posibilitățile reduse de protecție.

În modul protejat exprimarea adresei se face la fel prin adresă de segment și adresa de offset, însă cu anumite amendamente (vezi Figura 2.3):

- un registru segment păstrează un selector de segment și nu adresa de început a segmentului;
- selectorul este un indicator care arată locul unde se află o structură de date care descrie un segment și care poartă numele de descriptor de segment
- un descriptor de segment conține: adresa segmentului (pe 32 de biți) lungimea segmentului (pe 20 de biți), indicatori pentru determinarea drepturilor de acces și indicatori care arată tipul și modul de utilizare a segmentului
- adresa de offset se exprimă pe 32 de biți

Aceste modificări generează următoarele consecințe:

- spațiul maxim de adresare al memoriei se extinde la 4GB ($4G = 2^{32}$)
- un segment are o lungime variabilă, în interval larg de la 1 octet la 4GB
- se definesc trei nivele de protecție (0, cel mai prioritar)
- un segment este accesibil numai taskului alocat și eventual sistemului de operare
- anumite segmente pot fi blocate la scriere (ex: segmentele de cod)
- rezultă un mecanism complex de alocare și de protecție a zonelor de memorie

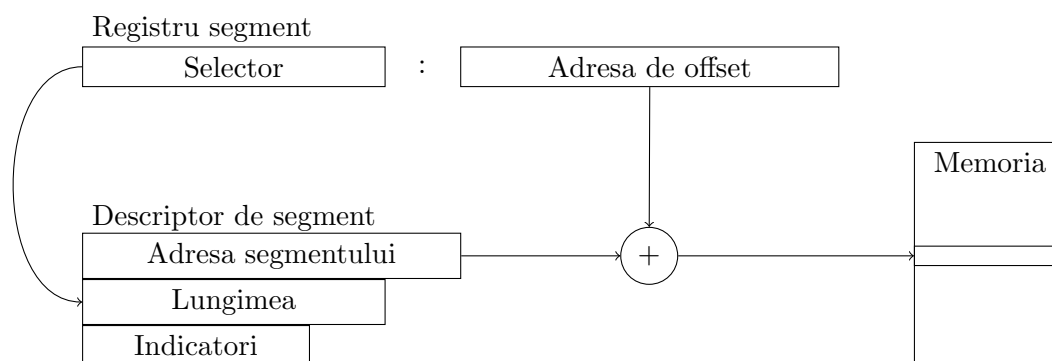


Figura 2.3: Calculul adresei fizice în modul protejat

În sistemele de operare Windows și Linux, pe 32 bit, s-a ales modelul memoriei *flat*. Acest lucru înseamnă că în cadrul programului, toți descriptorii de segment, cu excepția

segmentului FS, descriu un segment a cărui adresă de început este 0, iar lungimea este $2^{32} - 1$. Segmentul FS este un segment special, care în sistemul de operare Windows conține structura TIB (Thread Information Block), și are altă adresă de început și altă lungime.

Fiecare program are acces la o memorie virtuală de 4GB (din care doar prima jumătate este adresabilă în mod utilizator), în care este izolat de celelalte programe. Acest lucru nu înseamnă că fiecare program folosește 4 GB de memorie fizică. Sistemul menține o tabelă cu corepondența dintre paginile de memorie virtuală (specifice programului) și cadrele de memorie fizică (specifice întregului sistem), așa cum este ilustrat în Figura 2.4.

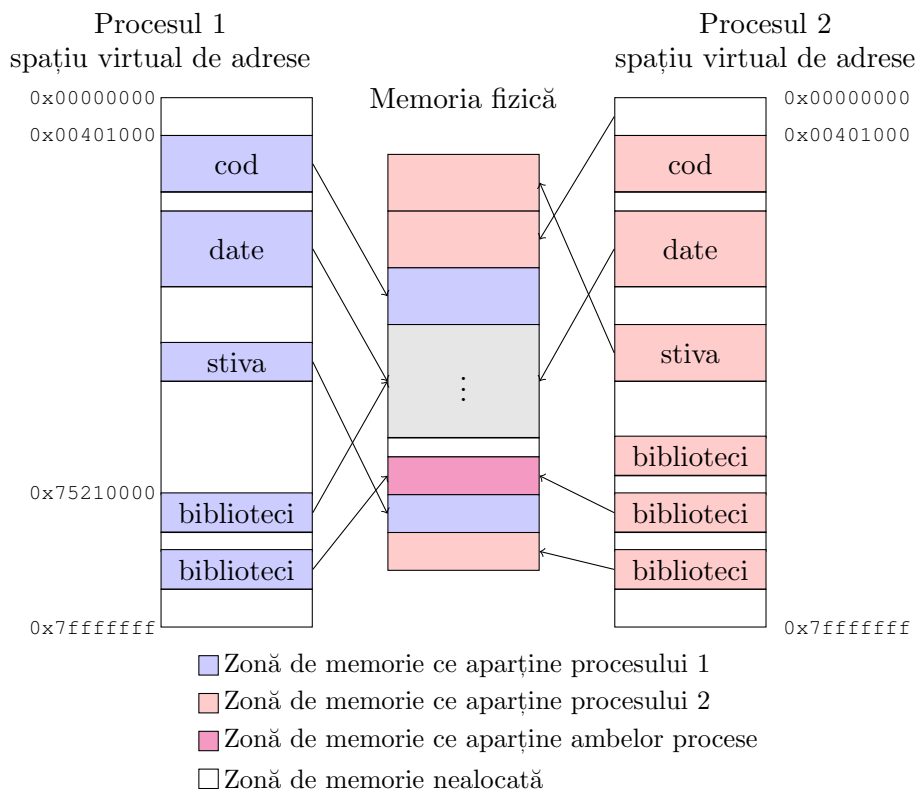


Figura 2.4: Corespondența între adresele virtuale și cele fizice

Astfel, fiecare program are propriul spațiu de adrese. Dacă un program scrie o dată la adresa 123456H în memorie, iar un alt program, ce rulează în paralel cu primul scrie altă valoare, la aceeași adresă, cele două valori nu se vor suprascrie una pe alta. Deși ambele programe scriu la adresa virtuală 123456H, în memoria fizică, cele două adrese virtuale vor avea corespondente diferite.

2.2 Elementele de bază ale limbajului de asamblare

2.2.1 Construcții de bază

O **instrucțiune** în limbaj de asamblare are următorul format:

```
[<etichetă>:] [<operație> [<operandi>]] [;<comentarii>]
```

<etichetă> este un nume format din litere, cifre sau caractere speciale, care începe neapărat cu o literă sau cu un caracter special. Etichetele sunt folosite pentru a ne referi la o anumită poziție, din cadrul programului.

<operație> este mnemonica unei instrucțiuni în asamblare.

<operanzi> pot fi zero, unu sau mai mulți, și reprezintă entitățile asupra cărora se efectuează operația. Pot fi regiștiri, locații de memorie, date imediate sau constante.

<comentarii> sunt orice text, ce începe cu ';', și continuă până la sfârșitul rândului. Acestea sunt ignorate de către asamblor, și utile pentru lizibilitatea codului.

Constantele numerice sunt specificate printr-un șir de cifre și litere, ce începe obligatoriu cu o cifră. Un număr zecimal se specifică pur și simplu printr-un șir de cifre. Un număr binar se specifică printr-un șir de cifre binare, terminate cu litera 'B'. Analog se folosește litera 'Q' pentru octal, respectiv 'H' pentru hexazecimal. Dacă un număr hexazecimal începe cu o literă (*A...F*), aceasta trebuie precedată de cifra 0.

Exemple: 010010100B, 26157Q, 4921, 4B52H, 0AB3H.

Caracterele sau șirurile de caractere se scriu între apostroafe (') sau ghilimele("). În mod intern, un șir de caractere este echivalent cu un șir de numere, pe 8 bit.

Simbolurile reprezintă poziții în memorie. Acestea pot fi etichete sau variabile.

Etichetele pot fi definite în zona de cod, și sunt folosite pentru a indica o anumită linie de cod. Pentru programator, e mai ușor să indice un salt la o anumită instrucțiune, decât să indice adresa, sau deplasamentul față de adresa curentă.

Variabilele sunt definite în zona de date. Acestea trebuie să aibă unul din tipurile simple: BYTE (1 octet), WORD (2 octeți), DWORD (4 octeți), QWORD (8 octeți), TWORD (10 octeți) sau un tip compus, STRUC sau RECORD.

Pentru a defini o variabilă, se scrie numele acesteia, urmat de un cuvânt cheie, care indică tipul variabilei: DB (de la Define Byte) pentru BYTE, DW pentru WORD, DD pentru DWORD, etc. Pentru a scrie un vector în locul unei variabile simple, valorile se pot scrie, separate de virgulă:

```
1 | vect DB 1, 2, 3, 4, 5
```

Pentru a defini <n> elemente, cu valoarea <x>, se poate scrie <n> DUP (<x>), de exemplu:

```
1 | vect DB 5 DUP(0)
```

Pentru a defini o **constantă**, se folosește cuvântul cheie EQU, într-o expresie de genul:

```
1 | <nume_const> EQU <expresie>
```

De exemplu, se poate scrie

```
1 | ZECE EQU 10
```

Valoarea constantei nu se va stoca în memoria programului, ci asamblorul va înlocui în codul sursă numele acesteia, cu valoarea (similar cu #define din C, nu cu const).

2.2.2 Structura generală a unui program MASM

Un program MASM are structura din Figura 2.5.

Pe prima linie a programului întâlnim directiva .386, care spune asamblorului să genereze un fișier binar, compatibil cu procesoarele din generația '386. Linia a doua conține directiva .model, prin care se specifică tipul memoriei, respectiv tipul limbajului. Modelul flat este cel folosit în Windows și implică folosirea întregului spațiu de adrese, pentru toate selectoarele, cu excepția selectorului FS. stdcall reprezintă convenția de apel a funcțiilor și a altor simboluri exportate de program.

Linia 6 conține directiva includelib, prin care se include în programul nostru biblioteca msvcrt. Această bibliotecă conține funcții și proceduri standard, cum ar fi

```

1  .386
2  .model flat, stdcall
3  ////////////////////////////////////
4
5  ;includem biblioteci, si declaram ce functii vrem sa importam
6  includelib msvcrt.lib
7  extern exit: proc
8  ////////////////////////////////////
9
10 ;declaram simbolul start ca public - de acolo incepe executia
11 public start
12 ////////////////////////////////////
13
14 ;sectiunile programului, date, respectiv cod
15 .data
16 ;aici declaram date
17
18 .code
19 start:
20     ;aici se scrie codul
21
22     ;terminarea programului
23     push 0
24     call exit
25 end start

```

Figura 2.5: Structura generală a unui program MASM

`printf`, `exit`. Pentru a folosi o astfel de funcție, aceasta trebuie declarată ca simbol extern, după cum se observă în linia 7.

Atunci când un program rulează, sistemul de operare trebuie să știe de unde să înceapă execuția. Din acest motiv, trebuie să marcăm acest lucru în mod explicit, în programul nostru, printr-o etichetă (linia 19), apoi să exportăm această etichetă ca simbol public (linia 11).

Directiva `.data` de la linia 15 marchează începutul secțiunii de date. În această secțiune se pot declara variabilele și constantele folosite de program. Analog, directiva `.code` de la linia 18 marchează începutul secțiunii de cod. Acolo se vor scrie toate instrucțiunile programului.

2.3 Instrumentele de lucru pentru compilare și depanare

2.3.1 Asamblorul și linker-ul

Pentru ca un program scris în limbaj de asamblare să se transforme în cod binar, care va fi executat direct de către procesor, este nevoie de un program special numit asamblor. În cadrul acestui laborator, asamblorul folosit este MASM (Microsoft Macro Assembler).

În directorul `masm_minimal`, se găsește fișierul `ml.exe`, cu ajutorul căruia se assemblează fișierele sursă. Pentru asamblare, se tastează următoarea comandă, în Command Prompt:

```
>ml.exe sursa.asm
```

unde `sursa.asm` este numele fișierului ce conține codul sursă. Rezultatul acestui apel nu generează un fișier executabil, ci un fișier obiect, ce conține codul binar, dar nu poate rula. Pentru a se obține un fișier executabil, trebuie editate legăturile programului. Pentru această operație, există fișierul `link.exe` din folderul `masm_minimal`. Asamblarea și link-editarea se pot face printr-o singură comandă:

```
>ml.exe sursa.asm /link /subsystem:console /entry:start msvcrt.lib
```

Această comandă assemblează fișierul `sursa.asm`, generând programul de consolă `sursa.exe`, ce are codul de la eticheta `start` ca punct de pornire și include biblioteca `msvcrt.lib`.

Pentru a nu fi necesară scrierea acestei comenzi de fiecare dată, în `masm_minimal` se găsește script-ul `build_masm.bat`, care primește ca parametru numele sursei, fără extensie, și încearcă să genereze programul executabil.

Pentru a asambla și link-edita fișierul `hello.asm`, și genera `hello.exe`, se folosește comanda:

```
>build_masm hello
```

În caz de eroare, în linia de comandă se va afișa eroarea și linia din codul sursă la care s-a produs eroarea respectivă. Se recomandă folosirea unui editor special pentru a scrie codul sursă (de exemplu, Notepad++), care numerotează liniile de cod. Versiunea de Notepad++ pentru acest laborator vine cu un plugin ce oferă suport pentru asamblarea, rularea și depanarea programelor scrise în limbaj de asamblare. Astfel comanda **Ctrl+F7** execută acțiunile echivalente script-ului `build_masm.bat`, iar **Ctrl+F6** rulează programul obținut, dacă acesta există. Pentru a porni depanatorul Olly Debugger se apasă **F6**.

2.3.2 Depanarea programelor

Pentru a putea înțelege mai bine execuția unui program în limbaj de asamblare, sau pentru depistarea erorilor, se folosesc programe de tip debugger. Acestea permit încărcarea unui fișier executabil, execuția acestuia în instrucțiune cu instrucțiune, vizualizarea conținutului memoriei și a regiștrilor, la fiecare pas, și chiar modificarea unor instrucțiuni sau date, în timp ce programul rulează.

Debugger-ul folosit la acest laborator va fi Olly Debugger. Lansarea acestuia în execuție se face făcând dublu-click pe executabilul acestuia, numit `ollydbg.exe`. Dacă se utilizează plugin-ul pentru MASM din Notepad++, se poate lansa depanarea programului curent apăsând **F6**.

În Olly Debugger, încărcarea unui program pentru depanare, se va face folosind tasta **F3**. Eventualele argumente cu care trebuie rulat programul se pot specifica în fereastra de dialog care apare. Dacă după încărcarea programului nu apare codul acestuia, se poate ajunge la el, folosind combinația de taste **Ctrl+F9**.

Fereastra debugger-ului arată ca în Figura 2.6, și este împărțită în 4 zone:

Zona 1 - conține regiștrii procesorului, împreună cu valorile acestora, la momentul curent al execuției. Regiștrii ai căror valori s-au modificat la instrucțiunea anterioară sunt marcați cu roșu, iar restul cu gri. Flag-urile sunt afișate și separat. Tot în această zonă sunt afișați și regiștrii coprocesorului matematic.

Zona 2 - numită și *dump*, poate afișa diverse porțiuni din memoria programului. La începutul execuției, aici se afișează secțiunea de date. Dump-ul este afișat prin 3 coloane. În prima avem adresa de început a liniei, în a doua, conținutul memoriei la adresa respectivă în format hexazecimal, iar în a 3-a, același conținut, în format text.

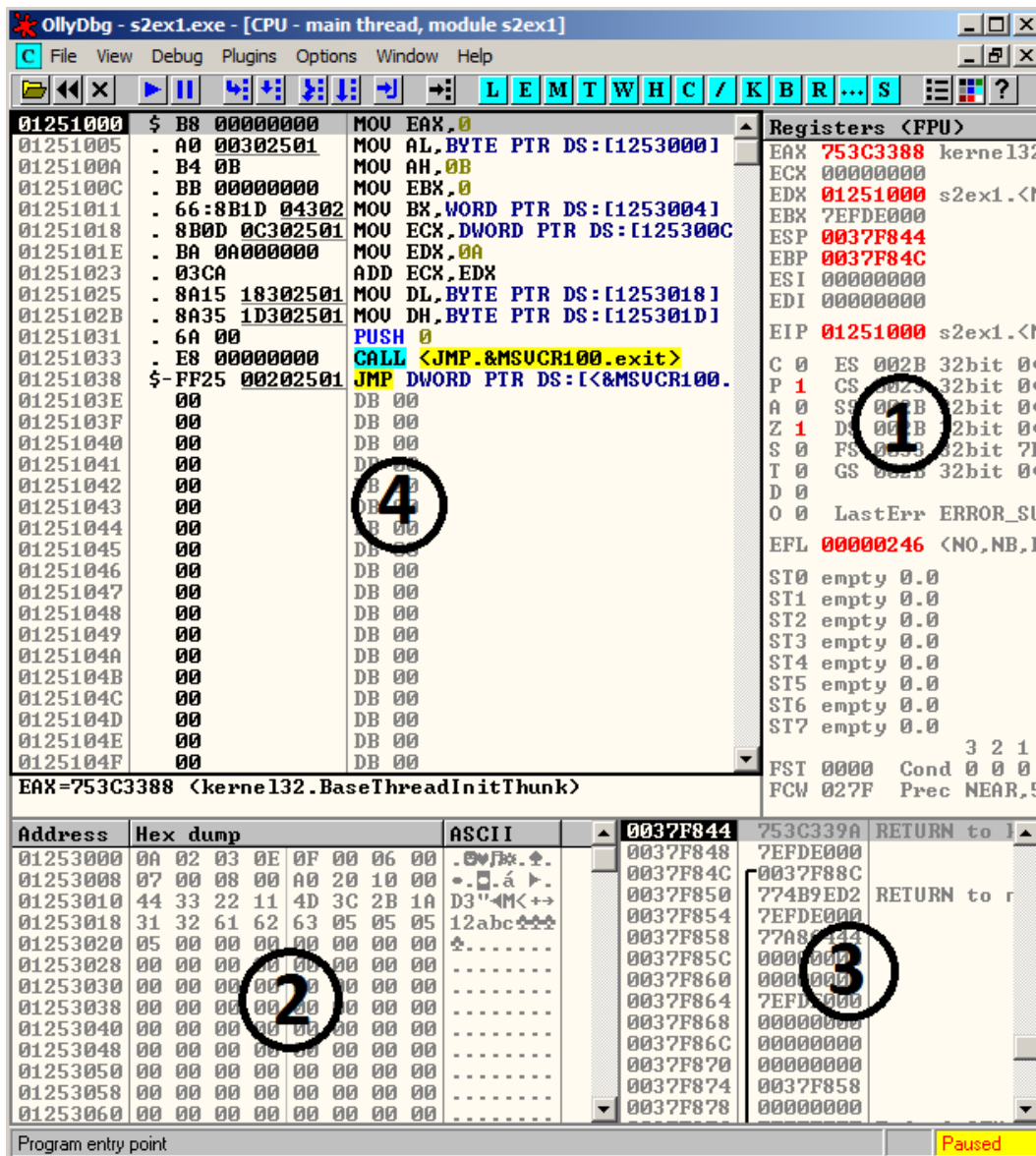


Figura 2.6: Spațiul de lucru în Olly Debugger

Dacă dorim să vedem conținutul memoriei de la o anumită adresă, care apare în una din cele 4 zone ale debugger-ului, se face click dreapta pe acea adresă, și din meniul apărut se alege opțiunea "Follow in Dump".

Zona 3 - conține stiva programului (zonă de memorie cu o întrebuințare specială, ce va fi dezbătută într-o lucrare viitoare). Pe prima coloană sunt afișate adresele, iar pe a doua valorile, în hexazecimal (câte un DWORD). Adresa vârfului stivei este evidențiată. Se poate observa că adresa vârfului stivei este valoarea din registrul ESP. Ca și pentru zona de dump, orice adresă ce aparține de stivă poate fi urmărită în aceasta, făcând click dreapta și alegând "Follow in Stack".

Zona 4 - este zona de afișare a codului. În prima coloană se găsesc adresele instrucțiunilor, în hexazecimal. Adresa instrucțiunii următoare este evidențiată. În a doua coloană se găsește codul binar, aferent instrucțiunii de la adresa respectivă. Codurile binare ale instrucțiunilor au lungimi variabile. Pe a 3-a coloană se regăsește codul programului,

dezasamblat. Ar trebui să se observe aceleași instrucțiuni ca și în programul sursă, eventual scrise într-o altă ordine.

La depanarea unui program, următoarele comenzi sunt utilizate mai des:

- Step Into - **F7** - trece la instrucțiunea următoare a programului, intrând în funcții, acolo unde se întâlnesc.
- Step Over - **F8** - trece la instrucțiunea următoare, sărind peste funcții (se execută întreaga funcție, ca și cum ar fi o singură instrucțiune).
- Breakpoint - **F2** - atunci când cursorul se află pe o anumită linie (acea linie este evidențiată prin culoarea gri deschis), se plasează o întrerupere la acea linie. Atunci când programul, în timpul rulării, ajunge la o instrucțiune pe care s-a pus un breakpoint, se va întrerupe execuția.
- Run - **F9** - pornește execuția normală a programului, de la poziția curentă, până la primul breakpoint întâlnit, sau până la final.
- Execute till Return - **Ctrl+F9** - la fel ca Run, dar execuția se oprește și la întâlnirea unei instrucțiuni RETN.
- Restart - **Ctrl+F2** - repornește programul depanat.

Instrucțiunile sau datele unui program pot fi modificate în timpul depanării. Se face click pe instrucțiunea sau datele dorite, pentru a poziționa cursorul acolo, apoi se apasă tasta Space. În fereastra de dialog apărută, se pot face modificările.

