Elemente de bază ale limbajului de asamblare

Regiştrii procesoarelor Intel

Orice procesor conţine un număr foarte mare de regiştri, utilizaţi pentru diverse acţiuni interne. Aceştia nu prezintă interes pentru programator. În schimb există o categorie specială, cea a regiştrilor de uz general, care pot fi folosiţi explicit în programe, în mod similar variabilelor de memorie. În cazul procesoarelor din familia Intel x86 pe 32 biţi, regiştrii de uz general sunt cei de mai jos:

	31	15 8	7 0	
EAX		AH	AL	AX
EBX		BH	BL	BX
ECX		CH	CL	CX
EDX		DH	DL	DX
ESI				SI
EDI				DI
EBP				BP
ESP				SP

Regiştrii generali pot fi folosiţi atât pentru a reţine date (toţi), cât şi pentru formarea adreselor locaţii de memorie (numai cei pe 32 biţi).

Atenţie: regiştrii din tabelul de mai sus nu sunt complet independenţi. De exemplu, faptul că registrul EAX îl conţine pe AX înseamnă că orice modificare a valorii lui AX afectează automat şi valoarea lui EAX şi reciproc; similar pentru regiştrii de 1 octet.

Formarea adreselor

Sintactic, o adresă trebuie scrisă între paranteze pătrate. Modurile de adresare, prezentate în continuare, arată cum poate fi precizată o adresă. Nu vom folosi denumirile uzuale, care pot crea confuzie.

a) Adresa este o constantă

Exemplu: [100]

b) Adresa este precizată prin valoarea unui registru general pe 32 biţi

Exemplu: [EAX] (se poate scrie şi cu litere mici)

c) Adresa este dată ca suma dintre valoarea unui registru general pe 32 biţi şi o constantă

Exemplu: [EBX+5]

d) Suma a doi regiştri generali pe 32 biţi

Exemplu: [ECX+ESI]

e) Combinaţia celor 2 variante anterioare: suma a 2 regiştri şi a unei constante

Exemplu: [EDX+EBP+14]

f) Suma a 2 regiştri, dintre care unul înmulţit cu 2, 4, sau 8, la care se poate aduna o constantă

Exemple: [EAX+EDI*2], [ECX+EDX*4+5]

Toţi regiştrii generali pe 32 biţi pot fi folosiţi la adresare, fără restricţii şi fără diferenţe între ei. Evident, într-un limbaj de nivel înalt pentru accesarea adreselor de date se folosesc variabile, indicate prin numele lor; la nivelul limbajului de asamblare trebuie precizate adresele corespunzătoare.

Teoretic, prima formă de adresare ar putea fi suficientă. În practică, o situaţie de tipul parcurgerii elementelor unui tablou arată deja că nu se poate lucra numai cu valori constante pentru adrese.

Indicatori de condiţii

Execuţia unor instrucţiuni poate produce, pe lângă rezultatele propriu-zise, şi alte efecte. De exemplu, în cazul operaţiei de adunare, o informaţie de interes este dacă rezultatul depus la destinaţie este corect; reamintim faptul că rezultatul corect poate să nu fie reprezentabil pe numărul de biţi al destinaţiei, caz în care vom primi o valoare eronată.

Pentru asemenea situaţii, procesorul include o serie de indicatori de condiţii (*flags*). Aceştia sunt biţi pe care procesorul îi poziţionează în funcţie de rezultatul calculat de instrucţiunea curentă, pentru a indica eventuala apariţie a unor situaţii speciale. Valorile lor pot fi ulterior testate prin program, astfel încât programatorul poate adapta fluxul de execuţie pe baza condiţiilor testate.

Indicatorii cel mai des utilizaţi sunt următorii:

- *Carry*: arată dacă, în urma unei operaţii de adunare, rezultatul necesită mai mulţi biţi decât are destinaţia
- *Overflow*: arată dacă, în urma unei operaţii de adunare pe numere cu semn, rezultatul obţinut este incorect, datorită apariţiei unei depăşiri

- *Zero*: arată dacă, în urma unei operaţii de adunare/scădere, rezultatul are valoarea zero
- *Sign*: indică semnul rezultatului unei operaţii de adunare/scădere

Rolul indicatorilor de condiţii este foarte important, deoarece acestea sunt singurele condiţii care pot fi testate în limbajul de asamblare. Ca urmare, aşa cum se va vedea ulterior, toate structurile de control din limbajele de nivel înalt pot fi implementate doar prin testarea indicatorilor de condiţii.

Instrucţiuni în limbaj de asamblare în Visual C++

Scopul acestor laboratoare nu este de a învăţa să scriem cod exclusiv în limbaj de asamblare, ci de a scrie porţiuni de cod în limbaj de asamblare inserate în programe C (sau C++).

În mediul de programare Visual Studio, pentru a insera cod scris în limbaj de asamblare într-un program C/C++, se foloseşte cuvântul cheie `_asm`:

```
_asm {
... //comentariile sunt la fel ca în C sau C++
}
```

Cu ajutorul acestui cuvânt cheie pot fi scrise instrucţiuni individuale sau blocuri de instrucţiuni, delimitate în acelaşi mod ca orice bloc de cod C/C++. Instrucţiunile în limbaj de asamblare pot fi separate prin ";" (ca în limbajul C) sau `Enter`. Pentru claritate, se recomandă a doua variantă, deci scrierea câte unei singure instrucţiuni pe un rând.

Instrucţiuni în limbaj de asamblare în gcc

Medii de programare precum Dev-C++ sau CodeBlocks se bazează pe compilatorul gcc. Acesta permite de asemenea scrierea de linii de cod în limbaj de asamblare. În acest sens, esenţial este macroul `asm`, care primeşte ca parametru un şir de caractere, care conţine forma literală a unei instrucţiuni în limbaj de asamblare.

Din păcate, sintaxa implicită pentru limbajul de asamblare este foarte complexă în cazul gcc, ceea ce face foarte dificilă utilizarea sa. Există totuşi posibilitatea de a utiliza (cu unele limitări) sintaxa Intel, întâlnită la Visual C++. Următorul exemplu arată aceeaşi secvenţă scurtă de cod scrisă în Visual C++, respectiv în gcc.

Visual C++:

```
_asm {
 mov eax,5
 mov ebx,5
}
```

gcc:

```
asm(".intel_syntax noprefix");
asm("mov eax,5");
asm("mov ebx,5");
asm(".att_syntax prefix");
```

De remarcat, în cazul gcc, prima şi ultima instrucţiune din bloc, responsabile pentru comutarea către sintaxa Intel şi respectiv înapoi către sintaxa standard a gcc. Absenţa oricăreia dintre aceste două instrucţiuni provoacă fie o eroare de compilare, fie o funcţionare eronată a programului.

În continuare, toate exemplele vor fi date în forma specifică Visual C++. Traducerea în forma gcc se poate realiza relativ uşor.

Instrucţiuni aritmetice

Instrucţiunea de atribuire

mov *destinatie, sursa*

Operanzii pot fi constante, regiştri sau adrese de memorie (variabile în terminologia C). Exemple de combinare a operanzilor:

```
mov eax,ebx
mov cx,5
mov bl,[eax]
mov [esi],edx
```

Regula de bază, care operează practic în cazul tuturor instrucţiunilor în limbaj de asamblare, este că operanzii trebuie să aibă acceaşi dimensiune.

