

Elemente de bază ale limbajului de asamblare

Regiștrii procesoarelor Intel

Orice procesor conține un număr foarte mare de regiștri, utilizați pentru diverse acțiuni interne. Aceștia nu prezintă interes pentru programator. În schimb există o categorie specială, cea a regiștrilor de uz general, care pot fi folosiți explicit în programe, în mod similar variabilelor de memorie. În cazul procesoarelor din familia Intel x86 pe 32 biți, regiștrii 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

Regiștrii generali pot fi folosiți atât pentru a reține date (toți), cât și pentru formarea adreselor locații de memorie (numai cei pe 32 biți).

Atenție: regiștrii din tabelul de mai sus nu sunt complet independenți. De exemplu, faptul că registrul EAX îl conține pe AX înseamnă că orice modificare a valorii lui AX afectează automat și valoarea lui EAX și reciproc; similar pentru regiștrii 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 biți

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

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

Exemplu: [EBX+5]

d) Suma a doi regiștri generali pe 32 biți

Exemplu: [ECX+ESI]

e) Combinația celor 2 variante anterioare: suma a 2 regiștri și a unei constante

Exemplu: [EDX+EBP+14]

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

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

Toți regiștrii generali pe 32 biți pot fi folosiți la adresare, fără restricții și fără diferențe 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 situație de tipul parcurgerii elementelor unui tablou arată deja că nu se poate lucra numai cu valori constante pentru adrese.

Indicatori de condiții

Execuția unor instrucțiuni poate produce, pe lângă rezultatele propriu-zise, și alte efecte. De exemplu, în cazul operației de adunare, o informație de interes este dacă rezultatul depășește la destinație este corect; reamintim faptul că rezultatul corect poate să nu fie reprezentabil pe numărul de biți al destinației, caz în care vom primi o valoare eronată.

Pentru asemenea situații, procesorul include o serie de indicatori de condiții (*flags*). Aceștia sunt biți pe care procesorul îi poziționează în funcție de rezultatul calculat de instrucțiunea curentă, pentru a indica eventuala apariție a unor situații speciale. Valorile lor pot fi ulterior testate prin program, astfel încât programatorul poate adapta fluxul de execuție pe baza condițiilor testate.

Indicatorii cel mai des utilizați sunt următorii:

- *Carry*: arată dacă, în urma unei operații de adunare, rezultatul necesită mai mulți biți decât are destinația

- *Overflow*: arată dacă, în urma unei operații de adunare pe numere cu semn, rezultatul obținut este incorect, datorită apariției unei depășiri

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

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

Instrucțiuni în limbaj de asamblare în Visual C++

Scopul acestor laboratoare nu este de a învăța să scriem cod exclusiv în limbaj de asamblare, ci de a scrie porțiuni 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 folosește cuvântul cheie `_asm`:

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

Cu ajutorul acestui cuvânt cheie pot fi scrise instrucțiuni individuale sau blocuri de instrucțiuni, delimitate în același mod ca orice bloc de cod C/C++. Instrucțiunile 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 instrucțiuni pe un rând.

Instrucțiuni 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, esențial este macroul `asm`, care primește ca parametru un șir de caractere, care conține forma literală a unei instrucțiuni 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 totuși posibilitatea de a utiliza (cu unele limitări) sintaxa Intel, întâlnită la Visual C++. Următorul exemplu arată aceeași secvență 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 și ultima instrucțiune din bloc, responsabile pentru comutarea către sintaxa Intel și respectiv înapoi către sintaxa standard a gcc. Absența oricăreia dintre aceste două instrucțiuni provoacă fie o eroare de compilare, fie o funcționare eronată a programului.

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

Instrucțiuni aritmetice

Instrucțiunea de atribuire

mov *destinație, sursa*

Operandii pot fi constante, regiștri sau adrese de memorie (variabile în terminologia C). Exemple de combinare a operandilor:

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

Regula de bază, care operează practic în cazul tuturor instrucțiunilor în limbaj de asamblare, este că operandii trebuie să aibă aceeași dimensiune.