Pentru a vizualiza întreg conținutul memoriei, se apasă combinația de taste **Alt+M**. În tabelul apărut este descrisă fiecare secțiune. Coloanele *Address* și *Size* indică adresa, respectiv dimensiunea secțiunii. În coloana *Owner*, apare numele modulului ce conține acea secțiune. Modulul principal are același nume ca și numele programului executabil. Coloana *Section* indică numele efectiv al secțiunii, în timp ce coloana *Contains* arată ce se găsește în aceasta.

Făcând dublu-click pe o linie a tabelului, se poate vizualiza conținutul secțiunii respective. Olly Debugger va încerca să "ghicească" tipul de conținut, afișând implicit datele în format hexazecimal și ASCII, respectiv dezasamblând codul. În cazul în care utilizatorul dorește vizualizarea informației în alt format, poate face click dreapta în zona de afișare, și alegerea formatului dorit (ca în Figura 2.7).

Pentru revenirea în mod CPU (fereastra principală, cu cele 4 zone), se apasă combinația de taste **Alt+C**.

2.4 Întrebări recapitulative

1. Care sunt cele 3 componente principale ale unui sistem de calcul?
2. În care din cele 3 componente ale unui sistem de calcul se află codul unui program care rulează?
3. Numiți cel puțin 2 caracteristici ale memoriei RAM.
4. Câtă memorie convențională se poate adresa pe un sistem pe 16 bit? Dar pe 32 bit?
5. Care sunt cei 8 regiștri de uz general?
6. Care este rolul registrului EIP?

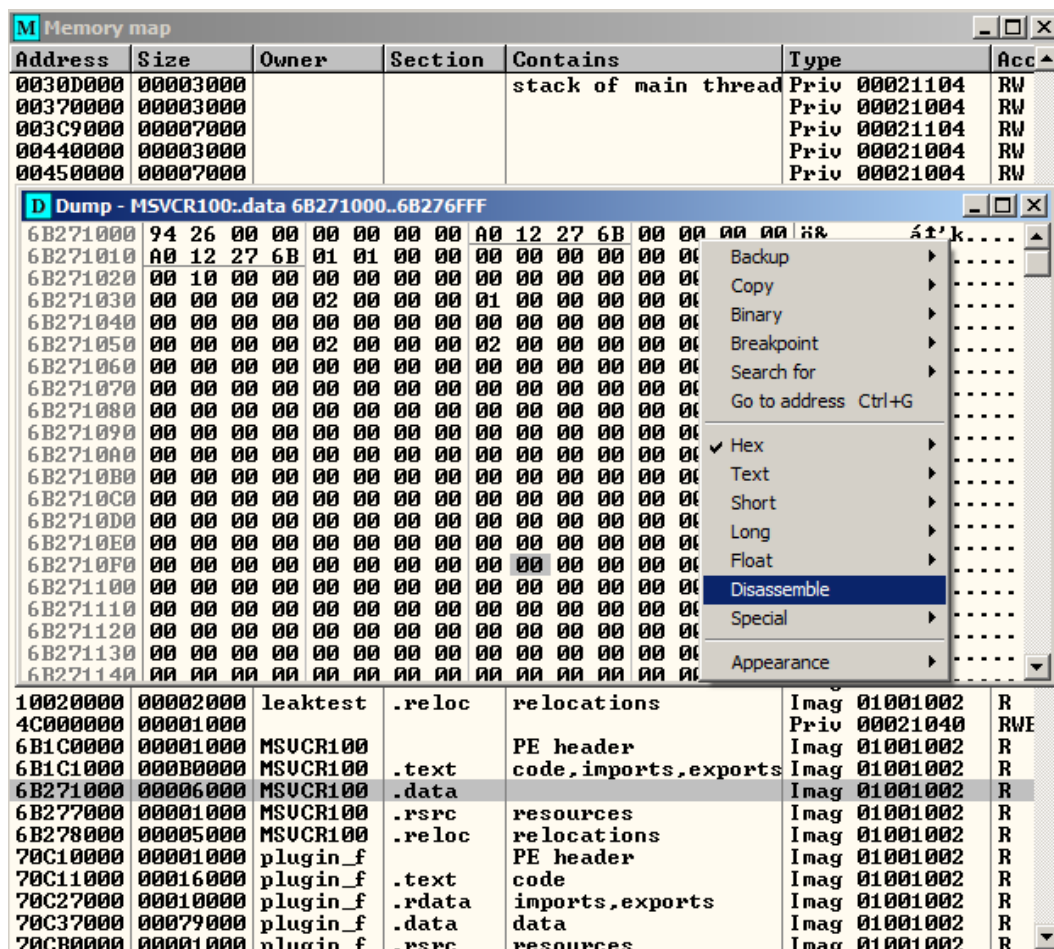


Figura 2.7: Harta memoriei în Olly Debugger

- Cum putem adresa biții 8...15, din registrul EBX? (bitul 0 e cel mai puțin semnificativ)
- Care sunt componentele registrului ECX? (sub-registrii adresabili separat)
- Cum se scrie un comentariu în limbaj de asamblare?
- Cum definim o constantă în limbaj de asamblare?

2.5 Mersul lucrării

- Discuții legate de arhitectura Intel x86.
- Prezentarea modului în care se execută o instrucțiune de către procesor.
- Se va deschide fișierul `s2model.asm` cu Notepad++. Fișierul conține un model de program în limbaj de asamblare. (Atenție: se recomandă folosirea versiunii de Notepad++ din folderul `asm_tools`, deoarece aceasta conține plugin-ul de MASM)
- Se va deschide fișierul `s2ex1.asm` cu Notepad++. Probleme de urmărit:
 - Modul în care sunt declarate datele, tipuri de date

Laborator 3

Setul de instrucțiuni al familiei de procesoare Intel x86: Sintaxa și clasele de instrucțiuni

3.1 Scopul lucrării

În cadrul acestei lucrări se prezintă o primă parte din instrucțiunile limbajului de asamblare al procesoarelor Intel x86 (varianta pe 32 bit). Restul instrucțiunilor vor fi prezentate în lucrarea următoare. În prima parte a lucrării se prezintă sintaxa generală a unei instrucțiuni de asamblare și câteva reguli de scriere a programelor. În partea practică a lucrării se vor scrie programe cu instrucțiunile prezentate și se va analiza efectul execuției acestora.

3.2 Prezentarea instrucțiunilor

Un limbaj de asamblare conține instrucțiuni corespunzătoare unor operații simple care sunt direct interpretate și executate de procesor. Fiecărei instrucțiuni din limbajul de asamblare îi corespunde în mod strict un singur cod executabil. În contrast, unei instrucțiuni dintr-un limbaj de nivel înalt (ex: C, Pascal, etc.) îi corespunde o secvență de coduri (instrucțiuni în cod mașină). Un anumit limbaj de asamblare este specific pentru un anumit procesor sau eventual pentru o familie de procesoare. Instrucțiunile sunt în directă corelație cu structura internă a procesorului. Un programator în limbaj de asamblare trebuie să cunoască această structură precum și tipurile de operații permise de structura respectivă.

Un program în asamblare scris pentru o anumită arhitectură de procesor nu este compatibil cu un alt tip de procesor. Pentru implementarea unei aplicații pe un alt procesor programul trebuie rescris. În schimb programele scrise în limbaj de asamblare sunt în general mai eficiente atât în ceea ce privește timpul de execuție cât și spațiul de memorie ocupat de program. De asemenea, programarea în limbaj de asamblare dă o mai mare flexibilitate și libertate în utilizarea resurselor unui calculator. Cu toate acestea astăzi utilizarea limbajului de asamblare este mai puțin frecventă deoarece eficiența procesului de programare este mai scăzută, există puține structuri de program și de date care să ușureze munca programatorului, iar programatorul trebuie să cunoască structura procesorului pentru care scrie aplicația. În plus programele nu sunt portabile, adică nu rulează și pe alte procesoare.

Un program scris în limbaj de asamblare conține instrucțiuni și directive. Instrucțiunile

sunt traduse în coduri executate de procesor; ele se regăsesc în programul executabil generat în urma compilării și a editării de legături. Directivele sunt construcții de limbaj ajutătoare care se utilizează în diferite scopuri (ex: declararea variabilelor, demarcarea secțiunilor și a procedurilor, etc.) și au rol în special în fazele de compilare și editare de legături. O directivă nu se traduce printr-un cod executabil și în consecință **nu** se execută de către procesor.

Sintaxa unei instrucțiuni în limbaj de asamblare

Așa cum s-a prezentat în capitolul anterior, o instrucțiune ocupă o linie de program și se compune din mai multe câmpuri, după cum urmează (parantezele drepte indică faptul că un anumit câmp poate să lipsească):

```
[<etichetă>:] [<operație> [<operandi>]] [;<comentariu>]
```

<etichetă> este un nume format din litere, cifre sau caractere speciale, care începe neapărat cu o literă sau cu un caracter special; este un nume simbolic (identificator) dat unei locații de memorie care conține instrucțiunea care urmează; scopul unei etichete este de a indica locul în care trebuie să se facă un salt în urma executării unei instrucțiuni de salt.

<operație> este reprezentată printr-o mnemonică, adică o combinație de litere care simbolizează o anumită instrucțiune (ex: add pentru adunare, mov pentru transfer, etc.); denumirile de instrucțiuni sunt cuvinte rezervate și nu pot fi utilizate în alte scopuri.

<operandi> pot fi zero, unu sau mai mulți, și reprezintă entitățile asupra cărora se efectuează operația. Pot fi regiștiri, locații de memorie, date imediate sau constante. De obicei primul operand reprezintă primul parametru și în același timp destinația instrucțiunii curente.

<comentarii> este un text explicativ care arată intențiile programatorului și efectul scontat în urma execuției instrucțiunii; având în vedere că programele scrise în limbaj de asamblare sunt mai greu de interpretat se impune aproape în mod obligatoriu utilizarea de comentarii; textul comentariului este ignorat de compilator; comentariul începe cu ';' și se considera până la sfârșitul liniei curente

Într-o linie de program nu toate câmpurile sunt obligatorii: poate să lipsească eticheta, parametrii, comentariul sau chiar instrucțiunea. Unele instrucțiuni nu necesită nici un parametru, altele au nevoie de unul sau doi parametri. În principiu primul parametru este destinația, iar al doilea este sursa.

Constantele numerice care apar în program se pot exprima în zecimal (modul implicit), în hexazecimal (constante terminate cu litera 'h') sau în binar (constante terminate cu litera 'b'). Constantele alfanumerice (coduri ASCII) se exprimă prin una sau mai multe litere între apostrof sau ghilimele.

3.2.1 Clase de instrucțiuni

În acest îndrumător de laborator nu se va face o prezentare exhaustivă a tuturor instrucțiunilor cu toate detaliile lor de execuție. Se vor prezenta acele instrucțiuni care se utilizează mai des și au importanță din punct de vedere al structurii și al posibilităților procesorului. Pentru alte detalii se pot consulta documentații complete referitoare la setul de instrucțiuni Intel x86. Referința canonică în acest sens este *“Intel 64 and IA-32 Architectures Software Developer Manuals”*, disponibilă pe Internet.

Instrucțiuni de transfer

Instrucțiunile de transfer realizează transferul de date între doi regiștri, între un registru și o locație de memorie sau încărcarea unei constante într-un registru sau locație de memorie. Transferurile de tip memorie-memorie nu sunt permise (cu excepția instrucțiunilor pe șiruri). La fel nu sunt permise transferurile directe între doi regiștri segment (selectori). Ambii parametri ai unui transfer trebuie să aibă aceeași lungime (număr de biți).

- instrucțiunea MOV

Este cea mai utilizată instrucțiune de transfer. Sintaxa este:

```
1 | mov <destinatie> <sursa>
```

unde:

```
<destinatie> := <registru> | <adr_offset> | <nume_var> | <expresie>
<sursa>      := <destinatie> | <constanta>
<registru>  := EAX|EBX|...|ESP|AX|BX|...|SP|AH|AL|BH|...|DL
<expresie>  := [<reg_index>[*<scalar>]][+<reg_baza>][+<deplasament>]
<reg_index> := EAX|EBX|ECX|EDX|ESI|EDI|EBP
<scalar>    := 1 | 2 | 4 | 8
<reg_bază>  := <registru>
<deplasament> := <constanta>
```

Exemple:

```
1 | mov EAX, EBX           1 | mov AH, [ESI+100h]
2 | mov CL, 12h           2 | eticheta1: mov AL, 'A'
3 | mov DX, var16         3 | mov SI, 1234h
4 | mov var32, EAX       4 | mov [ESI*4+EBX+30h], DX
5 | mov SI, BX           5 | sfarsit: mov ESI, [myvar]
```

Exemple de erori de sintaxă:

```
1 | mov AX, CL ;operanzi de lungimi diferite
2 | mov var1, var2 ;ambii operanzi sunt locatii de memorie
3 | mov AL, 1234h ;dimensiunea constantei este mai mare decat cea
   | a registrului
4 | mov [EAX+EBX+ECX], 10 ;deplasamentul nu este constant
```

- instrucțiunea LEA (*Load Effective Address*)

Se încarcă în registrul exprimat ca prim parametru adresa liniară a variabilei din parametru 2. Sintaxă:

```
1 | lea <parametru_1>, <parametru_2>
```

Exemple:

```
1 | lea ESI, var1           ;ESI ← offset(var1)
2 | lea EDI, [EBX+100]     ;EDI ← EBX+100
```

Instrucțiunea `lea` este echivalentă (are același efect) cu următoarea instrucțiune:

```
1 | mov <registru>, offset <memorie>
```

Utilizând `lea`, putem folosi circuitul de calcul al adreselor din procesor pentru a efectua operații aritmetice din mai puțini pași. Fără `lea`, operația din a doilea exemplu s-ar fi efectuat în doi pași, un `mov`, urmat de un `add`.

- instrucțiunea XCHG (*eXCHanGe*)

Această instrucțiune interschimbă conținutul celor doi operanzi.

```
1 | xchg <parametru_1>, <parametru_2>
```

Atenție: parametrii nu pot fi constante.

Exemple:

```
1 | xchg AL, BH
2 | xchg EBP, var32
```

- instrucțiunile PUSH și POP

Cele două instrucțiuni operează în mod implicit cu vârful stivei. Instrucțiunea push pune un operand pe stivă, iar pop extrage o valoare de pe stivă și o depune într-un operand. În ambele cazuri registrul indicator de stivă (ESP) se modifică corespunzător (prin decrementare, respectiv incrementare) astfel încât registrul ESP să indice poziția curentă a vârfului de stivă. Sintaxa instrucțiunilor este:

```
1 | push <parametru>
2 | pop <parametru>
```

Operandul trebuie să fie o valoare pe 16 sau 32 biți, dar **este recomandat să se folosească doar valori pe 32 de biți**. Aceste instrucțiuni sunt utile pentru salvarea temporară și refacerea conținutului unor registre. Aceste operații sunt necesare mai ales la apelul de rutine și la revenirea din rutine. În cazul introducerii în stivă, prima operație care se realizează este decrementarea indicatorului de stivă ESP cu 4 (la introducerea unei valori pe 32 biți) sau 2 (la introducerea unei valori pe 16 biți), urmată de memorarea operandului conform acestui indicator. În cazul extragerii din stivă prima operație care se realizează este citirea operandului conform indicatorului de stivă urmată de incrementarea cu 4 sau 2 a indicatorului.

Exemple:

```
1 | push EAX; echivalent cu:
2 |     ;sub ESP, 4
3 |     ;mov [ESP], EAX
4 | pop var32; echivalent cu:
5 |     ;mov var32, [ESP]
6 |     ;add ESP, 4
```

Instrucțiuni de transfer pentru indicatorii de condiție

În setul de instrucțiuni al microprocesorului x86 există instrucțiuni pentru încărcarea și memorarea indicatorilor de condiție. Acestea sunt:

- LAHF
- SAHF
- PUSHF
- POPF

Octetul mai puțin semnificativ al registrului indicatorilor de condiție poate fi încărcat în registrul AH folosind instrucțiunea lahf, respectiv poate fi înscris cu conținutul registrului AH folosind instrucțiunea sahf. Structura octetului care se transferă este următoarea:

bitul	7	6	5	4	3	2	1	0
	SF	ZF		AF		PF		CF

Dacă se dorește salvarea sau refacerea întregului registru al indicatorilor de condiție se folosesc instrucțiunile `pushf` și `popf` care salvează și recuperează de pe stivă întreg registrul `EFLAGS`, cu structura din Figura 2.2.

Instrucțiuni aritmetice

Aceste instrucțiuni efectuează cele patru operații aritmetice de bază: adunare, scădere, înmulțire și împărțire. Rezultatul acestor instrucțiuni afectează starea indicatorilor de condiție.

- instrucțiunile `ADD` și `ADC`

Aceste instrucțiuni efectuează operația de adunare a doi operanzi, rezultatul plasându-se în primul operand. A doua instrucțiune `adc` (add with carry) adună și conținutul indicatorului de transport `CF`. Această instrucțiune este utilă pentru implementarea unor adunări în care operanzii sunt mai lungi de 32 de biți.

```

1 | add <dest>, <sursa> ; <dest> ← <dest> + <sursa>
2 | adc <dest>, <sursa> ; <dest> ← <dest> + <sursa> + CF

```

Exemple:

```

1 | add EAX, 123456h
2 | add BX, AX
3 | adc DL, var8

```

- instrucțiunile `SUB` și `SBB`

Aceste instrucțiuni implementează operația de scădere. A doua instrucțiune, `sbb` (subtract with borrow) scade și conținutul indicatorului `CF`, folosit în acest caz pe post de bit de împrumut. Ca și `adc`, `sbb` se folosește pentru operanzi de lungime mai mare.

```

1 | sub <dest>, <sursa> ; <dest> ← <dest> - <sursa>
2 | sbb <dest>, <sursa> ; <dest> ← <dest> - <sursa> - CF

```

- instrucțiunile `MUL` și `IMUL`

Aceste instrucțiuni efectuează operația de înmulțire, `mul` pentru întregi fără semn și `imul` pentru întregi cu semn. De remarcat că la operațiile de înmulțire și împărțire trebuie să se țină cont de forma de reprezentare a numerelor (cu semn sau fără semn), pe când la adunare și scădere acest lucru nu este necesar. Pentru a evita dese depășiri de capacitate s-a decis ca rezultatul operației de înmulțire să se păstreze pe o lungime dublă față de lungimea operanzilor. Astfel dacă operanzii sunt pe octet rezultatul este pe cuvânt, iar dacă operanzi sunt pe cuvânt rezultatul este pe dublu-cuvânt. De asemenea se impune ca primul operand și implicit și rezultatul să se păstreze în registrul acumulator. De aceea primul operand nu se mai specifică.

```

1 | mul <operand_1>
2 | imul <operand_1>
3 | imul <operand_1>, <operand_2>
4 | imul <operand_1>, <operand_2>, <valoare_imediata>

```

Exemple:

```

1 | mul DH ;AX ← AL × DH
2 | mul EBX ;EDX:EAX ← EAX × EBX
3 | imul var8 ;AX ← AL × var8

```

Instrucțiunea `mul` are un singur operand explicit, ceilalți fiind implicați. În funcție de dimensiunea operandului explicit, înmulțirea are loc astfel:

```

1 | mul <op_8bit> ;AX ← AL × <op_8bit>
2 | mul <op_16bit> ;DX:AX ← AX × <op_16bit>
3 | mul <op_32bit> ;EDX:EAX ← EAX × <op_32bit>

```

Prin notația `DX:AX` înțelegem o valoare pe 32 bit, în care partea mai semnificativă o reprezintă registrul `DX`, iar partea mai puțin semnificativă registrul `AX`. Analog, `EDX:EAX` reprezintă o valoare pe 64 bit, obținută prin concatenarea regiștrilor `EDX` și `EAX`. Operandul explicit poate fi un registru sau o variabilă, dar nu poate fi o valoare imediată (constantă).

Aceleași reguli se aplica și la instrucțiunea `imul` cu 1 operand. La instrucțiunea `imul` cu 2 operanzi, rezultatul înmulțirii celor 2 operanzi este păstrat în primul operand. În cazul utilizării instrucțiunii `imul` cu 3 operanzi, primul operand păstrează rezultatul înmulțirii între al doilea operand și valoarea imediată.

- instrucțiunile `DIV` și `IDIV`

Aceste instrucțiuni efectuează operația de împărțire pe întregi fără semn și respectiv cu semn. Pentru a crește plaja de operare se consideră că primul operand, care în mod obligatoriu trebuie să fie în acumulator, are o lungime dublă față de al doilea operand. Primul operand nu se specifică.

```

1 | div <operand_>
2 | idiv <operand>

```

Exemple:

```

1 | div CL ;AL ← AX / CL, AH ← AX % CL (restul impartirii)
2 | div SI ;AX ← DX:AX / SI, DX ← DX:AX % SI

```

Instrucțiunile `div` și `idiv` au un singur operand explicit, ceilalți fiind implicați. În funcție de dimensiunea operandului explicit, împărțirea are loc astfel:

```

1 | div <op_8bit>
2 | ;AL ← AX / <op_8bit>, AH ← AX % <op_8bit>
3 | div <op_16bit>
4 | ;AX ← DX:AX / <op_16bit>, DX ← DX:AX % <op_16bit>
5 | div <op_32bit>
6 | ;EAX ← EDX:EAX / <op_32bit>, EDX ← EDX:EAX % <op_32bit>

```

Prin împărțirea unui număr mare la un număr mic, există posibilitatea ca rezultatul să depășească capacitatea de reprezentare. În acest caz, se va declanșa aceeași eroare ca și la împărțirea cu 0.

