Capitolul 6 – partea a II-a
Tehnici de programare în limbaj de asamblare

6.7 Proceduri recursive

În limbajul de asamblare nu există restricţii asupra recursivităţii: orice procedură proc_a se poate apela pe ea însăşi (recursivitate directă) sau poate apela o procedură proc_b care, la rândul ei apelează procedura proc_a (recursivitate indirectă).

În dezvoltarea unei proceduri recursive trebuie pornit de la specificarea algoritmului recursiv. Specificarea unui asemenea algoritm pune în evidenţă un caz de bază (care opreşte recursivitatea) şi un caz general, în care se invocă acelaşi algoritm, aplicat altui set de date. Specificarea corectă a cazului de bază şi garantarea faptului că acesta este atins întotdeauna, indiferent de setul de date de intrare primit, sunt punctele-cheie ale proiectării unui algoritm recursiv.

Să considerăm, ca exemplu, o procedură de afişare în baza 10 a unui număr pe 16 biţi fără semn. La nivelul funcţiilor de intrare/ieşire, avem posibilitatea de a afişa caractere ASCII. Dorim deci un algoritm recursiv care să genereze cifrele zecimale ale numărului, în ordinea afişării. În $2.2.5 a fost dat un algoritm nerecursiv, care producea cifrele zecimale ale numărului în ordine inversă.

Algoritmul recursiv se specifică în felul următor:

```
putu_proc (n)
{
        dacă (n < 10)
                afişează (n + '0');
        altfel {
                putu_proc (n/10);
                afişează (n MOD 10 + '0');
        }
}
```

Cazul de bază este (n < 10), pentru care se face doar afişarea cifrei şi revenirea în programul apelant. În Figura 6.7 sunt ilustrate apelurile recursive în cazul afişării numărului 123 şi valorile succesive ale parametrului n.

Regulile generale de implementare a procedurilor recursive sunt:

- fiecare apel al procedurii nu trebuie să afecteze datele apelurilor precedente (procedura să fiereentrantă);
- nu se folosesc variabile locale alocate static, ci numai în stivă sau în registre;
- parametrii se transmit prin stivă (se asigură implicit faptul că parametrii sunt locali fiecărui apel);
- registrele ale căror valori au semnificaţie de la un apel la altul se salvează în stivă;
- dacă procedura întoarce valori prin registre, rezultatele intermediare trebuie memorate în stivă.

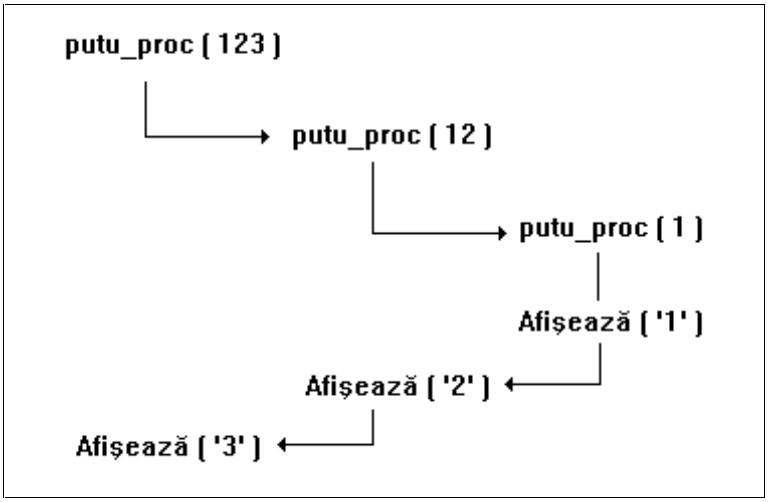

Figura 6.7 Apelul recursiv al procedurii putu_proc

Pentru implementarea procedurii putu_proc, se va utiliza funcţia DOS 2, care afişează la consolă caracterul primit în registrul DL. Pentru accesul la stivă, considerăm structura şablon:

```
sablon struc
        _bp             dw ?
        _cs_ip          dw ?
        n               dw ?
sablon ends
```

Pentru o implementare mai eficientă, rescriem algoritmul recursiv în forma:

```
putu_proc (n)
{
        dacă (n < 10)
                x = n;
        altfel {
                putu_proc (n/10);
                x = n MOD 10;
        }
        afişează (x + '0');
}
```

Alocarea variabilelor este următoarea:

- n - în stivă, accesibil prin expresia [bp].n
- x - în registrul DL (care se salvează/restaurează)

Implementarea este următoarea:

```
        putu_proc proc far
                push    bp                      ; Secvenţa
                mov     bp,sp                   ; standard
                push    dx                      ; Salvări
                push    ax                      ; registre
                push    bx                      ; folosite
;
                mov     ax, [bp].n              ; Test caz
                cmp     ax, 10                  ; de bază
                mov     dl, al                  ; x <-- n
                jb      p_u_1                   ; Salt la afişare
;
                mov     bx, 10                  ; Caz general
                xor     dx, dx                  ; Se calculează
```

```
                div     bx                      ; n/10 şi n MOD 10
                                                ; AX = n/10
                                                ; DX (DL) = x = n MOD 10
                push    ax                      ; Apel recursiv cu
                call    far ptr putu_proc       ; n/10 ca parametru
p_u_1:
                add     dl, '0'                 ; x + '0'
                mov     ah, 2                   ; Cod funcţie DOS
                int     21H                     ; Afişare
;
                pop     bx                      ; Refaceri
                pop     ax                      ; registre
                pop     dx
                pop     bp
                retf    2                       ; Revenire cu descărcarea stivei
putu_proc endp
```

În implementarea de mai sus, este esenţială integritatea variabilei x de la un apel la altul, adică salvarea/restaurarea registrului DX. Pentru a preveni depăşirile, se face împărţire 32 de biţi la 16 biţi, deci constanta 10 se încarcă într-un registru de 16 biţi. După împărţire, DX (de fapt DL) conţine restul n MOD 10, iar AX câtul (n/10). Se pune deci AX în stivă şi se apelează recursiv procedura. După revenire, se afişează cifra din DL.

Putem acum dezvolta o procedură care să afişeze un număr pe 16 biţi cu semn, după algoritmul:

```
        puti_proc (n) {
                dacă (n < 0) {
                        n = - n;
                        afişează ('-');
                }
                putu_proc (n)
        }
```

Dacă n e negativ, se complementează faţă de 2 şi se afişează semnul '-', după care se afişează numărul n complementat. Implementarea este următoarea:

```
        puti_proc proc far
                push    bp                      ; Secvenţa
                mov     bp, sp                  ; standard
                push    ax                      ; Salvări
                push    dx                      ; registre
;
                mov     ax, [bp].n              ; Parametrul n
                or      ax, ax                  ; Poziţionare bistabili
                jns     p_i_1                   ; Test bit de semn
                mov     dl, '-'                 ; Afişare
                mov     ah, 2                   ; semn '-'
                int     21H
                neg     word ptr [bp].n         ; Complementare n
        p_i_1:
                push [bp].n                     ; Pregătire parametru
                call far ptr putu_proc          ; Apel putu_proc
;
                pop     dx                      ; Refaceri
                pop     ax                      ; registre
                pop     bp
                retf    2                       ; Revenire cu descărcarea stivei
        puti_proc endp
```

Se observă plasarea forma directă de plasare a parametrului [bp].n în stivă, pentru apelul procedurii putu_proc.

Putem scrie acum şi macroinstructiuni de apel adecvate:

```
puti macro X
        push    X
        call    far ptr puti_proc
endm
putu macro X
        push    X
        call    far ptr putu_proc
endm
```

După modelul procedurii putu_proc, putem scrie o procedură care să afişeze la consolă reprezentarea unui număr pe 16 biţi fără semn într-o bază de numeraţie dată. Baza se consideră în domeniul 2...36, iar cifrele într-o bază mai mare ca 10 sunt '0'...'9', 'A', 'B' etc. Limitarea bazei la valoarea 36 derivă din această alegere a cifrelor.

Algoritmul este următorul:

```
put_base (n, baza) {
        dacă (baza < 2 sau baza > 36)
                return;
        dacă (n < baza)
                x = n;
        altfel {
                put_base (n/baza, baza);
                x = n MOD baza;
        }
        dacă (x < 10)
                afişează (x + '0');
        altfel
                afişează (x + 'A' - 10);
}
```

Se începe printr-un test de corectitudine a bazei: dacă aceasta nu este în domeniul admisibil, se revine imediat în programul apelant. Se execută apoi acelaşi algoritm recursiv ca în procedura putu_proc, înlocuind constanta 10 cu variabila baza. Afişarea cifrei curente x se face diferenţiat: dacă este mai mică sau egală decât 10, se afişează ca cifra zecimală; altfel, se afişează ca literă.