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

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

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

```
mov byte ptr [eax], [ebx] //greșit, compilatorul va semnaliza eroare
```

Toate precizările legate de tipul și dimensiunea operandilor sunt valabile și pentru celelalte instrucțiuni.

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

```
int a;
```

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

```
mov a, 3
```

Compilatorul va genera din această linie o instrucțiune de atribuire care poate fi înțeleasă și 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 operandilor, deși avem o adresă de memorie și o constantă, deoarece compilatorul o poate afla din declarația variabilei `a` (tipul `int` ocupă 4 octeți). Dacă însă dorim să accesăm doar octetul cel mai puțin semnificativ din `a`, putem scrie:

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

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

```
int a=8, b=9;
```

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

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

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

(Observație: regiștrii `eax` și `edx` pot fi folosiți întotdeauna fără teama că modificarea lor ar perturba programul.)

Testare

Să se scrie și 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 afișată pentru variabila *i*, care confirmă faptul că instrucțiunea *mov* produce efectul descris mai sus.

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

Adunarea

add *op1, op2*

Efect: *op1* += *op2*;

Combinații 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 și în cele care vor urma, prin [...] se înțelege un operand de tip adresă de memorie, putând fi folosit oricare mod de adresare posibil.

Suma a două numere este calculată corect de instrucțiunea *add*, indiferent dacă numerele sunt cu semn sau fără semn. Cu alte cuvinte, nu contează dacă operandii 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 prezența/absența acestei situații. Pentru numere cu semn este semnificativ indicatorul *overflow*.

Testare

Să se scrie și să se execute programul următor, pentru a urmări comportamentul instrucțiunii de adunare:

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

O situație 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 condiții, se pot face teste cu tipurile *int* și *unsigned*:

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

```

printf("%d\n", i1+i2); // se vede că rezultatul este greșit și 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 greșit
    _asm {
        mov eax, i2
        add i1, eax
    }
    printf("%u\n", i1);
}

```

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

Scăderea

sub *op1, op2*

Efect: *op1 -= op2*;

Putem menționa aici relevanța indicatorului *zero*, care în acest caz indică egalitatea celor doi operanzi inițiali. Acest indicator are semnificație și pentru operația de adunare, caz în care însă utilitatea sa practică este mai redusă.

Testare

Să se scrie și 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.

Observație: în al doilea bloc `_asm` nu putem presupune că valoarea registrului `eax` a rămas aceeași din primul bloc, pentru că între ele s-a apelat funcția `printf`; nu putem deci renunța la a doua instrucțiune `mov`.

Incrementarea și decrementarea

inc *op*

dec *op*

Efect:

op++

op--

Desigur, aceste instrucțiuni pot fi înlocuite cu ușurință de către `add`, respectiv `sub`.

Înmulțirea

`mul op`

Efect: operandul este înmulțit cu un registru implicit, iar rezultatul este depus într-o destinație de asemenea implicită; atât registrul implicit, cât și destinația depind de dimensiunea operandului. Regula generală este că operandii (cel explicit și cel implicit) trebuie să aibă aceeași dimensiune, iar destinația - dimensiune dublă. Concret:

dimensiune operand explicit	operand implicit	destinație rezultat
1	<code>al</code>	<code>ax</code>
2	<code>ax</code>	<code>(dx, ax)</code>
4	<code>eax</code>	<code>(edx, eax)</code>

Dacă dimensiunea operandilor este 4, rezultatul are 8 octeți. Cum nu există regiștri atât de mari, se folosește perechea `(edx, eax)`, în care registrul `edx` conține partea mai semnificativă; altfel spus, rezultatul are valoarea $edx \cdot 2^{32} + eax$. În cazul operandilor pe 2 octeți, deși rezultatul ar încăpea într-un registru de 4 octeți, destinație este perechea `(dx, ax)`, pentru a se păstra compatibilitatea cu procesoarele mai vechi, pe 16 biți.

