```
 0  (* 001 entier1 *)
 1  (* 11 avr 87 *)
 2
 3  (*$r+*)
 4  PROGRAM teste_entier_1;
 5    VAR entier, entier2: INTEGER;
 6    BEGIN
 7      entier:= entier2* $77;
 8      entier:= entier2 DIV $77;
 9      entier:= entier2 MOD $77;
10      entier:= SQR(entier2);
11      entier:= SUCC(entier2);
12      entier:= entier2+ 1;
13      entier:= $11* $22+ $33* ($44+ $55);
14    END.
```

Fig. 5.11 Le programme *desassemble_entiers*.

et voici le code désassemblé:

```
2D4D E849DD                    call    TU_PARAGRAPHES
2D50 0600                      hex     0600 ; options
2D52 7B06                      hex     7B06 ; cs
2D54 3F10                      hex     3F10 ; ds
2D56 DE02                      hex     DE02 ; code
2D58 2700                      hex     2700 ; globales
2D5A 0004                      hex     0004 ; min piles
2D5C 00A0                      hex     00A0 ; max piles
2D5E 1000                      hex     1000 ; fichiers
2D60 0000                      hex     0000 ; input
2D62 0000                      hex     0000 ; output

2D64 8BEC                      mov     bp, sp
2D66 E814DF                    call    TU_TAILLE_CODE
2D69 6700                      hex     6700 ; taille
2D6B 6D2D                      adr     SAUT_PRINCIPAL

2D6D E90000          SAUT_PRINCIPAL jmp PRINCIPAL

2D70 A16402          PRINCIPAL mov   ax, [$0264]
2D73 B97700                    mov     cx, $0077
2D76 F7E9                      imul    ax, cx
2D78 A36202                    mov     [$0262], ax
2D7B A16402                    mov     ax, [$0264]

2D7E B97700                    mov     cx, $0077
2D81 99                        cwd
2D82 F7F9                      idiv    ax, cx
2D84 A36202                    mov     [$0262], ax
2D87 A16402                    mov     ax, [$0264]
```



```
2D8A B97700                    mov     cx, $0077
2D8D 99                        cwd
2D8E F7F9                      idiv    ax, cx
2D90 92                        xchg    dx
2D91 A36202                    mov     [$0262], ax

2D94 A16402                    mov     ax, [$0264]
2D97 F7E8                      imul    ax, ax
2D99 A36202                    mov     [$0262], ax

2D9C A16402                    mov     ax, [$0264]
2D9F 40                        inc     ax
2DA0 A36202                    mov     [$0262], ax

2DA3 A16402                    mov     ax, [$0264]
2DA6 050100                    add     ax, $0001
2DA9 A36202                    mov     [$0262], ax

2DAC B81100                    mov     ax, $0011
2DAF B92200                    mov     cx, $0022
2DB2 F7E9                      imul    ax, cx
2DB4 50                        push    ax
2DB5 B83300                    mov     ax, $0033
2DB8 50                        push    ax
2DB9 B84400                    mov     ax, $0044
2DBC 055500                    add     ax, $0055
2DBF 59                        pop     cx
2DC0 F7E9                      imul    ax, cx
2DC2 59                        pop     cx
2DC3 03C1                      add     ax, cx
2DC5 A36202                    mov     [$0262], ax

2DC8 E90000                    jmp     FIN

2DCB 33C0            FIN       xor     ax, ax
2DCD E870DE                    call    TU_HALT
```

Fig. 5.12 Le code du programme *desassemble_entiers*.

Notez que:

- l'arithmétique simple (+, -, *, **DIV**, **MOD**, **SQR**, **SUCC**, **PRED**) est faite directement par des instructions 8088
- **SQR** utilise le même code que *entier** *entier*, mais **SUCC** ou **PRED** sont plus efficaces que +1 ou -1
- les expressions arithmétiques complexes utilisent la pile 8088 pour stocker les résultats intermédiaires (PUSH AX et POP CX).

5.3.1.2 COMPARAISON D'ENTIERS

Voici un programme contenant des comparaisons d'entiers:

```
 0  (* 001 centier *)
 1  (* 24 avr 87 *)
 2
 3  (*$r+*)
 4  PROGRAM desassemble_compare_entiers;
 5    VAR entier: INTEGER;
 6        booleen: BOOLEAN;
 7    BEGIN
 8      booleen:= entier= $11;
 9      booleen:= entier> $22;
10      booleen:= entier>= $33;
11      booleen:= entier< $44;
12      booleen:= entier<= $55;
13      booleen:= entier<> $66;
14    END.
15
16
```

Fig. 5.13 Le programme *compare_entiers*.

et voici le code correspondant:

```
2D4D E849DD                    call   TU_PARAGRAPHS
2D50 0600                      hex    0600 ; options
2D52 7B06                      hex    7B06 ; cs
2D54 3F10                      hex    3F10 ; ds
2D56 E102                      hex    E102 ; code
2D58 2700                      hex    2700 ; globales
2D5A 0004                      hex    0004 ; min piles
2D5C 00A0                      hex    00A0 ; max piles
2D5E 1000                      hex    1000 ; fichiers
2D60 0000                      hex    0000 ; input
2D62 0000                      hex    0000 ; output

2D64 8BEC                      mov    bp, sp
2D66 E814DF                    call   TU_TAILLE_CODE
2D69 9F00                      hex    9F00 ; taille programme
2D6B 6D2D                      adr    SAUT_PRINCIPAL

2D6D E90000          SAUT_PRINCIPAL jmp    PRINCIPAL
```



```
2D70 A16202          PRINCIPAL mov    ax, [$0262]
2D73 3D1100                    cmp    ax, $0011
2D76 B80100                    mov    ax, $0001
2D79 7401                      je     LR_2D7C
2D7B 48                        dec    ax
2D7C B90000          LR_2D7C   mov    cx, $0000
2D7F BA0100                    mov    dx, $0001
2D82 E8EFE1                    call   TU_RANGE_CHECK
2D85 A26402                    mov    [$0264], al

2D88 A16202                    mov    ax, [$0262]
2D8B 3D2200                    cmp    ax, $0022
2D8E B80100                    mov    ax, $0001
2D91 7F01                      jnle   $2D94
2D93 48                        dec    ax
2D94 B90000          LR_2D94   mov    cx, $0000
2D97 BA0100                    mov    dx, $0001
2D9A E8D7E1                    call   TU_RANGE_CHECK
2D9D A26402                    mov    [$0264], al

2DA0 A16202                    mov    ax, [$0262]
2DA3 3D3300                    cmp    ax, $0033
2DA6 B80100                    mov    ax, $0001
2DA9 7D01                      jnl    $2DAC
2DAB 48                        dec    ax
2DAC B90000          LR_2DAC   mov    cx, $0000
2DAF BA0100                    mov    dx, $0001
2DB2 E8BFE1                    call   TU_RANGE_CHECK
2DB5 A26402                    mov    [$0264], al

2DB8 A16202                    mov    ax, [$0262]
2DBB 3D4400                    cmp    ax, $0044
2DBE B80100                    mov    ax, $0001
2DC1 7C01                      jl     $2DC4
2DC3 48                        dec    ax
2DC4 B90000          LR_2DC4   mov    cx, $0000
2DC7 BA0100                    mov    dx, $0001
2DCA E8A7E1                    call   TU_RANGE_CHECK
2DCD A26402                    mov    [$0264], al

2DD0 A16202                    mov    ax, [$0262]
2DD3 3D5500                    cmp    ax, $0055
2DD6 B80100                    mov    ax, $0001
2DD9 7E01                      jle    $2DDC
2DDB 48                        dec    ax
2DDC B90000          LR_2DDC   mov    cx, $0000
2DDF BA0100                    mov    dx, $0001
2DE2 E88FE1                    call   TU_RANGE_CHECK
2DE5 A26402                    mov    [$0264], al
```

```
2DE8 A16202                    mov     ax, [$0262]
2DEB 3D6600                    cmp     ax, $0066
2DEE B80100                    mov     ax, $0001
2DF1 7501                      jnz     $2DF4
2DF3 48                        dec     ax
2DF4 B90000           LR_2DF4  mov     cx, $0000
2DF7 BA0100                    mov     dx, $0001
2DFA E877E1                    call    TU_RANGE_CHECK
2DFD A26402                    mov     [$0264], al

2E00 E90000                    jmp     FIN

2E03 33C0             FIN      xor     ax, ax
2E05 E838DE                    call    TU_HALT
```

Fig. 5.14 Le code du programme *desassemble_comparaisons_entiers*.

Notez que:

- toutes les comparaisons se font donc directement en 8088
- le code pour affecter une valeur à une variable booléenne sera examiné lorsque nous traiterons le type booléen.

5.3.1.3 AND, OR, XOR, SHR et SHL

Voici un programme qui utilise les instructions de traitement au niveau du bit:

```
 0  (* 001 entier3 *)
 1  (* 11 avr 87 *)
 2
 3  (*$r+*)
 4  PROGRAM desassemble_entier_3;
 5    VAR entier, entier2: INTEGER;
 6    BEGIN
 7      entier:= entier2 AND $11;
 8      entier:= entier2 OR $22;
 9      entier:= entier2 XOR $33;
10      entier:= NOT entier2;
11      entier:= entier2 SHR $44;
12      entier:= entier2 SHL $44;
13    END.
```

Fig. 5.15 Le programme *desassemble_entier_3*.

et voici le code correspondant:


```
2D4D E849DD                    call    TU_PARAGRAPHES
2D50 0600                      hex     0600 ; options
2D52 7B06                      hex     7B06 ; cs
2D54 3F10                      hex     3F10 ; ds
2D56 DC02                      hex     DC02 ; code
2D58 2700                      hex     2700 ; globales
2D5A 0004                      hex     0004 ; min piles
2D5C 00A0                      hex     00A0 ; max piles
2D5E 1000                      hex     1000 ; fichiers
2D60 0000                      hex     0000 ; input
2D62 0000                      hex     0000 ; output

2D64 8BEC                      mov     bp, sp
2D66 E814DF                    call    TU_TAILLE_CODE
2D69 4800                      hex     4800 ; taille
2D6B 6D2D                      adr     SAUT_PRINCIPAL

2D6D E90000           SAUT_PRINCIPAL jmp PRINCIPAL

2D70 A16402           PRINCIPAL mov   ax, [$0264]
2D73 251100                    and     ax, $0011
2D76 A36202                    mov     [$0262], ax

2D79 A16402                    mov     ax, [$0264]
2D7C 0D2200                    or      ax, $0022
2D7F A36202                    mov     [$0262], ax

2D82 A16402                    mov     ax, [$0264]
2D85 353300                    xor     ax, $0033
2D88 A36202                    mov     [$0262], ax

2D8B A16402                    mov     ax, [$0264]
2D8E F7D0                      not     w ax
2D90 A36202                    mov     [$0262], ax

2D93 A16402                    mov     ax, [$0264]
2D96 B94400                    mov     cx, $0044
2D99 D3E8                      shr     ax, cl
2D9B A36202                    mov     [$0262], ax

2D9E A16402                    mov     ax, [$0264]
2DA1 B94400                    mov     cx, $0044
2DA4 D3E0                      shl     ax, cl
2DA6 A36202                    mov     [$0262], ax

2DA9 E90000                    jmp     FIN

2DAC 33C0             FIN      xor     ax, ax
2DAE E88FDE                    call    TU_HALT
```

Fig. 5.16 Le code de programme *desassemble_entier_3*.

Ces opérateurs utilisent donc simplement les instructions 8088 correspondantes.

5.3.1.4 PROCEDURES ET FONCTIONS STANDARD

Voici finalement des instructions faisant appel à des routines librairie:

```
 0  (* 001 entier2 *)
 1  (* 11 avr 87 *)
 2
 3  (*$r+*)
 4  PROGRAM desassemble_entier2;
 5    VAR entier, entier2: INTEGER;
 6    BEGIN
 7      entier:= ABS(entier2);
 8      READ(entier);
 9      WRITE(entier);
10      WRITE(entier: $22);
11    END.
```

Fig. 5.17 Le programme *desassemble_entier2*.

et voici le code généré:

```
2D4D E849DD                     call    TU_PARAGRAPHS
2D50 0600                       hex     0600 ; options
2D52 7B06                       hex     7B06 ; cs
2D54 3F10                       hex     3F10 ; ds
2D56 DB02                       hex     DB02 ; code
2D58 2700                       hex     2700 ; globales
2D5A 0004                       hex     0004 ; min piles
2D5C 00A0                       hex     00A0 ; max piles
2D5E 1000                       hex     1000 ; fichiers
2D60 0000                       hex     0000 ; input
2D62 0000                       hex     0000 ; output

2D64 8BEC                       mov     bp, sp
2D66 E814DF                     call    TU_TAILLE_CODE
2D69 4500                       hex     4500 ; taille
2D6B 6D2D                       adr     SAUT_PRINCIPAL

2D6D E90000         SAUT_PRINCIPAL  jmp     PRINCIPAL

2D70 A16402         PRINCIPAL   mov     ax, [$0264]
2D73 E80EE3                     call    TU_ABS_ENTIER
2D76 A36202                     mov     [$0262], ax

2D79 E8D9F7                     call    TU_READ_DEFAUT
2D7C BF6202                     mov     di, $0262
2D7F 1E                         push    ds
2D80 E8BDF9                     call    TU_READ_ENTIER
2D83 E84DE2                     call    TU_IO_CHECK
```



```
2D86 E899F7                     call    TU_WRITE_DEFAUT
2D89 A16202                     mov     ax, [$0262]
2D8C 50                         push    ax
2D8D B80000                     mov     ax, $0000
2D90 E8F0FA                     call    TU_WRITE_ENTIER
2D93 E83DE2                     call    TU_IO_CHECK

2D96 E889F7                     call    TU_WRITE_DEFAUT
2D99 A16202                     mov     ax, [$0262]
2D9C 50                         push    ax
2D9D B82200                     mov     ax, $0022
2DA0 E8E0FA                     call    TU_WRITE_ENTIER
2DA3 E82DE2                     call    TU_IO_CHECK

2DA6 E90000                     jmp     FIN

2DA9 33C0           FIN         xor     ax, ax
2DAB E892DE                     call    TU_HALT
```

Fig. 5.18 Le code du programme *desassemble_entier2*.

Notez que:

- le programme utilise la routine TU_ABS_ENTIER pour calculer la valeur absolue
- pour la lecture:
  - TU_READ_DEFAUT prépare la lecture du fichier d'entrée par défaut
  - la lecture proprement dite est effectuée en plaçant dans DI le décalage de la variable, et en poussant son segment (ici DS) sur la pile. La routine TU_READ_ENTIER utilise cette adresse segmentée pour placer le résultat de la lecture dans la zone allouée à la variable
  - la routine TU_IO_CHECK est chargée de provoquer une erreur d'entrée / sortie au cas où une erreur a été rencontrée au niveau DOS ou BIOS
- pour l'affichage:
  - TU_WRITE_DEFAUT prépare l'affichage du fichier de sortie par défaut
  - l'affichage proprement dit est effectué en plaçant la valeur à afficher sur la pile, et le format de l'affichage dans AX. La routine TU_WRITE_ENTIER utilise ces deux paramètres pour effectuer l'affichage
  - la routine TU_IO_CHECK est chargée de provoquer une erreur d'entrée / sortie au cas ou une erreur a été rencontrée au niveau DOS ou BIOS

5.3.1.5 AFFECTATION ET EXPRESSION D'INTERVALLES

Les intervalles sont traités comme des entiers, avec en plus la vérification de bornes. Voici un exemple:

```
 0  (* 001 interv *)
 1  (* 11 avr 87 *)
 2
 3  (*$r+*)
 4  PROGRAM desassemble_intervalle;
 5    VAR intervalle: $0F..$2F;
 6        entier: INTEGER;
 7    BEGIN
 8      intervalle:= 1;
 9      READ(intervalle);
10      WRITE(intervalle);
11      intervalle:= entier;
12      intervalle:= intervalle+ 1;
13    END.
```

Fig. 5.19 Le programme *desassemble_intervalle*.

et voici le code:

```
2D4D E849DD                     call    TU_PARAGRAPHS
2D50 0600                       hex     0600 ; options
2D52 7B06                       hex     7B06 ; cs
2D54 3F10                       hex     3F10 ; ds
2D56 DD02                       hex     DD02 ; code
2D58 2700                       hex     2700 ; globales
2D5A 0004                       hex     0004 ; min piles
2D5C 00A0                       hex     00A0 ; max piles
2D5E 1000                       hex     1000 ; fichiers
2D60 0000                       hex     0000 ; input
2D62 0000                       hex     0000 ; output

2D64 8BEC                       mov     bp, sp
2D66 E814DF                     call    TU_TAILLE_CODE
2D69 6000                       hex     6000 ; taille
2D6B 6D2D                       adr     SAUT_PRINCIPAL

2D6D E90000             SAUT_PRINCIPAL  jmp     PRINCIPAL

2D70 B80100             PRINCIPAL       mov     ax, $0001
2D73 B90F00                             mov     cx, $000F
2D76 BA1F00                             mov     dx, $001F
2D79 E8F8E1                             call    TU_RANGE_CHECK
2D7C A26202                             mov     [$0262], al

2D7F E8D3F7                             call    TU_READ_DEFAUT
2D82 BF6202                             mov     di, $0262
2D85 1E                                 push    ds
2D86 E8B4F9                             call    TU_READ_INTERVA
2D89 E847E2                             call    TU_IO_CHECK
```



```
2D8C E893F7                             call    TU_WRITE_DEFAUT
2D8F A06202                             mov     al, [$0262]
2D92 32E4                               xor     ah, ah
2D94 50                                 push    ax
2D95 B80000                             mov     ax, $0000
2D98 E8E8FA                             call    TU_WRITE_ENTIER
2D9B E835E2                             call    TU_IO_CHECK

2D9E A16302                             mov     ax, [$0263]
2DA1 B90F00                             mov     cx, $000F
2DA4 BA1F00                             mov     dx, $001F
2DA7 E8CAE1                             call    TU_RANGE_CHECK
2DAA A26202                             mov     [$0262], al

2DAD A06202                             mov     al, [$0262]
2DB0 32E4                               xor     ah, ah
2DB2 050100                             add     ax, $0001
2DB5 B90F00                             mov     cx, $000F
2DB8 BA1F00                             mov     dx, $001F
2DBB E8B6E1                             call    TU_RANGE_CHECK
2DBE A26202                             mov     [$0262], al

2DC1 E90000                             jmp     FIN

2DC4 33C0                       FIN     xor     ax, ax
2DC6 E877DE                             call    TU_HALT
```

Fig. 5.20 Le code du programme *desassemble_intervalle*.

Notez que:

- comme nous travaillons en utilisant l'option Range Check (R+), toute affectation est précédée d'une vérification de borne
- comme les calculs sont effectués sur 16 bits, le compilateur se charge de mettre 0 dans les 8 bits forts de AX avant toute opération (**XOR** AH, AH). Cette opération se rencontrera pour tous les traitements de scalaires qui n'utilisent que 8 bits.
- l'affichage est le même que l'affichage d'un entier
- la lecture est spécifique mais ne semble guère se soucier des bornes précises de l'intervalle

Voici un programme qui traite des intervalles utilisant 2 octets:

```
 0  (* 001 interg *)
 1  (* 11 avr 87 *)
 2
 3  (*$r+*)
 4  PROGRAM desassemble_grand_intervalle;
 5    VAR intervalle: $00F0..$0110;
 6        intervalle2: -3..5;
 7    BEGIN
 8      intervalle:= $F2;
 9      intervalle2:= 2;
10    END.
```

Fig. 5.21 Le programme *desassemble_grand_intervalle*.

et voici le code correspondant:

```
2D4D E849DD                    call    TU_PARAGRAPHES
2D50 0600                      hex     0600 ; options
2D52 7B06                      hex     7B06 ; cs
2D54 3F10                      hex     3F10 ; ds
2D56 DA02                      hex     DA02 ; code
2D58 2700                      hex     2700 ; globales
2D5A 0004                      hex     0004 ; min piles
2D5C 00A0                      hex     00A0 ; max piles
2D5E 1000                      hex     1000 ; fichiers
2D60 0000                      hex     0000 ; input
2D62 0000                      hex     0000 ; output

2D64 8BEC                      mov     bp, sp
2D66 E814DF                    call    TU_TAILLE_CODE
2D69 2D00                      hex     2D00 ; programme
2D6B 6D2D                      adr     SAUT_PRINCIPAL

2D6D E90000          SAUT_PRINCIPAL    jmp     PRINCIPAL

2D70 B8F200          PRINCIPAL         mov     ax, $00F2
2D73 B9F000                            mov     cx, $00F0
2D76 BA1001                            mov     dx, $0110
2D79 E8F8E1                            call    TU_RANGE_CHECK
2D7C A36202                            mov     [$0262], ax

2D7F B80200                            mov     ax, $0002
2D82 B9FDFF                            mov     cx, $FFFD
2D85 BA0500                            mov     dx, $0005
2D88 E8E9E1                            call    TU_RANGE_CHECK
2D8B A36402                            mov     [$0264], ax

2D8E E90000                            jmp     FIN

2D91 33C0            FIN               xor     ax, ax
2D93 E8AADE                            call    TU_HALT
```

Fig. 5.22 Le programme *desassemble_grand_intervalle.*

Notez que:

- la routine TU_RANGE_CHECK est à nouveau invoquée pour vérifier que la valeur affectée est bien dans la plage déclarée.
- lorsque la donnée utilise plus de 8 bits, l'opération de mise à zéro des 8 bits forts n'est pas générée.

