Chapitre 13

Accès au langage machine sur PC

Comment faire réagir un microprocesseur tel que le 8088 ? Comment programmer en langage machine sur un système tel que le PC ? Voici les questions auxquelles nous allons répondre dans ce chapitre.

13.1 Premières instructions du 8088

Commençons par étudier quelques instructions du 8088 de façon à programmer de façon concrète ensuite.

13.1.1 Classification des instructions de transfert

La modélisation des ordinateurs que nous avons donnée ci-dessus nous conduit à distinguer quatre types d'instructions de transfert :

- l'**initialisation d'un registre** qui permet de transférer une valeur dans un registre directement ;
- les **transferts entre registres** qui permettent de transférer une valeur d'un registre dans un autre ;
- les **transferts entre registre et mémoire vive** qui permettent de transférer une valeur d'un registre vers un élément de mémoire vive, ou *vice-versa* ;
- les **accès à la mémoire vive** qui permettent de transférer une valeur d'un élément de mémoire vive à un autre ;
- les **entrées-sorties** qui permettent de transférer une valeur d'un registre vers un **périphérique**, ou *vice-versa*.

13.1.2 Les registres

Notion.- La notion de *mémoire* est importante pour un ordinateur. Comme nous l'avons vu, on distingue, d'un point de vue matériel, les *registres* qui sont les éléments de mémoire situés sur le microprocesseur lui-même, les éléments de *mémoire vive* installés sur la carte mère et la *mémoire de masse* située sur des périphériques (disquettes, disque dur, CD-rom...). Bien entendu cette distinction, d'origine matérielle, va avoir des retombées sur la programmation système.

Nombre de registres.- Seuls deux registres sont indispensables en théorie : l'**accumulateur**, qui contient les résultat des calculs, et le **compteur ordinal** (ou **pointeur d'instruction**), qui contient le numéro de la prochaine instruction à exécuter. Mais en général un microprocesseur comporte beaucoup plus de registres pour accélérer la vitesse de l'ordinateur : ceci évite le passage sans cesse de l'accumulateur à un élément de mémoire vive.

Numérotation des bits.- Par convention, les bits d'un élément de mémoire, et donc d'un registre, sont numérotés de façon à ce que le bit i contienne le i-ième chiffre binaire (celui correspondant à 2^i) si le contenu est considéré comme étant un entier naturel. Ils sont donc numérotés de 0 à 7 pour un octet.

Registres cachés.- Un registre est un élément mémoire inclus dans le microprocesseur. Les concepteurs d'un microprocesseur peuvent donc décider d'en placer beaucoup pour accélérer tel ou tel point. Certains registres peuvent être utiles pour accélérer les calculs de façon matérielle sans que le programmeur puisse y accéder explicitement. Les registres sur lesquels le programmeur peut agir explicitement reçoivent un nom. Les autres sont appelés **registres cachés**.

13.1.3 Premières instructions de transfert du 8088

Nous avons vu la notion de registre. Voyons comment initialiser un registre et comment transférer une valeur d'un registre à un élément de mémoire vide.

13.1.3.1 Syntaxe générale d'un transfert en langage symbolique

Syntaxe.- Toute instruction de transfert sera représentée en un **langage symbolique** (qui permet une meilleure compréhension à nous, pauvres humains) de la façon suivante :

```
MOV destination, source
```

où `MOV` est un mnémonique pour *MOVe* (*transférer* en anglais), `destination` ou `source` est le nom d'un registre et l'autre (`source` ou `destination` suivant le cas) dépend du contexte.

Remarque.- Attention à l'ordre, d'abord la destination, ensuite la source, ce qui n'est pas nécessairement l'ordre auquel on pourrait s'attendre. On peut penser à l'analogue en langage de haut niveau :

```
destination := source ;
```

Que transférer ?.- Le 8088 est un microprocesseur 16 bits, c'est-à-dire que les transferts se font sur seize bits, soit deux octets. En fait, puisque l'octet est la plus petite unité et que le bus des données est de huit bits, on peut également transférer un octet.

13.1.3.2 Initialisation d'un registre

Langage symbolique.- L'instruction d'initialisation d'un registre s'écrit :

```
MOV reg, constante
```

où `reg` désigne le registre et `constante` est une constante huit ou seize bits suivant la capacité du registre.