Operandul explicit poate fi un registru sau o locație 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 locație de memorie, trebuie precizată explicit și dimensiunea sa, altfel nu se va ști care este operandul implicit și nici unde trebuie depus rezultatul.

Instrucțiunea `mul` realizează înmulțirea numerelor fără semn. Pentru înmulțirea numerelor cu semn se folosește instrucțiunea `imul`. Aceasta are aceeași sintaxă, acceptă aceleași tipuri de operanzi și se comportă la fel, dar operandii sunt considerați numere cu semn și rezultatul este calculat în consecință. Spre deosebire de adunare și scădere, la înmulțire și împărțire algoritmi de calcul sunt diferiți pentru numere cu semn și fără semn, deci sunt necesare instrucțiuni diferite.

Testare

Să se realizeze calculul factorialului unui număr fără semn. Pentru comparație, 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 obține un rezultat pe 8 octeți din înmulțirea a două numere pe 4 octeți (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 același tip cu operandii. Restricția se manifestă și în programul de mai sus, deci registrul `edx` poate fi ignorat.

Împărțirea

`div op`

Efectul: analog cu instrucțiunea mul. Operandul explicit este împărțitorul. Deîmpărțitul este implicit și depinde de dimensiunea împărțitorului, la fel și destinațiile unde se depun câtul și restul:

dimensiune împărțitor	deîmpărțit	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 puțin semnificativă a deîmpărțitului, iar restul în jumătatea mai semnificativă a acestuia. Acest mod de plasare a rezultatelor permite reluarea operației de împărțire în buclă, dacă este cazul, fără a mai fi nevoie de operații de transfer suplimentare.

Analog cu înmulțirea, operandul explicit (împărțitorul) poate fi un registru sau o locație 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 înmulțire, pentru numere cu semn există instrucțiunea idiv.

Operația de împărțire ridică o problemă care nu se întâlnește în alte părți: împărțirea 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 semnaliza o eroare la execuție și va fi terminat forțat.

Se va modifica programul astfel încât valoarea atribuită ebx să fie 2 și se va relua execuția; nu interesează rezultatul în sine, importantă este observația că nu se mai obține eroare.

Se modifică din nou programul, astfel încât ebx primește valoarea 1. La rulare se obține 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, soluția 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 apariția unei asemenea erori și să nu mai executăm împărțirea. În felul acesta programul nu mai este terminat forțat; modul în care procedează mai departe depinde de scopul său și de programator.

Pentru a vedea cum putem determina apariția erorii din valorile operanzilor (fără a face împărțirea), considerăm cazul programului de mai sus. Condiția de apariție 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 $eax < 2^{32}$ și valorile regiștrilor sunt numere naturale, singura posibilitate de a satisface inecuația (deci de a avea eroare) este ca $ebx \leq edx$. Mai general, condiția necesară și suficientă pentru a avea eroare este ca jumătatea mai semnificativă a deîmpărțitului să fie mai mare sau egală cu împărțitorul.

O consecință imediată este că instrucțiunile 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 împărțirile în limbaj de asamblare.

Observații:

- în limbaj de asamblare este suficientă o singură instrucțiune de împărțire, în schimb trebuie inițializați regiștrii
- există tendința de a neglija inițializarea registrului `edx`; acesta poate duce nu doar la un rezultat greșit, ci și, așa cum am văzut mai sus, la terminarea programului cu eroare (în cazul de față, dacă se întâmplă ca valoarea lui `edx` să fie cel puțin 10).

Instrucțiuni pe biți

Instrucțiuni booleene

Sunt implementate operațiile booleene binare AND, OR, XOR și cea unară NOT. Denumirile instrucțiunilor sunt chiar acestea, iar operandii pot fi (la fel ca la atribuire sau adunare) variabile de memorie, regiștri, constante. La fel ca la instrucțiunile studiate anterior, primul operand este și destinația rezultatului:

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

Există aceleași limitări privitoare la combinațiile de operanzi: nu se poate ca ambii operanzi să fie adrese de memorie și nici ca operandii să aibă dimensiuni diferite. Evident, aceste restricții se aplică operațiilor binare. Ca urmare, instrucțiunile de mai jos sunt greșite:

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

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

La ce folosesc acești operatori? La fel ca și în limbajul C, sunt destul de rar folosiți. Utilitatea lor principală constă în lucrul cu măști.