5.3.1.6 RESUME

Les traitements sur les entiers ou intervalles d'entiers:

- sont directement effectués en utilisant AX et les instructions du 8088
- lors d'opérations arithmétiques complexes, les résultats intermédiaires sont poussés sur la pile et récupérés dans CX
- les opérations sur des valeurs n'utilisant qu'un octet sont effectuées sur 16 bits, en mettant à zéro les 8 bits forts.

5.3.2 AFFECTATION ET EXPRESSIONS REELLES

5.3.2.1 AFFECTATION D'UN REEL

Voici un programme qui affecte une expression réelle à un réel:

```
0  (* 001 reel0 *)
1  (* 10 avr 87 *)
2
3  PROGRAM desassemble_affectation_reele;
4    VAR reel1, reel2: REAL;
5    BEGIN
6      reel1:= reel2;
7    END.
```

Fig. 5.23 Le programme *desassemble_affecte_reel.*

Voici le code généré:

```
2D4D E849DD                    call    TU_PARAGRAPHS
2D50 0600                      hex     0600 ; options
2D52 780B                      hex     780B ; cs
2D54 3C15                      hex     3C15 ; ds
2D56 D902                      hex     D902 ; code
2D58 2700                      hex     2700 ; globales
2D5A 0004                      hex     0004 ; min piles
2D5C 00A0                      hex     00A0 ; max piles
2D5E 1000                      hex     1000 ; fichiers
2D60 0000                      hex     0000 ; input
2D62 0000                      hex     0000 ; output

2D64 8BEC                      mov     bp, sp
2D66 E814DF                    call    TU_TAILLE_CODE
2D69 1E00                      hex     1E00 ; taille
2D6B 6D2D                      adr     SAUT_PRINCIPAL
```

```
2D6D E90000          SAUT_PRINCIPAL  jmp     PRINCIPAL

2D70 BF6202          PRINCIPAL       mov     di, $0262
2D73 1E                              push    ds
2D74 57                              push    di

2D75 BF6802                          mov     di, $0268
2D78 1E                              push    ds
2D79 E8BAEB                          call    TU_POUSSE_REEL

2D7C E8D7EB                          call    TU_POP_REEL_DI
2D7F E90000                          jmp     FIN

2D82 33C0            FIN             xor     ax, ax
2D84 E8B9DE                          call    TU_HALT
2D87 0000                            hex     0000
```

Fig. 5.24 L'affectation d'un réel

L'affectation est donc réalisée:

- en poussant sur la pile le segment (ici DS) et le décalage du réel (ici en utilisant DI)
- l'expression réelle est calculée et est placée sur la pile
- la routine TU_POP_REEL_DI se charge de récupérer les 6 octets du réel et de les stocker à l'adresse segmentée qui se trouve sous la valeur du réel

5.3.2.2 ARITHMETIQUE REELLE

Voici un programme effectuant des opérations arithmétiques sur des réels:

```
0  (* 001 reel1 *)
1  (* 10 avr 87 *)
2
3  PROGRAM desassemble_arithmetique_reele;
4    VAR reel1, reel2, reel3: REAL;
5    BEGIN
6      reel1:= reel2+ reel3;
7      reel1:= reel2/ reel3;
8    END.
```

Fig. 5.25 Le programme *desassemble_artihmétique_reelle*.

et voici le code:


```
2D4D E849DD                          call    TU_PARAGRAPHS
2D50 0600                            hex     0600 ; options
2D52 780B                            hex     780B ; cs
2D54 3C15                            hex     3C15 ; ds
2D56 DB02                            hex     DB02 ; code
2D58 2800                            hex     2800 ; globales
2D5A 0004                            hex     0004 ; min piles
2D5C 00A0                            hex     00A0 ; max piles
2D5E 1000                            hex     1000 ; fichiers
2D60 0000                            hex     0000 ; input
2D62 0000                            hex     0000 ; output

2D64 8BEC                            mov     bp, sp
2D66 E814DF                          call    TU_TAILLE_CODE
2D69 4100                            hex     4100 ; taille
2D6B 6D2D                            adr     SAUT_PRINCIPAL

2D6D E90000          SAUT_PRINCIPAL  jmp     PRINCIPAL

2D70 BF6202          PRINCIPAL       mov     di, $0262
2D73 1E                              push    ds
2D74 57                              push    di
2D75 BF6802                          mov     di, $0268
2D78 1E                              push    ds
2D79 E8BAEB                          call    TU_POUSSE_REEL
2D7C BF6E02                          mov     di, $026E
2D7F 1E                              push    ds
2D80 E8B3EB                          call    TU_POUSSE_REEL
2D83 E8E3EB                          call    TU_ADD_REEL
2D86 E8CDEB                          call    TU_POP_REEL_DI

2D89 BF6202                          mov     di, $0262
2D8C 1E                              push    ds
2D8D 57                              push    di
2D8E BF6802                          mov     di, $0268
2D91 1E                              push    ds
2D92 E8A1EB                          call    TU_POUSSE_REEL
2D95 BF6E02                          mov     di, $026E
2D98 1E                              push    ds
2D99 E89AEB                          call    TU_POUSSE_REEL
2D9C E803EC                          call    TU_DIVISE_REEL
2D9F E8B4EB                          call    TU_POP_REEL_DI

2DA2 E90000                          jmp     FIN

2DA5 33C0            FIN             xor     ax, ax
2DA7 E896DE                          call    TU_HALT
2DAA 0000                            hex     0000
```

Fig. 5.26 Le code du programme *desassemble_artihmétique_reelle*.

Notez que:

- toute opération arithmétique utilise les routines de la librairie.
- chaque opération s'effectue:
  - en poussant la première valeur sur la pile
  - en poussant la seconde valeur
  - en appelant la routine qui se chargera de dépiler les deux valeurs, calculer le résultat et rempiler le tout

5.3.2.3 COMPARAISON DE REELS

A présent examinons les comparaisons de réels:

```
0  (* 001 reel2 *)
1  (* 10 avr 87 *)
2
3  PROGRAM desassemble_comparaison_reels;
4    VAR reel1, reel2, reel3: REAL;
5        booleen: BOOLEAN;
6    BEGIN
7      booleen:= reel1 = reel2;
8      booleen:= reel1 >= reel2;
9    END.
```

Fig. 5.27 Le programme *desassemble_comparaison_reels*.

Voici son code:

```
2D4D E849DD                          call   TU_PARAGRAPHS
2D50 0600                            hex    0600 ; options
2D52 780B                            hex    780B ; cs
2D54 3C15                            hex    3C15 ; ds
2D56 DB02                            hex    DB02 ; code
2D58 2800                            hex    2800 ; globales
2D5A 0004                            hex    0004 ; min piles
2D5C 00A0                            hex    00A0 ; max piles
2D5E 1000                            hex    1000 ; fichiers
2D60 0000                            hex    0000 ; input
2D62 0000                            hex    0000 ; output

2D64 8BEC                            mov    bp, sp
2D66 E814DF                          call   TU_TAILLE_CODE
2D69 3700                            hex    3700 ; taille
2D6B 6D2D                            adr    SAUT_PRINCIPAL

2D6D E90000          SAUT_PRINCIPAL  jmp    PRINCIPAL
```



```
2D70 BF6202          PRINCIPAL  mov    di, $0262
2D73 1E                         push   ds
2D74 E8BFEB                     call   TU_POUSSE_REEL
2D77 BF6802                     mov    di, $0268
2D7A 1E                         push   ds
2D7B E8B8EB                     call   TU_POUSSE_REEL
2D7E E850EC                     call   TU_EGAL_REEL
2D81 A27402                     mov    [$0274], al

2D84 BF6202                     mov    di, $0262
2D87 1E                         push   ds
2D88 E8ABEB                     call   TU_POUSSE_REEL
2D8B BF6802                     mov    di, $0268
2D8E 1E                         push   ds
2D8F E8A4EB                     call   TU_POUSSE_REEL
2D92 E870EC                     call   TU_SUP_EGAL_RE
2D95 A27402                     mov    [$0274], al

2D98 E90000                     jmp    FIN

2D9B 33C0            FIN        xor    ax, ax
2D9D E8A0DE                     call   TU_HALT
2DA0 0000                       hes    0000
```

Fig. 5.28 Comparaison de réels

Notez que:

- toutes les comparaisons utilisent une routine de la librairie qui attend deux valeurs réelles sur la pile et retourne une valeur booléenne dans AL

5.3.2.4 LES CONSTANTES ET VALEURS LITTERALES REELLES

Voici comment sont traitées les constantes réelles:

```
0  (* 001 reel3 *)
1  (* 10 avr 87 *)
2
3  PROGRAM desassemble_constante_reele;
4    VAR reel1, reel2: REAL;
5    BEGIN
6      reel1:= 1.5;
7      reel1:= reel2+ 1.5;
8    END.
```

Fig. 5.29 Le programme *desassemble_constante_reele*.

Voici le code généré:

```
2D4D E849DD                     call    TU_PARAGRAPHS
2D50 0600                       hex     0600 ; options
2D52 780B                       hex     780B ; cs
2D54 3C15                       hex     3C15 ; ds
2D56 DB02                       hex     DB02 ; code
2D58 2700                       hex     2700 ; globales
2D5A 0004                       hex     0004 ; min piles
2D5C 00A0                       hex     00A0 ; max piles
2D5E 1000                       hex     1000 ; fichiers
2D60 0000                       hex     0000 ; input
2D62 0000                       hex     0000 ; output

2D64 8BEC                       mov     bp, sp
2D66 E814DF                     call    TU_TAILLE_CODE
2D69 3B00                       hex     3B00 ; taille
2D6B 6D2D                       adr     SAUT_PRINCIPAL

2D6D E90000         SAUT_PRINCIPAL  jmp     PRINCIPAL

2D70 BF6202         PRINCIPAL   mov     di, $0262
2D73 1E                         push    ds
2D74 57                         push    di
2D75 E8CDEB                     call    TU_POUSSE_K_RE
2D78 810000000040               hex     810000000040
2D7E E8D5EB                     call    TU_POP_REEL_DI

2D81 BF6202                     mov     di, $0262
2D84 1E                         push    ds
2D85 57                         push    di
2D86 BF6802                     mov     di, $0268
2D89 1E                         push    ds
2D8A E8A9EB                     call    TU_POUSSE_REEL
2D8D E8B5EB                     call    TU_POUSSE_K_RE
2D90 810000000040               hex     810000000040
2D96 E8D0EB                     call    TU_ADD_REEL
2D99 E8BAEB                     call    TU_POP_REEL_DI

2D9C E90000                     jmp     FIN

2D9F 33C0           FIN         xor     ax, ax
2DA1 E89CDE                     call    TU_HALT
2DA4 0000                       hex     0000
```

Fig. 5.30 Traitement des constantes réelles

Notez que:

- une routine spéciale de la librairie, TU_POUSSE_K_REELLE est chargée de pousser sur la pile les 6 octets situés dans le code après son appel. La mécanique générale de calcul des expressions est alors la même que celle des variables réelles.

Le traitement serait identique pour des constantes réelles définies en utilisant CONST.

5.3.2.5 LES PROCEDURES STANDARD

Voici un appel de procédure standard:

```
0  (* 001 reel4 *)
1  (* 10 avr 87 *)
2
3  PROGRAM desassemble_procedure_reelle;
4    VAR reel1, reel2: REAL;
5    BEGIN
6      reel1:= ABS(reel2);
7      reel1:= COS(reel2);
8      reel1:= SQR(reel2);
9      reel1:= SQRT(reel2);
10   END.
```

Fig. 5.31 Le programme *desassemble_procedure_reelle*.

dont voici le code:

```
2D4D E849DD                     call    TU_PARAGRAPHS
2D50 0600                       hex     0600 ; options
2D52 780B                       hex     780B ; cs
2D54 3C15                       hex     3C15 ; ds
2D56 DD02                       hex     DD02 ; code
2D58 2700                       hex     2700 ; globales
2D5A 0004                       hex     0004 ; min piles
2D5C 00A0                       hex     00A0 ; max piles
2D5E 1000                       hex     1000 ; fichiers
2D60 0000                       hex     0000 ; input
2D62 0000                       hex     0000 ; output

2D64 8BEC                       mov     bp, sp
2D66 E814DF                     call    TU_TAILLE_CODE
2D69 5700                       hex     5700 ; taille
2D6B 6D2D                       adr     SAUT_PRINCIPAL

2D6D E90000         SAUT_PRINCIPAL  jmp     PRINCIPAL

2D70 BF6202         PRINCIPAL   mov     di, $0262
2D73 1E                         push    ds
2D74 57                         push    di
2D75 BF6802                     mov     di, $0268
2D78 1E                         push    ds
2D79 E8BAEB                     call    TU_POUSSE_REEL
2D7C E84AEC                     call    TU_ABS_REEL
2D7F E8D4EB                     call    TU_POP_REEL_DI
```

```
2D82 BF6202                     mov     di, $0262
2D85 1E                         push    ds
2D86 57                         push    di
2D87 BF6802                     mov     di, $0268
2D8A 1E                         push    ds
2D8B E8A8EB                     call    TU_POUSSE_REEL
2D8E E8DDED                     call    TU_COS
2D91 E8C2EB                     call    TU_POP_REEL_DI

2D94 BF6202                     mov     di, $0262
2D97 1E                         push    ds
2D98 57                         push    di
2D99 BF6802                     mov     di, $0268
2D9C 1E                         push    ds
2D9D E896EB                     call    TU_POUSSE_REEL
2DA0 E8CAEC                     call    TU_SQR
2DA3 E8B0EB                     call    TU_POP_REEL_DI

2DA6 BF6202                     mov     di, $0262
2DA9 1E                         push    ds
2DAA 57                         push    di
2DAB BF6802                     mov     di, $0268
2DAE 1E                         push    ds
2DAF E884EB                     call    TU_POUSSE_REEL
2DB2 E85BED                     call    TU_SQRT
2DB5 E89EEB                     call    TU_POP_REEL_DI

2DB8 E90000                     jmp     FIN

2DBB 33C0               FIN     xor     ax, ax
2DBD E880DE                     call    TU_HALT
2DC0 0000                       hex     0000
```

Fig. 5.32 Les appels de procédures standard

Notez que:

- les fonctions standard utilisent toutes une routine particulière de la librairie, qui attend une valeur réelle sur la pile et y laissent le résultat du calcul.

5.3.2.6 READ ET WRITE DE REELS

A présent lisons et écrivons quelques réels:


```
0  (* 001 reel5 *)
1  (* 10 avr 87 *)
2
3  PROGRAM desassemble_read_reel;
4    VAR reel1: REAL;
5    BEGIN
6      READ(reel1);
7      WRITE(reel1);
8      WRITE(reel1: $11: $22);
9    END.
```

Fig. 5.33 Le programme *desassemble_read_reel*.

Voici comment cela est codé:

```
2D4D E849DD                     call    TU_PARAGRAPHS
2D50 0600                       hex     0600 ; options
2D52 780B                       hex     780B ; cs
2D54 3C15                       hex     3C15 ; ds
2D56 DC02                       hex     DC02 ; code
2D58 2700                       hex     2700 ; globales
2D5A 0004                       hex     0004 ; min piles
2D5C 00A0                       hex     00A0 ; max piles
2D5E 1000                       hex     1000 ; fichiers
2D60 0000                       hex     0000 ; input
2D62 0000                       hex     0000 ; output

2D64 8BEC                       mov     bp, sp
2D66 E814DF                     call    TU_TAILLE_CODE
2D69 4A00                       hex     4A00 ; taille
2D6B 6D2D                       adr     SAUT_PRINCIPAL

2D6D E90000             SAUT_PRINCIPAL  jmp     PRINCIPAL

2D70 E8E2F7             PRINCIPAL       call    TU_READ_DEFAUT
2D73 BF6202                     mov     di, $0262
2D76 1E                         push    ds
2D77 E8E5F9                     call    TU_READ_REEL
2D7A E856E2                     call    TU_IO_CHECK

2D7D E8A2F7                     call    TU_WRITE_DEFAUT
2D80 BF6202                     mov     di, $0262
2D83 1E                         push    ds
2D84 E8AFEB                     call    TU_POUSSE_REEL
2D87 B81200                     mov     ax, $0012
2D8A 50                         push    ax
2D8B B8FFFF                     mov     ax, $FFFF
2D8E E827FB                     call    TU_WRITE_REEL
2D91 E83FE2                     call    TU_IO_CHECK
```

```
2D94 E88BF7                    call   TU_WRITE_DEFAUT
2D97 BF6202                    mov    di, $0262
2D9A 1E                        push   ds
2D9B E898EB                    call   TU_POUSSE_REEL
2D9E B81100                    mov    ax, $0011
2DA1 50                        push   ax
2DA2 B82200                    mov    ax, $0022
2DA5 E810FB                    call   TU_WRITE_REEL
2DA8 E828E2                    call   TU_IO_CHECK

2DAB E90000                    jmp    FIN

2DAE 33C0              FIN     xor    ax, ax
2DB0 E88DDE                    call   TU_HALT
2DB3 0000                      hex    0000
```

Fig. 5.34 Lecture et écriture de rééls.

Notez que:

- la lecture s'effectuant en plaçant sur la pile le segment de la variable réelle (ici DS), dans DI son décalage et en utilisant une routine spéciale, TU_READ_REEL
- l'écriture s'effectue en poussant la valeur réelle sur la pile, puis le nombre de caractères total (par défaut $12), et en plaçant dans AX le nombre de décimales. TU_WRITE_REEL se charge du reste.

5.3.2.7 LES REELS ET LES ENTIERS

Voici un programme qui montre comment sont résolues les conversions implicites entre les entiers et les réels:

```
 0  (* 001 reel6 *)
 1  (* 10 avr 87 *)
 2
 3  PROGRAM desassemble_reel_entier;
 4    VAR reel1, reel2: REAL;
 5        entier: INTEGER;
 6        octet: BYTE;
 7    BEGIN
 8      reel1:= entier;
 9      reel1:= $2F;
10      reel1:= octet;
11
12      reel1:= entier* 1.5;
13      reel1:= 1.5* entier;
14    END.
```

Fig. 5.35 Le programme *désassemble_reel_entier*.

Voici le code de ce programme:

```
2D4D E849DD                    call   TU_PARAGRAPHS
2D50 0600                      hex    0600 ; options
2D52 780B                      hex    780B ; cs
2D54 3C15                      hex    3C15 ; ds
2D56 DE02                      hex    DE02 ; code
2D58 2800                      hex    2800 ; globales
2D5A 0004                      hex    0004 ; min piles
2D5C 00A0                      hex    00A0 ; max piles
2D5E 1000                      hex    1000 ; fichiers
2D60 0000                      hex    0000 ; input
2D62 0000                      hex    0000 ; output

2D64 8BEC                      mov    bp, sp
2D66 E814DF                    call   TU_TAILLE_CODE
2D69 7000                      hex    7000 ; taille
2D6B 6D2D                      adr    SAUT_PRINCIPAL

2D6D E90000            SAUT_PRINCIPAL  jmp    PRINCIPAL

2D70 BF6202            PRINCIPAL      mov    di, $0262
2D73 1E                        push   ds
2D74 57                        push   di
2D75 A16E02                    mov    ax, [$026E]
2D78 E877ED                    call   TU_FLOTTE_ENTI
2D7B E8D8EB                    call   TU_POP_REEL_DI

2D7E BF6202                    mov    di, $0262
2D81 1E                        push   ds
2D82 57                        push   di
2D83 B82F00                    mov    ax, $002F
2D86 E869ED                    call   TU_FLOTTE_ENTI
2D89 E8CAEB                    call   TU_POP_REEL_DI

2D8C BF6202                    mov    di, $0262
2D8F 1E                        push   ds
2D90 57                        push   di
2D91 A07002                    mov    al, [$0270]
2D94 32E4                      xor    ah, ah
2D96 E859ED                    call   TU_FLOTTE_ENTI
2D99 E8BAEB                    call   TU_POP_REEL_DI

2D9C BF6202                    mov    di, $0262
2D9F 1E                        push   ds
2DA0 57                        push   di
2DA1 A16E02                    mov    ax, [$026E]
2DA4 50                        push   ax
2DA5 E89DEB                    call   TU_POUSSE_K_RE
2DA8 810000000040              hex    810000000040
2DAE E84AED                    call   TU_FLOTTE_PREC
2DB1 E8DFEB                    call   TU_MULTIPL_REE
2DB4 E89FEB                    call   TU_POP_REEL_DI
```

```
2DB7 BF6202                        mov     di, $0262
2DBA 1E                            push    ds
2DBB 57                            push    di
2DBC E886EB                        call    TU_POUSSE_K_RE
2DBF 810000000040                  hex     810000000040
2DC5 A16E02                        mov     ax, [$026E]
2DC8 E827ED                        call    TU_FLOTTE_ENTI
2DCB E8C5EB                        call    TU_MULTIPL_REE
2DCE E885EB                        call    TU_POP_REEL_DI

2DD1 E90000                        jmp     FIN

2DD4 33C0                  FIN     xor     ax, ax
2DD6 E867DE                        call    TU_HALT
2DD9 0000                          hex     0000
```

Fig. 5.36 Le code du programme *désassemble_reel_entier*.

Notez que:

- la conversion d'un entier au format réel (sa flottaison) est réalisée en plaçant sa valeur dans AX et en invoquant la routine TU_FLOTTE_ENTIE. Ceci est valable pour les variables entières, les littéraux et les constantes.
- lorsque le premier opérateur est entier, il est poussé sur la pile, le second est poussé, et TU_FLOTTE_PRECE est invoqué pour convertir le premier entier au format réel
- lorsque le code contient un littéral réel, la routine TU_TRAITE_K_REEL est utilisée pour pousser ce littéral sur la pile
- lorsque la variable entière est comprise dans l'intervalle 0..255, AH est mis à zéro pour pouvoir effectuer des calculs sur 16 bits.

