

Capitolul 6 – partea I-a

Tehnici de programare în limbaj de asamblare

Acest capitol este dedicat tehnicilor de programare în limbaj de asamblare, adică modalităților de proiectare și implementare a modulelor de program. Deși codul mașină rezultat este mai scurt, programele sursă scrise în ASM tind să aibă dimensiuni mari. De aceea, este esențială o abordare sistematică și ordonată a dezvoltării programelor. Utilizarea disciplinată a procedurilor, a mecanismelor standard de transfer de parametri și a macroinstrucțiunilor contribuie esențial la obținerea de programe clare, eficiente și ușor de întreținut.

În esență, specificarea modulelor de program ASM nu diferă de cea specifică limbajelor de nivel înalt, în sensul că se pornește de la o descriere abstractă a algoritmului care trebuie implementat. O problemă specifică este asignarea variabilelor. Dacă într-un limbaj de nivel înalt acest lucru nu creează probleme (introducem câte variabile dorim, fără a ne pune problema spațiului alocat), în ASM trebuie să asignăm explicit variabile.

Prima modalitate este să asignăm cât mai multe variabile în registrele procesorului. Cum numărul acestora este limitat, vom fi nevoiți să asignăm variabile și în segmente de date (statice) sau în stivă. Pentru a nu crește numărul variabilelor peste o limită rezonabilă, este esențial ca modulele de program să implementeze subprobleme de dimensiuni adecvate (nu foarte mari), ceea ce implică o descompunere a problemei inițiale în subprobleme bine specificate.

6.1 Decizia simplă și decizia compusă. Evaluarea condițiilor logice

Majoritatea operațiilor de bază din programarea structurată (decizia, selecția, ciclurile cu test la partea inferioară și superioară) implică în mod inerent evaluarea unor condiții logice. În ASM aceste condiții sunt în general de tip comparație între valori numerice.

Decizia simplă, codificată în pseudo-cod prin:

```
if (condiție)
    Ramură_if
```

se implementează în ASM prin șablonul:

```
    Evaluatează condiție
    Salt condiționat (pe condiție falsă) la eticheta_1
    ; Ramură_if
eticheta_1:
```

Decizia compusă, codificată în pseudo-cod prin:

```
if (condiție)
    Ramură_if
else
    Ramură_else
```

se implementează după șablonul:

```
    Evaluatează condiție
```

```
    Salt condiționat (pe condiție falsă) la et_1
    ; Ramură_if
    jmp et_2
et_1:
    ; Ramură_else
et_2:
```

Evaluarea condițiilor logice simple se realizează prin instrucțiuni de comparație, aritmetice etc., care poziționează bistabilii de condiție. De exemplu, secvența pseudo-cod:

```
    if (ax < bx)
        ; Ramură_if
    else
        ; Ramură_else
```

se implementează prin:

```
    cmp ax, bx
    jge et1:
    ; Ramură_if
    jmp et2
et1:
    ; Ramură_else
et2:
```

Evaluarea condițiilor complexe se abordează în manieră ordonată. Primul caz de bază este cel în care subcondițiilor sunt conectate prin operatorul logic AND. Astfel, să considerăm o condiție logică de forma:

```
C = C1 AND C2 AND C3 AND ... AND Cn
```

și decizia compusă:

```
    if (C)
        ; Ramură_if
    else
        ; Ramură_else
```

Implementarea este următoarea:

```
    Evaluatează C1
    Salt condiționat (pe condiție falsă) la et_1
    Evaluatează C2
    Salt condiționat (pe condiție falsă) la et_1
    .....
    Evaluatează Cn
    Salt condiționat (pe condiție falsă) la et_1
    ;
    ; Ramură_if
    ;
    jmp et_2
et_1:
    ;
    ; Ramură_else
    ;
et_2:
```

Să considerăm de exemplu, secvența pseudo-cod:

```
    if (car >= '0' AND car <= '9') {
        sir [i] = car - '0';
        i = i + 1;
    }
```

În care presupunem că variabila car se află în AL iar indicele i în registrul BX. Implementarea este următoarea:

```

    cmp    al, '0'          ; Evaluare al >= '0'
    jb     et_1             ; Salt pe cond. falsă (al < '0')
    cmp    al, '9'          ; Evaluare al <= '9'
    ja     et_1             ; Salt pe cond. falsă (al > '9')
    sub    al, '0'          ; Calcul car - '0'
    mov    sir [bx], al      ; Depunere în sir [i]
    inc    bx               ; Incrementare i
et_1:

```

Al doilea caz de bază este cel în care subcondițiile sunt conectate prin operatorul OR. Să considerăm condiția compusă:

C = C1 OR C2 OR C3 OR ... OR Cn

și aceeași formă de decizie compusă:

```

if (C)
    ; Ramură_if
else
    ; Ramură_else

```

Implementarea este următoarea:

```

    Evaluatează C1
    Salt condiționat (pe condiție adevărată) la et_1
    Evaluatează C2
    Salt condiționat (pe condiție adevărată) la et_1
    .....
    Evaluatează Cn
    Salt condiționat (pe condiție adevărată) la et_1
    ;
    ; Ramură_else
    ;
    jmp et_2
et_1:
    ;
    ; Ramură_if
    ;
et_2:

```

Cele două șabloane de implementare pentru condiții compuse de tip AND și OR se bazează pe proprietățile elementare ale operațiilor logice respective. Astfel la operația AND, e suficient ca un singur operand (o subcondiție) să fie fals, pentru ca întreaga condiție să fie falsă. Similar, la operația OR, e suficient ca un singur operand să fie adevărat, pentru ca întreaga condiție să fie adevărată.

A treia schemă de dezvoltare se referă la implementarea condițiilor negate. O secvență pseudo-cod de forma:

```

if (NOT condiție)
    ; Ramură_if
else
    ; Ramură_else

```

se implementează la fel cu schema if-else obișnuită, dar cu inversarea saltului condiționat:

```

    Evaluatează condiție
    Salt condiționat (pe condiție adevărată) la et_1
    ;

```

```

        ; Ramură_if
        ;
        jmp et_2
et_1:
        ;
        ; Ramură_else
        ;
et_2:

```

Cu aceste trei operații de bază (AND, OR, NOT), putem acum evalua orice tip de condiție logică. Să considerăm secvența pseudo-cod:

```

        if ((C1 AND C2) OR C3)
            ; Ramură_if
        else
            ; Ramură_else

```

Considerăm subcondițiile C1 AND C2 și C3 și aplicăm pentru început șablonul de la operația OR:

```

        Evaluatează (C1 AND C2)
        Salt condiționat (pe condiție adevărată) la et_1
et_3:
        Evaluatează (C3)
        Salt condiționat (pe condiție adevărată) la et_1
        ;
        ; Ramură_else
        ;
        jmp et_2
et_1:
        ;
        ; Ramură_if
        ;
et_2:

```

Detaliem acum evaluarea subcondiției (C1 AND C2), observând că se sare la eticheta et_1 dacă ambele subcondiții C1 și C2 sunt adevărate; altfel se sare la eticheta et_3:

```

        Evaluatează C1
        Salt condiționat (pe condiție falsă) la et_3
        Evaluatează C2
        Salt condiționat (pe condiție adevărată) la et_1
et_3:
        Evaluatează (C3)
        Salt condiționat (pe condiție adevărată) la et_1
        ;
        ; Ramură_else
        ;
        jmp et_2
et_1:
        ;
        ; Ramură_if
        ;
et_2:

```

După aceste modele, se pot evalua pas cu pas condiții oricât de complicate. Șabloanele de evaluare se utilizează și la celelalte operații din programarea structurată.

6.2 Cicluri cu test la partea superioară și inferioară

Ciclul cu test la partea superioară, descris în pseudo-cod prin:

```
while (condiție)
    ; Bloc
```

se implementează după șablonul:

```
et_1:
    Evaluatează condiție
    Salt condiționat (pe condiție falsă) la et_2
    ;
    ;
    ; Bloc
    jmp et_1
```

Să considerăm secvența următoare în limbajul C:

```
while (*s >= '0' && *s <= '9') {
    n = 10 * n + *s - '0';
    s++;
}
```

unde s este un pointer la char iar n un întreg. Această secvență este tipică pentru conversia ASCII-întreg. Presupunem variabila n asignată în AX, iar pointerul s asignat (ca adresă near) în registrul SI și aplicăm șablonul de implementare:

```
et_1:
    Evaluatează ( [SI] >= '0')
    Salt pe condiție falsă la et_2
    Evaluatează ( [SI] <= '9')
    Salt pe condiție falsă la et_2
    AX = AX * 10 + [SI] - '0'
    SI = SI + 1
    jmp et_1
```

et_2:

În codificarea propriu-zisă se va încărca [si] în registrul al pentru o operare mai eficientă:

```
    mov bx, 10
et_1:
    mov cl, [si]
    xor ch, ch                ; CX <-- caracterul de la adresa SI
    cmp dl, '0'              ; Prima subcondiție
    jb et_2                  ; Salt pe condiție falsă
    cmp dl, '9'              ; A doua subcondiție
    ja et_2                  ; Salt pe condiție falsă
    mul bx                   ; n = n * 10
    sub cl, '0'              ; *s - '0'
    add ax, cx               ; Valoare finală n
    inc si                   ; s++
    jmp et_1
```

