

Capitolul 4

Structurarea programelor. Definirea și inițializarea datelor. Operatori

Capitolul curent și cel următor sunt dedicate limbajului de asamblare. Pe lângă instrucțiunile propriu-zise, un program ASM poate cuprinde directive, prin intermediul cărora se definesc date, etichete și proceduri, se structurează segmente, se definesc și se utilizează macroinstrucțiuni, se controlează în general procesul de asamblare etc. Directivele nu reprezintă instrucțiuni ci comenzi către asamblor, efectul lor manifestându-se exclusiv în faza de asamblare.

4.1 Segmentare. Directive pentru definirea segmentelor

Un modul de program în limbajul de asamblare poate conduce la:

- o porțiune dintr-un segment;
- un segment;
- porțiuni de segmente diferite;
- mai multe segmente;

Mai multe module obiect se leagă prin operația de link-editare.

4.1.1 Directivele **SEGMENT** și **ENDS**

Indiferent de modul de dezvoltare al unui program ASM, atât instrucțiunile cât și datele trebuie să se găsească în interiorul unui segment. Directiva **SEGMENT** controlează:

- numele segmentului;
- alinierea;
- combinarea cu alte segmente;
- continuitatea (adiacența) segmentelor.

Forma generală a directivei **SEGMENT** este:

```

nume SEGMENT [tip_aliniere] [tip_combinare] ['nume_clasă']
    .
    .
    .
nume ENDS

```

Parametrii incluși în paranteze drepte sunt opționali. Dacă sunt prezenți, aceștia trebuie să fie în ordinea specificată. Semnificația parametrilor este:

- **tip_aliniere** - specifică la ce limită va fi relocat segmentul în memorie; poate fi:
 - ♦ **PARA** (implicit) - aliniere la paragraf (segmentul fizic, adică adresa pe 20 de biți, va fi relocat la prima adresă absolută divizibilă prin 16);
 - ♦ **BYTE** - fără aliniere (segmentul se va reloca la următorul octet liber);
 - ♦ **WORD** - aliniere la cuvânt (segmentul se va reloca la următorul octet liber aflat la adresă pară);
 - ♦ **DWORD** - aliniere la dublu-cuvânt (segmentul se va reloca la prima adresă divizibilă cu 4);
 - ♦ **PAGE** - aliniere la pagină (segmentul se va reloca la prima adresă absolută divizibilă prin 256).

Să considerăm următoarele definiții de segmente:

```

DAT1 SEGMENT BYTE
    x1      db 7 dup (?)
DAT1 ENDS
DAT2 SEGMENT WORD
    x21     dw 300 dup (?)
    x22     dw ?
DAT2 ENDS
DAT3 SEGMENT PARA
    x3      db 5 dup (?)
DAT3 ENDS
    
```

Dacă prima adresă disponibilă este 1000H, cele trei segmente vor fi reloocate la următoarele adrese fizice:

```

DAT1      1000:0 ÷ 1000:6
DAT2      1000:8 ÷ 1000:261
DAT3      1027:0 ÷ 1027:4
    
```

Adresele de segment se obțin prin trunchierea la primele patru cifre ale adresei fizice de început, iar offseturile se calculează corespunzător. Offset-urile la execuție diferă în general de offseturile la asamblare. Offsetul la asamblare pentru x21 este 0, dar la execuție este 8.

Operatorul OFFSET, care furnizează deplasamentul unei variabile sau etichete în cadrul unui segment, va produce offset-ul de la execuție. Instrucțiunea:

```
mov  bx, OFFSET x21
```

va încărca în BX valoarea 8.

Dacă dorim ca offset-ul la asamblare să coincidă cu offset-ul la execuție, putem folosi tipul de aliniere PARA (ultima cifră a adresei fizice de început a segmentului este 0).

- **tip_combinare** - specifică dacă și cum se va combina segmentul respectiv cu alte segmente, la operația de link-editare; poate fi:
 - ♦ necombinabil (implicit) - nu se scrie nimic;
 - ♦ PUBLIC - segmentul curent va fi concatenat cu alte segmente cu același nume și cu atributul PUBLIC. Aceste segmente pot fi în alte module de program. Se va forma deci un singur segment final cu numele respectiv, cu o unică adresă de început; lungimea unui segment cu atributul PUBLIC este suma lungimilor segmentelor cu același nume.
 - ♦ COMMON -specifică faptul că segmentul curent și toate segmentele cu același nume și cu tipul COMMON se vor suprapune în memorie (vor începe la aceeași adresă fizică); lungimea unui segment COMMON este cea mai mare lungime a segmentelor componente;
 - ♦ STACK - marchează segmentul curent ca reprezentând stiva programului; dacă sunt mai multe segmente cu tipul STACK, ele vor fi tratate ca la PUBLIC; următorul exemplu este o definire și o inițializare explicită a unui segment de stivă:

```

stiva  SEGMENT STACK
        db 256 dup(?)
        stiva_sp label WORD
stiva  ENDS
cod    SEGMENT
        mov ax, stiva
        mov ss, ax
        mov sp, OFFSET stiva_sp
    
```

```
cod    ENDS
```

- ♦ AT EXPRESIE - specifică faptul că segmentul va fi plasat la o adresă fizică absolută de memorie; următorul exemplu ilustrează definirea explicită a tabeli de vectori de întreruperi:

```
INT_TABLE SEGMENT AT 0
    dd PROC_NIV_0
    dd PROC_NIV_1
    .
    .
INT_TABLE ENDS
```

Aceste forme se folosesc când programul este dezvoltat pentru echipamente dedicate. În cazul unui calculator de tip IBM-PC, există funcții DOS pentru accesul la tabela de întreruperi.

- 'nume_clasă': specifică un nume de clasă pentru segment, extinzînd astfel numele segmentului; de exemplu, dacă segmentul are și atributul 'nume_clasă', atunci attributele COMMON sau PUBLIC vor acționa numai asupra segmentelor cu același nume și același nume de clasă; ca nume de clase, se folosesc de obicei 'code', 'data' și 'stack'.