Pentru accesul la stivă, se utilizează structura şablon de mai jos

```
sablon struc
        _bp     dw ?
        _cs_ip dd ?
        baza    dw ?
        n       dw ?
sablon ends
```

Implementarea este următoarea (se consideră că stiva este descărcată de programul apelant):

```
put_base proc far
        push    bp                          ; Secvenţa standard
        mov     bp, sp
        cmp     word ptr [bp].baza, 2       ; Test baza corectă
        jb      err_exit ; baza < 2
        cmp     word ptr [bp].baza, 36
```

```
            ja      err_exit                    ; baza > 36
;
            push    ax                          ; Salvări
            push    dx                          ; registre
            mov     ax, [bp].n                  ; Test
            cmp     ax, [bp].baza               ; n < baza ?
            mov     dl, al                      ; x <-- n
            jb      p_b_1                       ; Salt la afişare
            xor     dx, dx                      ; Calcul n/baza
            div     [bp].baza                   ; şi n MOD baza
                                                ; AX = n/baza
                                                ; DX (DL) = n MOD baza
        ;
        ;       Pregătire parametri pentru apelul recursiv
        ;
            push    ax                          ; n/baza
            push    [bp].baza                   ; baza
            call    far ptr put_base            ; Apel recursiv
            add     sp, 4                       ; Descărcare stivă
        p_b_1:
            mov     ah, 2                       ; Cod funcţie DOS
            cmp     dl, 10                      ; x < 10 ?
            jae     p_b_2
            add     dl, '0'                     ; x + '0'
            jmp     p_b_3
        p_b_2:
            add     dl, 'A' - 10                ; x + 'A' - 10
        p_b_3:
            int     21H                         ; Afişare
            pop     dx                          ; Refaceri
            pop     ax                          ; registre
        err_exit:
            pop     bp
            ret                                 ; Revenire
        put_base endp
```

Deoarece toate variabilele implicate sunt fără semn, se folosesc salturi condiţionate de ti "below" şi "above".

Un mic program de test (listat în continuare) citeşte un număr şi o bază de la consolă şi afişează numărul în baza dată, după care reia procesul de citire. Dacă numărul introdus este 0, programul se termină.

```
        .model large
        include io.h
        .stack 1024
        .code
            start:
                init_ds_es
            reluare:
                putsi   <cr, lf, 'Numar = '>    ; Prompter
                getu                            ; Citire întreg
                or      ax,ax                   ; Test oprire ?
                jz      exit                    ; Salt la ieşire
                push    ax                      ; Plasare n în stivă
                putsi   <'Baza = '>             ; Prompter
                getu                            ; Citire întreg
                push    ax                      ; Plasare baza în stivă
                call    far ptr put_base        ; Apel
                add     sp, 4                   ; Descărcare stivă
```

```
                jmp     reluare                 ; Reluare
        exit:
                exit_dos
        end start
```

Un caz deosebit îl constituie funcţiile recursvie, adică procedurle care întorc valori. Trebuie acordată atenţie memorării valorilor întoarse de apelurile recursive. Să considerăm o funcţie recursivă care întoarce numărul Fibonacci de ordinul n, definită recusiv prin:

```
        fib (0) = fib (1) = 1
        fib (n) = fib (n-1) + fib (n-2), pentru n >= 2
```

Această funcţie a fost evaluată prin iteraţie în $2.4.4. Vom studia acum implementarea recursivă.

Se observă că, pentru valuarea lui fib(n), sunt necesare două apeluri recursive şi salvarea rezultatului întors de primul apel. Presupunem că parametrul n se transmite prin stivă, că procedura este de tip FAR, că întoarce rezultatul în AX (ca număr fără semn) şi că stiva este descărcată de programul apelant. Şablonul de acces este definit prin structura:

```
        sablon struc
                _bp     dw ?
                _cs_ip dd ?
                n       dw ?
        sablon ends
```

iar implementarea este următoarea:

```
fib proc far
;
;       Primeste in stivă numărul n
;       Intoarce în AX valoarea fib (n)
;
        push    bp
        mov     bp, sp
        push    bx                      ; Salvare registru folosit
;
        mov     ax, [bp].n              ; Preia argumentul n
        cmp     ax, 1                   ; Cazul de bază: n <= 1
        jbe     fib_1                   ; Salt la evaluare imediată
;
        dec     ax                      ; Pregătire apel
        push    ax                      ; pentru n-1
        call    fib                     ; AX = fib (n-1)
        add     sp, 2                   ; Descărcare stivă
        mov     bx, ax                  ; Rezultat intermediar
;
        mov     ax, [bp].n              ; Pregătire
        dec     ax                      ; apel
        dec     ax                      ; pentru
        push    ax                      ; n-2
        call    fib                     ; AX = fib (n-2)
        add     sp,2                    ; Descărcare stivă
;
; În acest moment, AX = fib (n-2), BX = fib (n-1)
;
        add     ax, bx                  ; Calcul fib (n)
        jmp     fib_2
fib_1:
```

```
        mov     ax, 1                   ; Evaluare fib (0) şi fib (1)
fib_2:
        pop     bx                      ; Restaurare
        pop     bp
        ret                             ; Revenire
fib endp
```

În implementarea de mai sus, este esenţială salvarea şi restaurarea registrului BX, care memorează rezultattul intermediar fib (n-1). Programul de test al acestei proceduri este lăsat ca exerciţiu pentru cititor.

Nu putem încheia discuţia despre proceduri recursive fără a vorbi despre problema clasică a turnurilor din Hanoi. În această problemă, se consideră n discuri de diametre distincte, aşezate pe un suport vertical (turn sursă), în ordinea descrescătoare a diametrelor (vezi Figura 6.8). Se cere să se mute întreg ansamblul pe un alt turn (destinaţie), folosind un turn de manevră. Discurile se pot muta unul câte unul, cu restricţia că un disc nu se poate aşeza decât deasupra unui disc de diametru mai mare.

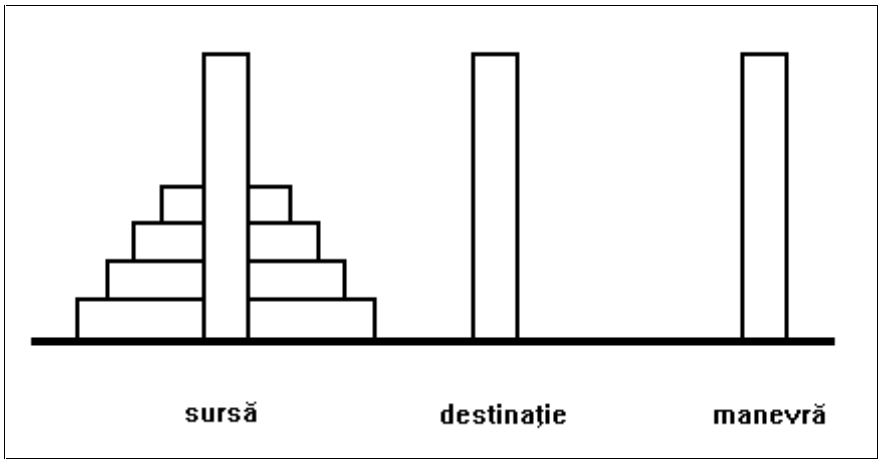

Figura 6.8 Turnurile din Hanoi

Algoritmul care rezolvă această problemă este în mod natural recursiv şi se bazează pe observaţia că, dacă n = 1, adică există un singur disc, acesta se poate muta direct pe turnul destinaţie. Dacă notăm simbolic operaţia de mutare prin:

```
        move (n, sursă, destinaţie, manevră)
```

unde sursă, destinaţie şi manevră sunt cele trei turnuri, algoritmul se implementează prin:

```
        move (n, sursă, destinaţie, manevră)
        {
                dacă (n = 1)
                        Mută discul de pe sursă pe destinaţie
                altfel {
                        move (n-1, sursă, manevră, destinaţie)
                        move (1, sursă, destinaţie, manevră)
                        move (n-1, manevră, destinaţie, sursă)
                }
        }
```

Se mută primele n-1 discuri de pe turnul sursă pe turnul manevră, folosind ca spaţiu de lucru turnul destinaţie. În acest moment, turnul sursă conţine un singur disc, turnul destinaţie este liber iar turnul manevră conţine n-1 discuri.

Se mută unicul disc de pe turnul sursă pe turnul destinaţie. În acest moment turnul sursă este liber, turnul destinaţie conţine discul cu diametrul cel mai mare iar turnul manevră conţine celelalte n-1 discuri. Se aplică recursiv algoritmul, mutând n-1 discuri de pe turnul manevră pe turnul destinaţie şi folosind turnul sursă ca spaţiu de lucru.

Pentru un n dat, se deplasează explicit discul cu diametrul cel mai mare, ceea ce asigură îndeplinirea restricţiei din enunţ.

Implementarea algoritmului se va face prin mesaje explicite la consolă, turnurile fiind codificate prin caractere.

Implementarea algoritmului, împreună cu un program de test, este listată în continuare. Structură şablon defineşte ordinea de plasare a parametrilor în stivă: n, sursă, destinaţie, manevră, toate de tip word. Toate apelurile recursive vor utiliza această ordine.

