asm4

Lucrul cu stiva

Procesorul foloseşte o parte din memoria RAM pentru a o accesa după principiul LIFO. Aceasta o vom denumi stiva.

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*.

Instrucţiunile care permit lucrul cu stiva sunt ***push*** şi ***pop***.

Instrucţiunea PUSH

Realizează *introducerea* unei valori în stivă.

Sintaxa: *push* operand;

Operandul poate fi registru, locaţie de memorie sau constantă numerică. Stiva lucrează doar cu valori de 2 sau 4 octeţi, pentru uniformitate preferându-se numai operanzi de 4 octeţi (varianta cu 2 se păstrază pentru compatibilitate cu procesoarele mai vechi).

Exemple:

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

Introducerea valorii în stivă se face astfel: se *scade* din ESP dimensiunea, în octeţi, a valorii care se vrea depusa în stivă, dupa care procesorul *scrie* valoarea operandului la adresa indicată de registrul ESP (vârful stivei); dimensiunea poate fi 2 sau 4 (se observă că se avansează "în jos", de la adresele mai mari la adresele mai mici); în acest mod, vârful stivei este pregătit pentru următoarea operaţie de scriere.

De exemplu, instrucţiunea

```
push eax;
```

ar fi echivalentă cu:

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

Prin folosirea lui *push* în locul secvenţei echivalente se reduce, însă, riscul erorilor.

Extrage vârful stivei într-un operand destinaţie.

Sintaxa: *pop* operand;

Operandul poate fi registru sau locaţie de memorie, de 2 sau 4 octeţi.

Exemple:

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

Extragerea valorii din stivă se face prin depunerea în destinaţie a valorii aflate în vârful stivei (la adresa [ESP]) şi adunarea, la ESP, a numărului de octeţi ai operandului (acesta indică, practic, numărul de octeţi scoşi din stivă).

Rolul stivei

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

```
push eax;   //se salveaza temporar valoarea lui eax pe stiva
...         // utilizare eax
pop eax     //restaurare
```

Variabilele locale (cu excepţia celor statice) sunt plasate de asemenea în stivă (deoarece au caracter temporar: sunt create la intrarea în funcţie şi distruse la ieşire).

În lucrul cu stiva, *instrucţiunile de introducere în stivă trebuie riguros compensate de cele de scoatere*, din punctul de vedere al numărului de instrucţiuni şi al dimensiunii operanzilor. Orice eroare poate afecta mai multe date, din cauza decalajelor.

```
push edx;
push eax;
... //utilizare registri
pop  ax  //se recupereaza doar 2 octeti din valoarea anterioara a lui eax
pop edx  //nu se recupereaza edx, ci 2 octeti din eax, 2 din edx
         //decalajul se poate propaga astfel pana la capatul stivei
```

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 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.

Instrucţiunea CALL

Apelul unei funcţii se realizează prin instrucţiunea *call*.

Sintaxa: *call* adresa

În Visual C++ vom folosi *nume simbolice* pentru a preciza adresa, cu menţiunea că de data asta nu este vorba de etichete, ca la salturi, ci chiar de numele funcţiilor apelate.

Efectul instrucţiunii *call*: 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 se preferă call pentru evitarea erorilor.

Instrucţiunea RET

Revenirea dintr-o funcţie se face prin instrucţiunea *ret*, care poate fi folosită fără operand. În acest caz, se preia adresa de revenire din vârful stivei (similar unei instrucţiuni pop) şi se face saltul la adresa respectivă. Din motive de conlucrare cu Visual Studio, nu vom folosi această instrucţiune.

Transmiterea parametrilor

Parametrii sunt tot nişte 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 le 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++ (nu în toate), 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, care ar putea altera inutil un registru, de exemplu, ci se adună la ESP numărul total de octeţi ocupat de parametri (atenţie, pe stivă se lucrează în general cu 4 octeţi, chiar dacă operanzii au dimensiuni mai mici).

Să luăm ca exemplu funcţia următoare:

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

Apelul dif(5,9) se traduce prin secvenţa care se poate vedea mai jos:

```
void main() {
  _asm {
    push dword ptr 9
    push dword ptr 5
    call show_dif
    add esp, 8
  }
}
```

Returnarea unei valori

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 - în regiştii EDX şi EAX

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

Vom modifica exemplul de mai sus astfel încât funcţia să returneze diferenţa pe care o calculează într-o secvenţă de instrucţiuni în limbaj de asamblare:

```
#include <stdio.h>

int compute_dif(int a,int b) {
  _asm {

    mov eax, a;
    sub eax, b;
                // in eax ramane rezultatul, care
                // va fi preluat la termiarea functiei
  };
}

void main() {
  int c;
  _asm{
    push dword ptr 9
    push dword ptr 5
    call compute_dif  // se salveaza adresa de revenire pe stiva
    mov c, eax;
    add esp,8         // "stergerea" parametrilor din stiva
  }
  printf("Diferenta este %d.\n", c);
}
```

Parametri

Exista o alta modalitate de accesa in cadrul unei functii parametrii acesteia in cadrul unui bloc limbaj de asamblare. Ei se gasesc pe stiva incepand cu adresa **[ebp+8]** si ocupa numarul de octeti al tipului de date respectiv.

Exemplu, o functie cu 3 parametri de tip **int** (o variabila de tip **int** are 4 octeti):

```
void functie(int a, int b, int c){
  _asm{

      mov eax, [ebp+8]  // muta in eax valoarea lui a
      mov ebx, [ebp+12] // muta in ebx valoarea lui b
      mov ecx, [ebp+16] // muta in ecx valoarea lui c

  };
}
```

Scris in aceasta maniera, exemplul de mai sus ar arata in felul urmator:

```
#include <stdio.h>

int compute_dif(int ,int ) { // nu mai este nevoie sa punem nume variabilelor, deoarece
vom lucra direct cu stiva
  _asm{

    mov eax, [ebp+8];
    sub eax, [ebp+12];
                // in eax ramane rezultatul, care
                // va fi preluat la termiarea functiei
  };
}

void main() {
  int c;
  _asm {
    push dword ptr 9
    push dword ptr 5
    call compute_dif  // se salveaza adresa de revenire pe stiva
    mov c, eax;
    add esp,8         // "stergerea" parametrilor din stiva
  }
  printf("Diferenta este %d.\n", c);
}
```

Exerciţii

1. Să se scrie un program pentru calculul factorialului unui număr fără semn. În funcţia main() se va face *în limbaj de amblare* apel la o funcţie ce calculează *iterativ* factorialul (tot în limbaj de asamblare) şi îl returnează.