Mentionnons aussi que:

- certaines versions de TURBO PASCAL n'effectuaient pas correctement la flottaison d'une constante entière placée en premier:

  reel:= 3* *reel2;*

  devait être remplacé par:

  reel:= 3.0* *reel2*

- l'utilisation de réels est impérative lorsqu'une opération risque de dépasser **MAXINT**. Nous sommes alors obligés de travailler en réel, ou au moins de tester que 32767.0 n'est pas atteint. Les réels sont aussi utilisés pour obtenir des valeurs dans l'intervalle 0..65535, plutôt que de travailler avec des valeurs négative d'entiers. Ces valeurs négatives présentent aussi l'inconvénient de trier les valeur négatives avant les valeurs positives. Dans le cas du tri, la soustraction de $8000 de toutes les valeurs entières résoud le problème.

5.3.2.8 RESUME

Tous les traitements des réels sont effectués en utilisant la pile et des routines de traitement de la librairie:

- la récupération d'une valeur pousse le segment de la variable sur la pile et place son décalage dans DI
- l'affectation dépile le réel et le stocke à l'adresse segmentée située sous le réel
- les routines arithmétiques traitent le (ou les) réels situés sur la pile et y placent le résultat

5.3.3 LE TYPE ENUMERE

Voici un programme qui traite des types énumérés:

```
 0  (* 001 enum *)
 1  (* 10 avr 87 *)
 2
 3  (*$r+*)
 4  PROGRAM desassemble_enumere;
 5    TYPE t_jour= (lundi, mardi, mercredi, jeudi);
 6    VAR enumere: t_jour;
 7    BEGIN
 8      enumere:= mercredi;
 9      enumere:= t_jour(2);
10      enumere:= SUCC(enumere);
11    END.
```

Fig. 5.37 Le programme *desassemble_enumere*.

et voici le code:

```
2D4D E849DD                        call    TU_PARAGRAPHS
2D50 0600                          hex     0600 ; options
2D52 7B06                          hex     7B06 ; cs
2D54 3F10                          hex     3F10 ; ds
2D56 DB02                          hex     DB02 ; code
2D58 2700                          hex     2700 ; globales
2D5A 0004                          hex     0004 ; min piles
2D5C 00A0                          hex     00A0 ; max piles
2D5E 1000                          hex     1000 ; fichiers
2D60 0000                          hex     0000 ; input
2D62 0000                          hex     0000 ; output
```

```
2D64 8BEC                          mov   bp, sp
2D66 E814DF                        call  TU_TAILLE_CODE
2D69 3F00                          hex   3F00 ; taille
2D6B 6D2D                          adr   SAUT_PRINCIPAL

2D6D E90000           SAUT_PRINCIPAL jmp   PRINCIPAL

2D70 B80200           PRINCIPAL    mov   ax, $0002
2D73 B90000                        mov   cx, $0000
2D76 BA0300                        mov   dx, $0003
2D79 E8F8E1                        call  TU_RANGE_CHECK
2D7C A26202                        mov   [$0262], al

2D7F B80300                        mov   ax, $0002
2D82 B90000                        mov   cx, $0000
2D85 BA0300                        mov   dx, $0003
2D88 E8E9E1                        call  TU_RANGE_CHECK
2D8B A26202                        mov   [$0262], al

2D8E A06202                        mov   al, [$0262]
2D91 32E4                          xor   ah, ah
2D93 40                            inc   ax
2D94 B90000                        mov   cx, $0000
2D97 BA0300                        mov   dx, $0003
2D9A E8D7E1                        call  TU_RANGE_CHECK
2D9D A26202                        mov   [$0262], al
2DA0 E90000                        jmp   FIN

2DA3 33C0             FIN          xor   ax, ax
2DA5 E898DE                        call  TU_HALT
```

Fig. 5.38 Le code du programme *desassemble_enumere*.

Notez que:

- le code ne traite directement la valeur ordinale
- à ce niveau, le type énuméré est considéré comme un intervalle 0..dernière_valeur_ordinale, et le code effectue les vérifications de bornes
- le code généré par le surtypage est identique à celui généré par l'affectation d'une valeur énumérée

RESUME

> Le type énuméré est traité par les instructions 8088 comme une valeur numérique appartenant à l'intervalle 0..dernière_valeur_ordinale.

5.3.4 LE TYPE BOOLEEN

5.3.4.1 AFFECTATION DE BOOLEENS

Voici un programme affectant des valeurs à une variable booléenne:

```
0  (* 001 bool1 *)
1  (* 10 avr 87 *)
2
3  (*$r+*)
4  PROGRAM desassemble_affectation_booleens;
5    VAR booleen, booleen1, booleen2: BOOLEAN;
6        entier: INTEGER;
7    BEGIN
8      booleen:= FALSE;
9      booleen:= TRUE;
10     booleen1:= booleen2;
11     booleen:= entier> $77;
12   END.
```

Fig. 5.39 Le programme *desassemble_affecte_booleen*.

et voici le code généré:

```
2D4D E849DD                        call  TU_PARAGRAPHS
2D50 0600                          hex   0600 ; options
2D52 780B                          hex   780B ; cs
2D54 3C15                          hex   3C15 ; ds
2D56 DD02                          hex   DD02 ; code
2D58 2700                          hex   2700 ; globales
2D5A 0004                          hex   0004 ; min piles
2D5C 00A0                          hex   00A0 ; max piles
2D5E 1000                          hex   1000 ; fichiers
2D60 0000                          hex   0000 ; input
2D62 0000                          hex   0000 ; output

2D64 8BEC                          mov   bp, sp
2D66 E814DF                        call  TU_TAILLE_CODE
2D69 5600                          hex   5600 ; taille
2D6B 6D2D                          adr   SAUT_PRINCIPAL

2D6D E90000           SAUT_PRINCIPAL jmp   PRINCIPAL

2D70 B80100           PRINCIPAL    mov   ax, $0001
2D73 B90000                        mov   cx, $0000
2D76 BA0100                        mov   dx, $0001
2D79 E8F8E1                        call  TU_RANGE_CHECK
2D7C A26202                        mov   [$0262], al
```

```
2D7F B80100                     mov     ax, $0000
2D82 B90000                     mov     cx, $0000
2D85 BA0100                     mov     dx, $0001
2D88 E8E9E1                     call    TU_RANGE_CHECK
2D8B A26202                     mov     [$0262], al

2D8E A06402                     mov     al, [$0264]
2D91 32E4                       xor     ah, ah
2D93 B90000                     mov     cx, $0000
2D96 BA0100                     mov     dx, $0001
2D99 E8D8E1                     call    TU_RANGE_CHECK
2D9C A26302                     mov     [$0263], al

2D9F A16502                     mov     ax, [$0265]
2DA2 3D7700                     cmp     ax, $0077
2DA5 B80100                     mov     ax, $0001
2DA8 7F01                       jnle    $2DAB
2DAA 48                         dec     ax
2DAB B90000             LR_2DAB mov     cx, $0000
2DAE BA0100                     mov     dx, $0001
2DB1 E8C0E1                     call    TU_RANGE_CHECK
2DB4 A26202                     mov     [$0262], al

2DB7 E90000                     jmp     FIN

2DBA 33C0               FIN     xor     ax, ax
2DBC E881DE                     call    TU_HALT
2DBF 0000                       hex     0000
```

Fig. 5.40 L'affectation de booléens

Notez que:

- fondamentalement le type booléen est traité comme un type énuméré (**FALSE**, **TRUE**)
- par conséquent des vérifications de bornes sont effectuées avant chaque affectation. De ce fait une variable booléenne non initialisée peut provoquer une erreur de débordement d'intervalle lorsque vous travaillez avec l'option R+.
- la constante **FALSE** correspond à la valeur $00 et **TRUE** à la valeur $01
- lorsque nous chargeons une variable booléenne (qui utilise 1 octet), le code met à zéro AH, pour pouvoir effectuer par la suite des opérations sur 16 bits
- pour calculer la valeur d'une expression booléenne résultant de comparaisons, le programme:
  - compare les deux valeurs (ici les valeurs sont scalaires et CMP est utilisé)
  - place la valeur **TRUE** dans AX ($01)
  - si la condition n'est pas réalisée, la valeur de AX est décrémentée, et prend la valeur **FALSE** ($00)

5.3.4.2 ARITHMETIQUE BOOLEENNE

Voici à présent les opérations permises avec des expressions booléennes:

```
 0  (* 001 bool2 *)
 1  (* 10 avr 87 *)
 2
 3  (*$r+*)
 4  PROGRAM desassemble_arithmetique_booleens;
 5    VAR booleen, booleen1, booleen2: BOOLEAN;
 6    BEGIN
 7      booleen:= booleen1 AND booleen2;
 8      booleen:= booleen1 OR booleen2;
 9      booleen:= NOT booleen;
10    END.
```

Fig. 5.41 Le programme *desassemble_arithmetique_booleens*.

et voici son code:

```
2D4D E849DD                     call    TU_PARAGRAPHS
2D50 0600                       hex     0600 ; options
2D52 780B                       hex     780B ; cs
2D54 3C15                       hex     3C15 ; ds
2D56 DD02                       hex     DD02 ; code
2D58 2700                       hex     2700 ; globales
2D5A 0004                       hex     0004 ; min piles
2D5C 00A0                       hex     00A0 ; max piles
2D5E 1000                       hex     1000 ; fichiers
2D60 0000                       hex     0000 ; input
2D62 0000                       hex     0000 ; output

2D64 8BEC                       mov     bp, sp
2D66 E814DF                     call    TU_TAILLE_CODE
2D69 5600                       hex     5600 ; taille
2D6B 6D2D                       adr     SAUT_PRINCIPAL

2D6D E90000             SAUT_PRINCIPAL  jmp     PRINCIPAL

2D70 A06302             PRINCIPAL       mov     al, [$0263]
2D73 32E4                               xor     ah, ah
2D75 50                                 push    ax
2D76 A06402                             mov     al, [$0264]
2D79 32E4                               xor     ah, ah
2D7B 59                                 pop     cx
2D7C 23C1                               and     ax, cx
2D7E B90000                             mov     cx, $0000
2D81 BA0100                             mov     dx, $0001
2D84 E8EDE1                             call    TU_RANGE_CHECK
2D87 A26202                             mov     [$0262], al
```

```
2D8A A06302                     mov     al, [$0263]
2D8D 32E4                       xor     ah, ah
2D8F 50                         push    ax
2D90 A06402                     mov     al, [$0264]
2D93 32E4                       xor     ah, ah
2D95 59                         pop     cx
2D96 0BC1                       or      ax, cx
2D98 B90000                     mov     cx, $0000
2D9B BA0100                     mov     dx, $0001
2D9E E8D3E1                     call    TU_RANGE_CHECK
2DA1 A26202                     mov     [$0262], al

2DA4 A06202                     mov     al, [$0262]
2DA7 32E4                       xor     ah, ah
2DA9 3401                       xor     al, $01
2DAB B90000                     mov     cx, $0000
2DAE BA0100                     mov     dx, $0001
2DB1 E8C0E1                     call    TU_RANGE_CHECK
2DB4 A26202                     mov     [$0262], al
2DB7 E90000                     jmp     FIN

2DBA 33C0               FIN     xor     ax, ax
2DBC E881DE                     call    TU_HALT
2DBF 0000                       hex     0000
```

Fig. 5.42 Les opérations entre booléens.

Notez que:

- les opérations sont directement effectuées avec les valeurs chargées dans les registres.

5.3.4.3 AFFICHAGE DE BOOLEENS

Voici comment est effectué l'affichage d'une expression booléenne:

```
0  (* 001 BOOLEAN *)
1  (* 10 avr 87 *)
2
3  (*$r+*)
4  PROGRAM desassemble_ecris_booleens;
5    VAR booleen, booleen1, booleen2: BOOLEAN;
6    BEGIN
7      WRITE(booleen);
8    END.
```

Fig. 5.43 Le programme *desassemble_ecris_booleen.*

dont voici le code:

```
2D4D E849DD                     call    TU_PARAGRAPHS
2D50 0600                       hex     0600 ; options
2D52 780B                       hex     780B ; cs
2D54 3C15                       hex     3C15 ; ds
2D56 D902                       hex     D902 ; code
2D58 2700                       hex     2700 ; globales
2D5A 0004                       hex     0004 ; min piles
2D5C 00A0                       hex     00A0 ; max piles
2D5E 1000                       hex     1000 ; fichiers
2D60 0000                       hex     0000 ; input
2D62 0000                       hex     0000 ; output

2D64 8BEC                       mov     bp, sp
2D66 E814DF                     call    TU_TAILLE_CODE
2D69 2100                       hex     2100 ; taille
2D6B 6D2D                       adr     SAUT_PRINCIPAL

2D6D E90000             SAUT_PRINCIPAL  jmp     PRINCIPAL

2D70 E8AFF7             PRINCIPAL       call    TU_READ_DEFAUT
2D73 A06202                     mov     al, [$0262]
2D76 32E4                       xor     ah, ah
2D78 50                         push    ax
2D79 B80000                     mov     ax, $0000
2D7C E851FB                     call    TU_WRITE_BOOLE
2D7F E851E2                     call    TU_IO_CHECK

2D82 E90000                     jmp     FIN

2D85 33C0               FIN     xor     ax, ax
2D87 E8B6DE                     call    TU_HALT
2D8A 0000                       hex     0000
```

Fig. 5.44 L'affichage d'une expression booléenne.

Notez que:

- une routine particulière de la librairie est chargée de l'affichage de **TRUE** ou **FALSE**. Ces constantes sont dans la librairie et pourraient être changées en VRAI et FAUX (cf chapitre 7).

5.3.4.4 RESUME

> Le type booléen est traité par les instructions 8088 comme une valeur numérique dans l'intervalle 0..1.
>
> L'affichage d'une valeur booléenne utilise une routine spécifique de la librairie.

5.3.5 LES CARACTERES

5.3.5.1 AFFECTATION DE CARACTERES

Voici un programme qui traite des types **CHAR**:

```
 0  (* 001 car1 *)
 1  (* 11 avr 87 *)
 2
 3  (*$r+*)
 4  PROGRAM desassemble_caractere;
 5    VAR caractere: CHAR;
 6        booleen: BOOLEAN;
 7        entier: INTEGER;
 8    BEGIN
 9      caractere:= 'a';
10      booleen:= caractere= 'a';
11
12      entier:= ORD('a');
13      caractere:= CHR($77);
14    END.
```

Fig. 5.45 Le programme *desassemble_caractere*.

et voici le code généré:

```
2D4D E849DD                          call   TU_PARAGRAPHS
2D50 0600                            hex    0600 ; options
2D52 780B                            hex    780B ; cs
2D54 3C15                            hex    3C15 ; ds
2D56 DC02                            hex    DC02 ; code
2D58 2700                            hex    2700 ; globales
2D5A 0004                            hex    0004 ; min piles
2D5C 00A0                            hex    00A0 ; max piles
2D5E 1000                            hex    1000 ; fichiers
2D60 0000                            hex    0000 ; input
2D62 0000                            hex    0000 ; output

2D64 8BEC                            mov    bp, sp
2D66 E814DF                          call   TU_TAILLE_CODE
2D69 4D00                            hex    4D00 ; taille
2D6B 6D2D                            adr    SAUT_PRINCIPAL

2D6D E90000          SAUT_PRINCIPAL  jmp    PRINCIPAL

2D70 B86100          PRINCIPAL       mov    ax, $0061
2D73 B90000                          mov    cx, $0000
2D76 BAFF00                          mov    dx, $00FF
2D79 E8F8E1                          call   TU_RANGE_CHECK
2D7C A26202                          mov    [$0262], al
```



```
2D7F A06202                          mov    al, [$0262]
2D82 32E4                            xor    ah, ah
2D84 3D6100                          cmp    ax, $0061
2D87 B80100                          mov    ax, $0001
2D8A 7401                            je     $2D8D
2D8C 48                              dec    ax
2D8D B90000          LR_2D8D         mov    cx, $0000
2D90 BA0100                          mov    dx, $0001
2D93 E8DEE1                          call   TU_RANGE_CHECK
2D96 A26302                          mov    [$0263], al

2D99 B86100                          mov    ax, $0061
2D9C A36402                          mov    [$0264], ax

2D9F B87700                          mov    ax, $0077
2DA2 B90000                          mov    cx, $0000
2DA5 BAFF00                          mov    dx, $00FF
2DA8 E8C9E1                          call   TU_RANGE_CHECK
2DAB A26202                          mov    [$0262], al

2DAE E90000                          jmp    FIN

2DB1 33C0            FIN             xor    ax, ax
2DB3 E88ADE                          call   TU_HALT
2DB6 0000                            hex    0000
```

Fig. 5.46 traitement des caractères

Notez que:

- les instructions de traitement interne manipulent essentiellement des valeurs de l'intervalle 0..255. Le type **CHAR** ne sert donc qu'à vérifier la cohérence des opérations effectuées (interdire la multiplication d'un caractère par un autre par exemple).

5.3.5.2 LECTURE ET ECRITURE DE CARACTERES

Voici un programme qui lit et affiche des caractères:

```
 0  (* 001 car2 *)
 1  (* 11 avr 87 *)
 2
 3  (*$r+*)
 4  PROGRAM desassemble_entree_sortie_caractere;
 5    VAR caractere: CHAR;
 6    BEGIN
 7      READ(caractere);
 8      WRITE(caractere);
 9      WRITE(caractere: $77);
10    END.
```

Fig. 5.47 Le programme *desassemble_e_s_caractere*.

Voici le code généré:

```
2D4D E849DD                     call    TU_PARAGRAPHS
2D50 0600                       hex     0600 ; options
2D52 780B                       hex     780B ; cs
2D54 3C15                       hex     3C15 ; ds
2D56 DB02                       hex     DB02 ; code
2D58 2700                       hex     2700 ; globales
2D5A 0004                       hex     0004 ; min piles
2D5C 00A0                       hex     00A0 ; max piles
2D5E 1000                       hex     1000 ; fichiers
2D60 0000                       hex     0000 ; input
2D62 0000                       hex     0000 ; output

2D64 8BEC                       mov     bp, sp
2D66 E814DF                     call    TU_TAILLE_CODE
2D69 4000                       hex     4000 ; taille
2D6B 6D2D                       adr     SAUT_PRINCIPAL

2D6D E90000         SAUT_PRINCIPAL  jmp     PRINCIPAL

2D70 E8E2F7         PRINCIPAL   call    TU_READ_DEFAUT
2D73 BF6202                     mov     di, $0262
2D76 1E                         push    ds
2D77 E8B2F9                     call    TU_READ_CAR
2D7A E856E2                     call    TU_IO_CHECK

2D7D E8A2F7                     call    TU_WRITE_DEFAUT
2D80 A06202                     mov     al, [$0262]
2D83 32E4                       xor     ah, ah
2D85 50                         push    ax
2D86 B80000                     mov     ax, $0000
2D89 E8DBFA                     call    TU_WRITE_CAR
2D8C E844E2                     call    TU_IO_CHECK

2D8F E890F7                     call    TU_WRITE_DEFAUT
2D92 A06202                     mov     al, [$0262]
2D95 32E4                       xor     ah, ah
2D97 50                         push    ax
2D98 B87700                     mov     ax, $0077
2D9B E8C9FA                     call    TU_WRITE_CAR
2D9E E832E2                     call    TU_IO_CHECK

2DA1 E90000                     jmp     FIN

2DA4 33C0           FIN         xor     ax, ax
2DA6 E897DE                     call    TU_HALT
2DA9 0000                       hex     0000
```

Fig. 5.48 Entrée / sortie de caractères

Notez que:

- deux routines spécifiques de la librairie sont chargées d'effectuer les lectures et les affichages de caractères

RESUME

Les caractères sont traités par les instructions 8088 comme des valeurs dans l'intervalle 0..255

5.3.6 RESUME SUR LES TYPES SIMPLES

Pour les types scalaires (entier, énuméré, booléen, caractère) et leurs intervalles (**BYTE** ou autre):

- toute expression est calculée en utilisant AX. Si l'expression est complexe, les valeurs intermédiaires sont poussées sur la pile, puis récupérées dans CX. Le chargement de valeurs se fait directement dans AX, en mettant AH à zéro lorsque la variable n'utilise qu'un seul octet, les calculs étant alors effectués sur des registres de 16 bits.
- des routines spécifiques à chaque type sont utilisées pour effectuer les lectures et les écritures
- tout affectation est effectuée en plaçant le résultat du membre de droite dans AX. Si le membre de gauche est une variable de type intervalle et que l'option R+ est active, une vérification d'intervalle est effectuée.

Pour les réels:

- tous les traitements (affectation, arithmétique, comparaison, lecture et écriture) sont effectués en poussant la valeur du réel sur la pile. Les routines de la librairie se chargent de dépiler le nombre de réels dont elles ont besoin, en replaçant au besoin un résultat sur la pile pour la suite des opérations.

5.4 LES TYPES STRUCTURES

5.4.1 GENERALITES

Les types structurés comprennent:

- les tableaux
- les enregistrements
- les ensembles
- les pointeurs
- les fichiers

Les types structurés comportent deux types d'opérations:

- les opérations globales sur toute une structure. Il s'agit essentiellement de l'affectation:

  tableau1:=tableau2.

  Ces opérations sont en général effectuées par une routine de la librairie.

- les opérations sur les éléments du type

  entier:= tableau[4];

  Les instructions comportant des éléments d'un type structuré doivent donc utiliser des instructions qui récupèrent l'élément dans la structure.