Să considerăm două module ASM, asamblate distinct. În primul modul avem următoarele definiții de segmente:

```
dseg1 SEGMENT PARA PUBLIC 'data'
    db    100 dup (?)
dseg1 ENDS
dseg2 SEGMENT PARA COMMON
    db 50 dup (?)
dseg2 ENDS
cseg  SEGMENT PARA PUBLIC 'code'
    db    20 dup (?)
cseg  ENDS
```

În al doilea modul, există următoarele definiții:

```
dseg1 SEGMENT PARA PUBLIC 'data'
    db 30 dup (?)
dseg1 ENDS
dseg2 SEGMENT PARA COMMON
    db    200 dup (?)
dseg2 ENDS
cseg SEGMENT PARA PUBLIC 'code'
    db    70 dup (?)
cseg ENDS
```

Imaginea segmentelor rezultate în urma link-editării și relocării este cea din Figura 4.1

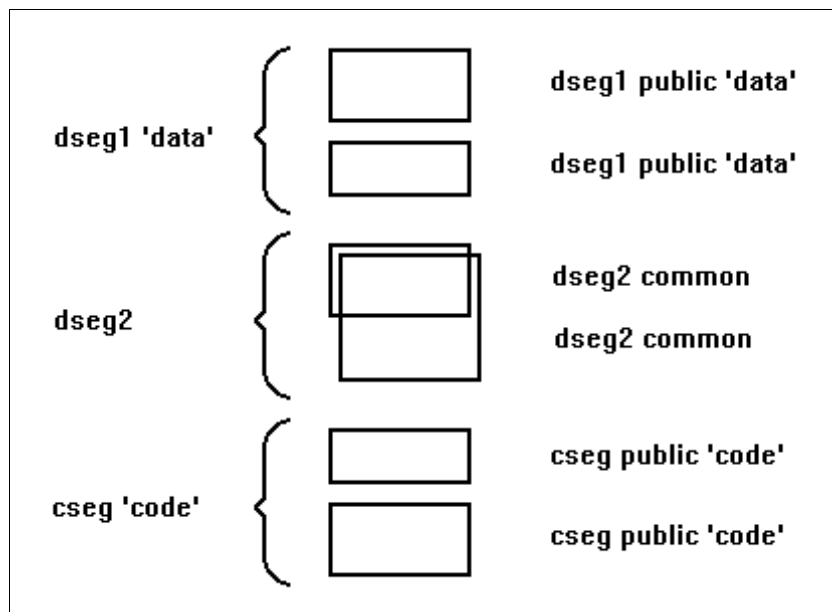


Figura 4.1 Combinarea segmentelor este controlată de directiva SEGMENT

4.1.2 Directiva ASSUME

Directiva ASSUME realizează o conexiune simbolică (logică) între definirea instrucțiunilor și datelor în segmente logice (cuprinse între SEGMENT și ENDS) la momentul asamblării și accesul, la execuție, la instrucțiunii și date prin registrele de segment. Are forma generală:

```
ASSUME reg_seg:expresie, ...
```

În care reg_seg este un registru de segment, iar expresie poate fi;

- ♦ un nume de segment;
- ♦ un nume de grup de segmente;
- ♦ SEG nume segment;
- ♦ NOTHING;

În aceeași directivă ASSUME se pot asocia mai multe registre de segment, de exemplu:

```
ASSUME CS:CSEG, DS:DSEG1, ES:DSEG2, SS:SSEG
```

Directiva ASSUME nu ne scutește de încărcarea registrelor de segment cu adresele de segment. Dacă un nume de segment nu apare într-o directivă ASSUME, datele definite în acel segment pot fi accesate numai prin prefixe de segment. Să considerăm următorul exemplu:

```
dseg  SEGMENT
      BETA  dw 5
dseg  ENDS
cseg  SEGMENT
      ASSUME cs:cseg, es:dseg
      mov  ax, dseg
      mov  es, ax
      mov  bx, BETA      ; se va genera o instrucțiune MOV,
                        ; cu adresare prin registrul ES, conform
                        ; asocierii ES:dseg
cseg  ENDS
```

Dacă nu s-ar fi asociat registrul ES cu segmentul dseg prin directiva ASSUME, accesul la variabila BETA trebuia scris în forma:

```
mov bx, es:BETA
```

În formatul intern al instrucțiunii (în codificare), trebuie să apară prin ce registru de segment se calculează adresa fizică. Dacă simbolul care apare într-o instrucțiune (de exemplu, BETA), face parte dintr-un segment care este asociat, printr-o directivă ASSUME, cu un registru de segment, atunci se va codifica intern accesul prin acel registru de segment (în cazul de față, prin ES). Codificarea registrului segment se face:

- ♦ implicit - datele sunt accesate în toate modurile de adresare prin registrul DS, cu excepția celor care implică registrul BP, caz în care se folosește SS, iar instrucțiunile sunt adresate totdeauna prin CS;
- ♦ explicit - prin utilizarea prefixelor de segment.

4.1.3 Directiva GROUP

Directiva GROUP servește la gruparea mai multor segmente (inclusiv cu nume și atribute diferite), a căror lungime totală nu depășește 64 KO, sub același nume. Are forma generală:

```
nume_grup GROUP nume_seg1, nume_seg2, ...
```

Această directivă reprezintă o altă modalitate de combinare a mai multor segmente, pe lângă cea oferită de atributul PUBLIC.

Numele de grup se poate folosi în directive ASSUME, la inițializarea unor registre de segment sau la calculul unui offset. Să considerăm următorul exemplu:

```
d1    SEGMENT
      y      dw    20
d1    ENDS
d2    SEGMENT
      x      db    2
d2    ENDS
data  GROUP  d1, d2
```

Putem scrie instrucțiuni de forma:

```
mov  ax, data
mov  ds, ax
mov  ax, OFFSET x
mov  ax, OFFSET data:x
```

sau directive de forma:

```
ASSUME  ds:data
```

Expresia OFFSET x furnizează offset-ul variabilei x în cadrul segmentului d1, iar expresia OFFSET data:x produce offset-ul variabilei x în cadrul grupului de segmente data.

4.2 Definirea simplificată a segmentelor

Această modalitate de definire este introdusă în variantele recente ale asamblelor. Avantajul major este faptul că se respectă același format (structură a programului obiect) ca și la programele dezvoltate în limbaj de nivel înalt. Concret, dacă utilizăm definirea simplificată, se vor genera segmente cu nume și atribute identice cu cele generate de compilatoarele de limbaje de nivel înalt. Toate directivele simplificate încep cu un punct.