Simularea mutărilor se face prin afişarea unui mesaj, definit în zona de date. Câmpurile sursa_asc şi dest_asc sunt variabile şi vor identifica (prin câte un caracter) turnurile respective. Când n = 1 (cazul de bază), se depun caracterele care identifică turnurile în şirul de caractere mesaj şi se afişează acest şir. Implementarea apelurilor recursive constă în plasarea în stivă a parametrilor în ordinea precizată şi apelul procedurii.

Programul principal citeşte ciclic de la consolă numărul n, afişează un mesaj de identificare, pregăteşte parametrii în stivă şi apelează procedura move. Bucla se opreşte când se introduce valoarea 0 pentru n.

```
.model large
include io.h
.stack   4096
sablon struc
        _bp             dw ?
        _cs_ip          dw 2 dup (?)
        manevra         dw ?
        dest            dw ?
        sursa           dw ?
        n               dw ?
sablon ends
.data
        mesaj           db cr, lf, 'Muta discul de pe turnul '
        sursa_asc       db ?
                        db ' pe turnul '
        dest_asc        db ?
                        db 0
.code
move proc far
;
; Muta n discuri de pe sursa pe dest
; folosind manevra ca zonă de lucru
;
        push    bp
        mov     bp, sp
        cmp     word ptr [bp].n, 1              ; Test caz de bază (n = 1)
        jne     move_1                          ; Salt la cazul general
;
; Mutare explicită
;
```

```
                mov     bx, [bp].sursa          ; Pregătire mesaj
                mov     sursa_asc, bl           ; Disc sursă
                mov     bx,[bp].dest
                mov     dest_asc,bl             ; Disc destinaţie
                puts    mesaj                   ; Afişare mesaj
                pop     bp
                retf    8                       ; Revenire cu descărcare
        move_1:
        ;
        ; Cazul general
        ;
                mov     ax, [bp].n              ; Pregătire parametri
                dec     ax                      ; n-1
                push    ax
                push    [bp].sursa              ; Sursă
                push    [bp].manevra            ; Actuala manevră devine
                                                ; destinaţie
                push    [bp].dest               ; Actuala destinaţie
                                                ; devine manevră
                call    far ptr move            ; Primul apel recursiv
        ;
                mov     ax, 1                   ; Pregătire parametri
                push    ax                      ; n = 1
                push    [bp].sursa              ; Sursă
                push    [bp].dest               ; Destinaţie
                push    [bp].manevra            ; Manevră
                call    far ptr move            ; Al doilea apel recursiv
        ;
                mov     ax, [bp].n              ; Pregătire parametri
                dec     ax
                push    ax ; n-1
                push    [bp].manevra            ; Actuala manevră devine
                                                ; sursă
                push    [bp].dest               ; Destinaţie
                push    [bp].sursa              ; Actuala sursă devine
                                                ; manevră
                call far ptr move               ; Al treilea apel recursiv
        ;
                pop     bp                      ; Revenire cu
                retf    8                       ; descărcare
        move endp
        ;
        ; Program principal
        ;
        start:
                init_ds_es
        reluare:
                putsi   <cr,lf,'Numar discuri: '>
                geti
                or      ax, ax
                jz      exit                    ; n = 0, stop
                putsi   <cr,lf,'Muta '>
                puti    ax ; Afişare n
                putsi   <' turnuri, de pe A pe B,'>
                putsi   <' folosind C ca'>
                putsi   <' manevra', cr, lf>
;
                push    ax                      ; n
                mov     ax, 'A'
```

```
        push    ax                      ; Sursă
        mov     ax, 'B'
        push    ax                      ; Destinaţie
        mov     ax, 'C'
        push    ax                      ; Manevră
        call    far ptr move            ; Apel
        jmp     reluare                 ; Reluare
exit:
        exit_dos
end start
```

La implementarea algoritmilor recursivi, trebuie făcută o estimare a spaţiului necesar de stivă. Spre exemplu, procedura move de mai sus necesită n nivele de apel recursiv. La fiecare nivel, se consumă 12 octeţi din stivă (vezi structura şablon). Ca atare, volumul necesr de stivă este de ordinul 12*n. Dacă estimăm că, într-o aplicaţie dată, valoarea lui n nu va creşte peste 100, stiva acelei aplicaţii trebuie să fie de cel puţin 1200 de octeţi.

Înainte de a trece la implementarea unui algoritm recursiv, trebui să ne convingem (uneori prin demonstraţii teoretice complicate), că algoritmul se termină (se atinge cazul de bază) după un număr finit de paşi. Nu toate funcţiile definite recursiv au această proprietate. De exemplu, funcţia f, "definită" pentru orice n pozitiv prin:

```
f(n) = 5, pentru n = 7
f(n) = f(f(n+2)), pentru n diferit de 7
```

nu este nici măcar o funcţie corect definită. Încercarea de a evalua f(5) conduce imediat la o tautologie (f(5) depinde de f(5)), iar pentru n diferit de 5 şi de 7 se obţine un ciclu infinit.

Complexitatea algoritmilor recursivi este un alt subiect important, dar care depăşeşte cadrul lucrării de faţă.

Trebuie deci acordată atenţie descrierii teoretice a algoritmului recursiv, înainte de a trece la implementare. Observaţiile despre implementarea algoritmilor recursivi sunt valabile şi în cazul limbajelor de nivel înalt.

6.8 Proceduri cu număr variabil de parametri

Dezvoltarea procedurilor cu număr variabil de parametri are ca scop, pe de o parte, punerea în evidenţă a flexibilităţii limbajului de asamblare şi, pe de altă parte, ilustrarea modului de implementare a unui mecanism care există în anumite limbaje de de nivel înalt (de exemplu în C).

Dezvoltarea unei proceduri cu număr variabil de parametri se bazează pe următoarele reguli:

- paramatrii se transmit prin stivă;
- există un număr (cel puţin 1) de parametri ficşi (care nu pot lipsi) şi care se transmit în vârful stivei (imediat după adresa de revenire);
- unul din parametrii care sunt totdeauna prezenţi conţine informaţii (codificate într-o formă oarecare) despre numărul şi tipul parametrilor actuali;
- în procedură se analizează parametrii ficşi, deducându-se numărul şi tipul parametrilor variabili, care sunt apoi extraşi din stivă;
- parametrii variabile ca număr transmişi în stivă trebuie să corespundă cu descrierea lor în parametrii ficşi;
- stiva este descărcată de către programul apelant.

Regulile de mai sus nu sunt alese întâmplător. Astfel, faptul că parametrii ficşi se găsesc în vârful stivei face ca aceştia să fie accesibili prin metoda standard. Programul apelant este cel care ştie precis câţi octeţi a plasat în stivă înainte de apel, deci este natural ca acesta să descarce stiva.

Se înţelege acum de ce, în limbajul C, ordinea plasării argumentelor unei proceduri în stivă este de la dreapta la stânga: în acest fel, la o procedură cu număr variabil de parametri, parametrii ficşi (care se află la începutul listei de argumente) vor fi plasaţi imediat după adresa de revenire.

Imaginea stivei la intrarea în procedură este ilustrată în Figura 6.9.

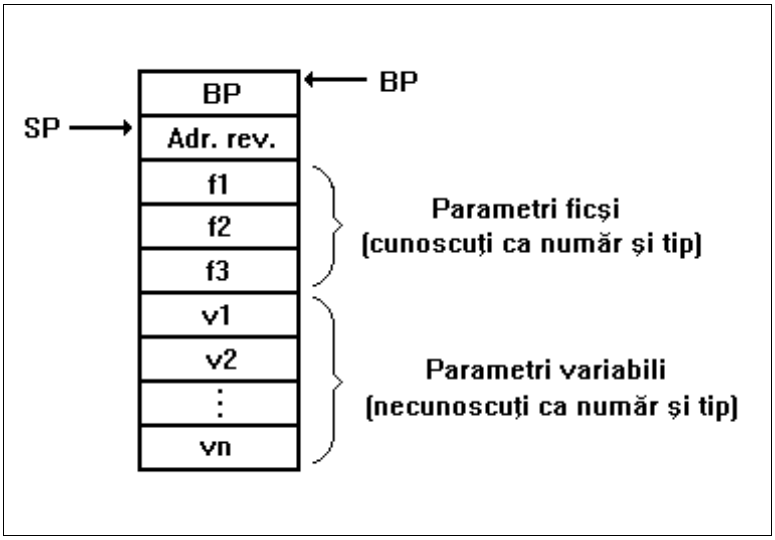

Figura 6.9 Stiva la intrarea într-o procedură cu număr variabil de parametri

Să considerăm un exemplu tipic de procedură cu număr variabil de parametri: o procedură printf_proc de afişare cu format. Există un parametru fix, de tip adresă NEAR a unui şir de caractere care descrie formatul de afişare. Şirul de caractere poate conţine:

- caractere obişnuite, care se afişează ca atare;
- descriptori de format, compuşi din caracterul % şi un al doilea caracter care descrie tipul afişării:
  - %s pentru şiruri de caractere;
  - %d pentru întregi pe 16 biţi cu semn;
  - %u pentru întregi pe 16 biţi fără semn.
- secvenţe escape:
  - '\n' pentru CR şi LF;
  - '\\' pentru caracterul '\';
  - '%%' pentru caracterul '%';

Pentru fiecare specificator de format, se presupune că există un parametru variabil în stivă, care trebuie afişat corespunzător. Pentru întregi, se transmite în stivă conţinutul care trebuie afişat, iar pentru şiruri, se transmite adresa de început, de tip NEAR. Se observă că, în toate situaţiile, se trnsmite în stivă un cuvânt (word), ceea ce facilitează implementeare. Dacă dimensiunile parametrilor variabili ar fi diferite, ar trebui folosit operatorul de conversie de tip PTR, pentru a încărca parametrul în mod corect.