Ciclurile cu test la partea inferioară, descris în pseudo-cod prin una din formele:

do	repeat
; Bloc	; Bloc
while (condiție)	until (condiție)

se implementează după șabloanele:

et:	et:
; Bloc	; Bloc
Evaluează condiție	Evaluează condiție
Salt pe condiție adevărată la et	Salt pe condiție falsă la et

Să considerăm un algoritm tipic pentru conversia întreg-ASCII, descris în limbajul C:

```
do {
    *s++ = n % 10 + '0';
    n = n/10;
} while (n != 0);
*s = 0;
```

Prin împărțiri succesive la 10 se generează cifrele corespunzătoare întregului n și se depun în șirul de caractere s (cifrele rezultă în ordine inversă). Implementarea este următoarea (considerăm n memorat în registrul AX, iar pointerul n în registrul index DI):

```
et:    mov    bx, 10
xor     dx, dx                ; Deîmpărțit = DX:AX
div     bx                    ; AX = n / 10, DX (DL) = n % 10
add     dl, '0'               ; n % 10 + '0'
mov     [di], dl              ; Depunere în *s
inc     di                    ; și apoi incrementare s
test    ax, ax                ; Compară n cu 0
jnz     et                    ; Reluare ciclu
mov     byte ptr [di], 0      ; *s = 0
```

Un caz particular de ciclu cu test la partea superioară este ciclul cu contor ascendent, descris în pseudo-cod prin:

```
for (contor = vi to vf step pas)
    ; Bloc
```

în care se consideră că pas este o valoare strict pozitivă. Dacă specificația pasului lipsește, se consideră implicit pasul 1. Această descriere se poate detalia într-un ciclu de tip while și o inițializare:

```
contor = vi;
while (contor <= vf) {
    ; Bloc
    contor += pas;
}
```

ceea ce arată că se poate aplica șablonul de implementare de la ciclul while.

Ciclul cu contor descendent, descris în pseudo-cod prin:

```
for (contor = vi downto vf step pas)
    ; Bloc
```

în care, de asemenea, se consideră că pas este pozitiv și implici 1, se poate detalia în:

```
contor = vi;
while (contor >= vf) {
    ; Bloc
    contor = contor - pas;
}
```

Să considerăm, de exemplu, o secvență tipică de translatare la dreapta cu o poziție a elementelor unui tablou de întregi:

```
for (i = 99 downto 1)
    TAB [i] = TAB [i-1];
```

Asignăm variabila *i* la registrul SI. Trebuie observat că indicii variază după elementele tabloului, dar adresele variază cu câte 2 octeți la fiecare element. Pentru o implementare ordonată, vom varia variabila indice SI exact ca în specificarea algoritmului și o vom ajusta temporar prin înmulțire cu 2 înaintea accesării elementelor tabloului. Detalierea ciclului este:

```
    i = 99;
et_1:    Evaluatează ( i >= 1)
        Salt pe condiție falsă la et_2
        TAB [i] = TAB [i-1];
        i = i - 1;
et_2:
```

După această detaliere, implementarea devine de rutină:

```
et_1:    mov    di, 99                ; i = 99;
        cmp    di, 1                ; Evaluatează (i >= 1)
        jl     et_2                ; Salt pe condiție falsă (i < 1)
        shl    di, 1                ; Temporar, DI <-- 2*DI
        mov    ax, TAB[si]          ; TAB [i]
        mov    TAB[si-2], ax        ; TAB [i-1]
        shr    di, 1                ; Refacere DI
        dec    di                  ; i = i - 1
et_2:
```

Secvența de mai sus ar putea fi implementată și prin instrucțiuni specifice șirurilor de cuvinte (se presupune că registrele DS și ES indică segmentul în care este memorat tabloul TAB):

```
    lea    si, TAB [98*2]          ; Sursă inițială
    lea    di, TAB [99*2]          ; Destinație inițială
    std                     ; Sens descendent
    mov    cx, 99                 ; Număr iterații
    rep    movsw                  ; Transfer
```

O altă formă posibilă, care permite generalizări interesante este copierea la nivel de octet:

```
    lea    si, TAB [98*2 + 1]      ; Începem de la
    lea    di, TAB [99*2 + 1]      ; ultimul octet
    std                     ; Sens descendent
    mov    cx, 99*2                ; Număr de octeți
    rep    movsb                  ; Copiere
```

În general, prelucrarea tablourilor de tip oarecare presupune înmulțirea indicilor de acces cu numărul de octeți ai tipului de bază al tabloului. În acest sens, inițializările registrelor SI și DI din ultima secvență se pot generaliza astfel:

```
    lea si, TAB [98 * (TYPE TAB) + (TYPE TAB - 1)]
    lea di, TAB [99 * (TYPE TAB) + (TYPE TAB - 1)]
```

Dacă secvența de copiere propriu-zisă se înlocuiește cu:

```

mov    cx, 99 * (TYPE TAB)
rep     movsb

```

atunci se obține o secvență care implementează algoritmul dat indiferent de tipul tabloului. Să considerăm un tablou de structuri de tipul

```

MY_STRUC struc
    n        dw ?
    text     db 10 dup (0)
ENDS

```

și definim un tablou de 20 de structuri:

```

.data
    tab MY_REC 20 dup (< , >)

```

Secvența de traducere se poate scrie:

```

lea     si, tab [(LENGTH tab - 1) * (TYPE MY_REC) - 1]
lea     di, tab [(LENGTH tab) * (TYPE MY_REC) - 1]
std
mov     cx, (LENGTH tab - 1) * (TYPE MY_REC)
rep     movsb

```

prin care se poziționează DI (destinația) pe ultimul octet al ultimei înregistrări din tablou, iar SI (sursa) pe ultimul octet al penultimului element din tablou. Numărul de iterații la nivel de elemente este numărul de elemente al tabloului micșorat cu o unitate; la nivel de octeți, se înmulțește această valoare cu dimensiunea unui element.

O altă formă posibilă de scriere exploatează legătura dintre operatorii LENGTH, TYPE și SIZE. De exemplu, inițializarea contorului CX s-ar mai putea scrie:

```

mov     cx, SIZE tab - TYPE tab

```

În cazurile în care ciclul cu contor se poate descrie prin:

```

for (contor = n downto 1)
    ; Bloc

```

sau atunci când se repetă de n ori o anumită operație, iar valoarea curentă a contorului nu contează, ciclul se poate implementa prin instrucțiunea LOOP:

```

mov     cx, n
et:
    ; Bloc
loop    et

```

Să considerăm o secvență de determinare a maximului și minimului unui tablou de întregi cu semn, definit prin:

```

.data
    TABLOU    dw 100 dup(?)
    n         dw ($-TABLOU)/2
    val_max   dw ?
    val_min   dw ?
    i_max     dw ?
    i_min     dw ?

```

Se dorește determinarea atât a valorilor maxime și minime cât și a indicilor pe care apar aceste elemente.

Secvența se poate descrie în pseudo-cod prin:


```

val_max = TABLOU [0];
val_min = TABLOU [0];
i_max = 0;
i_min = 0;
for (i = 1 to n-1) {
    if (TABLOU [i] > val_max) {
        val_max = TABLOU [i];
        i_max = i;
    }
    else if (TABLOU [i] < val_min) {
        val_min = TABLOU [i];
        i_min = i;
    }
}

```

Pentru implementare, vom presupune că adresa elementului TABLOU[i] este alocată în registrul BX. Se va incrementa direct această adresă, iar ciclul va fi implementat printr-o instrucțiune loop. Indicele elementului curent se obține printr-o diferență între adresa curentă (BX) și adresa de început a tabloului (SI) și o împărțire la 2.

```

.code
    lea    bx, TABLOU          ; Adresa elementului TABLOU [0]
    mov    si, bx              ; Copiată și în SI
aici_1:
    mov    cx, n
    dec    cx                  ; Sunt n-1 iterații
aici_2:
    mov    ax, [bx]
    mov    val_max, ax         ; val_max = TABLOU [0]
    mov    val_min, ax         ; val_min = TABLOU [0]
    mov    i_max, 0            ; i_max = 0
    mov    i_min, 0            ; i_min = 0
    add    bx, 2               ; Adresa elementului TABLOU [1]
et_1:
    mov    ax, [bx]            ; AX <-- TABLOU [i]
    cmp    ax, val_max         ; Evaluează (TABLOU [i] > val_max)
    jle    et_2                ; Salt pe condiție negată
    mov    val_max, ax         ; val_max = TABLOU [i]
    push    bx                 ; Salvare adresă curentă
    sub     bx, si              ; Adresa curentă - adresa de început
    shr     bx, 1               ; supra 2
    mov    i_max, bx           ; egal indicele i_max
    pop     bx                 ; Refacere adresă curentă
et_2:
    cmp    ax, val_min         ; Evaluează (TABLOU [i] < val_min)
    jge    et_3                ; Salt pe condiție negată
    mov    val_min, ax         ; val_min = TABLOU [i]
    push    bx                 ; Similar
    sub     bx, si
    shr     bx, 1
    mov    i_min, bx
    pop     bx
et_3:
    add     bx, 2               ; Adresa elementului următor
    loop   et_1                ; Ciclu după CX

```

Se observă că implementarea ciclului prin instrucțiunea LOOP complică determinarea indicelui elementului curent. Dacă sa-r fi implementat un ciclu ascendent obișnuit, înlocuind secvența dintre etichetele aici_1 și aici_2 prin:

```
aici_1:
    mov    cx, 1
    cmp    cx, n
    jl     et_4
```

aici_2:

iar secvența de după eticheta et_3 prin:

```
et_3:
    add    bx, 2
    inc    cx
    jmp    et_1
```

et_4:

atunci, la fiecare iterație, indicele elementului curent ar fi fost disponibil în registrul CX, iar secvențele de determinare a indicilor i_max și i_min s-ar fi redus la simple transferuri de forma:

```
mov i_min, cx
```

Exemplul de mai sus arată că implementarea ciclurilor cu contor prin instrucțiunea LOOP, aparent mai simplă, poate conduce la complicații în interiorul ciclului.