4.2.1 Modele de memorie

Directiva pentru specificarea modelului de memorie are forma generală:

.model tip

În care tip poate fi tiny, small, medium, large sau huge. Semnificația acestor tipuri este:

- **tiny** - toate segmentele (date, cod, stivă) se pot genera într-un spațiu de 64KO și formează un singur grup de segmente. Se folosește la programele de tip COM. Toate salturile, apelurile și definițiile de proceduri sunt implicit de tip NEAR;
- **small** - datele și stiva sunt grupate într-un singur segment iar codul în alt segment. Fiecare din acestea nu trebuie să depășească 64KO. Toate salturile, apelurile și definițiile de proceduri sunt implicit de tip NEAR;
- **medium** - datele și stiva sunt grupate într-un singur segment (cel mult egal cu 64KO), dar codul poate fi în mai multe segmente separate (nu se grupează), deci poate depăși 64KO. Salturile și apelurile sunt implicit tip FAR iar definițiile de proceduri sunt implicit de tip far.
- **compact** - codul generat ocupă cel mult 64KO (se grupează), dar datele și stiva sunt în segmente separate (pot depăși 64KO). Apelurile și salturile sunt implicit de tip NEAR. Se utilizează adrese complete (segment și offset) atunci când se accesează date definite în alte segmente.
- **large** - atât datele cât și codul generat pot depăși 64KO.
- **huge** - este asemănător modelului large, dar structurile de date pot depăși 64KO; se utilizează adrese complete normalizate în care offsetul este redus la minim (în domeniul 0-15), ceea ce face ca o adresă fizică să fie descrisă de o unică pereche (segment, offset).

La modelele compact și large, o structură compactă de date (de tip tablou) nu poate depăși limitele unui segment fizic (64KO); la modelul huge, această restricție dispare.

Folosirea directivelor simplificate prezintă și avantajul că nu mai sunt necesare directive ASSUME: asocierile între segmentele generate și registrele de segment sunt implicite.

Se utilizează următoarea terminologie:

- modele de **date mici**: small, compact;
- modele de **cod mic**: small, medium;
- modele de **date mari**: medium, large, huge;
- modele de **cod mare**: compact, large, huge.

4.2.2 Directive simplificate de definire a segmentelor

Formele generale ale acestor directive sunt:

- .stack dimensiune
- .code [nume]
- .data
- .data? ; date neinițializate în l.n.î
- .fardata [nume] ; segmente de date utilizate prin
- .fardata? [nume] ; adrese complete în l.n.î
- .const ; definire de constante în l.n.î

Dacă parametrul nume lipsește, se atribuie nume implicite segmentelor generate, după cum urmează:

- pentru .code TEXT (la modele de cod mic) sau nume_fișier_sursă_TEXT (la modele de cod mare)
- pentru .data? _BSS
- pentru .data DATA
- pentru .const CONST
- pentru .stack STACK
- pentru .fardata _FAR_DATA
- pentru .fardata? _FAR_BSS

Se generează grupul de segmente DGROUP. Acesta cuprinde:

- toate segmentele, la modelul de date tiny;
- DATA, _BSS, CONST, STACK, în rest.

Se consideră că există o directivă ASSUME implicită (care nu mai trebuie deci scrisă în textul sursă), de forma:

- la modelele small și compact:

```
ASSUME cs:TEXT, ds:DGROUP, ss:DGROUP
```

- la modelele large și huge:

```
ASSUME cs:nume_TEXT, ds:DGROUP, ss:DGROUP  
; nume este numele din directiva .code sau numele fișierului sursă;
```

- la modelul de memorie tiny:

```
ASSUME cs:DGROUP, ds:DGROUP, cs:DGROUP
```

Dacă se folosesc directivele .fardata și/sau .fardata?, atunci segmentele respective trebuie gestionate explicit prin directive ASSUME.

Adresele de început ale segmentelor și grupurilor de segmente definite cu directivele simplificate sunt disponibile prin simbolii globali:

@data, @data?, @fardata, @fardata?, dgroup.

Dacă în același text sursă (model de cod mare) se folosesc mai multe directive .code cu nume diferite, atunci, până la întâlnirea unei alte directive de segment, este ca și cum s-ar fi scris o directivă ASSUME CS: cu numele respectiv.

Segmentele generate au numele de clasă 'STACK', 'CODE', 'DATA', 'BSS', 'FAR_BSS', 'FAR_DATA'. Segmentul CONST are numele de clasă DATA.

Tipurile de combinare sunt:

- PUBLIC (la .code, .data, .data? și .const);
- STACK (la .stack);
- fără combinare (la .fardata și .fardata?).

Tipul de aliniere este PARA (pentru .stack, .fardata și .fardata?) și WORD (pentru celelalte).

Să considerăm fișierul sursă exemplu.asm, cu conținutul de mai jos:

```
.model large  
.data  
.const  
.stack 256  
.data?
```

```
.fardata
.fardata?
.code
end
```

Fișierul listing generat cuprinde (între altele), următoarele informații:

Symbol	Name	Type	Value
??DATE	Text		"06/02/96"
??FILENAME	Text		"exemplu"
??TIME	Text		"15:21:50"
??VERSION	Number		0200
@CODE	Text		EXEMPLU_TEXT
@CODESIZE	Text		1
@CPU	Text		0101H
@CURSEG	Text		EXEMPLU_TEXT
@DATA	Text		DGROUP
@DATASIZE	Text		1
@FARDATA	Text		FAR_DATA
@FARDATA?	Text		FAR_BSS
@FILENAME	Text		EXEMPLU
@MODEL	Text		5
@WORDSIZE	Text		2

Groups & Segments	Bit	Size	Align	Combine	Class
DGROUP Group					
CONST	16	0000	Word	Public	DATA
STACK	16	0100	Para	Stack	STACK
_BSS	16	0000	Word	Public	BSS
_DATA		16	0000	Word	Public DATA
EXEMPLU_TEXT	16	0000	Word	Public	CODE
FAR_BSS	16	0000	Para	none	FAR_BSS
FAR_DATA	16	0000	Para	none	FAR_DATA

Listingul relevă o serie de simbolii predefiniți și valorile asociate. Se remarcă adresele de început ale segmentelor, cel mai important fiind simbolul @DATA, care are aceeași valoare cu adresa grupului DGROUP. Partea a doua a listingului descrie segmentele și grupurile de segmente.

Inițializarea registrelor DS și ES la începutul unui program principal se poate face cu simbolul @DATA sau cu simbolul DGROUP. Macroinstrucțiunea init_ds_es realizează operația:

```
mov ax, dgroup
mov ds, ax
mov es, ax
```

4.3 Directive pentru legarea modulelor

Când programul dezvoltat se compune din mai multe module asamblate separat, este necesară specificarea simbolilor care sunt definiți într-un modul și utilizați în alte module. Acești simbolii sunt nume de variabile, etichete sau nume de proceduri. În mod normal, un simbol este vizibil doar în modulul în care este definit (simbol global). Simbolii care sunt vizibili în mai multe module de program se numesc simbolii globali.

