Chapitre 18

Programmation modulaire

Nous avons vu, lors de l'initiation à la programmation, la notion de *sous-programme* (éventuellement sous des noms divers de fonction, procédure...) et en quoi la *programmation modulaire* est intéressante. Nous allons voir comment mettre en place les sous-programmes en langage machine du microprocesseur 8086/8088.

18.1 Notions générales

Voyons quels sont les problèmes posés par un programmeur pour la mise en place d'un sous-programme en langage symbolique.

Repérage d'un sous-programme.- Un sous-programme est repéré par un nom dans le cas d'un langage évolué. Dans le cas d'un langage machine il sera repéré par une *adresse de début*. Il s'agit en général de l'adresse de sa première instruction mais pas toujours : il peut lui-même contenir des sous-programmes qui viennent avant lui ; le sous-programme proprement dit peut-être précédé de (déclaration de) constantes.

Arguments d'un sous-programme.- Contrairement au cas des langages évolués, un sous-programme en langage machine n'a pas d'arguments. Autrement dit tous les paramètres sont passés comme variables globales, soit à l'aide des registres, soit comme emplacement mémoire, soit à l'aide de la pile que nous étudierons dans le chapitre suivant.

Fin d'un sous-programme.- Il faut évidemment indiquer la dernière instruction d'un sous-programme, sinon toutes les instructions qui suivent cette instruction (considérées par nous comme la dernière) seraient effectuées par le microprocesseur (en allant en particulier dans le sous-programme suivant). Un sous-programme se termine par une instruction spéciale indiquant qu'il s'agit de la dernière instruction.

Adresse de retour.- Lorsqu'on a terminé un sous-programme, le microprocesseur doit savoir où retourner dans le programme principal, autrement dit quelle est l'**adresse de retour**. Pour faciliter la vie du programmeur, cette adresse est en général conservée par le microprocesseur, au moment de l'appel du sous-programme, sur une *pile* (et qu'importe ce que c'est pour l'instant), puis récupérée, également par le microprocesseur de façon transparente pour le programmeur, au moment du retour ; le programmeur n'a pas à s'en péoccupper.

Place d'un sous-programme.- On peut placer le code d'un sous-programme où l'on veut. Si on ne commence pas par le *programme principal*, il faut le faire précéder d'un saut inconditionnel, ce qui permet de passer par-dessus le code du sous-programme.

Appel d'un sous-programme.- On pourrait appeler un sous-programme par un saut mais il est plus astucieux d'utiliser une instruction spéciale (qui s'occupe entre autre de placer l'adresse de retour sur la pile), dont l'opérande est l'adresse du sous-programme.

18.2 Cas du microprocesseur 8086/8088

18.2.1 Appel du sous-programme

Langage symbolique.- Un sous-programme peut être appelé grâce à l'instruction `CALL` (*appel* en anglais) dont l'opérande est l'adresse du sous-programme :

```
CALL adresse
```

Langage machine.- L'appel d'un sous-programme peut s'effectuer dans le même segment (IP changé mais pas CS) ou dans un autre segment (IP et CS changés), de façon directe (en spécifiant l'adresse par une constante) ou indirecte (l'adresse se trouve dans un registre ou un emplacement mémoire) :

– Dans le cas d'un appel intra-segmentaire direct, on précise en fait l'adresse par un déplacement (entier relatif) par rapport à la valeur en cours de IP :

```
| 1110 1000 | déplacement bas | déplacement haut |
```

L'appel a lieu pour un programme commençant en [IP] + `déplacement`, où [IP] est l'adresse de l'instruction sui suit l'appel.

– Dans le cas d'un appel inter-segmentaire direct, on précise à la fois la valeur de IP et celle de CS :

```
| 1001 1010 | IP bas | IP haut | CS bas | CS haut |
```

– Dans le cas d'un appel intra-segmentaire indirect, on précise la valeur de IP :

```
| 1111 1111 | mod 010 r/m |        |        |
```

– Dans le cas d'un appel inter-segmentaire indirect, on précise la valeur de IP et de CS, contenus dans un emplacement mémoire (un registre n'est pas suffisant) :

```
| 1111 1111 | mod 011 r/m |        |        |
```

avec $mod \neq 11$. Dans ce cas [IP] et [CS] sont remplacés par les contenus des cases mémoire (16 bits) définies par l'adresse effective et l'adresse effective + 2.

18.2.2 Retour d'un sous-programme

Langage symbolique.- Un sous-programme doit contenir au moins une instruction `RET` (pour *RETurn* ou *RETour*), indiquant la fin du sous-programme (et donc le retour au programme principal) de façon inconditionnelle :

```
RET
```

Langage machine.- Il existe en fait deux instructions de retour : le retour court correspondant à un appel court, c'est-à-dire intrasegmentaire alors que le retour long correspond à un appel long, c'est-à-dire intersegmentaire. Cette différence s'explique par le fait que, dans le premier cas, la valeur de IP doit être rétablie alors que dans le second cas, il faut rétablir IP et CS.