Un caz special apare atunci când un operand este o adresă de memorie, iar celălalt o constantă. În exemplele de mai sus, cel puţin unul dintre operanzi a fost întotdeauna un registru, care impunea dimensiunea celuilalt operand. Aici, atât adresa de memorie, cât şi constanta pot ocupa 1, 2 sau 4 octeţi. Din acest motiv, trebuie ca programatorul să precizeze dimensiunea operanzilor:

```
mov byte ptr [ecx],5  //operanzi pe 1 octet
mov word ptr [ecx],5  //operanzi pe 2 octeţi
mov dword ptr [ecx],5  //operanzi pe 4 octeţi
```

Nu este posibil ca ambii operanzi să fie adrese de memorie:

```
mov byte ptr [eax],[ebx]  //greşit, compilatorul va semnala eroare
```

Toate precizările legate de tipul şi dimensiunea operanzilor sunt valabile şi pentru celelalte instrucţiuni.

Adresele variabilelor declarate într-un program C nu sunt cunoscute de programator. Pentru ca variabilele să poată fi totuşi accesate din limbajul de asamblare, Visual C++ permite folosirea denumirilor simbolice. Cu alte cuvinte, dacă într-un program avem următoarea declaraţie:

```
int a;
```

putem scrie o instrucţiune de tipul următor în cadrul unui bloc `_asm`:

```
mov a,3
```

Compilatorul va genera din această linie o instrucţiune de atribuire care poate fi înţeleasă şi executată de procesor, înlocuind numele variabilei `a` cu adresa sa. Mai mult, se observă că în acest caz nu mai este necesar să se precizeze dimensiunea operanzilor, deşi avem o adresă de memorie şi o constantă, deoarece compilatorul o poate afla din declaraţia variabilei `a` (tipul `int` ocupă 4 octeţi). Dacă însă dorim să accesăm doar octetul cel mai puţin semnificativ din `a`, putem scrie:

```
mov byte ptr a,9
```

Pe de altă parte, următoarea secvenţă de cod nu este corectă:

```
int a=8,b=9;
_asm {
mov a,b
}
```

Ambii operanzi fiind adrese de memorie, compilatorul nu poate genera o instrucţiune pentru procesor care să realizeze transferul direct. Trebuie scris de exemplu:

```
_asm {
mov eax,b
mov a,eax
}
```

(Observaţie: regiştrii `eax` şi `edx` pot fi folosiţi întotdeauna fără teama că modificarea lor ar perturba programul.)

Testare

Să se scrie şi să se execute programul următor:

```
#include <stdio.h>
void main() {
int i=10;
_asm {
mov i,5
}
```

```
printf("%d\n",i);
}
```

Se va urmări valoarea afişată pentru variabila `i`, care confirmă faptul că instrucţiunea `mov` produce efectul descris mai sus.

De asemenea, se vor face testări ale cazurilor de eroare care pot apărea, pentru a vedea reacţia compilatorului.

Adunarea

add *op1*, *op2*

Efect: *op1* += *op2*;

Combinaţii de operanzi - la fel ca la atribuire:

```
add eax,ebx
add dl,3
add si,[...]
add [...],ebp
add byte ptr [...],14
add word ptr [...],14
add dword ptr [...],14
```

În continuare, nu este posibil ca ambii operanzi să fie adrese de memorie:

```
add byte ptr [...],[...]  // eroare
```

În exemplele de mai sus, ca şi în cele care vor urma, prin `[...]` se înţelege un operand de tip adresă de memorie, putând fi folosit oricare mod de adresare posibil.

Suma a două numere este calculată corect de instrucţiunea `add`, indiferent dacă numerele sunt cu semn sau fără semn. Cu alte cuvinte, nu contează dacă operanzii corespund în limbajul C tipurilor `int` sau `unsigned`.

În cazul adunării este posibil ca rezultatul să nu fie corect, dacă numerele sunt prea mari. Pentru numere fără semn, indicatorul *carry* indică prezenţa/absenţa acestei situaţii. Pentru numere cu semn este semnificativ indicatorul *overflow*.

Testare

Să se scrie şi să se execute programul următor, pentru a urmări comportamentul instrucţiunii de adunare:

```
#include <stdio.h>
void main()
{
int a=10;
_asm {
 add a,5
 }
printf("%d\n",a);
}
```

O situaţie care trebuie analizată este calculul sumei a două variabile într-o a treia variabilă:

```
#include <stdio.h>
void main()
{
int a=10,b=5,c;
_asm {
 mov eax,a
 add eax,b
 mov c,eax
 }
printf("%d\n",c);
}
```

Relativ la indicatorii de condiţii, se pot face teste cu tipurile `int` şi `unsigned`:

```
#include <stdio.h>
void main()
{
int i1=1000000000,i2=2000000000;
```

```
printf("%d\n",i1+i2);  // se vede că rezultatul este greşit şi în C, nu e o eroare de programare
_asm {
 mov eax,i2
 add i1,eax
 }
printf("%d\n",i1);
}
#include <stdio.h>
void main()
{
unsigned i1=1000000000,i2=2000000000;
printf("%u\n",i1+i2);  // acum rezultatul este bun
i1=3000000000;
printf("%u\n",i1+i2);  // acum este greşit
_asm {
 mov eax,i2
 add i1,eax
 }
printf("%u\n",i1);
}
```

De remarcat faptul că procesorul poziţionează întotdeauna atât *carry*, cât şi *overflow* (de fapt toţi indicatorii), indiferent dacă lucrăm cu numere cu semn sau fără semn, pentru că nu are de unde şti în care situaţie ne aflăm. Este sarcina programatorului să testeze indicatorul potrivit.

Scăderea

sub *op1*, *op2*
Efect: *op1 -= op2*;

Putem menţiona aici relevanţa indicatorului *zero*, care în acest caz indică egalitatea celor doi operanzi iniţiali. Acest indicator are semnificaţie şi pentru operaţia de adunare, caz în care însă utilitatea sa practică este mai redusă.

Testare

Să se scrie şi să se ruleze pas cu pas programul:

```
#include <stdio.h>
void main()
{
int a=10,b=5;
_asm {
 mov eax,b
 sub a,eax
 }
printf("%d\n",a);
_asm {
 mov eax,b
 sub a,eax
 }
printf("%d\n",a);
}
```

Scopul este de a urmări valoarea indicatorului *zero* după cele două scăderi.

Observaţie: în al doilea bloc `_asm` nu putem presupune că valoarea registrului `eax` a rămas aceeaşi din primul bloc, pentru că între ele s-a apelat funcţia `printf`; nu putem deci renunţa la a doua instrucţiune `mov`.

Incrementarea şi decrementarea

inc *op*
dec *op*
Efect:
op++
op--

Desigur, aceste instrucţiuni pot fi înlocuite cu uşurinţă de către `add`, respectiv `sub`.