Să considerăm că vrem să testăm bitul de pe poziția 3 din registrul `ax`. Ne putem folosi de indicatorul `zero` al procesorului; din păcate, acesta ne dă informații doar despre întreg operandul și nu doar despre un bit al său. Nu ne rămâne deci decât să alterăm registrul `ax` astfel încât acesta să aibă valoarea 0 dacă și numai dacă bitul său de pe poziția 3 este 0. În acest scop, trebuie să forțăm toți biții din `ax` pe valoarea 0, cu excepția celui de pe poziția 3, care va păstra valoarea inițială (cea pe care vrem să o testăm). Funcția booleană care poate realiza aceste prelucrări (forțarea pe 0, respectiv conservare) este AND. Va trebui deci să facem AND între `ax` și un operand construit astfel încât să aibă valoarea 1 pe poziția 3 și 0 în rest:

```
and ax, 8
ax  B15 B14 B13 B12 B11 B10 B9 B8 B7 B6 B5 B4 B3 B2 B1 B0
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  0  B3 0  0  0
```

În concluzie: dacă $B_3=0$, la final $ax=0$ va avea în valoarea 0; dacă $B_3=1$, $ax \neq 0$, deci indicatorul `zero`, care dă informații despre `ax`, va indica de fapt valoarea bitului B_3 din `ax`.

Valoarea constantă cu care se realizează operația AND se numește mască; se observă că, într-adevăr, rolul său este de a "ascunde" biții de pe pozițiile care nu interesează, forțându-i la o valoare neutră. Pentru exemplul de mai sus, dacă dorim să testăm bitul de pe poziția i , masca va avea valoarea 2^i .

Desigur, se pot folosi și celelalte funcții booleene. De asemenea, măștile pot lăsa nemodificați mai mulți biți, 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 execuției; depinde doar de ceea ce ne propunem să obținem. Totuși, operația cel mai des întâlnită este AND. Ajunși aici, putem face observația că, de fapt, instrucțiunea de mai sus nu este cu adevărat o testare, deoarece modifică operandul testat. Oricât de mult ne-ar interesa valoarea bitului de pe poziția 3, nu ne dorim să pierdem valorile din ceilalți biți. Bineînțeles că se pot găsi soluții; proiectanții procesorului rezolvă însă problema punându-ne la dispoziție o instrucțiune specială, numită `test`. Astfel, în locul instrucțiunii de mai sus, putem scrie:

```
test ax, 8
```

Funcția booleană realizată este tot AND. Diferența constă în faptul că registrul `ax` nu este modificat - de fapt, rezultatul nu este depus nicăieri. În schimb, flagul `zero` este poziționat în același mod. Dacă dorim să facem doar o testare (nedistructivă) și nu o prelucrare a valorii registrului `ax`, este exact ceea ce ne trebuie.

Nu există instrucțiuni similare pentru celelalte funcții logice, ceea ce arată că, într-adevăr, funcția AND este cea mai folosită.

Instrucțiuni de deplasare

Sunt instrucțiuni care schimbă pozițiile biților în cadrul operandilor.

Deplasarea la stânga pe o poziție are efectul următor:

Carry ← ← ← ... ← ← 0

Similar pentru deplasarea la dreapta.

Instrucțiunile de deplasare sunt:

```
shl eax,1 // deplasare la stânga a registrului eax cu o poziție
shl byte ptr [...],3 // se poate și cu mai multe poziții
shl dx,c1 /* numărul de poziții pe care se face deplasarea poate fi specificat în c1 - singurul registru
care poate fi folosit pentru așa ceva (nu merge nici cx sau ecx) */
```

```
shr bh,1 // deplasare la dreapta
```

Desigur, dacă deplasarea se face pe mai multe poziții, flagul *carry* va reține doar ultimul bit ieșit.