Atenție: cea mai frecventă eroare la utilizarea instrucțiunii `div` este neglijarea registrului `DX` (la utilizarea unui operand pe 16 bit) sau a registrului `EDX` (la utilizarea unui operand pe 32 bit). Consecința este că deîmpărțitul va conține și registrul neglijat, deci va avea o altă valoare decât se așteaptă programatorul. Pentru a evita acest lucru, setați registrul `DX` sau `EDX` pe 0, înaintea efectuării împărțirii.

- instrucțiunile `INC` și `DEC`

Aceste instrucțiuni realizează incrementarea și respectiv decrementarea cu o unitate

a operandului. Aceste instrucțiuni sunt eficiente ca lungime și ca viteză. Ele se folosesc pentru contorizare și pentru parcurgerea unor șiruri prin incrementarea sau decrementarea adreselor.

```
1 | inc <param> ; <param> ← <param> + 1
2 | dec <param> ; <param> ← <param> - 1
```

Exemple:

```
1 | inc ESI ; ESI ← ESI + 1
2 | dec var1 ; var1 ← var1 - 1
```

- instrucțiunea **CMP** Această instrucțiune compară cei doi operanzi prin scăderea lor. Rezultatul scăderii nu se memorează. Instrucțiunea are efect numai asupra următorilor indicatori de condiție: ZF, SF, OF, CF. Valorile indicatorilor de condiție pot fi apoi interpretate în mod diferit dacă valorile comparate au fost cu semn sau fără semn. Această instrucțiune precede de obicei o instrucțiune de salt condiționat. Printr-o combinație de instrucțiune de comparare și o instrucțiune de salt se pot verifica relații de egalitate, mai mare, mai mic, mai mare sau egal, etc.

```
1 | cmp <param_1>, <param_2>
```

Exemplu:

```
1 | cmp AX, 50
```

Instrucțiuni logice

Aceste instrucțiuni implementează operațiile de bază ale logicii booleene. Operațiile logice se efectuează la nivel de bit, adică se combină printr-o operație logică fiecare bit al operandului 1 cu bitul corespunzător din operandul al doilea. Rezultatul se generează în primul operand.

- instrucțiunile **AND**, **OR**, **NOT** și **XOR**
Aceste instrucțiuni implementează cele patru operații de bază: *și*, *SAU*, *Negație* și *SAU-Exclusiv*.

```
1 | and <dest>, <sursa>
2 | or <dest>, <sursa>
3 | not <param>
4 | xor <dest>, <sursa>
```

Exemple:

```
1 | and AL, 0Fh
2 | or BX, 0000111100001111b
3 | and AL, CH
4 | xor EAX, EAX ; sterge conținutul lui EAX
```

- instrucțiunea **TEST**
Această instrucțiune efectuează operația *și logică* fără a memora rezultatul. Scopul operației este de a modifica indicatorii de condiție. Instrucțiunea evită distrugerea conținutului primului operand.

```
1 | test <param_1>, <param_2>
```

Exemple:

```

1 | test AL, 00010000b ;se verifica daca bitul 4 din AL este
   |   setat
2 | test BL, 0Fh ;se verifica daca cifra hexazecimala cea mai
   |   putin semnificativa din BL este 0

```

Instrucțiunile de deplasare și rotire

- instrucțiunile SHL (SAL), SHR și SAR

Aceste instrucțiuni realizează deplasarea (eng. *shift*) la stânga respectiv la dreapta a conținutului unui operand. La deplasarea "logică" (`shl`, `shr`) biții se copiază în locațiile învecinate (la stânga sau la dreapta), iar pe locurile rămase libere se înscrie 0 logic. La deplasarea "aritmetică" (`sal`, `sar`) se consideră că operandul conține un număr cu semn, iar prin deplasare la stânga și la dreapta se obține o multiplicare și respectiv o divizare cu puteri ale lui doi (ex: o deplasare la stânga cu 2 poziții binare este echivalent cu o înmulțire cu 4). La deplasarea la dreapta se dorește menținerea semnelui operandului, de aceea bitul mai semnificativ (semnul) se menține și după deplasare. La deplasarea la stânga acest lucru nu este necesar, de aceea instrucțiunile `shl` și `sal` reprezintă aceeași instrucțiune.

În Figura 3.1 s-a reprezentat o deplasare logică și o deplasare aritmetică la dreapta. Se observă că bitul care iese din operand este înscris în indicatorul de transport CF. Formatul instrucțiunilor:

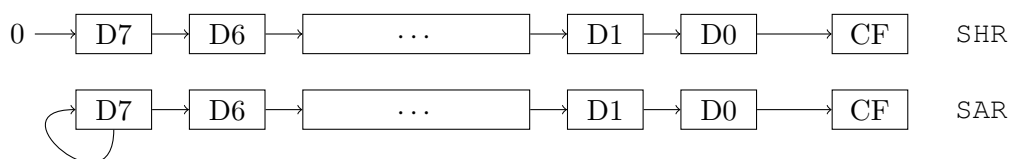


Figura 3.1: Instrucțiunile `shr` și `sar`

```

1 | shl <param>, <nr_bit>
2 | sal <param>, <nr_bit>
3 | shr <param>, <nr_bit>
4 | sar <param>, <nr_bit>

```

Primul parametru este în concordanță cu definițiile anterioare; al doilea parametru specifică numărul de poziții binare cu care se face deplasarea și poate fi specificat ca valoare imediată (constantă) sau prin registrul CL.

Exemple:

```

1 | shl AX, 1 ;AX ← AX × 2
2 | sar EBX, 3 ;EBX ← EBX / 23
3 | shr var, CL ;var ← var / 2CL

```

- instrucțiunile de rotire ROR, ROL, RCR și RCL

Aceste instrucțiuni realizează rotirea la dreapta sau la stânga a operandului, cu un număr de poziții binare. Diferența față de instrucțiunile anterioare de deplasare constă în faptul că în poziția eliberată prin deplasare se introduce bitul care iese din operand. Rotirea se poate face cu implicarea indicatorului de transport (CF) în

procesul de rotație (*rcr*, *rcl*) sau fără (*ror*, *rol*). În ambele cazuri bitul care iese din operand se regăsește în indicatorul de transport CF. Figura 3.2 indică cele două moduri de rotație pentru o rotație la dreapta.

Formatul instrucțiunilor:

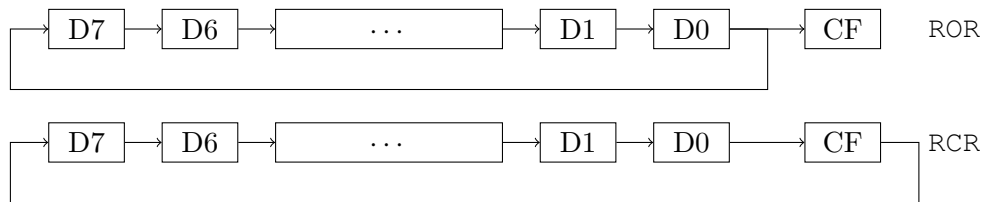


Figura 3.2: Instrucțiunile *ror* și *rcr*

```

1 | ror <param>, <nr_bit>
2 | rol <param>, <nr_bit>
3 | rcr <param>, <nr_bit>
4 | rcl <param>, <nr_bit>

```

Referitor la parametri, se aplică aceleași observații ca și la instrucțiunile de deplasare.

Instrucțiuni de intrare/ieșire

Aceste instrucțiuni se utilizează pentru efectuarea transferurilor cu registrele (porturile) interfețelor de intrare/ieșire. Trebuie remarcat faptul că la procesoarele Intel acestea sunt singurele instrucțiuni care operează cu porturi.

- instrucțiunile *IN* și *OUT*

Instrucțiunea *in* se folosește pentru citirea unui port de intrare, iar instrucțiunea *out* pentru scrierea unui port de ieșire. Sintaxa instrucțiunilor este:

```

1 | in <acumulator>, <adr_port>
2 | out <adr_port>, <acumulator>

```

unde:

- <acumulator> - registrul EAX/AX/AL pentru transfer pe 32/16/8 bit
- <adr_port> - o adresă exprimabilă pe 8 bit sau registrul DX

Se observă că dacă adresa portului este mai mare decât 255 atunci adresa portului se transmite prin registrul DX.

Exemple:

```

1 | in AL, 20h
2 | mov DX, adr_port
3 | out DX, AX

```

Instrucțiuni speciale

În această categorie s-au inclus acele instrucțiuni care au efect asupra modului de funcționare al procesorului.

- instrucțiunile *CLC*, *STC* și *CMC*

Aceste instrucțiuni modifică starea indicatorului CF de transport. Au următoarele efecte:

- `clc` (*CLear Carry*) - $CF \leftarrow 0$
 - `stc` (*SeT Carry*) - $CF \leftarrow 1$
 - `cmc` (*CoMplement Carry*) - $CF \leftarrow \overline{CF}$ (se inversează indicatorul CF)
- instrucțiunile `CLI` și `STI`
 Aceste instrucțiuni șterg respectiv setează indicatorul de întrerupere IF. În starea setată (IF=1) procesorul detectează întreruperile mascabile, iar în starea inversă blochează toate întreruperile mascabile.
 - instrucțiunile `CLD` și `STD`
 Aceste instrucțiuni modifică starea indicatorului de direcție DF. Prin acest indicator se controlează modul de parcurgere a șirurilor la operațiile pe șiruri: prin incrementare (DF=0) sau prin decrementare (DF=1).
 - instrucțiunea `CPUID` Această instrucțiune permite identificarea tipului de procesor pe care rulează programul.

3.3 Întrebări recapitulative

1. Care din următoarele instrucțiuni sunt greșite? De ce?

- | | |
|-----------------------------------|----------------------------------|
| (a) <code>mov EAX, 12</code> | (e) <code>mov AL, 1234</code> |
| (b) <code>add EBX, 101010b</code> | (f) <code>mov BX, 12</code> |
| (c) <code>mov ECX, A12h</code> | (g) <code>add ECX, 10001h</code> |
| (d) <code>add EDX, 1234</code> | (h) <code>add EDX, EBX</code> |

2. Ce valoare va fi în registrul EAX după secvențele:

<pre>1 xor EBX, EBX 2 add EBX, 5 3 lea EAX, [EBX+3]</pre>	<pre>1 lea EAX, [ESI+2] 2 sub EAX, ESI</pre>
---	--

3. Dacă în starea inițială EAX=1, EBX=2, ECX=3, ce valori vor avea regiștrii, după secvențele:

<pre>1 push EAX 2 push EBX 3 push ECX 4 pop EAX 5 pop EBX 6 pop ECX</pre>	<pre>1 push EBX 2 push EAX 3 pop EBX 4 push ECX 5 pop EBX 6 pop EAX</pre>
---	---

4. Explicați ce efect au instrucțiunile:

- (a) `div CX`
- (b) `mul EBX`

5. Ce valoare va fi în registrul AL după instrucțiunile:

1	mov AL, 5	1	mov AL, 91h
2	shl AL, 2	2	ror AL, 3

3.4 Mersul lucrării

3.4.1 Probleme rezolvate

Pentru fiecare problemă de mai jos, se va compila programul dat, se va executa în Olly Debugger și se vor urmări schimbările care au loc la nivelul regiștrilor, memorie și a flagurilor.

1. (s3ex1.asm) Să se implementeze în limbaj de asamblare expresia de mai jos folosind instrucțiuni de deplasare:

$$EAX = 7 * EAX - 2 * EBX - EBX / 8$$

2. (s3ex2.asm) Să se scrie un program în limbaj de asamblare care generează un întreg reprezentabil pe octet și îl pune în locația de memorie rez după formula:

$$rez = AL * num1 + num2 * AL + BL$$

rez, num1 și num2 sunt valori reprezentate pe octet, aflate în memorie.

3.4.2 Probleme propuse

1. Să se implementeze în limbaj de asamblare expresia de la problema rezolvată 1, folosind instrucțiuni aritmetice.
2. Să se scrie un program în limbaj de asamblare care generează un întreg reprezentabil pe cuvânt și îl pune în locația de memorie rez după formula:

$$rez = AX * num1 + num2 * (AX + BX)$$

rez, num1 și num2 sunt valori reprezentate pe cuvânt, aflate în memorie.

3. Instrucțiunea `bswap` (*Byte Swap*) inversează octeții unui registru. Dacă în EAX avem valoarea 12345678h, prin apelul `bswap EAX`, registrul va conține 78563412h. Implementați această operație folosind instrucțiuni de rotație (`rol`, `ror`) și interschimbare (`xchg`).
4. Dându-se o variabilă x de tip DWORD aflată în memorie, să se scrie un program care pune în registrul EAX valoarea 0 dacă și numai dacă x este o putere întreagă a lui 2.
Indiciu: 78 20 26 20 28 78 20 2D 20 31 29

Laborator 4

Modurile de adresare ale procesorului Intel x86

4.1 Scopul lucrării

În cadrul acestei lucrări se prezintă modurile de adresare permise de procesorul Intel x86. Partea aplicativă a lucrării are drept scop exersarea acestor mecanisme de adresare și identificarea celor mai adecvate soluții de adresare a diferitelor structuri de date.

4.2 Prezentarea modurilor de adresare

Prin ”mod de adresare” se înțelege un anumit mecanism folosit pentru determinarea operanzilor unei instrucțiuni. Așa cum se va vedea în continuare, un anumit operand care participă la execuția unei instrucțiuni se poate regăsi în diferite feluri: poate fi o constantă, un registru intern al procesorului, o locație de memorie sau un port al unei interfețe. Un anumit mod de adresare indică mecanismul de căutare a operandului și eventual modul de calcul a adresei, dacă operandul se află în memorie.

Diferitele moduri de adresare oferă suportul necesar pentru regăsirea elementelor unei structuri mai complexe de date. De exemplu adresarea indexată permite adresarea elementelor unui vector, sau adresarea bazată permite extragerea unor date dintr-o structură de tip înregistrare.

Mecanismul de adresare folosit influențează viteza de execuție a instrucțiunilor și determină lungimea acestora. Astfel mecanismele complexe de adresare necesită un timp de execuție mai mare însă oferă o mai mare flexibilitate în regăsirea datelor.

4.2.1 Adresarea imediată

Adresarea imediată este cea mai simplă formă de adresare. Operandul este o constantă, care se păstrează în codul instrucțiunii. Astfel, odată cu citirea instrucțiunii are loc și citirea operandului. Constanta se poate exprima în zecimal (este forma implicită), în hexazecimal (cu terminația *h*), în binar (terminația *b*) sau sub formă de coduri ASCII. Constanta este întotdeauna al doilea operand al unei instrucțiuni.

Exemple:

```
1 | mov EAX, 1234h
2 | add CX, 30
3 | and BH, 01111b
```

Acest mod de adresare este relativ rapid deoarece nu necesită transfer suplimentar pentru aducerea operandului. dar flexibilitatea este limitată, în sensul că o instrucțiune operează cu o singură valoare.

4.2.2 Adresarea de tip registru

La această adresare operandul se află într-un registru al procesorului. Acest mod este de fapt o formă mai eficientă de adresare directă. Eficiența se datorează mai multor factori:

- regiștrii sunt în interiorul procesorului ceea ce elimină necesitatea unui transfer suplimentar cu memoria
- adresa unui registru se exprimă pe un număr redus de biți (8 regiștri - 3 biți), ceea ce contribuie la reducerea dimensiunii instrucțiunilor și implicit la o execuție mai rapidă a acestora
- transferurile între regiștri se fac la o viteză mult mai mare deoarece se utilizează magistralele interne ale procesorului

Exemple:

```
1 | mov AX, BX
2 | cmp AH, BH
```

Dezavantajul acestui mod constă în faptul că numărul de regiștri interni este limitat și din această cauză nu toate variabilele unui program pot fi păstrate în regiștri.

4.2.3 Adresarea directă

Adresarea directă presupune prezența adresei operandului în codul instrucțiunii. Operandul este o locație de memorie sau cu alte cuvinte o variabilă. Adresa operandului se poate exprima printr-o valoare sau printr-un nume simbolic dat variabilei. Pentru a evita confuzia cu adresarea imediată, valoarea adresei se plasează între paranteze pătrate. De altfel ori de câte ori se folosesc paranteze pătrate, conținutul lor trebuie interpretat ca și o adresă și nu ca o constantă.

Exemple:

```
1 | mov AX, [403000h] ;403000h este adresa de offset a operandului
2 | add DS:[405001h], CH
3 | cmp DX, var16 ;var16 trebuie declarat in prealabil ca variabila pe
   | 16 bit
```

Adresarea directă permite accesarea unei singure locații de memorie; o altă locație necesită o altă instrucțiune.

4.2.4 Moduri de adresare indirectă

În cazul modurilor de adresare indirectă instrucțiunea nu conține operandul sau adresa acestuia ci o indicație asupra locului unde se află adresa operandului. De cele mai multe ori adresa se păstrează într-un registru sau se calculează ca o sumă de regiștri și eventual o constantă. Deși inițial, la procesoarele 8086 (pe 16 bit) exista un număr restrâns de regiștri care puteau fi utilizați pentru a păstra adrese, și anume SI, DI pentru adresarea indexată și BX, BP pentru adresare bazată, începând cu procesoarele 80386 pot fi folosiți pentru adresare toți regiștrii generali pe 32 bit, EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP. Singura excepție este că ESP nu se poate folosi ca index.

Modurile de adresare indirectă la procesoarele Intel x86 sunt: adresarea indexată, adresarea bazat-indexată și adresarea scalat-indexată. Schema generală de adresare este reprezentată în Figura 4.1.

$$\begin{pmatrix} CS \\ DS \\ SS \\ ES \\ FS \\ GS \end{pmatrix} : \left[\begin{pmatrix} EAX \\ EBX \\ ECX \\ EDX \\ ESI \\ EDI \\ EBP \\ ESP \end{pmatrix} \right] + \left[\begin{pmatrix} EAX \\ EBX \\ ECX \\ EDX \\ ESI \\ EDI \\ EBP \end{pmatrix} * \begin{pmatrix} 1 \\ 2 \\ 4 \\ 8 \end{pmatrix} \right] + [\text{deplasament}]$$

Figura 4.1: Adresarea indirectă

Adresarea indexată

Adresarea indexată permite utilizarea unui registru pe 32 bit (indexul) și a unei constante (reprezentând deplasamentul) pentru calculul adresei. Deplasamentul indică adresa de început a vectorului, iar registrul index poziția relativă a elementului față de adresa de început. Acest mod de adresare este utilizat pentru regăsirea elementelor unei structuri de tip vector, prin simpla incrementare sau decrementare a unui registru denumit registru index. O singură instrucțiune poate prelucra toate elementele unui vector.

Exemple:

```

1 | mov AX, [ESI]
2 | mov byte ptr var[ESI], 30h ;trebuie specificat ca ne referim la
   |   byte-ul de la adresa respectiva si nu la word sau dword
3 | mov [EDI], CL
4 | mov EAX, [EBP]
5 | sub DX, [var + ECX]
```

	v(0)	v(1)	v(2)	v(3)	v(4)
adresa	403000h	403001h	403002h	403003h	403004h
		↙			
	<code>add AH, [ESI + 403000h]</code>				

Trecerea la elementul următor din vector se face prin incrementarea sau decrementarea explicită a registrului index. Dacă elementele vectorului sunt octeți atunci factorul de incrementare este 1 iar dacă sunt cuvinte atunci factorul este 2, respectiv 4 pentru dublu-cuvinte.

Acest mod de adresare este mai puțin eficient din punct de vedere al vitezei de execuție deoarece implică un calcul matematic și o adresare suplimentară a memoriei. În schimb este o metodă flexibilă de adresare a structurilor de tip vector sau tablou. În cazul unor prelucrări în buclă aceeași instrucțiune adresează succesiv toate elementele unui vector.

Adresarea bazat-indexată

Adresarea bazat-indexată permite utilizarea a doi regiștri pe 32 bit (baza și indexul) și a unei constante (reprezentând deplasamentul) pentru calculul adresei. Primul registru

utilizat este registrul bază, al doilea este registrul index. Este permis să se folosească același registru atât ca baza cât și ca index.

Acest mod se utilizează în cazul unor structuri de date complexe de tip vectori de înregistrări, înregistrări de vectori, tabele bidimensionale, etc. Este o formă mai flexibilă de adresare dar în același timp ineficientă. Adresarea operandului implică două adunări și un transfer suplimentar cu memoria.

Exemple:

```

1 | mov AX, [EBX][ESI]
2 | sub var[EBP+ESI], CH
3 | add EDX, [EAX+ECX+403000h]

```

Adresarea indexat-scalată

Adresarea indexat-scalată permite utilizarea a doi regiștri pe 32 bit (baza și indexul) și a unei constante (reprezentând deplasamentul) pentru calculul adresei. În acest mod de adresare este permisă înmulțirea registrului index cu o valoare egală cu 1, 2, 4 sau 8.

Exemple:

```

1 | mov AX, [ESI*2]
2 | sub 401000h[EBX][EDI*8], CH
3 | add EAX, 402000h[ESI*4]

```

4.2.5 Adresarea pe șiruri

Adresarea pe șiruri este o formă specială de adresare indexată. La această adresare registrele index folosite nu se precizează, ele fiind definite în mod implicit: ESI pentru șirul sursă și EDI pentru șirul destinație. În plus registrul ECX este folosit pe post de contor. La execuția instrucțiunii are loc modificarea automată (prin incrementare sau decrementare) a regiștrilor index astfel încât să se treacă automat la elementele următoare din șir. Prin utilizarea prefixului de repetare (`rep`, `repz`, ...) se obține transferul sau prelucrarea unui bloc de date printr-o singură instrucțiune. Este o metodă elegantă de lucru cu structuri de date de tip șir sau vector.