Înmulţirea

mul *op*

Efect: operandul este înmulţit cu un registru implicit, iar rezultatul este depus într-o destinaţie de asemenea implicită; atât registrul implicit, cât şi destinaţia depind de dimensiunea operandului. Regula generală este că operanzii (cel explicit şi cel implicit) trebuie să aibă aceeaşi dimensiune, iar destinaţia - dimensiune dublă. Concret:

dimensiune operand explicit	operand implicit	destinaţie rezultat
1	`al`	`ax`
2	`ax`	`(dx,ax)`
4	`eax`	`(edx,eax)`

Dacă dimensiunea operanzilor este 4, rezultatul are 8 octeţi. Cum nu există regiştri atât de mari, se foloseşte perechea `(edx,eax)`, în care registrul `edx` conţine partea mai semnificativă; altfel spus, rezultatul are valoarea $edx \cdot 2^{32}+eax$. În cazul operanzilor pe 2 octeţi, deşi rezultatul ar încăpea într-un registru de 4 octeţi, destinaţie este perechea `(dx,ax)`, pentru a se păstra compatibilitatea cu procesoarele mai vechi, pe 16 biţi.

Operandul explicit poate fi un registru sau o locaţie de memorie, dar nu o constantă:

```
mul ebx  // eax·ebx→(edx,eax)
mul cx
mul al  // se ridică al la pătrat
mul byte ptr [...]  // eax·[...]→(edx,eax)
mul word ptr [...]
mul dword ptr [...]
mul byte ptr 10  // eroare
```

Când operandul explicit este o locaţie de memorie, trebuie precizată explicit şi dimensiunea sa, altfel nu se va şti care este operandul implicit şi nici unde trebuie depus rezultatul.

Instrucţiunea `mul` realizează înmulţirea numerelor fără semn. Pentru înmulţirea numerelor cu semn se foloseşte instrucţiunea `imul`. Aceasta are aceeaşi sintaxă, acceptă aceleaşi tipuri de operanzi şi se comportă la fel, dar operanzii sunt consideraţi numere cu semn şi rezultatul este calculat în consecinţă. Spre deosebire de adunare şi scădere, la înmulţire şi împărţire algoritmi de calcul sunt diferiţi pentru numere cu semn şi fără semn, deci sunt necesare instrucţiuni diferite.

Testare

Să se realizeze calculul factorialului unui număr fără semn. Pentru comparaţie, programul va fi scris mai întâi integral în C.

```
#include <stdio.h>
void main()
{
unsigned n,i,f=1;
for(i=1;i<=n;i++)
 _asm {
      mov eax,f
      mul i
      mov f,eax
 }
printf("%u\n",f);
}
```

În limbajul C nu avem nici o posibilitate de a obţine un rezultat pe 8 octeţi din înmulţirea a două numere pe 4 octeţi (cum este tipul `unsigned`). Pe de o parte nu există un tip întreg de asemenea dimensiune, pe de altă parte compilatorul impune ca rezultatul să aibă acelaşi tip cu operanzii. Restricţia se manifestă şi în programul de mai sus, deci registrul `edx` poate fi ignorat.

Împărţirea

div *op*

Efectul: analog cu instrucţiunea `mul`. Operandul explicit este împărţitorul. Deîmpărţitul este implicit şi depinde de dimensiunea împărţitorului, la fel şi destinaţiile unde se depun câtul şi restul:

dimensiune împărţitor	deîmpărţit	cât	rest
1	`ax`	`al`	`ah`
2	`(dx,ax)`	`ax`	`dx`
4	`(edx,eax)`	`eax`	`edx`

După cum se observă, în toate cazurile, câtul este depus în jumătatea mai puţin semnificativă a deîmpărţitului, iar restul în jumătatea mai semnificativă a acestuia. Acest mod de plasare a rezultatelor permite reluarea operaţiei de împărţire în buclă, dacă este cazul, fără a mai fi nevoie de operaţii de transfer suplimentare.

Analog cu înmulţirea, operandul explicit (împărţitorul) poate fi un registru sau o locaţie de memorie, dar nu o constantă:

```
div ebx
div cx
div dh
div byte ptr [...]
div word ptr [...]
div dword ptr [...]
div byte ptr 10  // eroare
```

La fel ca la înmulţire, pentru numere cu semn există instrucţiunea `idiv`.

Operaţia de împărţire ridică o problemă care nu se întâlneşte în alte părţi: împărţirea la 0. Pentru testare se poate scrie un program de forma:

```
#include <stdio.h>
void main()
{
_asm {
 mov eax,1
 mov edx,1
 mov ebx,0
 div ebx
 }
}
```

Programul va semnala o eroare la execuţie şi va fi terminat forţat.

Se va modifica programul astfel încât valoarea atribuită `ebx` să fie 2 şi se va relua execuţia; nu interesează rezultatul în sine, importantă este observaţia că nu se mai obţine eroare.

Se modifică din nou programul, astfel încât `ebx` primeşte valoarea 1. La rulare se obţine din nou eroare. Cauza este că, în conformitate cu regulile de mai sus, câtul trebuie depus în `eax`, dar dimensiunea sa este prea mare. Într-un asemenea caz, soluţia este de a trece la operanzi de dimensiuni mai mari, ceea ce însă nu este întotdeauna posibil. Tot ce ne putem propune este să detectăm dinainte apariţia unei asemenea erori şi să nu mai executăm împărţirea. În felul acesta programul nu mai este terminat forţat; modul în care procedează mai departe depinde de scopul său şi de programator.

Pentru a vedea cum putem determina apariţia erorii din valorile operanzilor (fără a face împărţirea), considerăm cazul programului de mai sus. Condiţia de apariţie a erorii este:

$$\frac{edx \cdot 2^{32} + eax}{ebx} \geq 2^{32} \Leftrightarrow edx \cdot 2^{32} + eax \geq ebx \cdot 2^{32} \Leftrightarrow eax \geq (ebx - edx) \cdot 2^{32}$$

Deoarece $eax < 2^{32}$ şi valorile regiştrilor sunt numere naturale, singura posibilitate de a satisface inecuaţia (deci de a avea eroare) este ca $ebx \leq edx$. Mai general, condiţia necesară şi suficientă pentru a avea eroare este ca jumătatea mai semnificativă a deîmpărţitului să fie mai mare sau egală cu împărţitorul.

O consecinţă imediată este că instrucţiunile de mai jos produc întotdeauna eroare:

```
div edx
div dx
div ah
```

Un ultim program de test:

```
#include <stdio.h>
void main()
{
```

```
unsigned a=500007,b=10,c,d;
c=a/b;
d=a%b;
printf("%u %u\n",c,d);
}
```

Pentru programul de mai sus trebuie scrise împărţirile în limbaj de asamblare.

Observaţii:

- în limbaj de asamblare este suficientă o singură instrucţiune de împărţire, în schimb trebuie iniţializaţi regiştrii
- există tendinţa de a neglija iniţializarea registrului `edx`; acesta poate duce nu doar la un rezultat greşit, ci şi, aşa cum am văzut mai sus, la terminarea programului cu eroare (în cazul de faţă, dacă se întâmplă ca valoarea lui `edx` să fie cel puţin 10).

