Chapitre 12

Compléments sur les entiers

Nous avons étudié au chapitre 4 les opérations sur les entiers naturels. Ceci est en théorie suffisant puisque les entiers relatifs, par exemple, sont émulables par des entiers naturels. Ceci conduit cependant à une implémentation des entiers relatifs qui n'est pas efficace. Les concepteurs des microprocesseurs câblent donc les opérations sur les entiers relatifs. Nous allons étudier dans ce chapitre ce qu'il en est pour le microprocesseur `i8086`. Mais avant cela, nous allons passer en revue une représentation proche de la numération décimale.

12.1 Le décimal codé binaire compact

12.1.1 Notion

Un entier naturel peut être représenté de différentes façons : dans un *système de représentation additifs*, tel que la numération en *chiffres romains*, ou dans un *système de représentation positionnel*, en utilisant le plus souvent des *chiffres arabes*. La notion de *base* est importante : nous utilisons le plus souvent la base dix dans la vie de tous les jours mais aussi quelquefois d'autres bases, par exemple la base douze pour les heures. L'informatique nous a habitué à utiliser la base deux (*nombres binaires*) et la base seize (*nombres hexadécimaux*) pour plus de lisibilité.

Pour des raisons d'efficacité, de nos jours tous les calculs sur ordinateurs sont effectués en base deux, avec une traduction lors de l'entrée et une autre pour le résultat final.

Cette façon de faire n'a pas été adoptée d'emblée aux débuts du développement des ordinateurs. On essayait, coûte que coûte, de rester proche de la numération décimale. Une représentation des entiers était, par exemple, en *DCB* sur lequel nous allons revenir. Le microprocesseur `i8086` conserve encore des opérations pour ce type de représentation, pour des raisons de compatibilité ascendante avec le tout premier microprocesseur, le `i4004`. Celui-ci avait été conçu sur commande pour réaliser des calculatrices de bureau. L'efficacité des calculs n'était pas la raison première alors qu'on avait à afficher pratiquement le résultat de chaque pas de calcul, d'où l'intérêt d'une représentation proche du décimal.

Définition 1.- *En* **décimal codé binaire** *(***DCB** *en abrégé ou* **BCD** *pour l'anglais Binary Coded Decimal), tout chiffre décimal est représenté sur un* **quartet**, *c'est-à-dire quatre bits binaires (*nibble *en anglais)* :

Décimal	DCB
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

qui est tout simplement la représentation binaire du chiffre correspondant. Les quartets allant de `1010b` *à* `1111b` *n'ont pas de sens en DCB.*

La plus petite unité de mémoire adressable par le microprocesseur `i8086` est l'octet et non le quartet.

Définition 2.- *On parle de* **DCB compacté** *lorsqu'on représente deux chiffres DCB sur un octet et de* **DCB étendu** *(ou normal) lorsqu'on n'en représente*

qu'un seul (le quartet de poids faible).

12.1.2 Addition décimale

Principe.- Pour travailler plus facilement sur la représentation décimale, on représente les nombres en DCB compacté. On exécute l'addition de façon habituelle (l'addition sera donc effectuée en hexadécimal) puis on doit corriger le résultat obtenu, grâce à l'instruction :

```
DAA
```

(pour l'anglais *Decimal Adjust for Addition*) qui doit être effectuée immédiatement après l'instruction d'addition. Cette instruction n'agit que sur le registre `al`.

Exemple 1.- Écrivons un programme `debug` qui additionne 12 à 63 :

```
C:\>debug
-a
15BD:0100 mov ax,63
15BD:0103 add ax,12
15BD:0106 daa
15BD:0107 int3
15BD:0108
-g

AX=0075  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=15BD  ES=15BD  SS=15BD  CS=15BD  IP=0107   NV UP EI PL NZ NA PO NC
15BD:0107 CC            INT     3
-q
```

Remarquons que l'initialisation des nombres en DCB est facile avec `debug` puisqu'il suffit de les entrer comme si on pouvait travailler directement avec du décimal. Il s'agit en fait d'hexadécimal sans permettre les chiffres 'A' à 'F'. De même, la lecture en est également facile.