Exemplu: codul de mai jos copiază conținutul din `sir1` în `sir2`.

```

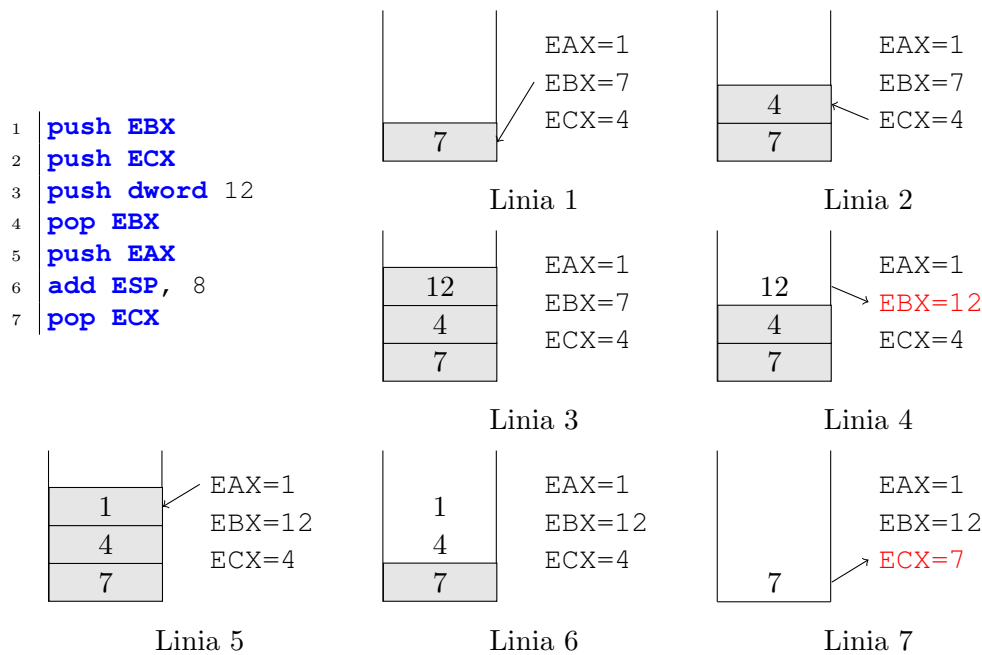
1 | lea ESI, sir1
2 | lea EDI, sir2
3 | mov ECX, lung_sir
4 | rep movsb

```

4.2.6 Adresarea de tip stivă

Adresarea de tip stivă utilizează în mod implicit registrul ESP pentru adresarea unuia dintre operanzi. Acest mod de adresare este folosit numai la 2 instrucțiuni, cele care operează cu vârful stivei: `push` și `pop`. În urma transferului, registrul ESP se modifică în așa fel încât la următoarea operație să se adreseze din nou vârful stivei. La arhitectura Intel x86 stiva crește către adrese mai mici (ESP se decrementează la salvarea pe stivă) și descrește către adrese mai mari (ESP se incrementează la descărcarea unui element de pe stivă). Este recomandat ca instrucțiunile `push` și `pop` să se utilizeze doar cu valori pe 32 bit, pentru a păstra alinierea stivei.

Exemplu: Știind că în starea inițială avem EAX=1, EBX=7, ECX=4, cum vor arăta regiștrii și stiva după următoarea secvență de instrucțiuni?



4.3 Întrebări recapitulative

1. Ce moduri de adresare se folosesc în instrucțiunile de mai jos?

- | | |
|-------------------------------------|------------------------------|
| (a) <code>mov EAX, var1</code> | (c) <code>mov AX, 120</code> |
| (b) <code>mov EBX, [403000h]</code> | (d) <code>mov AL, BH</code> |

2. Care din următoarele instrucțiuni sunt greșite? De ce?

- | | |
|---------------------------------------|---------------------------------------|
| (a) <code>mov EAX, [EAX+EBX]</code> | (e) <code>mov EAX, EDX+ESI</code> |
| (b) <code>mov EBX, EBP</code> | (f) <code>mov EBX, [ECX+AL]</code> |
| (c) <code>lea ECX, [EDX+ESI+3]</code> | (g) <code>mov EDX, [EBX+4*EDI]</code> |
| (d) <code>mov EDX, [EBX+3*EDI]</code> | (h) <code>mov EAX, [EAX+EBX+1]</code> |

3. Definiți și exemplificați:

- Adresarea indexat-scalată
- Adresarea bazat-indexată

4. Dacă în starea inițială $EAX=1$, $EBX=2$, $ECX=3$, ce valori vor avea regiștrii, după secvențele:

```

1  push EAX
2  push ECX
3  push EBX
4  add ESP, 8
5  pop ECX

```

```

1  push EAX
2  push ECX
3  push EBX
4  pop EAX
5  add ESP, 4
6  pop ECX

```

5. Cu ce șir de instrucțiuni sunt echivalente instrucțiunile de mai jos?

(a) `movsb`

(b) `push`

4.4 Mersul lucrării

4.4.1 Probleme rezolvate

Pentru fiecare problemă de mai jos, se va compila programul dat, se va executa în Olly Debugger și se vor urmări schimbările care au loc la nivelul regiștrilor, memorie și a flag-urilor.

1. (`s4ex1.asm`) Să se scrie un program care copiază un șir de valori din locații consecutive de memorie în alta locație, în ordine inversă.
2. (`s4ex2.asm`) Să se scrie un program pentru adunarea a două matrici bidimensionale.

4.4.2 Probleme propuse

1. Să se scrie un program care calculează media unui șir de numere întregi din memorie. Numerele sunt de tip octet. Media va fi memorată ca valoare întreaga într-o variabilă de tip octet.
2. Să se scrie un program care însumează salariile unor angajați. Datele referitoare la angajați se păstrează într-un tablou de înregistrări.

Exemplu tablou de înregistrări:

```

1 ANGAJAT struct
2     nume DB 20 dup(0)
3     salariu DW 0
4 ANGAJAT ends
5
6 nrAngajati DW 4
7 angajati ANGAJAT {"georgescu", 100}, {"pop", 100},
8                  {"popescu", 300}, {"ionescu", 450}

```

3. Folosind doar instrucțiunile `push` și `pop`, scrieți un program care rotește regiștrii `EAX`, `EBX`, `ECX` și `EDX` (valoarea din `EAX` merge în `EBX`, din `EBX` în `ECX`, din `ECX` în `EDX` iar din `EDX` în `EAX`).
4. Într-un șir de $2n - 1$ elemente se găsesc într-o ordine aleatoare toate numerele de la 1 la n , de două ori fiecare, mai puțin un număr k care se găsește o singură dată. Scrieți un program care găsește numărul k .

Laborator 5

Setul de instrucțiuni al familiei de procesoare Intel x86: Controlul fluxului de instrucțiuni

5.1 Scopul lucrării

Microprocesoarele din familia x86 au o largă varietate de instrucțiuni care permit controlul fluxului de instrucțiuni. Ele se împart în patru categorii: instrucțiuni de salt, de ciclare, de apel a procedurilor și de întrerupere.

5.2 Considerații teoretice

5.2.1 Instrucțiuni de salt

Saltul este metoda cea mai directă de modificare a fluxului de instrucțiuni. La nivel intern, instrucțiunile de salt lucrează prin schimbarea valorii registrului EIP, astfel încât adresa instrucțiunii următoare existentă în acest registru să fie schimbată cu adresa destinație.

Saltul necondiționat

Instrucțiunea JMP este folosită pentru efectuarea unui salt necondiționat la o adresă specificată pe 8/16/32 bit. Sintaxa este următoarea:

```
1 | jmp target
```

Din punct de vedere al modului de specificare a adresei destinației există salturi directe și indirecte. În cazul salturilor directe, adresa destinație este specificată printr-o etichetă.

Exemplu:

```
1 | alfa:  
2 |     ...  
3 |     jmp alfa
```

În cazul salturilor indirecte, adresa destinație se specifică printr-un operand, sintaxa fiind următoarea:

```
1 | jmp {registru | memorie}
```

Exemple:

```

1 | jmp EAX
2 | jmp [EBX]
3 | jmp var32

```

Un salt necondiționat poate fi folosit ca o formă de salt condiționat dacă adresa destinație este specificată într-un registru sau o locație de memorie.

Salturi condiționate

Saltul condiționat este metoda cea mai frecventă de modificare a fluxului de instrucțiuni. Presupune un proces în doi pași. În primul pas se testează condiția, iar în pasul al doilea se efectuează saltul dacă condiția este adevărată sau se trece la executarea instrucțiunii următoare dacă condiția este falsă.

Sintaxa instrucțiunilor de salt condiționat este următoarea:

```

1 | J<conditie> target

```

Deși destinația saltului se specifică în cod tot printr-o etichetă, la asamblarea programului (traducerea în cod mașină), destinația nu va fi o valoare absolută ci va fi relativă la următoarea instrucțiune. Deoarece majoritatea salturilor se fac la adrese apropiate, respectând principiul localității, operandul va fi mai scurt, de cele mai multe ori putându-se reprezenta pe 8 sau 16 bit (față de 32 cât ar trebui pentru o adresă absolută).

Salturile condiționate folosesc ca și condiție starea indicatorilor de condiție sau combinații ale acestora. Pasul de testare se realizează cu ajutorul instrucțiunilor care afectează indicatorii de condiție. Pentru acest scop cel mai frecvent se folosesc instrucțiunile `cmp` și `test`. Pasul de salt se face folosind una din instrucțiunile de salt condiționat.

Exemplu:

```

1 | cmp AX, 7
2 | je eticheta1
3 | cmp AX, 10
4 | jg eticheta2
5 | ...
6 | eticheta1:
7 | ...
8 | eticheta2:
9 | ...

```

Comparare și salt

Instrucțiunea `cmp` compară doi operanzi prin scăderea operandului sursă din operandul destinație fără afectarea destinației și cu înscrierea corespunzătoare a indicatorilor de condiție. Sintaxa:

```

1 | cmp {registru|memorie}, {registru|memorie|valoare_imediata}

```

Este folosită pentru testarea următoarelor relații: egal, inegal, mai mare, mai mic, mai mare sau egal, mai mic sau egal.

Instrucțiunea de salt condiționat utilizată după instrucțiunea de comparare are mnemonica, în concordanță cu relația testată generată utilizând literele din Tabelul 5.1.

În Tabelul 5.2 se prezintă instrucțiunile de salt condiționat corespunzătoare fiecărei relații.

Exemple:

Tabel 5.1: Literele care formează instrucțiunile de salt condiționat

literă	semnificație
J	Jump
G	> Greater than (pentru numere cu semn)
L	< Less than (pentru numere cu semn)
A	> Above (pentru numere fără semn)
B	< Below (pentru numere fără semn)
E	= Equal
N	Not (se inversează condiția)
O	Overflow
C	Carry
Z	Zero
S	Sign
P	Parity

Tabel 5.2: Condițiile de salt după comparație

Condiție de salt		Comparație cu semn		Comparație fără semn	
		Instrucțiune	Flag-uri	Instrucțiune	Flag-uri
Egal	=	JE	$ZF = 1$	JE	$ZF = 1$
Diferit	\neq	JNE	$ZF = 0$	JNE	$ZF = 0$
Mai mare	>	JG sau JNLE	$ZF = 0$ și $SF = OF$	JA sau JNBE	$ZF = 0$ și $CF = 0$
Mai mic	<	JL sau JNGE	$SF \neq OF$	JB sau JNAE	$CF = 1$
Mai mare sau egal	\geq	JGE sau JNL	$SF = OF$	JAE sau JNB	$CF = 0$
Mai mic sau egal	\leq	JLE sau JNG	$ZF = 1$ sau $SF \neq OF$	JBE sau JNA	$ZF = 1$ sau $CF = 1$

```

1 | ;if (CX < -20) then DX ← 30 else DX ← 20
2 |     cmp CX, -20
3 |     jl less
4 |     mov DX, 20
5 |     jmp continuation
6 | less:
7 |     mov DX, 30
8 | continuation:
9 |     ...

```

```

1 | ;if (CX ≥ 20) then DX ← 30 else DX ← 20
2 |     cmp CX, 20
3 |     jge not_geq ;verificam opusul conditiei
4 |     mov DX, 20
5 |     jmp continuation
6 | not_geq:
7 |     mov DX, 30
8 | continuation:
9 |     ...

```

Pe lângă comparații, se mai pot face salturi condiționale și bazându-ne direct pe flag-

urile din registrul EFLAGS folosind instrucțiunile din Tabelul 5.3.

Tabel 5.3: Salturi bazate pe flag-uri

instrucțiuni	condiții de salt
JO	$OF = 1$
JNO	$OF = 0$
JC	$CF = 1$
JNC	$CF = 0$
JZ	$ZF = 1$
JNZ	$ZF = 0$
JS	$SF = 1$
JNS	$SF = 0$
JP, JPE	$PF = 1$ (parity even)
JNP, JPO	$PF = 0$ (parity odd)
JECXZ	$ECX = 0$

Se observă că `jecz` este singura instrucțiune de salt condiționat care nu testează indicatorii de condiție ci conținutului registrului ECX.

```

1 |      add EAX, EBX
2 |      jo  overflow
3 |      ...
4 | overflow:
5 |      ...

```

5.2.2 Instrucțiuni de ciclare

Instrucțiunile de ciclare permit o programare ușoară a structurilor de control de tip ciclu cu testul la sfârșit.

Sintaxa acestor instrucțiuni este următoarea:

```

1 | loop  <eticheta> ;se decrementează ECX și dacă ECX este nenul se
   |   efectuează saltul
2 | loope <eticheta> ;se decrementează ECX și dacă ECX este nenul și ZF
   |   =0 se efectuează saltul
3 | loopz <eticheta> ;identice cu loope
4 | loopne <eticheta> ;se decrementează ECX și dacă ECX este nenul și
   |   ZF=1 se efectuează saltul
5 | loopnz <eticheta> ;identice cu loopne

```

Instrucțiunile de ciclare decrementează conținutul registrului ECX și dacă condiția de salt este îndeplinită se face saltul.

Exemplu:

```

1 |      mov ECX, 200 ;initializare contor
2 | next:
3 |      ...
4 |      loop next ;repetare dacă ECX e nenul
5 |      ... ;continuare după ciclu

```

Această buclă are același efect ca și cea din exemplul următor:

```

1 |      mov ECX, 200
2 | next:

```

```

3 |     ...
4 |     dec ECX
5 |     cmp ECX, 0
6 |     jne next
7 |     ...

```

Atenție: Instrucțiunea `loop` face întâi decrementarea registrului `ECX`, apoi comparația cu 0. Asta înseamnă că dacă `ECX` are valoare 0, prin decrementare se va ajunge la `0FFFFFFFh`, deci bucla se va repeta de 2^{32} ori până să se ajungă la 0.

Folosirea instrucțiunii `jecxz` permite realizarea unor instrucțiuni de control de tip ciclu cu testul la început.

Exemplu:

```

1 | next:
2 |     jecxz continuation
3 |     ...
4 |     loop next
5 | continuation:
6 |     ...

```

5.2.3 Instrucțiuni pe șiruri

Aceste instrucțiuni s-au introdus cu scopul de a accelera accesul la elementele unei structuri de tip șir sau vector. Instrucțiunile folosesc în mod implicit registrele index `ESI` și `EDI` pentru adresarea elementelor șirului sursă și respectiv destinație. După efectuarea operației propriu-zise (specificată prin mnemonica instrucțiunii), regiștrii index sunt incrementați sau decrementați automat pentru a trece la elementele următoare din șir. Indicatorul *DF* (direction flag) determină direcția de parcurgere a șirurilor: $DF = 0$ prin incrementare, $DF = 1$ prin decrementare. Registrul `ECX` este folosit pentru contorizarea numărului de operații efectuate. După fiecare execuție registrul `ECX` se decrementează.

Instrucțiunile **MOVS***

Aceste instrucțiuni transferă un element din șirul sursă în șirul destinație. Instrucțiunea `movsb` operează pe octet (eng. *MOVe String on Byte*), `movsw` operează pe cuvânt (*w* - word), iar `movsd` operează pe dublu cuvânt (*d* - dword). La operațiile pe cuvânt regiștrii index se incrementează sau se decrementează cu 1, 2 sau 4 unități, în funcție de numărul de octeți pe care operează instrucțiunea. Instrucțiunile nu au parametrii; programatorul trebuie să încarce în prealabil adresele șirurilor în regiștrii `ESI` și `EDI`, iar lungimea șirului în `ECX`.

Exemplu:

```

1 |     mov ESI, offset sir_sursa ;offset este un operator care
   |     determina adresa variabilei
2 |     mov EDI, offset sir_destinatie
3 |     mov ECX, lung_sir
4 | eticheta:
5 |     movsb ;[EDI] ← [ESI]
   |         ;ESI ← ESI + 1
6 |         ;EDI ← EDI + 1
7 |         ;ECX ← ECX - 1
8 |
9 |     jnz eticheta

```

Instrucțiunile **LODS*** și **STOS***

Instrucțiunile `lodsb`, `lodsw` și `lodsd` realizează încărcarea succesivă a elementelor unui șir în registrul acumulator (AL, AX, respectiv EAX). Instrucțiunile `stosb`, `stosw` și `stosd` realizează operația inversă de salvare a registrului acumulator într-un șir. și la aceste instrucțiuni registrul index (ESI pentru încărcare și EDI pentru salvare) se incrementează sau se decrementează automat, iar registrul ECX se decrementează. Terminațiile `b`, `w` sau `d` indică lungimea pe care se face transferul: octet, cuvânt sau dublu-cuvânt.

Instrucțiunile **CMPS*** și **SCAS***

Aceste instrucțiuni realizează operații de comparare cu elemente ale unui șir. `cmpsb`, `cmpsw` și `cmpsd` compară între ele elementele a două șiruri, iar `scasb`, `scasw` și `scasd` compară conținutul registrului acumulator cu câte un element al șirului (operație de scanare).

Prefixele **REP**, **REPZ**, **REPE**, **REPZ** și **REPNE**

Aceste prefixe permit execuția multiplă a unei instrucțiuni pe șiruri. Prin amplasarea unui astfel de prefix în fața unei instrucțiuni pe șiruri procesorul va repeta operația până ce condiția de terminare este satisfăcută. La prima variantă, `rep`, condiția de terminare este `ECX=0`. La instrucțiunile `repz` și `repe` operația se repetă atâta timp cât rezultatul este zero sau operanzii sunt egali. La `repnz` și `repne` operația se repetă atâta timp cât rezultatul este diferit de zero sau operanzii sunt diferiți.

Exemplu:

```

1 | lea ESI, sir_sura
2 | lea EDI, sir_destinatie
3 | mov ECX, lung_sir
4 | rep movsb ;transfera sirul sursa in sirul destinatie

```

5.3 Întrebări recapitulative

- Definiți și exemplificați:
 - saltul necondiționat direct
 - saltul necondiționat indirect
- Care instrucțiune este echivalentă cu `JZ`?
 - `JNE`
 - `JE`
 - `JA`
 - `JNZ`
- Care instrucțiune este echivalentă cu `JAE`?
 - `JA`
 - `JB`
 - `JNE`
 - `JNB`
- Care este diferența dintre instrucțiunile `JA` și `JG`?
- Precizați dacă se face sau nu saltul, în următoarele situații:

```

1 | mov AL, 200
2 | cmp AL, 25
3 | jl eticheta

```

```

1 | mov AL, 1
2 | mov BL, -2
3 | cmp AL, BL
4 | ja eticheta

```


6. Ce valori vor avea regiștrii EAX, EBX, ECX și EDX după instrucțiunile?

<pre> 1 mov EAX, 1 2 mov EBX, 2 3 mov ECX, 3 4 mov EDX, 4 5 eticheta: 6 mul EBX 7 sub EDX, 1 8 loop eticheta </pre>	<pre> 1 mov EAX, 1 2 mov EBX, 2 3 mov ECX, 3 4 mov EDX, 4 5 eticheta: 6 mul ECX 7 loop eticheta </pre>
--	--

5.4 Mersul lucrării

5.4.1 Probleme rezolvate

1. Implementarea unei structuri de tip *for* în limbaj de asamblare.

Pseudocod:

```

for  $i = 0 \rightarrow n - 1$  do
    EAX  $\leftarrow$  EAX + 1
    EBX  $\leftarrow$  EBX - 5
end for

```

Limbaj de asamblare, varianta 1:

```

1 |      mov EDI, n
2 |      mov ESI, 0 ;corespondentul lui i
3 | et_for:
4 |      inc EAX
5 |      sub EBX, 5
6 |      inc ESI ;incrementarea lui i
7 |      cmp ESI, EDI
8 |      jbe et_for

```

Limbaj de asamblare, varianta 2:

```

1 |      mov ECX, n ;ECX retine numarul de pasi de efectuat
2 | et_loop:
3 |      inc EAX
4 |      sub EBX, 5
5 |      loop et_loop ;ECX  $\leftarrow$  ECX - 1, ECX == 0?

```

2. (s5ex1.asm) Să se determine minimul și maximul dintr-un șir de numere fără semn reprezentate pe octet și să se scrie valorile găsite în memorie. Se va compila fișierul sursă, se va depana executabilul cu Olly Debugger și se vor urmări schimbările la nivelul regiștrilor, memoriei și flag-urilor.

5.4.2 Probleme propuse

1. Să se determine minimul și maximul dintr-un șir de numere cu semn reprezentate pe cuvânt și să se scrie valorile găsite în memorie.

2. Să se scrie un program pentru calculul sumei unui șir de numere întregi reprezentate pe octet.

3. Să se scrie un program care număra câți biți de 1 sunt într-un număr întreg reprezentat pe cuvânt, păstrat în memorie.