Instrucţiuni pe biţi

Instrucţiuni booleene

Sunt implementate operaţiile booleene binare AND, OR, XOR şi cea unară NOT. Denumirile instrucţiunilor sunt chiar acestea, iar operanzii pot fi (la fel ca la atribuire sau adunare) variabile de memorie, regiştri, constante. La fel ca la instrucţiunile studiate anterior, primul operand este şi destinaţia rezultatului:

```
not eax  // se complementează fiecare bit din eax
and bx,16  // se face AND între biţii de pe aceeaşi poziţie din cei doi operanzi, pentru toate poziţiile
or byte ptr [...],100
xor [...],ecx
```

Există aceleaşi limitări privitoare la combinaţiile de operanzi: nu se poate ca ambii operanzi să fie adrese de memorie şi nici ca operanzii să aibă dimensiuni diferite. Evident, aceste restricţii se aplică operaţiilor binare. Ca urmare, instrucţiunile de mai jos sunt greşite:

```
and word ptr [...],[...]
or dx,eax
```

Operatorii echivalenţi din limbajul C sunt ~, &, |, ^, în ordinea exemplelor de mai sus.

La ce folosesc aceşti operatori? La fel ca şi în limbajul C, sunt destul de rar folosiţi. Utilitatea lor principală constă în lucrul cu măşti.

Să considerăm că vrem să testăm bitul de pe poziţia 3 din registrul ax. Ne putem folosi de indicatorul *zero* al procesorului; din păcate, acesta ne dă informaţii doar despre întreg operandul şi nu doar despre un bit al său. Nu ne rămâne deci decât să alterăm registrul ax astfel încât acesta să aibă valoarea 0 dacă şi numai dacă bitul său de pe poziţia 3 este 0. În acest scop, trebuie să forţăm toţi biţii din ax pe valoarea 0, cu excepţia celui de pe poziţia 3, care va păstra valoarea iniţială (cea pe care vrem să o testăm). Funcţia booleană care poate realiza aceste prelucrări (forţarea pe 0, respectiv conservare) este AND. Va trebui deci să facem AND între ax şi un operand construit astfel încât să aibă valoarea 1 pe poziţia 3 şi 0 în rest:

```
and ax,8
```

ax	B_{15}	B_{14}	B_{13}	B_{12}	B_{11}	B_{10}	B_9	B_8	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
8	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
ax	0	0	0	0	0	0	0	0	0	0	0	0	B_3	0	0	0

În concluzie: dacă B_3=0, la final ax=0 va avea în valoarea 0; dacă B_3=1, ax≠0, deci indicatorul *zero*, care dă informaţii despre ax, va indica de fapt valoarea bitului B_3 din ax.

Valoarea constantă cu care se realizează operaţia AND se numeşte mască; se observă că, într-adevăr, rolul său este de a "ascunde" biţii de pe poziţiile care nu interesează, forţându-i la o valoare neutră. Pentru exemplul de mai sus, dacă dorim să testăm bitul de pe poziţia *i*, masca va avea valoarea 2^i.

Desigur, se pot folosi şi celelalte funcţii booleene. De asemenea, măştile pot lăsa nemodificaţi mai mulţi biţi, nu doar unul singur. Mai mult, putem folosi ca mască un registru sau o variabilă (în locul constantei), astfel încât masca se poate modifica pe parcursul execuţiei; depinde doar de ceea ce ne propunem să obţinem. Totuşi, operaţia cel mai des întâlnită este AND. Ajunşi aici, putem face observaţia că, de fapt, instrucţiunea de mai sus nu este cu adevărat o testare, deoarece modifică operandul testat. Oricât de mult ne-ar interesa valoarea bitului de pe poziţia 3, nu ne dorim să pierdem valorile din ceilalţi biţi. Bineînţeles că se pot găsi soluţii; proiectanţii procesorului rezolvă însă problema punându-ne la dispoziţie o instrucţiune specială, numită test. Astfel, în locul instrucţiunii de mai sus, putem scrie:

```
test ax,8
```

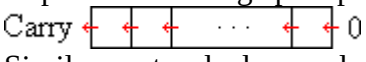
Funcţia booleană realizată este tot AND. Diferenţa constă în faptul că registrul ax nu este modificat - de fapt, rezultatul nu este depus nicăieri. În schimb, flagul *zero* este poziţionat în acelaşi mod. Dacă dorim să facem doar o testare (nedistructivă) şi nu o prelucrare a valorii registrului ax, este exact ceea ce ne trebuie.

Nu există instrucţiuni similare pentru celelalte funcţii logice, ceea ce arată că, într-adevăr, funcţia AND este cea mai folosită.

Instrucţiuni de deplasare

Sunt instrucţiuni care schimbă poziţiile biţilor în cadrul operanzilor.

Deplasarea la stânga pe o poziţie are efectul următor:



Similar pentru deplasarea la dreapta.

Instrucţiunile de deplasare sunt:

`shl eax,1` // deplasare la stânga a registrului `eax` cu o poziţie

`shl byte ptr [...],3` // se poate şi cu mai multe poziţii

`shl dx,cl` /* numărul de poziţii pe care se face deplasarea poate fi specificat în `cl` - singurul registru care poate fi folosit pentru aşa ceva (nu merge nici `cx` sau `ecx`) */

`shr bh,1` // deplasare la dreapta

Desigur, dacă deplasarea se face pe mai multe poziţii, flagul *carry* va reţine doar ultimul bit ieşit.

Operatorii C echivalenţi sunt << şi respectiv >> (mai exact <<= şi >>=).

Deplasările sunt folosite cel mai des în scopuri aritmetice. E bine cunoscut că o deplasare la stânga pe n poziţii este echivalentă cu înmulţirea cu 2^n, în timp ce deplasarea la dreapta reprezintă o împărţire la 2^n. Folosirea deplasărilor cu precădere în acest scop ridică însă şi o problemă: cum procedăm dacă numărul este cu semn? Să considerăm deplasarea la dreapta. Dacă iniţial bitul cel mai semnificativ avea valoarea 1, iar instrucţiunea de deplasare introduce valoarea 0 pe poziţia respectivă, s-a schimbat semnul numărului, ceea ce nu este corect din punct de vedere aritmetic. Din acest motiv, procesorul mai are o instrucţiune de deplasare la dreapta, special pentru numerele cu semn:

```
sar eax,1
```

Efectul său este asemănător cu al instrucţiunii `shr`, dar bitul cel mai semnificativ îşi păstrează valoarea iniţială (şi trece şi spre dreapta, desigur). Se poate demonstra că în acest mod se realizează o împărţire corectă la 2^n. Avem deci două instrucţiuni de deplasare spre dreapta; una e potrivită pentru tipurile de date fără semn (`shr`), cealaltă pentru tipurile cu semn (`sar`).

La deplasarea spre stânga, din păcate, nu există o soluţie similară. Limbajul de asamblare acceptă instrucţiunea `sal`, dar este de fapt alt nume pentru `shl`. E uşor de văzut de fapt că putem interveni doar asupra bitului care "intră", care în acest caz este cel mai puţin semnificativ, deci nu are efect asupra semnului. Evident că se poate găsi şi aici o soluţie, dar trebuie scrise mai multe instrucţiuni.

Testare

Să se execute următoarea o secvenţă de cod:

```
unsigned u;
for(u=1;u!=0;u<<=1)
    printf("%u\n",u);
```

Apoi se va schimba codul astfel:

```
unsigned u=1;
while(u!=0) {
    printf("%u\n",u);
    _asm {
        shl u,1
    }
}
```

Se va observa faptul că, în cele două cazuri, comportarea este aceeaşi.

Instrucţiuni de salt

Din punct de vedere practic, o instrucţiune de salt modifică valoarea registrului contor program şi implicit face ca instrucţiunea următoare care se execută să nu fie cea care urmează în memorie. Utilitatea acestor instrucţiuni este clară: orice structură de control (testare sau buclă) se poate implementa la nivelul limbajului procesorului doar prin salturi.