Există și cicluri cu mai multe puncte de ieșire (o ieșire normală și una sau mai multe ieșiri forțate), ca în secvența pseudo-cod următoare, în care prin break s-a marcat ieșirea forțată din ciclu:

```
while (condiție_1) {
    ; Bloc_1
    if (condiție_2)
        break;
    ; Bloc_2
}
```

O asemenea situație se implementează combinând sabloanele de la while și if:

```
et_1:
    Evaluatează (condiție_1)
    Salt pe condiție falsă la et_2
    ; Bloc_1
    Evaluatează (condiție_2)
    Salt pe condiție adevărată la et_2
    ; Bloc_2
    Salt la et_1
et_2:
```

6.3 Selecția. Tabele de salt sau de apel de proceduri

Operația de selecție se descrie în pseudo-cod prin:

```
selectează (c) dintre {
    c1:          Bloc_1;
    c2:          Bloc_2;
    .....
    cn:          Bloc_n;
    [default:    Bloc_d;]
}
```

În care **c1**, **c2**, ..., **c_n** sunt așa-numitele cazuri (case). Acestea sunt de fapt valori de același tip cu variabila **c**. Cazul default corespunde situației în care variabila **c** nu are nici una din valorile **c1**, **c2**, ..., **c_n** și este opțional. Blocurile de instrucțiuni **Bloc_1**, **Bloc_2**, ..., **Bloc_n** pot fi și vide.

Implementarea naturală a selecției pornește de la observația că o asemenea operație este echivalentă cu o succesiune de decizii, după cum urmează:

```
if (c = c1)
    ; Bloc_1
else if (c = c2)
    ; Bloc_2
.....
else if (c = cn)
    ; Bloc_n
else
    ; Bloc_d
```

Se pot aplica acum șabloanele de implementare de la operația de decizie, ceea ce înseamnă comparații succesive și salturi condiționate. Această soluție de implementare devine incomodă atunci când numărul cazurilor este mare.

O altă soluție de implementare, mult mai eficientă, utilizează tabele de salt sau de apel de proceduri. Să presupunem că blocurile de instrucțiuni sunt organizate în felul următor:

```
et_1:
    ; Bloc_1
    jmp et
et_2:
    ; Bloc_2
    jmp et
.....
et_n:
    ; Bloc_n
    jmp et
et_d:
    ; Bloc_d
et:
```

adică ieșirea din operația de selecție se face pe la eticheta **et**.

Ideea soluției de implementare este definirea unei tabele de salt, inițializată cu punctele de intrare în blocurile de instrucțiuni (cu adresele etichetelor **et_i**) și calculul automat al adresei corespunzătoare de salt, pe baza căutării valorii curente **c** într-un tabel de cazuri posibile.

Pentru a fixa ideile, presupunem variabila **c** și cazurile **c1, c2, ..., cn** ca fiind reprezentabile pe câte un octet; de asemenea, presupunem toate blocurile de instrucțiuni definite în același segment de cod; registrele DS și ES indică segmentul curent de date. Variabilele **n** și **c** au semnificațiile din descrierea operației (variabila de selecție, respectiv numărul de cazuri posibile). Definiția celor două tabele este:

```
.data
case      db c1, c2, c3, ..., cn
tabjmp    dw et_1, et_2, ..., et_n
c         db ?
n         dw ?
```

Să considerăm un exemplu concret. Se citește un caracter de la tastatură și, funcție de valoarea sa, se afișează un mesaj la consolă. Considerăm numărul cazurilor posibile ca fiind 4, iar constantele de selecție ca fiind caracterele 'a', 'b', 'c' și 'd'. Citirea caracterelor se execută într-o buclă din care se iese la apăsarea tastei Enter. Implementarea este următoarea:

```
.model large
    include io.h
.stack 1024
.data
    case                db 'a', 'b', 'c', 'd'
    tabjmp              dw et_1, et_2, et_3, et_4
    n                   dw 4
.code
start:
    init_ds_es          ; Inițializare DS și ES
iar:
    getc                ; Citire caracter în AL
    cmp    al, cr        ; Este Enter ?
    je     gata          ; Dacă da, oprire
    lea    di, case      ; Adresă tabelă de cazuri
    mov    cx, n         ; Număr de cazuri explicite
    cld                     ; Direcție ascendentă
    repne scasb          ; Căutare caz (AL) în tabelă
    jne    et_d          ; Dacă ZF = 0, înseamnă că nu s-a identificat
                        ; nici un caz explicit, deci este cazul default
    dec    di            ; S-a găsit un caz explicit
                        ; DI este poziționat pe octetul
                        ; următor, așa că îl decrementăm
    lea    bx, case      ; Adresă tabelă de cazuri
    sub    di, bx        ; Diferența = deplasament, în gama: 0...n-1
    shl    di, 1         ; Înmulțire cu 2 (adrese pe word)
    jmp    tabjmp [di]    ; Salt indirect la cazul respectiv
gata:
    exit_dos
;
; Blocurile de instrucțiuni care tratează
; cazurile explicite și implicite
;
et_1:
    putsi    <cr, lf, 'Cazul 1 (a)', cr, lf>
    jmp     et
et_2:
    putsi    <cr, lf, 'Cazul 2 (b)', cr, lf>
    jmp     et
et_3:
    putsi    <cr, lf, 'Cazul 3 (c)', cr, lf>
    jmp     et
et_4:
    putsi    <cr, lf, 'Cazul 4 (d)', cr, lf>
    jmp     et
et_d:
    putsi    <cr, lf, 'Cazul default', cr, lf>
et:
;
; Punct de ieșire din operația de selecție
;
    jmp iar
end start
```

Căutarea în tabela **case** se face prin instrucțiuni cu șiruri de caractere, după modelul standard descris la aceste instrucțiuni. Din adresa elementului identificat în tabelă se calculează indicele acestuia (de la 0 la n-1) și, prin înmulțire cu 2, poziția corespunzătoare din tabela de adrese de salt. Se execută apoi un salt indirect intrasegment.

Alte variante posibile de implementare sunt:

- cazurile memorate pe mai mult de un octet - se definește tabela case în mod corespunzător, și se execută o secvență de căutare explicită, după modelul:

```
for (i = 0 to n-1)
    if (case [i] == c)
        break;
if (i < n)
    jmp tabjmp [i]
else
    jmp et_d
```

- blocurile asociate cazurilor sunt în segmente de cod diferite - se definește tabela de salt cu adrese de tip far, iar etichetele et_i și et_d se definesc cu directiva LABEL și atributul FAR;
- blocurile asociate cazurilor sunt implementate ca proceduri - se definește tabela de adrese inițializată cu numele procedurilor respective și se înlocuiește instrucțiunea de salt indirect cu una de apel indirect de procedură; punctul de ieșire din operația de selecție va fi cel imediat următor apelului indirect de procedură.

6.4 Transferul parametrilor către proceduri

Proiectarea ordonată și sistematică a procedurilor este un punct cheie în dezvoltarea unui sistem de programe în limbaj de asamblare. Problemele de bază care trebuie urmărite sunt:

- transferul parametrilor;
- întoarcerea rezultatelor;
- zone de date proprii procedurilor;
- controlul stivei;
- recursivitatea;
- proceduri cu număr variabil de parametri.

Pentru fiecare din aceste probleme, există tehnici sistematice de abordare, care vor fi prezentate în continuare.

Prima problemă se referă la transferul parametrilor către proceduri. Dacă în limbajele de nivel înalt, acest lucru este impus de sintaxa limbajului (se definește o listă de parametri), în limbaj de asamblare există multiple posibilități de transfer. Aceasta este, de altfel, și cauza pentru care o proiectare nesistematică a procedurilor, poate conduce la programe greu de citit și înțeles, predispuse la erori și foarte greu de întreținut.

Pe de altă parte, o proiectare îngrijită a procedurilor, combinată eventual cu macroinstrucțiuni adecvate, care respectă tehnicile standard de transmiterea a parametrilor, contribuie esențial la dezvoltarea unor programe ușor de înțeles și de întreținut, asemănătoare - din acest punct de vedere - programelor în limbaje de nivel înalt.