Sémantique.- La signification de cette instruction est l'initialisation du registre `reg` par cette constante.

Langage machine.- L'instruction en langage symbolique n'est là que pour nous aider à s'en souvenir. Il faut maitenant faire le lien avec le microprocesseur. Une telle instruction en langage machine exige deux octets ou trois octets, suivant que la constante occupe un ou deux octets :

$\boxed{\text{opcode}}\ \boxed{\text{constante}}\ [\ \boxed{\text{constante}}\ ]$

le premier octet est appelé **code opération**, abrégé en **opcode**.

Opcodes des initialisations.- On pourrait choisir les codes opération au hasard. Pour une meilleure ergonomie (le 8088 comprend plusieus centaines d'instructions), ceci n'est pas le cas. Le code d'opération d'une instruction d'initialisation est, en binaire :

```
1011 wreg
```

où :

- `w` (pour l'anglais *word*, soit mot, le nom pour une donnée de deux octets) est un bit valant 1 si la transfert se effectue sur deux octets (soit un mot) et 0 s'il s'effectue sur un octet.

– `reg` est une suite de trois bits permettant de déterminer le registre de la façon spécifiée par la figure 13.1.

16 bits ($w = 1$)	8 bits ($w = 0$)	Segment
000 AX	000 AL	00 ES
001 CX	001 CL	01 CS
010 DX	010 DL	10 SS
011 BX	011 BL	11 DS
100 SP	100 AH	
101 BP	101 CH	
110 SI	110 DH	
111 DI	111 BH	

FIG. 13.1 – Désignation des registres du 8088

Remarquez que les registres de segment sont désignés par deux bits seulement et qu'ils ne peuvent donc pas être initialisés directement.

Exercice corrigé.- *Traduire l'instruction :*

```
MOV AL, 10
```

en langage machine.

D'après ce que nous venons de dire, cette instruction se traduit par les deux octets (représentés en binaire) :

```
1011 0000 0000 1010
```

soit &HB0 &H0A en hexadécimal.

Remarque.- En général on utilisera l'hexadécimal en langage symbolique, c'est-à-dire qu'on aurait écrit :

```
MOV AL, &H0A
```

dès le départ.

Vocabulaire.- Il est traditionnel de parler d'**adressage immédiat** au lieu d'initialisation des registres.

13.1.3.3 Codage des mots

Nous avons commencé par coder une instruction de transfert sur un octet car un autre problème se pose pour le transfert sur un mot, à savoir le codage de celui-ci. Un mot comprend deux octets : un appelé **octet de plus fort poids** (**MSB** pour l'anglais *Most Significant Byte*) et l'autre **octet de plus faible poids** (**LSB** pour l'anglais *Least Significant Byte*). On comprend facilement que la valeur du mot est :

$$w = 256 \times MSB + LSB$$

mais comment coder *w* : par MSB LSB ou par LSB MSB ? Il s'agit d'une vielle discussion entre informaticiens, la bataille des petits boutiens (*little endian* en anglais) contre les grands boutiens (*great endian* en anglais), par référence aux *Voyages de Gulliver* de Daniel DEFOE dans lesquels on discute où casser un œuf pour le gober : par le grand bout ou le petit bout.

Cela dépend des concepteurs de microprocesseur. Intel a choisi l'ordre LSB MSB, qui n'est pas nécessairement l'ordre auquel on pourrait s'attendre.

Exercice corrigé.- *Traduire l'instruction* :

```
MOV AX, 10
```

en langage machine.

D'après ce que nous venons de dire, cette instruction se traduit par les trois octets (représentés en binaire) :

```
1011 1000 0000 1010 0000 0000
```

soit &HB8 &H0A &H00 en hexadécimal.

13.1.3.4 Transfert d'accumulateur à mémoire vive

Langage symbolique.- L'instruction de transfert d'accumulateur à mémoire vive s'écrit :

```
MOV [decalage], A
```

où `A` désigne l'un des registres `AL` ou `AX` et `decalage` le décalage de l'adresse, qui est un entier de seize bits. Le segment par défaut est le segment de données.

Sémantique.- La signification de cette instruction est la copie du contenu de l'accumulateur dans l'élément de mémoire vive d'adresse spécifiée.

Remarque.- Il s'agit de l'analogue de l'instruction POKE du BASIC.