Operatorii C echivalenți sunt << și respectiv >> (mai exact <=< și >>=).

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

```
sar eax,1
```

Efectul său este asemănător cu al instrucțiunii *shr*, dar bitul cel mai semnificativ își păstrează valoarea inițială (și trece și spre dreapta, desigur). Se poate demonstra că în acest mod se realizează o împărțire corectă la 2^n . Avem deci două instrucțiuni 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 soluție similară. Limbajul de asamblare acceptă instrucțiunea *sal*, dar este de fapt alt nume pentru *shl*. E ușor de văzut de fapt că putem interveni doar asupra bitului care "intră", care în acest caz este cel mai puțin semnificativ, deci nu are efect asupra semnului. Evident că se poate găsi și aici o soluție, dar trebuie scrise mai multe instrucțiuni.

Testare

Să se execute următoarea o secvență 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 aceeași.

Instrucțiuni de salt

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

Saltul necondiționat

jmp *adresa*

Deși există și 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 cunoaște adresa reală din memorie; ca urmare, vom lăsa gestiunea adreselor în seama compilatorului, prin utilizarea etichetelor, la fel ca în cazul instrucțiunii `goto` din C. Visual C++ este flexibil în această privință: se pot face salturi spre etichete definite în același bloc `_asm`, în alt bloc sau în partea de cod C. De altfel, este posibil și invers: o instrucțiune `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 și a doua atribuire.

Utilitatea saltului necondiționat, luat singur, nu este foarte mare, deoarece nu introduce cu adevărat o ramificație în program - nu avem variante de execuție.

Salturi condiționate

În acest caz avem două variante de execuție:

- condiția de salt este adevărată - se face saltul la adresa indicată
- condiția de salt este falsă - se continuă cu instrucțiunea următoare din memorie, ca și cum n-ar fi existat nici o instrucțiune de salt

Cele mai simple sunt instrucțiunile care testează indicatorii de condiții individuali. Vom considera în continuare doar indicatorii cei mai folosiți: *carry*, *overflow*, *zero* și *sign*.

Pentru fiecare indicator există două instrucțiuni de salt condiționat: una care face saltul când indicatorul testat are valoarea 1 și una care face saltul când indicatorul are valoarea 0.

indicator testat	salt pe valoarea 1	salt pe valoarea 0
Carry	<code>jc</code>	<code>jnc</code>
Overflow	<code>jo</code>	<code>jno</code>
Zero	<code>jz</code>	<code>jnz</code>
Sign	<code>js</code>	<code>jns</code>

Evident, pentru fiecare instrucțiune de mai sus trebuie precizată și adresa de salt.

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

cmp *op1, op2*

Combinațiile de operanzi care pot fi folosiți sunt aceleași ca la instrucțiunile studiate anterior, începând cu atribuirea. Această instrucțiune realizează intern o scădere, însă fără a altera operanzii (nu depune rezultatul nicăieri); în schimb, indicatorii de condiții sunt poziționați la fel. Putem deci folosi în același scop și instrucțiunea `sub`, dacă nu ne afectează modificarea primului operand. Cel mai important este faptul că, analizând toți indicatorii (nu doar unul), putem decide care a fost relația între operanzii instrucțiunii de

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

relație	instrucțiune salt
op1 < op2	jb
op1 <= op2	jbe
op1 > op2	ja
op1 >= op2	jae
op1 == op2	jle
op1 != op2	jne

Problemă: relațiile între operanzi diferă în funcție 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 relația între ei se inversează. Pe de altă parte, instrucțiunea de comparație rămâne aceeași. Instrucțiunile de mai sus sunt de fapt pentru operanzi fără semn. Pentru operanzi cu semn avem alt set de instrucțiuni:

relație	instrucțiune salt
op1 < op2	jl
op1 <= op2	jle
op1 > op2	jg
op1 >= op2	jge
op1 == op2	jle
op1 != op2	jne

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

Putem face observația că unele din aceste instrucțiuni testează un singur indicator, deci sunt identice cu unele din primul set. De exemplu, je și jz reprezintă de fapt aceeași instrucțiune. Nu este însă necesar să reținem aceste detalii.