6.4.1 Tipuri de transfer (prin valoare sau prin referință)

O primă chestiune care trebuie decisă în proiectarea unei proceduri este tipul de transfer al parametrilor. Se pot utiliza două asemenea tipuri:

- transfer prin valoare, care implică transmiterea conținutului unei variabile;
- transfer prin referință, care implică transmiterea adresei de memorie a unei variabile.

Alegerea între aceste două tipuri de transfer se poate face după următoarele criterii:

- dacă variabila care trebuie transmisă nu este alocată în memorie, ci într-un registru, se va alege transmiterea prin valoare;
- structurile de date de volum mare (tablouri, structuri, tablouri de structuri etc.) vor fi transmise totdeauna prin referință;
- dacă procedura trebuie să modifice o variabilă parametru formal, care este alocată în memorie, se va alege transferul prin referință; (modificarea unui parametru formal din interiorul procedurii trebuie totuși utilizată cât mai puțin, deoarece este o cauză majoră de erori; este preferabil un transfer prin valoare și întoarcerea valorii modificate);

Pentru a fixa ideile, să considerăm o procedură **pro_add**, de tip near, care adună două numere pe 32 de biți, întorcând rezultatul în perechea de registre DX:AX. Datele sunt inițial memorate în variabilele **n1** și **n2** iar rezultatul trebuie depus în variabila **rez**:

```
.data
n1    dd 10000H
n2    dd 20000H
rez   dd ?
```

Să presupunem că transmitem parametrii prin registre, ceea ce înseamnă că avem nevoie de 4 registre generale, de exemplu DX:AX pentru primul parametru și CX:BX pentru al doilea. Secvența de apel a procedurii este următoarea:

```
.code
mov    ax, word ptr n1
mov    dx, word ptr n1 + 2      ; DX:AX = primul parametru
mov    bx, word ptr n2
mov    cx, word ptr n1 + 2      ; CX:BX = al doilea parametru
call   near ptr pro_add
mov    word ptr rez, ax         ; Rezultat în DX:AX
mov    word ptr rez + 2, dx
```

Procedura **pro_add** se detaliază astfel:

```
pro_add proc near
    add    ax, bx
    adc    dx, cx
    ret
pro_add endp
```

Să considerăm acum varianta transmiterii prin referință a parametrilor, în care se transmit către procedură adresele de tip near ale variabilelor **n1** și **n2**, prin registrele SI și DI. Secvența de apel este:

```

lea    si, n1
lea    di, n2
call   near ptr pro_add
mov     word ptr rez, ax           ; Rezultat în DX:AX
mov     word ptr rez + 2, dx

```

Procedura se dezvoltă în felul următor:

```

pro_add proc near
    mov     ax, [si]               ; Partea low din primul număr
    add     ax, [di]               ; + partea low din al doilea
    mov     dx, [si+2]             ; Partea high din primul număr
    adc     dx, [di+2]             ; + partea high din al doilea
    ret
pro_add endp

```

În esență, se vede că transmiterea prin referință implică operații de adresare indirectă în interiorul procedurii.

Un aspect important este salvarea și restaurarea registrelor implicate în transferul parametrilor, ca și a celor utilizate în interiorul procedurii.

6.4.2 Transfer prin registre

Vom trece acum la analiza modalităților efective de transfer a parametrilor, fie ei valori sau adrese. O primă modalitate este transferul prin registrele mașinii. Avantajul acestei soluții este faptul că, în procedură, parametrii actuali sunt disponibili imediat.

Pentru conservarea registrelor, acestea se salvează în stivă înainte de apel și se refac după revenirea din procedură. Secvența de apel este deci organizată după șablonul:

```

; Salvare în stivă a registrelor implicate în transfer
; Încărcare registre cu parametrii actuali
; Apel de procedură
; Refacere registre din stivă

```

Dezavantajele acestei modalități sunt:

- numărul limitat de registre al mașinii - e posibil să existe registre ocupate sau pur și simplu să fie mai mulți parametri decât registrele disponibile;
- neuniformitatea metodei - nu există o modalitate ordonată de transfer, fiecare procedură având propriile reguli de transfer.

6.4.3 Transfer prin zonă de date

În această variantă, se pregătește anterior o zonă de date și se transmite către procedură adresa acestei zone de date. În această formă, organizarea zonei de date și secvența de apel a procedurii `pro_add` din paragraful anterior este:

```

.data
    zona    label dword
    n1      dd 10000H
    n2      dd 20000H
    rez     dd ?
.code
    lea     bx, zona
    call    pro_add

```

Procedura `pro_add` se scrie acum în forma:

```

pro_add proc near
    mov     ax, [bx]           ; Partea low din primul număr
    add     ax, [bx+4]         ; + partea low din al doilea
    mov     dx, [bx+2]         ; Partea high din primul număr
    adc     dx, [bx+6]         ; + partea high din al doilea
    ret
pro_add endp

```

Pentru un acces comod la zona de parametri, se recomandă definirea unei structuri care să descrie organizarea zonei de date:

```

TIP_ZONA struc
    nr1     dd ?
    nr2     dd ?
    rez     dd ?
TIP_ZONA ends
.data
    zona    TIP_ZONA <10000, 20000, ?>
.code
    lea     bx, zona
    call    pro_add

```

Procedura `pro_add` se scrie astfel:

```

pro_add proc near
    mov     ax, [bx].nr1       ; Partea low din primul număr
    add     ax, [bx].nr2       ; + partea low din al doilea
    mov     dx, [bx].nr1+2     ; Partea high din primul număr
    adc     dx, [bx+6].nr2+2   ; + partea high din al doilea
    ret
pro_add endp

```

6.4.4 Transfer prin stivă. Descărcarea stivei

Transferul parametrilor prin stivă este cea mai importantă modalitate de transfer. Avantajele acestei metode sunt uniformitatea (se asigură un mecanism unic de transfer pentru toate procedurile) și compatibilitatea cu limbajele de nivel înalt (majoritatea compilatoarelor utilizează această metodă). Transferul prin stivă este obligatoriu în situația în care aplicația conține atât module ASM cât și module în limbaj de nivel înalt.

În principiu, transferul prin stivă constă în plasarea în stivă (prin instrucțiuni de tip `PUSH`) a parametrilor, înainte de apelul procedurii. Astfel, procedura va găsi parametrii în stivă, imediat după adresa de revenire.

Problemele de bază care trebuie avute în vedere la implementarea acestui tip de transfer sunt:

- tipul procedurii (FAR sau NEAR);
- tipul parametrilor, în special al celor de tip adresă (FAR sau NEAR);
- ordinea de plasare a parametrilor în stivă;
- accesul la parametri din interiorul procedurii;
- descărcarea stivei (de către programul apelant sau de către procedură).

Tipul procedurii, tipul parametrilor și ordinea lor sunt importante pentru calculul deplasamentelor în stivă și pentru accesul corect la parametri.

Tehnica de acces standard la parametrii procedurii se bazează pe adresarea bazată (eventual și indexată) prin registrul BP, care presupune registrul SS ca registru implicit de segment. Accesul se realizează prin operațiile următoare, efectuate chiar la intrarea în procedură:

- se salvează BP în stivă;
- se copiază SP în BP;
- se salvează în stivă (eventual) registrele utilizate de procedură;
- se accesează parametrii prin adresare indirectă cu BP.

La încheierea procedurii, se execută operațiile următoare:

- se refac registrele salvate;
- se reface BP;
- se revine în programul apelant prin RET.

Să considerăm aceeași procedură `pro_add` (de data aceasta de tip FAR), implementată prin această tehnică. Secvența de apel va fi:

```
.data
    n1    dd 10000H
    n2    dd 20000H
    rez    dd ?
.code
    push  word ptr n1+2      ; Partea high la adrese mari
    push  word ptr n1        ; Partea low la adrese mici
    push  word ptr n2+2      ; Similar pentru n2
    push  word ptr n2
    call  far ptr pro_add    ; Apel
    add   sp, 8              ; Descărcare stivă
    mov   word ptr rez, ax   ; Depunere
    mov   word ptr rez+2, dx ; rezultat
```

Plasarea datelor în stivă trebuie să țină seama de modul de reprezentare în memorie. Astfel, numerele pe 32 de biți se plasează în stivă în așa fel încât la adrese mici să se găsească partea mai puțin semnificativă. De asemenea, se observă descărcarea stivei (refacerea registrului SP la valoarea dinaintea secvenței de apel), prin adunarea explicită la SP a numărului de octeți care a fost plasat în stivă.