Langage machine.- Le transfert d'accumulateur à mémoire exige trois octets :

```
opcode faible fort
```

où `faible` est l'octet de poids faible du décalage de l'adresse, `fort` l'octet de poids fort et `opcode` est :

```
1010 001w
```

avec, évidemment, `w` égal à 0 pour `AL` et à 1 pour `AX`, soit &HA2 ou $HA3.

Exercice corrigé.- *Traduire l'instruction* :

```
MOV [A900h], AX
```

en langage machine.

D'après ce que nous venons de dire, cette instruction se traduit par les trois octets :

```
1010 0011 0000 000 1010 1001
```

en binaire, soit &HA3, &H00, &HA9 en hexadécimal.

Vocabulaire.- Il est traditionnel de parler d'**adressage absolu** ou d'**adressage étendu** dans le cas de copie entre accumulateur et mémoire.

13.1.3.5 Choix du segment

Nous venons comment transférer le contenu de l'accumulateur en mémoire. On spécifie le décalage et, par défaut, le segment est celui des données. Il s'agit effectivement du segment le plus approprié. Si ce n'est pas ce que l'on désire, il faut faire précéder l'instruction par un octet spécifiant le segment désiré, appelé **préfixe de changement de segment** (*override prefix* en anglais), qui peut être CS ou SS à la place de DS :

```
001 reg 110
```

Exercice corrigé.- *Traduire l'instruction* :

```
MOV CS :[A900h], AX
```

en langage machine.

Il suffit d'ajouter le préfixe `001 01 110` à l'instruction que nous venons de voir, ce qui donne &H2E &HA33, &H00, &HA9 en hexadécimal.

13.1.4 Instruction de retour

Notion.- Nous verrons comment faire appel en BASIC à un sous-programme écrit en langage machine. Lorsqu'on fait appel à un sous-programme, l'adresse du programme à laquelle on se trouve (c'est-à-dire les contenus des registres `CS` et `IP`, même si ce dernier registre n'est pas visible) est sauvegardée de façon à pouvoir revenir à cette instruction une fois le sous-programme exécuté. Le sous-programme doit lui-même indiquer qu'il a terminé son travail de façon à retourner à cette adresse. L'instruction `STOP` ne conviendrait pas : elle indiquerait que le programme en son entier est terminé.

Langage symbolique.- Cette instruction s'écrit tout simplement :

```
RETF
```

pour l'anglais *RETurn Far* (l'instruction RET que nous verrons plus tard change uniquement IP mais pas CS).

Langage machine.- Cette instruction s'écrit en un octet `1100 1011`, soit &HCB en hexadécimal.

13.2 Programmation machine en QBasic sous MS-DOS

13.2.1 Un programme machine

Écrivons notre premier programme machine, un programme suffisamment court et qui donne quelque chose de palpable : placer 17 dans l'accumulateur puis dans un emplacement mémoire, faute de mieux au déplacement &H0000. En langage symbolique nous avons :

```
MOV AX,11
MOV [00], AX
RETF
```

Nous avons vu précédemment comment on peut le traduire en langage machine :

```
&HB8 &H11 &H00
&HA3 &H00 &H00
&HCB
```

écrit ici en conservant des passages à la ligne pour bien faire le lien avec le programme écrit en langage symbolique, mais ceci n'a vraiment aucune importance (ni aucun sens) en langage machine.

Remarque.- En fait les premières instructions qu'on aimerait bien voir en langage machine concernent les entrées-sorties, comme avec le BASIC. Ceci n'est pas possible car elles n'existent pas en langage machine. Nous verrons plus tard comment utiliser les appels système du système d'exploitation.

On ne peut d'ailleurs utiliser que des sous-programmes en langage machine de la façon que nous allons indiquer, et non un programme complet.

13.2.2 Appel d'un sous-programme en langage machine en QBasic

L'instruction CALL ABSOLUTE.- QBasic possèe une instruction pour faire appel à un sous-programme écrit en langage machine :

```
CALL ABSOLUTE(adresse)
```

où `adresse` est le déplacement de l'adresse dans le segment courant ou spécifié par l'instruction DEF SEG.