– Le retour court est codé sur un octet par `1100 0011`, soit &HC3.

– Le retour long, que nous avons déjà vu depuis le départ, est également codé sur un octet par `1100 1011`, soit &HCB.

18.2.3 Un exemple

Écrivons un programme BASIC qui demande un entier naturel n compris entre 0 et 255, et qui affiche la valeur de l'octet faible de $2^n + n$. Le calcul de cette fonction effectué grâce à un sous-programme écrit en langage machine.

Nous avons déjà écrit un programme en langage machine permettant de calculer l'exponentiation X^Y. Nous retrouverons ainsi souvent des parties de programme déjà écrites. Autant les placer dans des sous-programmes et les réutiliser.

De façon à réutiliser au mieux notre sous-programme d'exponentiation vu précédemment, le programme BASIC principal placera n à l'adresse &H0002. Le programme machine placera 2 à l'adresse &H0000, fera appel au sous-programme de calcul de l'exponentielle, qui place le résultat à l'adresse &H0004, puis ajoutera le contenu de la case mémoire &H0002 à celui de celle d'adresse &H0004.

Le sous-programme s'écrit de la façon suivante en langage symbolique :

```
MOV AX, 2
MOV CS :[0000], AX
CALL EXP
MOV AX, CS :[0002]
ADD CS :[0004], AX
RETF
EXP : MOV CX, CS :[0002]
MOV BX, CS :[0000]
MOV AX, 1
DEBUT : MUL BX
DEC CX
JNZ DEBUT
MOV CS :[0004], AX
RET
```

L'appel et le retour du sous-programme du programme en langage machine peut être court.

Traduisons maintenant ce programme écrit en langage symbolique en langage machine, en recopiant à partir de la septième instruction le programme écrit pour l'exponentiation, simplement en rempļçant &HCB par &HC3 :

```
&HB8 &H02 &H00
&H2E &HA3 &H00 &H00
&HE8 &H?? &H00
&H2E &HA1 &H02 &H00
&H2E &H01 &H06 &H04 &H00
&HCB
&H2E &H8B &H0E &H02 &H00
&H2E &H8B &H1E &H00 &H00
&HB8 &H01 &H00
&HF7 &HE3
&H49
&H75 &HFB
&H2E &HA3 &H04 &H00
&HC3
```

On peut maintenant calculer le décalage, qui est dix, c'est-à-dire la somme des longueurs des quatrième, cinquième et sixième instructions. La troisième instruction s'écrit donc :

```
&HE8 &H0A &H00
```

Nous pouvons utiliser le programme QBasic suivant pour tester notre sous-programme, sachant que la donnée, les données auxiliaires et le résultat occupent 8 octets et le code 43 octets :

```
CLS
DIM PM%(26)
SegPM% = VARSEG(PM%(0))
OffPM% = VARPTR(PM%(0))
'Saisie des donnees
INPUT "N = ", N%
PM%(1) = N%
'Passage au segment du code machine
DEF SEG = SegPM%
```

```
'Initialisation du tableau par le code machine
FOR I% = 0 TO 42
  READ Octet$
  POKE OffPM% + 8 + I%, VAL("&H" + Octet$)
NEXT I%
'Visualisation du contenu du debut de ce segment avant execution
FOR I% = 0 TO 14
  PRINT PEEK(I%),
NEXT I%
'Execution du programme machine
CALL ABSOLUTE(OffPM% + 8)
'Visualisation du contenu du debut de ce segment apres execution
FOR I% = 0 TO 14
  PRINT PEEK(I%),
NEXT I%
'Retour au segment de depart
DEF SEG
'Affichage du resultat
PRINT PM%(2)
END
'Code machine
DATA B8,02,00
DATA 2E,A3,00,00
DATA E8,0A,00
DATA 2E,A1,02,00
DATA 2E,01,06,04,00
DATA CB
DATA 2E,8B,0E,02,00
DATA 2E,8B,1E,00,00
DATA B8,01,00
DATA F7,E3
DATA 49
DATA 75,FB
DATA 2E,A3,04,00
DATA C3
```

L'exécution se comporte comme prévu :

```
N = 3
0     0    3     0    0
0     0    0     B8   2
0     2E   A3    0    0

0     0    3     0    B
0     0    0     B8   2
0     2E   A3    0    0

 11
```

ce qui donne le bon résultat, puisque $2^3 + 3 = 11$.

18.3 Historique

Les modèles de calcul, dont la machine de Turing, utilisent la notion de sous-programme

Comme nous l'avons vu au chapitre 14, le premier langage symbolique (EDSAC, 1949) ne prévoit pas d'instructions pour manipuler les sous-programmes bien qu'il en utilise un dès le premier programme proposé, celui sur le calcul des carrés. Le mécanisme de sous-programme est entièrement géré par l'utilisateur.