Saltul necondiţionat

jmp *adresa*

Deşi există şi alte forme de exprimare a adresei de salt, în continuare ne vom rezuma la cazul în care adresa este o constantă. Evident, la momentul scrierii codului nu putem cunoaşte adresa reală din memorie; ca urmare, vom lăsa gestiunea adreselor în seama compilatorului, prin utilizarea etichetelor, la fel ca în cazul instrucţiunii `goto` din C. Visual C++ este flexibil în această privinţă: se pot face salturi spre etichete definite în acelaşi bloc `_asm`, în alt bloc sau în partea de cod C. De altfel, este posibil şi invers: o instrucţiune `goto` să sară la o etichetă definită într-un bloc `_asm`.

Testare

```
int i;
_asm {
 mov i,16
 jmp et
 mov i,5
et:
 add i,3
}
printf("%d\n",i);
```

La rularea programului de mai sus se observă că variabila `i` are la final valoarea 19, deci se execută adunarea, dar nu şi a doua atribuire.

Utilitatea saltului necondiţionat, luat singur, nu este foarte mare, deoarece nu introduce cu adevărat o ramificaţie în program - nu avem variante de execuţie.

Salturi condiţionate

În acest caz avem două variante de execuţie:

- condiţia de salt este adevărată - se face saltul la adresa indicată
- condiţia de salt este falsă - se continuă cu instrucţiunea următoare din memorie, ca şi cum n-ar fi existat nici o instrucţiune de salt

Cele mai simple sunt instrucţiunile care testează indicatorii de condiţii individuali. Vom considera în continuare doar indicatorii cei mai folosiţi: *carry*, *overflow*, *zero* şi *sign*.

Pentru fiecare indicator există două instrucţiuni de salt condiţionat: una care face saltul când indicatorul testat are valoarea 1 şi una care face saltul când indicatorul are valoarea 0.

indicator testat	**salt pe valoarea 1**	**salt pe valoarea 0**
Carry	`jc`	`jnc`
Overflow	`jo`	`jno`
Zero	`jz`	`jnz`
Sign	`js`	`jns`

Evident, pentru fiecare instrucţiune de mai sus trebuie precizată şi adresa de salt.

În realitate, aceşti indicatori individuali nu sunt foarte utili. După cum se ştie, în limbajul C se fac teste în principal pe baza operatorilor relaţionali (<, <=, ==, !=, >, >=). În general, aceste relaţii între operanzi nu pot fi determinate testând câte un singur indicator, ci sunt necesare instrucţiuni mai complexe. În acest sens, considerăm mai întâi instrucţiunea de comparare:

cmp *op1,op2*

Combinaţiile de operanzi care pot fi folosiţi sunt aceleaşi ca la instrucţiunile studiate anterior, începând cu atribuirea. Această instrucţiune realizează intern o scădere, însă fără a altera operanzii (nu depune rezultatul nicăieri); în schimb, indicatorii de condiţii sunt poziţionaţi la fel. Putem deci folosi în acelaşi scop şi instrucţiunea `sub`, dacă nu ne afectează modificarea primului operand. Cel mai important este faptul că, analizând toţi indicatorii (nu doar unul), putem decide care a fost relaţia între operanzii instrucţiunii de

comparare. Deoarece ar fi prea complicat să scriem mai multe salturi condiţionate relaţionate între ele, proiectanţii procesoarelor Intel au introdus un set de salturi condiţionate care fac testele corespunzătoare:

relaţie	instrucţiune salt
op1 < op2	jb
op1 <= op2	jbe
op1 > op2	ja
op1 >= op2	jae
op1 == op2	je
op1 != op2	jne

Problemă: relaţiile între operanzi diferă în funcţie de tipul acestora (cu semn/fără semn). De exemplu, să presupunem că operanzii arată astfel:

01001011
10011010

Dacă operanzii sunt fără semn, evident al doilea operand este mai mare. Dacă însă sunt numere cu semn, primul operand e pozitiv, iar al doilea negativ, astfel încât relaţia între ei se inversează. Pe de altă parte, instrucţiunea de comparaţie rămâne aceeaşi. Instrucţiunile de mai sus sunt de fapt pentru operanzi fără semn. Pentru operanzi cu semn avem alt set de instrucţiuni:

relaţie	instrucţiune salt
op1 < op2	jl
op1 <= op2	jle
op1 > op2	jg
op1 >= op2	jge
op1 == op2	je
op1 != op2	jne

Se observă (cum era de aşteptat) că testele de egalitate şi inegalitate sunt identice în cele două cazuri.

Putem face observaţia că unele din aceste instrucţiuni testează un singur indicator, deci sunt identice cu unele din primul set. De exemplu, `je` şi `jz` reprezintă de fapt aceeaşi instrucţiune. Nu este însă necesar să reţinem aceste detalii.

Exemplu:

Presupunem că avem o variabilă `a` (întreg fără semn) şi dorim să numărăm biţii cu valoarea 1 din aceasta. Pentru simplitate, considerăm că ne permitem să alterăm valoarea variabilei `a`. O variantă ar fi următoarea (scrisă integral în C):

```
nr=0;
for(i=0;i<32;i++) {
 if(a&1) nr++;
 a>>=1;
}
printf("%u\n",nr);
```

De acum avem cunoştinţele necesare pentru a scrie integral acest cod în limbaj de asamblare, mai puţin afişarea. Pentru început putem lucra tot cu variabile (apoi, ca exerciţiu, le putem înlocui pe unele cu regiştri):

```
_asm {
 mov nr,0
 mov i,0
e1:
 cmp i,32
 jae e3
 test a,1
 jz e2
 add nr,1
e2:
 shr a,1
 jmp e1
e3:
}
printf("%u\n",nr);
```

Structuri de control

În continuare vom studia modul de implementare la nivelul limbajului de asamblare al structurilor de control specifice limbajului C.

Structura if

În cazul structurii `if` care nu are ramura `else` este suficientă o singură instrucţiune de salt. În limbajul de asamblare, saltul se face în cazul condiţiei negate faţă de codul C echivalent. Explicaţia acestei inversări este simplă: în limbajul C, o condiţie adevărată are ca efect executarea unui set de instrucţiuni; în limbaj de asamblare, o condiţie adevărată are ca efect efectuarea unui salt, deci evitarea execuţiei unor instrucţiuni.

Ca exemplu, considerăm următorul cod C:

```
if(x>5)
 x--;
```

Conform celor de mai sus, traducerea în limbaj de asamblare este:

```
 cmp x,5
 jle nu
 dec x
nu:
```

Observaţie: S-a considerat că variabila `x` este de tip `int` (întreg cu semn), motiv pentru care s-a folosit instrucţiunea `jle` şi nu `jbe`.

Dacă apare şi o ramură `else`, trebuie utilizate două instrucţiuni de salt. A doua instrucţiune de salt este de tip necondiţionat şi are ca rol de a asigura că, la terminarea execuţiei instrucţiunilor din prima ramură, nu vor fi executate şi instrucţiunile din a doua ramură.