Se folosesc următoarele noțiuni:

- **simboli publici** - sunt simbolii care sunt definiți într-un modul curent de program și folosiți (eventual) și în alte module; acești simbolii se declară ca simbolii publici în modulul în care sunt definiți;
- **simboli externi** - sunt simbolii care sunt folosiți într-unul din modulele în care nu sunt definiți; acești simbolii se declară ca simbolii externi în modulele în care sunt folosiți.

Se observă că un simbol global trebuie declarat ca simbol public în modulul în care este definit și ca simbol extern în modulele în care este folosit, altele decât cel în care este definit.

Declarația unui simbol ca simbol public, respectiv extern se face cu directivele PUBLIC, respectiv EXTRN.

4.3.1 Directiva PUBLIC

Directiva PUBLIC are forma generală:

```
PUBLIC nume, nume, ...
```

Simbolii declarați ca publici pot fi variabile, etichete, nume de proceduri sau constante numerice simbolice.

4.3.2 Directiva EXTRN

Directiva EXTRN are forma generală:

```
EXTRN nume:tip, nume:tip, ...
```

În care tip precizează tipul simbolului. Acesta poate fi:

- BYTE, WORD, DWORD sau QWORD, în cazul în care simbolul este o variabilă;
- NEAR sau FAR, în cazul în care simbolul este o etichetă sau un nume de procedură;
- ABS, în cazul în care simbolul este o constantă numerică simbolică.

Să considerăm două module de program, M1.ASM și M2.ASM:

<pre>; Modul M1. ASM ; .model large extrn var1:word, output:far public var2 .data var2 dw 7 .code start: mov ax, dgroup mov ds, ax add var1, 3 call output end start</pre>	<pre>; Modul M2. ASM ; .model large public var1, output extrn var2: word .data var1 dw 5 .code output proc far add var2, 1 retf output endp end</pre>
---	--

În acest exemplu, accesul la simbolii globali este facilitat de faptul că atât var1 cât și var2 sunt în același segment de date, fiind accesibile prin registrul DS, inițializat cu grupul de segmente DGROUP.

Dacă nu am fi folosit directive simplificate, accesul la var1 și var2 ar fi fost mai complicat, deoarece nu s-ar fi cunoscut segmentul în care acestea sunt definite. În asemenea situații, se folosește operatorul SEG, care produce adresa de segment a simbolului:

```
mov  ax, SEG var1
mov  es, ax
```

```
add    es:var1, 3
```

4.3.2 Directiva END

Această directivă marchează sfârșitul logic al unui modul de program și e obligatorie în toate modulele. Tot ce se găsește în fișierul sursă după această directivă este ignorat la asamblare. Forma generală este:

```
END [punct_de_start]
```

În care punct_de_start este o etichetă sau un nume de procedură care marchează punctul în care se va da controlul după încărcarea programului în memorie. Într-o aplicație compusă din mai multe module și care se constituie într-un program executabil, un singur modul trebuie să aibă parametru în directiva END.

4.4 Contoare de locații și directiva ORG

Contoarele de locații controlează procesul de asamblare, arătând la ce offset în cadrul segmentului curent se vor asambla instrucțiunea sau datele următoare. Un contor de locații poate fi accesat explicit prin simbolul \$.

La prima utilizare a unui nume de segment, contorul de locații este inițializat cu zero. Dacă se revine într-un segment care a mai fost utilizat, contorul de locații revine la ultima valoare utilizată în cadrul acelui segment, ca în exemplu următor:

```
.data
    ; $ = 0
    db    5 dup(?)
    ; $ = 5

.code
    ; $ = 0

.data
    db    7
    ; $ = 5
```

Contoarele de locații sunt utile la calculul automat al unor deplasamente sau dimensiuni. În exemplul următor, se definește un tablou de cuvinte TAB și o variabilă LNG care conține numărul de cuvinte din tablou:

```
TAB    dw 1, 2, 3, 4, 5, 6, 7, 8, 9
LNG    dw ($ - TAB)/2
```

Expresia **\$ - TAB** reprezintă numărul de octeți de la adresa TAB până la adresa curentă. Împărțind acest număr la 2, se deduce numărul de elemente din tablou. Putem acum adăuga sau elimina date din tabloul TAB fără a actualiza lungimea LNG: asamblorul va calcula corect numărul de elemente din tablou.

Alte exemple de utilizare pot fi:

```
JMP    $      ; Ciclu infinit: $ este chiar adresa
           ; instrucțiunii JMP
JMP    $ + 5   ; Salt relativ
```

Directiva **ORG** (**O**ri**G**in - Inițializează contorul de locații) modifică explicit contorul de locații curent, având forma generală:

```
ORG expresie
```

De exemplu:

```
ORG    $ + 5    ; Sare 5 octeți la asamblare
ORG    100H    ; Asamblează la offset-ul absolut 100H
```

4.5 Definirea și inițializarea datelor

Asamblorul recunoaște trei categorii sintactice de bază:

- constante;
- variabile
- etichete (inclusiv nume de proceduri).

Constantele pot fi absolute (numerice) sau simbolice. Constantele simbolice reprezintă nume generice asociate unor valori numerice. Variabilele identifică datele (un spațiu de memorie rezervat), iar etichetele identifică codul (program sau procedură).

Instrucțiuni de tipul MOV, ADD, AND etc. folosesc variabile și constante iar instrucțiunile JMP și CALL folosesc etichete.

Variabilele și etichetele au asociate atribute, cum ar fi segmentul în care sunt definite, offset-ul la care sunt definite în interiorul segmentului etc.

4.5.1 Constante Constante numerice absolute pot fi:

- constante binare - se utilizează sufixul B sau b;
- constante octale - se utilizează sufixele O, Q, o sau q;
- constante zecimale - fără sufix sau cu sufixele D sau d;
- constante hexazecimale - se utilizează sufixele H sau h și prefixul 0 dacă prima cifră este mai mare ca 9; Pentru cifrele peste 9 se utilizează simbolii A...F sau a...f.
- constante ASCII - se scriu unul sau mai multe caractere ASCII între apostroafe sau ghilimele;

Iată exemple de constante absolute: 123, 123D, 10001100B, 177q, 0AAH, 3fh, 'a', "AB".