Exemplu:

Presupunem că avem o variabilă a (întreg fără semn) și dorim să numărăm biții 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 cunoștințele necesare pentru a scrie integral acest cod în limbaj de asamblare, mai puțin afișarea. Pentru început putem lucra tot cu variabile (apoi, ca exercițiu, le putem înlocui pe unele cu regiștri):

```
_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 instrucțiune de salt. În limbajul de asamblare, saltul se face în cazul condiției negate față de codul C echivalent. Explicația acestei inversări este simplă: în limbajul C, o condiție adevărată are ca efect executarea unui set de instrucțiuni; în limbaj de asamblare, o condiție adevărată are ca efect efectuarea unui salt, deci evitarea execuției unor instrucțiuni.

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:
```

Observație: S-a considerat că variabila *x* este de tip *int* (întreg cu semn), motiv pentru care s-a folosit instrucțiunea *jle* și nu *jbe*.

Dacă apare și o ramură *else*, trebuie utilizate două instrucțiuni de salt. A doua instrucțiune de salt este de tip necondiționat și are ca rol de a asigura că, la terminarea execuției instrucțiunilor din prima ramură, nu vor fi executate și instrucțiunile 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 instrucțiuni de salt: una pentru a decide dacă se execută o nouă iterație sau se părăsește bucla; a doua, la finalul corpului buclei, pentru a relua cu o nouă iterație. La fel ca în cazul structurii *if*, la prima instrucțiune saltul se face pe condiția inversă celei din codul C, iar a doua instrucțiune este de salt necondiționat.

Cod C:

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

Cod în limbaj de asamblare:

```
buc1a:
cmp x, 5
jge afara
inc x      // corpul buclei
jmp buc1a
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 diferență constă în apariția unor instrucțiuni auxiliare, care delimitează mai clar inițializarea și respectiv actualizarea valorilor variabilelor pentru iterația următoare. Trebuie deci respectată cu strictețe ordinea acestor instrucțiuni componente:

1. inițializarea (executată o singură dată)
2. testul de continuare cu o nouă iterație sau ieșire
3. corpul buclei
4. actualizare variabile pentru iterația următoare
5. resluare de la pasul 2

Structura do...while

Structura de tip `do...while` este asemănătoare celei de tip `while`, dar testul de continuare/ieșire este executat după corpul buclei. Este singura structură repetitivă în care condiția de salt în limbajul de asamblare este aceeași 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 și utilizarea funcțiilor

Instrucțiuni pentru lucrul cu stiva

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

Instrucțiunea de introducere în stivă se numește, evident, `push` și 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 octeți. De fapt, pentru uniformitate, se recomandă a lucra numai cu operanzi pe 4 octeți; varianta cu 2 octeți este prezentă doar pentru compatibilitatea cu procesoarele mai vechi.

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

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

Diferența e dată de faptul că procesorul realizează singur toate aceste acțiuni (și astfel reduce riscul apariției erorilor). Observăm aici că stiva avansează "în jos", adică de la adrese mari spre adrese mici.

Instrucțiunea inversă, de extragere din stivă, se numește `pop`. După cum era de așteptat, lucrează tot numai cu operanzi de 2 sau 4 octeți:

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

Evident, la execuția acestei instrucțiuni, operandul joacă rolul destinației în care se depune valoarea luată din vârful stivei (ESP); apoi, la valoarea ESP se adună numărul de octeți ai operandului.

Rolul stivei procesorului este de a stoca informații cu caracter temporar. De exemplu, dacă avem nevoie să folosim un registru pentru unele operații, dar nu avem la dispoziție 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 inițială a fost păstrată și restabilită la final.

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

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

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

Înmulțirea variabilei cu 5 se realizează corect. În schimb, pentru că o instrucțiune `pop` a fost uitată, sunt afectate valorile ambilor regiștri implicați: `eax` primește altă valoare decât avea inițial, iar valoarea `edx` rămâne cea rezultată din înmulțire, fără a fi restaurată cea veche. Similar stau lucrurile dacă se execută prea multe instrucțiuni `pop`. În majoritatea cazurilor, operandul depus în stivă printr-o instrucțiune `push` este și destinația instrucțiunii `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 alocă spațiu unei variabile locale (neinițializată), e suficient a scădea din ESP dimensiunea variabilei respective. Similar, la distrugerea variabilei, valoarea ESP este crescută. Aici nu se folosesc în general instrucțiuni `push`, respectiv `pop`, deoarece nu interesează valorile implicate, ci doar ocuparea și eliberarea de spațiu. Se preferă adunarea și scăderea direct cu registrul ESP; evident că o eroare în aceste operații are consecințe de aceeași natură ca și cele de mai sus.

Apelul funcțiilor

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

Apelul unei funcții se realizează prin instrucțiunea `call`, care are următoarea sintaxă:

call *adresa*

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

Efectul instrucțiunii `call` este următorul: se introduce în stivă adresa instrucțiunii următoare (adresa de revenire) și se face salt la adresa indicată. Aceste acțiuni puteau fi realizate și cu instrucțiuni `push` și `jmp`, dar din nou procesorul face singur totul și astfel ne ferește de erori.

Revenirea dintr-o funcție se face prin instrucțiunea `ret`, care preia adresa de revenire din vârful stivei (similar unei instrucțiuni `pop`) și face saltul la adresa respectivă.

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

Parametri

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

Să luăm ca exemplu funcția următoare:

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

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

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


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

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

Returnarea valorilor

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

- pentru tipurile de date de dimensiune 1 octet - în registrul `al`
- pentru tipurile de date de dimensiune 2 octeți - în registrul `ax`
- pentru tipurile de date de dimensiune 4 octeți - în registrul `eax`
- pentru tipurile de date de dimensiune 8 octeți (există și așa ceva) - în regiștii `edx` și `eax`

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

Păstrarea valorilor regiștrilor

Un alt aspect important: programul pe care îl scriem este întotdeauna C/C++, deci nu putem ști cum folosește compilatorul regiștrii. Ca urmare, dacă o funcție scrisă de noi în limbaj de asamblare folosește unii regiștri, aceștia trebuie salvați în stivă la începutul execuției funcției și refăcuți la terminare. Excepție fac regiștrii `eax` și `edx`, care, după cum am văzut, sunt folosiți 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 funcției mai trebuie realizate și alte acțiuni, de care noi nu avem cunoștință. Compilatorul este cel care generează instrucțiunea `ret` - și o face în mod corect.

Exerciții

- Să se implementeze în limbaj de asamblare funcția `diff` prezentată mai sus. Variabila `c` nu va mai fi necesară, din moment ce avem regiștrii la dispoziție. Apelul către `printf` va rămâne însă în limbajul C, pentru că nu avem încă toate cunoștințele necesare pentru a-l scrie în limbaj de asamblare.

- Să se modifice forma funcției implementate anterior, astfel încât să nu afișeze diferența calculată, ci să o returneze. La revenire rezultatul trebuie preluat din `eax` într-o variabilă, pentru că nu poate fi afișat direct dintr-un registru.

- Să se implementeze în limbaj de asamblare o funcție recursivă care calculează factorialul unui număr natural, dat ca parametru.

Tablouri și structuri

Tablouri și pointeri

Deoarece discuția 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 puțin modul de accesare din limbaj de asamblare al elementelor unui tablou. Prima observație este că, evident, adresa acestui element (necesară pentru a-l putea accesa) poate fi determinată pe baza a două informații: adresa de început a tabloului, respectiv deplasamentul elementului în cadrul tabloului.

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

În ceea ce privește 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 // înlocuiește variabila suma
mov eax,0 // înlocuiește variabila i
buc1a:
    cmp eax,5
    jge afara
    add ebx,t[eax*4]
    inc eax
    jmp buc1a
afara:
```