Cod C:

```
if(x>5)
 x--;
else
 x++;
```

Cod în limbaj de asamblare:

```
 cmp x,5
 jle nu
 dec x       // ramura executată în cazul x>5
 jmp afara   // salt peste a doua ramură
nu:
 inc x       // ramura executată în cazul x≤5
afara:
```

Structura while

O structură de tip `while` implică de asemenea utilizarea a două instrucţiuni de salt: una pentru a decide dacă se execută o nouă iteraţie sau se părăseşte bucla; a doua, la finalul corpului buclei, pentru a relua cu o nouă iteraţie. La fel ca în cazul structurii `if`, la prima instrucţiune saltul se face pe condiţia inversă celei din codul C, iar a doua instrucţiune este de salt necondiţionat.

Cod C:

```
while(x<10)
 x++;
```

Cod în limbaj de asamblare:

```
bucla:
 cmp x,5
 jge afara
 inc x       // corpul buclei
 jmp bucla
afara:
```

Structura for

Structura de tip `for` este semantic similară celei de tip `while`. Ca urmare, implementarea sa în limbaj de asamblare este de asemenea similară. Singura diferenţă constă în apariţia unor instrucţiuni auxiliare, care delimitează mai clar iniţializarea şi respectiv actualizarea valorilor variabilelor pentru iteraţia următoare. Trebuie deci respectată cu stricteţe ordinea acestor instrucţiuni componente:

1. iniţializarea (executată o singură dată)
2. testul de continuare cu o nouă iteraţie sau ieşire
3. corpul buclei
4. actualizare variabile pentru iteraţia următoare
5. reluare de la pasul 2

Structura do...while

Structura de tip `do...while` este asemănătoare celei de tip `while`, dar testul de continuare/ieşire este executat după corpul buclei. Este singura structură repetitivă în care condiţia de salt în limbajul de asamblare este aceeaşi ca în limbajul C.

Cod C:

```
do
 x++;
while(x<10);
```

Cod în limbaj de asamblare:

```
bucla:
inc x  // corpul buclei
cmp x,10
jl bucla
```

Lucrul cu stiva şi utilizarea funcţiilor

Instrucţiuni pentru lucrul cu stiva

Întrucât o structură de tip stivă este necesară în multe situaţii, procesorul foloseşte şi el o parte din memoria RAM pentru a o accesa printr-o disciplină de tip LIFO. După cum se ştie, singura informaţie fundamentală pentru gestiunea stivei este vârful acesteia. În cazul procesorului, adresa la care se află vârful stivei este memorată în registrul ESP.

Instrucţiunea de introducere în stivă se numeşte, evident, `push` şi are un singur operand. Exemple:

```
push eax
push dx
push dword ptr [...]
push word ptr [...]
push dword ptr 5
push word ptr 14
```

Se observă că stiva lucrează doar cu valori de 2 sau 4 octeţi. De fapt, pentru uniformitate, se recomandă a lucra numai cu operanzi pe 4 octeţi; varianta cu 2 octeţi este prezentă doar pentru compatibilitatea cu procesoarele mai vechi.

Ce se întâmplă când se execută o asemenea instrucţiune? Procesorul scrie valoarea operandului la adresa indicată de ESP (vârful stivei), după ce mai întâi scade din valoarea ESP dimensiunea în octeţi a operandului care va fi pus în stivă (2 sau 4). De exemplu, instrucţiunea `push eax` ar fi echivalentă cu:

```
sub esp,4
mov [esp],eax
```

Diferenţa e dată de faptul că procesorul realizează singur toate aceste acţiuni (şi astfel reduce riscul apariţiei erorilor). Observăm aici că stiva avansează "în jos", adică de la adrese mari spre adrese mici.

Instrucţiunea inversă, de extragere din stivă, se numeşte `pop`. După cum era de aşteptat, lucrează tot numai cu operanzi de 2 sau 4 octeţi:

```
pop eax
pop cx
pop dword ptr [...]
pop word ptr [...]
```

Evident, la execuţia acestei instrucţiuni, operandul joacă rolul destinaţiei în care se depune valoarea luată din vârful stivei (ESP); apoi, la valoarea ESP se adună numărul de octeţi ai operandului.

Rolul stivei procesorului este de a stoca informaţii cu caracter temporar. De exemplu, dacă avem nevoie să folosim un registru pentru unele operaţii, dar nu avem la dispoziţie nici un registru a cărui valoare curentă să ne permitem să o pierdem, procedăm ca mai jos:

```
push eax
... // utilizare eax
pop eax
```

În acest mod am putut folosi temporar registrul `eax`, dar valoarea sa iniţială a fost păstrată şi restabilită la final.

De asemenea, variabilele locale sunt plasate tot în stivă. Reamintim că o variabilă locală este creată la momentul apelului funcţiei în care este declarată şi distrusă la terminarea acelei funcţii, deci are tot un caracter temporar. (Excepţie fac variabilele locale statice, despre care nu discutăm aici.)

Evident, lucrul cu stiva necesită multă atenţie; instrucţiunile de introducere în stivă trebuie riguros compensate de cele de scoatere din stivă - aici este vorba atât de numărul acestor instrucţiuni, cât şi de dimensiunea operanzilor. Orice eroare afectează în general mai multe date. De exemplu, să considerăm secvenţa următoare:

```
push eax
push edx
mov eax,[esi]
mov edx,5
mul edx
mov [esi],eax
pop eax
```

Înmulţirea variabilei cu 5 se realizează corect. În schimb, pentru că o instrucţiune `pop` a fost uitată, sunt afectate valorile ambilor regiştri implicaţi: `eax` primeşte altă valoare decât avea iniţial, iar valoarea `edx` rămâne cea rezultată din înmulţire, fără a fi restaurată cea veche. Similar stau lucrurile dacă se execută prea multe instrucţiuni `pop`. În majoritatea cazurilor, operandul depus în stivă printr-o instrucţiune `push` este şi destinaţia instrucţiunii `pop` corespunzătoare (dar nu este 100% obligatoriu, depinzând de natura programului).

O altă eroare poate apărea atunci când registrul ESP este manipulat direct. De exemplu, pentru a aloca spaţiu unei variabile locale (neiniţializată), e suficient a scădea din ESP dimensiunea variabilei respective. Similar, la distrugerea variabilei, valoarea ESP este crescută. Aici nu se folosesc în general instrucţiuni `push`, repectiv `pop`, deoarece nu interesează valorile implicate, ci doar ocuparea şi eliberarea de spaţiu. Se preferă adunarea şi scăderea direct cu registrul ESP; evident că o eroare în aceste operaţii are consecinţe de aceeaşi natură ca şi cele de mai sus.

Apelul funcţiilor

Un apel de funcţie arată la prima vedere ca o instrucţiune de salt, în sensul că se întrerupe execuţia liniară a programului şi se sare la o cu totul altă adresă. Diferenţa fundamentală constă în faptul că la terminarea funcţiei se revine la adresa de unde s-a făcut apelul şi se continuă cu instrucţiunea următoare. Din moment ce într-un program se poate apela o funcţie de mai multe ori, din mai multe locuri, şi întotdeauna se revine unde trebuie, este clar că adresa la care trebuie revenit este memorată şi folosită atunci când este cazul. Cum adresa de revenire este în mod evident o informaţie temporară, locul său este tot pe stivă.

