Chapitre 6

Manipulation des bits

Nous avons vu comment accéder aux octets, qui est la plus petite unité adressable. Cependant la plus petite quantité d'information est le bit. On n'a pas à y accéder lors de la manipulation des chaînes de caractères et on a rarement intérêt à y accéder pour effectuer des calculs (mais cela peut être interessant pour la multiplication, par exemple). La manipulation des bits est, par contre, intéressante en programmation système, plus exactement pour contrôler des périphériques. En effet on a souvent besoin, dans ce cas, de coder de l'information binaire : on pourrait utiliser un octet pour une telle information mais on perdrait alors beaucoup d'espace et, surtout, beaucoup de temps lors des échanges avec les périphériques. On a donc intérêt à coder ces informations par des bits individuels, ce qui permet l'utilisation d'un octet au lieu de huit (tout au moins dans le cas optimum). Il faut donc pouvoir accéder à chacun des bits individuellement pour cela. C'est l'objet des *instructions de manipulation de bits*.

6.1 Instructions logiques

6.1.1 Mise en place

Introduction.- Les **instructions logiques** correspondent à l'implémentation de certains connecteurs logiques tels que la négation, la conjonction et la disjonction. Un octet peut être considéré comme un octuplet de bits. Lors d'une instruction logique, on exécute la même opération logique en parallèle sur chacun des huit bits, 1 représentant le vrai et 0 le faux.

Liste des instructions logiques du `i8086`.- Il y a une instruction unaire :

```
NOT
```

et trois instructions binaires :

```
AND
OR
XOR
```

représentant la négation (le 'non' parallèle), la conjonction (le 'et' parallèle), la disjonction inclusive (le 'ou' parallèle) et la disjonction exclusive (le 'ou bien ou bien' parallèle).

Opérandes.- Les opérandes peuvent être des constantes, des registres ou des emplacements mémoire avec les seules configurations suivantes :

```
NOT registre
NOT memoire
```

pour `NOT` et :

```
OP registre,constante
OP memoire,constante
OP registre1,registre2
OP registre,memoire
OP memoire,registre
```

pour les instructions binaires.

Exemple.- Si le contenu de `ax` est :

0000101011100011

et celui de `bx` est :

1001100000100001

alors celui des `ax and bx` est :

0000100000100001

celui de `ax or bx` est :

1001101011100011

et celui de `ax xor bx` est :

1001000011000010.

Exemple.- Vérifions les résultats ci-dessus en utilisant `debug` :

```
C:\>debug
-a
156B:0100 mov ax,0AE3
156B:0103 mov bx,9821
156B:0106 or  ax,bx
156B:0108 int3
156B:0109
-g

AX=9AE3  BX=9821  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=156B  ES=156B  SS=156B  CS=156B  IP=0108   NV UP EI NG NZ NA PO NC
156B:0108 CC            INT     3
-q
```

Ce n'est pas très facile car c'est à nous de transformer à la main :

`ax = 0000101011100011b = 0000 1010 1110 0011b = 0AE3h`

`bx = 1001100000100001b = 1001 1000 0010 0001b = 9821h`

et de décoder le résultat :

`ax = 9AE3h = 1001 1010 1110 0011b`

pour vérifier qu'il est juste.

6.1.2 Application au masquage

On veut souvent déterminer la valeur d'un ou plusieurs bits. On pourrait évidemment amener chacun de ces bits à la première position (grâce aux instructions de décalage que nous verrons ci-après) et tester. Mais ceci représente beaucoup d'instructions. On utilise donc plutôt la technique du **masquage** que nous allons décrire maintenant.

6.1.2.1 Masquage simple

Introduction.- On parlera de **masquage simple** si on veut connaître la valeur d'un seul bit.

Principe.- L'instruction `and` affecte tous les indicateurs arithmétiques et peut donc être utilisée pour déterminer si le bit d'une certaine position est égal à 0 ou à 1. Par exemple, pour déterminer si le bit de la position 3 du registre `al` est égal à 1, on place `00001000b`, valeur appelé **masque**, dans le registre `bl` et on exécute l'instruction `and al,bl`. Il peut y avoir au plus un 1 dans le résultat : à la position 3. De plus on obtient un tel 1 si, et seulement si, la valeur du troisième bit de `al` est égale à 1. Ainsi si le troisième bit de `al` est égal à 1 si, et seulement, l'indicateur `ZF` est égal à 0 après cette instruction.

6.1.2.2 Masquage et traduction

Introduction.- Le masquage peut servir pour déterminer la valeur d'un bit, comme nous venons de le voir. Il peut également servir dans certain cas à la traduction.

Application.- Dans le codage ASCII, les chiffres '0' à '9' sont codés par les valeurs `30h` à `39h`. Lorsqu'on récupére le code ASCII d'un chiffre, pour obtenir la valeur numérique correspondante, il suffit donc de soustraire `30h`. Ceci peut

être obtenu par une soustraction mais on peut aussi utiliser une technique de masquage.

Supposons en effet que le code ASCII se trouve dans le registre `al`. Plaçons `0Fh`, c'est-à-dire `00001111b` dans le registre `bl`. Alors `and al,bl` donne la valeur numérique du chiffre correspondant.

6.1.2.3 Masquage multiple

On peut vouloir tester plusieurs bits. On peut le faire un par un. On peut aussi les tester tous à la fois, mais uniquement pour savoir s'ils sont tous présents (avec `xor`) ou si au moins l'un d'eux est présent (avec `or`).