Se observă un detaliu important: în timp ce în limbajul C indicele este exprimat în numărul de elemente față de adresa de început a tabloului, în limbajul de asamblare deplasamentul este exprimat în octeți. Din acest motiv, deși registrul `eax` ia aceleași valori ca și variabila `i`, la calculul adresei fiecărui element valoarea sa trebuie înmulțită cu 4 (deoarece elementele tabloului, fiind de tip `int`, au dimensiunea de 4 octeți).

Observație: 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 primește 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 diferență 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
buc1a:
    cmp eax,5
    jge afara
    add ebx,[ecx+eax*4]
    inc eax
    jmp buc1a
afara:
```

Tablourile ca parametri ai funcțiilor

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

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

În ambele cazuri, parametrul x este un pointer care conține adresa de început a tabloului. Tot ca urmare a acestei suprapunerii, funcția 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 și produc același efect:

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

Tablouri bidimensionale și 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 funcție care primește ca parametru o matrice. Similar cazului unidimensional, sunt două forme sintactice de scriere a antetului funcției:

```
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 . Diferența 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 , deși formal este bidimensional, elementele sunt plasate liniar în memorie. Începând de la adresa de început a tabloului, în memorie se află în ordine elementele $t[0][0]$, $t[0][1]$, $t[0][2]$, $t[1][0]$ ș.a.m.d. Prin urmare, pentru a determina adresa unui element oarecare $t[i][j]$, deplasamentul său față de începutul tabloului este $i \cdot 3 + j$, unde 3 este numărul de coloane al matricii. Menționăm faptul că, în cazul primei forme a funcției f , parametrul x reprezintă adresa de început a matricii, la fel ca la tablourile bidimensionale.

b) În al doilea caz, accesul la elementul $p[i][j]$ se face în doi pași. 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 conținând adresa de început a unui tablou cu 3 elemente de tip int , se accesează $p[i][j]$.

În aceste condiții 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 funcției 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 și, 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 octeți, iar câmpurile a , b și c sunt plasate în memorie, față de adresa de început a structurii, la deplasamentele 0, 1 și 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 și b , care sunt de tipul $char$ și deci ocupă câte un octet, sunt plasate în continuare la deplasamentele 0 și

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

Fie următoarea structură, folosită pentru reprezentarea numerelor complexe:

```
struct complex {  
    int re;  
    int im;  
};
```

Date fiind două variabile `c1` și `c2` de tip `complex`, ne propunem scrierea în limbaj de asamblare a codului care calculează suma celor două numere complexe și o depune tot în `c1`.

Observăm că, în cadrul structurii `complex`, câmpul `re` este plasat la deplasament 0 (deci începe la aceeași adresă la care începe structura), iar câmpul `im` la deplasament 4. În principal, codul constă în execuția a două adunări, pentru câmpurile `re` și `im` ale celor două variabile:

```
mov eax, dword ptr c2  
add dword ptr c1, eax  
mov eax, dword ptr c2+4  
add dword ptr c1+4, eax
```

În acest caz a trebuit folosit modificatorul `dword ptr` pentru a indica faptul că operanzii implicați au dimensiunea de 4 octeți. La utilizarea numelor `c1` și `c2`, compilatorul observă că ele corespund unor variabile de dimensiune 8 octeți, astfel încât ar semnala eroare la utilizarea lor în conjuncție cu un registru de 4 octeți.

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

```
mov eax, dword ptr c2  
add dword ptr c1, eax  
mov eax, dword ptr c2[4]  
add dword ptr c1[4], eax
```

Exercițiu

Rescrieți codul de mai sus pentru a implementa o funcție care primește ca parametri adresele a două structuri de tip `complex`:

```
void suma(complex *p1, complex *p2);
```

Funcția adună numerele complexe indicate de cei doi operanzi și 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.