Pentru afişarea propriu zisă se vor folosi macroinstrucţiunile de apel puts (şiruri), puti (întregi cu semn), putu (întregi fără semn) şi putc (afişare caracter), descrise în Capitolul 2.

Procedura şi toate adresele implicate se consideră de tip NEAR. În Figura 6.10 este descris conţinutul stivei la un apel tipic al acestei proceduri.

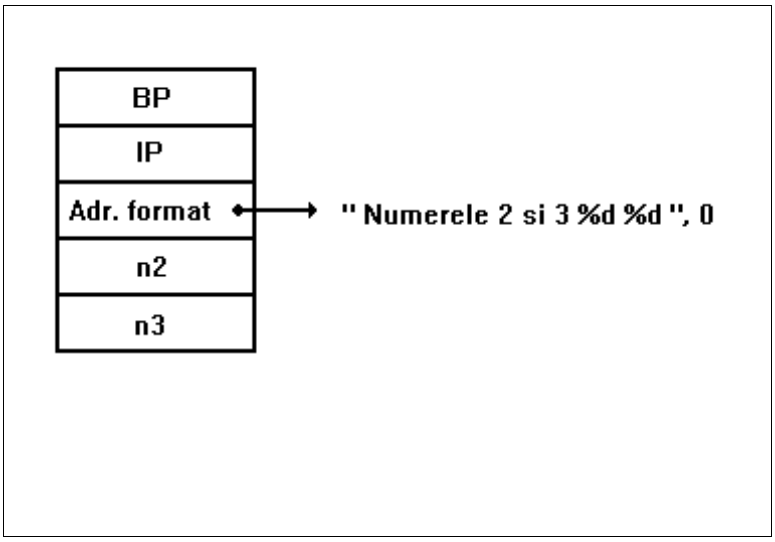

Figura 6.10 Conţinutul stivei la intrarea în printf_proc

Pentru accesul la stivă se va utiliza şablonul:

```
sablon struc
        _bp         dw ?
        _ip         dw ?
        format      dw ?            ; Adresă sir format
        par         dw ?            ; Primul parametru
sablon ends
```

Procedura va consta dintr-o buclă de analiză a caracterelor din şirul format, a cărui adresa e gestionată în BX. Astfel, [BX] va reprezenta caracterul curent. Pentru specificatorii de format, se gestionează un contor de parametri variabili, în registrul DI, care va lua valorile 0, 2, 4 etc. (toţi parametrii variabili sunt pe 16 biţi). Accesul la parametrii variabili se va face prin expresia [BP][DI].PAR, deci printr-o adresare bazată şi indexată, cu deplasament. Se tratează coespunzător secvenţele escape (care încep cu \) şi specificatorii de format (care încep cu %), ambele situaţii presupunând citirea unui caracter suplimentar din şirul care descrie formatul. Dacă parametrii nu ar avea toţi aceeşi dimensiune, s-ar utiliza expresii de genul DWORD PTR [BP][DI].PAR, caz în care registrul DI ar fi actualizat cu dimensiunea parametrului (în cazul de faţă, cu 4).

Implementarea este următoarea:

```
printf_proc     proc    near
        push    bp                              ; Salvare BP
        mov     bp,sp
        mov     bx, [bp].format                 ; Adresă şir format
        mov     di, 0                           ; Contor parametri
                                                ; variabili
pr_loop:
        mov     al, [bx]                        ; Caracter curent
        test    al, al                          ; Test sfârşit de şir
```

```
        jnz     aici1                                   ; care descrie formatul
        jmp     pr_end
aici1:
        cmp     al, '%'                                 ; Test spec. format
        jne     pr_char                                 ; Nu este spec.
        inc     bx                                      ; Dacă da, se citeşte
        mov     al, [bx]                                ; caracterul următor
        cmp     al, 's'                                 ; Este %s ?
        je       pr_str                                 ; Da, salt la sfişare
        cmp     al, 'd'                                 ; Este %d ?
        je      pr_int                                  ; Da, salt la sfişare
        cmp     al, 'u'                                 ; Este %u ?
        je      pr_u                                    ; Da, salt la sfişare
        cmp     al, '%'                                 ; Este %% ?
        je      aici2                                   ;
        jmp     pr_err                                  ; Nu a fost nici unul
                                                        ; din specificatorii
                                                        ; aşteptaţi: se afişează
                                                        ; un mesaj de eroare
aici2:
        putc    al                                      ; Afişare caracter
        inc     bx                                      ; Avans adresă şir
        jmp     pr_loop                                 ; Reluare
pr_char:
        cmp     al, '\'                                 ; Este secvenţă escape ?
        je      pr_esc                                  ; Da, salt la tratare
        putc    al                                      ; Dacă nu, e vorba de
                                                        ; un car. obişnuit, care
        inc     bx                                      ; se afişează şi se reia
        jmp     pr_loop                                 ; bucla de parcurgere
pr_esc:
        inc     bx                                      ; Tratare secvenţă escape
        mov     al, [bx]                                ; Luăm următorul caracter
        cmp     al, 'n'                                 ; Este \n ?
        jne     esc1                                    ; Nu, salt
        putc    cr                                      ; Da, se afişează
        putc    lf                                      ; CR LF şi se reia
        inc     bx                                      ; bucla de
        jmp     pr_loop                                 ; parcurgere
esc1:
        cmp     al, '\'                                 ; Este \\ ?
        jne     pr_err                                  ; Nu, secvenţă escape
                                                        ; greşită
        putc    al                                      ; Da, afişează \
        inc     bx                                      ; şi reia bucla
        jmp     pr_loop                                 ; de prelucrare

        ;
        ; Afişarea parametrilor variabili
        ;
pr_str:
        push    bx                                      ; Afişare şir. Preluăm
        mov     bx, [bp+di].par                         ; adresa din stivă şi
        puts    [bx]                                    ; şi afişăm şirul (cu
        pop     bx                                      ; salvare/restaurare BX
pr_next:
        inc     bx                                      ; Car. următor în format
        add     di, 2                                   ; Contor par. variabili
        jmp     pr_loop                                 ; Reluare
```

```
pr_int:
        puti    [bp+di].par                             ; Întreg cu semn
        jmp     pr_next                                 ; Reluare
pr_u:
        putu    [bp+di].par                             ; Întreg fără semn
        jmp     pr_next                                 ; Reluare
pr_err:
                                                        ; Mesaj de eroare
        putsi   <cr, lf, 'printf: Eroare format', cr, lf>
pr_end:
        pop     bp                                      ; Refacere BP
        ret                                             ; Revenire
printf_proc     endp
```

Ne propunem acum să dezvoltăm o macroinstrucţiune de apel, numită printf, care să se apropie cât mai mult de un limbaj de nivel înalt, deci să putem scrie în textul sursă ceva de genul:

```
        printf <"Numerele m n şi p sunt: %d %d %d\n">, <m, n, p>
```

Macroinstrucţiunea va avea deci un prim parametru (şirul care descrie formatul) şi o listă de parametri variabili (care poate chiar lipsi):

```
        printf macro format, param
```

Scrierea unei astfel de macroinstrucţiuni ridică probleme deosebite, dar pe care operatorii şi directivele limbajului de asamblare le pot rezolva.

Şirul care descrie formatul se defineşte în segmentul de date, printr-o tehnică similară celei de la macroinstrucţiunea putsi, descrisă în Capitolul 5.

Problema cea mai dificilă este că parametrii variabili trebuie puşi în stivă în ordine inversă decât apar în lista din apelul macroinstrucţiunii. Lista poate fi parcursă cu instrucţiunea IRP, dar numai de la stânga la dreapta.

Pentru a pune parametrii în ordinea dorită, se va parcurge lista lor de două ori. La prima parcurgere, se contorizează numărul de parametri (cu operatorul de atribuire =) şi se creează spaţiu în stivă, prin secvenţa:

```
        N = 0
        irp     x, <param>
                sub  sp, 2
                N = N + 1
        endm
```

La a doua parcurgere, se plasează parametrii în stivă. Problema este că indicatorul SP trebui să fie incrementat cu 2, deci nu se poate utiliza o simplă instrucţiune PUSH. Se execută deci secvenţa:

```
        irp     x, <param>
                add     sp, 2
                push    x
                add     sp, 2
        endm
```

care pune parmetrii x în stivă, de sus în jos (vezi Figura 6.11).