6.1.3 Le test

L'inconvénient d'utiliser l'opération `and` pour déterminer la présence de bits est qu'elle détruit la valeur du premier registre.

L'instruction `test` agit comme l'instruction `and` au niveau des indicateurs mais ne change pas la valeur de la destination.

6.2 Les décalages logiques

Introduction.- Puisque les microprocesseurs manipulent des nombres en binaire, ils peuvent effectuer facilement une multiplication ou une division par deux, en décalant tous les bits d'une position à gauche (éventuellement avec un débordement) ou à droite (en perdant le premier bit). Ceci explique l'intérêt des **décalages**.

Les concepteurs du microprocesseur `i8086` ont implémenté deux familles de décalages : l'une pour multiplier et diviser par deux les entiers naturels (appelés **décalages logiques**) et l'autre pour les entiers relatifs (appelés **décalages arithmétiques**). Nous allons nous intéresser au premier type de décalages dans ce chapitre, renvoyant l'étude des décalages arithmétiques à leur place naturelle, au chapitre 12.

Instructions.- L'instruction :

```
SHL registre,1
SHL memoire,1
```

(pour l'anglais *SHift Left*) décale tous les bits d'une position vers la gauche et place un zéro comme bit le plus à droite. Lorsqu'il y a débordement, l'indicateur de débordement `CF` est positionné à un.

L'instruction :

```
SHR registre,1
SHR memoire,1
```

(pour l'anglais *SHift Right*) décale tous les bits d'une position vers la droite et place un zéro comme bit le plus à gauche. Le bit de droite est placé comme indicateur de débordement `CF`.

Exemple.- Utilisons `debug` pour vérifier que le décalage à droite de `8Fh`, soit `1000 1111b`, donne bien `0100 0111b`, soit `47h` :

```
C:\>debug
```

```
-a
15BF:0100 mov al,8F
15BF:0102 shr al,1
15BF:0104 int3
15BF:0105
-g

AX=0047  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=15BF  ES=15BF  SS=15BF  CS=15BF  IP=0104   OV UP EI PL NZ NA PE CY
15BF:0104 CC            INT     3
-q
```

On remarquera que l'indicateur de retenue est positionné puisque le bit de poids le plus faible de `al` était 1.

6.3 Les rotations

Lorsqu'on effectue un décalage à gauche de 4 bits, les 4 bits les plus à gauche sont perdus (à part l'un d'eux qui peut être récupéré dans l'indicateur de débordement). Dans certains cas, on ne veut pas perdre ces bits. La famille des **instructions de rotation** permet de réarranger les bits sans les perdre.

Il existe deux types de rotations : l'une ne concerne que l'octet sur lequel on travaille, l'autre fait intervenir l'indicateur de retenue.

6.3.1 Les rotations sur un octet

Syntaxe.- Les instructions de rotation sur un octet peuvent prendre l'une des formes suivantes :

```
ROL memoire/registre,1
ROL memoire/registre,CL
ROR memoire/registre,1
ROR memoire/registre,CL
```

avec `ROL` pour l'anglais *ROtate Left* et `ROR` pour l'anglais *ROtate Right*.

Sémantique.- Ces instructions permettent une rotation à droite ou à gauche.

Exemple.- Si le contenu de `AL` est `0100 0011b`, soit `43h` alors l'exécution de `ROR AL,1` place `1010 0001b` dans `AL`, soit `A1h`.

Utilisons `debug` pour le vérifier :

```
C:\>debug
-a
15BF:0100 mov al,43
15BF:0102 ror al,1
15BF:0104 int3
15BF:0105
-g

AX=00A1  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=15BF  ES=15BF  SS=15BF  CS=15BF  IP=0104   OV UP EI PL NZ NA PO CY
15BF:0104 CC            INT     3
-q
```

6.3.2 Les rotations sur un octet et CF

Syntaxe.- Les instructions de rotation sur un octet et `CF` peuvent prendre l'une des formes suivantes :

```
RCL memoire/registre,1
RCL memoire/registre,CL
RCR memoire/registre,1
RCR memoire/registre,CL
```

avec `RCL` pour l'anglais *Rotate through Carry Left* et `RCR` pour l'anglais *Rotate through Carry Right*.

Sémantique.- Ces instructions permettent une rotation à droite ou à gauche sur 9 bits en utilisant l'indicateur de retenue `CF` comme neuvième bit.

Exemple.- Si le contenu de `AL` est `0100 0011b`, soit 43h, et celui de `CF` est 0 alors l'exécution de `RCR AL,1` place `0010 0001b`, soit `21h` dans `AL` et 1 dans `CF`.

Utilisons `debug` pour le vérifier :

```
C:\>debug
-a
15BF:0100 mov al,43
15BF:0102 ror al,1
15BF:0104 int3
15BF:0105
-g

AX=00A1  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=15BF  ES=15BF  SS=15BF  CS=15BF  IP=0104   OV UP EI PL NZ NA PO CY
15BF:0104 CC            INT     3
-q
```

Oon remarquera que l'indicateur de retenue est positionné.

6.3.3 Changer l'indicateur de débordement

Introduction.- Pour une meilleure utilisation des rotations sur un octet (ou un mot) et `CF`, on doit pouvoir changer la valeur de `CF`.

Les instructions.- On peut initialiser la valeur de `CF` grâce aux deux instructions suivantes :

```
STC
```

(pour *SeT Carry flag*) qui place 1 dans `CF` et :

```
CLC
```

(pour *CLear Carry flag*) qui place 0 dans `CF`.