Où placer le code machine ?.- QBasic nous donne donc l'opportunité de placer le code machine là où on veut. Mais si on veut revenir du sous-programme sans dommage, en particulier sans rendre le système d'exploitation instable, nous pouvons pas le placer n'importe où. En effet, les systèmes d'exploitation « modernes » (et MS-DOS en est un dans ce sens là), gèrent la mémoire vive pour nous. Le système d'exploitation marque toutes les cellules mémoire utilisées ; lorsqu'on déclare une variable, par exemple, il recherche un nombre adéquat de cellules mémoire non marquées compatible avec la taille du contenu de la variable, marque ces cellules et renvoie l'adresse de la première cellule. Placer nous-même quelque chose en mémoire risque de venir surcharger une ou plusieurs cellules utilisées pour autre chose et donc de rendre notre programme instable, sinon le système d'exploitation en entier. Dans ce dernier cas, la seule façon de s'en sortir est de redémarrer le système, ce qui vous est déjà certainement arrivé avec MS-DOS.

Une astuce classique consiste à déclarer une variable tableau (entiers de seize bits) d'un dimension pouvant recevoir le code machine, d'y placer ce code machine, de se placer dans le segment de cette variable et de faire un CALL ABSOLUTE au déplacement de ce tableau.

Exemple.- Commençons par changer un peu le programme machine désiré, pour deux raisons. Le programme machine va se trouver dans un certain segment, certainement au décalage 0 ; lorsqu'on exécute ce code machine, on ne connaît pas la valeur du segment de registre DS utilisé ; utiliser le programme ci-dessus ira bien écrire 17 au déplacement 0 d'un segment mais nous ne savons pas lequel et nous pouvons donc pas aller le récupérer pour voir si tout s'est bien passé. On va donc faire précéder la seconde instruction de déplacement du préfixe `&H2E` pour que cela soit écrit dans le segment dont l'adresse est donnée par le CS au moment de l'exécution du programme en code machine :

```
MOV CS :[00], AX
```

La seconde raison est, qu'en faisant ainsi, nous détruisons une partie de la première instruction. Ce n'est pas grave car le code machine n'est pas réutilisé. Il vaut mieux, cependant, aller placer le résultat après le code machine. Puisque le code machine occupe huit octets avec le préfixe, on peut aller le placer dans l'octet commençant au déplacement 8. Notre code machine est donc maintenant :

```
&HB8 &H11 &H00
&H2E &HA3 &H08 &H00
&HCB
```

Écrivons donc le programme QBasic suivant :

```
CLS
DIM PM%(4)      'PM comme Programme Machine
SegPM% = VARSEG(PM%(0))
OffPM% = VARPTR(PM%(0))
'Initialisation du tableau par le code machine
PM%(0) = &H11B8
PM%(1) = &H2E00
PM%(2) = &H8A3
PM%(3) = &HCB00
'Passage au segment du code machine
DEF SEG = SegPM%
'Visualisation du contenu du debut de ce segment avant execution
PRINT PEEK(0), PEEK(1), PEEK(2), PEEK(3), PEEK(4)
PRINT PEEK(5), PEEK(6), PEEK(7), PEEK(8), PEEK(9)
'Execution du programme machine
CALL ABSOLUTE(OffPM%)
'Visualisation du contenu du debut de ce segment apres execution
PRINT PEEK(0), PEEK(1), PEEK(2), PEEK(3), PEEK(4)
PRINT PEEK(5), PEEK(6), PEEK(7), PEEK(8), PEEK(9)
'Retour au segment de d\'epart
DEF SEG
PRINT PM%(4)
END
```

dont l'affichage de 17 (= &H11) :

```
184   17     0    46   163
8      0   203     0     0

184   17     0    46   163
8      0   203    17     0
```

```
17
```

de deux façons différentes, nous laisse penser que le sous-programme en langage machine s'est correctement déroulé.

<u>Commentaires</u>.- 1^o) Nous avons avons appelé le tableau PM pour rappeler qu'il va contenir le programme machine. On aurait évidemment pu l'appeler comme on veut.

- 2^o) Par contre il est important de bien le typer par des entiers codés sur seize bits (sur huit bits aurait été mieux pour ce que l'on veut faire, mais ce n'est pas possible en QBasic) de façon à ce que le code soit rangé correctement au moment de l'initialisation du tableau.

- 3^o) Nous avons besoin de huit octets pour le programme et de deux octets pour le résultat, soit dix octets. Puisque le tableau est un tableau d'entiers de seize bits, il suffit donc qu'il soit de dimension 5.