Pentru a accesa corect parametrii în stivă, este bine să figurăm imaginea stivei după salvarea registrului BP, ținând cont de tipul procedurii, de numărul, tipul și ordinea parametrilor în stivă. Această imagine este ilustrată în Figura 6.1 (reamintim că adresele cresc de sus în jos). Procedura `pro_add` se scrie în felul următor:

```
pro_add proc far
    push  bp                ; Secvența tipică
    mov   bp, sp            ; de acces
    mov   ax, [bp+10]       ; n1 low
    add   ax, [bp+6]         ; n2 low
    mov   dx, [bp+12]       ; n1 high
    adc   dx, [bp+8]        ; n2 high
    pop   bp                ; Refacere bp
    ret                     ; Revenire
pro_add endp
```

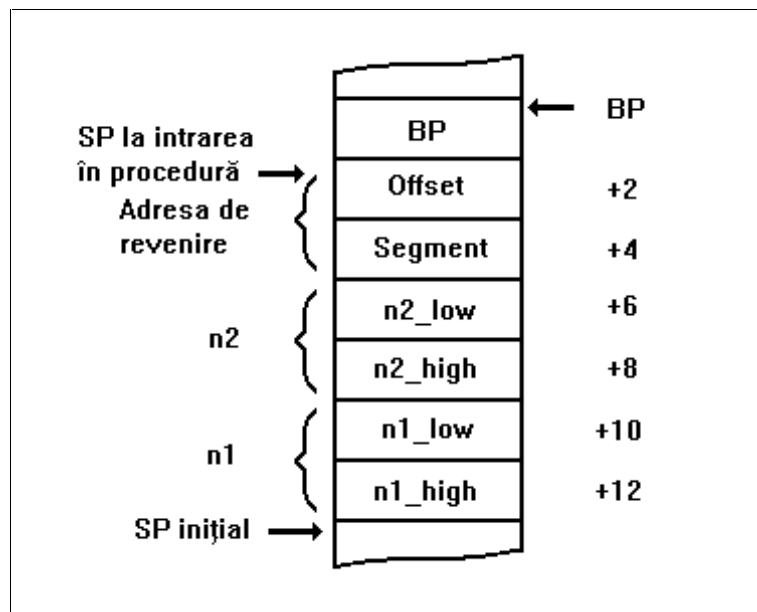


Figura 6.1 Imaginea stivei la intrarea în procedura `pro_add`

Calculul explicit al deplasamentelor parametrilor (de tipul `[bp+8]`, `[bp+10]` etc.) reprezintă o sursă potențială de greșeli. În plus, întreținerea procedurii este foarte greoaie. Dacă se schimbă tipul procedurii din FAR în NEAR sau dacă se schimbă ordinea celor doi parametri, toate liniile de program care conțin deplasamente de acest gen trebuie rescrise.

Aceste probleme se rezolvă elegant prin definirea unei structuri șablon care să conțină imaginea stivei, de la registrul BP în jos. Dacă am figurat grafic imaginea stivei, definirea structurii șablon este imediată (vezi Figura 6.1):

```
sablon_1 struc
    _bp      dw ?           ; BP
    _cs_ip   dw 2 dup (?)   ; Adresă de revenire
    n2_low   dw ?           ;
    n2_high  dw ?           ; Parametri
    n1_low   dw ?           ;
    n1_high  dw ?           ;
sablon_1 ends
```

Procedura se rescrie acum în forma:

```
pro_add proc far
    push bp
    mov bp, sp
    mov ax, [bp].n1_low
    add ax, [bp].n2_low
    mov dx, [bp].n1_high
    adc dx, [bp].n2_high
    pop bp
    ret
pro_add endp
```

ceea ce este mult mai clar decât versiunea anterioară. În plus, dacă se modifică tipul procedurii sau ordinea parametrilor, nu trebuie modificată decât definiția structurii șablon; asamblorul va calcula corect noile deplasamente.

În forma de mai sus, procedura `pro_add` corespunde unei funcții C cu prototipul:

```
long _pro_add (long x1, long x2);
```

Să presupunem acum că procedura este fără tip (nu întoarce nimic) dar în lista de parametri se transmite adresa rezultatului, ceea ce ar corespunde unui prototip C de forma:

```
void _pro_add(long x1, long x2, long *adr_rez);
```

Dacă presupunem că adresa rezultatului este de tip FAR, secvența de apel a procedurii va fi:

```
push ax ; Salvare temporară AX
;
; Aici începe secvența de apel
;
push word ptr n1+2 ; Partea high la adrese mari
push word ptr n1 ; Partea low la adrese mici
push word ptr n2+2 ; Similar pentru n2
push word ptr n2
mov ax, SEG rez ; Adresă de segment
push ax ; La 286 se poate direct:
; push SEG rez
mov ax, OFFSET rez ; Offset
push ax
call far ptr pro_add ; Apel
add sp, 12 ; Descărcare stivă
;
; Aici se termină secvența de apel
;
pop ax ; Refacere AX
```

Plasarea unor adrese FAR în stivă trebuie să țină seama de modul de reprezentare al pointerilor de tip FAR (definiți de exemplu prin directiva Define DoubleWord): la adrese mici se memorează offset-ul iar la adrese mari, adresa de segment. Dacă este cazul, se salvează în stivă registrele utilizate în secvența de apel (în cazul de față, AX).

Structura de tip șablon de acces se rescrie, adăugând noul parametru de tip adresă:

```
sablon_2 struc
    _bp dw ? ; BP
    _cs_ip dw 2 dup (?) ; Adresă de revenire
    adr_rez dd ? ; Adresă rezultat (FAR)
    n2_low dw ? ;
    n2_high dw ? ; Parametri
    n1_low dw ? ;
    n1_high dw ? ;
sablon_2 ends
```

În această variantă, procedura pro_add este:

```
pro_add proc far
    push bp ; Secvența tipică de acces
    mov bp, sp
    push ax
    push bx ; Salvări registre utilizate
    push es
    les bx, [bp].adr_rez ; Pointer la rezultat
    mov ax, [bp].n1_low
    add ax, [bp].n2_low ; Calcul parte low
```

```

mov es:[bx], ax          ; Depunere rezultat low
mov ax, [bp].n1_high
adc ax, [bp].n2_high     ; Calcul parte high
mov es:[bx+2], ax        ; Depunere rezultat high
pop es
pop bx                   ; Refacere registre
pop ax
pop bp                   ; Refacere BP
ret                      ; Revenire
pro_add endp

```

În procedură nu putem face presupuneri despre segmentul în care este definită variabila rezultat. Ca atare, pentru a o adresa, utilizăm perechea de registre ES:BX, încărcată cu adresa parametrului, preluată din stivă. Aici apare necesitatea ca datele de orice fel (în cazul de față adresa FAR a rezultatului) să fie reprezentate în stivă în același fel ca în memoria de date. Dacă nu am fi respectat convenția de reprezentare a adreselor FAR (offset-ul la adrese mici), instrucțiunea LES BX nu ar fi încărcat corect adresa în perechea de registre ES:BX. Accesul la variabila rezultat este ilustrat în Figura 6.2.

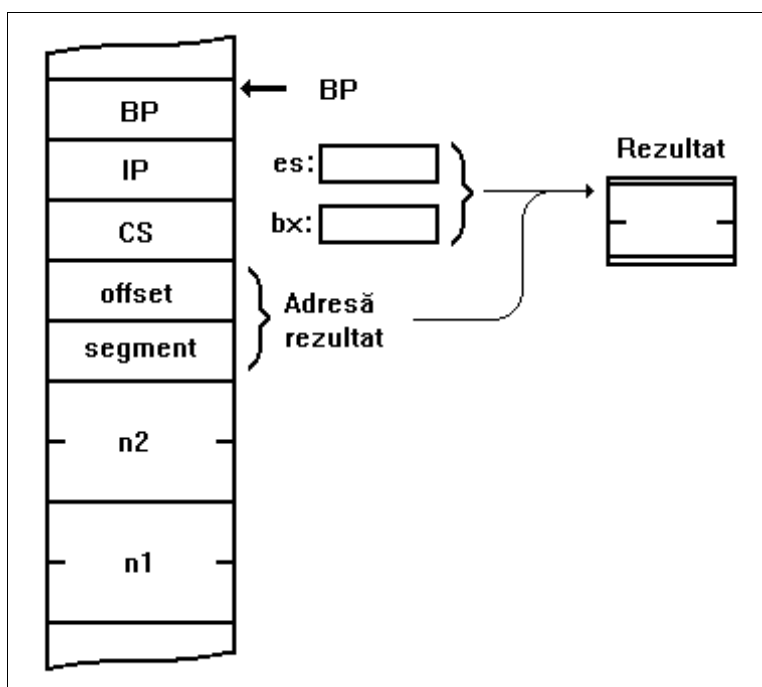


Figura 6.2 Transmiterea unei adrese FAR ca parametru

În toate exemplele de mai sus, descărcarea stivei a fost făcută de către programul apelant, printr-o instrucțiune ADD SP. Este posibilă și varianta în care descărcarea stivei se face de către procedură. Acest lucru se implementează printr-o instrucțiune return de forma:

```
ret N
```

unde N este numărul octeților care au fost puși pe stivă ca parametri. Evident, în acest caz nu se mai scrie instrucțiunea ADD SP în programul apelant.

Problematika transferului parametrilor prin stivă trebuie cunoscută în amănunt atunci când interfațăm module ASM cu module scrise în limbaje de nivel înalt. Este posibil ca diverse proprietăți să difere de la limbaj la limbaj, sau chiar de la compilator la compilator.