Exemplu: Numărul $1580 = 0000\ 0110\ 0010\ 1100_2$ conține 5 biți de 1.

Pentru rezolvarea acestei probleme se poate utiliza instrucțiunea de deplasare logică (`shl` sau `shr`), urmată de verificarea valorii regăsite în flag-ul *CF*.

Bonus: Problema de mai sus se poate rezolva și folosind flag-ul *PF* (parity flag), care reține bitul de paritate pentru cel mai puțin semnificativ octet din rezultat (este setat dacă numărul de biți de 1 din octetul respectiv este par).

4. Să se implementeze un program care caută un string *s1* în alt string *s2*. În caz că *s1* se găsește, registrul *EAX* va conține poziția din *s2* unde s-a găsit *s1*. Altfel, *EAX* va conține valoarea -1 (0FFFFFFFFh).

Laborator 6

Utilizarea bibliotecilor de funcții

6.1 Scopul lucrării

În această lucrare de laborator, se va discuta utilizarea funcțiilor de bibliotecă. Principalele convenții de apel, pentru aceste funcții sunt `cdecl`, `stdcall` și `fastcall`. Biblioteca `msvcrt` pune la dispoziție funcțiile standard, întâlnite în limbajul C, dintre care se vor prezenta cele pentru afișare pe ecran, citire de la tastatură, respectiv citire/scriere din/în fișiere.

6.2 Rolul sistemului de operare și al bibliotecilor de funcții

Deși limbajul de asamblare folosește în mod direct componentele hardware ale sistemului, există porțiuni de cod utilizate frecvent, care ar fi impractic să fie scrise de către programator de fiecare dată. În plus, comunicarea cu dispozitivele de intrare/ieșire presupune de cele mai multe ori protocoale complexe, a căror utilizare necorespunzătoare poate duce inclusiv la avarii fizice ale acestora. Unul dintre rolurile sistemului de operare este acela de a abstractiza mașina hardware pentru programator. Din acest motiv, un program ce rulează în spațiu utilizator nu va accesa direct dispozitivele de intrare/ieșire (limbajul de asamblare permite acest lucru, prin instrucțiunile `in` și `out`), ci va apela la sistemul de operare.

Deasemenea, pentru anumite operații frecvente, cum ar fi afișarea datelor într-un anumit format, găsirea unui subsir într-un șir sau diverse funcții matematice, există biblioteci de funcții, ce pot fi apelate.

Utilizarea unei funcții presupune saltul la porțiunea de cod corespunzătoare funcției, execuția acestui cod, urmată de revenirea la instrucțiunea de după cea care a apelat funcția, ca în Figura 6.1.

6.3 Utilizarea funcțiilor externe. Convenții de apel

Pentru apelul unei funcții se folosește instrucțiunea `call`. Această instrucțiune pune adresa instrucțiunii următoare pe stivă, apoi sare la începutul funcției apelate. Punerea pe stivă a adresei instrucțiunii următoare (adresa de revenire) se face pentru ca la finalul execuției să se poată reveni la codul apelant. Se poate considera că apelul de funcție (`call`) și saltul necondiționat (`jmp`) sunt similare, cu excepția faptului că apelul de funcție pune pe stivă adresa de revenire.

De cele mai multe ori, funcțiile pot primi parametri, și pot întoarce rezultate. Există mai multe convenții pentru a face aceste lucruri, dintre care vom discuta 3 în acest lab-

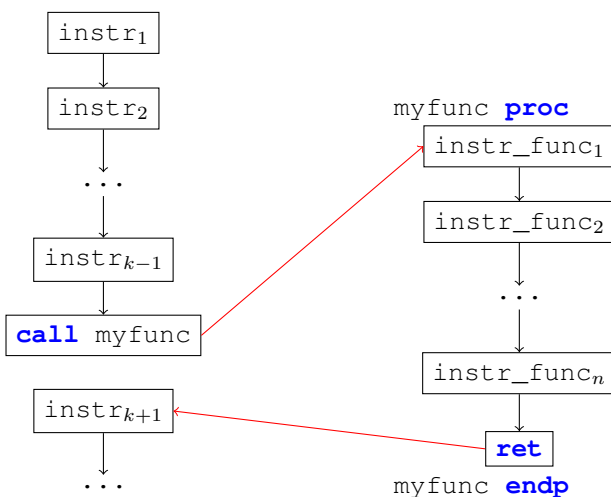


Figura 6.1: Fluxul de instrucțiuni la apelul unei funcții

orator: *cdecl*, *stdcall* și *fastcall*. Pentru a exemplifica, vom considera următoarea funcție, dintr-un limbaj de nivel înalt (C):

```
int myfunc(int x, int y, int z, int t)
```

Vom dori să apelăm această funcție, cu parametrii a, b, c, și d, obținând rezultatul în variabila res:

```
res = myfunc(a, b, c, d)
```

Trebuie remarcat că o convenție de apel nu ține de sintaxa limbajului de asamblare, ci este un "contract" între autorul funcției și utilizatorii acesteia, ce specifică modul de transmitere a parametrilor, respectiv de întoarcere a rezultatului. Cine scrie o funcție poate folosi orice convenție de apel dorește, putând inclusiv să inventeze una proprie. Este important în schimb, ca dacă funcția este apelată de altcineva, acesta să cunoască convenția de apel folosită. În limbaj de asamblare, nu este nevoie să se specifice asamblorului convenția folosită.

6.3.1 Convenția cdecl

La această convenție, argumentele funcției se vor pune pe stivă, în ordine inversă (de la dreapta la stânga), iar rezultatul va fi returnat în registrul EAX. Regiștrii EAX, ECX și EDX pot fi folosiți în interiorul funcției (acest lucru înseamnă că după apel, valorile acestora pot diferi față de starea dinaintea apelului). Funcțiile ce folosesc convenția cdecl nu vor curăța argumentele de pe stivă, această sarcină revenind apelantului.

O excepție de la regula de returnare a rezultatului apare la funcțiile care returnează un rezultat în virgulă flotantă. Acest rezultat nu se va mai pune în registrul EAX, ci în registrul flotant ST(0).

De exemplu, codul în asamblare, pentru a face apelul de mai sus, este:

```

1 | push d
2 | push c
3 | push b
4 | push a
5 | call myfunc
6 | add ESP, 16
7 | mov res, EAX
  
```

La liniile 1-4 se pun pe stivă argumentele funcției (presupunem că variabilele a, b, c, d, res au fost declarate de tip `DWORD`). Urmează apelul funcției la linia 5, apoi curățarea stivei, la linia 6. Pentru a curăța stiva trebuie să adunăm la registrul `ESP`, ce reține adresa vârfului, lungimea totală a datelor puse pe stivă (deoarece atunci când punem ceva pe stivă, adresa vârfului descrește). Am pus pe stivă 4 variabile de tip `DWORD`, ce ocupă fiecare 4 octeți, deci trebuie să adunăm 16. Nu în ultimul rând, rezultatul obținut în registrul `EAX`, va fi mutat în variabila `res`, la linia 7.

Ca exemplu de funcții ce respectă convenția `cdecl`, avem funcțiile standard din limbajul C (de exemplu `printf`).

6.3.2 Convenția `stdcall`

Această convenție de apel este similară cu `cdecl`, diferența fiind că sarcina de a curăța argumentele de pe stivă revine funcției apelate, nu apelantului. Această convenție de apel este potrivită doar pentru funcțiile cu număr fix de parametri.

Pentru exemplul de mai sus, apelul se face în modul următor:

```

1 | push d
2 | push c
3 | push b
4 | push a
5 | call myfunc
6 | mov res, EAX

```

Observăm că singura diferență apare la linia 6, renunțându-se la curățarea stivei.

O convenție similară cu `stdcall` este convenția `pascal`, diferența fiind că argumentele se vor pune pe stivă în ordine, începând cu argumentul cel mai din stânga.

Convenția `stdcall` este specifică API-urilor Win32 (funcții externe, oferite de sistemele de operare Windows pe 32 bit).

6.3.3 Convenția `fastcall`

Convenția `fastcall` (în varianta Microsoft) presupune transmiterea primilor 2 parametri ai funcției, de la stânga la dreapta, care încap ca reprezentare într-un `DWORD`, în regiștrii `ECX` și `EDX`, restul parametrilor punându-se pe stivă, de la dreapta la stânga. Rezultatul se va returna în registrul `EAX` (cu excepția rezultatelor în virgulă flotantă), iar stiva va fi curățată de funcția apelată, la fel ca la convenția `stdcall`.

```

1 | mov ECX, a
2 | mov EDX, b
3 | push d
4 | push c
5 | call myfunc
6 | mov res, EAX

```

Observăm transmiterea diferită a parametrilor, în liniile 1-4.

Avantajul acestei convenții de apel este viteza. Citirea unor date din regiștri este mult mai rapidă decât citirea din memoria principală.

6.4 Funcții standard din `msvcrt`

6.4.1 Afișarea pe ecran și citirea de la tastatură

Pentru afișarea unui text pe ecran, ce respectă un anumit format, se folosește funcția `printf`:

```
int printf(const char * format, ...);
```

Primul argument al funcției este un string ce conține formatul afișării, urmat de un număr de argumente egal cu cel specificat în cadrul formatului. String-ul transmis în parametrul `format` poate conține anumite marcaje de formatare, ce încep cu caracterul '%', care vor fi înlocuite de valorile specificate în următoarele argumente, formatare corespunzător.

Formatul complet al unui astfel de marcator este:

```
%[flags][width][.precision][length]specifier
```

O parte din specificatori sunt prezentați în Tabelul 6.1.

Tabel 6.1: Specificatorii de format ai funcției `printf`

Specificator	Ce se afișează	Exemplu
c	Caracter	a
d sau i	Întreg zecimal cu semn	392
u	Întreg zecimal fără semn	7235
o	Număr în octal fără semn	610
x	Număr în hexazecimal fără semn	7fa
X	Număr în hexazecimal fără semn (literele A-F mari)	7FA
e	Notăție științifică (mantisă/exponent) cu caracterul 'e'	3.9265e+2
E	Notăție științifică (mantisă/exponent) cu caracterul 'E'	3.9265E+2
f	Număr cu zecimale	392.65
s	String (șir de caractere, terminat cu 0)	exemplu
p	Pointer (în hexazecimal)	4031b2

Funcția `printf` respectă convenția `cdecl`, deci argumentele sale se vor transmite prin stivă, de la dreapta la stânga, iar curățarea acestora va cade în sarcina apelantului.

Exemplu de program care afișează un întreg și un string:

```

1 | .data
2 | nume DB "Ion", 0 ;lucram cu stringuri terminate in 0
3 | varsta DD 20 ;printf lucreaza cu intregi pe 32 bit
4 | format DB "Ma numesc %s si am %d de ani.", 0
5 | .code
6 | start:
7 |     push varsta
8 |     push offset nume ;pe stiva se pune adresa string-ului, nu
   |     continutul
9 |     push offset format
10 |    call printf
11 |    add ESP, 12 ;curatam 3 argumente de pe stiva
12 |    push 0
13 |    call exit
14 | end start

```

Pentru a citi date de la tastatură se folosește funcția `scanf`:

```
int scanf(const char * format, ...);
```

Sintaxa acestei funcții este similară cu cea a funcției `printf`. Diferența majoră constă în faptul că argumentele sale nu trebuie să fie valori, ci adrese în memorie, unde se vor stoca valorile citite.

Codul de mai jos va afișa mesajul "n=", apoi va citi de la tastatură valoarea numărului n.

```

1 | .data
2 | msg DB "n=", 0
3 | n DD 0
4 | format DB "%d", 0
5 | .code
6 | start:
7 |     push offset msg
8 |     call printf
9 |     add ESP, 4
10 |    push offset n ;echivalent cu &n din C
11 |    push offset format
12 |    call scanf
13 |    add ESP, 8
14 |    push 0
15 |    call exit
16 | end start

```

6.4.2 Lucrul cu fișiere text

Atunci când utilizăm biblioteca `msvcrt`, conceptul de fișier text este similar cu cel din limbajul C. Un fișier se va deschide, se vor efectua operații de citire sau scriere, asupra lui, apoi se va închide.

Deschiderea și închiderea unui fișier se fac folosind funcțiile `fopen` și `fclose`.

```
FILE * fopen(const char * filename, const char * mode);
int fclose(FILE * stream);
```

La fel ca celelalte funcții specifice limbajului C, `fopen` și `fclose` respectă convenția `cdecl`.

`fopen` va primi ca parametri un string cu numele fișierului ce trebuie deschis și un string cu modul de deschidere. Rezultatul returnat va fi un pointer spre fișierul deschis. Cele mai comune valori pentru modul de deschidere sunt prezentate în Tabelul 6.2.

Tabel 6.2: Moduri de deschidere pentru funcția `fopen`

Mod	Utilizare
r	Deschidere în mod citire, pentru fișiere text (<i>read</i>)
rb	Deschidere în mod citire, pentru fișiere binare (<i>read binary</i>)
w	Deschidere în mod scriere, pentru fișiere text (<i>write</i>)
wb	Deschidere în mod scriere, pentru fișiere binare (<i>write binary</i>)
a	Deschidere în mod scriere, adăugare (<i>append</i>)

Codul de mai jos va deschide fișierul `fișier.txt` în mod citire, apoi îl va închide.

```

1 | .data
2 | mode_read DB "r", 0

```

```

3 | file_name DB "fisier.txt", 0
4 | .code
5 | start:
6 |     push offset mode_read
7 |     push offset file_name
8 |     call fopen
9 |     add ESP, 8
10 |    push EAX ;in eax a fost returnat pointer-ul spre fisier
11 |    call fclose
12 |    add ESP, 4
13 |    push 0
14 |    call exit
15 | end start

```

Pentru a citi/scrie un fișier în mod text, se folosesc funcțiile `fprintf` și `fscanf`:

```

int fprintf(FILE * stream, const char * format, ...);
int fscanf(FILE * stream, const char * format, ...);

```

Utilizarea acestora este similară cu `printf` și `scanf`, excepția fiind primul parametru, ce trebuie să fie un pointer spre un fișier deja deschis.

În cazul unui fișier binar (caz general, ce poate include și fișiere text), funcțiile folosite sunt `fread` și `fwrite`:

```

size_t fread(void * ptr, size_t size, size_t count, FILE * stream);
size_t fwrite(const void * ptr, size_t size, size_t count, FILE * stream);

```

Spre deosebire de `fprintf` și `fscanf`, acestea nu citesc din fișier date cu un anumit format, ci vor citi conținut "pur" binar. Primul parametru reprezintă adresa de început în memorie, a unui buffer ce va fi citit/scriș. Al doilea parametru este dimensiunea unității de scriere. Dacă vrem să scriem câte un BYTE, valoarea va fi 1, pentru WORD 2, iar pentru DWORD 4. Al treilea parametru va fi numărul de elemente, de dimensiune `size`, ce vor fi scrise. Ultimul parametru este un pointer spre un fișier ce a fost deschis în prealabil în mod corespunzător.

6.5 Întrebări recapitulative

1. Care este diferența dintre instrucțiunile `JMP` și `CALL`?
2. De unde "știe" procesorul să revină la codul apelant, la finalul unui apel de funcție?
3. Cum se transmit parametrii în convenția de apel `stdcall`?
4. Cum se returnează rezultatul unei funcții `cdecl`?
5. Ce convenție de apel se folosește în următoarele secvențe de instrucțiuni?

1 <code>push EAX</code>	1 <code>mov ECX, 1</code>
2 <code>push EBX</code>	2 <code>mov EDX, 2</code>
3 <code>call functie</code>	3 <code>push dword 3</code>
4 <code>add ESP, 8</code>	4 <code>call functie</code>

6. Care este avantajul funcțiilor care respectă convenția `fastcall`?

7. Ce convenție de apel respectă funcția `printf`?
8. Cu ce funcție se poate citi un fișier în mod binar?
9. Care este diferența dintre funcțiile `fprintf` și `fwrite`?

6.6 Mersul lucrării

6.6.1 Probleme rezolvate

1. (`s6ex1.asm`) Să se afișeze pe ecran un mesaj, folosind funcția `printf` și diverse moduri de formatare. Programul va fi asamblat cu MASM, execuția sa se va trasa în Olly Debugger, urmărind în mod special stiva. Deasemenea programul se va rula în consolă, și se va observa rezultatul afișării.
2. (`s6ex2.asm`) Să se afișeze pe ecran conținutul unui fișier binar în hexazecimal.

6.6.2 Probleme propuse

1. Să se scrie în limbaj de asamblare un program care cere utilizatorului 2 numere, de la tastatură, apoi afișează suma acestora pe ecran.
2. Să se citească de la tastatură un șir de caractere, și să se scrie într-un fișier text, șirul inversat.
3. Să se scrie un program care cere utilizatorului să ghicească un număr. Cât timp utilizatorul nu a ghicit numărul, programul va afișa unul din mesajele “mai mic” sau “mai mare” și va citi următoarea încercare a utilizatorului. Atunci când numărul a fost ghicit, se va afișa pe ecran numărul de încercări. *Observație:* pentru a genera un număr aleator se poate folosi instrucțiunea `rdtsc`.
4. Scrieți un program care citește un fișier text pe mai multe linii și scrie liniile într-un alt fișier, în ordine inversă. Se consideră că o linie se poate păstra în memorie, dar nu și întreg conținutul fișierului. Pentru deplasare în cadrul unui fișier se poate folosi funcția `fseek`.

Laborator 7

Scrierea de macroui și proceduri

7.1 Scopul lucrării

În această lucrare de laborator, se vor prezenta două metode de re folosire a codului, macrouile și procedurile (funcțiile).

7.2 Scrierea și utilizarea macrouilor

Atunci când scriem programe, sunt situații în care dorim să folosim în mai multe locuri, o porțiune de cod, deja scris. Pentru a obține acest lucru, se pot folosi macrouile.

Macrouile sunt facilități pentru programatorii în limbaj de asamblare. Un macrou este o pseudo-operație care permite includerea repetată de cod în program. Macroul odată definit, apelul lui prin nume permite inserarea lui ori de câte ori este nevoie. La întâlnirea unui nume de macrou asamblorul expandează numele lui în codul corespunzător corpului de macrou. Din acest motiv se spune că macrouile sunt executate in-line deoarece cursul de execuție secvențial al programului nu este întrerupt.

Macrouile pot fi create în cadrul programului utilizator sau grupate într-un alt fișier numit bibliotecă de macroui. O bibliotecă de macroui este un simplu fișier care conține o serie de macroui și care este invocat în timpul asamblării programului, la prima trecere a asamblorului peste programul sursă. De menționat faptul că o bibliotecă de macroui conține linii sursă neasamblate. Din această cauză bibliotecile de macroui trebuie să fie incluse în programul sursă al utilizatorului prin pseudoinstrucțiunea INCLUDE. Aceasta este diferența majoră față de o bibliotecă de proceduri în cod obiect care conține proceduri asamblate sub formă de cod obiect și care este invocată la link-editare.

Definirea unui macrou se face cu următoarea secvență:

```
1 | nume MACRO <parametrii_macroului>
2 | LOCAL <etichete_locale> ;acestea sunt expandate cu nume diferite
   |     la apelul repetat al macroului
3 |     <corpul_macroului>
4 | ENDM
```

Exemplu: Un macro care calculează valoarea $n!$ în registrul EAX.

```
1 | factorial MACRO n
2 | LOCAL fact_bucla, fact_final
3 |     push ECX ;salveaza pe stiva registrii utilizati
4 |     push EDX
5 |     mov EAX, 1
6 |     mov ECX, n
```

```

7 |     test ECX, ECX ;vedem daca nu cumva ECX=0
8 |     jz fact_final ;daca e 0, nu mai facem inmultiri
9 | fact_bucla:
10 |     mul ECX
11 |     loop fact_bucla
12 | fact_final:
13 |     pop EDX
14 |     pop ECX
15 | ENDM

```

Ca să utilizăm acest macro într-un program, putem fie să scriem codul de mai sus, în codul sursă al programului, fie să creem un fișier separat, de exemplu "mylib.inc", în care să îl scriem, apoi să includem acest fișier, scriind "`include mylib.inc`".

Dacă dorim să obținem valoarea lui 5! în registrul EAX, putem scrie pur și simplu

```
1 | factorial 5
```

Macro-ul poate primi ca parametru și un registru, de exemplu:

```
1 | factorial ESI
```

7.3 Scrierea de proceduri în limbaj de asamblare

Atât procedurile cât și macrourele, ajută la reutilizarea codului. Dezavantajul macrourele este că în programul asamblat, codul macrourele se va repeta de atâtea ori, de câte ori este apelat. Un alt dezavantaj este imposibilitatea de a utiliza recursivitatea, folosind doar macrourele (deoarece o funcție recursivă se apelează pe ea însăși de un număr variabil de ori).

Procedurile sau funcțiile (denumite în limbaj de asamblare și rutine), sunt porțiuni de cod, care pot fi reutilizate, prin modificarea fluxului de execuție, către locația lor în memorie. În laboratorul anterior, s-a studiat modul de apel a procedurilor. Pentru a defini o procedură, folosim secvența:

```

1 | nume PROC
2 |     <corpul_procedurii>
3 |     ret [<dimensiune_parametri>]
4 | nume ENDP

```

Se observă că un corp de procedură este similar cu unul de macro, apărând în plus instrucțiunea `ret`. Aceasta realizează revenirea la fluxul de instrucțiuni anterior apelării procedurii. Mai exact, la instrucțiunea `ret` se citește de pe vârful stivei adresa de revenire (adresă ce a fost pusă pe stivă de instrucțiunea `call`), apoi se sare la aceasta. Din acest motiv, este foarte important ca vârful stivei să fi la finalul procedurii, același ca la început. În plus, `ret` mai poate primi un parametru suplimentar, ce reprezintă numărul de octeți ce trebuie curățați de pe stivă. Astfel, dacă se scrie o funcție folosind convenția de apel `stdcall` sau `fastcall`, se poate curăța stiva în cadrul instrucțiunii `ret`.