Constante simbolice se definesc cu directiva EQU, în forma generală:

```
Nume EQU expresie
```

De exemplu, liniile de program

```
CR    EQU 0DH
LF    EQU 0AH
```

definesc constantele simbolice CR și LF, cu valorile 0DH și 0AH. Putem folosi aceste constante simbolice în orice context în care este permisă folosirea unei constante numerice.

4.5.2 Definirea și utilizarea variabilelor

Pentru definirea variabilelor se utilizează directivele DB, DW, DD, DQ sau DT, care au fost introduse în Capitolul 1. Forma generală a unei definiții de date este:

```
nume    directivă    listă_de_valori
```

În care nume este identificatorul asociat definiției respective, iar listă_de_valori este lista valorilor inițiale, care poate cuprinde:

- constante numerice (absolute sau simbolice);
- ?
- o adresă, adică un nume de variabile sau de etichetă; se folosește la DW și DD;
- un șir ASCII;
- operatorul DUP (expresie), în care expresie poate fi:
 - ♦ constantă numerică;
 - ♦ listă de valori;
 - ♦ simbolul ?
 - ♦ operatorul DUP.

Semnificația simbolului ? este locație neinițializată iar cea a operatorului DUP, repetarea de un număr de ori a expresiei din paranteză.

Să considerăm definițiile următoare:

```
var_a db 2 dup (0, 3 dup (1))
var_b db 1, 2, 3, ?, ?, ?
adr_n dw var_a
adr_f dd var_b
```

Prima definiție este echivalentă cu:

```
var_a db 0, 1, 1, 1, 0, 1, 1, 1
```

Atributele datelor definite sunt:

- segment - cel curent;
- offset - cel curent;
- tip - 1, 2, 4, 8, 10, după cum s-a utilizat ca directivă de bază DB, DW, DD, DQ sau DT.

Variabilele pot fi utilizate ca:

- variabile simple - var_a, var_b;
- variabile indexate - var_a[bx], var_b[2], ALFA [si][bx];

În cazul variabilelor indexate, trebuie ținut seama de tipul variabilei de bază. De exemplu, în urma definiției tabloului:

```
B dw 10 dup(?)
```

expresiile indexate corecte din punct de vedere logic, care accesează elementele lui B sunt B[0], B[2] etc. Din punct de vedere sintactic, putem scrie și:

```
mov ax, B [1]
```

dar această formă va încărca în AL partea high a primului cuvânt din tablou și în AH partea low a celui de-al doilea cuvânt din tabloul B. Dacă acest lucru s-a urmărit, perfect !

În cazul accesării variabilelor prin expresii anonime, poate apare necesitatea operatorului PTR. Instrucțiunile:

```
lea bx, B
inc byte ptr [bx][4]
inc word ptr [bx][4]
```

incrementează octetul, respectiv cuvântul de la adresa BX + 4.

4.6 Definirea etichetelor. Directiva LABEL

Etichetele sunt folosite pentru specificarea punctelor țintă la instrucțiuni de salt sau apel sau pentru o specificare alternativă a datelor. În ambele cazuri, numele etichetei devine un nume simbolic asociat adresei curente de memorie. Atributele etichetelor sunt: **segment**, **offset** și **tip**. Până acum am întâlnit două modalități de definire a etichetelor:

- prin nume urmat de caracterul : - se definește o etichetă de tip near;
- prin directiva PROC - numele procedurii este interpretat ca o etichetă cu tipul derivat din tipul procedurii.

O altă posibilitate este directiva LABEL, care are forma generală:

```
nume LABEL tip
```

Dacă ceea ce urmează reprezintă instrucțiuni (cod), tipul etichetei va fi de regulă NEAR sau FAR și eticheta va fi folosită ca punct țintă în instrucțiuni de timp JMP/CALL. Dacă ceea ce urmează reprezintă definiții de date, tipul etichetei va fi de regulă BYTE, WORD, DWORD etc.

Atributele NEAR sau FAR pot apare numai dacă eticheta este definită într-un segment asociat cu registrul CS printr-o directivă ASSUME (explicită sau implicită).

Directiva LABEL este utilă atunci când dorim să accesăm variabile în moduri diferite. De exemplu, în urma definiției:

```
ALFAB    label BYTE
ALFAW    dw 1234H
```

o instrucțiune de forma inc ALFAB va incrementa octetul mai puțin semnificativ al cuvântului iar una de tip inc ALFAW va incrementa întreg cuvântul; un efect similar se poate obține prin operatorul PTR.

Similar, definiția:

```
adr_proc    label dword
proc_off     dw ?
proc_seg     dw ?
```

permite definirea explicită a segmentului și offset-ului dar și accesul la nivel de dublu-cuvânt, de exemplu într-o instrucțiune de apel indirect itsegment.

4.7 Definirea structurilor. Operații specifice

Structurile reprezintă colecții de date (câmpuri sau membri) plasate succesiv în memorie, grupate sub un unic nume sintactic. Dimensiunea unei structuri este suma dimensiunilor câmpurilor componente. Forma generală a definiției unei structuri este:

```
nume_structură    STRUC
.
.
.
nume_membru        definiție_date
.
.
.
nume_structură    ENDS
```

Prin aceasta se definește tipul structurii (șablonul) fără a se rezerva spațiu de memorie. Definițiile de date pot cuprinde directive DB, DW etc. cu valori inițiale asociate, ?, DUP etc., exact ca orice definiție de date. Numele membrilor structurii trebuie să fie distincte, chiar dacă aparțin unor structuri distincte. Iată un exemplu de definiție de tip de structură:

```
ALFA STRUC
    a      db ?
    b      dw 1111H
    c      dd 1234
    d      db ?
ALFA ENDS
```

O structură definită este interpretată ca un tip nou de date, care poate apoi participa la definiții concrete de date, cu rezervare de spațiu de memorie. De exemplu, putem defini o structură de tipul ALFA, cu numele x:

```
ALFA x <, , 777, 5>
```

În care ALFA este tipul de date, x este numele variabilei iar între paranteze unghiulare se trec valorile inițiale ale câmpurilor structurii x. Prezența virgulelor din lista de valori inițiale precizează că primii doi membri sunt inițializați cu valorile implicite de la definiția tipului ALFA, al 3-lea membru este inițializat cu valoarea 777 iar al 4-lea cu valoarea 5. Astfel, definiția variabilei x este echivalentă cu secvența de definiții:

```
a      db ?
b      dw 1111H
c      dd 777
d      db 5
```