Exemple 2.- L'instruction tient également compte des retenues, comme le montre l'addition de 59 et de 12 :

```
C:\>debug
-a
15BD:0100 mov al,59
15BD:0102 add al,12
15BD:0104 aad
15BD:0106
-g

AX=0071  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=15BD  ES=15BD  SS=15BD  CS=15BD  IP=0107   NV UP EI PL NZ AC PE NC
15BD:0107 CC            INT     3
-q
```

12.1.3 Soustraction décimale

Principe.- De même, on exécute la soustraction de façon habituelle puis on corrige le résultat obtenu, grâce à l'instruction :

```
DAS
```

(pour l'anglais *Decimal Adjust for Substraction*) qui doit être effectuée immédiatement après l'instruction de soustraction et qui n'agit que sur le registre `al`.

Exemple.- Écrivons un programme `debug` qui soustrait 12 à 63 :

```
C:\>debug
-a
15BD:0100 mov al,63
15BD:0102 sub al,12
15BD:0104 das
15BD:0105 int3
15BD:0106
-g

AX=0051  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=15BD  ES=15BD  SS=15BD  CS=15BD  IP=0105   NV UP EI PL NZ NA PO NC
15BD:0105 CC            INT     3
-q
```

12.1.4 Cas de la multiplication et de la division

Comme nous l'avons déjà dit, la multiplication et la soustraction ne sont pas implémentées sur tous les microprocesseurs, car elles exigent un câblage complexe. En particulier, elles ne l'étaient pas sur le tout premier microprocesseur, le `i4004`. On ne trouve donc rien sur celles-ci en BCD compacté sur le `i8086`.

12.2 BCD non compacté

12.2.1 Notion

Comme nous l'avons déjà dit, les microprocesseurs ne savent pas comment les caractères sont codés. Ils ne connaissent pas le code utilisé. Le microprocesseur `i8086` suppose seulement qu'un caractère est codé sur un octet et que les chiffres décimaux sont codés de façon contigu de `X0h` à `X9h`, pour '0' à '9', sans connaître la valeur de `X`. Ceci est le cas du code ASCII, les chiffres décimaux ayant les codes `30h` à `39h`.

Dans la pratique on entrera les chiffres comme caractères, donc un chiffre sera codé sur un octet et non sur un quartet. Le BCD non compacté permet d'effectuer une conversion rapide entre un chiffre entré comme caractère et la représentation BCD non compactée associée. En utilisant comme masque celui dont le quartet de poids fort égal à zéro et le quartet de poids faible égal à `1111b`, on traduit facilement le caractère en chiffre correspondant.

12.2.2 Opérations

Le microprocesseur `i8086` dispose d'instructions spécifiques pour manipuler les opérations en BCD non compacté.

12.2.2.1 Addition

Principe.- On exécute l'addition de deux entiers en BCD non compacté de façon habituelle (l'addition sera donc effectuée en hexadécimal) puis on corrige le résultat obtenu, grâce à l'instruction :

```
DAA
```

(pour l'anglais *Ascii Adjust for Addition*) qui doit être effectuée immédiatement après l'instruction d'addition. Cette instruction n'agit que sur le registre `ax`.

Remarque.- L'interprétation du premier 'A' pour Ascii n'est pas d'origine. Comme nous l'avons déjà dit, cette instruction n'est pas particulièrement liée au code ASCII. En fait l'un des 'A' faisait référence à 'accumulateur' (ou à `ax`, comme l'on veut).

Exemple.- Écrivons un programme `debug` qui initialise le registre `al` à 6 (ou à `36h` si on préfère), qui lui ajoute 5 et qui corrige le résultat pour le BCD non compacté :

```
C:\>debug
C:\>debug
-a
15BD:0100 mov ax,6
15BD:0103 add al,5
15BD:0105 aaa
15BD:0106 int3
15BD:0107
-g

AX=0101  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=15BD  ES=15BD  SS=15BD  CS=15BD  IP=0106   NV UP EI PL NZ AC PO CY
15BD:0106 CC            INT     3
-q
```

Le résultat est placé dans le registre `ax`. On a bien 11 en BCD non compacté.

12.2.2.2 Soustraction

Principe.- De même on exécute la soustraction de deux entiers en BCD non compacté de façon habituelle puis on corrige le résultat obtenu, grâce à l'instruction :

`AAS`

(pour l'anglais *Ascii Adjust for Substraction*) qui doit être effectuée immédiatement après l'instruction de soustraction et qui n'agit que sur le registre `ax`.