  Deux cas se présentent:

  - si le compilateur dispose de toutes les informations pour calculer l'emplacement de l'élément, l'adresse est calculée au moment de la compilation. Il n'y a alors aucune différence avec une expression comportant une variable du même type que l'élément:

    entier:= tableau[4];
    entier:= entier2;

    utilisent fondamentalement le même code.

  - si l'élément est désigné par des variables, le compilateur génère le code qui permettra le calcul de l'adresse au moment de l'exécution:

    entier:= tableau[indice];

    nécessitera le calcul de l'adresse de la cellule, en utilisant l'adresse de départ du tableau, la taille de la cellule et la valeur de l'indice au moment où l'instruction est exécutée.

    Pour réaliser ces calculs, le code va utiliser certains registres, qu'il faudra prendre soin de sauvegarder dans le cas de calculs emboîtés.

5.4.2 LES TABLEAUX

5.4.2.1 TRAITEMENT DES INDICES CONSTANTS

Voici un programme utilisant des cellules indicées par des constantes:

```
0  (* 001 table1 *)
1  (* 10 avr 87 *)
2
3  (*$r+*)
4  PROGRAM desassemble_indice_constant;
5     VAR tableau: ARRAY[$10..$3F] OF $0080..$0200;
6         entier, indice: INTEGER;
```



```
7  BEGIN
8     tableau[$20]:= $0100;
9     entier:= tableau[$22];
10 END.
```

Fig. 5.49 Le programme *desassemble_indices_constants*.

Et voici le code:

```
2D4D E849DD                    call   TU_PARAGRAPHS
2D50 0600                      hex    0600 ; options
2D52 780B                      hex    780B ; cs
2D54 3C15                      hex    3C15 ; ds
2D56 D902                      hex    D902 ; code
2D58 2D00                      hex    2D00 ; globales
2D5A 0004                      hex    0004 ; min piles
2D5C 00A0                      hex    00A0 ; max piles
2D5E 1000                      hex    1000 ; fichiers
2D60 0000                      hex    0000 ; input
2D62 0000                      hex    0000 ; output

2D64 8BEC                      mov    bp, sp
2D66 E814DF                    call   TU_TAILLE_CODE
2D69 2400                      hex    2400 ; taille
2D6B 6D2D                      adr    SAUT_PRINCIPAL

2D6D E90000          SAUT_PRINCIPAL   jmp    PRINCIPAL

2D70 B80001          PRINCIPAL        mov    ax, $0100
2D73 B98000                           mov    cx, $0080
2D76 BA0002                           mov    dx, $0200
2D79 E8F8E1                           call   TU_RANGE_CHECK
2D7C A38202                           mov    [$0282], ax

2D7F A18602                           mov    ax, [$0286]
2D82 A3C202                           mov    [$02C2], ax

2D85 E90000                           jmp    FIN

2D88 33C0            FIN              xor    ax, ax
2D8A E8B3DE                           call   TU_HALT
2D8D 0000                             hex    0000
```

Fig. 5.50 Utilisation d'indices constants

Notez que:

- le compilateur utilise bien la valeur connue de l'indice pour calculer l'adresse de la cellule.

5.4.2.2 AFFECTATION A UNE CELLULE

Examinons alors le cas plus intéressant de l'affectation d'une valeur à une cellule:

```
 0  (* 001 table2 *)
 1  (* 10 avr 87 *)
 2
 3  PROGRAM desassemble_affecte_tableau;
 4    VAR tableau: ARRAY[$10..$3F] OF $0080..$0200;
 5        entier, indice: INTEGER;
 6    BEGIN
 7      tableau[indice]:= entier;
 8      (*$r+*)
 9      tableau[indice]:= entier;
10    END.
```

Fig. 5.51 Le programme *desassemble_affecte_tableau*.

Voici le code généré:

```
2D4D E849DD                          call    TU_PARAGRAPHS
2D50 0600                            hex     0600 ; options
2D52 780B                            hex     780B ; cs
2D54 3C15                            hex     3C15 ; ds
2D56 DB02                            hex     DB02 ; code
2D58 2D00                            hex     2D00 ; globales
2D5A 0004                            hex     0004 ; min piles
2D5C 00A0                            hex     00A0 ; max piles
2D5E 1000                            hex     1000 ; fichiers
2D60 0000                            hex     0000 ; input
2D62 0000                            hex     0000 ; output

2D64 8BEC                            mov     bp, sp
2D66 E814DF                          call    TU_TAILLE_CODE
2D69 3B00                            hex     3B00 ; taille
2D6B 6D2D                            adr     SAUT_PRINCIPAL

2D6D E90000          SAUT_PRINCIPAL  jmp     PRINCIPAL

2D70 A1C402          PRINCIPAL       mov     ax, [$02C4]
2D73 D1E0                            shl     ax, 1
2D75 97                              xchg    di
2D76 A1C202                          mov     ax, [$02C2]
2D79 89854202                        mov     [di+ $0242], ax
```



```
2D7D A1C402                          mov     ax, [$02C4]
2D80 2D1000                          sub     ax, $0010
2D83 B93000                          mov     cx, $0030
2D86 E8E1E1                          call    TU_INDEX_CHECK
2D89 D1E0                            shl     ax, 1
2D8B 97                              xchg    di
2D8C A1C202                          mov     ax, [$02C2]
2D8F B98000                          mov     cx, $0080
2D92 BA0002                          mov     dx, $0200
2D95 E8DCE1                          call    TU_RANGE_CHECK
2D98 89856202                        mov     [di+ $0262], ax

2D9C E90000                          jmp     FIN

2D9F 33C0                FIN         xor     ax, ax
2DA1 E89CDE                          call    TU_HALT
2DA4 0000                            hex     0000
```

Fig. 5.52 L'affectation à une cellule

Notez que:

- nous avons, pour une fois, compilé en option R-, et en option R+. Le code sans vérifications d'intervalle est en effet plus simple pour comprendre la technique d'accès
- En mode R- (sans vérification d'intervalle ou d'indice), le code effectue le traitement suivant:
  - la valeur de l'indice est chargée dans AX
  - cette valeur est multipliée par la taille de la cellule. Comme la cellule utilise 2 octets, SHL 1 est utilisé pour cette multiplication par 2
  - cette valeur est stockée dans DI
  - la valeur de l'expression à placer dans la cellule est calculée dans AX
  - la valeur de AX est placée dans la cellule en utilisant un adressage indexé par DI.
- en mode R+, le compilateur vérifie en outre:
  - que l'indice est bien dans l'intervalle
  - que la valeur affectée à la cellule est bien dans l'intervalle

Plusieurs remarques:

- comme DI est calculé avant le calcul de l'expression de la cellule, et utilisé à la fin pour l'affectation, il faudra prendre soin de sauvegarder DI chaque fois que le calcul de l'expression modifie DI
- le calcul en mode R- et R+ n'est pas tout à fait identique lorsque l'indice ne commence pas à 0:
  - en mode R- le compilateur modifie le décalage ajouté à DI pour tenir compte de l'indice de départ:

    MOV [DI+ $0242], AX

– en mode R+ ce calcul a déjà été effectué pour réaliser le contrôle d'indice

```
MOV [DI+$0262], AX
```

5.4.2.3 OPTIMISATION DU CALCUL D'INDICE

Le calcul de la position de la cellule est optimisé en fonction de la taille de la cellule. En effet, les multiplications sont relativement coûteuses en temps, et il est plus simple pour les petites valeurs d'effectuer des décalages.

Voici un programme qui met cela en évidence:

```
0  (* 001 table3 *)
1  (* 10 avr 87 *)
2
3  PROGRAM desassemble_calcul_tableau;
4    TYPE string9= STRING[9];
5    VAR tableau_1: ARRAY[$10..$3F] OF BYTE;
6        tableau_2: ARRAY[$10..$3F] OF INTEGER;
7        tableau_9: ARRAY[$10..$3F] OF string9;
8        indice: INTEGER;
9        octet: BYTE;
10       entier: INTEGER;
11       chaine9: string9;
12   BEGIN
13     tableau_1[indice]:= $11;
14     tableau_2[indice]:= entier;
15     tableau_9[indice]:= chaine9;
16   END.
```

Fig. 5.53 Le programme *desassemble_calcul_tableau.*

dont voici le code:

```
2D4D E849DD                call    TU_PARAGRAPHS
2D50 0600                  hex     0600 ; options
2D52 780B                  hex     780B ; cs
2D54 3C15                  hex     3C15 ; ds
2D56 DB02                  hex     DB02 ; code
2D58 4F00                  hex     4F00 ; globales
2D5A 0004                  hex     0004 ; min piles
2D5C 00A0                  hex     00A0 ; max piles
2D5E 1000                  hex     1000 ; fichiers
2D60 0000                  hex     0000 ; input
2D62 0000                  hex     0000 ; output

2D64 8BEC                  mov     bp, sp
2D66 E814DF                call    TU_TAILLE_CODE
2D69 4200                  hex     4200 ; taille
2D6B 6D2D                  adr     SAUT_PRINCIPAL
```



```
2D6D E90000          SAUT_PRINCIPAL  jmp     PRINCIPAL

2D70 A1D204          PRINCIPAL       mov     ax, [$04D2]
2D73 97                              xchg    di
2D74 B81100                          mov     ax, $0011
2D77 88855202                        mov     [di+ $0252], al

2D7B A1D204                          mov     ax, [$04D2]
2D7E D1E0                            shl     ax, 1
2D80 97                              xchg    di
2D81 A1D504                          mov     ax, [$04D5]
2D84 89857202                        mov     [di+ $0272], ax

2D88 A1D204                          mov     ax, [$04D2]
2D8B B90A00                          mov     cx, $000A
2D8E F7E1                            mul     ax, cx
2D90 97                              xchg    di
2D91 81C75202                        add     di, $0252
2D95 1E                              push    ds
2D96 57                              push    di
2D97 BFD704                          mov     di, $04D7
2D9A 1E                              push    ds
2D9B E8E4E3                          call    TU_STRING
2D9E B109                            mov     cl, $09
2DA0 E80EE4                          call    TU_STRING_CHEC

2DA3 E90000                          jmp     $2DA6

2DA6 33C0            LR_2DA6         xor     ax, ax
2DA8 E895DE                          call    TU_HALT
2DAB 0000                            hex     0000
```

Fig. 5.54 L'optimisation du calcul de l'indice

Notez que:

– lorsque la cellule utilise 1 octet, il n'y a aucune multiplication à effectuer
– lorsque la cellule utilise 2 octets, SHL AX, 1 est utilisé.

Sont optimisés de façon similaire:

– les calculs pour les cellules de taille 4 (pointeurs), à l'aide de:

```
SHL AX, 1
SHL AX, 1
```

– les calculs pour les cellules de taille 6 (réels), à l'aide de:

```
SHL AX, 1
MOV CX, AX
SHL AX, 1
ADD AX, CX
```

– pour les autres tailles de cellules, une multiplication est utilisée

5.4.2.4 ACCES A UNE CELLULE DANS UNE EXPRESSION

Voyons à présent comment sont récupérées les cellules d'un tableau. Voici un exemple:

```
 0  (* 001 table4 *)
 1  (* 10 avr 87 *)
 2
 3  PROGRAM desassemble_expression_tableau;
 4    VAR tableau: ARRAY[$10..$3F] OF $0080..$0200;
 5        entier, indice, indice2: INTEGER;
 6    BEGIN
 7      entier:= tableau[indice];
 8
 9      tableau[indice]:= tableau[indice2];
10    END.
```

Fig. 5.55 Le programme *desassemble_expression_tableau.*

Voici le code généré:

```
2D4D E849DD                      call   TU_PARAGRAPHS
2D50 0600                        hex    0600 ; options
2D52 780B                        hex    780B ; cs
2D54 3C15                        hex    3C15 ; ds
2D56 DA02                        hex    DA02 ; code
2D58 2D00                        hex    2D00 ; globales
2D5A 0004                        hex    0004 ; min piles
2D5C 00A0                        hex    00A0 ; max piles
2D5E 1000                        hex    1000 ; fichiers
2D60 0000                        hex    0000 ; input
2D62 0000                        hex    0000 ; output

2D64 8BEC                        mov    bp, sp
2D66 E814DF                      call   TU_TAILLE_CODE
2D69 3200                        hex    3200 ; taille
2D6B 6D2D                        adr    SAUT_PRINCIPAL

2D6D E90000          SAUT_PRINCIPAL jmp  PRINCIPAL

2D70 A1C402          PRINCIPAL   mov    ax, [$02C4]
2D73 D1E0                        shl    ax, 1
2D75 97                          xchg   di
2D76 8B854202                    mov    ax, [di+ $0242]
2D7A A3C202                      mov    [$02C2], ax
```



```
2D7D A1C402                      mov    ax, [$02C4]
2D80 D1E0                        shl    ax, 1
2D82 97                          xchg   di
2D83 57                          push   di
2D84 A1C602                      mov    ax, [$02C6]
2D87 D1E0                        shl    ax, 1
2D89 97                          xchg   di
2D8A 8B854202                    mov    ax, [di+ $0242]
2D8E 5F                          pop    di
2D8F 89854202                    mov    [di+ $0242], ax

2D93 E90000                      jmp    FIN

2D96 33C0                FIN     xor    ax, ax
2D98 E8A5DE                      call   TU_HALT
2D9B 0000                        hex    0000
```

Fig. 5.56 Accès à une cellule dans une expression

Notez que:

- le calcul de l'adresse de la cellule est effectué exactement comme pour l'affectation, en utilisant DI, qui sert à indicer l'adresse de base du tableau
- de ce fait, si le résultat doit être placé dans une cellule dont l'indice est calculé, il faut sauvegarder DI sur la pile avant de calculer l'expression, et récupérer DI pour placer la valeur dans la cellule; C'est ce que démontre le second exemple.

5.4.2.5 TABLEAU A DEUX DIMENSIONS

La mécanique de calcul d'indice est en fait tout à fait générale et peut être utilisée pour des tableaux à plusieurs dimensions. En voici un exemple:

```
 0  (* 001 table5 *)
 1  (* 10 avr 87 *)
 2
 3  PROGRAM desassemble_deux_dimensions;
 4    VAR tableau, tableau1: ARRAY[$10..$2F, $40..$6F] OF INTEGER;
 5        indice1, indice2: INTEGER;
 6    BEGIN
 7      tableau[$20, $50]:= $11;
 8      tableau[indice1, indice2]:= $33;
 9    END.
```

Fig. 5.57 Le programme *desassemble_deux_dimensions.*

Voici le code généré:

```
2D4D E849DD                     call    TU_PARAGRAPHS
2D50 0600                       hex     0600 ; options
2D52 780B                       hex     780B ; cs
2D54 3C15                       hex     3C15 ; ds
2D56 DA02                       hex     DA02 ; taille code
2D58 A701                       hex     A701 ; taille globales
2D5A 0004                       hex     0004 ; min stack/heap
2D5C 00A0                       hex     00A0 ; max stack/heap
2D5E 1000                       hex     1000 ; fichiers
2D60 0000                       hex     0000 ; input
2D62 0000                       hex     0000 ; output

2D64 8BEC                       mov     bp, sp
2D66 E814DF                     call    TU_TAILLE_CODE
2D69 2C00                       hex     2C00 ; taille
2D6B 6D2D                       adr     SAUT_PRINCIPAL

2D6D E90000         SAUT_PRINCIPAL  jmp     PRINCIPAL

2D70 B81100         PRINCIPAL   mov     ax, $0011
2D73 A38208                     mov     [$0882], ax

2D76 A1621A                     mov     ax, [$1A62]
2D79 B96000                     mov     cx, $0060
2D7C F7E1                       mul     ax, cx
2D7E 97                         xchg    di

2D7F A1641A                     mov     ax, [$1A64]
2D82 D1E0                       shl     ax, 1
2D84 03F8                       add     di, ax

2D86 B83300                     mov     ax, $0033
2D89 8985E2FB                   mov     [di+ $FBE2], ax

2D8D E90000                     jmp     FIN

2D90 33C0           FIN         xor     ax, ax
2D92 E8ABDE                     call    TU_HALT
2D95 0000                       hex     0000
```

Fig. 5.58 Un tableau à deux dimensions

Notez que:

- lorsque les indices sont des constantes, l'adresse est calculée à la compilation

- lorsque les deux indices sont des variables:
  - le premier indice calculé dans AX, puis stocké dans DI par:

    XCHG DI

  - les indices suivants sont calculés dans AX, puis ajoutés à DI par:

    ADD DI, AX

Cette mécanique est généralisable à n indices. Elle est utilisée aussi bien pour le calcul de l'adresse d'affectation que l'adresse de récupération d'une valeur. Lorsque DI est utilisé à la fois pour l'affectation et l'accès, la valeur utilisée pour l'affectation devra être poussée sur la pile pour être utilisée après le calcul de l'expression.

5.4.2.6 LES OPERATIONS GLOBALES SUR LES TABLEAUX

Voici des exemples d'affectations globales de tableaux ou de sous-tableaux:

```
0  (* 001 table( *)
1  (* 10 avr 87 *)
2
3  (*$r+*)
4  PROGRAM desassemble_tableau_global;
5    TYPE t_sous_tableau= ARRAY[$40..$6F] OF REAL;
6    VAR tableau, tableau1: ARRAY[$10..$2F] OF INTEGER;
7
8        tableau3, tableau4: ARRAY[$10..$2F, $40..$6F] OF REAL;
9
10       tableau5: ARRAY[$10..$2F] OF t_sous_tableau;
11       tableau6: t_sous_tableau;
12   BEGIN
13     tableau:= tableau1;
14
15     (* refuse par le compilateur: IF tableau<> tableau1 ... *)
16
17     tableau3:= tableau4;
18
19     (* refuse par le compilateur: tableau3[$20]:= tableau6; *)
20
21     tableau5[$20]:= tableau6;
22   END.
```

Fig. 5.59 Le programme *desassemble_tableau_global*.

Voici le code généré:

```
2D4D E849DD                     call    TU_PARAGRAPHS
2D50 0600                       hex     0600 ; options
2D52 780B                       hex     780B ; cs
2D54 3C15                       hex     3C15 ; ds
2D56 DB02                       hex     DB02 ; code
2D58 0107                       hex     0107 ; globales
2D5A 0004                       hex     0004 ; min piles
2D5C 00A0                       hex     00A0 ; max piles
2D5E 1000                       hex     1000 ; fichiers
2D60 0000                       hex     0000 ; input
2D62 0000                       hex     0000 ; output

2D64 8BEC                       mov     bp, sp
2D66 E814DF                     call    TU_TAILLE_CODE
2D69 3C00                       hex     3C00 ; taille
2D6B 6D2D                       adr     SAUT_PRINCIPAL

2D6D E90000         SAUT_PRINCIPAL jmp  PRINCIPAL

2D70 BF6202         PRINCIPAL   mov     di, $0262
2D73 1E                         push    ds
2D74 57                         push    di
2D75 BFA202                     mov     di, $02A2
2D78 1E                         push    ds
2D79 B94000                     mov     cx, $0040
2D7C E854E1                     call    TU_COPIE_STRUC

2D7F BFE202                     mov     di, $02E2
2D82 1E                         push    ds
2D83 57                         push    di
2D84 BFE226                     mov     di, $26E2
2D87 1E                         push    ds
2D88 B90024                     mov     cx, $2400
2D8B E845E1                     call    TU_COPIE_STRUC

2D8E BFE25C                     mov     di, $5CE2
2D91 1E                         push    ds
2D92 57                         push    di
2D93 BFE26E                     mov     di, $6EE2
2D96 1E                         push    ds
2D97 B92001                     mov     cx, $0120
2D9A E836E1                     call    TU_COPIE_STRUC

2D9D E90000                     jmp     FIN

2DA0 33C0           FIN         xor     ax, ax
2DA2 E89BDE                     call    TU_HALT
2DA5 0000                       hex     0000
```

Fig. 5.60 Traitement globaux de tableaux.

Notez que:

- l'affectation globale est effectuée en utilisant la routine TU_AFFECTE_TAB, qui se contente de décaler les octets du premier tableau dans le second, et ceci quelle que soit sa taille.
- les tests d'égalité et de différence ne sont pas permis en TURBO PASCAL. Sans doute parce que les bits inutilisés de certaines données ne sont pas systématiquement mis à zéro.
- le compilateur refuse de même l'affectation de sous-tableaux qui n'utilisent pas le même nom de type.

5.4.2.7 RESUME SUR LES TABLEAUX

L'accès aux données d'un tableau est effectué:

- en calculant directement l'adresse de la cellule au moment de la compilation lorsque l'indice (ou les indices) sont des constantes
- en utilisant DI:
  - comme accumulateur pour le calcul de l'adresse
  - comme indice relatif par rapport à l'adresse de la première cellule

  Dans le cas d'une affectation, DI est sauvegardé sur la pile lorsque l'expression utilise aussi DI
- les vérifications d'indice ou d'intervalle ne sont effectuées que lorsque le mode R+ est utilisé

5.4.3 LE TYPE STRING

5.4.3.1 PARTICULARITES DU TYPE STRING

Bien que le type **STRING** soit fondamentalement un tableau, il joue un rôle particulier du fait:

- de l'indice initial, toujours 0
- de la gestion dynamique de la taille dans la cellule d'indice 0
- des possibilités de comparaisons entre chaînes
- de la présence des valeurs littérales
- des possibilités d'entrée sortie par **READ** et **WRITE**
- de la compatibilité des caractères, considérés comme des chaînes de taille 1, dans les expressions de chaînes
- des nombreuses procédures et fonctions de traitement de chaînes

Nous allons examiner comment le code traite ces aspects spécifiques des chaînes de caractères.