Principalul avantaj al structurilor este accesul la membri într-o formă asemănătoare limbajelor de nivel înalt. De exemplu, pentru a încărca în SI câmpul b al structurii x, putem scrie:

```
mov    si, x.b
```

Accesul la membrii structurii x poate fi deci detaliat în:

- x.a - variabilă de tip byte
- x.b - variabilă de tip word
- x.c - variabilă de tip dword
- x.d - variabilă de tip byte

Putem acum defini un tablou de 5 structuri de tip ALFA:

```
y      ALFA 5 dup <, , , 10>
```

În care primii 3 membri sunt inițializați cu valorile de la definiția tipului structurii iar al 4-lea cu valoarea 10. Au acum sens expresii de forma: y[0].a, y[SI], y[8].c, y[BX].d etc. De fapt, toate aceste expresii se evaluează din adresa de început a variabilei y la care se adaugă un deplasament corespunzător. Expresia y[11].d este echivalentă cu y + 11 + deplasamentul câmpului d față în cadrul tipului ALFA.

Se observă gestiunea deplasamentelor cade în sarcina utilizatorului. Dacă dorim să accesăm câmpul c al celei de-a doua structuri din tabloul y, nu putem

scrie y[1].c, așa cum am scrie într-un limbaj de nivel înalt, ci y[8].c. Limbajul de asamblare ne pune la dispoziție operatorul TYPE, care întoarce numărul de octeți ocupat de o structură. Pentru a generaliza accesul la al i-lea element al tabloului y (i cuprins între 0 și 4), putem scrie y[i*TYPE ALFA].c.

Problema se complică dacă indicele i este cunoscut de-abia la momentul execuției (fiind memorat într-un registru de exemplu). O expresie de forma:

```
mov    al, y [SI*TYPE ALFA].d
```

are sens numai la procesoarele care acceptă adresare cu factor de scală (80386 și peste) și numai dacă factorul de scală este 1, 2, 4, sau 8. În caz contrar, deplasamentul trebuie calculat explicit prin operații aritmetice.

Cu toate aceste limitări, structurile sunt intens utilizate atunci când se lucrează cu anumite șabloane fixe, cum ar fi, de exemplu, imaginea stivei la intrarea într-o procedură sau transmiterea prin referință a unei zone de date. Să considerăm o înregistrare logică descrisă prin câmpuri:

- câmpul NUME - șir ASCII de 30 de caractere
- câmpul PRENUME - șir ASCII de 20 de caractere
- câmpul COD - întreg pe 4 octeți

Dorim să transmitem unei proceduri far cu numele PRELUCRARE o asemenea înregistrare. Cel mai simplu mod este de a transmite adresa (near sau far) a zonei de memorie în care este stocată înregistrarea, de exemplu printr-un registru sau o pereche de registre; pentru accesul comod la câmpuri definim structura:

```
PERSOANA STRUC
    NUME      db 30 dup (?)
    PRENUME   db 20 dup (?)
    COD       dd ?
PERSOANA ENDS
```

Definiția concretă a datelor și secvența de apel a procedurii este:

```
.data
    om          PERSOANA <'Ionescu', 'Ion', 77206>
    adr_om      dd om          ; Pointer la om
.code
    les        di, adr_om
    call       PRELUCRARE
```

Se pune acum problema accesului la câmpurile înregistrării, din interiorul procedurii. Beneficiem acum de structura PERSOANA. Ca să încărcăm adresa câmpului PRENUME vom scrie:

```
lea     bx, es:[di].PRENUME
```

Perechea ES:DI conține adresa structurii om, iar expresia es:[di].PRENUME reprezintă deplasamentul câmpului PRENUME. În acest moment, în ES:BX avem adresa șirului de caractere de tip PRENUME. Similar se procedează și pentru celelalte câmpuri: ca să preluăm în DX:AX câmpul COD, putem scrie:

```
mov     ax, word ptr es:[di].COD
mov     ax, word ptr es:[di].COD
```

Este evident că toate secvențele de mai sus se puteau înlocui cu deplasamente calculate explicit de către programator. Încărcarea câmpului COD ar fi putut fi scrisă:

```
mov ax, word ptr es:[di + 50]
mov dx, word ptr es:[di + 52]
```

dar forma cu nume de câmpuri este mult mai clară și mai comodă. În plus, noi putem greși la calculul unui deplasament, dar asamblorul nu.

Această tehnică este benefică și din punctul de vedere al întreținerii unui program. Dacă se modifică ulterior înregistrarea PERSOANA și se adaugă încă un câmp între PRENUME și COD, toate deplasamentele calculate explicit trebuie modificate. În varianta cu structură, trebuie dor să schimbăm definiția structurii, restul fiind făcut automat la asamblare.

4.8 Definirea înregistrărilor. Operații specifice

Înregistrările (RECORD) corespund de fapt unor structuri împachetate din limbajele de nivel înalt. Concret, o înregistrare este o definiție de câmpuri de biți de lungime maximă 8 sau 16. Din punct de vedere sintactic, definiția unei înregistrări este similară cu cea a unei structuri, forma generală fiind:

```
nume_înregistrare RECORD nume_câmp:valoare, ...
```

sau, cu inițializare implicită:

```
nume_înregistrare RECORD nume_câmp:valoare = expresie, ...
```

Valorile care apar asociate cu numele de câmpuri dau de fapt numărul de biți pe care se memorează câmpul respectiv. În varianta cu inițializare, expresia care apare după semnul egal se evaluează la o cantitate care se reprezintă pe numărul de biți asociat câmpului respectiv. La fel ca la structuri, numele câmpurilor trebuie să fie distincte, chiar dacă aparțin unor înregistrări diferite.