Exemple.- Écrivons un programme `debug` qui initialise le registre `al` à 8, qui lui soustrait 3 et qui corrige le résultat :

```
C:\>debug
-a
15BD:0100 mov ax,8
15BD:0103 sub al,3
15BD:0105 aas
15BD:0106 int3
15BD:0107
-g

AX=0005  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=15BD  ES=15BD  SS=15BD  CS=15BD  IP=0106   NV UP EI PL NZ NA PE NC
15BD:0106 CC            INT     3
-q
```

12.2.2.3 Multiplication

Principe.- On exécute la multiplication de deux entiers en BCD non compacté de façon habituelle puis on corrige le résultat obtenu, grâce à l'instruction :

`AAM`

(pour l'anglais *Ascii Adjust for Multiplication*) qui doit être effectuée immédiatement après l'instruction de multiplication et qui n'agit que sur le registre `ax`.

Exemple.- Écrivons un programme `debug` qui initialise le registre `al` à 8, le registre `bl` à 3, qui multiplie le contenu de l'accumulateur par celui du registre `bl` et qui corrige le résultat pour le BCD non compacté :

```
C:\>debug
-a
15BD:0100 mov al,8
15BD:0102 mov bl,3
15BD:0104 mul bl
15BD:0106 aam
15BD:0108 int3
15BD:0109
-g

AX=0204  BX=0003  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=15BD  ES=15BD  SS=15BD  CS=15BD  IP=0108   NV UP EI PL NZ NA PO NC
15BD:0108 CC            INT     3
-q
```

Le résultat dans `ax` est bien 24 en BCD non compacté.

12.2.2.4 Division euclidienne

Principe.- Avant d'effectuer la division euclidienne d'un entier se trouvant dans le registre `ax` en BCD non compacté, on doit représenter celui-ci sous forme hexadécimale grâce à l'instruction :

```
AAD
```

(pour l'anglais *Ascii Adjust before Division*).

Exemple.- Écrivons un programme `debug` qui initialise le registre `ax` à 25 en BCD non compacté, qui représente ce nombre en hexadécimal puis qui divise son contenu par 3 :

```
C:\>debug
-a
15BD:0100 mov ax,205
15BD:0103 aad
15BD:0105 mov bl,3
15BD:0107 div bl
15BD:0109 int3
15BD:010A
-g

AX=0108  BX=0003  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=15BD  ES=15BD  SS=15BD  CS=15BD  IP=0109   NV UP EI PL NZ NA PO NC
15BD:0109 CC            INT     3
-q
```

Après exécution du programme, le registre `ax` indique bien un quotient égal à 3 et un reste égal à 1.

12.3 Entiers relatifs

12.3.1 Représentations

12.3.1.1 Bit de signe et valeur absolue

La première idée pour représenter un nombre négatif est d'utiliser un **bit de signe** et les autres bits pour la valeur absolue. Il est traditionnel de réserver le bit de plus haut poids comme bit de signe avec la convention :

- 0 représente le signe plus (+) ;
- 1 représente le signe moins (-).

Dans ces conditions, l'intervalle couvert sur un octet est de :

- `0000 0000b` à `0111 1111b` pour les entiers positifs, soit de 0 à 127 ;
- `1000 0000b` à `1111 1111b` pour les entiers négatifs, soit de -0 à -127.

Cette méthode de représentation présente trois inconvénients :

- Il n'y a pas compatibilité ascendante lorsqu'on décide, par exemple, de passer de un octet à deux octets : `1000 0001b` représente -1 sur un octet ; lorsqu'on lui ajoute un octet (nul) à gauche, on obtient `0000 0000 1000 0001b`, soit 129 ; si on lui ajoute un octet (nul) à droite, on obtient `1000 0001 0000 0000b`, soit -257.
- Zéro n'a pas une représentation unique mais deux représentations : `0000 0000b`, soit +0, et `1000 0000b`, soit -0.
- L'algorithme pour effectuer l'addition de deux entiers relatifs va distinguer quatre cas suivant le signe de chacun des deux termes.

Ces inconvénients ont conduit à choisir une autre représentation des entiers relatifs.