5.4.3.2 AFFECTATION DE CHAINE

Voici un programme qui affecte une chaîne à une variable de type STRING:

```
 0  (* 001 strin1 *)
 1  (* 11 avr 87 *)
 2
 3  (*$r+*)
 4  PROGRAM desassemble_affecte_string;
 5    VAR chaine, chaine2: STRING[5];
 6    BEGIN
 7      chaine:= chaine2;
 8      chaine:= 'ABCD';
 9      chaine:= 'ABCDEFGH';
10    END.
```

Fig. 5.61 Le programme *desassemble_affecte_chaine.*

Voici le code généré:

```
2D4D E849DD                        call   TU_PARAGRAPHS
2D50 0600                          hex    0600 ; options
2D52 780B                          hex    780B ; cs
2D54 3C15                          hex    3C15 ; ds
2D56 DC02                          hex    DC02 ; code
2D58 2700                          hex    2700 ; globales
2D5A 0004                          hex    0004 ; min piles
2D5C 00A0                          hex    00A0 ; max piles
2D5E 1000                          hex    1000 ; fichiers
2D60 0000                          hex    0000 ; input
2D62 0000                          hex    0000 ; output

2D64 8BEC                          mov    bp, sp
2D66 E814DF                        call   TU_TAILLE_CODE
2D69 4800                          hex    4800 ; taille
2D6B 6D2D                          adr    SAUT_PRINCIPAL

2D6D E90000           SAUT_PRINCIPAL jmp    PRINCIPAL

2D70 BF6202           PRINCIPAL    mov    di, $0262
2D73 1E                            push   ds
2D74 57                            push   di
2D75 BF6802                        mov    di, $0268
2D78 1E                            push   ds
2D79 E806E4                        call   TU_POUSSE_STRING
2D7C B105                          mov    cl, $05
2D7E E830E4                        call   TU_POP_STRING
```



```
2D81 BF6202                        mov    di, $0262
2D84 1E                            push   ds
2D85 57                            push   di
2D86 E812E4                        call   TU_POUSSE_K_ST
2D89 04                            hex    04
2D8A 41424344                      asc    'ABCD'
2D8E B105                          mov    cl, $05
2D90 E81EE4                        call   TU_POP_STRING

2D93 BF6202                        mov    di, $0262
2D96 1E                            push   ds
2D97 57                            push   di
2D98 E800E4                        call   TU_POUSSE_K_ST
2D9B 08                            hex    08
2D9C 4142434445464748              asc    'ABCDEFGH'
2DA4 B105                          mov    cl, $05
2DA6 E808E4                        call   TU_POP_STRING

2DA9 E90000                        jmp    FIN

2DAC 33C0                 FIN      xor    ax, ax
2DAE E88FDE                        call   TU_HALT
2DB1 0000                          hex    0000
```

Fig. 5.62 Affectation d'une chaîne

Le code généré est un peu similaire à celui utilisé pour les réels:

- l'adresse segmentée de la chaîne à laquelle sera affectée la valeur est poussée sur la pile
- l'expression est poussée sur la pile
- la routine TU_POP_STRING est chargée de récupérer

Notez que:

- au sommet de la pile se trouve la taille de la chaîne poussée. Cette taille permet de retrouver l'adresse segmentée de la chaîne destinataire
- lorsque l'expression est une chaîne littérale, la routine TU_POUSSE_K_STRING est chargée de placer sur la pile les octets qui la suivent dans le code
- lorsque vous affectez une chaîne de plus grande taille que la chaîne destination, la routine TU_POP_STRING se charge de discrètement tronquer la chaîne. Aucun message ne vous l'indiquera.

5.4.3.3 COMPARAISON DE CHAINES

Voici des comparaisons de chaînes:

```
 0  (* 001 strin2 *)
 1  (* 11 avr 87 *)
 2
 3  (*$r+*)
 4  PROGRAM teste_compare_string;
 5    VAR chaine, chaine2: STRING[5];
 6        booleen: BOOLEAN;
 7    BEGIN
 8      booleen:= chaine= 'AB';
 9      booleen:= chaine> chaine2;
10    END.
```

Fig. 5.63 Le programme *desassemble_compare_string*.

Voici le code:

```
2D4D E849DD                    call   TU_PARAGRAPHS
2D50 0600                      hex    0600 ; options
2D52 780B                      hex    780B ; cs
2D54 3C15                      hex    3C15 ; ds
2D56 DC02                      hex    DC02 ; code
2D58 2700                      hex    2700 ; globales
2D5A 0004                      hex    0004 ; min piles
2D5C 00A0                      hex    00A0 ; max piles
2D5E 1000                      hex    1000 ; fichiers
2D60 0000                      hex    0000 ; input
2D62 0000                      hex    0000 ; output

2D64 8BEC                      mov    bp, sp
2D66 E814DF                    call   TU_TAILLE_CODE
2D69 4800                      hex    4800 ; taille
2D6B 6D2D                      adr    SAUT_PRINCIPAL

2D6D E90000          SAUT_PRINCIPAL   jmp    PRINCIPAL

2D70 BF6202          PRINCIPAL        mov    di, $0262
2D73 1E                               push   ds
2D74 E80BE4                           call   TU_POUSSE_STRING
2D77 E821E4                           call   TU_POUSSE_K_ST
2D7A 02                               hex    02
2D7B 4142                             asc    'AB'
2D7D E8B0E4                           call   TU_EGAL_STRING
2D80 B90000                           mov    cx, $0000
2D83 BA0100                           mov    dx, $0001
2D86 E8EBE1                           call   TU_RANGE_CHECK
2D89 A26E02                           mov    [$026E], al
```



```
2D8C BF6202                    mov    di, $0262
2D8F 1E                        push   ds
2D90 E8EFE3                    call   TU_POUSSE_STRING
2D93 BF6802                    mov    di, $0268
2D96 1E                        push   ds
2D97 E8E8E3                    call   TU_POUSSE_STRING
2D9A E8C3E4                    call   TU_SUP_STRING
2D9D B90000                    mov    cx, $0000
2DA0 BA0100                    mov    dx, $0001
2DA3 E8CEE1                    call   TU_RANGE_CHECK
2DA6 A26E02                    mov    [$026E], al

2DA9 E90000                    jmp    FIN

2DAC 33C0                FIN   xor    ax, ax
2DAE E88FDE                    call   TU_HALT
2DB1 0000                      hex    0000
```

Fig. 5.64 Comparaison de chaînes

Notez que:

- les deux chaînes à comparer sont poussées sur la pile
- la routine adéquate compare les deux chaînes sur la pile et retourne dans AL le résultat du test
- ce résultat est alors placé dans la variable booléenne

5.4.3.4 LECTURE ET ECRITURE DE CHAINES

Voici les routines de lecture et d'écriture de chaînes:

```
 0  (* 001 strin3 *)
 1  (* 11 avr 87 *)
 2
 3  (*$r+*)
 4  PROGRAM desassemble_e_s_string;
 5    VAR chaine: STRING[5];
 6    BEGIN
 7      READ(chaine);
 8      WRITE(chaine);
 9      WRITE('ABCD': 2);
10    END.
```

Fig. 5.65 Le programme *desassemble_e_s_string*.

Et voici son code:

```
2D4D E849DD                     call    TU_PARAGRAPHS
2D50 0600                       hex     0600 ; options
2D52 780B                       hex     780B ; cs
2D54 3C15                       hex     3C15 ; ds
2D56 DB02                       hex     DB02 ; code
2D58 2700                       hex     2700 ; globales
2D5A 0004                       hex     0004 ; min piles
2D5C 00A0                       hex     00A0 ; max piles
2D5E 1000                       hex     1000 ; fichiers
2D60 0000                       hex     0000 ; input
2D62 0000                       hex     0000 ; output

2D64 8BEC                       mov     bp, sp
2D66 E814DF                     call    TU_TAILLE_CODE
2D69 4500                       hex     4500 ; taille
2D6B 6D2D                       adr     SAUT_PRINCIPAL

2D6D E90000             SAUT_PRINCIPAL  jmp     PRINCIPAL

2D70 E8E2F7             PRINCIPAL       call    TU_READ_DEFAUT
2D73 BF6202                             mov     di, $0262
2D76 1E                                 push    ds
2D77 B105                               mov     cl, $05
2D79 E801FA                             call    TU_READ_CHAINE
2D7C E854E2                             call    TU_IO_CHECK

2D7F E8A0F7                             call    TU_WRITE_DEFAUT
2D82 BF6202                             mov     di, $0262
2D85 1E                                 push    ds
2D86 E8F9E3                             call    TU_POUSSE_STRING
2D89 B80000                             mov     ax, $0000
2D8C E861FB                             call    TU_WRITE_CHAIN
2D8F E841E2                             call    TU_IO_CHECK

2D92 E88DF7                             call    TU_WRITE_DEFAUT
2D95 E803E4                             call    TU_POUSSE_K_ST
2D98 04                                 hex     04
2D99 41424344                           asc     'ABCD'
2D9D B80200                             mov     ax, $0002
2DA0 E84DFB                             call    TU_WRITE_CHAIN
2DA3 E82DE2                             call    TU_IO_CHECK

2DA6 E90000                             jmp     FIN

2DA9 33C0               FIN             xor     ax, ax
2DAB E892DE                             call    TU_HALT
2DAE 0000                               hex     0000
```

Fig. 5.66 Lecture et écriture de chaînes

Notez que:

– deux routines spécifiques de la librairie sont utilisées pour la lecture et l'écriture de chaînes

5.4.3.5 PROCEDURES ET FONCTIONS STRING STANDARD

Voici à présent quelques procédures et fonctions manipulant des chaînes:

```
 0  (* 001 strin4 *)
 1  (* 11 avr 87 *)
 2
 3  (*$r+*)
 4  PROGRAM desassemble_fonctions_string;
 5    VAR chaine, chaine2, chaine3: STRING[$20];
 6        entier, resultat: INTEGER;
 7        reel: REAL;
 8    BEGIN
 9      DELETE(chaine, $AA, $BB);
10
11      STR(entier, chaine);
12      VAL(chaine, reel, resultat);
13
14      chaine:= CONCAT(chaine2, chaine3, 'AAA');
15    END.
```

Fig. 5.67 Le programme *desassemble_procedure_string*.

Voici le code généré:

```
2D4D E849DD                     call    TU_PARAGRAPHS
2D50 0600                       hex     0600 ; options
2D52 780B                       hex     780B ; cs
2D54 3C15                       hex     3C15 ; ds
2D56 DE02                       hex     DE02 ; code
2D58 2D00                       hex     2D00 ; globales
2D5A 0004                       hex     0004 ; min piles
2D5C 00A0                       hex     00A0 ; max piles
2D5E 1000                       hex     1000 ; fichiers
2D60 0000                       hex     0000 ; input
2D62 0000                       hex     0000 ; output

2D64 8BEC                       mov     bp, sp
2D66 E814DF                     call    TU_TAILLE_CODE
2D69 6700                       hex     6700 ; taille
2D6B 6D2D                       adr     SAUT_PRINCIPAL

2D6D E90000             SAUT_PRINCIPAL  jmp     PRINCIPAL

2D70 BF6202             PRINCIPAL       mov     di, $0262
2D73 1E                                 push    ds
2D74 57                                 push    di
2D75 B8AA00                             mov     ax, $00AA
2D78 50                                 push    ax
2D79 B8BB00                             mov     ax, $00BB
2D7C E86BE6                             call    TU_DELETE
```

```
2D7F A1C502                     mov     ax, [$02C5]
2D82 50                         push    ax
2D83 B80000                     mov     ax, $0000
2D86 50                         push    ax
2D87 BF6202                     mov     di, $0262
2D8A 1E                         push    ds
2D8B B120                       mov     cl, $20
2D8D E833E0                     call    TU_STR_ENTIER

2D90 BF6202                     mov     di, $0262
2D93 1E                         push    ds
2D94 E8EBE3                     call    TU_POUSSE_STRING
2D97 BFC902                     mov     di, $02C9
2D9A 1E                         push    ds
2D9B 57                         push    di
2D9C BFC702                     mov     di, $02C7
2D9F 1E                         push    ds
2DA0 E899E0                     call    TU_VAL_REEL

2DA3 BF6202                     mov     di, $0262
2DA6 1E                         push    ds
2DA7 57                         push    di
2DA8 BF8302                     mov     di, $0283
2DAB 1E                         push    ds
2DAC E8D3E3                     call    TU_POUSSE_STRING
2DAF BFA402                     mov     di, $02A4
2DB2 1E                         push    ds
2DB3 E8CCE3                     call    TU_POUSSE_STRING
2DB6 E8FAE4                     call    TU_CONCAT
2DB9 E8DFE3                     call    TU_POUSSE_K_ST
2DBC 03                         hex     03
2DBD 414141                     asc     'AAA'
2DC0 E8F0E4                     call    TU_CONCAT
2DC3 B120                       mov     cl, $20
2DC5 E8E9E3                     call    TU_POP_STRING

2DC8 E90000                     jmp     FIN

2DCB 33C0               FIN     xor     ax, ax
2DCD E870DE                     call    TU_HALT
2DD0 0000                       hex     0000
```

Fig. 5.68 procedures et fonctions traitant des chaînes

Notez que:

- toutes les procédures et fonctions sont codées en utilisant le même principe: les paramètres nécessaires à la routine sont soit placés dans des registres, soit poussés sur la pile.
- les procédures VAL et STR peuvent utiliser soit des réels soit des entiers. Le compilateur invoque l'une des 4 routines adéquates. Si vous ne précisez pas le paramètre de résultat, une valeur $0000 est poussée sur la pile.
- Lorsque nous utilisons une fonction dont le résultat est de type STRING (comme CONCAT), l'adresse segmentée est poussée sur la pile, et TU_POP_STRING invoqué pour affecter la valeur.
- CONCAT effectue la concaténation au fur et à mesure, en utilisant les deux chaînes poussées sur la pile avant l'appel de TU_CONCAT. Le résultat de la concaténation est laissé sur la pile, en vue d'une autre concaténation ou d'une utilisation du résultat.

5.4.3.6 RESUME

Les chaînes de caractères:

- sont des tableaux dont l'octet 0 est utilisé pour gérer dynamiquement la taille
- utilisent des routines particulières de la librairie pour effectuer toutes les opérations (affectation, addition et comparaison). Les chaînes à utiliser sont poussées sur la pile, et le résultat récupéré sur la pile.
- peuvent utiliser des procédures et fonctions standard, qui récupèrent leurs paramètres dans les registres ou sur la pile

5.4.3.7 ASSIMILATION DES CARACTERES AUX CHAINES

Pour faciliter certaines opérations sur les chaînes, TURBO PASCAL assimile les caractères à des chaînes de taille 1 (un peu comme le mélange d'entiers dans une expression réelle). Voici un programme qui mélange des caractères et des chaînes:

```
 0  (* 001 strc1 *)
 1  (* 11 avr 87 *)
 2
 3  (*$r+*)
 4  PROGRAM desassemble_string_caractere;
 5    VAR chaine: STRING[5];
 6        caractere: CHAR;
 7        booleen: BOOLEAN;
 8    BEGIN
 9      chaine:= caractere;
10      caractere:= chaine;
11
12      booleen:= chaine< caractere;
13
14      chaine:= 'A';
15    END.
```

Fig. 5.69 Le programme *desassemble_string_caractere*.

Voici le code généré:

```
2D4D E849DD                     call    TU_PARAGRAPHS
2D50 0600                       hex     0600 ; options
2D52 780B                       hex     780B ; cs
2D54 3C15                       hex     3C15 ; ds
```

```
2D56 DE02                       hex     DE02 ; code
2D58 2700                       hex     2700 ; globales
2D5A 0004                       hex     0004 ; min piles
2D5C 00A0                       hex     00A0 ; max piles
2D5E 1000                       hex     1000 ; fichiers
2D60 0000                       hex     0000 ; input
2D62 0000                       hex     0000 ; output

2D64 8BEC                       mov     bp, sp
2D66 E814DF                     call    TU_TAILLE_CODE
2D69 6B00                       hex     6B00 ; taille
2D6B 6D2D                       adr     SAUT_PRINCIPAL

2D6D E90000             SAUT_PRINCIPAL  jmp     PRINCIPAL

2D70 BF6202             PRINCIPAL       mov     di, $0262
2D73 1E                         push    ds
2D74 57                         push    di
2D75 A06802                     mov     al, [$0268]
2D78 32E4                       xor     ah, ah
2D7A 8AE0                       mov     ah, al
2D7C B001                       mov     al, $01
2D7E 50                         push    ax
2D7F B105                       mov     cl, $05
2D81 E82DE4                     call    TU_POP_STRING

2D84 BF6202                     mov     di, $0262
2D87 1E                         push    ds
2D88 E8F7E3                     call    TU_POUSSE_STRING
2D8B E8A5E6                     call    TU_MOV_CAR_STR
2D8E B90000                     mov     cx, $0000
2D91 BAFF00                     mov     dx, $00FF
2D94 E8DDE1                     call    TU_RANGE_CHECK
2D97 A26802                     mov     [$0268], al

2D9A BF6202                     mov     di, $0262
2D9D 1E                         push    ds
2D9E E8E1E3                     call    TU_POUSSE_STRING
2DA1 A06802                     mov     al, [$0268]
2DA4 32E4                       xor     ah, ah
2DA6 8AE0                       mov     ah, al
2DA8 B001                       mov     al, $01
2DAA 50                         push    ax
2DAB E8BEE4                     call    TU_INF_STRING
2DAE B90000                     mov     cx, $0000
2DB1 BA0100                     mov     dx, $0001
2DB4 E8BDE1                     call    TU_RANGE_CHECK
2DB7 A26902                     mov     [$0269], al

2DBA BF6202                     mov     di, $0262
2DBD 1E                         push    ds
2DBE 57                         push    di
2DBF B84100                     mov     ax, $0041
2DC2 8AE0                       mov     ah, al
2DC4 B001                       mov     al, $01
2DC6 50                         push    ax
2DC7 B105                       mov     cl, $05
2DC9 E8E5E3                     call    TU_POP_STRING
```



```
2DCC E90000                     jmp     FIN

2DCF 33C0               FIN     xor     ax, ax
2DD1 E86CDE                     call    TU_HALT
2DD4 0000                       hex     0000
```

Fig. 5.70 Les caractères et les chaînes

Notez que:

- pour affecter un caractère à une chaîne, le code:
  - charge la valeur du caractère dans AX
  - met à zéro AH (inutile ici)
  - place le caractère dans AH
  - place 1 dans AL, ce qui constitue la taille de cette chaîne
  - l'ensemble est poussé sur la pile
- pour affecter une chaîne à un caractère, le code:
  - pousse la chaîne sur la pile
  - utilise une routine spécifique pour dépiler la chaîne dans AX
  - la vérification d'intervalle sera effectuée si vous travaillez en mode R+
- cette mécanique est générale. Elle est utilisée dans les expressions de chaînes (comparaisons, concaténation...), ainsi que pour les valeurs littérales et les constantes de 1 caractère.

RESUME

> Les caractères et les chaînes de 1 caractère sont rendus compatibles, et peuvent être utilisés indifféremment dans les affectations et les expressions de type caractère ou de type chaîne.

5.4.3.8 LES TABLEAUX DE CARACTERES

Les tableaux de caractères, un peu similaires à des chaînes de taille fixe, bénéficient de 3 particularités:

- nous pouvons utiliser des valeurs littérales pour les initialiser
- ils peuvent être lus et écrits par **READ** et **WRITE**.

Et voici le programme correspondant:

```
 0  (* 001 tabc *)
 1  (* 11 avr 87 *)
 2
 3  (*$r+*)
 4  PROGRAM desassemble_tableau_caracters;
 5    VAR tableau_caracteres, tableau_caracteres2: ARRAY[5..9]
OF CHAR;
 6    BEGIN
 7      tableau_caracteres:= 'ABCDE';
 8
 9      READ(tableau_caracteres);
10      WRITE(tableau_caracteres);
11    END.
```

Fig. 5.71 Le programme *desassemble_tableau_caracteres*.

Voici le code correspondant:

```
2D4D E849DD                        call   TU_PARAGRAPHS
2D50 0600                          hex    0600 ; options
2D52 780B                          hex    780B ; cs
2D54 3C15                          hex    3C15 ; ds
2D56 DC02                          hex    DC02 ; code
2D58 2700                          hex    2700 ; globales
2D5A 0004                          hex    0004 ; min piles
2D5C 00A0                          hex    00A0 ; max piles
2D5E 1000                          hex    1000 ; fichiers
2D60 0000                          hex    0000 ; input
2D62 0000                          hex    0000 ; output

2D64 8BEC                          mov    bp, sp
2D66 E814DF                        call   TU_TAILLE_CODE
2D69 4900                          hex    4900 ; taille
2D6B 6D2D                          adr    SAUT_PRINCIPAL

2D6D E90000          SAUT_PRINCIPAL jmp    PRINCIPAL

2D70 BF6202          SAUT_PRINCIPAL mov    di, $0262
2D73 1E                            push   ds
2D74 57                            push   di
2D75 EB05                          jmp_s  $2D7C
2D77 4142434445                    asc    'ABCDE'
2D7C BF772D          LR_2D7C       mov    di, $2D77
2D7F 0E                            push   cs
2D80 B90500                        mov    cx, $0005
2D83 E84DE1                        call   TU_AFFECTE_TAB
```



```
2D86 E8CCF7                        call   TU_READ_DEFAUT
2D89 BF6202                        mov    di, $0262
2D8C 1E                            push   ds
2D8D B105                          mov    cl, $05
2D8F E818FA                        call   TU_READ_ARRAY
2D92 E83EE2                        call   TU_IO_CHECK

2D95 E88AF7                        call   TU_WRITE_DEFAUT
2D98 BF6202                        mov    di, $0262
2D9B 1E                            push   ds
2D9C B105                          mov    cl, $05
2D9E E838E4                        call   TU_ARRAY_TO_ST
2DA1 B80000                        mov    ax, $0000
2DA4 E849FB                        call   TU_WRITE_CHAIN
2DA7 E829E2                        call   TU_IO_CHECK

2DAA E90000                        jmp    FIN

2DAD 33C0                FIN       xor    ax, ax
2DAF E88EDE                        call   TU_HALT
2DB2 0000                          hex    0000
```

Fig. 5.72 Les tableaux de caractères

Notez que:

- comme la valeur littérale n'aura à subir aucun traitement cumulé, il est inutile de la pousser sur la pile (comme c'est fait pour les réels ou les chaînes). Le code transmet simplement le segment (ici CS) et le décalage dans DI. Une routine spécifique se charge d'effectuer le transfert
- la lecture est réalisée par une routine de la librairie qui vérifie que le nombre d'octets lu correspond bien au nombre déclaré
- pour l'affichage, aucune vérification n'est nécessaire. Le tableau est par conséquent poussé sur la pile comme une chaîne et la routine d'affichage des chaînes est utilisée.