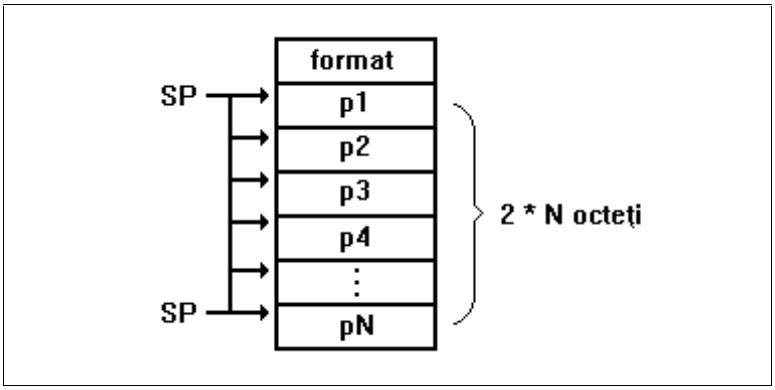

Figura 6.11 Plasarea parametrilor în stivă la macroinstrucţiunea printf

Toată această secvenţă se execută numai dacă lista param este nevidă. După depunerea parametrilor variabili, se decrementează SP cu 2*N (deci se aduce SP pe primul parametru), se depunde în stivă adresa formatului şi se apelează procedura printf_proc. Contorizarea parametrilor în constanta simbolică N asigură şi o descărcare comodă a stivei, prin adunarea valorii 2*(N+1). Dacă şirul format lipseşte, se forţează o eroare de asamblare.

Implementarea completă a macroinstrucţiunii este următoarea:

```
        printf macro format, param
        local loc_for
        ifb     <format>
                %out : Lipsa format....
                .err
                exitm
        endif
        .data
        loc_for         db      format          ;; Definire şir format
                        db      0
        .code
        N = 0                                   ;; Contor par. variabili
        ifnb    <param>                         ;; Dacă există par. var.
                irp     x, <param>
                        sub     sp, 2           ;; Creare spaţiu
                        N = N + 1               ;; Contor
                endm

                irp     x, <param>
                        add     sp, 2           ;; În jos
                        push    x               ;; În sus
                        add     sp,2            ;; În jos
                endm
                sub     sp, 2*N                 ;; SP pe primul par. var.
        endif
        lea     ax, loc_for                     ;; Adresă format
        push    ax                              ;; Depunere în stivă
        call    printf_proc                     ;; Apel
        add     sp, 2*(N+1)                     ;; Descărcare stivă
endm
```

Să considerăm un exemplu de apel, în care avem definite datele:

```
.data
        n1      dw      1234
        n2      dw      -3
        n3      dw      4321
```

```
sir1    db      "abcdef",0
sir2    db      "1234567890",0
sir3    db      "Acesta e un sir...",0
s1      dw      sir1
s2      dw      sir2
s3      dw      sir3
```

Secvenţa de test (programul principal) este următoarea:

```
.code
start:
        init_ds_es
        printf  <"Numarul 1: %d\n">, <n1>
        printf  <"Numerele 2 si 3: %d %d">, <n2,n3>
        printf  <"\nSirul 1: %s %% \\ %%">, <s1>
        printf  <"\nSirul 2: %s\nSirul 3: %s">, <s2,s3>
        printf  <"\nDoar format (fara descriptori) \n">
        printf  <"n2 cu semn = %d, n2 fara semn = %u\n">, <n2,n2>
        exit_dos
end start
```

Se observă necesitatea definirii unor variabile pointer (s2, s2, s3) care conţin adresele şirurilor de caractere sir1, sir2 şi sir3. Rularea acestui exemplu produce ieşirea la consolă:

```
Numarul 1: 1234
Numerele 2 si 3: -3 4321
Sirul 1: abcdef % \ %
Sirul 2: 1234567890
Sirul 3: Acesta e un sir...
Doar format (fara descriptori)
n2 cu semn = -3, n2 fara semn = 65533
```

Mai trebuie observat că tehnica de transfer a parametrilor variabili, expusă mai sus, asigură şi o bună protecţie a controlului programului în situaţii de eroare, adică atunci când descriptorii de format nu coincid cu parametrii, atât ca număr cât şi ca tip. Să considerăm, de exemplu, apelurile:

```
printf <"Ceva gresit %d %d %d\n">, <n1, n2>
printf <"Ceva si mai gresit %d %d\n">, <n1, s2, s3>
```

În primul caz, formatul precizează că în stivă sunt trei parametri de tip întreg, dar la apel nu se transmit decât doi. Procedura va afişă un al treilea parametru fictiv dar, deoarece stiva este descărcată de programul apelant, revenirea se face corect şi programul nu se distruge. În al doilea caz, primul parametru se afişează corect, al doilea (o adresă de şir) este afişat incorect ca întreg, iar al treilea este pur şi simplu ignorat. Şi în acest caz, revenirea în programul apelant şi continuarea sa nu sunt compromise.

6.9 Tehnici avansate de programare

În subcapitolele anterioare au fost descrise elementele de bază ale programării în ASM (acţiunile principale din programarea structurată, proceduri, definiţii de date etc.). Cu aceste elemente de bază, se pot aplica orice tehnici de programare cunoscute de la limbaje de nivel înalt. Vom exemplifica o asemenea tehnică, anume cea a automatelor de stare.

Considerăm problema elementară a numărării cuvintelor dintr-un şir de caractere. Se consideră o serie de caractere speciale, numite delimitatori (în

cazul de faţă: spaţiu, punct, virgulă, semn de întrebare, semn de exclamare şi cratimă. Evident, se poate specifica orice fel de delimitatori.

Incrementarea numărului de cuvinte nu depinde numai de caraterul curent din şir. De exemplu, un caracter care nu este delimitator va conduce la incrementarea numărului de cuvinte numai în cazul în care caracterul anterior a fost un delimitator. Acest tip de decizie, care se bazează pe o informaţie anterioară, sugerează utilizarea unei variabile de stare, care să memoreze starea curentă a prelucrării.

În cazul de faţă, vom avea două stări: în intriorul unui cuvânt şi în exteriorul unui cuvânt. Notăm cele două stări cu IN_W şi OUT_W. Pentru memorarea lor este suficientă o variabilă de stare binară.

Algoritmul se modelează ca un automat, care acceptă intrări şi generează ieşiri. Funcţie de intrări şi de starea curentă, se trece într-o stare următoare, emiţând sau nu o ieşire. În cazul de faţă intrările sunt:

- in_1 = caracterul curent din şir nu este delimitator;
- in_2 = caracterul curent din şir este delimitator.

Ieşirea automatului este:

- out = incrementează numărul de cuvinte

Trebuie acum să codificăm stările, intrările şi ieşirile. Pentru simplitate, vom utiliza registrele procesorului:

- stare = IN_W: CX = 1
- stare = OUT_W: CX = 0
- in_1: BX = 1
- in_2: BX = 0
- out: Incrementează AX.

Descriem acum funcţionarea automatului, prin diagrama din Figura 6.12.

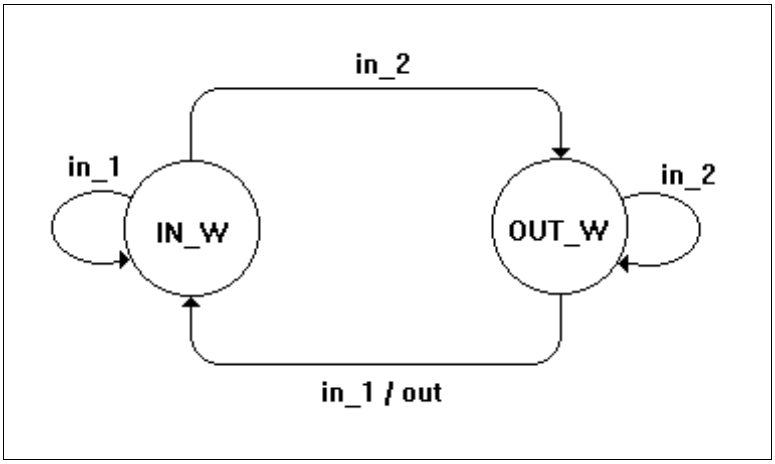

Figura 6.12 Descrierea funcţionării unui automat de stare

Dacă suntem în interiorul unui cuvânt (starea IN_W), atunci un delimitator (intrarea in_2) provoacă trecerea în exteriorul unui cuvânt (starea OUT_W) iar un caracter care nu este delimitator (intrarea in_1) menţine starea IN_W. Dacă suntem în exteriorul unui cuvânt (starea OUT_W), atuci un delimitator (intrarea in_2) menţine starea OUT_W iar un caracter care nu este delimitator (intrarea in_1) provoacă trecerea în starea IN_W şi forţarea semnalului de

ieşire out. Iniţial, automatul se află în starea OUT_W. Se remarcă faptul că, în codificarea aleasă, starea următoare este dictată de codificarea intrării (mai precis, comutarea stării va însemna o simplă instrucţiune MOV CX, BX).