Apelul unei funcţii se realizează prin instrucţiunea `call`, care are următoarea sintaxă:

call *adresa*

Evident, în Visual C++ vom folosi nume simbolice pentru a preciza adresa, cu singura menţiune că de data acesta nu este vorba de etichete, ca la salturi, ci chiar de numele funcţiilor apelate.

Efectul instrucţiunii `call` este următorul: se introduce în stivă adresa instrucţiunii următoare (adresa de revenire) şi se face salt la adresa indicată. Aceste acţiuni puteau fi realizate şi cu instrucţiuni `push` şi `jmp`, dar din nou procesorul face singur totul şi astfel ne fereşte de erori.

Revenirea dintr-o funcţie se face prin instrucţiunea `ret`, care preia adresa de revenire din vârful stivei (similar unei instrucţiuni `pop`) şi face saltul la adresa respectivă.

Acum devine clar că o eroare în lucrul cu stiva are consecinţe şi mai grave decât s-a văzut anterior. O instrucţiune `pop` omisă (sau una în plus) are ca efect faptul că, la execuţia unei instrucţiuni `ret`, se va prelua din stivă o cu totul altă valoare decât adresa corectă de revenire. Efectul practic al acestui salt greşit este blocarea programului sau terminarea sa forţată.

Parametri

Parametrii unei funcţii sunt tot variabile locale, deci se găsesc pe stivă. Cel care face apelul are responsabilitatea de a-i pune pe stivă la apel şi de a-i scoate de pe stivă la revenirea din funcţia apelată. Avem la dispoziţie instrucţiunea `push` pentru plasarea în stivă. Evident, această operaţie trebuie realizată imediat înainte de apelul propriu-zis. În plus, în limbajul C/C++, parametrii trebuie puşi în stivă în ordine inversă celei în care se găsesc în lista de parametri. La revenire, parametrii trebuie scoşi din stivă, nemaifiind necesari. Cum nu ne interesează preluarea valorilor lor, nu se foloseşte instrucţiunea `pop`, ci se adună la ESP numărul total de octeţi ocupat de parametri (reamintim faptul că pe stivă se lucrează în general cu 4 octeţi, chiar şi atunci când operanzii au dimensiuni mai mici).

Să luăm ca exemplu funcţia următoare:

```
void dif(int a,int b)
{
 int c;
 c=a-b;
 printf("%d\n",c);
}
```

Apelul `dif(9,5)` se traduce prin secvenţa de mai jos:

```
push dword ptr 9
push dword ptr 5
call dif
add esp,8
```

În practică, parametrii pot fi accesaţi fie prin numele lor, aşa cum sunt acestea precizate în antetul funcţiei (forma cea mai simplă), fie prin intermediul adreselor lor (mai puţin intuitiv, dar uneori nu poate fi evitat). Pentru precizarea adreselor parametrilor poate fi utilizat registrul EBP, astfel:

- Primul parametru din antetul funcţiei se găseşte întotdeauna la adresa EBP+8.
- Următorii parametri, mergând spre dreapta în lista parametrilor, sunt la adrese corespunzători mai mari: EBP+12, EBP+16 etc.

Returnarea valorilor

Cum poate o funcţie să returneze o valoare? Convenţia în Visual C++ (şi la majoritatea compilatoarelor) este că rezultatul se depune într-un anumit registru, în funcţie de dimensiunea sa:

- pentru tipurile de date de dimensiune 1 octet - în registrul `al`
- pentru tipurile de date de dimensiune 2 octeţi - în registrul `ax`
- pentru tipurile de date de dimensiune 4 octeţi - în registrul `eax`
- pentru tipurile de date de dimensiune 8 octeţi (există şi aşa ceva) - în regiştii `edx` şi `eax`

Evident, la revenirea din funcţie, cel care a făcut apelul trebuie să preia rezultatul din registrul corespunzător.

Păstrarea valorilor regiştrilor

Un alt aspect important: programul pe care îl scriem este întotdeauna C/C++, deci nu putem şti cum foloseşte compilatorul regiştrii. Ca urmare, dacă o funcţie scrisă de noi în limbaj de asamblare foloseşte unii regiştri, aceştia trebuie salvaţi în stivă la începutul execuţiei funcţiei şi refăcuţi la terminare. Excepţie fac regiştrii `eax` şi `edx`, care, după cum am văzut, sunt folosiţi pentru a returna valori, deci îi putem modifica întotdeauna fără a fi necesară salvarea/restaurarea lor. Tot din motive de conlucrare cu Visual C++, nu vom scrie niciodată `ret`. Motivul este acela că la terminarea funcţiei mai trebuie realizate şi alte acţiuni, de care noi nu avem cunoştinţă. Compilatorul este cel care generează instrucţiunea `ret` - şi o face în mod corect.

Exerciţii

- Să se implementeze în limbaj de asamblare funcţia `dif` prezentată mai sus. Variabila `c` nu va mai fi necesară, din moment ce avem regiştrii la dispoziţie. Apelul către `printf` va rămâne însă în limbajul C, pentru că nu avem încă toate cunoştinţele necesare pentru a-l scrie în limbaj de asamblare.
- Să se modifice forma funcţiei implementate anterior, astfel încât să nu afişeze diferenţa calculată, ci să o returneze. La revenire rezultatul trebuie preluat din `eax` într-o variabilă, pentru că nu poate fi afişat direct dintr-un registru.
- Să se implementeze în limbaj de asamblare o funcţie recursivă care calculează factorialul unui număr natural, dat ca parametru.

Tablouri şi structuri

Tablouri şi pointeri

Deoarece discuţia se mută aici spre manipularea unor structuri de date compuse, să considerăm ca exemplu concret calculul sumei elementelor unui tablou de numere întregi:

```
int t[]={4,9,12,3,7};
int i,suma=0;
for(i=0;i<5;i++)
 suma+=t[i];
```

Toate conceptele implicate au fost deja discutate, mai puţin modul de accesare din limbaj de asamblare al elementelor unui tablou. Prima observaţie este că, evident, adresa acestui element (necesară pentru a-l putea accesa) poate fi determinată pe baza a două informaţii: adresa de început a tabloului, respectiv deplasamentul elementului în cadrul tabloului.

Pentru prima componentă se va folosi direct numele variabilei, în cazul de faţă `t`. Compilatorul înlocuieşte în mod curent numele variabilelor cu adresa la care se află acestea în memorie, deci este aplicat şi aici mecanismul general.

În ceea ce priveşte indicele, nu este posibilă utilizarea unei variabile de memorie (cum este aici `i`) pentru accesarea altei variabile, deci va trebui folosit în loc un registru. Codul de mai jos arată rezolvarea problemei propuse:

```
 mov ebx,0 // înlocuieşte variabila suma
 mov eax,0 // înlocuieşte variabila i
bucla:
 cmp eax,5
 jge afara
 add ebx,t[eax*4]
 inc eax
 jmp bucla
afara:
```

Se observă un detaliu important: în timp ce în limbajul C indicele este exprimat în numărul de elemente faţă de începutul tabloului, în limbajul de asamblare deplasamentul este exprimat în octeţi. Din acest motiv, deşi registrul `eax` ia aceleaşi valori ca şi variabila `i`, la calculul adresei fiecărui element valoarea sa trebuie înmulţită cu 4 (deoarece elementele tabloului, fiind de tip `int`, au dimensiunea de 4 octeţi).