RESUME

Les tableaux de caractères:

- peuvent être initialisés par une instruction d'affectation.
- peuvent être lus par **READ** et affichés par **WRITE**

5.4.4 LES ENREGISTREMENTS

5.4.4.1 AFFECTATION ET EXPRESSION D'ENREGISTREMENTS

Voici à présent un programme utilisant des enregistrements:

```
0  (* 001 RECORD *)
1  (* 11 avr 87 *)
2
3  (*$r+*)
4  PROGRAM desassemble_record;
5    VAR entier: INTEGER;
6        enregistrement, enregistrement1: RECORD
7                                           age: INTEGER;
8                                           nom: STRING[$1F];
9                                           numero: INTEGER;
10                                        END;
11   BEGIN
12     entier:= enregistrement.age;
13     enregistrement.numero:= entier;
14     enregistrement:= enregistrement1;
15   END.
```

Fig. 5.73 Le programme *desassemble_record*.

et voici le code:

```
2D4D E849DD                    call   TU_PARAGRAPHS
2D50 0600                      hex    0600 ; options
2D52 7B06                      hex    7B06 ; cs
2D54 3F10                      hex    3F10 ; ds
2D56 DA02                      hex    DA02 ; code
2D58 2B00                      hex    2B00 ; globales
2D5A 0004                      hex    0004 ; min piles
2D5C 00A0                      hex    00A0 ; max piles
2D5E 1000                      hex    1000 ; fichiers
2D60 0000                      hex    0000 ; input
2D62 0000                      hex    0000 ; output

2D64 8BEC                      mov    bp, sp
2D66 E814DF                    call   TU_TAILLE_CODE
2D69 2A00                      hex    2A00 ; taille
2D6B 6D2D                      adr    SAUT_PRINCIPAL

2D6D E90000          SAUT_PRINCIPAL   jmp    PRINCIPAL

2D70 A16402          PRINCIPAL        mov    ax, [$0264]
2D73 A36202                           mov    [$0262], ax
```



```
2D76 A16202                           mov    ax, [$0262]
2D79 A38602                           mov    [$0286], ax

2D7C BF6402                           mov    di, $0264
2D7F 1E                               push   ds
2D80 57                               push   di
2D81 BF8802                           mov    di, $0288
2D84 1E                               push   ds
2D85 B92400                           mov    cx, $0024
2D88 E848E1                           call   TU_COPIE_STRUC

2D8B E90000                           jmp    FIN

2D8E 33C0                FIN          xor    ax, ax
2D90 E8ADDE                           call   TU_HALT
```

Fig. 5.74 Le code du programme *desassemble_record*.

Notez que:

- l'adresse de la zone utilisée est calculée directement (il ne peut pas y avoir de "zone variable", alors que dans un tableau nous pouvons utiliser une variable comme indice)
- l'affectation est réalisée par une simple copie d'octets

5.4.4.2 STRUCTURES DE STRUCTURES

Voici un programme comportant des structures plus complexes:

```
0  (* 001 recrec *)
1  (* 10 avr 87 *)
2
3  (*$r+*)
4  PROGRAM desassemble_record_record;
5    VAR enregistrement: RECORD
6                          nom: STRING[$F];
7                          date: RECORD
8                                  jour, mois: INTEGER;
9                                END;
10                       END;
11       tableau_record: ARRAY[$10..$1F] OF RECORD
12                                           nom: STRING[$1F];
13                                           age: INTEGER;
14                                         END;
15       record_tableau: RECORD
16                         nom: STRING[$3F];
17                         tableau: ARRAY[$30..$5F] OF INTEGER;
18                       END;
19       indice: INTEGER;
```

```
20    BEGIN
21      enregistrement.date.jour:= $11;
22      tableau_record[$18].age:= $22;
23      tableau_record[indice].age:= $33;
24      record_tableau.tableau[$40]:= $44;
25      record_tableau.tableau[indice]:= $55;
26    END.
```

Fig. 5.75 Le programme *desassemble_record_record.*

et voici le code:

```
2D4D E849DD                       call    TU_PARAGRAPHS
2D50 0600                         hex     0600 ; options
2D52 7B06                         hex     7B06 ; cs
2D54 3F10                         hex     3F10 ; ds
2D56 DC02                         hex     DC02 ; code
2D58 5400                         hex     5400 ; globales
2D5A 0004                         hex     0004 ; min piles
2D5C 00A0                         hex     00A0 ; max piles
2D5E 1000                         hex     1000 ; fichiers
2D60 0000                         hex     0000 ; input
2D62 0000                         hex     0000 ; output

2D64 8BEC                         mov     bp, sp
2D66 E814DF                       call    TU_TAILLE_CODE
2D69 5000                         hex     5000 ; taille
2D6B 6D2D                         adr     SAUT_PRINCIPAL

2D6D E90000             SAUT_PRINCIPAL    jmp     PRINCIPAL

2D70 B81100             PRINCIPAL         mov     ax, $0011
2D73 A37202                       mov     [$0272], ax

2D76 B82200                       mov     ax, $0022
2D79 A3A603                       mov     [$03A6], ax

2D7C A13605                       mov     ax, [$0536]
2D7F 2D1000                       sub     ax, $0010
2D82 B91000                       mov     cx, $0010
2D85 E8E2E1                       call    TU_INDEX_CHECK
2D88 B92200                       mov     cx, $0022
2D8B F7E1                         mul     ax, cx
2D8D 97                           xchg    di
2D8E B83300                       mov     ax, $0033
2D91 89859602                     mov     [di+ $0296], ax

2D95 B84400                       mov     ax, $0044
2D98 A3F604                       mov     [$04F6], ax
```



```
2D9B A13605                       mov     ax, [$0536]
2D9E 2D3000                       sub     ax, $0030
2DA1 B93000                       mov     cx, $0030
2DA4 E8C3E1                       call    TU_INDEX_CHECK
2DA7 D1E0                         shl     ax, 1
2DA9 97                           xchg    di
2DAA B85500                       mov     ax, $0055
2DAD 8985D604                     mov     [di+ $04D6], ax

2DB1 E90000                       jmp     FIN

2DB4 33C0               FIN       xor     ax, ax
2DB6 E887DE                       call    TU_HALT
```

Fig. 5.76 Le programme *desassemble_record_record.*

Notez que:

- lorsque le compilateur peut calculer les adresses, il le fait (zones ou indices constants)
- dans le cas de tableau d'enregistrements ou d'enregistrement de tableaux, le calcul de l'indice n'est pas modifié, mais la constante ajoutée à la place occupée par les cases précédentes n'est plus l'adresse du début du tableau mais l'adresse du début du tableau plus le décalage de la zone.

5.4.4.3 L'INSTRUCTION WITH

Voici à présent un programme utilisant **WITH**:

```
0  (* 001 WITH *)
1  (* 10 avr 87 *)
2
3  (*$r+*)
4  PROGRAM desassemble_with;
5    TYPE t_enregistrement= RECORD
6                             age: INTEGER;
7                             nom: STRING[$1F];
8                             numero: INTEGER;
9                           END;
10   VAR entier: INTEGER;
11       enregistrement: t_enregistrement;
12       tableau: ARRAY[$10..$1F] OF t_enregistrement;
13       indice: INTEGER;
14   BEGIN
15     WITH enregistrement DO
16     BEGIN
17       entier:= age;
18       numero:= entier;
19     END;
20
```

```
21      WITH tableau[indice] DO
22      BEGIN
23        age:= $33;
24        numero:= entier;
25        indice:= indice+ 1;
26        numero:= entier;
27      END;
28    END.
```

Fig. 5.77 Le programme *desassemble_with.*

et le code correspondant:

```
2D4D E849DD                          call    TU_PARAGRAPHS
2D50 0600                            hex     0600 ; options
2D52 7B06                            hex     7B06 ; cs
2D54 3F10                            hex     3F10 ; ds
2D56 DD02                            hex     DD02 ; code
2D58 4D00                            hex     4D00 ; globales
2D5A 0004                            hex     0004 ; min piles
2D5C 00A0                            hex     00A0 ; max piles
2D5E 1000                            hex     1000 ; fichiers
2D60 0000                            hex     0000 ; input
2D62 0000                            hex     0000 ; output

2D64 8BEC                            mov     bp, sp
2D66 E814DF                          call    TU_TAILLE_CODE
2D69 5C00                            hex     5C00 ; taille
2D6B 6D2D                            adr     SAUT_PRINCIPAL

2D6D E90000          SAUT_PRINCIPAL  jmp     PRINCIPAL

2D70 A16402          PRINCIPAL       mov     ax, [$0264]
2D73 A36202                          mov     [$0262], ax

2D76 A16202                          mov     ax, [$0262]
2D79 A38602                          mov     [$0286], ax

2D7C A1C804                          mov     ax, [$04C8]
2D7F 2D1000                          sub     ax, $0010
2D82 B91000                          mov     cx, $0010
2D85 E8E2E1                          call    TU_INDEX_CHECK
2D88 B92400                          mov     cx, $0024
2D8B F7E1                            mul     ax, cx
2D8D 97                              xchg    di
2D8E 81C78802                        add     di, $0288
2D92 1E                              push    ds
2D93 57                              push    di

2D94 C47EFC                          les     di, [bp+ $FC]
2D97 B83300                          mov     ax, $0033
2D9A 26                              seg     es
2D9B 8905                            mov     [di], ax
```



```
2D9D C47EFC                          les     di, [bp+ $FC]
2DA0 A16202                          mov     ax, [$0262]
2DA3 26                              seg     es
2DA4 894522                          mov     [di+ $22], ax

2DA7 A1C804                          mov     ax, [$04C8]
2DAA 050100                          add     ax, $0001
2DAD A3C804                          mov     [$04C8], ax

2DB0 C47EFC                          les     di, [bp+ $FC]
2DB3 A16202                          mov     ax, [$0262]
2DB6 26                              seg     es
2DB7 894522                          mov     [di+ $22], ax

2DBA 83C404                          add     sp, $04

2DBD E90000                          jmp     FIN

2DC0 33C0                FIN         xor     ax, ax
2DC2 E87BDE                          call    TU_HALT
```

Fig. 5.78 Le code du programme *desassemble_with.*

Notez que:

- il n'y a aucune différence dans le code généré avec ou sans **WITH** lorsque les adresses des zones ou des cases sont calculables à la compilation
- lorsque **WITH** est employé avec un tableau comportant un indice variable le code est plus complexe:
  - l'indice et la taille de la case sont utilisés pour calculer le décalage de la case, l'adresse du début du tableau est ajoutée (ici $0288), et le tout poussé sous forme d'adresse segmentée sur la pile
  - pour accéder à une zone, les registres ES et DI sont chargés à partir de la pile (par le biais de BP), et la zone est accédée par ES
  - lorsque le with se termine, la pile est libérée des 4 octets de l'adresse segmentée du début de la case
- si nous modifions l'indice au sein du **WITH**, le calcul n'est pas refait. C'est donc l'ancienne cellule qui sera modifiée.

5.4.4.4 RESUME

Pour traiter les zones d'un enregistrement:

- le compilateur calcule directement l'adresse si adresse est calculable (l'enregistrement ne fait partie d'un tableau).
- lorsque l'élément fait partie d'une structure complexe comportant un tableau avec un indice variable, le calcul de l'adresse est effectué comme dans le cas d'un tableau usuel, en ajoutant à l'adresse de la première cellule du tableau le décalage de la zone
- lorsque WITH est utilisé:
  - si aucun indice variable n'est employé, le compilateur calcule directement l'adresse constante de la zone
  - si un indice est employé, l'adresse segmentée de la cellule est stockée sur la pile. ES et DI la récupèrent alors pour accéder aux zones

5.4.5 LES ENSEMBLES

5.4.5.1 AFFECTATION D'UN ENSEMBLE

Voici un programme qui utilise des ensembles;

```
0  (* 001 ens1 *)
1  (* 11 avr 87 *)
2
3  (*$r+*)
4  PROGRAM desassemble_affecte_ensemble;
5    VAR ensemble, ensemble1: SET OF $14..$3C;
6        octet, octet1, octet2: BYTE;
7    BEGIN
8      ensemble:= [];
9      ensemble:= [$30, $32..$34];
10     ensemble:= [octet];
11
12     ensemble:= ensemble1;
13   END.
```

Fig. 5.79 Le programme *desassemble_affecte_ensemble.*

Ce programme est mis en oeuvre par le code suivant:

```
2D4D E849DD                    call   TU_PARAGRAPHS
2D50 0600                      hex    0600 ; options
2D52 780B                      hex    780B ; cs
2D54 3C15                      hex    3C15 ; ds
2D56 DE02                      hex    DE02 ; code
2D58 2800                      hex    2800 ; globales
2D5A 0004                      hex    0004 ; min piles
2D5C 00A0                      hex    00A0 ; max piles
2D5E 1000                      hex    1000 ; fichiers
2D60 0000                      hex    0000 ; input
2D62 0000                      hex    0000 ; output

2D64 8BEC                      mov    bp, sp
2D66 E814DF                    call   TU_TAILLE_CODE
2D69 6600                      hex    6600 ; taille
2D6B 6D2D                      adr    SAUT_PRINCIPAL

2D6D E90000          SAUT_PRINCIPAL   jmp    PRINCIPAL

2D70 BF6202          PRINCIPAL        mov    di, $0262
2D73 1E                        push   ds
2D74 57                        push   di
2D75 E847E7                    call   TU_INITIE_SET
2D78 B90602                    mov    cx, $0206
2D7B E879E7                    call   TU_AFFECTE_SET

2D7E BF6202                    mov    di, $0262
2D81 1E                        push   ds
2D82 57                        push   di
2D83 E839E7                    call   TU_INITIE_SET
2D86 B83000                    mov    ax, $0030
2D89 E845E7                    call   TU_AJ_EL_SET
2D8C B83200                    mov    ax, $0032
2D8F 50                        push   ax
2D90 B83400                    mov    ax, $0034
2D93 E842E7                    call   TU_AJ_INTERV_S
2D96 B90602                    mov    cx, $0206
2D99 E85BE7                    call   TU_AFFECTE_SET

2D9C BF6202                    mov    di, $0262
2D9F 1E                        push   ds
2DA0 57                        push   di
2DA1 E81BE7                    call   TU_INITIE_SET
2DA4 A06E02                    mov    al, [$026E]
2DA7 32E4                      xor    ah, ah
2DA9 E825E7                    call   TU_AJ_EL_SET
2DAC B90602                    mov    cx, $0206
2DAF E845E7                    call   TU_AFFECTE_SET
```

```
2DB2 BF6202                     mov     di, $0262
2DB5 1E                         push    ds
2DB6 57                         push    di
2DB7 BF6802                     mov     di, $0268
2DBA 1E                         push    ds
2DBB B90602                     mov     cx, $0206
2DBE E8CEE6                     call    TU_POUSSE_SET
2DC1 B90602                     mov     cx, $0206
2DC4 E830E7                     call    TU_AFFECTE_SET

2DC7 E90000                     jmp     FIN

2DCA 33C0               FIN     xor     ax, ax
2DCC E871DE                     call    TU_HALT
2DCF 0000                       hex     0000
```

Fig. 5.80 Le code du programme *desassemble_affecte_ensemble*.

Notez que:

- les traitements d'ensembles se font tous sur la pile, en utilisant 32 octets, ce qui correspond au plus grand ensemble en TURBO PASCAL (à raison d'un bit par élément, 32* 8= 256, ce qui permet de traiter des ensembles d'octets ou de caractères du jeu ASCII étendu). Ainsi:
  - TU_INITIE_SET pousse 32 zéros sur la pile
  - TU_AFFECTE_SET affecte à l'ensemble une valeur, en dépilant le nombre d'octets utilisés par l'ensemble. De plus:
    - CH contient le nombre d'octets non utilisés (ici 2 parce que l'ensemble commence à $14: $0..$F ne sont pas nécessaires)
    - CL contient le nombre d'octets utilisés par l'ensemble (ici 6, pour coder les éléments entre $10 et $3F)
  - TU_AJ_EL_SET ajoute un élément aux 256 bits sur la pile
  - TU_AJ_INTERV_SET ajoute un intervalle
- l'affectation de l'ensemble vide consiste donc à simplement pousser les 256 bits de zéro et à dépiler le nombre correct de bits dans l'ensemble actuel
- l'affectation d'un ensemble à un autre est effectué en poussant l'ensemble sur la pile à l'aide d'une routine spécifique, puis à dépiler celui-ci dans le membre de gauche.

5.4.5.2 ARITHMETIQUE SUR LES ENSEMBLES

Voici un programme qui effectue des opérations globales sur les ensembles:


```
0  (* 001 ens2 *)
1  (* 11 avr 87 *)
2
3  (*$r+*)
4  PROGRAM desassemble_arithmetique_ensemble;
5    VAR ensemble, ensemble1, ensemble2: SET OF $14..$3C;
6    BEGIN
7      ensemble:= ensemble1+ ensemble2;
8    END.
```

Fig. 5.81 Le programme *desassemble_arithmetique_ensemble*.

et voici le code généré:

```
2D4D E849DD                     call    TU_PARAGRAPHS
2D50 0600                       hex     0600 ; options
2D52 780B                       hex     780B ; cs
2D54 3C15                       hex     3C15 ; ds
2D56 DA02                       hex     DA02 ; code
2D58 2800                       hex     2800 ; globales
2D5A 0004                       hex     0004 ; min piles
2D5C 00A0                       hex     00A0 ; max piles
2D5E 1000                       hex     1000 ; fichiers
2D60 0000                       hex     0000 ; input
2D62 0000                       hex     0000 ; output

2D64 8BEC                       mov     bp, sp
2D66 E814DF                     call    TU_TAILLE_CODE
2D69 3100                       hex     3100 ; taille
2D6B 6D2D                       adr     SAUT_PRINCIPAL

2D6D E90000             SAUT_PRINCIPAL  jmp     PRINCIPAL

2D70 BF6202             PRINCIPAL       mov     di, $0262
2D73 1E                         push    ds
2D74 57                         push    di
2D75 BF6802                     mov     di, $0268
2D78 1E                         push    ds
2D79 B90602                     mov     cx, $0206
2D7C E810E7                     call    TU_POUSSE_SET
2D7F BF6E02                     mov     di, $026E
2D82 1E                         push    ds
2D83 B90602                     mov     cx, $0206
2D86 E806E7                     call    TU_POUSSE_SET
2D89 E8ECE7                     call    TU_ADD_SET
2D8C B90602                     mov     cx, $0206
2D8F E865E7                     call    TU_AFFECTE_SET

2D92 E90000                     jmp     FIN

2D95 33C0               FIN     xor     ax, ax
2D97 E8A6DE                     call    TU_HALT
2D9A 0000                       hex     0000
```

Fig. 5.82 Le code du programme *desassemble_arithmetique_ensemble*.