Evoluţia variabilei de stare, a intrărilor in_1 şi in_2 precum şi a ieşirii out în cazul prelucrării şirului de caractere "Un text oarecare." sunt ilustrate în Figura 6.13.

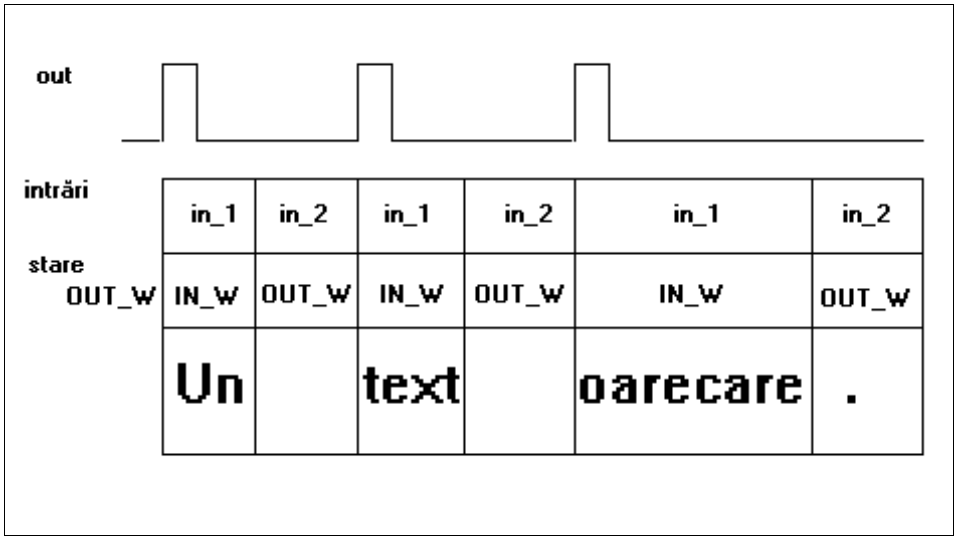

Figura 6.13 Evoluţia automatului de stare în cazul şirului "Un text oarecare."

Implementarea algoritmului urmăreşte îndeaproape funcţionarea automatului şi constă în principal din:

- evaluarea intrărilor;
- comutarea stării;
- forţarea ieşirii.

Presupunem că adresa şirului este transmisă procedurii în DS:SI şi că rezultatul este întors în AX. Implementarea este următoarea:

```
        count  proc near
        push   cx                          ; Salvări
        push   bx                          ; registre
        push   si
        mov    cx, 0                       ; Stare iniţială = OUT_W
        mov    ax, cx                      ; Număr de cuvinte = 0
reluare:
        cmp    byte ptr [si], 0            ; Test sfârşit de şir
        je     gata
        cmp    byte ptr [si], ' '          ; Evaluare intrări: Blank
        je     et_1
        cmp    byte ptr [si], tab          ; Tab
        je     et_1
        cmp    byte ptr [si], '.'          ; Punct
        je     et_1
        cmp    byte ptr [si], ','          ; Virgulă
        je     et_1
        cmp    byte ptr [si], '?'          ; Semn de întrebare
        je     et_1
        cmp    byte ptr [si], '!'          ; Semn de exclamare
        je     et_1
        cmp    byte ptr [si], '-'          ; Cratimă
        je     et_1
```

```
        mov     bx, 1                           ; Nu e delimitator
        jmp     et_2
et_1:
        mov     bx, 0                           ; Delimitator
;
;       În acest moment, intrările sunt poziţionate
;       BX = 1 (in_1) sau BX = 0 (in_2)
et_2:
        cmp     cx, 0                           ; Stare = OUT_W ?
        je      et_3
        mov     cx, bx                          ; Stare = intrare
        jmp     et_5
et_3:
        cmp     bx, 1                           ; In starea OUT_W si
        jne     et_4                            ; cu intrarea in_1,
        inc     ax                              ; se genereaza iesirea
                                                ; out (incrementeaza nr.
                                                ; de cuvinte)
et_4:
        mov     cx, bx                          ; Stare = intrare
et_5:
        inc     si                              ; Trecere la
        jmp     reluare                         ; caracterul următor
gata:
        pop     si                              ; Refaceri
        pop     bx                              ; registre
        pop     cx
        ret                                     ; Rezultat în AX
count   endp
```

Un program de test al procedurii count este listat mai jos. Se citesc de la consolă şiruri de caractere (până la introducerea şirului vid), se apelează procedura count şi se afişează numărul de cuvionte din şirul introdus.

```
.model          small
include         io.h
.data
        sir     db      80 dup (0)              ; Spaţiu de lucru
.stack 1024
.code

start:
        init_ds_es
iar:
        putsi   <'Introduceti un sir: '>
        gets    sir                             ; Citire şir
        mov     al, sir                         ; Test şir
        test    al, al                          ; vid
        jz      exit
        putsi   <'Sirul are '>
        lea     si, sir                         ; Apel procedură
        call    count                           ; Nr. cuvinte în AX
        puti    ax                              ; Afişare
        putsi   <' cuvinte', cr, lf>
        jmp     iar                             ; Reluare
exit:
        exit_dos
end     start
```

6.10 Programarea coprocesoarelor matematice

În Capitolul 2 a fost prezentat setul de instrucţiuni al coprocesoarelor matematice 80x87. Coprocesorul recunoaşte variabile întregi şi toate cele trei tipuri de numere reale (precizie simplă, dublă şi extinsă).

În acest subcapitol ne propunem să dezvoltăm un set de proceduri şi macroinstrucţiuni care să rezolve problema conversie din formatul real (în simplă precizie), în format ASCII (şir de caractere), ceea ce va permite apoi şi introducerea şi afişarea de numere reale la consolă.

6.10.1 Conversia ASCII-real

Dezvoltăm o procedură de tip far, cu numele ATOF_PROC, care primeşte în stivă o adresă near a unui şir de caractere şi o adresă near a unui număr real. Şablonul de acces la stivă este:

```
sablon_atof struc
                        dw      ?       ; Loc pentru BP
                        dd      ?       ; Loc pentru adresa de revenire
        adr_num         dw      ?
        adr_sir         dw      ?
sablon_atof ends
```

Procedura de conversie se bazează pe următorul algoritm:

```
        valoare = 0;
        divizor = 1.0
        punct = FALSE;
        minus = FALSE;
        preia caracter din şirul sursă;
        if (carcater == '-' ) {
                minus = TRUE;
                preia următorul caracter din şirul sursă;
        }
        while (caracterul curent este cifră zecimală sau '.') {
                if (caracter == '.')
                        punct = TRUE;
                else {
                        cifra = caracter - '0';
                        cifra_float  = (float) cifra;
                        valoare = 10.0 * valoare + digit_float;
                        if (punct)
                                divizor = divizor * 10.0;
                }
                preia următorul caracter din şir;
        }
        if (punct)
                valoare = valoare/divizor;
        if (minus)
                valoare = - valoare;
        return valoare;
```

Ideea generală este calculul numărului ca şi când nu ar avea punctul zecimal şi împărţirea apoi la o putere a lui 10, calculată după numărul de zecimale. De exemplu, şirul 123.456 va conduce la calculul valorii 123456, care se împarte apoi la 1000.