12.3.1.2 Complément à un

Dans la méthode dite du **complément à un**, le bit de plus grand poids est également le bit de signe mais les autres bits ne représentent pas la valeur absolue :

- Les entiers positifs sont représentés comme dans la première méthode.
- La représentation d'un entier négatif est la valeur absolue de cet entier dont chaque bit est inversé (1 est remplacé par 0 et 0 par 1, soit x par $1 - x$, d'où le nom de la méthode). Par exemple -5 sera représenté sur un octet par `1111 1010b`, puisque $5 =$ `0000 0101b`.

Remarquons qu'il n'y a toujours pas compatibilité ascendante avec cette méthode et que zéro a toujours deux représentations.

12.3.1.3 Complément à deux

Principe.- L'algorithme d'addition des entiers relatifs n'est pas facile à mettre en place avec la représentation par (bit de) signe et valeur absolue. On peut utiliser le fait que ce ne sont pas *tous* les entiers relatifs que l'on représente mais

seulement ceux qui tiennent sur N bits, avec N plus ou moins grand (N = 8 pour un octet).

Le principe repose sur le fait qu'en arithmétique modulaire :

$$a - b\ [mod\ c] = a + (c - b)\ [mod\ c]$$

Si on considère des entiers de N chiffres au plus en base R, chaque entier est plus petit que R^{N+1} et :

$$a - b\ [mod\ R^{N+1}] = a + (R^{N+1} - b)\ [mod\ R^{N+1}]$$

De ce fait, $R^{N+1} - b$ s'appelle le **complément à R** de b. Remarquons que la dénomination n'est pas bien choisie puisque'elle fait référence à la base R mais pas au nombre de chiffres N. Dans le cas du binaire, on parlera donc de **complément à deux**.

Calcul du complément à deux.- Puisque :

$$2^{N+1} - b = (2^{N+1} - 1) - B + 1$$

et que $2^{N+1} - 1$ est l'entier de N chiffres binaires tous égaux à 1, le calcul du complément à deux d'un entier (positif ou négatif) est facile :

- on calcule son complément à un (opération NOT) ;
- on lui ajoute un.

Calculons la représentation de -5 en complément à deux sur un octet :

- considérons sa valeur absolue : 5 = `0000 0101b` ;
- calculons son complément à un : `1111 1010b` ;
- ajoutons-lui 1 (et ignorons la retenue finale lorsqu'elle existe) : `1111 1010b + 1` = `1111 1011b`.

Avantages.- Dans ce système zéro a une représentation et la soustraction se ramène à une addition. Par contre il n'y a toujours pas compatibilité ascendante.

Remarque.- Le bit de plus haut poids peut encore être considéré comme bit de signe car il est égal à 1 pour les entiers compris entre -2^N et -1 età zéro pour ceux compris entre 0 et $2^N - 1$.

12.3.2 Cas du microprocesseur `i8086`

Le microprocesseur `i8086` a un jeu d'instructions spécifiques pour travailler avec les entiers relatifs représentés en complément à deux.

12.3.2.1 Calcul de l'opposé

On a vu que le calcul de l'opposé $-b$ d'un nombre b en complément à deux est facile puisqu'il suffit d'une conjonction bit à bit et d'une incrémentation. Cependant le microprocesseur `i8086` a câblé cette opération en une seule instruction, dite de **négation** là où on pourrait s'attendre à *opposition* :

`NEG` *operande*

où l'opérande est de huit ou de seize bits.

Exemple.- Vérifions avec `debug` que l'opposé de 5 est bien `1111 1011b` = `FBh` :

```
C:\>debug
-a
15BA:0100 mov al,5
15BA:0102 neg al
15BA:0104 int3
15BA:0105
-g

AX=00FB  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=15BA  ES=15BA  SS=15BA  CS=15BA  IP=0104   NV UP EI NG NZ AC PO CY
15BA:0104 CC            INT     3
-q
```

12.3.2.2 Addition et soustraction

Introduction.- Le complément à eux a été choisi pour que l'addition de deux entiers relatifs puisse s'effectuer avec les instructions d'addition câblées *a priori* pour les entiers naturels. Il n'y a donc pas grand chose à en dire. L'utilisateur choisit l'interprétation qu'il veut pour les octets ou les mots : entiers naturels ou entiers relatifs ; il effectue l'addition et il interprète le résultat de la façon qu'il a choisi.