Spre exemplu, compilatoarele Borland C utilizează următoarea tehnică de transfer a parametrilor:

- parametrii sunt evaluați și plasați în stivă în ordinea inversă a listei de argumente a funcției, adică primul parametru din listă este în vârful stivei, imediat după adresa de revenire;
- stiva este descărcată de programul apelant.

În schimb, compilatoarele Borland Pascal lucrează exact pe dos:

- parametrii sunt plasați în stivă în ordinea din lista de argumente a procedurii (funcției), adică ultimul parametru din listă este în vârful stivei, imediat după adresa de revenire;
- stiva este descărcată de programul apelant.

Imaginea stivei la intrarea într-o procedură C, respectiv Pascal de forma:

```
void f_C (int par_a, int par_b, int par_c);  
procedure f_Pascal (int par_a, par_b, par_c)
```

este ilustrată în Figura 6.3.

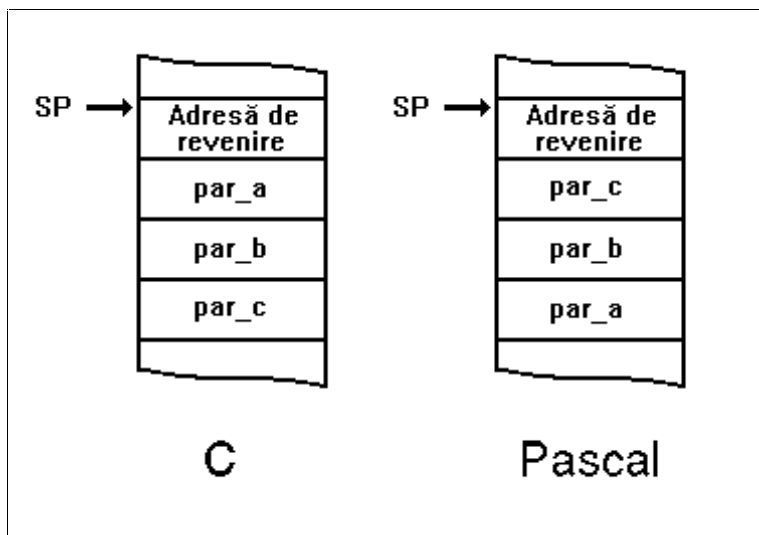


Figura 6.3 Transferul parametrilor în C și Pascal

Secvențele de apel ale celor două proceduri se scriu astfel:

```
push    c           ;  
push    b           ; Secvența de apel  
push    a           ;  
call    f_C         ; pentru o funcție C  
add     sp, 6       ;
```

respectiv:

```
push    a           ;  
push    b           ; Secvența de apel  
push    c           ; pentru o procedură  
call    f_Pascal    ; (funcție) Pascal
```

6.5 Întoarcerea datelor de către proceduri

Procedurile care întorc valori corespund funcțiilor din Pascal sau funcțiilor cu tip nevid din C. În limbaj de asamblare, ne punem problema în sens mai larg,

anume ce modalități există pentru a furniza un rezultat programului apelant (inclusiv prin tehnici neortodoxe). Aceste modalități sunt:

- a) în lista de parametri apar adresele rezultatelor, sau adresa unei zone de date care conține câmpuri pentru rezultate;
- b) rezultatele se întorc prin registre;
- c) rezultatele se întorc în vârful stivei.

Tehnica a) a fost deja descrisă la transmiterea parametrilor prin zonă de date sau prin stivă. Practic, în interiorul procedurii se depun explicit rezultatele la adresele conținute în parametrii formali respectivi.

Deși această tehnică nu este recomandată ca model în programarea structurată, ea nu se poate evita atunci când rezultatele au dimensiuni mari și sunt prin natura lor gestionate prin adrese. Un exemplu clar este cel al șirurilor de caractere sau al tablourilor în general.

Mai mult decât atât, compilatoarele de nivel înalt utilizează această tehnică (în mod transparent pentru utilizator) atunci când trebuie să se întoarse tipuri de volum mare prin numele funcției. Concret, se transmite către funcție adresa unei zone temporare de date (ca parametru suplimentar al funcției), în care să se depună rezultatul.

Tehnica b) este folosită cel mai frecvent. De obicei, se folosește registrul acumulator, eventual extins (adică AL, AX, respectiv DX:AX, după cum rezultatul este pe 1, 2 sau 4 octeți).

Dezavantajul acestei tehnici este limitarea la 32 de biți a tipului de date returnat. Totuși, compilatoarele Borland o utilizează ca metodă standard de returnarea a tipurilor de date de maxim 32 de biți.

Tehnica c) se folosește destul de rar, fiind total nestandard. Constă în plasarea rezultatelor în vârful stivei din momentul revenirii în programul apelant. Aceasta înseamnă că practic, rezultatele se suprapun în stivă peste parametrii de apel (se descarcă implicit stiva) și chiar peste adresa de revenire, ceea ce face foarte complicată scrierea procedurii.

Vom prezenta totuși această tehnică pentru că este un exemplu de operație complexă asupra stivei. Considerăm ca de obicei procedura `pro_add`, de tip far. Parametrii se plasează în stivă în ordinea `n1, n2`. Secvența de apel este:

```
push    word ptr n1+2
push    word ptr n1
push    word ptr n2+2
push    word ptr n2
call    far ptr pro_add          ; În acest moment, în
                                ; vârful stivei se ; găsește rezultatul
                                ; adunării
pop      ax                     ; Preia
mov      word ptr rez, ax        ; rezultat
pop      ax                     ; și descarcă
mov      word ptr rez + 2, ax     ; stiva
```

Știm acum ce ar trebui să execute procedura `pro_add`. Pentru a vedea exact ce acțiuni trebuie implementate, se pornește de la conținutul stivei în momentul intrării în procedură și de la cum

trebuie să arate stiva înainte de instrucțiune de revenire (RET) din procedură. Aceste două situații sunt ilustrate în Figura 6.4.

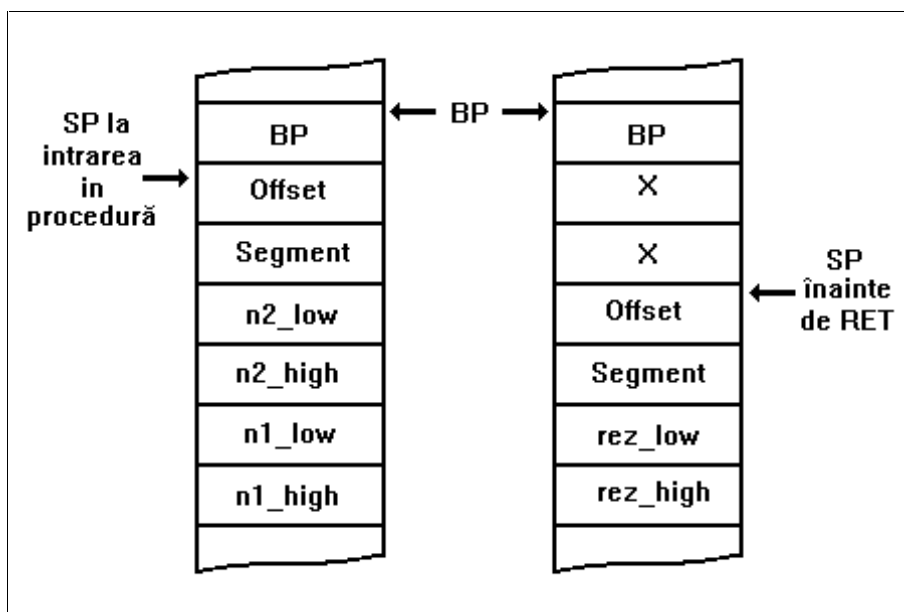


Figura 6.4 Întoarcerea rezultatelor în vârful stivei

Evident, procedura trebuie să construiască imaginea stivei din momentul revenirii în programul apelant, iar secvența de apel trebuie să readucă registrul SP la valoarea inițială.

Pornind de la aceste considerente, definim structuri de acces la stivă conform șablonului de la intrare, respectiv de la ieșire. Pentru șablonul de intrare utilizăm structura sablon_3:

```
sablon_3 struc
    _bp_3      dw ?      ; BP
    _ip_3      dw ?      ; Offset revenire
    _cs_3      dw ?      ; Segmentrevenire
    n2_low     dw ?      ;
    n2_high    dw ?      ; Parametri
    n1_low     dw ?      ;
    n1_high    dw ?      ;
sablon_3 ends
```

Pentru șablonul de ieșire definim structura sablon_4:

```
sablon_4 struc
    _bp_4      dw ?
    dw ?
    dw ?
    _ip_4      dw ?
    _cs_4      dw ?
    rez_low    dw ?
    rez_high   dw ?
sablon_4 ends
```

Câmpurile _bp_3 și _bp_4 se găsesc în aceeași poziție din memorie. S-au utilizat nume diferite deoarece sintaxa structurilor impune acest lucru.