Să considerăm un exemplu:

```
BETA record X:7, Y:4, Z:5
```

prin care se definește un șablon de 16 biți, grupați în câmpurile X, Y și Z (vezi Figura 4.2)

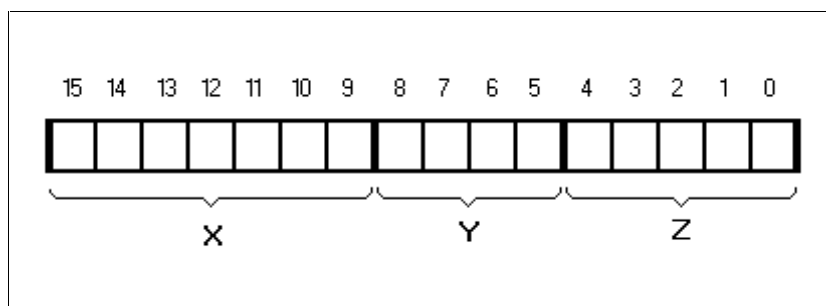


Figura 4.2 Definiția unei înregistrări

La fel ca la structuri, putem acum defini variabile de tipul înregistrării BETA:

```
VAL BETA <5, 2, 7>
```

în care valorile dintre parantezele unghiulare inițializează cele 3 câmpuri de biți. Se observă că această definiție este echivalentă cu definiția explicită:

```
VAL dw 0000101001000111B
```


Există doi operatori specifici înregistrărilor: MASK și WIDTH. Operatorul MASK primește un nume de câmp și furnizează o mască cu biți 1 pe poziția câmpului respectiv și 0 în rest. Astfel, expresiile MASK X și MASK Y sunt echivalente cu constantele binare 1111111000000000B și 0000000111100000B.

Asemenea expresii pot fi utilizate pentru selectarea câmpurilor respective, ca în exemplele de mai jos:

AND	AX, MASK Y	; Filtrează câmpul Y
AND	BX, NOT MASK Z	; Forțează câmpul Z la 0

Operatorul WIDTH primește un nume de înregistrare sau un nume de câmp dintr-o înregistrare, întorcând numărul de biți ai înregistrării sau ai câmpului respectiv. De exemplu, secvența:

MOV	AL, WIDTH BETA	; AL <--- 16
MOV	BL, WIDTH Y	; BL <--- 4

va încărca în AL valoarea 16 și în BL valoarea 4.

Dacă se utilizează un nume de câmp într-o instrucțiune de tip MOV, se va încărca un contor de deplasare, util pentru a deplasa câmpul respectiv pe pozițiile cele mai puțin semnificative.

Să presupunem că dorim să testăm valoarea numerică a câmpului Y din înregistrarea VAL. Nu este suficient să-l izolăm și să folosim o instrucțiune de comparație. Înainte de comparație, câmpul Y trebuie adus pe pozițiile cele mai din dreapta. Realizăm aceste operații prin secvența:

MOV	AX, VAL	; Preluare înregistrare
AND	AX, MASK Y	; Selectare câmp Y
MOV	CL, Y	; Încărcare în CL a valorii 5
SHR	AX, CL	; Deplasare câmp Y la dreapta

4.9 Operatori în limbajul de asamblare

Limbajul de asamblare dispune de un set de operatori cu care se pot forma expresii aritmetice și logice. Aceste expresii sunt evaluate la momentul asamblării, producând de fapt constante numerice. Este esențial să deosebim instrucțiunile executabile (codul mașină) de operațiile care se fac la momentul asamblării.

4.9.1 Operatori aritmetici și logici

Operatorii aritmetici sunt: **+**, **-**, *****, **/**, **MOD**, **SHL** și **SHR**. Primii patru au semnificații evidente și operează numai cu cantități întregi. Operatorul MOD produce restul la împărțire iar SHL și SHR provoacă deplasare la stânga sau la dreapta.

De exemplu, instrucțiunea:

mov	ax, 1 SHL 3	; 1 deplasat la stânga cu 3 biți
-----	-------------	----------------------------------

este echivalentă cu:

mov	ax, 100B
-----	----------

Similar, în directiva:

COUNT dw (\$ - TAB)/2

se va evalua la momentul asamblării expresia $(\$ - TAB)/2$, adică diferența dintre contorul curent de locații și adresa TAB, împărțită la 2.

Operatorii logici sunt **NOT**, **AND**, **OR** și **XOR**, cu semnificații evidente. Toți acești operatori acceptă drept operanzi constante întregi; operațiile respective se fac la nivel de bit. Ei nu trebuie confundați cu instrucțiunile executabile cu același nume. În instrucțiunea:

```
and al, (1 SHL 3) OR (1 SHL 5)
```

registrul AL va fi încărcat cu constanta 101000B.

4.9.2 Operatori relaționali

Operatorii relaționali sunt **EQ**, **NE**, **LT**, **LE**, **GT**, **GE**, cu semnificații evidente. Aceștia întorc valori logice, codificate ca 0 sau ca secvențe de 1 pe un număr corespunzător de biți. În urma definițiilor:

x	db	(1 EQ 2)
y	dw	(1 NE 2)
z	dd	(1 LT 2)

variabilele x, y și z vor fi inițializate cu constantele 0, 0FFFFH, respectiv 0FFFFFFFFH.

Operatorii relaționali sunt utilizați în special în directivele de asamblare condiționată.

4.9.3 Operatorul de atribuire =

Operatorul de atribuire = definește constante simbolice, fiind similar cu directiva EQU, dar permite redefinirea simbolilor utilizați. O definire de forma:

```
N EQU 1
N EQU 2
```

este semnalată ca eroare, dar secvența:

```
N = 1
N = 2
```

este corectă.

Acest operator este utilizat în special în definițiile de macroinstrucțiuni.

4.9.4 Operatori care întorc valori

Acești operatori se aplică unor entități ale programului în limbaj de asamblare (variabile, etichete), întorcând valori asociate acestora.

Operatorul SEG

Se aplică atât variabilelor cât și etichetelor și furnizează adresa de segment asociată variabilei respective. Dacă var este o variabilă, atunci putem scrie secvența: mov ax, SEG var

```
mov ds, ax
```

Operatorul OFFSET

Este similar cu SEG, furnizând offset-ul asociat variabilei sau etichetei:

```
mov bx, OFFSET var
```

Operatorul THIS

Acest operator creează un operand care are asociate o adresă de segment și un offset identice cu contorul curent de locații. Forma generală a operandului este:

THIS tip

în care tip poate fi BYTE, WORD, DWORD, QWORD sau TBYTE pentru definiții de date, respectiv NEAR sau FAR pentru etichete. Operatorul THIS se utilizează de obicei cu directiva EQU. De exemplu, definiția constantei simbolice:

ALFA EQU THIS WORD

este echivalentă cu definiția unei etichete:

ALFA LABEL WORD

Operatorul TYPE

Se aplică variabilelor și etichetelor, întorcând tipul acestora. Pentru variabile întoarce valorile 1, 2, 4, 8 sau 10 pentru variabile simple (definite cu directivele DB, DW, DD, DQ respectiv DT), iar pentru structuri întoarce numărul de octeți pe care este memorată structura respectivă. Pentru etichete, întoarce tipul etichetei (NEAR sau FAR).