Pour la soustraction, on utilise le fait que :

`a - b = a + neg(b)`

Exemple.- Écrivons un programme `debug` qui permet d'effectuer l'addition de `25h` et de `-12h` et vérifions que cela donne bien `13h` :

```
C:\>debug
-a
15BD:0100 mov al,25
15BD:0102 mov bl,12
15BD:0104 neg bl
15BD:0106 add al,bl
15BD:0108 int 3
15BD:0109
-g

AX=0013  BX=00EE  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=15BD  ES=15BD  SS=15BD  CS=15BD  IP=0108   NV UP EI PL NZ AC PO CY
15BD:0108 CC            INT     3
-q
```

12.3.2.3 Multiplication et division

Introduction.- La représentation des entiers relatifs par le complément à deux nous empêche d'utiliser les instructions `mul` et `div`, câblées pour les entiers naturels, pour effectuer une multiplication ou une division sur les entiers relatifs. Il y a donc deux nouvelles instructions pour le `i8086` :

`imul` *source*

(pour l'anglais *Integer MULtiply*) avec les mêmes règles de syntaxe que pour `mul` et :

`idiv` *diviseur*

(pour l'anglais *Integer DIVide*) avec les mêmes règles de syntaxe que pour `div`.

Exemple.- Écrivons un programme `debug` qui permette, dans un premier temps, de multiplier 5 par -3 puis, dans un deuxième temps d'effectuer la négation du résultat pour une meilleure lisibilité :

```
C:\>debug
-a
15BD:0100 mov al,5
15BD:0102 mov bl,3
15BD:0104 neg bl
15BD:0106 imul bl
15BD:0108 neg al
15BD:010A int3
15BD:010B q
erreur
15BD:010B
-g

AX=FF0F  BX=00FD  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=15BD  ES=15BD  SS=15BD  CS=15BD  IP=010A   NV UP EI PL NZ AC PE CY
15BD:010A CC            INT     3
-q
```

À la fin le contenu du registre `al` est bien `Fh`, c'est-à-dire 15.

12.3.2.4 Conversions

Nous avons fait remarquer plus haut qu'il n'y avait pas compatibilité ascendante pour le complément à deux en passant de huit à seize bits, d'où l'intérêt des deux instructions suivante :

```
CBW
```

(pour l'anglais *Convert Byte to Word*) qui convertit l'entier relatif de huit bits placé dans le registre `al` en le même entier relatif, mais sur seize bits maintenant, placé dans le registre `ax` et :

```
CWD
```

(pour l'anglais *Convert Word to Double word*) qui convertit l'entier relatif de seize bits placé dans le registre `ax` en le même entier relatif, mais sur trente-deux bits maintenant, placé dans la paire de registres `dx:ax`.

12.3.2.5 Décalages arithmétiques

Introduction.- Les **décalages arithmétiques** permettent de multiplier ou diviser par deux les entiers relatifs, autrement dit le bit de signe n'est pas changé.

Instructions.- L'instruction :

```
SAL registre,1
SAL memoire,1
```

(pour l'anglais *Shift Arithmetic Left*) décale les six bits de poids faible d'une position vers la gauche et place un zéro comme bit le plus à droite. L'indicateur de retenue `CF` prend la valeur de l'ancien septième bit.

L'instruction :

```
SAR registre,1
SAR memoire,1
```

(pour l'anglais *SHift Arithmetic Right*) décale les sept bits de poids fort d'une position vers la droite et laisse en place le bit de poids fort. L'indicateur de retenue `CF` prend la valeur de l'ancien bit de poids le plus faible.

Exemple.- Utilisons `debug` pour vérifier que le décalage arithmétique à droite de `8Fh`, soit `1000 1111b`, donne bien `1100 0111b`, soit `C7h` :

```
C:\>debug
-a
15BF:0100 mov al,8F
15BF:0102 sar al,1
15BF:0104 int3
15BF:0105
-g

AX=00C7  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=15BF  ES=15BF  SS=15BF  CS=15BF  IP=0104   NV UP EI NG NZ NA PO CY
15BF:0104 CC            INT     3
-q
```