- 4^o) Les fonctions VARSEG et VARPTR prennent en argument une variable numérique (et non tableau), d'où le choix de l'argument `PM%(0)` et non `PM%`.

- 5^o) Lors de l'initialisation du tableau, on notera l'inversion des octets par rapport au code machine de façon à ce qu'ils soient bien placés en mémoire.

- 6^o) La visualisation du début du segment dans lequel se trouve le code machine, non nécessaire pour ce qu'on veut faire, nous montre, avant exécution, que le code machine est bien placé là où nous voulons et, après exécution, que 17 a bien été placé là où on voulait.

- 7^o) Il ne faut pas oublier de revenir au segment d'où nous étions parti après exécution du programme machine, faute de ne plus rien voir à l'écran et d'être obligé de redémarrer la machine.

13.2.3 Amélioration de l'ergonomie de chargement du programme

Nous venons de voir comment faire exécuter un sous-programme écrit en langage machine. Nous allons donc pouvoir exploré le langage machine à partir du chapitre suivant. Mais il faut bien dire que la façon d'entrer le programme machine est un peu contraignante, surtout lorsque le programme sera plus long.

<u>Première amélioration : lecture des données</u>.- Une première amélioration consiste à placer le code machine en données et à le faire lire octet par octet :

```
CLS
DIM PM%(4)      'PM comme Programme Machine
SegPM% = VARSEG(PM%(0))
OffPM% = VARPTR(PM%(0))
'Passage au segment du code machine
DEF SEG = SegPM%
'Initialisation du tableau par le code machine
FOR i% = 0 TO 7
  READ Octet%
  POKE OffPM% + i%, Octet%
NEXT i%
'Visualisation du contenu du debut de ce segment avant execution
PRINT PEEK(0), PEEK(1), PEEK(2), PEEK(3), PEEK(4)
PRINT PEEK(5), PEEK(6), PEEK(7), PEEK(8), PEEK(9)
'Execution du programme machine
```

```
CALL ABSOLUTE(OffPM%)
'Visualisation du contenu du debut de ce segment apres execution
PRINT PEEK(0), PEEK(1), PEEK(2), PEEK(3), PEEK(4)
PRINT PEEK(5), PEEK(6), PEEK(7), PEEK(8), PEEK(9)
'Retour au segment de d\'epart
DEF SEG
PRINT PM%(4)
END
DATA &HB8,&H11,&H00
DATA &H2E,&HA3,&H08,&H00
DATA &HCB
```

Ceci permet de ne pas avoir à répéter '`PM%(0) =`' et à placer les octets du code machine dans l'ordre naturel et même, si on le désire (ce qui est fait ici), en des lignes rappelant le code en langage symbolique.

Commentaires.- 1^o) Attention à bien passer au segment de code machine avant d'utiliser les instruction POKE.

- 2^o) L'instruction END qui était facultative dans le programme précédent est maintenant indispensable à cause des DATA.

Deuxième amélioration.- On peut même aller plus loin en omettant le préfixe '`&H`' des données, en les ajoutant seulement au momemt de la récupération :

```
CLS
DIM PM%(4)      'PM comme Programme Machine
SegPM% = VARSEG(PM%(0))
OffPM% = VARPTR(PM%(0))
'Passage au segment du code machine
DEF SEG = SegPM%
'Initialisation du tableau par le code machine
FOR i% = 0 TO 7
  READ Octet$
  POKE OffPM% + i%, VAL("&H" + Octet$)
NEXT i%
'Visualisation du contenu du debut de ce segment avant execution
PRINT PEEK(0), PEEK(1), PEEK(2), PEEK(3), PEEK(4)
PRINT PEEK(5), PEEK(6), PEEK(7), PEEK(8), PEEK(9)
'Execution du programme machine
CALL ABSOLUTE(OffPM%)
'Visualisation du contenu du debut de ce segment apres execution
PRINT PEEK(0), PEEK(1), PEEK(2), PEEK(3), PEEK(4)
PRINT PEEK(5), PEEK(6), PEEK(7), PEEK(8), PEEK(9)
'Retour au segment de d\'epart
DEF SEG
PRINT PM%(4)
END
DATA B8,11,00
DATA 2E,A3,08,00
DATA HCB
```

Commentaire.- N'oubliez pas de changer le type de la variable `Octet` dans cette deuxième amélioration !