Observaţie: În locul sintaxei `t[eax*4]` poate fi folosită sintaxa echivalentă `t+eax*4`, care este mai sugestivă în unele contexte.

Să considerăm acum o variabilă de tip pointer, care este primeşte ca valoare adresa de început a tabloului `t`:

```
int *p=t;
```

Dacă vom scrie din nou codul C prin care calculăm suma elementelor, utilizând de data aceasta variabila `p` în locul `t`, singura diferenţă va fi că vom scrie `p[i]` în loc de `t[i]`. La nivelul limbajului de asamblare, lucrurile nu stau la fel. Variabila `p` nu poate fi folosită pentru a exprima adresa de început a tabloului. În schimb, valoarea sa trebuie copiată într-un registru, care va fi utilizat ca adresă de start:

```
 mov ebx,0
 mov eax,0
 mov ecx,p // preia adresa de început a tabloului
bucla:
 cmp eax,5
 jge afara
 add ebx,[ecx+eax*4]
 inc eax
 jmp bucla
afara:
```

Tablourile ca parametri ai funcţiilor

În limbajele C/C++, tablourile nu sunt niciodată transmise în mod direct ca parametri pentru funcţii. Pentru a economisi memoria, întotdeauna este transmisă ca parametru adresa de început a tabloului. Ca urmare, următoarele antete de funcţii sunt interschimbabile:

```
void f(int x[]);
void f(int *x);
```

În ambele cazuri, parametrul `x` este un pointer care conţine adresa de început a tabloului. Tot ca urmare a acestei suprapuneri, funcţia `f` de mai sus, în oricare dintre cele două forme, poate primi la apel, ca parametru efectiv, fie un tablou (caz în care pe stivă va fi pusă adresa sa de început), fie un pointer. Utilizând variabilele introduse mai sus, ambele apeluri sunt valide şi produc acelaşi efect:

```
f(t);
f(p);
```

Tablouri bidimensionale şi pointeri

Să considerăm o matrice bidimensională de dimensiune 3×3. Aceasta poate fi alocată static (la declararea tabloului) sau dinamic:

```
int t[3][3];
int **p;
p=new int*[3];
for(i=0;i<3;i++)
 p[i]=new int[3];
```

Fie de asemenea o funcţie care primeşte ca parametru o matrice. Similar cazului unidimensional, sunt două forme sintactice de scriere a antetului funcţiei:

```
void f(int x[][3]);
void f(int **x);
```

De această dată, cele două forme nu mai sunt interschimbabile. Prima formă acceptă ca parametru efectiv doar tabloul `t`, în timp ce a doua formă acceptă doar pointerul `p`. Diferenţa vine din faptul că, spre deosebire de cazul unidimensional, modul de accesare al elementelor celor două tipuri de tablouri este foarte diferit.

a) În cazul tabloului `t`, deşi formal este bidimensional, elementele sunt plasate liniar în memorie. Începând de la adresa de început a tabloului, în memorie se află în ordine elementele `t[0][0]`, `t[0][1]`, `t[0][2]`, `t[1][0]` ş.a.m.d. Prin urmare, pentru a determina adresa unui element oarecare `t[i][j]`, deplasamentul său faţă de începutul tabloului este `i`·3+`j`, unde 3 este numărul de coloane al matricii. Menţionăm faptul că, în cazul primei forme a funcţiei `f`, parametrul `x` reprezintă adresa de început a matricii, la fel ca la tablourile unidimensionale.

b) În al doilea caz, accesul la elementul `p[i][j]` se face în doi paşi. În primul pas, pornind de la valoarea pointerului `p`, se accesează elementul `p[i]`. În al doilea pas, pornind de la valoarea `p[i]`, care reprezintă la rândul său un pointer conţinând adresa de început a unui tablou cu 3 elemente de tip `int`, se accesează `p[i][j]`.

În aceste condiţii devine evident faptul că, de exemplu, un tablou precum `t`, ale cărui elemente trebuie accesate în modul descris la punctul a, nu poate fi transmis ca parametru efectiv celei de-a doua forme a funcţiei `f`, care presupune că elementele sunt accesate în modul descris la punctul b.

Structuri

Structurile au în comun cu tablourile faptul că sunt tipuri compuse, formate din elemente plasate unul după altul în memorie. În schimb, componentele unei structuri pot avea tipuri diferite şi, implicit, nu mai pot fi accesate pe baza unui indice, ci doar pe baza deplasamentului în cadrul structurii.

Să considerăm următorul exemplu:

```
struct S {
 char a,b;
 int c;
 };
```

La prima vedere, o variabilă de acest tip va ocupa 6 octeţi, iar câmpurile `a`, `b` şi `c` sunt plasate în memorie, faţă de adresa de început a structurii, la deplasamentele 0, 1 şi respectiv 2. În practică, majoritatea compilatoarelor, inclusiv Visual C++, folosesc o tehnică numită alinierea adreselor, care impune ca fiecare câmp să înceapă la un deplasament multiplu de dimensiunea câmpului respectiv. În exemplul nostru, câmpurile `a` şi `b`, care sunt de tipul `char` şi deci ocupă câte un octet, sunt plasate în continuare la deplasamentele 0 şi

respectiv 1; pe de altă parte, câmpul `c`, fiind de tip `int` şi ocupând 4 octeţi, se află la deplasamentul 4, iar structura în ansamblul său ocupă 8 octeţi. În mod evident, între câmpurile `b` şi `c` există 2 octeţi neutilizaţi.

Fie următoarea structură, folosită pentru reprezentarea punctelor în plan:

```
struct point {
 int x;
 int y;
};
```

Date fiind două variabile `p1` şi `p2` de tip `point`, ne propunem scrierea în limbaj de asamblare a codului care determină mijlocul segmentului dintre cele două puncte şi depune coordonatele acestuia tot în `p1`.

Observăm că, în cadrul structurii `point`, câmpul `x` este plasat la deplasament 0 (deci începe la aceeaşi adresă la care începe structura), iar câmpul `y` la deplasament 4. În principal, codul constă în execuţia a două adunări, pentru câmpurile `x` şi `y` ale celor două variabile:

```
mov eax,dword ptr p2
add dword ptr p1,eax
mov eax,dword ptr p2+4
add dword ptr p1+4,eax
```

În acest caz a trebuit folosit modificatorul `dword ptr` pentru a indica faptul că operanzii implicaţi au dimensiunea de 4 octeţi. La utilizarea numelor `p1` şi `p2`, compilatorul observă că ele corespund unor variabile de dimensiune 8 octeţi, astfel încât ar semnala eroare la utilizarea lor în conjuncţie cu un registru de 4 octeţi.

Alternativ, este posibilă utilizarea sintaxei `[...]` pentru a indica deplasamentul câmpului în cadrul structurii:

```
mov eax,dword ptr p2
add dword ptr p1,eax
mov eax,dword ptr p2[4]
add dword ptr p1[4],eax
```

Exerciţiu

Rescrieţi codul de mai sus pentru a implementa o funcţie care primeşte ca parametri adresele a două structuri de tip `point`:

```
void middle(point *p1,point *p2);
```

Funcţia depune rezultatul în structura indicată de primul operand.

Parametrii sunt pointeri deoarece limbajul C (spre deosebire de C++) nu permite transmiterea directă a parametrilor de tip structură.