Implementarea este următoarea:

```
.data
        _zece           dd      10.0
        _punct          db      ?
        _minus          db      ?
        _cifra          dw      ?
.code
        atof_proc proc far
        push    bp
        mov     bp, sp
        push    bx                              ; Salvări
        push    si                              ; registre

        fld1                                    ; ST ← 1.0
        fldz                                    ; ST ← 0.0, ST(1) ← 1.0
;
;       Asignăm valoare în ST şi divizor în ST(1)
;
        mov     _punct, 0
        mov     _minus, 0
        xor     bh, bh                          ; Necesar la depunere întreg
;
        mov     si, [bp].adr_sir                ; Adresă şir
        cmp     byte ptr [si], '-'              ; Test '-'
        jne     atof_1
        mov     _minus, 1
        inc     si
atof_1:
        mov     bl, [si]                        ; Preia caracter
        cmp     bl, '.'                         ; Test '.'
        jne     atof_2
        mov     _punct, 1
        jmp     atof_3                          ; Salt la reluare
atof_2:
        cmp     bl, '0'                         ; Test
        jb      atof_4                          ; cifră
        cmp     bl, '9'                         ; zecimală
        ja      atof_4
        sub     bl, '0'                         ; Conversie la întreg
        mov     _cifra, bx                      ; Memorăm ca întreg
        fmul    _zece                           ; valoare ← 10*valoare
        fiadd   _cifra                          ; valoare ← valoare +cifra
;
        cmp     _punct, 1                       ; Test punct
        jne     atof_3
        fxch                                    ; Schimbăm ST cu ST(1)
        fmul    _zece                           ; pentru că vrem să înmulţim
        fxch                                    ; divizorul cu zece, apoi
                                                ; schimbăm la loc
atof_3:
        inc     si                              ; Reluare
        jmp     atof_1                          ; buclă
atof_4:
        fdivr                                   ; Împărţire ST la ST(1) cu
                                                ; descărcarea stivei; Rezultat
                                                ; în ST
        cmp     _minus, 1                       ; Test semn
        jne     atof_5
        fchs                                    ; Schimbare semn
atof_5:
```

```
        mov     bx, [bp].adr_num                ; Adresă număr real
        fstp    dword ptr [bx]                  ; Depunere ST cu descărcarea
        fwait                                   ; stivei, apoi FWAIT

        pop     si                              ; Refaceri
        pop     bx                              ; registre
        pop     bp
        retf                                    ; Revenire
atof_proc endp
```

Se observă că stiva coprocesorului este lăsată în starea în care era la intrarea în procedură, lucru care trebuie avut în vedere la toate programele care lucrează cu 80x87. Într-o implementare mai pretenţioasă, s-ar salva toate cele 8 registre ale coprocesorului ca variabile de tip TBytes, într-o zonă proprie de memorie şi s-ar reface la revenirea în programul apelant.

De asemenea, bucla de citire caractere ar trebui să testeze dacă s-a întâlnit deja punctul zecimal. Pentru simplitate, s-a presupus aprioric că şirul sursă conţine cel mult un punct zecimal.

Scriem acum o macroinstrucţiune de apel, cu numele atof, care are ca parametri şirul de caractere sursă şi variabila reală în care se va depune rezultatul. Ambii parametri sunt folosiţi în instrucţiuni LEA, care preiau adresele near şi le transmit către procedura atof_proc.

```
atof    macro   a_sir, a_num
        push    si
        lea     si, a_sir
        push    si
        lea     si, a_num
        push    si
        call    far ptr atof_proc
        add     sp, 4
        pop     si
endm
```

Dispunând de macroinstrucţiunea atof, putem scrie o macroinstrucţiune cu numele getf, care să citească un număr real de la consolă. Operaţia se compune dintr-o citire de şir de caractere şi o conversie ASCII-real. Parametrul macroinstrucţiuni este variabila dword în care se depune numărul real citit.

```
getf    macro   a_num
local sir
        .data
                sir     db 80 dup (0)
        .code
                gets    sir
                atof    sir, a_num
endm
```

6.10.2 Compararea numerelor reale

O problemă importantă care se pune este compararea a două numere reale. Procesorul dispune de instrucţiuni de comparare, în urma cărora se poziţionează o serie de flaguri interne din cuvântul de stare (Status Word). Acest cuvânt poate fi depus în memorie cu instrucţiuni FSTSW şi citit apoi prin instrucţiuni 80x86. Preferăm totuşi o altă abordare, mai simplă, anume efectuarea diferenţei celor doi operanzi şi poziţionarea corespunzătoare a bistabililor ZF şi CF din 80x86. Astfel, se vor putea utiliza instrucţiuni de slat

condiţionat specifice 80x86 (JE, JB, JGE etc.). Testarea semnului diferenţei se face prin exeminarea fizică a bitului de semn din reprezentarea internă a numerelor în simplă precizie. Testarea cazului de egalitate se face testând dacă toţi cei patru octeţi ai diferenţei sunt nuli.

Dezvoltăm pentru început o primă variantă a unei macroinstrucţiuni cu numele F_COMP, având un singur un operand, care compară numărul real din vârful stivei coprocesorului cu operandul specificat.

```
f_comp macro val
        local   temp, et_gt, et_lt, et_eq, gata
.data
        temp    dd ?                        ; Spaţiu de lucru
.code
        push    ax                          ; Salvare AX
        fld     st                          ; Se face o copie a vârfului stivei,
                                            ; pentru a nu altera ST iniţial
        fsub    dword ptr val               ; ST ← ST - val
        fstp    dword ptr temp              ; Depunem diferenţa
        mov     al, byte ptr temp+3         ; Luăm ultimul octet din reprezentare
        and     al, 10000000B               ; Filtru bit de semn
        jnz     et_lt                       ; Diferenţă negativă ?
        mov     ax, word ptr temp           ; Nu, testăm dacă nu este zero
        or      ax, word ptr temp+2         ; Zero real are toţi cei 4 octeţi nuli
        jz      et_eq                       ; Este zero ?
et_gt:                                      ; Nu, înseamnă că e mai mare
        mov     ax, 2                       ; Facem o comparaţie pentru a
        cmp     ax, 1                       ; poziţiona indicatorii
        jmp     gata
et_lt:
        mov     ax, 1                       ; Cazul mai mic
        cmp     ax, 2
        jmp     gata
et_eq:
        mov     ax, 1                       ; Cazul egal
        cmp     ax, 1
        jmp     gata
gata:
        pop     ax                          ; Refacere AX şi gata
endm
```

În această variantă, se efectuează o scădere temporară a celor doi operanzi şi apoi se poziţionează flagurile procesorului de bază, exact ca la o instrucţiune de comparaţie 80x86. Pentru claritate, s-au forţat comparaţii între valorile întregi 1 şi 2.

O a doua variantă a acestei macroinstrucţiuni se bazează pe instrucţiunea FSTSW a coprocesoarelor încorporate în procesoarele 80486 şi cele ulterioare. Această instrucţiune poate salva registrul de stare 80x87 (de 16 biţi) în registrul AX al procesorului de bază. Este prima instrucţiune în care cele două procesoare comunică direct, şi nu prin intermediul memoriei. În plus, cei 3 biţii care raportează rezultatul comparaţiei în registrul de stare 80x87 (din cei mai semnificativi 8 biţi ai registrului de stare) corespund cu flagurile CF, ZF şi PF din registrul de flaguri. Astfel, o instrucţiune SAHF va poziţiona flagurile CF şî ZF cu aceeaşi semnificaţie ca la comparaţiile de întregi. Flagul PF setat va corespunde unor operanzi în virgulă mobilă necomparabili.

Concret, asta înseamnă că putem folosi salturi condiţionate ce conţin "below" şi "above" (JB, JBE, JA, JAE etc.)

De remarcat că noile procesoare Intel (Pentium Pro şi ulterioare) dispun şi de alte mecanisme de testare a rezultatului comparaţiei a două valori reale. A se vedea secţiunile 7.3.3 şi 7.5.6 din "Intel Architecture – Software Developers Manual", vol. I - "Basic Architecture".

A doua implementare a macroinstrucţiunii F_COMP este următoarea:

```
f_comp macro val
local temp
.data
        temp   dw ?                          ; Spaţiu de lucru
.code
        push   ax
        fcom   dword ptr val
        fstsw  word ptr temp
;
;       fstsw  ax                            ; Numai de la 80486 în sus!
;
        fwait
        lahf
        mov    ax, word ptr temp
        sahf
        pop    ax                            ; Putem folosi JB, JBE, JA, JAE şi JE
endm
```

6.10.3 Conversia real-ASCII

Dezvoltăm acum procedura ftoa_proc, care primeşte în stivă adresa near a unui număr real şi adresa near a unui şir de caractere. Procedura converteşte numărul real într-un şir de caractere care conţine reprezentarea valorii reale în format ştiinţific:

```
        x.xxxxxxE±yy
```

Accesul la parametrii din stivă se face prin structura şablon:

```
sablon_ftoa struc
                dw      ?
                dd      ?
        val     dd      ?
        siradr  dw      ?
sablon_ftoa end
```

Implementarea foloseşte o serie de constante şi variabile de lucru, definite mai jos. Algoritmul de conversie este următorul:

```
        ;
        ;       val = Valoarea reală care se converteşte
        ;       sir = Adresa şirului destinaţie
        ;
        if (val < 0.0) {
                *s++ = '-';
                val = -val;
        }
        _exp = 0;
        if (val != 0.0) {
                if (val >= 0.0) {
```

```
                do {
                        val = val / 10.0;
                        _exp = _exp + 1;
                } while (value >= 10.0);
        }
        else {
                while (val < 1.0) {
                        val = val * 10.0;
                        _exp = _exp - 1;
                }
        }
        val = val + 0.0000005;
        if (value > 10.0) {
                value = value / 10.0;
                _exp = _exp + 1;
        }
}
_cifra = (int) val;                     /* Obligatoriu cu trunchiere, nu cu rotunjire */
*s ++ = _cifra + '0';
*s++ = '.';
for (i = 1; i <= 6; i++) {
        val = 10.0 * (val - (float) _cifra);
        _cifra = (int) val;             /* Obligatoriu cu trunchiere, nu cu rotunjire */
        *s++ = _cifra + '0';
}
*s++ = 'E';
if (_exp < 0) {
        *s++ = '+';
        _exp = - _exp;
}
else
        *s++ = '+';
/* scrie _exp ca două cifre în sir */
*s = 0;                                 /* Terminator de şir */
```

Se aduce numărul în intervalul [1.0, 10.0), prin împărţiri sau înmulţiri succesive cu 10. Înmulţirile sau împărţirile cu 10 sunt contorizate în variabila întregă _exp (exponent zecimal). În acest moment, partea întreagă a numărului are o singură cifră, care se poate extrage şi depune în şir. Se determină partea fracţionară, prin scăderea părţii întregi, se înmulţeşte cu 10 şi se reia generarea cifrelor. Acest proces se repetă de 5 ori, pentru cifrele de după virgulă. În prealabil, se realizează o rotunjire a numărului, prin adunarea valorii reale 0.000005.