Din imaginile stivei din Figura 6.4 rezultă și operațiile care trebuie executate. După calculul sumei, adresa de revenire va trebui deplasată cu 4 octeți în jos în stivă, rezultatul se va depune după noua poziție a adresei de revenire, se va

poziționa registrul SP pe adresa de revenire și se va executa RET. Procedura pro_add se implementează astfel:

```

pro_add proc far
    push    bp
    mov     bp, sp
    push    ax                ; Salvare
    push    dx                ; registre folosite
    mov     ax, [bp].n1_low   ; Calcul
    add     ax, [bp].n2_low   ; rezultat
    mov     dx, [bp].n1_high
    adc     dx, [bp].n2_high
    mov     [bp].rez_low, ax  ; Depunere rezultat
    mov     [bp].rez_high, dx ; conform șablonului de ieșire
    mov     ax, [bp]._ip_3    ; Pregătire offset adresă
    mov     [bp]._ip_4, ax    ; de revenire
    mov     ax, [bp]._cs_3    ; Pregătire segment adresă
    mov     [bp]._cs_4, ax    ; de revenire
    pop     dx                ; Refacere
    pop     ax                ; registre folosite
    pop     bp                ; Refacere BP
    add     sp, 4             ; Poziționare SP pe adresa
    ; de revenire
    ret
pro_add endp

```

În unele situații este posibil ca zona de stivă în care se depun rezultatele să fie suprapusă peste vechea zonă în care se găsește adresa de revenire. În acest caz, se plasează întâi adresa de revenire în noua poziție din stivă și apoi se depune rezultatul.

6.6 Proceduri cu zone de date proprii (variabile locale)

În multe cazuri, nu putem alocă toate variabilele dintr-o procedură în registrele procesorului. În această situație, procedura trebuie să utilizeze variabile locale proprii, altele decât parametrii formali și decât variabilele alocate în registre. Evident, se pune problema zonelor de memorie în care să fie alocate aceste variabile locale.

Există trei modalități de bază pentru această alocare: în segmentul de date global (.data) vizibil din toate modulele, în stivă sau într-un segment de date propriu.

Prima modalitate, care este evidentă, contrazice de fapt caracterul local al acestor variabile și nu se recomandă a fi folosită. Vor fi deci detaliate celelalte două metode.

6.6.1 Variabile locale definite în stivă

Ca și tehnica de transmitere a parametrilor prin stivă, această metodă este standardizată, fiind de fapt o extensie a tehnicii de transfer prin stivă. Operațiile care se execută la intrarea în procedură sunt următoarele:

- se salvează BP în stivă;
- se decrementează registrul SP cu numărul necesar de octeți pentru variabilele locale;
- se copiază SP în BP;

- se salvează (eventual) registrele folosite în procedură;
- se adresează parametri formali și variabilele locale conform șablonului stivei.

În secvența de ieșire din procedură, se execută următoarele operații:

- se refac registrele salvate;
- se incrementează SP cu același număr de octeți cu care a fost incrementat în secvența de intrare;
- se reface registrul BP din stivă;
- se revine în programul apelant (eventual cu descărcarea stivei de parametri.

Pentru exemplificare, să considerăm o procedură `pro_local`, de tip NEAR care are doi parametri formali de tip WORD și trei variabile locale, toate de tip word. Persupunem că stiva este descărcată de parametri formali de către procedură. Conform operațiilor de mai sus, definim o structură șablon pentru accesul în stivă:

```
sablon struc
    loc_1      dw ?
    loc_2      dw ?
    loc_3      dw ?
    _bp        dw ?
    _ip        dw ?
    par_2      dw ?
    par_1      dw ?
sablon ends
```

Schema de dezvoltare a procedurii `pro_local` este:

```
pro_local proc near
    push    bp                ; Salvare BP
    sub     sp, 6             ; Spațiu pentru variabile locale
    mov     bp, sp           ; Acces prin BP
                                ; Acces la parametri formali prin
                                ; expresiile [bp]. par_1, [bp].par_2
                                ; Acces la variabilele locale prin
                                ; expresiile [bp].loc_1, [bp].loc_2, [bp].loc_3
    add     sp, 6             ; Refacere spațiu local
    pop     bp               ; Refacere BP
    ret 4                    ; Revenire cu descărcare
pro_local endp
```

Imaginea stivei după secvența de intrare în procedură este ilustrată în Figura 6.5.

Variabilele alocate în stivă au unele caracteristici care derivă din metoda de alocare:

- spațiul de memorie alocat nu există decât pe durata procedurii;
- variabilele nu au alocate adrese fixe de memorie: adresele depind de poziția curentă a stivei;
- variabilele nu-și păstrează valoarea de la un apel la altul.

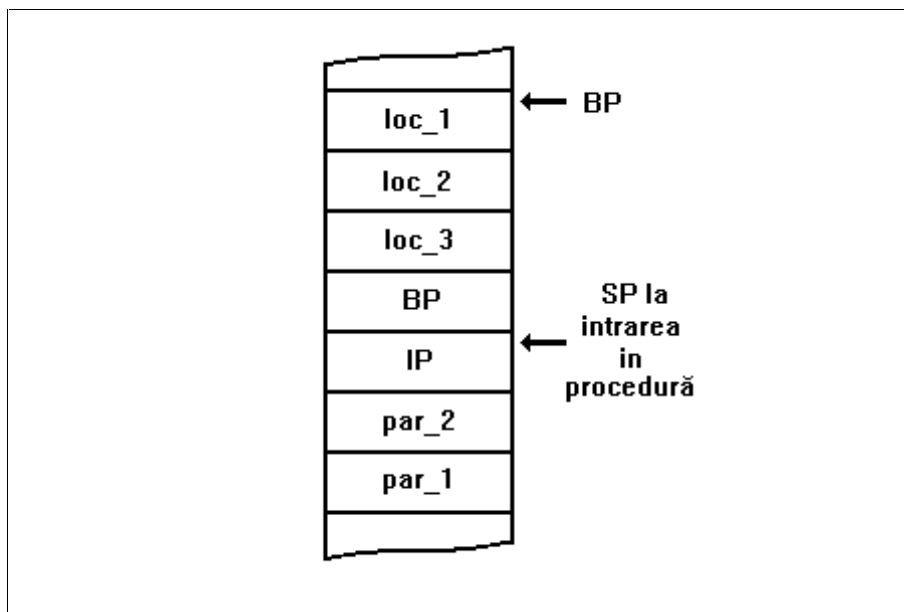


Figura 6.5 Variabile locale alocate în stivă

Cea de-a doua proprietate este mai importantă, deoarece există situații în care se dorește în mod explicit ca o variabilă locală să-și păstreze valoarea de la un apel la altul al procedurii. În acest caz, e obligatorie alocarea în segmente proprii de date, deci la adrese fixe.

6.6.2 Variabile locale alocate static

Alocarea variabilelor în segmente proprii de date se mai numește și alocare statică, deoarece variabilele au asociate adrese fixe de memorie.

Ca mod practic de implementare, se definește un segment de date local cu ajutorul directivei `SEGMENT`, ceea ce previne gruparea acestui segment cu alte segmente. Accesul la segmentul local se va realiza prin unul din registrele `DS` sau `ES`, urmând ca celălalt registru de segment de date să fie utilizat (dacă este cazul) pentru accesul la segmentul de date al programului apelant.

Să considerăm aceeași procedură `pro_local`, de data aceasta de tip `FAR`, cu trei variabile locale și doi parametri formali. Definim segmentul local prin:

```
local_data segment
    loc_1  dw ?
    loc_2  dw ?
    loc_3  dw ?
local_data ends
```

și convenim să accesăm acest segment prin registrul `ES`. Ca atare, definiția procedurii `pro_local` va fi încadrată de directive `ASSUME` corespunzătoare.

Parametrii formali sunt transmiși prin stivă, conform șablonului de acces:

```
sablon struc
    _bp          dw ?
    _cs_ip       dd ?
    par_2        dw ?
    par_1        dw ?
sablon ends
ASSUME es:local_data
pro_local proc far
    push bp
```

```

        mov     bp, sp
        push    es
        mov     es, SEG local_data
        ;
        ; DS : așa cum vine din programul apelant
        ; ES : poziționat pe segmentul local
        ;
        ; Acces la parametrii formali prin
        ; expresiile [bp]. par_1, [bp].par_2
        ;
        ; Acces la variabilele locale prin
        ; expresiile loc_1, loc_2, loc_3
        ; sau, mai clar, es:loc_1, es:loc_2, es:loc_3
        ;
        pop     es
        pop     bp
        ret     4
pro_local endp
ASSUME es:nothing

```

Există situații în care unul sau mai mulți parametri formali sunt pointeri (adrese) far. În acest caz, se salvează unul din registrele DS și ES în stivă și se încarcă pointerul respectiv într-o pereche adecvată de registre (cu instrucțiunile LDS sau LES).

Să presupunem, de exemplu, că unul din parametrii formali, descris în șablonul de acces prin par_far, indică un cuvânt care trebuie copiat în variabila locală loc_1. Secvența de copiere va fi:

```

        push    ds           ; Salvare
        lds     bx, [bp].par_far ; Încărcare pointer
        mov     ax, [bx]      ; Acces
        mov     es:loc_1, ax  ; Copiere în segment local
        pop     ds

```

Dacă unul din parametri este o adresă far de procedură, se poate executa direct un apel indirect, prin:

```

        call    dword ptr [bp].par_far

```

Să considerăm un exemplu în care utilizarea variabilelor locale statice este obligatorie, anume un generator simplu de numere aleatoare pe 16 biți, implementat prin procedura de tip FAR rand_asm. Procedura are ca parametru un întreg pe 16 biți fără semn (notat generic n) și întoarce în AX un număr aleator fără semn în domeniul 0...n-1.