În practică, atunci când scriem proceduri complexe, dorim să primim argumentele pe stivă, și deasemenea să folosim variabile locale. Variabilele locale nu pot fi definite în secțiunea de date, deoarece atunci când rulează mai multe instanțe ale procedurii simultan (programare pe mai multe thread-uri, sau apeluri recursive), fiecare trebuie să aibă propriile variabile locale. Din acest motiv, variabilele locale se salvează tot pe stivă, decrementând registrul ESP, cu dimensiunea variabilelor locale. Pentru ca locația în memorie a parametrilor și a variabilelor locale să nu depindă de vârful stivei, ce se poate modifica pe

parcursul procedurii, vârful stivei se salvează la început în registrul EBP, și toate adresările se vor face relativ la acesta.

Noul șablon de scriere a unei proceduri va deveni:

```

1 | nume PROC
2 |     push EBP
3 |     mov EBP, ESP
4 |     sub ESP, <dimensiune_variabile_locale>
5 |     <corpul_procedurii>
6 |     mov ESP, EBP
7 |     pop EBP
8 |     ret [<dimensiune_parametri>]
9 | nume ENDP

```

Pe linia 2 se va salva valoarea registrului EBP, pe stivă, iar la linia 3, valoarea curentă a lui ESP (vârful stivei), se salvează în EBP. La linia 4, se alocă spațiu pe stivă pentru variabilele locale. De exemplu, dacă avem nevoie de 2 variabile locale, de tip DWORD, vom scrie `sub ESP, 8`.

La linia 6 se reface valoarea registrului ESP, apoi la linia 7 se recuperează valoarea inițială a lui EBP, ce a fost pusă pe stivă la linia 2.

În corpul procedurii, ne putem referi la parametri sau la variabilele locale, folosind adrese relative la registrul EBP, ca în Figura 7.1.

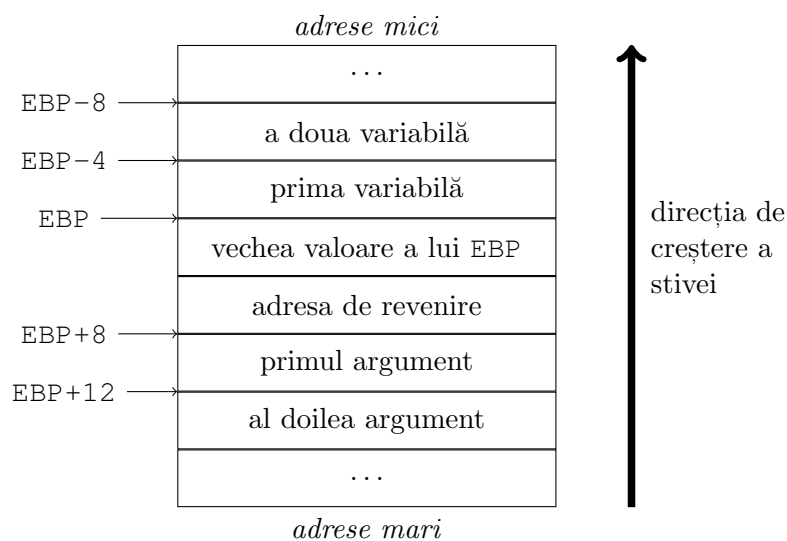


Figura 7.1: Starea stivei în cadrul unei proceduri

EBP va pointa la adresa vârfului stivei, imediat ce vechea sa valoare a fost pusă pe stivă. Imediat sub vechea valoare, se află adresa de revenire, ce a fost pusă pe stivă de instrucțiunea `call`. Primul argument se va găsi, deci, la adresa dată de `EBP+8`, și presupunând că argumentele sunt de tip `DWORD`, al doilea se va găsi la `EBP+12`, al treilea la `EBP+16`, ș.a.m.d. Variabilele locale se vor găsi imediat deasupra lui EBP, adică la adrese mai mici. Presupunând deasemenea că variabilele locale sunt de tip `DWORD`, prima va fi la adresa `EBP-4`, a doua la `EBP-8`.

Vârful stivei, indicat de registrul ESP va pointa la ultima variabilă locală. Cum stiva crește înspre adrese mai mici, variabilele locale nu vor fi suprascrise, la adăugarea de lucruri noi pe stivă.

7.4 Întrebări recapitulative

1. Prin ce diferă un macro de o procedură?
2. Ce rol au etichetele locale din cadrul unui macro?
3. Dacă avem o porțiune scurtă de cod, care se folosește frecvent, care este cea mai bună opțiune de re folosire: macro sau procedură? Dar pentru o porțiune lungă de cod?
4. Scrieți o secvență de instrucțiuni echivalentă cu instrucțiunea `ret`.

5. Ce efect are codul de mai jos?

```

1 | push dword 401230h
2 | ret

```

6. Considerând o funcție care începe cu instrucțiunile `push EBP` și `mov EBP, ESP`, cum se va adresa al 4-lea argument al funcției, știind că toate argumentele sunt de tip `DWORD`?

7.5 Mersul lucrării

7.5.1 Probleme rezolvate

1. (`s7ex1.asm`, `s7lib.inc`) În `s7lib.inc` este definit un macro ce calculează factorialul unui număr dat. `s7ex1.asm` folosește această bibliotecă pentru a calcula factorialul unor numere, precum și un alt macro, ce afișează un număr pe ecran. Programul va fi asamblat cu MASM, execuția sa se va trasa în Olly Debugger, urmărind cum arată codul, după ce macrourile au fost expandate. Deasemenea programul se va rula în consolă, și se va observa rezultatul afișării.
2. (`s7ex2.asm`) Să se calculeze termenul n din șirul Fibonacci folosind definiția recursivă:

$$\begin{aligned}
 F_0 &= 0 \\
 F_1 &= 1 \\
 F_n &= F_{n-1} + F_{n-2}
 \end{aligned}$$

Se va urmări în mod special adresarea parametrului și a variabilei. Ce s-ar întâmpla dacă am reține valorile intermediare în regiștri, în loc de variabile locale? Ce convenție de apel se folosește?

Observație: implementarea șirului Fibonacci printr-o funcție recursivă nu este cea mai bună idee în practică. Implementarea de față are doar scop didactic.

7.5.2 Probleme propuse

1. Să se scrie un macro care calculează suma numerelor impare mai mici decât o valoare transmisă ca parametru. Macro-ul se va apela de mai multe ori în cadrul unui program, și se vor afișa pe ecran rezultatele.

2. Să se scrie o funcție care primește 2 parametri și verifică dacă primul este divizibil cu al doilea. Folosind această funcție, să se scrie o alta, care verifică dacă un număr este prim. Prima funcție va folosi convenția de apel `fastcall`, iar a doua convenția `cdecl`.
3. Scrieți macro-urile `apel0`, `apel1`, `apel2`, `apel3`, ... care apelează o funcție ce respectă convenția `cdecl` cu 0, 1, 2, 3, ... parametri.
Exemplu: `apel2 printf, offset format, EAX`
va efectua apelul funcției `printf` cu două argumente (formatul de afișare și registrul `EAX`) și va curăța stiva în urma apelului.
4. Căutare binară: se dă un șir ordonat, de numere de tip `DWORD` și lungimea acestuia. Să se scrie o funcție `bsearch(sir, lungime, x)` care verifică dacă parametrul `x` primit apare în șir. Pentru punctaj complet, căutarea se va face în mod binar: la fiecare pas comparăm cu `x` elementul de la jumătatea șirului curent, iar dacă nu l-am găsit căutăm în subșirul drept sau stâng, în funcție de rezultatul comparației.

Laborator 8

Utilizarea coprocesorului matematic

8.1 Scopul lucrării

Lucrarea de față își propune familiarizarea utilizatorului cu folosirea funcțiilor coprocesorului matematic, utilizarea instrucțiunilor de lucru cu numere reale și a altor funcții ale coprocesorului.

8.2 Considerații generale

Cu toate că procesorul 8086 respectiv 80286, 80386 și 80486 au o serie de instrucțiuni aritmetice puternice (care există în repertoriul microprocesoarelor de generația a doua, cum sunt înmulțirea și împărțirea binară) nu sprijină operațiile aritmetice cu numerele reprezentate în virgula flotantă sau pentru numerele întregi reprezentate pe mai mulți octeți. Dacă vrem să efectuăm astfel de operații atunci aceste operații trebuie să fie realizate prin biblioteci de macrouri sau prin subrutine. O cale mai simplă este utilizarea limbajelor de nivel înalt, în aceste limbaje operațiile de mai sus fiind realizate prin biblioteci.

Pentru evitarea greutăților amintite, firma Intel a dezvoltat coprocesorul aritmetic Intel 8087 (80287, 80387). Acestea după cum arată și numele sunt "co"-procesoare, care cooperează cu procesorul principal al calculatorului. Coprocesorul nu poate lucra independent de procesor. Coprocesorul nu poate să extragă singur instrucțiunile din memorie, acesta fiind realizat de procesor.

8.2.1 Principiul de funcționare

Coprocesorul se inițializează odată cu activarea semnalului *RESET*, generat în sistemul de calcul. În urma acestui semnal coprocesorul este adus într-o stare inițială (cu mascarea erorilor, ștergerea regiștrilor, inițializarea stivei, rotunjirea implicită, etc.). După prima instrucțiune executată de procesor coprocesorul poate detecta cu ce procesor lucrează (dacă este 8086 pe pinul 34 va fi 0 logic-semnalul / BHE-iar dacă este 8088 va fi 1 logic-semnalul / SS0). În funcție de tipul procesorului, coprocesorul se configurează corespunzător.

Coprocesorul se conectează pe magistrala locală a procesorului prin liniile de adrese/date, stare, ceas, ready, reset, test și request/grant. Fiind conectat la magistrala locală microprocesorul permite coprocesorului accesul la toată memoria și resursele de intrare/ieșire,

prin intermediul cererii de magistrală request/grant. Cele două procesoare lucrând în paralel este nevoie de sincronizarea proceselor care rulează în ele.

De regulă sincronizarea erorilor și a instrucțiunilor este rezolvată de compilatoare sau asamblatoare, iar sincronizarea datelor trebuie să o facă utilizatorul care lucrează în limbajul de asamblare.

În interiorul coprocesorului avem o memorie de 80 de octeți organizată sub forma unei stive de opt elemente de 10 octeți fiecare. Pe cei zece octeți numerele în virgulă mobilă se reprezintă sub format intern, cu precizie extinsă. Coprocesorul poate accesa memoria calculatorului, cu orice mod de adresare cunoscut de 8086, orice dată de format legal. Datele aduse din memorie se convertesc în formatul intern al coprocesorului și se pun pe vârful stivei. La scrierea în memoria principală se face conversia din formatul intern în formatul specificat de utilizator.

Condiția pentru efectuarea operațiilor în virgulă flotantă în coprocesor este ca operandul (pentru operații cu doi operanzi, cel puțin unul din operanzi) să fie în vârful stivei. Deci cu ajutorul coprocesorului putem efectua următoarele operații:

- citirea datelor în memoria internă a coprocesorului (pe stivă) din memoria calculatorului
- efectuarea operațiilor aritmetice necesare
- scrierea rezultatului în memoria calculatorului

8.2.2 Tipuri de date cunoscute de Intel 8087

Marele avantaj al coprocesorului este faptul că nu lucrează doar cu numere în virgulă flotantă, ci și cu numere întregi și recunoaște și tipuri de date zecimale împachetate. Deci dacă avem o operație complicată de efectuat între numere întregi și ea trebuie să fie foarte rapid, iar acest lucru se poate realiza cu ajutorul coprocesorului, nu avem nevoie de conversie, costisitoare în timp, din întreg în virgulă flotantă și după aceea invers doar pentru ca coprocesorul să poată lucra cu ele.

Tipurile de date în virgulă flotantă sunt:

- Real scurt, reprezentat pe 32 bit (1 bit de semn, 23 bit mantisa, 8 bit caracteristica)
- Real lung, reprezentat pe 64 bit (1 bit de semn, 52 bit mantisa, 11 bit caracteristica)
- Real cu precizie ridicată, reprezentat pe 80 bit (1 bit de semn, 64 bit mantisa, 15 bit caracteristica)

8.2.3 Erori de operație (excepții)

La efectuarea operațiilor în virgulă flotantă putem avea numeroase erori începând de la erori logaritmice triviale, până la erorile provenite din limitele reprezentării. Acestea le vom numi excepții. În continuare vom cunoaște tipurile acestor erori și posibilitățile principale de manevrare a lor.

În cazul apariției erorilor, coprocesorul poate avea două tipuri de comportare. Anunță eroarea printr-o întrerupere dacă utilizatorul validează acest lucru. Dacă nu validăm întreruperea, coprocesorul va trata intern eroarea și în funcție de erorile apărute va acționa în modul prezentat în continuare. Proiectanții coprocesorului au clasificat erorile în următoarele 6 clase:

- *Invalid operation*: operație invalidă
Aceasta poate fi: depășire superioară sau inferioară a stivei interioare a coprocesorului. Depășirea inferioară apare în cazul în care vrem să accesăm un element din stivă care nu există. Acestea sunt de obicei erori (destul de grave) algoritmice; coprocesorul nu afectează operația. Avem rezultat nedefinit în cazul în care încercăm să împărțim 0.0 cu 0.0, coprocesorul nu este pregătit pentru aceasta. Cazuri similare apar la scăderea lui infinit din infinit, etc. Aceste erori (deși se pot evita prin algoritm) nu sunt erori atât de grave ca cele de depășire inferioară sau superioară a stivei. Avem tot acest "rezultat" dacă o funcție de coprocesor este apelată cu parametri necorespunzători. În cazul apariției rezultatului nedefinit coprocesorul înscrie în caracteristică o valoare rezervată pentru acest caz (biți de zero).
- *Overflow*: depășire superioară
Rezultatul depășește numărul cel mai mare ce se poate reprezenta. Coprocesorul înscrie infinit în locul rezultatului și continuă lucrul.
- *Zero Divisor*: împărțire cu zero
Împărțitorul împărțirii de efectuat este zero, iar deîmpărțitul nu este zero sau infinit. Coprocesorul înscrie în locul rezultatului infinit și continuă lucrul.
- *Underflow*: depășire inferioară
Valoarea rezultatului în modul este mai mică decât numărul cel mai mic reprezentabil. Rezultatul va fi zero, coprocesorul continuă lucrul.
- *Denormalized*: operand nenormalizat
Această excepție apare dacă unul din operanzi nu este normalizat sau rezultatul nu se poate reprezenta normalizat (de exemplu este atât de mic încât este imposibilă normalizarea lui). Coprocesorul continuă lucrul (valorile diferite de zero se pierd, vor fi zero).
- *Inexact result*: rezultat inexact
Rezultatul operației este inexact din cauza unor rotunjiri prescrise sau necesare. Putem avea astfel de rezultate după împărțire, dacă împărțim de exemplu 2.0 cu 3.0 rezultatul se poate reprezenta doar ca o fracție infinită. Coprocesorul efectuează rotunjirea și continuă lucrul.

Cele de mai sus sunt prezentate în funcție de gravitatea erorii. Dacă apare o depășire de stivă atunci programul este eronat; nu merită să se continue programul.

În același timp nu e nevoie să se trateze o de eroare de rotunjire. Nici pe hârtie nu putem manevra ușor fracții cu repetiție infinită sau cu numere iraționale. Din punct de vedere practic este indiferent dacă pierdem a 20-a cifra a fracției sau nu, deoarece nu aceasta este elementul care poartă informația de bază. Pentru rezolvarea acestei probleme este necesară o analiză adâncă a situațiilor și rezultatelor care pot apărea în timpul execuției, a preciziei de reprezentare a numerelor, timpul de rulare și mărimea memoriei. După cum am văzut la reprezentarea numerelor, precizia numerelor reale scurte nu este de ajuns pentru orice aplicație practică. Precizia numerelor reale lungi este mai mult ca sigur suficientă, dar necesită un spațiu dublu de memorie.

8.3 Setul de instrucțiuni al coprocesorului

Programarea coprocesorului se face în limbajul de asamblare cu ajutorul instrucțiunii ESC. Această instrucțiune trimite pe magistrala de date un cod de operație pe 6 biți și dacă

este necesar trimite pe magistrala de date o adresă de memorie. Coprocesorul sesizează și "captează" instrucțiunea ce i se adresează și începe execuția instrucțiunii. Există două posibilități de resincronizare a procesorului cu coprocesorul, ambele la inițiativa procesorului:

- procesorul testează starea coprocesorului
- procesorul lansează o instrucțiune WAIT

8.3.1 Instrucțiuni de transfer de date

Instrucțiunile de transfer de date asigură schimbul de date între memoria calculatorului și stiva coprocesorului. Ele se pot împărți în următoarele categorii:

- *instrucțiuni de încărcare (LOAD)*
 - **fld** adr - Încarcă pe stivă variabila întreagă de la adresa adr. Variabila din memorie de tipul definit la declararea lui (DB, DW, DD) se convertește în formatul intern al coprocesorului în timpul încărcării.
 - **fld** adr - Încarcă pe stivă valoarea reală (scurtă sau lungă) de la adresa de memorie adr. Variabila din memorie de tipul definit la declararea lui (DD, DQ, DT) se convertește în formatul intern al coprocesorului în timpul încărcării.
 - **fbld** adr - Încarcă pe stivă variabila din memorie de tipul zecimal împachetat (definit cu DT) de la adresa de memorie adr. Are loc convertirea în formatul intern al coprocesorului în timpul încărcării.
- *instrucțiuni de memorare (STORE)*
 - **fist** adr - Memorează la adresa adr valoarea de pe stivă ($ST(0)$) ca număr întreg. Valoarea memorată poate fi de tip cuvânt sau dublu-cuvânt, în funcție de definiția de la adresa adr (DW sau DD). Indicatorul de stivă nu se modifică în urma memorării. În timpul memorării are loc convertirea.
 - **fistp** adr - Memorează la adresa adr valoarea de pe stivă ($ST(0)$) ca număr întreg. Valoarea memorată poate fi orice număr întreg (pe cuvânt sau dublu-cuvânt în funcție de definiția de la adresa adr DW, DD sau DQ). În timpul memorării are loc convertirea necesară. Instrucțiunea afectează stiva: $ST(0)$ este eliminat prin decrementarea indicatorului de stivă.
 - **fst** adr - Memorează la adresa adr valoarea de pe stivă ($ST(0)$) ca număr în virgulă mobilă. Valoarea memorată poate fi un cuvânt sau dublu-cuvânt în funcție de definiția de la adresa adr DD sau DQ. În timpul memorării are loc convertirea necesară. Indicatorul de stivă și conținutul stivei nu se modifică în urma memorării.
 - **fstp** adr - Memorează la adresa adr valoarea de pe stivă ($ST(0)$) ca număr în reprezentarea în virgulă mobilă. Valoarea memorată poate număr real scurt, cu precizie dublă sau extins, în funcție de definiția de la adresa adr (DD, DQ sau DT). În timpul memorării are loc convertirea necesară din formatul intern. Instrucțiunea afectează stiva: $ST(0)$ este eliminat prin decrementarea indicatorului de stivă.
 - **fbstp** adr - Memorează la adresa adr valoarea de pe stivă ($ST(0)$) ca număr zecimal împachetat (definit la adr de regulă cu DT). Indicatorul de stivă este decrementat. În timpul memorării are loc convertirea necesară din formatul intern.

8.3.2 Instrucțiuni transfer de date intern

- **fld ST(i)** - Pune pe stivă valoarea de pe $ST(i)$. Deci valoarea din $ST(i)$ se va găsi de două ori: în $ST(0)$ și $ST(i+1)$.
- **fst ST(i)** - Valoarea din $ST(0)$ este copiată în elementul i din stivă. Valoarea veche din $ST(i)$ se pierde.
- **fstp ST(i)** - Valoarea $ST(0)$ este copiată în elementul i din stivă. Valoarea veche din $ST(i)$ se pierde. $ST(0)$ este eliminat prin decrementarea indicatorului din stivă.
- **fxch ST(i)** - Se schimbă între ele $ST(0)$ și $ST(i)$.

8.3.3 Instrucțiuni încărcare a constantelor

- **fldz** - Încarcă zero pe vârful stivei.
- **fld1** - Încarcă 1.0 pe vârful stivei.
- **fldpi** - Încarcă π pe vârful stivei.
- **fldl2t** - Încarcă pe vârful stivei $\log_2 10$.
- **fldl2e** - Încarcă pe vârful stivei $\log_2 e$.
- **fldlg2** - Încarcă pe vârful stivei $\log_{10} 2$.
- **fldln2** - Încarcă pe vârful stivei $\log_e 2$.

8.3.4 Instrucțiuni aritmetice și de comparare

Instrucțiunile aritmetice sunt în general cu doi operanzi. Unul din operanzi este totdeauna în vârful stivei și de regulă tot aici se generează rezultatul. Operațiile de bază se pot executa fără restricții cu următoarele variante:

- se scrie numai mnemonica instrucțiunii fără operand. În acest caz operanzii implicați sunt $ST(0)$ și $ST(1)$.
- se scrie mnemonica instrucțiunii și operandul. Operandul poate fi o locație de memorie sau un element de pe stivă (evident, operandul poate fi inclusiv $ST(1)$ se poate dar e inutil).
- se scrie mnemonica instrucțiunii și doi operanzi: primul un element de pe stivă (diferit de $ST(0)$), al doilea $ST(0)$. În acest caz rezultatul se va depune în locul primului operand iar $ST(0)$ se șterge de pe stivă. (În mnemonica instrucțiunii apare litera P).