Algoritmul depinde esenţial de modul în care se face conversia real-întreg. Este obligatoriu ca această conversie să se realizeze prin trunchiere, nu prin rotunjire. De exemplu, valoarea 2.999 trebuie să genereze prima cifră 2 şi nu 3.

Coprocesorul poate opera în mai multe moduri de rotunjire/trunchiere. Aceste moduri sunt controlate de biţii 10 şi 11 (câmpul RC sau Round Control), din cuvântul de control (Control Word). Pentru situaţia de faţă, este adecvat modul RC = 1. Procedura trebuie deci să poziţioneze câmpul RC din cuvântul de control, printr-o salvare/încărcare în memorie. La revenirea în programul apelant, se va reface câmpul RC original. A se vedea secţiunea 7.3.4.3 din "Intel Architecture – Software Developers Manual", vol. I - "Basic Architecture".

Procedura necesită o serie de constante reale şi variabile de manevră, definite mai jos. Numele variabilelor coincid cu cele de la descrierea algoritmului. Evident, vom aloca variabila reală în vârful stivei 80x87. Exponentul este gestionat ca număr întreg.

```
.data
        _zece  dd      10.0
        _unu   dd      1.0
        _zero  dd      0.0
        _rounddd       0.0000005
        _exp   dw      ?
        _cifra dw      ?
        _temp dd       ?
        _cw    dw      ?
```

Implementarea propriu-zisă este următoarea:

```
.code
ftoa_proc proc far
        push            bp
        mov             bp, sp

        push            ax                              ; Salvare
        push            bx                              ; registre
        push            si
        push            cx

        fstcw           _cw                             ; Salvare Control Word
        mov             ax, _cw
        and             ax, NOT 0000110000000000B       ; Filtru biţii 10 şi 11
        or              ax, 0000010000000000B           ; Forţare RC = 1
        mov             _cw, ax                         ; Înapoi în memorie
        fldcw           _cw                             ; Înapoi în 80x87

        mov             _exp, 0                         ; Exponent
        mov             si, [bp].siradr                 ; Adresă şir de caractere
        fld             dword ptr [bp].val              ; Valoare reală în ST
        f_comp          _zero
        ja              ftoa_poz                        ; Este > 0 ?
        mov             byte ptr [si], '-'              ; Nu, depunem '-'
        inc             si                              ; în şir şi schimbăm
        fchs                                            ; semnul numărului real
ftoa_poz:
        f_comp          _zero
        jne             ftoa_aici                       ; Este diferit de 0.0 ?
        jmp             ftoa_1
ftoa_aici:                                              ; Da, îl convertim
        f_comp          _zece
        jnge            ftoa_2                          ; Este < 10 ?
ftoa_3:
        fdiv            _zece                           ; Nu, îl împărţim la 10
        add             _exp, 1                         ; şi ţinem minte la exponent
        f_comp          _zece                           ; până când ajunge
        jnl             ftoa_3                          ; mai mic strict decât 10
ftoa_2:
        f_comp          _unu                            ; Este mai mare sau egal ca 1 ?
        jge             ftoa_1                          ; Da, salt
        fmul            _zece                           ; Nu, îl înmulţim cu 10 şi ţinem minte
        sub             _exp, 1                         ; la exponent, până ajunge
        jmp             ftoa_2                          ; mai mare sau egal ca 1
```

```
ftoa_1:
        fadd        _round                      ; Rotunjim la 7 zecimale
        f_comp      _zece                       ; Poate a depăşit acum 1o ?
        jna         ftoa_4
        fdiv        _zece                       ; Dacă da, corectăm
        add         _exp, 1
ftoa_4:
        fist        _cifra                      ; Generăm partea întreagă
        mov         bl, byte ptr _cifra         ; Luăm cifra propriu-zisă,
        add         bl, '0'                     ; o convertim la caracter ASCII
        mov         [si], bl                    ; şi o depunem în şir
        inc         si
        mov         byte ptr [si], '.'          ; Apoi punctul zecimal
        inc         si

        mov         cx, 6                       ; Buclă de 6 cifre după punct
ftoa_5:
        fisub       _cifra                      ; ST = partea fracţionară
        fmul        _zece                       ; ST = ST * 10
        fist        _cifra                      ; Partea întreagă
        mov         bl, byte ptr _cifra         ; Determinare cifră
        add         bl, '0'
        mov         [si], bl
        inc         si
        loop        ftoa_5

        fstp        _temp                       ; Descărcăm stiva 8087
        mov         byte ptr [si], 'E'          ; Notaţie ştiinţifică
        inc         si
        cmp         _exp, 0                     ; Test semn exponent
        jge         ftoa_6
        mov         byte ptr [si], '-'          ; Negativ
        neg         _exp                        ; Schimbăm semnul
        jmp         ftoa_7
ftoa_6:
        mov         byte ptr [si], '+'          ; Pozitiv
ftoa_7:
        inc         si
        mov         ax, _exp                    ; Exponent
        aam                                     ; AH = _exp / 10;
                                                ; AL = _exp MOD 10
        or          ax, '00'                    ; Ambele cifre convertite la ASCII
        mov         [si], ah                    ; Depunere în şir
        inc         si
        mov         [si], al
        inc         si
        mov         byte ptr [si], 0            ; În fine, terminatorul de şir

        pop         cx                          ; Refacere
        pop         si                          ; registre
        pop         bx
        pop         ax

        pop         bp
        retf                                    ; Revenire
ftoa_proc endp
```

Se observă utilitatea foarte mare a macroinstrucţiunii f_comp, care face ca implementarea algoritmului să fie o simplă transpunere a descrierii de tip pseudo-cod.

Dezvoltăm acum o macroinstrucţiune de apel cu numele ftoa şi cu parametri adecvaţi (valoarea reală şi şîrul destinaţie):

```
ftoa    macro   val, sir
        push    ax
        lea     ax, sir
        push    ax
        mov     ax, word ptr val + 2
        push    ax
        mov     ax, word ptr val
        push    ax
        call    far ptr ftoa_proc
        add     sp, 6
        pop     ax
endm
```

Dispunând de macroinstrucţiunea ftoa, putem dezvolta o macroinstrucţiune putf, care să afişeze la consolă un număr real, prin conversie real-ASCII şi afişare şir:

```
putf    macro   val
        local   sir
        .data
                sir     db 30 dup (0)
        .code
                ftoa    val, sir
                puts    sir
endm
```

6.10.4 Un program de test

Dacă presupunem macroinstrucţiunile de mai sus definite într-un fişier float.h, putem scrie un mic program de test, care citeşte un număr real de la consolă şi afişează valorile incremenate de 10 ori.

```
        .model          large
        .8087
        include         io.h
        include         float.h
        .stack          1024
        .data
                numar           dd      ?
                unu             dd      1.0
        .code
        start:
                init_ds_es
                finit
                putsi           <'Introduceti un numar real: '>
                getf            numar
                putsi           <'Valorile incrementate sunt:', cr, lf>
                mov             cx, 10
        iar:
                putf            numar
                putsi           <cr, lf>
                fld             numar
                fadd            unu
                fstp            numar
```

```
            loop        iar
            exit_dos
      end   start
```

Se observă instrucţiunea FINIT, care iniţializează coprocesorul matematic. Directiva .8087 instruieşte asamblorul să accepte mnemonice specifice coprocesorului 8087. Funcţie de tipul coprocesorului şi al procesorului de bază, se pot utiliza directivele .287, .387 sau .486.

Procedurile de mai sus operează cu numere în simplă precizie. Trecere la precizie dublă şi extinsă se face însă foarte uşor. Trebuie doar ca variabilele externe şî constantele reale utilizate să fie declarate de tip qword (8 octeţi), respectiv tbytes (10 octeţi), deoerece formatul intern al procesorului este cel de precizie maximă.

Deşi procedurile care implică operaţii cu numere rele par complicate, implementările din acest subcapitol arată că o abordare sistematică şi disciplinată permite dezvoltarea unui cod sursă clar, uşor de urmărit şi uşor de întreţinut.