Notez que:

- le code pousse sur la pile les deux ensembles, et appelle la routine de la librairie chargée d'effectuer le résultat. Celui-ci est alors dépilé dans l'ensemble destination
- le code pour l'intersection ou la différence est similaire

5.4.5.3 LES TESTS SUR LES ENSEMBLES

Examinons finalement les opérations de comparaisons effectuées sur des ensembles:

```
 0  (* 001 ens3 *)
 1  (* 24 avr 87 *)
 2
 3  (*$r+*)
 4  PROGRAM desassemble_teste_ensemble;
 5    VAR ensemble, ensemble1: SET OF $28..$9F;
 6        booleen: BOOLEAN;
 7    BEGIN
 8      (* refuses par le compilateur:
 9      booleen:= ensemble> [31];
10      booleen:= ensemble< [127];
11      *)
12
13      booleen:= $30 IN ensemble;
14      booleen:= ensemble>= [$31];
15      booleen:= ensemble<= ensemble1;
16    END.
```

Fig. 5.83 Le programme *desassemble_teste_ensemble*.

et voici le code généré:

```
2D4D E849DD                          call   TU_PARAGRAPHS
2D50 0600                            hex    0600 ; options
2D52 780B                            hex    780B ; cs
2D54 3C15                            hex    3C15 ; ds
2D56 DE02                            hex    DE02 ; code
2D58 2900                            hex    2900 ; globales
2D5A 0004                            hex    0004 ; min piles
2D5C 00A0                            hex    00A0 ; max piles
2D5E 1000                            hex    1000 ; fichiers
2D60 0000                            hex    0000 ; input
2D62 0000                            hex    0000 ; output

2D64 8BEC                            mov    bp, sp
2D66 E814DF                          call   TU_TAILLE_CODE
2D69 7100                            hex    7100 ; taille
2D6B 6D2D                            adr    SAUT_PRINCIPAL

2D6D E90000          SAUT_PRINCIPAL  jmp    PRINCIPAL
```



```
2D70 B83000          PRINCIPAL  mov    ax, $0030
2D73 50                         push   ax
2D74 BF6202                     mov    di, $0262
2D77 1E                         push   ds
2D78 B90F05                     mov    cx, $050F
2D7B E811E7                     call   TU_POUSSE_SET
2D7E E823E8                     call   TU_IN_SET
2D81 B90000                     mov    cx, $0000
2D84 BA0100                     mov    dx, $0001
2D87 E8EAE1                     call   TU_RANGE_CHECK
2D8A A28002                     mov    [$0280], al

2D8D BF6202                     mov    di, $0262
2D90 1E                         push   ds
2D91 B90F05                     mov    cx, $050F
2D94 E8F8E6                     call   TU_POUSSE_SET
2D97 E825E7                     call   TU_INITIE_SET
2D9A B83100                     mov    ax, $0031
2D9D E831E7                     call   TU_AJ_EL_SET
2DA0 E8B0E7                     call   TU_SUP_EGAL_SE
2DA3 B90000                     mov    cx, $0000
2DA6 BA0100                     mov    dx, $0001
2DA9 E8C8E1                     call   TU_RANGE_CHECK
2DAC A28002                     mov    [$0280], al

2DAF BF6202                     mov    di, $0262
2DB2 1E                         push   ds
2DB3 B90F05                     mov    cx, $050F
2DB6 E8D6E6                     call   TU_POUSSE_SET
2DB9 BF7102                     mov    di, $0271
2DBC 1E                         push   ds
2DBD B90F05                     mov    cx, $050F
2DC0 E8CCE6                     call   TU_POUSSE_SET
2DC3 E891E7                     call   TU_INF_EGAL_SE
2DC6 B90000                     mov    cx, $0000
2DC9 BA0100                     mov    dx, $0001
2DCC E8A5E1                     call   TU_RANGE_CHECK
2DCF A28002                     mov    [$0280], al

2DD2 E90000                     jmp    FIN

2DD5 33C0            FIN        xor    ax, ax
2DD7 E866DE                     call   TU_HALT
2DDA 0000                       hex    0000
```

Fig. 5.84 Le code du programme *desassemble_teste_ensemble*.

Notez que:

- le test d'appartenance **IN** est effectué:
  - en poussant sur la pile la valeur à tester
  - en poussant l'ensemble sur la pile
  - en appelant la routine TU_IN_SET

  Le résultat de la comparaison est retourné dans AX.

– Les comparaisons d'ensemble sont effectuées comme l'arithmétique, en poussant sur la pile les deux ensembles à comparer

Remarquez aussi que les ensembles littéraux (par exemple [5..7]) sont recalculés chaque fois que l'instruction est exécutée.

5.4.5.4 RESUME

> Les traitements sur les ensembles:
>
> – sont tous effectués en poussant l'ensemble (ou les ensembles) sur la pile
> – ces calculs sur la pile utilisent le nombre d'octets correspondant au plus grand ensemble possible (256 bits, soit 32 octets)
> – l'affectation dépile le nombre d'octets correspondant à la définition de l'ensemble

5.4.6 LES POINTEURS

5.4.6.1 AFFECTATION DE POINTEURS

Voici un programme qui utilise des ensembles;

```
0  (* 001 xpt1 *)
1  (* 11 avr 87 *)
2
3  (*$r+*)
4  PROGRAM desassemble_pointeurs;
5    VAR pointeur, pointeur2: ^INTEGER;
6        booleen: BOOLEAN;
7    BEGIN
8      pointeur:= pointeur2;
9      pointeur:= NIL;
10
11     booleen:= pointeur= pointeur2;
12     booleen:= pointeur<> NIL;
13   END.
```

Fig. 5.85 Le programme *desassemble_pointeur*.

et voici le code:

```
2D4D E849DD                call    TU_PARAGRAPHS
2D50 0600                  hex     0600 ; options
2D52 780B                  hex     780B ; cs
2D54 3C15                  hex     3C15 ; ds
2D56 DD02                  hex     DD02 ; code
2D58 2700                  hex     2700 ; globales
2D5A 0004                  hex     0004 ; min piles
2D5C 00A0                  hex     00A0 ; max piles
2D5E 1000                  hex     1000 ; fichiers
2D60 0000                  hex     0000 ; input
2D62 0000                  hex     0000 ; output

2D64 8BEC                  mov     bp, sp
2D66 E814DF                call    TU_TAILLE_CODE
2D69 6300                  hex     6300 ; taille
2D6B 6D2D                  adr     SAUT_PRINCIPAL

2D6D E90000      SAUT_PRINCIPAL    jmp     PRINCIPAL

2D70 C4066602    PRINCIPAL         les     ax, [$0266]
2D74 8CC2                  mov     dx, es
2D76 A36202                mov     [$0262], ax
2D79 89166402              mov     [$0264], dx

2D7D 33C0                  xor     ax, ax
2D7F 33D2                  xor     dx, dx
2D81 A36202                mov     [$0262], ax
2D84 89166402              mov     [$0264], dx

2D88 C4066202              les     ax, [$0262]
2D8C 8CC2                  mov     dx, es
2D8E 52                    push    dx
2D8F 50                    push    ax
2D90 C4066602              les     ax, [$0266]
2D94 8CC2                  mov     dx, es
2D96 5B                    pop     bx
2D97 59                    pop     cx
2D98 E854E8                call    TU_EGAL_PTEUR
2D9B B90000                mov     cx, $0000
2D9E BA0100                mov     dx, $0001
2DA1 E8D0E1                call    TU_RANGE_CHECK
2DA4 A26A02                mov     [$026A], al

2DA7 C4066202              les     ax, [$0262]
2DAB 8CC2                  mov     dx, es
2DAD 52                    push    dx
2DAE 50                    push    ax
2DAF 33C0                  xor     ax, ax
2DB1 33D2                  xor     dx, dx
2DB3 5B                    pop     bx
2DB4 59                    pop     cx
2DB5 E846E8                call    TU_DIFF_PTEUR
2DB8 B90000                mov     cx, $0000
2DBB BA0100                mov     dx, $0001
2DBE E8B3E1                call    TU_RANGE_CHECK
2DC1 A26A02                mov     [$026A], al
```

```
2DC4 E90000                     jmp     FIN

2DC7 33C0               FIN     xor     ax, ax
2DC9 E874DE                     call    TU_HALT
2DCC 0000                       hex     0000
```

Fig. 5.86 Le code du programme *desassemble_pointeur*.

Notez que:

- pour affecter une valeur à un pointeur, le code place la valeur du pointeur dans AX et DX. Si le pointeur est une variable, LES AX est utilisé. Dans le cas de NIL, AX et DX sont simplement initialisés à la valeur zéro.
- la valeur NIL correspond ainsi bien à une adresse $0000:$0000

- les comparaisons de pointeurs se font en plaçant dans AX, DX, la valeur du premier pointeur, et dans BX et CX la valeur du second. Les routines de comparaison (égalité et différence) sont alors invoquées

5.4.6.2 LES PROCEDURES DE TRAITEMENT DE POINTEURS

Voici un programme utilisant des procédures ou des fonctions avec des pointeurs:

```
 0  (* 001 xptr2 *)
 1  (* 11 avr 87 *)
 2
 3  (*$r+*)
 4  PROGRAM desassemble_procedures_pointeurs;
 5    VAR pointeur: ^INTEGER;
 6        paragraphes: INTEGER;
 7        booleen: BOOLEAN;
 8    BEGIN
 9      NEW(pointeur);
10
11      paragraphes:= MEMAVAIL;
12
13      pointeur:= PTR($AAAA, $BBBB);
14    END.
```

Fig. 5.87 Le programme *desassemble_procedures_pointeurs*.

Voici le code généré:


```
2D4D E849DD                     call    TU_PARAGRAPHS
2D50 0600                       hex     0600 ; options
2D52 780B                       hex     780B ; cs
2D54 3C15                       hex     3C15 ; ds
2D56 DA02                       hex     DA02 ; code
2D58 2700                       hex     2700 ; globales
2D5A 0004                       hex     0004 ; min piles
2D5C 00A0                       hex     00A0 ; max piles
2D5E 1000                       hex     1000 ; fichiers
2D60 0000                       hex     0000 ; input
2D62 0000                       hex     0000 ; output

2D64 8BEC                       mov     bp, sp
2D66 E814DF                     call    TU_TAILLE_CODE
2D69 2E00                       hex     2E00 ; taille
2D6B 6D2D                       adr     SAUT_PRINCIPAL

2D6D E90000             SAUT_PRINCIPAL  jmp     PRINCIPAL

2D70 BF6202             PRINCIPAL       mov     di, $0262
2D73 1E                         push    ds
2D74 B90200                     mov     cx, $0002
2D77 E82ED8                     call    TU_NEW

2D7A E818DA                     call    TU_MEMAVAIL
2D7D A36602                     mov     [$0266], ax

2D80 B8AAAA                     mov     ax, $AAAA
2D83 50                         push    ax
2D84 B8BBBB                     mov     ax, $BBBB
2D87 5A                         pop     dx
2D88 A36202                     mov     [$0262], ax
2D8B 89166402                   mov     [$0264], dx

2D8F E90000                     jmp     FIN

2D92 33C0               FIN     xor     ax, ax
2D94 E8A9DE                     call    TU_HALT
2D97 0000                       hex     0000
```

Fig. 5.88 Les procédures de pointeurs

Notez que:

- les procédures comme **NEW, MARK, DISPOSE...** placent sur la pile le segment du pointeur (ici DS), dans DI le décalage du pointeur dans ce segment. Les autres paramètres sont placés dans les registres (la taille de la variable dans CX pour **NEW**), et la routine correspondante de la librairie est invoquée.
- **MEMAVAIL** retourne simplement dans AX le nombre de paragraphes disponibles. Ce calcul est effectué en utilisant les globales de TURBO.
- **PTR** utilise AX et DX, exactement comme l'affectation vue précédemment.

5.4.6.3 AFFECTATION A LA VARIABLE POINTEE

Voici comment une valeur est placée dans une variable pointée:

```
 0  (* 001 ptr3 *)
 1  (* 11 avr 87 *)
 2
 3  (*$r+*)
 4  PROGRAM desassemble_pointe;
 5    TYPE string9= STRING[9];
 6    VAR pt_octet: ^BYTE;
 7        pt_entier: ^INTEGER;
 8        pt_reel: ^REAL;
 9        pt_string9: ^string9;
10        chaine9: string9;
11    BEGIN
12      pt_octet^:= $22;
13      pt_entier^:= $1234;
14      pt_reel^:= 1.5;
15      pt_string9^:= chaine9;
16    END.
```

Fig. 5.89 Le programme *desassemble_pointé*.

Voici le code correspondant:

```
2D4D E849DD                    call   TU_PARAGRAPHS
2D50 0600                      hex    0600 ; options
2D52 780B                      hex    780B ; cs
2D54 3C15                      hex    3C15 ; ds
2D56 DC02                      hex    DC02 ; code
2D58 2800                      hex    2800 ; globales
2D5A 0004                      hex    0004 ; min piles
2D5C 00A0                      hex    00A0 ; max piles
2D5E 1000                      hex    1000 ; fichiers
2D60 0000                      hex    0000 ; input
2D62 0000                      hex    0000 ; output

2D64 8BEC                      mov    bp, sp
2D66 E814DF                    call   TU_TAILLE_CODE
2D69 5000                      hex    5000 ; taille
2D6B 6D2D                      adr    SAUT_PRINCIPAL

2D6D E90000          SAUT_PRINCIPAL   jmp    PRINCIPAL

2D70 C43E6202        PRINCIPAL        les    di, [$0262]
2D74 B82200                    mov    ax, $0022
2D77 B90000                    mov    cx, $0000
2D7A BAFF00                    mov    dx, $00FF
2D7D E8F4E1                    call   TU_RANGE_CHECK
2D80 26                        seg    es
2D81 8805                      mov    [di], al
```



```
2D83 C43E6602                  les    di, [$0266]
2D87 B83412                    mov    ax, $1234
2D8A 26                        seg    es
2D8B 8905                      mov    [di], ax

2D8D C43E6A02                  les    di, [$026A]
2D91 06                        push   es
2D92 57                        push   di
2D93 E8AFEB                    call   TU_POUSSE_K_RE
2D96 810000000040              hex    810000000040
2D9C E8B7EB                    call   TU_POP_REEL_DI

2D9F C43E6E02                  les    di, [$026E]
2DA3 06                        push   es
2DA4 57                        push   di
2DA5 BF7202                    mov    di, $0272
2DA8 1E                        push   ds
2DA9 E8D6E3                    call   TU_POUSSE_STRING
2DAC B109                      mov    cl, $09
2DAE E800E4                    call   TU_POP_STRING

2DB1 E90000                    jmp    FIN

2DB4 33C0                FIN   xor    ax, ax
2DB6 E887DE                    call   TU_HALT
2DB9 0000                      hex    0000
```

Fig. 5.90 code de *desassemble_pointee*.

Notez que:

- pour affecter une valeur à une variable pointée scalaire:
  - l'adresse contenue dans le pointeur est placée dans ES et DI:

    LES DI, [adresse_du_pointeur]

  - l'expression est calculée dans AX
  - l'affectation a lieu en utilisant le segment ES et DI:

    SEG ES
    MOV [DI], AX

- pour affecter une valeur à une variable traitée sur la pile, il suffit de:
  - pousser l'adresse segmentée de la variable pointée sur la pile
  - calculer la valeur de l'expression
  - utiliser la routine qui dépile la valeur en utilisant l'adresse segmentée poussée avant le calcul de l'expression

5.4.6.4 LES POINTEURS DE TABLEAUX:

Voici un programme qui utilise des pointeurs de tableaux:

```
 0  (* 001 ptr5 *)
 1  (* 11 avr 87 *)
 2
 3  (*$r+*)
 4  PROGRAM desassemble_pointeur_tableau;
 5    TYPE t_tableau= ARRAY[5..9] OF INTEGER;
 6    VAR pointeur_tableau: ^t_tableau;
 7        tableau_pointeur: ARRAY[$0F..$2F] OF ^INTEGER;
 8        indice: INTEGER;
 9    BEGIN
10      pointeur_tableau^[indice]:= $88;
11      tableau_pointeur[indice]^:= $1234;
12    END.
```

Fig. 5.91 Le programme *desassemble_pointeur_tableau*.

Voici le code généré:

```
2D4D E849DD                          call   TU_PARAGRAPHS
2D50 0600                            hex    0600 ; options
2D52 780B                            hex    780B ; cs
2D54 3C15                            hex    3C15 ; ds
2D56 DB02                            hex    DB02 ; code
2D58 2F00                            hex    2F00 ; globales
2D5A 0004                            hex    0004 ; min piles
2D5C 00A0                            hex    00A0 ; max piles
2D5E 1000                            hex    1000 ; fichiers
2D60 0000                            hex    0000 ; input
2D62 0000                            hex    0000 ; output

2D64 8BEC                            mov    bp, sp
2D66 E814DF                          call   TU_TAILLE_CODE
2D69 4400                            hex    4400 ; taille
2D6B 6D2D                            adr    SAUT_PRINCIPAL

2D6D E90000          SAUT_PRINCIPAL  jmp    PRINCIPAL

2D70 C43E6202        PRINCIPAL       les    di, [$0262]
2D74 A1EA02                          mov    ax, [$02EA]
2D77 2D0500                          sub    ax, $0005
2D7A B90500                          mov    cx, $0005
2D7D E8EAE1                          call   TU_INDEX_CHECK
2D80 D1E0                            shl    ax, 1
2D82 03F8                            add    di, ax
2D84 B88800                          mov    ax, $0088
2D87 26                              seg    es
2D88 8905                            mov    [di], ax
```



```
2D8A A1EA02                          mov    ax, [$02EA]
2D8D 2D0F00                          sub    ax, $000F
2D90 B92100                          mov    cx, $0021
2D93 E8D4E1                          call   TU_INDEX_CHECK
2D96 D1E0                            shl    ax, 1
2D98 D1E0                            shl    ax, 1
2D9A 97                              xchg   di
2D9B C4BD6602                        les    di, [di+ $0266]
2D9F B83412                          mov    ax, $1234
2DA2 26                              seg    es
2DA3 8905                            mov    [di], ax

2DA5 E90000                          jmp    FIN

2DA8 33C0            FIN             xor    ax, ax
2DAA E893DE                          call   TU_HALT
2DAD 0000                            hex    0000
```

Fig. 5.92 Pointeurs et tableaux

Notez que:

- lorsqu'il s'agit d'un pointeur de tableau:
  - la valeur du pointeur (l'adresse du tableau) et stockée dans ES et DI
  - le calcul de l'indice est effectué comme pour un tableau global, en ajoutant le décalage de la cellule à DI
  - l'expression est calculée dans AX
  - ES et DI sont alors utilisés pour affecter la valeur
- dans le cas de tableau de pointeurs:
  - l'indice est utilisé pour calculer l'adresse où se trouve le pointeur. Ce calcul utilise DI, comme pour un indice usuel
  - la valeur du pointeur est placée dans ES et DI par:

    LES DI, [DI+ adresse_base_tableau]

  - l'expression est calculée dans AX
  - ES et DI sont utilisés pour affecter la valeur

5.4.6.5 POINTEURS ET ENREGISTREMENTS

Voici un programme contenant des pointeurs d'enregistrements:

```
0  (* 001 ptr4 *)
1  (* 11 avr 87 *)
2
3  (*$r+*)
4  PROGRAM desassemble_pointeur_enregistrement;
5    TYPE t_record= RECORD
6                     nom: STRING[$1F];
7                     age: INTEGER;
8                   END;
9
10       t_pt_cellule= ^t_cellule;
11       t_cellule= RECORD
12                    age: INTEGER;
13                    pt_suite: t_pt_cellule;
14                  END;
15   VAR pointeur_record: ^t_record;
16       pointeur_pile, auxiliaire: t_pt_cellule;
17       indice: INTEGER;
18   BEGIN
19     pointeur_record^.age:= $77;
20     pointeur_pile^.pt_suite:= auxiliaire;
21     pointeur_pile^.pt_suite^.pt_suite:= auxiliaire;
22   END.
```

Fig. 5.93 Le programme *desassemble_pointeur_enregistrement.*

Voici le code généré:

```
2D4D E849DD                    call   TU_PARAGRAPHS
2D50 0600                      hex    0600 ; options
2D52 780B                      hex    780B ; cs
2D54 3C15                      hex    3C15 ; ds
2D56 DC02                      hex    DC02 ; code
2D58 2700                      hex    2700 ; globales
2D5A 0004                      hex    0004 ; min piles
2D5C 00A0                      hex    00A0 ; max piles
2D5E 1000                      hex    1000 ; fichiers
2D60 0000                      hex    0000 ; input
2D62 0000                      hex    0000 ; output

2D64 8BEC                      mov    bp, sp
2D66 E814DF                    call   TU_TAILLE_CODE
2D69 4A00                      hex    4A00 ; taille
2D6B 6D2D                      adr    SAUT_PRINCIPAL

2D6D E90000          SAUT_PRINCIPAL   jmp    PRINCIPAL

2D70 C43E6202        PRINCIPAL        les    di, [$0262]
2D74 B87700                    mov    ax, $0077
2D77 26                        seg    es
2D78 894520                    mov    [di+ $20], ax
```



```
2D7B C43E6602                  les    di, [$0266]
2D7F 06                        push   es
2D80 57                        push   di
2D81 C4066A02                  les    ax, [$026A]
2D85 8CC2                      mov    dx, es
2D87 5F                        pop    di
2D88 07                        pop    es
2D89 26                        seg    es
2D8A 894502                    mov    [di+ $02], ax
2D8D 26                        seg    es
2D8E 895504                    mov    [di+ $04], dx

2D91 C43E6602                  les    di, [$0266]
2D95 26                        seg    es
2D96 C47D02                    les    di, [di+ $02]
2D99 06                        push   es
2D9A 57                        push   di
2D9B C4066A02                  les    ax, [$026A]
2D9F 8CC2                      mov    dx, es
2DA1 5F                        pop    di
2DA2 07                        pop    es
2DA3 26                        seg    es
2DA4 894502                    mov    [di+ $02], ax
2DA7 26                        seg    es
2DA8 895504                    mov    [di+ $04], dx

2DAB E90000                    jmp    FIN

2DAE 33C0                FIN   xor    ax, ax
2DB0 E88DDE                    call   TU_HALT
2DB3 0000                      hex    0000
```

Fig. 5.94 code de *desassemble_pointeur_enregistrement.*

Notez que:

- lorsqu'il s'agit d'un simple pointeur d'enregistrement, le compilateur place directement le décalage de la zone dans l'instruction d'accès:

```
SEG  ES
MOV  [DI+ decalalge_zone], AX
```

- lorsque l'une des zones est un pointeur:
  - l'adresse du pointeur est chargée dans ES et DI
  - ces valeurs sont sauvegardées sur la pile
  - la valeur à placer dans le pointeur est placée dans AX et DX comme pour une affectation de pointeur global. Ces calculs peuvent utiliser ES et DI, d'où la sauvegarde précédente
  - les valeurs de ES et DI sont restaurées pour réaliser l'affectation

- cette mécanique est généralisable lorsqu'il s'agit de pointeurs de pointeurs...