Instrucțiuni aritmetice

În cele ce urmează vom folosi următoarele notații:

- $ST(i)$ - registrul numărul i al coprocesorului matematic
- $m32fp$ - o variabilă pe 32 bit (declarată ca DD) ce reține un număr în virgulă mobilă (*fp* - *Floating Point*)

- `m64fp` - o variabilă pe 64 bit (declarată ca `DQ`) ce reține un număr în virgulă mobilă
- `m16int` - o variabilă pe 16 bit (declarată ca `DW`) ce reține un număr întreg
- `m32int` - o variabilă pe 32 bit (declarată ca `DD`) ce reține un număr întreg

Următoarele instrucțiuni realizează în diverse moduri principalele operații aritmetice (adunare, scădere, înmulțire și împărțire). `<op>` se va înlocui pe rând cu `add`, `sub`, `mul` și `div`, operatorul \odot luând valorile `+`, `-`, `*` și `/`.

- `f<op>` $ST(1) \leftarrow ST(1) \odot ST(0)$, elimină `ST(0)`
- `f<op> m32fp` $ST(0) \leftarrow ST(0) \odot m32fp$
- `f<op> m64fp` $ST(0) \leftarrow ST(0) \odot m64fp$
- `f<op> ST(0), ST(i)` $ST(0) \leftarrow ST(0) \odot ST(i)$
- `f<op> ST(i), ST(0)` $ST(i) \leftarrow ST(i) \odot ST(0)$
- `f<op>p` $ST(1) \leftarrow ST(1) \odot ST(0)$, elimină `ST(0)`
- `f<op>p ST(i), ST(0)` $ST(i) \leftarrow ST(i) \odot ST(0)$, elimină `ST(0)`
- `fi<op> m32int` $ST(0) \leftarrow ST(0) \odot m32int$
- `fi<op> m16int` $ST(0) \leftarrow ST(0) \odot m16int$

Deoarece operațiile de scădere și împărțire nu sunt comutative, avem și operațiile inverse pentru acestea, care au aceeași formă dar se termină cu litera `r` (de la *reverse*):

- `f<op>r` $ST(1) \leftarrow ST(0) \odot ST(1)$, elimină `ST(0)`
- `f<op>r m32fp` $ST(0) \leftarrow m32fp \odot ST(0)$
- `f<op>r m64fp` $ST(0) \leftarrow m64fp \odot ST(0)$
- `f<op>r ST(0), ST(i)` $ST(0) \leftarrow ST(i) \odot ST(0)$
- `f<op>r ST(i), ST(0)` $ST(i) \leftarrow ST(0) \odot ST(i)$
- `f<op>rp` $ST(1) \leftarrow ST(0) \odot ST(1)$, elimină `ST(0)`
- `f<op>rp ST(i), ST(0)` $ST(i) \leftarrow ST(0) \odot ST(i)$, elimină `ST(0)`
- `fi<op>r m32int` $ST(0) \leftarrow m32int \odot ST(0)$
- `fi<op>r m16int` $ST(0) \leftarrow m16int \odot ST(0)$

Instrucțiuni pentru compararea valorilor numerice

Prin compararea valorilor numerice, se setează flag-urile interne ale coprocesorului, în mod similar cu setarea flag-urilor din registrul `EFLAGS` de către instrucțiunea `cmp`. Toate instrucțiunile de tipul `fcom*` compară vârful stivei `ST(0)` cu un alt operand, setând flag-urile `C3`, `C2` și `C0` conform Tabelului 8.1.

Tabel 8.1: Flag-urile coprocesorului în urma instrucțiunii de comparație

Condiție	<code>C3</code>	<code>C2</code>	<code>C0</code>
$ST(0) > SRC$	0	0	0
$ST(0) < SRC$	0	0	1
$ST(0) = SRC$	1	0	0
nesortate	1	1	1

Instrucțiunea `fcom` fără compară regiștrii `ST(0)` și `ST(1)`. Varianta cu un operand, compară registrul `ST(0)` cu operandul dat, acesta putând fi o locație de memorie pe 32 sau 64 bit, respectiv un element al stivei coprocesorului. Instrucțiunea `fcomp` este identică cu `fcom`, realizând în plus eliminarea registrului `ST(0)` din vârful stivei, iar instrucțiunea `fcompp` care există doar în varianta fără parametri elimină atât registrul `ST(0)` cât și registrul `ST(1)`.

Instrucțiunea **ftst** compară registrul $ST(0)$ cu valoarea 0.0, setând în mod similar flag-urile $C3$, $C2$ și $C0$.

8.3.5 Funcții în virgulă mobilă

- **fsqrt** $ST(0) \leftarrow \sqrt{ST(0)}$, $ST(0)$ trebuie să fie pozitiv
- **fscale** $ST(0) \leftarrow ST(0) \times 2^{\lfloor ST(1) \rfloor}$
- **fprem** $ST(0) \leftarrow ST(0) \bmod ST(1)$
- **frndint** $ST(0)$ se rotunțește la un întreg
- **fextract** $ST(0)$ se desparte în mantisă ($ST(0)$) și exponent ($ST(1)$)
- **fabs** $ST(0) \leftarrow |ST(0)|$
- **fchs** $ST(0) \leftarrow -ST(0)$
- **fptan** $ST(1) \leftarrow \tan(ST(0))$, $ST(0) \leftarrow 1$
- **fptan** $ST(1) \leftarrow \arctan \frac{ST(1)}{ST(0)}$, elimină $ST(0)$
- **f2xm1** $ST(0) \leftarrow 2^{ST(0)} - 1$, inițial trebuie să avem $-1.0 \leq ST(0) \leq 1.0$
- **fy12x** $ST(1) \leftarrow ST(1) \times \log_2(ST(0))$, elimină $ST(0)$
- **fy12xp1** $ST(1) \leftarrow ST(1) \times \log_2(ST(0) + 1.0)$, elimină $ST(0)$,
inițial $- \left(1 - \frac{\sqrt{2}}{2}\right) \leq ST(0) \leq \left(1 - \frac{\sqrt{2}}{2}\right)$

Observație: Pentru calculul de exponent $ST(0)^{ST(1)}$ se recomandă utilizarea funcțiilor **fy12x**, **fscale** și **f2xm1**, conform ecuației:

$$a^b = 2^{\log_2(a^b)} = 2^{b \cdot \log_2 a} = 2^{\lfloor b \cdot \log_2 a \rfloor} \cdot 2^{b \cdot \log_2 a - \lfloor b \cdot \log_2 a \rfloor}$$

A fost necesară împărțirea expresiei $b \cdot \log_2 a$ în parte întreagă și parte fracționară deoarece funcția **f2xm1** acceptă doar exponenți subunitari.

8.3.6 Instrucțiuni de comandă

Instrucțiunile de comandă au ca sarcină coordonarea acțiunilor coprocesorului. De obicei nu au o semnificație aritmetică, dar există câteva care influențează serios acțiunile aritmetice ale coprocesorului, deoarece ele salvează sau încarcă starea coprocesorului, adică toate registrele de lucru. În aceste registre este inclusă și stiva, deci aceste instrucțiuni pot fi privite ca instrucțiuni gigantice de scriere și salvare.

- **finit** - Inițializare-aducerea coprocesorului într-o stare inițială cunoscută ("software reset"). După efectuarea instrucțiunii **finit** toate registrele coprocesorului se vor afla în starea inițială iar stiva va fi goală.
- **feni** - Acceptarea întreruperilor-pentru ca coprocesorul să genereze o întrerupere la apariția unei erori, pe lângă poziționarea biților corespunzători registrului de comandă este nevoie de acceptarea explicită a întreruperilor.
- **fdisi** - Ignorarea întreruperilor-această instrucțiune ignoră întreruperile indiferent de starea biților corespunzători ai registrului de comandă; pentru acceptarea unor noi întreruperi trebuie să avem o nouă instrucțiune **feni**.
- **fldcw** *adr* - Se încarcă în registrul de comandă cuvântul de la adresa *adr* în memorie.

- **fstcw** adr - Salvarea registrului de comandă în variabila pe un cuvânt aflată în memorie.
- **fstsw** adr - Salvarea registrului de stare într-un cuvânt de memorie aflat la adresa adr.
- **fclex** - ștergerea biților de definire a excepțiilor - instrucțiunea șterge biții respectivi indiferent de starea biților de eroare.
- **fstenv** adr - Salvarea mediului - se salvează regiștrii interni ai coprocesorului într-o zonă de memorie începând de la adresa adr și având o lungime de 14 octeți.
- **fldenv** adr - Încărcarea mediului - se încarcă din memorie de la adresa adr o zonă de 14 octeți în regiștrii interni ai coprocesorului.
- **fsave** adr - Salvarea stării - salvarea stării coprocesorului (regiștrii interni și stiva) în zona de memorie care începe la adresa adr și are o lungime de 94 octeți.
- **frstor** adr - Restaurarea stării - încărcarea stării coprocesorului (regiștrii interni și stiva) din zona de memorie care începe la adresa adr și are o lungime de 94 octeți.
- **fincstp** - Rotirea regiștrilor de stivă cu o poziție, prin incrementarea poziției vârfului stivei.
- **fdecstp** - Rotirea regiștrilor de stivă cu o poziție, prin decrementarea poziției vârfului stivei.
- **ffree ST**(i) - ștergerea elementului *i* din stivă; operația nu afectează indicatorul de stivă.
- **fnop** - Nici o operație.
- **fwait** - Așteptare terminare operație curentă.

8.4 Întrebări recapitulative

1. Câți regiștri are stiva coprocesorului? Cum pot fi aceștia adresați?
2. Câți octeți ocupă stiva coprocesorului?
3. Cum inițializăm coprocesorul matematic pentru a efectua operații cu acesta?
4. Care este diferența între declarațiile `var1 DD 3` și `var2 DD 3.0`?
5. Cum transferăm o variabilă din memoria principală în regiștrii coprocesorului și invers?
6. Care este efectul instrucțiunilor `fadd` și `fsub`?
7. De ce este necesară instrucțiunea `fdivr`?

8.5 Mersul lucrării

8.5.1 Probleme rezolvate

1. (s8ex1.asm) Să se calculeze aria unui cerc și volumul unei sfere, dându-se raza acestora. Să se traseze programul, utilizând Olly Debugger, urmărind în special stiva coprocesorului. De ce apar instrucțiuni în plus, față de codul scris?

8.5.2 Probleme propuse

1. Să se scrie un program care calculează sinusul unui unghi, utilizând instrucțiunea `fptan` (nu se va folosi direct instrucțiunea `fsin`).

Indicație: $\tan(x) = \frac{\sin(x)}{\cos(x)}$ și $\sin^2(x) + \cos^2(x) = 1$.

2. Se va scrie un program pentru a calcula $\sqrt[3]{2}$ și $\sqrt[3]{5}$.

Indicație: se vor folosi instrucțiunile `f2xm1` și `fy12x`.

3. Scrieți un program care calculează următoarea expresie:

$$E(n) = \frac{\sqrt{1}}{1} + \frac{\sqrt{2}}{2} + \dots + \frac{\sqrt{n}}{n}$$

4. Se dă un polinom P de grad n , ca o listă de coeficienți și un număr real X . Să se calculeze $P(X)$.

Exemplu: Pentru $P(X) = 1.2 + 3X + 4.9X^3 + 8.27X^4$, polinomul se va defini în felul următor:

```

1 | p DD 1.2, 3, 0, 4.9, 8.27
2 | n EQU ($-p)/4 -1

```


Anexa A

Exemple de cod

```
1  .386
2  .model flat, stdcall
3  ;
4  ;
5  ;includem biblioteci, si declaram ce functii vrem sa importam
6  includelib msvcrt.lib
7  extern exit: proc
8  ;
9  ;
10 ;declaram simbolul start ca public - de acolo incepe executia
11 public start
12 ;
13 ;
14 ;sectiunile programului, date, respectiv cod
15 .data
16 ;aici declaram date
17 sir db 1,2,3
18 .code
19 start:
20     ;aici se scrie codul
21     mov esi,0
22     mov ebx,0
23     mov al, sir[ebx+esi+1]
24
25     ;terminarea programului
26     push 0
27     call exit
28 end start
```

Figura A.1: s2model.asm

```

1  .386
2  .model flat, stdcall
3  ;
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
4
5  ;includem msvcrt.lib, si declaram ce functii vrem sa importam
6  includelib msvcrt.lib
7  extern exit: proc
8  ;
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
9
10 ;declaram simbolul start ca public - de acolo incepe executia
11 public start
12 ;
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
13
14 ;sectiunile programului, date, respectiv cod
15 .data
16 var1 DB 10,2,3,14
17 var2 DW 15,6,7,8
18 var3 DD 1020A0h, 011223344h, 01A2B3C4Dh
19 var4 equ 10
20 var5 DB '1', '2', "abc"
21 var6 DB 4 DUP(5)
22
23 .code
24 start:
25     mov eax, 0
26     mov al, var1
27     mov ah, 11
28     mov ebx, 0
29     mov bx, var2
30     mov ecx, var3
31     mov edx, var4
32     add ecx, edx
33     mov dl, var5
34     mov dh, var6
35
36     ;terminarea programului
37     push 0
38     call exit
39 end start

```

Figura A.2: s2ex1.asm

```

1  .386
2  .model flat, stdcall
3
4  includelib msvcrt.lib
5  extern exit: proc
6
7  public start
8
9  .data
10
11 .code
12 start:
13     ;initializez pe eax si ebx cu valori oarecare
14     mov eax, 2
15     mov ebx, 12
16
17     ;pastrez vechea valoare a lui eax in edx
18     mov edx, eax
19
20     ;eax*7 = eax*8 - aex
21     ;eax*8 = eax << 3 (eax deplasat aritmetic la stanga cu 3
22     ;pozitii)
23     mov ecx, 3
24     sal eax, cl
25
26     ;eax=eax*7
27     sub eax, edx
28
29     ;pastrez vechea valoare a lui ebx in edx
30     mov edx, ebx
31
32     ;ebx*2
33     sal ebx, 1
34
35     ;eax = eax-ebx*2
36     sub eax, ebx
37
38     mov ebx, edx
39     ;ebx/8 = ebx >> 3 (deplasare la dreapta cu 3 pozitii)
40     sar ebx, cl
41
42     sub eax, ebx
43
44     ;apel functie exit
45     push 0
46     call exit

```

Figura A.3: s3ex1.asm

```
1  .386
2  .model flat, stdcall
3
4  includelib msvcrt.lib
5  extern exit: proc
6
7  public start
8
9  .data
10 NUM1 DB 10
11 NUM2 DB 5
12 REZ DB 0
13
14 .code
15 start:
16     ;initializam pe eax cu 0
17     mov eax, 0
18
19     ;in al si bl punem niste valori oarecare
20     mov al, 2
21     mov bl, 4
22
23     ;pastram vechea valoare a lui al pe stiva
24     push eax
25
26     mov ecx, 0
27     mov cl, NUM2
28     ;al*NUM2
29     mul cl
30
31     add REZ, bl
32     add REZ, al
33
34     ;luam de pe stiva vechea valoare a lui al
35     pop eax
36     mov cl, NUM1
37     ;al*NUM1
38     mul cl
39
40     add REZ, al
41
42     ;apel functie exit
43     push 0
44     call exit
45 end start
```

Figura A.4: s3ex2.asm

```
1  .386
2  .model flat, stdcall
3
4  includelib msvcrt.lib
5  extern exit: proc
6
7  public start
8
9  .data
10 sir1 dw 1,2,3
11 sir2 dw 0,0,0
12
13 .code
14 start:
15
16     lea esi, sir1
17     lea edi, sir2
18
19     ;iau din primul sir si pun in stiva
20     push word ptr[esi]
21     add esi, 2
22     push word ptr[esi]
23     add esi, 2
24     push word ptr[esi]
25
26     ;scot de pe stiva si pun in al doilea sir
27     pop word ptr[edi]
28     add edi, 2
29     pop word ptr[edi]
30     add edi, 2
31     pop word ptr[edi]
32
33     ;apel functie exit
34     push 0
35     call exit
36 end start
```

Figura A.5: s4ex1.asm

```

1  .386
2  .model flat, stdcall
3
4  includelib msvcrt.lib
5  extern exit: proc
6
7  public start
8
9  .data
10 m1 db 1,2,3
11     db 4,5,6
12 m2 db 7,8,9
13     db 10,11,12
14
15 .code
16 start:
17     mov eax, 0 ;in ax se vor pastra sumele elementelor
18             ;de pe aceeasi pozitie
19     mov esi, 0 ;pentru parcurgerea liniilor
20     mov ebx, 0 ;pentru parcurgerea coloanelor
21             ;rezultatul se pastreaza in matricea m1
22             ;suma prima linie
23     add al, m1[ebx][esi]
24     add al, m2[ebx][esi]
25     mov m1[ebx][esi], al
26     mov al, 0
27     inc esi
28     add al, m1[ebx][esi]
29     add al, m2[ebx][esi]
30     mov m1[ebx][esi], al
31     mov al, 0
32     inc esi
33     add al, m1[ebx][esi]
34     add al, m2[ebx][esi]
35     mov m1[ebx][esi], al
36     ;suma a doua linie
37     mov al, 0
38     mov esi, 0
39     add ebx, 3 ;bx se aduna cu 3
40     add al, m1[ebx][esi]
41     add al, m2[ebx][esi]
42     mov m1[ebx][esi], al
43     mov al, 0
44     inc esi
45     add al, m1[ebx][esi]
46     add al, m2[ebx][esi]
47     mov m1[ebx][esi], al
48     mov al, 0
49     inc esi
50     add al, m1[ebx][esi]
51     add al, m2[ebx][esi]
52     mov m1[ebx][esi], al
53     push 0
54     call exit
55 end start

```

Figura A.6: s4ex2.asm


```

1  .386
2  .model flat, stdcall
3
4  includelib msvcrt.lib
5  extern exit: proc
6
7  public start
8
9  .data
10 sir db 10,21,3,14,2,22,11
11 lgsir dd $-sir
12 min db 0
13 max db 0
14
15 .code
16 start:
17
18     mov ax, 0 ; al-minim, ah-maxim
19
20     mov esi, 0
21     mov ecx, lgsir
22     mov al, sir[esi]
23     mov ah, sir[esi]
24
25     et_loop:
26     cmp al, sir[esi]
27     JB cmp_max
28     mov al, sir[esi]
29     cmp_max:
30     cmp ah, sir[esi]
31     JA urmator
32     mov ah, sir[esi]
33     urmator:
34     inc esi
35     loop et_loop
36
37     mov min, al
38     mov max, ah
39
40     ;apel functie exit
41     push 0
42     call exit
43 end start

```

Figura A.7: s5ex1.asm

```
1  .386
2  .model flat, stdcall
3
4  includelib msvcrt.lib
5  extern printf: proc
6  extern exit: proc
7
8  public start
9
10 .data
11 msg db "Hello world!", 13, 10, 0
12 ;un string trebuie sa fie NULL terminated
13 ;pentru a putea fi afisat cu printf
14
15 .code
16 start:
17     mov eax, 25
18     push eax
19     push eax
20     push offset msg ;argumentele unei functii se pun pe stiva,
21                     ;in ordine inversa
22     call printf
23     ;conventia de apel la printf e cdecl, ceea ce inseamna ca
24     ;cine apeleaza, trebuie sa curete stiva
25     add esp, 12 ;am pus 3 dword pe stiva(pe 32bit, un pointer
26     ;e tot dword), deci trebuie sa eliminam ultimii 3*4=12
27     ;octeti
28     push 0
29     call exit
30 end start
```

Figura A.8: s6ex1.asm

```

1  .386
2  .model flat, stdcall
3
4  includelib msvcrt.lib
5  extern printf: proc
6  extern fread: proc
7  extern fopen: proc
8  extern fclose: proc
9  extern exit: proc
10
11 public start
12
13 .data
14 filename db "fisier.dat", 0
15 mode_rb db "rb", 0
16 format db "%02X ", 0
17 buffer db 0
18
19 .code
20 start:
21     ;apelam fopen
22     push offset mode_rb
23     push offset filename
24     call fopen
25     add esp, 8
26     mov esi, eax ;salvam pointer-ul la fisier
27
28     ;punem pe stiva parametrii pentru fread
29     push esi ;stream
30     push 1 ;count
31     push 1 ;size
32     push offset buffer
33 bucla_citire:
34     call fread
35     test eax, eax
36     jz inchidere_fisier
37     xor eax, eax ;facem eax sa fie 0
38     mov al, buffer
39     push eax
40     push offset format
41     call printf
42     add esp, 8
43     jmp bucla_citire
44
45 inchidere_fisier:
46     add esp, 16 ;curatam stiva de la fread
47     ;apelam fclose
48     push esi ;stream
49     call fclose
50     add esp, 4
51     push 0
52     call exit
53 end start

```

Figura A.9: s6ex2.asm

```
1  .386
2  .model flat, stdcall
3
4  includelib msvcrt.lib
5  extern exit: proc
6  extern printf: proc
7
8  include mylib.asm
9
10 public start
11
12 .data
13 ;aici declaram date
14 format db "%d", 13, 10, 0
15 my_int dd 0
16 .code
17 show_int macro nr
18     push nr
19     push offset format
20     call printf
21     add esp, 4
22 endm
23
24 start:
25     factorial 3
26     show_int eax
27     mov ecx, 10
28     factorial ecx
29     show_int eax
30     factorial my_int
31     show_int eax
32     ;terminarea programului
33     push 0
34     call exit
35 end start
```