Operatorul LENGTH

Se aplică numai variabilelor și întoarce numărul de elemente definite în variabilă respectivă. De exemplu, în definiția:

A DW 100 dup (?)

expresia LENGTH A are valoarea 100.

Operatorul SIZE

Se aplică numai variabilelor și întoarce dimensiunea în octeți a variabilei respective. În definiția de mai sus, expresia SIZE A se evaluează are valoarea 200.

Pentru o variabilă oarecare var, are loc totdeauna identitatea:

SIZE var = (LENGTH var) * (TYPE var)

Acești operatori sunt utili la prelucrarea tablourilor. Secvența următoare nu depinde de tipul de bază al tabloului TAB și nici de dimensiunea sa:

```
.data
    TAB    dd 100 dup (?)
.code
    mov    cx, LENGTH TAB
    lea    si, TAB
buc1a:
    ....
    add    si, TYPE TAB
    loop   buc1a
```

Dacă înlocuim acum tipul de bază al tabloului TAB cu o structură de tip PERSOANA (definită în 4.7), modificând și dimensiunea:

```
PERSOANA STRUC
    .....
PERSOANA ENDS
```

```
.data
    TAB    PERSOANA 50 dup <,,,>
```

partea de program nu trebuie modificată, deoarece asamblorul va evalua corect expresiile LENGTH TAB și TYPE TAB.

4.9.5 Operatorul PTR

Acest operator se aplică atât variabilelor cât și etichetelor, având ca efect schimbarea tipului variabilei sau etichetei respective. Este obligatoriu în cazul în care se folosesc referințe anonime la memorie, din care nu se poate deduce tipul operandului:

```
add    word ptr [bx], 2
call   dword ptr [bx]
call   near ptr proc
```

4.10 Directive de asamblare condiționată

Permit ignorarea unor porțiuni din textul sursă, funcție de o condiție care se poate evalua la asamblare. Directiva de bază este IF, cu forma generală:

```
IF expresie
    .
    .
    .
[ELSE]
    .
    .
    .
ENDIF
```

Dacă expresia din directiva IF este adevărată (adică este diferită de 0) se assemblează porțiunea de program dintre IF și ELSE și se ignoră porțiunea dintre ELSE și ENDIF. Dacă expresia din IF este falsă, se assemblează porțiunea de program dintre ELSE și ENDIF și se ignoră porțiunea dintre IF și ELSE. Ramura ELSE este opțională.

Aceste operații se petrec la momentul asamblării, permițând întreținerea comodă a unui program de dimensiuni mari sau chiar menținerea în același text sursă a mai multor variante de program executabil. Să presupunem că dorim un program care să poată fi configurat rapid pentru modele de date "mici", respectiv "mari". Aceasta presupune ca orice secvență de instrucțiuni care definește, citește sau scrie o adresă să fie inclusă în directive de asamblare condiționată:

```
FALSE    equ    0
TRUE     equ    NOT FALSE
DATE_MARI equ    TRUE
DATE_MICI equ    NOT DATE_MARI
.data
    X            dw    100 dup (?)
IF DATE_MARI
    ADR_X        dd    X
ELSE
    ADR_X        dw    X
ENDIF
.code
IF DATE_MARI
```

```

        lds    di, ADR_X
    ELSE
        mov    si, ADR_X
    ENDIF
    ;
    ; Prelucrare TABLOU
    ;

```

Există și directivele IFDEF/ENDIF și IFNDEF/ENDIF, care testează dacă un simbol este definit sau nu și directivele IFB/ENDIF (If Blank), respectiv IFNB/ENDIF (If Not Blank), care testează dacă un simbol este vid sau nevid. Acestea din urmă vor fi utilizate la definirea macroinstrucțiunilor.

În expresiile care apar în directivele de tip IF se utilizează de obicei operatori logici și relaționali.

Să considerăm un exemplu de program sursă multifuncțional. Se dorește scrierea unei secvențe care să calculeze în acumulator suma elementelor unui tablou A (de octeți sau de cuvinte), definit în segmentul curent de date, ca sumă pe octeți su pe cuvinte, indiferent de tipul și dimensiunea tabloului. Tipul sumei este dictat de constantele simbolice SUM_B și SUM_W.

```

FALSE      equ 0
TRUE       equ NOT FALSE
SUM_B      equ TRUE
SUM_W      equ NOT SUM_B
.code
if SUM_B
    mov     cx, SIZE A                ; Număr de iterații
else
    mov     cx, (SIZE A)/2
endif
    xor     bx, bx                    ; Indice inițial
    mov     ax, bx                    ; Sumă inițială
bucla:
if SUM_B
    add     al, byte ptr A [bx]       ; Sumă pe octet
    inc     bx                        ; Actualizare adresă
else
    add     ax, word ptr A [bx]       ; Sumă pe cuvânt
    inc     bx                        ; Actualizare
    inc     bx                        ; adresă
endif
    loop    bucla                     ; Buclă

if SUM_W AND (TYPE A EQ 1) AND ((SIZE A) MOD 2) EQ 1
    push    ax
    mov     al, byte ptr A [bx]       ; Un eventual
    cbw                                          ; ultim
    mov     dx, ax                     ; octet
    pop     ax
    add     ax, dx
endif

```

Se observă că nu putem inițializa contorul CX cu expresia LENGTH A, deoarece numărul de iterații este dictat de modul de calcul al sumei și nu de tipul tabloului. Ca atare, luăm dimensiunea în octeți a tabloului, pe care o împărțim

la 2 în cazul sumei pe cuvânt. Calculul sumei și actualizarea adresei sunt evidente.

Apare o problemă specială în următoarea situație: se cere suma pe cuvânt iar tabloul, definit ca tablou de octeți, are un număr impar de elemente. În această situație, este evident că ultimul octet nu este considerat în bucla de sumare. Ca atare, el este adăugat explicit la sfârșit.

De remarcat formularea situației de mai sus în termenii condiției logice de la ultima directivă IF:

- SUM_W codifică cererea de calcul a sumei la nivel de cuvânt;
- (TYPE A EQ 1) codifică situația în care tabloul A este definit la nivel de octet;
- ((SIZE A) MOD 2) codifică situația în care tabloul are un număr impar de octeți.