5.4.6.6 LES POINTEURS ET WITH

Voici un programme utilisant des pointeurs avec l'instruction **WITH**:

```
0  (* 001 ptr7 *)
1  (* 11 avr 87 *)
2
3  PROGRAM desassemble_pointeur_with;
4    TYPE t_record= RECORD
5                     nom: STRING[$1F];
6                     age: INTEGER;
7                   END;
8         t_tableau_record= ARRAY[$10..$2F] OF t_record;
9    VAR pointeur_record: ^t_record;
10       pointeur_tableau_record: ^t_tableau_record;
11       indice: INTEGER;
12   BEGIN
13     WITH pointeur_record^ DO
14       age:= $33;
15     WITH pointeur_tableau_record^[indice] DO
16       age:= $33;
17   END.
```

Fig. 5.95 Le programme *desassemble_pointeur_with.*

Voci le code généré:

```
2D4D E849DD                    call   TU_PARAGRAPHS
2D50 0600                      hex    0600 ; options
2D52 780B                      hex    780B ; cs
2D54 3C15                      hex    3C15 ; ds
2D56 DB02                      hex    DB02 ; code
2D58 2700                      hex    2700 ; globales
2D5A 0004                      hex    0004 ; min piles
2D5C 00A0                      hex    00A0 ; max piles
2D5E 1000                      hex    1000 ; fichiers
2D60 0000                      hex    0000 ; input
2D62 0000                      hex    0000 ; output

2D64 8BEC                      mov    bp, sp
2D66 E814DF                    call   TU_TAILLE_CODE
2D69 4300                      hex    4300 ; taille
2D6B 6D2D                      adr    SAUT_PRINCIPAL

2D6D E90000          SAUT_PRINCIPAL jmp  PRINCIPAL
```



```
2D70 C43E6202        PRINCIPAL les    di, [$0262]
2D74 06                        push   es
2D75 57                        push   di
2D76 C47EFC                    les    di, [bp+ $FC]
2D79 B83300                    mov    ax, $0033
2D7C 26                        seg    es
2D7D 894520                    mov    [di+ $20], ax
2D80 83C404                    add    sp, $04

2D83 C43E6602                  les    di, [$0266]
2D87 A16A02                    mov    ax, [$026A]
2D8A B92200                    mov    cx, $0022
2D8D F7E1                      mul    ax, cx
2D8F 03F8                      add    di, ax
2D91 81C7E0FD                  add    di, $FDE0
2D95 06                        push   es
2D96 57                        push   di
2D97 C47EFC                    les    di, [bp+ $FC]
2D9A B83300                    mov    ax, $0033
2D9D 26                        seg    es
2D9E 894520                    mov    [di+ $20], ax
2DA1 83C404                    add    sp, $04

2DA4 E90000                    jmp    FIN

2DA7 33C0            FIN       xor    ax, ax
2DA9 E894DE                    call   TU_HALT
2DAC 0000                      hex    0000
```

Fig. 5.96 Pointeurs avec une instructions WITH

Notez que:

- nous avons travaillé en R- pour alléger le code
- l'accès se fait de la façon suivante:
  - l'adresse début de l'enregistrement est poussée sur la pile
  - cette adresse est chargée dans ES et DI en utilisant BP
  - l'expression est calculée
  - l'affectation se fait en utilisant ES et DI
  - la pile est libérée des 4 octets de l'adresse segmentée

La mécanique est donc similaire à celle utilisée dans le cas du **WITH** d'un enregistrement non pointé.

5.4.6.7 RESUME

L'affectation d'une valeur à un pointeur est effectuée en plaçant la valeur à affecter dans AX et BX

L'accès aux variables pointées se fait:

- en chargeant dans ES et DI la valeur du pointeur
- en modifiant éventuellement DI pour tenir compte d'un indice ou d'une zone
- en utilisant ES et DI pour traiter la valeur pointée (manipulation directe pour les scalaires, utilisation de la pile pour les réels, chaînes, ensembles...)

Dans le cas du **WITH**, l'adresse segmentée du début de l'enregistrement pointé est poussé sur la pile, où il est récupéré dans ES et DI grâce à BP.

5.5 READ ET WRITE

5.5.1 INTRODUCTION

PASCAL assimile tous les périphériques à des fichiers. Il existe fondamentalement deux types de fichiers distincts:

- les fichiers de caractères (écran, clavier, modem)
- les fichiers à structure de bloc (disquettes, disques durs, ram-disques)

Voici comment sont traités ces fichiers.

5.5.2 WRITE(OUTPUT) ET READ(INPUT)

5.5.2.1 WRITE VERS OUTPUT

Voici un programme effectuant des **WRITE** en utilisant le fichier de sortie par défaut:


```
0  (* 001 write1 *)
1  (* 11 avr 87 *)
2
3  (*$r+*)
4  PROGRAM desassemble_write;
5    VAR entier: INTEGER;
6        caractere: CHAR;
7        reel: REAL;
8        booleen: BOOLEAN;
9        chaine: STRING[$0F];
10       tableau: ARRAY[$20..$2F] OF CHAR;
11   BEGIN
12     WRITE(entier);
13     WRITE(caractere);
14     WRITE(reel);
15     WRITE(booleen);
16     WRITE(chaine);
17     WRITE(tableau);
18   END.
```

Fig. 5.97 Le programme *desassemble_write*.

Voici le code généré:

```
2D4D E849DD                    call   TU_PARAGRAPHS
2D50 0600                      hex    0600 ; options
2D52 780B                      hex    780B ; cs
2D54 3C15                      hex    3C15 ; ds
2D56 DF02                      hex    DF02 ; code
2D58 2900                      hex    2900 ; globales
2D5A 0004                      hex    0004 ; min piles
2D5C 00A0                      hex    00A0 ; max piles
2D5E 1000                      hex    1000 ; fichiers
2D60 0000                      hex    0000 ; input
2D62 0000                      hex    0000 ; output

2D64 8BEC                      mov    bp, sp
2D66 E814DF                    call   TU_TAILLE_CODE
2D69 8200                      hex    8200 ; taille
2D6B 6D2D                      adr    SAUT_PRINCIPAL

2D6D E90000          SAUT_PRINCIPAL   jmp    PRINCIPAL

2D70 E8AFF7          PRINCIPAL        call   TU_WRITE_DEFAUT
2D73 A16202                           mov    ax, [$0262]
2D76 50                               push   ax
2D77 B80000                           mov    ax, $0000
2D7A E806FB                           call   TU_WRITE_ENTIE
2D7D E853E2                           call   TU_IO_CHECK
```

```
2D80 E89FF7                        call    TU_WRITE_DEFAUT
2D83 A06402                        mov     al, [$0264]
2D86 32E4                          xor     ah, ah
2D88 50                            push    ax
2D89 B80000                        mov     ax, $0000
2D8C E8D8FA                        call    TU_WRITE_CAR
2D8F E841E2                        call    TU_IO_CHECK

2D92 E88DF7                        call    TU_WRITE_DEFAUT
2D95 BF6502                        mov     di, $0265
2D98 1E                            push    ds
2D99 E89AEB                        call    TU_POUSSE_REEL
2D9C B81200                        mov     ax, $0012
2D9F 50                            push    ax
2DA0 B8FFFF                        mov     ax, $FFFF
2DA3 E812FB                        call    TU_WRITE_REEL
2DA6 E82AE2                        call    TU_IO_CHECK

2DA9 E876F7                        call    TU_WRITE_DEFAUT
2DAC A06B02                        mov     al, [$026B]
2DAF 32E4                          xor     ah, ah
2DB1 50                            push    ax
2DB2 B80000                        mov     ax, $0000
2DB5 E818FB                        call    TU_WRITE_BOOLE
2DB8 E818E2                        call    TU_IO_CHECK

2DBB E864F7                        call    TU_WRITE_DEFAUT
2DBE BF6C02                        mov     di, $026C
2DC1 1E                            push    ds
2DC2 E8BDE3                        call    TU_POUSSE_STRING
2DC5 B80000                        mov     ax, $0000
2DC8 E825FB                        call    TU_WRITE_CHAIN
2DCB E805E2                        call    TU_IO_CHECK

2DCE E851F7                        call    TU_WRITE_DEFAUT
2DD1 BF7C02                        mov     di, $027C
2DD4 1E                            push    ds
2DD5 B110                          mov     cl, $10
2DD7 E8FFE3                        call    TU_ARRAY_TO_ST
2DDA B80000                        mov     ax, $0000
2DDD E810FB                        call    TU_WRITE_CHAIN
2DE0 E8F0E1                        call    TU_IO_CHECK

2DE3 E90000                        jmp     FIN

2DE6 33C0                  FIN     xor     ax, ax
2DE8 E855DE                        call    TU_HALT
2DEB 0000                          hex     0000
```

Fig. 5.98 L'affichage par défaut

Notez que:

- la lecture se fait en initialisant le fichier par défaut, puis en invoquant pour chaque type affichable une routine particulière de la librairie.

- pour les types réels, chaînes et tableaux de caractères, la donnée est poussée sur la pile
- comme nous avons compilé en mode I+ (mode par défaut), une vérification d'erreur est effectuée après chaque entrée / sortie.

5.5.2.2 WRITELN

Voici un programme utilisant **WRITELN**:

```
 0  (* 001 write2 *)
 1  (* 11 avr 87 *)
 2
 3  (*$r+*)
 4  PROGRAM desassemble_writeln;
 5    VAR entier: INTEGER;
 6    BEGIN
 7      WRITE;
 8
 9      WRITELN;
10      WRITELN(entier);
11
12      WRITE(entier, $77);
13    END.
```

Fig. 5.99 Le programme *desassemble_writeln*.

Et voici son code:

```
2D4D E849DD                        call    TU_PARAGRAPHS
2D50 0600                          hex     0600 ; options
2D52 780B                          hex     780B ; cs
2D54 3C15                          hex     3C15 ; ds
2D56 DC02                          hex     DC02 ; code
2D58 2700                          hex     2700 ; globales
2D5A 0004                          hex     0004 ; min piles
2D5C 00A0                          hex     00A0 ; max piles
2D5E 1000                          hex     1000 ; fichiers
2D60 0000                          hex     0000 ; input
2D62 0000                          hex     0000 ; output

2D64 8BEC                          mov     bp, sp
2D66 E814DF                        call    TU_TAILLE_CODE
2D69 4B00                          hex     4B00 ; taille
2D6B 6D2D                          adr     SAUT_PRINCIPAL

2D6D E90000            SAUT_PRINCIPAL  jmp     PRINCIPAL

2D70 E8AFF7            PRINCIPAL       call    TU_WRITE_DEFAUT
2D73 E85DE2                        call    TU_IO_CHECK
```

```
2D76 E8A9F7                         call    TU_WRITE_DEFAUT
2D79 E8C3FB                         call    TU_WRITELN
2D7C E854E2                         call    TU_IO_CHECK

2D7F E8A0F7                         call    TU_WRITE_DEFAUT
2D82 A16202                         mov     ax, [$0262]
2D85 50                             push    ax
2D86 B80000                         mov     ax, $0000
2D89 E8F7FA                         call    TU_WRITE_ENTIE
2D8C E8B0FB                         call    TU_WRITELN
2D8F E841E2                         call    TU_IO_CHECK

2D92 E88DF7                         call    TU_WRITE_DEFAUT
2D95 A16202                         mov     ax, [$0262]
2D98 50                             push    ax
2D99 B80000                         mov     ax, $0000
2D9C E8E4FA                         call    TU_WRITE_ENTIE
2D9F B87700                         mov     ax, $0077
2DA2 50                             push    ax
2DA3 B80000                         mov     ax, $0000
2DA6 E8DAFA                         call    TU_WRITE_ENTIE
2DA9 E827E2                         call    TU_IO_CHECK

2DAC E90000                         jmp     FIN

2DAF 33C0               FIN         xor     ax, ax
2DB1 E88CDE                         call    TU_HALT
2DB4 0000                           hex     0000
```

Fig. 5.100 WRITELN

Notez que:

- **WRITE** sans aucun paramètre est légal en TURBO PASCAL.
- **WRITELN** est effectué en invoquant la routine de la librairie TU_WRITELN
- **WRITELN**(parametre) est réalisé en effectuant un **WRITE**(parametre) normal, et en appelant TU_WRITELN après la routine d'affichage spécifique du paramètre. Nous économisons toutefois l'appel à la routine d'initialisation.
- conformément au souhait de WIRTH:

    WRITE(parametre1, parametre2);

  est équivalent à:

    WRITE(parametre1); **WRITE**(parametre2);

  Nous économisons ici encore un appel à TU_WRITE_DEFAUT

5.5.2.3 FORMATAGE DE L'AFFICHAGE

Voici comment nous pouvons formater l'affichage:


```
 0  (* 001 write3 *)
 1  (* 11 avr 87 *)
 2
 3  (*$r+*)
 4  PROGRAM desassemble_write_formate;
 5    VAR entier: INTEGER;
 6        reel: REAL;
 7    BEGIN
 8      WRITE(entier: $30);
 9      WRITE(reel: $20: $10);
10    END.
```

Fig. 5.101 Le programme *desassemble_write_formate*.

Voici le code généré:

```
2D4D E849DD                         call    TU_PARAGRAPHS
2D50 0600                           hex     0600 ; options
2D52 780B                           hex     780B ; cs
2D54 3C15                           hex     3C15 ; ds
2D56 DB02                           hex     DB02 ; code
2D58 2700                           hex     2700 ; globales
2D5A 0004                           hex     0004 ; min piles
2D5C 00A0                           hex     00A0 ; max piles
2D5E 1000                           hex     1000 ; fichiers
2D60 0000                           hex     0000 ; input
2D62 0000                           hex     0000 ; output

2D64 8BEC                           mov     bp, sp
2D66 E814DF                         call    TU_TAILLE_CODE
2D69 3600                           hex     3600 ; taille
2D6B 6D2D                           adr     SAUT_PRINCIPAL

2D6D E90000             SAUT_PRINCIPAL  jmp  PRINCIPAL

2D70 E8AFF7             SAUT_PRINCIPAL  call TU_WRITE_DEFAUT
2D73 A16202                         mov     ax, [$0262]
2D76 50                             push    ax
2D77 B83000                         mov     ax, $0030
2D7A E806FB                         call    TU_WRITE_ENTIE
2D7D E853E2                         call    TU_IO_CHECK

2D80 E89FF7                         call    TU_WRITE_DEFAUT
2D83 BF6402                         mov     di, $0264
2D86 1E                             push    ds
2D87 E8ACEB                         call    TU_POUSSE_REEL
2D8A B82000                         mov     ax, $0020
2D8D 50                             push    ax
2D8E B81000                         mov     ax, $0010
2D91 E824FB                         call    TU_WRITE_REEL
2D94 E83CE2                         call    TU_IO_CHECK
```

```
2D97 E90000                     jmp     FIN

2D9A 33C0               FIN     xor     ax, ax
2D9C E8A1DE                     call    TU_HALT
2D9F 0000                       hex     0000
```

Fig. 5.102 L'affichage formaté

Notez que:

- les routines d'affichage utilisent systématiquement 1 paramètre de formatage (2 dans le cas des réels. Si vous ne précisez rien, le compilateur utilise une valeur 0 ($12 et $FFFF pour les réels).

5.5.2.4 READ DE INPUT

Voici un programme effectuant des **READ** avec le fichier d'entrée par défaut:

```
 0  (* 001 read1 *)
 1  (* 11 avr 87 *)
 2
 3  (*$r+*)
 4  PROGRAM desassemble_read;
 5    VAR entier: INTEGER;
 6        caractere: CHAR;
 7        reel: REAL;
 8        booleen: BOOLEAN;
 9        chaine: STRING[$0F];
10        tableau: ARRAY[$20..$2F] OF CHAR;
11    BEGIN
12      READ(entier);
13      READ(caractere);
14      READ(reel);
15      (* pas accepte: READ(booleen); *)
16      READ(chaine);
17      READ(tableau);
18    END.
```

Fig. 5.103 Le programme *desassemble_read*.

Et voici son code:

```
2D4D E849DD                     call    TU_PARAGRAPHS
2D50 0600                       hex     0600 ; options
2D52 780B                       hex     780B ; cs
2D54 3C15                       hex     3C15 ; ds
```



```
2D56 DC02                       hex     DC02 ; code
2D58 2900                       hex     2900 ; globales
2D5A 0004                       hex     0004 ; min piles
2D5C 00A0                       hex     00A0 ; max piles
2D5E 1000                       hex     1000 ; fichiers
2D60 0000                       hex     0000 ; input
2D62 0000                       hex     0000 ; output

2D64 8BEC                       mov     bp, sp
2D66 E814DF                     call    TU_TAILLE_CODE
2D69 5400                       hex     5400 ; taille
2D6B 6D2D                       adr     SAUT_PRINCIPAL

2D6D E90000             SAUT_PRINCIPAL  jmp     PRINCIPAL

2D70 E8E2F7             PRINCIPAL       call    TU_READ_DEFAUT
2D73 BF6202                             mov     di, $0262
2D76 1E                                 push    ds
2D77 E8C6F9                             call    TU_READ_ENTIER
2D7A E856E2                             call    TU_IO_CHECK

2D7D E8D5F7                             call    TU_READ_DEFAUT
2D80 BF6402                             mov     di, $0264
2D83 1E                                 push    ds
2D84 E8A5F9                             call    TU_READ_CAR
2D87 E849E2                             call    TU_IO_CHECK

2D8A E8C8F7                             call    TU_READ_DEFAUT
2D8D BF6502                             mov     di, $0265
2D90 1E                                 push    ds
2D91 E8CBF9                             call    TU_READ_REEL
2D94 E83CE2                             call    TU_IO_CHECK

2D97 E8BBF7                             call    TU_READ_DEFAUT
2D9A BF6C02                             mov     di, $026C
2D9D 1E                                 push    ds
2D9E B10F                               mov     cl, $0F
2DA0 E8DAF9                             call    TU_READ_CHAINE
2DA3 E82DE2                             call    TU_IO_CHECK

2DA6 E8ACF7                             call    TU_READ_DEFAUT
2DA9 BF7C02                             mov     di, $027C
2DAC 1E                                 push    ds
2DAD B110                               mov     cl, $10
2DAF E8F8F9                             call    TU_READ_ARRAY
2DB2 E81EE2                             call    TU_IO_CHECK

2DB5 E90000                             jmp     FIN

2DB8 33C0               FIN             xor     ax, ax
2DBA E883DE                             call    TU_HALT
2DBD 0000                               hex     0000
```

Fig. 5.104 READ de INPUT

Notez que:

- la lecture s'effectue:
  - en initialisant le fichier d'entrée
  - en poussant le segment de la variable sur la pile, et en plaçant dans DI son décalage
  - en invoquant une routine particulière pour chaque type de donnée
- dans le cas des chaînes et des tableaux de caractères, CL contient en outre la taille maximale
- comme nous avons compilé en I+ (option par défaut), le compilateur génère le code de vérification d'erreur

5.5.2.5 READLN

Voici à présent un programme avec **READLN**:

```
 0  (* 001 read2 *)
 1  (* 11 avr 87 *)
 2
 3  (*$r+*)
 4  PROGRAM desassemble_readln;
 5    VAR entier, entier2: INTEGER;
 6    BEGIN
 7      READ;
 8      READLN;
 9      READLN(entier);
10      READ(entier, entier2);
11    END.
```

Fig. 5.105 Le programme *desassemble_readln*.

Voici son code:

```
2D4D E849DD              call    TU_PARAGRAPHS
2D50 0600                hex     0600 ; options
2D52 780B                hex     780B ; cs
2D54 3C15                hex     3C15 ; ds
2D56 DB02                hex     DB02 ; code
2D58 2700                hex     2700 ; globales
2D5A 0004                hex     0004 ; min piles
2D5C 00A0                hex     00A0 ; max piles
2D5E 1000                hex     1000 ; fichiers
2D60 0000                hex     0000 ; input
2D62 0000                hex     0000 ; output

2D64 8BEC                mov     bp, sp
2D66 E814DF              call    TU_TAILLE_CODE
2D69 3C00                hex     3C00 ; taille
2D6B 6D2D                adr     SAUT_PRINCIPAL
```



```
2D6D E90000              SAUT_PRINCIPAL  jmp     PRINCIPAL

2D70 E8E2F7              PRINCIPAL       call    TU_READ_DEFAUT
2D73 E85DE2                              call    TU_IO_CHECK

2D76 E8D8F7                              call    TU_READLN
2D79 E857E2                              call    TU_IO_CHECK

2D7C E8D2F7                              call    TU_READLN
2D7F BF6202                              mov     di, $0262
2D82 1E                                  push    ds
2D83 E8BAF9                              call    TU_READ_ENTIER
2D86 E84AE2                              call    TU_IO_CHECK

2D89 E8C9F7                              call    TU_READ_DEFAUT
2D8C BF6202                              mov     di, $0262
2D8F 1E                                  push    ds
2D90 E8ADF9                              call    TU_READ_ENTIER
2D93 BF6402                              mov     di, $0264
2D96 1E                                  push    ds
2D97 E8A6F9                              call    TU_READ_ENTIER
2D9A E836E2                              call    TU_IO_CHECK

2D9D E90000                              jmp     FIN

2DA0 33C0                FIN             xor     ax, ax
2DA2 E89BDE                              call    TU_HALT
2DA5 0000                                hex     0000
```

Fig. 5.106 READLN

Notez que:

- TURBO PASCAL accepte un **READ** sans aucun paramètre
- **READLN** est effectué en invoquant TU_READLN, qui initialise le fichier d'entrée par défaut, en précisant comment se terminera la saisie. Le mécanisme de tampon par défaut explique cette similitude entre **READ** et **READLN**.
- la lecture de plusieurs paramètres dans un même **READ** est décomposée en plusieurs lectures, avec une seule initialisation du fichier d'entrée

5.5.3 LES FICHIERS SUR DISQUE

5.5.3.1 TRAITEMENT D'UN FICHIER DE FICHES

Voici un programme utilisant un fichier de fiches (**FILE OF**):