Metoda de generare se bazează pe o variabilă locală X, asupra căreia se execută operația:

```

        X <-- partea mediană (X*X + C)

```

unde X*X este pătratul lui X (pe 32 de biți) iar C este o constantă pe 32 de biți. Prin parte mediană se înțeleg biții 8...23 din cei 32 de biți ai expresiei calculate (vezi Figura 6.6).

După ce s-a calculat noua valoare a lui X, se întoarce programului apelant valoarea X MOD n, deci un număr între 0 și n- 1. Toți operanzii se consideră fără semn.

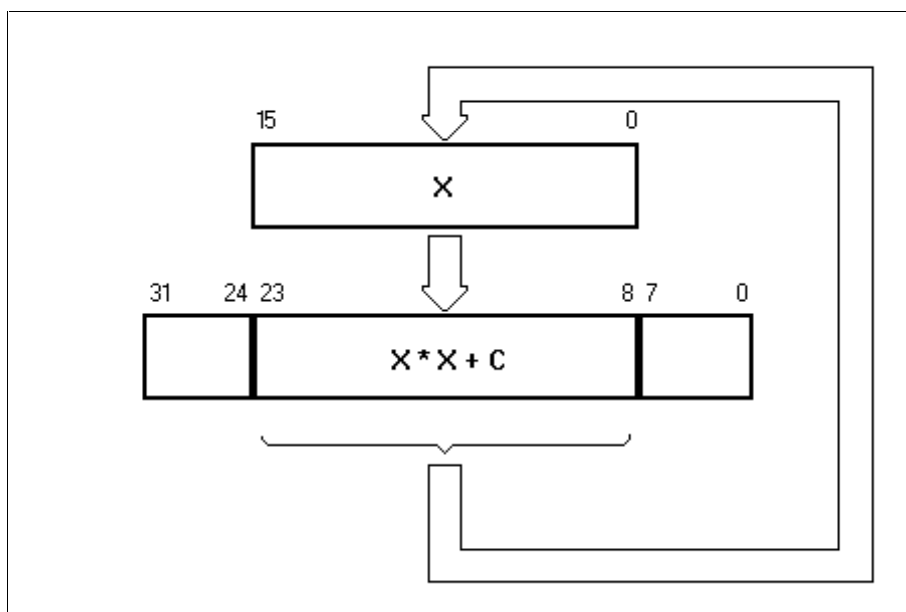


Figura 6.6 Un generator simplu de numere aleatoare

În astfel de operații, trebuie analizată cu atenție problema depășirilor. Calculul $X \bmod n$ se face printr-o împărțire a lui X la n , unde atât X cât și n sunt pe 16 biți. Se va executa de fapt o împărțire $DX:AX$ la n , cu $DX = 0$, fapt care asigură evitarea depășirii: cea mai mare valoare posibilă a lui X (65535), împărțită la cea mai mică valoare nebanală a lui n (2) nu produce depășire.

Caracterul pseudo-aleator al algoritmului de mai sus provine în mod esențial din faptul că variabila X își păstrează valoare de la un apel la altul al procedurii (vechea valoare a lui X participă la calculul noii valori).

Pentru accesul la parametrul n se utilizează o structură șablon.

Conform metodei de dezvoltare expuse mai sus, definim un segment local în care rezervăm spațiu pentru variabila de tip word X și pentru constanta de tip double-word C (ambele inițializate cu câte o valoare oarecare).

Deoarece va trebui să executăm o adunare pe 32 de biți, avem nevoie de acces explicit la părțile low și high ale constantei C , păstrînd definiția ei ca variabilă double-word. Acest lucru este realizat prin operatorul `THIS`, cu ajutorul căruia se definesc constantele simbolice `c_lo` și `c_hi`.

Calculele vor folosi registrul `AX`, respectiv perechea de registre `DX:AX`, care este operand implicit la înmulțiri și împărțiri. Implementarea este următoarea (presupunem un fișier sursă cu numele `rand.asm`):

```
;
; Fișier RAND.ASM
;
.model large
    public rand_asm, init_asm

    sablon struc
        _bp dw ?           ; Șablon de
        _cs_ip dd ? ; de acces
        n dw ? ; la stivă
    sablon ends
```

```

l      local_data segment
        x      dw 111                ; Variabila locală statică X
        c_lo   equ this word
        c_hi   equ c_lo + 2
        c      dd 115321            ; Constanta C
    local_data ends

.code
assume es:local_data
    rand_asm proc far
        push    bp                    ; Secvența standard
        mov     bp, sp                ; de intrare
        pushf                                ; Salvare bistabili
        cmp     [bp].n, 2              ; Test caz banal (n = 0,1)
        mov     ax, [bp].n            ; Se întoarce chiar n
        jb      gata
        push    es                    ; Salvări registre
        push    dx                    ; folosite
        mov     ax, SEG local_data    ; Poziționare ES
        mov     es, ax                ; pe segmentul local
        mov     ax, es:x              ; Variabila X
        mul     es:x                  ; Ridicare la pătrat (DX:AX)
        add     ax, c_lo               ; Adunare
        adc     dx, c_hi               ; cu C
        mov     al, ah                ; Luarea părții mediane
        mov     ah, dl                ; din DX:AX în AX
        mov     es:x, ax              ; Depunere în X
        xor     dx, dx                ; Pregătire împărțire
        div     [bp].n                ; Împărțire (0:X) la n
        mov     ax, dx                ; Câțul se pune în AX
        pop     dx                    ; Refaceri
        pop     es                    ; registre
gata:
        popf                            ; Refacere bistabili
        pop     bp                    ; Secvența standard
        ret                            ; de ieșire
    rand_asm endp
;
; Inițializare generator
;
    init_rand proc far
        push    bp
        mov     bp, sp
        push    es                    ; Salvări
        push    ax
        mov     ax, SEG local_data
        mov     es, ax
        mov     ax, [bp].n            ; Copiere parametru
        mov     es:x, ax              ; în variabila X
        pop     ax
        pop     es                    ; Refaceri
        pop     bp
    ret
    init_rand endp
end

```

Procedura rand_asm este completată cu o rutină de inițializare a generatorului, având un parametru de tip word transmis prin stivă, care inițializează variabila X. Se folosește aceeași structură șablon pentru accesul la stivă.

Să considerăm un exemplu de apel al acestor proceduri, pe care le presupunem definite în fișierul sursă `rand.asm`. Dorim inițializarea cu valori aleatoare în gama 0...9999 a unui tablou de 256 de întregi, pe care îl afișăm, îl sortăm crescător și apoi îl afișăm din nou. Pentru afișare și sortare utilizăm procedurile `tipvec` și `bubble`, dezvoltate în §2.7, pe care le presupunem definite într-un fișier sursă cu numele `bubble.asm`, care conține declarații `PUBLIC` ale procedurilor în cauză.

În programul principal (presupus în fișierul sursă `main.asm`), cele patru proceduri se declară ca simboluri externe și se folosesc conform metodei proprii de transfer al parametrilor:

```
;
; Fișier MAIN.ASM
;
.model large
include io.h
    extrn  rand_asm: far, init_rand: far, tipvec: far, bubble: far
.stack 1024
.data
    vector dw 256 dup (?)
    dim     dw ($-vector)/2
.code
start:
    init_ds_es
    mov     ax, 1131                ; Valoare inițială
    push    ax                     ; a generatorului
    call    init_rand
    add     sp, 2                   ; Descărcare stivă
    ;
    mov     cx, dim                 ; Dimensiune vector
    lea     bx, vector              ; Adresă vector
reluare:
    mov     ax, 10000               ; Domeniu 0...9999
    push    ax
    call    rand_asm
    add     sp, 2                   ; Descărcare stivă
    mov     [bx], ax                ; Depunere valoare
    add     bx, 2                   ; Actualizare adresă
    loop    reluare
    ;
    putsi   <'Vector nesortat', cr, lf>
    lea     si, vector              ; Adresă tablou
    mov     cx, dim                 ; Număr de elemente
    call    tipvec                  ; Afișare tablou nesortat
    ;
    lea     bx, vector              ; Adresă tablou
    mov     cx, dim                 ; Număr de elemente
    call    bubble                  ; Sortare tablou
    ;
    putsi   <cr, lf, 'Vector sortat', cr, lf>
    lea     si, vector
    mov     cx, dim
    call    tipvec                  ; Afișare tablou sortat
    ;
    exit_dos                        ; Ieșire în DOS
end start
```

Secvența de dezvoltare a acestei aplicații este:

```
> tasm rand.asm  
> tasm bubble.asm  
> tasm main.asm  
> tlink main rand bubble io, main
```

În urma căreia rezultă fișierul executabil **main.exe**.

Exemplul de mai sus constituie și un model de dezvoltare modulară a unei aplicații. Se observă că o proiectare îngrijită a modulelor permite refolosirea lor în diverse contexte.