Figura A.10: s7ex1.asm

```
1 ;calculeaza in eax, n!  
2 factorial macro n  
3 local fact_bucla, fact_final  
4     push ecx  
5     push edx  
6     mov eax, 1  
7     mov ecx, n  
8     test ecx, ecx ; vedem daca nu cumva ecx=0  
9     jz fact_final ; daca e 0, nu mai facem inmultiri  
10 fact_bucla:  
11     mul ecx  
12     loop fact_bucla  
13 fact_final:  
14     pop edx  
15     pop ecx  
16 endm
```

Figura A.11: mylib.asm

```

1  .386
2  .model flat, stdcall
3
4  includelib msvcrt.lib
5  extern exit: proc
6  extern printf: proc
7
8  public start
9
10 .data
11 format db "%d", 13, 10, 0
12 .code
13
14 fibo proc
15     push ebp
16     mov ebp, esp ; pregatim stack frame-ul
17     sub esp, 4 ; facem loc pe stiva pentru o variabila
18     mov eax, [ebp+8] ; citim primul argument de pe stiva
19     cmp eax, 0 ; verificam daca argumentul e 0
20     je fibo_final
21     cmp eax, 1
22     je fibo_final ; verificam daca argumentul e 1
23     dec eax
24     push eax
25     call fibo ; calculam fibo(n-1)
26     mov [ebp-4], eax ; punem rezultatul in variabila locala
27     mov eax, [ebp+8] ; luam din nou argumentul
28         ; in cadrul apelului, eax s-a modificat
29     sub eax, 2
30     push eax
31     call fibo ; calculam si fibo(n-2)
32     add eax, [ebp-4] ; fibo(n) = fibo(n-2) + fibo(n-1)
33 fibo_final:
34     mov esp, ebp
35     pop ebp
36     ret 4 ; salt inapoi la adresa de return, si curata
37         ; parametrii de pe stiva
38
39 fibo endp
40
41 start:
42     push 6
43     call fibo
44     push eax
45     push offset format
46     call printf
47     add esp, 8
48     ;terminarea programului
49     push 0
50     call exit
51 end start

```

Figura A.12: s7ex2.asm

```

1  .386
2  .model flat, stdcall
3
4  includelib msvcrt.lib
5  extern exit: proc
6  extern printf: proc
7
8  public start
9
10 .data
11 RAZA      DQ 8.567
12 ARIE      DQ 0
13 VOLUM     DQ 0
14 PATRU     DD 4.0
15 TREI      DD 3.0
16 format    DB "Raza=%lf; Arie=%lf; Volum=%lf", 0
17
18 .code
19 start:
20          FINIT                ;INITIALIZARE COPROCESOR
21          FLD RAZA              ;INCARCARE RAZA IN STIVA COPROC
22          FMUL RAZA             ;CALCUL R*R
23          FLDPI                ;INCARCARE PI PE STIVA COPOC
24          FMUL                 ;CALCUL R*R*PI
25          FSTP ARIE            ;SALVARE REZULTAT
26
27          LEA ESI, VOLUM        ;ADRESA VLUM IN SI
28          FINIT                ;INITIALIZARE COPROCESOR
29          FLD RAZA              ;CALCUL
30          FMUL RAZA             ;R*R
31          FMUL RAZA             ;R*R*R
32          FLDPI                ;ICARCARE PI
33          FMUL                 ;INMULTIRE CU PI
34          FMUL PATRU            [U+FFFD]NMULTIRE PATRU
35          FDIV TREI             ;IMPARTIRE CU TREI
36          FSTP QWORD PTR [ESI] ;SALVARE REZULTAT
37
38          ;ca sa punem pe stiva un QWORD, trebuie sa punem pe rand
39          ;cei 2 DWORD din care e format
40          push dword ptr [VOLUM+4]
41          push dword ptr [VOLUM]
42          push dword ptr [ARIE+4]
43          push dword ptr [ARIE]
44          push dword ptr [RAZA+4]
45          push dword ptr [RAZA]
46          push offset format
47          call printf
48          add esp, 28
49
50          ;terminarea programului
51          push 0
52          call exit
53          end start

```

Figura A.13: s8ex1.asm

Anexa B

Lista instrucțiunilor uzuale în Limbaj de Asamblare

În această anexă se va prezenta o listă ordonată alfabetic a instrucțiunilor uzuale din Limbajul de Asamblare, care au fost întâlnite în acest laborator. Pentru fiecare instrucțiune se va descrie operația efectuată, respectiv numerele de pagină în care este prezentată pe larg în laborator.

În specificarea operanzilor, vom folosi următoarele notații:

- $r8$:= AL|AH|BL|BH|CL|CH|DL|DH - regiștri pe 8 bit
- $r16$:= AX|BX|CX|DX|SI|DI|BP|SP - regiștri pe 16 bit
- $r32$:= EAX|EBX|ECX|EDX|ESI|EDI|EBP|ESP - regiștri pe 32 bit
- r := $r8|r16|r32$ - orice registru de uz general
- $m8, m16, m32$ - variabile din memorie pe 8, 16 sau 32bit
- m := $m8|m16|m32$ - orice variabilă din memorie
- $i8, i16, i32$ - valoare imediată (constantă) pe 8, 16 sau 32bit
- i := $i8|i16|i32$ - orice valoare imediată

Dacă o instrucțiune are doi operanzi, cu excepția cazurilor în care se specifică contrariul, acești operanzi trebuie să aibă dimensiuni egale și nu au voie să fie ambii variabile din memorie.

adc op1, op2 op1 := r m op2 := r m i	$op1 \leftarrow op1 + op2 + CF$ (<i>Add With Carry</i>)	37
add op1, op2 op1 := r m op2 := r m i	$op1 \leftarrow op1 + op2$	37
and op1, op2 op1 := r m op2 := r m i	$op1 \leftarrow op1 \text{ AND } op2$ (ȘI logic pe biți)	39
call procedure	Apel de procedură. Se pune adresa instrucțiunii următoare pe stivă și se sare la codul procedurii.	59

clc	$CF \leftarrow 0$ (Clear Carry Flag)	41
cld	$DF \leftarrow 0$ (Clear Direction Flag)	42
cli	$IF \leftarrow 0$ (Clear Interrupt Flag)	42
cmc	$CF \leftarrow \overline{CF}$ (Complement Carry Flag)	41
cmp op1, op2 op1 := r m op2 := r m i	Compară op1 și op2 și setează flag-urile ca și la scădere, fără a modifica operanzii.	39
cmpsb cmpsw cmpsd	Compară BYTE-ul, WORD-ul sau DWORD-ul de la adresa ESI cu cel de la EDI, setând flag-urile. Regiștrii ESI și EDI sunt incrementați sau decrementați (în funcție de <i>DF</i>) automat cu 1, 2, sau 4.	56
cpuid	EAX, EBX, ECX, EDX ← Informații despre procesor	42
dec op op := r m	$op \leftarrow op - 1$	38
div op op := r m	op=r8 m8 ⇒ $AL \leftarrow AX/op, AH \leftarrow AX\%op$ op=r16 m16 ⇒ $AX \leftarrow DX : AX/op, DX \leftarrow DX : AX\%op$ op=r32 m32 ⇒ $EAX \leftarrow EDX : EAX/op, EDX \leftarrow EDX : EAX\%op$ Valorile se consideră fără semn.	38
idiv op op := r m	op=r8 m8 ⇒ $AL \leftarrow AX/op, AH \leftarrow AX\%op$ op=r16 m16 ⇒ $AX \leftarrow DX : AX/op, DX \leftarrow DX : AX\%op$ op=r32 m32 ⇒ $EAX \leftarrow EDX : EAX/op, EDX \leftarrow EDX : EAX\%op$ Valorile se consideră cu semn.	38
imul op op := r m	op=r8 m8 ⇒ $AX \leftarrow AL \times op$ op=r16 m16 ⇒ $DX : AX \leftarrow AX \times op$ op=r32 m32 ⇒ $EDX : EAX \leftarrow EAX \times op$ Valorile se consideră cu semn.	37
imul op1, op2 op1 := r16 r32 op2 := r16 r32 m16 m32	$op1 \leftarrow op1 \times op2$ Valorile se consideră cu semn.	37
imul op1, op2, op3 op1 := r16 r32 op2 := r16 r32 m16 m32 op3 := i	$op1 \leftarrow op2 \times op3$ Valorile se consideră cu semn.	37
in op1, op2 op1 := AL AX EAX op2 := i8 DX	$op1 \leftarrow$ valoarea de pe portul op2	41
inc op op := r m	$op \leftarrow op + 1$	38
ja label_dst	(Jump if Above) Efectuează saltul dacă rezultatul comparației fără semn este mai mare. $CF = 0$ și $ZF = 0$	53
jae label_dst	(Jump if Above or Equal) Efectuează saltul dacă rezultatul comparației fără semn este mai mare sau egal. $CF = 0$	53

jb label_dst	(<i>Jump if Below</i>) Efectuează saltul dacă rezultatul comparației fără semn este mai mic. $CF = 1$	53
jbe label_dst	(<i>Jump if Below or Equal</i>) Efectuează saltul dacă rezultatul comparației fără semn este mai mic sau egal. $CF = 1$ sau $ZF = 1$	53
jc label_dst	(<i>Jump if Carry</i>) Efectuează saltul dacă la operația anterioară s-a generat un transport. $CF = 1$	54
jecxz label_dst	(<i>Jump if ECX=0</i>) Efectuează saltul dacă registrul ECX are valoarea 0. $ECX = 0$	54
je label_dst	(<i>Jump if Equal</i>) Efectuează saltul dacă numerele comparate au fost egale. $ZF = 1$	53
jg label_dst	(<i>Jump if Greater</i>) Efectuează saltul dacă rezultatul comparației cu semn este mai mare. $ZF = 0$ și $SF = OF$	53
jge label_dst	(<i>Jump if Greater or Equal</i>) Efectuează saltul dacă rezultatul comparației cu semn este mai mare sau egal. $SF = OF$	53
jl label_dst	(<i>Jump if Lower</i>) Efectuează saltul dacă rezultatul comparației cu semn este mai mic. $SF \neq OF$	53
jle label_dst	(<i>Jump if Lower or Equal</i>) Efectuează saltul dacă rezultatul comparației cu semn este mai mic sau egal. $ZF = 1$ sau $SF \neq OF$	53
jmp label_dst	(<i>Jump</i>) Salt necondiționat.	51
jna label_dst	(<i>Jump if Not Above</i>) Efectuează saltul dacă rezultatul comparației fără semn nu este mai mare. $CF = 1$ sau $ZF = 1$	53
jnae label_dst	(<i>Jump if Not Above or Equal</i>) Efectuează saltul dacă rezultatul comparației fără semn nu este mai mare sau egal. $CF = 1$	53
jnb label_dst	(<i>Jump if Not Below</i>) Efectuează saltul dacă rezultatul comparației fără semn nu este mai mic. $CF = 0$	53
jnbe label_dst	(<i>Jump if Not Below or Equal</i>) Efectuează saltul dacă rezultatul comparației fără semn nu este mai mic sau egal. $CF = 0$ și $ZF = 0$	53
jnc label_dst	(<i>Jump if Not Carry</i>) Efectuează saltul dacă la operația anterioară nu s-a generat transport. $CF = 0$	54
jne label_dst	(<i>Jump if Not Equal</i>) Efectuează saltul dacă la comparația anterioară nu s-a obținut egalitate. $ZF = 0$	53
jng label_dst	(<i>Jump if Not Greater</i>) Efectuează saltul dacă rezultatul comparației cu semn nu este mai mare. $ZF = 1$ sau $SF \neq OF$	53

jnge label_dst	(<i>Jump if Not Greater or Equal</i>) Efectuează saltul dacă rezultatul comparației cu semn nu este mai mare sau egal. $SF \neq OF$	53
jnl label_dst	(<i>Jump if Not Lower</i>) Efectuează saltul dacă rezultatul comparației cu semn nu este mai mic. $SF = OF$	53
jnle label_dst	(<i>Jump if Not Lower or Equal</i>) Efectuează saltul dacă rezultatul comparației cu semn nu este mai mic sau egal. $ZF = 0$ sau $SF = OF$	53
jno label_dst	(<i>Jump if Not Overflow</i>) Efectuează saltul dacă la operația anterioară nu s-a produs overflow. $OF = 0$	54
jnp label_dst	(<i>Jump if Not Parity</i>) Efectuează saltul dacă la operația anterioară nu s-a setat bitul de paritate. $PF = 0$	54
jns label_dst	(<i>Jump if Not Signed</i>) Efectuează saltul dacă la operația anterioară nu s-a obținut un număr negativ. $SF = 0$	54
jnz label_dst	(<i>Jump if Not Zero</i>) Efectuează saltul dacă la operația anterioară nu s-a obținut rezultatul 0. $ZF = 0$	53
jo label_dst	(<i>Jump if Overflow</i>) Efectuează saltul dacă la operația anterioară s-a produs overflow. $OF = 1$	54
jp label_dst	(<i>Jump if Parity</i>) Efectuează saltul dacă la operația anterioară s-a setat bitul de paritate. $PF = 1$	54
jpe label_dst	(<i>Jump if Parity Even</i>) Efectuează saltul dacă la operația anterioară s-a obținut un număr par de biți de 1 în ultimul octet. $PF = 1$	54
jpo label_dst	(<i>Jump if Parity Odd</i>) Efectuează saltul dacă la operația anterioară s-a obținut un număr impar de biți de 1 în ultimul octet. $PF = 0$	54
js label_dst	(<i>Jump if Signed</i>) Efectuează saltul dacă la operația anterioară s-a obținut un număr negativ. $SF = 1$	54
jz label_dst	(<i>Jump if Zero</i>) Efectuează saltul dacă la operația anterioară s-a obținut rezultatul 0. $ZF = 1$	53
lahf	$AH \leftarrow lsb(EFLAGS)$ În registrul AH se încarcă byte-ul mai puțin semnificativ din EFLAGS.	36
lea op1, op2 op1 := r16 r32 op2 := m	$op1 \leftarrow offset(op2)$ În op1 se încarcă adresa efectivă a lui op2 (<i>Load Effective Address</i>).	35
lods	$AL \leftarrow byte\ ptr\ [ESI]$	56
lodsw	$AX \leftarrow word\ ptr\ [ESI]$	56
lodsd	$EAX \leftarrow dword\ ptr\ [ESI]$	56

loop label_dst	Decrementează registrul ECX, iar dacă rezultatul e diferit de zero, efectuează saltul.	54
mov op1, op2 op1 := r m op2 := r m i	$op1 \leftarrow op2$	35
movsb movsw movsd	Mută BYTE-ul, WORD-ul sau DWORD-ul de la adresa ESI la adresa EDI. Regiștrii ESI și EDI sunt incrementați sau decrementați (în funcție de <i>DF</i>) automat cu 1, 2, sau 4.	55
mul op op := r m	op=r8 m8 $\Rightarrow AX \leftarrow AL \times op$ op=r16 m16 $\Rightarrow DX : AX \leftarrow AX \times op$ op=r32 m32 $\Rightarrow EDX : EAX \leftarrow EAX \times op$ Valorile se consideră fără semn.	37
neg op op := r m	Înlocuiește argumentul cu complementul acestuia față de 2 (se inversează semnul).	
nop	Nu se efectuează nici o operație (<i>No operation</i>).	
not op op := r m	Înlocuiește argumentul cu complementul acestuia față de 1 (se inversează fiecare bit).	39
or op1, op2 op1 := r m op2 := r m i	$op1 \leftarrow op1 \text{ OR } op2$ (SAU logic pe biți)	39
out op1, op2 op1 := i8 DX op2 := AL AX EAX	Transmite valoarea din op2 de pe portul op1.	41
pop op op := r16 r32 m16 m32	Citește valoarea din vârful stivei în operand. op=r16 m16 $\Rightarrow op \leftarrow [ESP]; \quad ESP \leftarrow ESP + 2$ op=r32 m32 $\Rightarrow op \leftarrow [ESP]; \quad ESP \leftarrow ESP + 4$	36
popa	Citește de pe stivă valorile regiștrilor de uz general, cu excepția ESP. Ordinea în care regiștrii sunt încărcăți este: EDI, ESI, EBP, se sar 4 octeți pentru ESP, EBX, EDX, ECX, EAX. Registrul ESP se incrementează cu 32.	
popf	Citește DWORD-ul din vârful stivei în registrul EFLAGS. Registrul ESP se incrementează cu 4.	36
push op op := r16 r32 m16 m32	Pune operandul pe vârful stivei. op=r16 m16 $\Rightarrow ESP \leftarrow ESP - 2; \quad [ESP] \leftarrow op$ op=r32 m32 $\Rightarrow ESP \leftarrow ESP - 4; \quad [ESP] \leftarrow op$	36
pusha	Pune pe stivă valorile regiștrilor de uz general. Ordinea în care regiștrii sunt puși pe stivă este: EDI, ESI, EBP, valoarea originală a lui ESP, EBX, EDX, ECX, EAX. Registrul ESP se decrementează cu 32 (dimensiunea datelor).	
pushf	Pune valoarea registrului EFLAGS pe vârful stivei în registrul. Registrul ESP se decrementează cu 4.	36
rcl op1, op2 op1 := r m op2 := i8 CL	(<i>Rotate with Carry Left</i>) Rotește la stânga biții din op1 cu op2 poziții, trecând prin flag-ul <i>CF</i> . Valoarea din <i>CF</i> ajunge în bit-ul 0 (cel mai puțin semnificativ), iar valoarea din bit-ul 7 ajunge în <i>CF</i> .	40

rcr op1, op2 op1 := r m op2 := i8 CL	(<i>Rotate with Carry Right</i>) Rotește la dreapta biții din op1 cu op2 poziții, trecând prin flag-ul <i>CF</i> . Valoarea din <i>CF</i> ajunge în bit-ul 7 (cel mai semnificativ), iar valoarea din bit-ul 0 ajunge în <i>CF</i> .	40
rdtsc	(<i>Read Time-Stamp Counter</i>) Citește numărul de cicli de ceas executați de procesor de la pornire, în regiștrii <i>EDX:EAX</i> .	
rep instr	(<i>Repeat</i>) Prefix care adăugat în fața unei instrucțiuni o face să se repete și să decrementeze registrul <i>ECX</i> , până ce acesta devine 0.	56
repz instr repe instr	(<i>Repeat while Zero/Equal</i>) Prefix care adăugat în fața unei instrucțiuni o face să se repete și să decrementeze registrul <i>ECX</i> , până ce acesta devine 0 sau până ce <i>ZF</i> devine 0.	56
repe instr repz instr	(<i>Repeat while Equal/Zero</i>) Prefix care adăugat în fața unei instrucțiuni o face să se repete și să decrementeze registrul <i>ECX</i> , până ce acesta devine 0 sau până ce <i>ZF</i> devine 0.	56
repne instr repnz instr	(<i>Repeat while Not Equal/Zero</i>) Prefix care adăugat în fața unei instrucțiuni o face să se repete și să decrementeze registrul <i>ECX</i> , până ce acesta devine 0 sau până ce <i>ZF</i> devine 1.	56
ret	Întoarcere din procedură, prin extragerea adresei de revenire de pe vârful stivei și salt la aceasta.	68
ret op op := i16	Întoarcere din procedură, prin extragerea adresei de revenire de pe vârful stivei și salt la aceasta. În plus, se mai elimină și op octeți de pe vârful stivei, de sub adresa de revenire.	68
rol op1, op2 op1 := r m op2 := i8 CL	(<i>Rotate Left</i>) Rotește la stânga biții din op1 cu op2 poziții.	40
ror op1, op2 op1 := r m op2 := i8 CL	(<i>Rotate Right</i>) Rotește la dreapta biții din op1 cu op2 poziții.	40
sahf	Încarcă byte-ul cel mai puțin semnificativ al registrului <i>EFLAGS</i> din <i>AH</i> .	36
sal op1, op2 op1 := r m op2 := i8 CL	(<i>Shift Arithmetic Left</i>) Deplasează la stânga biții din op1 cu op2 poziții.	40
sar op1, op2 op1 := r m op2 := i8 CL	(<i>Shift Arithmetic Right</i>) Deplasează la dreapta biții din op1 cu op2 poziții. Pozițiile libere rămase în partea stângă se completează cu bitul de semn.	40
sbb op1, op2 op1 := r m op2 := r m i	$op1 \leftarrow op1 - op2 - CF$ (<i>Subtract with Borrow</i>)	37
shl op1, op2 op1 := r m op2 := i8 CL	(<i>Shift Left</i>) Deplasează la stânga biții din op1 cu op2 poziții.	40

shr op1, op2 op1 := r m op2 := i8 CL	(<i>Shift Right</i>) Deplasează la dreapta biții din op1 cu op2 poziții. Pozițiile libere rămase în partea stângă se completează cu zerouri.	40
stc	$CF \leftarrow 1$ (<i>Set Carry Flag</i>)	41
std	$DF \leftarrow 1$ (<i>Set Direction Flag</i>)	42
sti	$IF \leftarrow 1$ (<i>Set Interrupt Flag</i>)	42
stosb	byte ptr [EDI] ← AL	56
stosw	word ptr [EDI] ← AX	56
stosd	dword ptr [EDI] ← EAX	56
sub op1, op2 op1 := r m op2 := r m i	$op1 \leftarrow op1 - op2$ (<i>Subtract</i>)	37
test op1, op2 op1 := r m op2 := r m i	Se efectuează operația logică AND între op1 și op2 fără a stoca rezultatul și se setează flag-urile <i>ZF</i> , <i>SF</i> și <i>PF</i> conform acestuia.	39
xchg op1, op2 op1 := r m op2 := r m	$op1 \leftrightarrow op2$ (<i>Exchange</i>)	36
xor op1, op2 op1 := r m op2 := r m i	$op1 \leftarrow op1 \text{ XOR } op2$ (SAU EXCLUSIV pe biți)	39