Chapitre 17

Les entiers relatifs

Nous avons vu, dans les chapitres précédents, les instructions fondamentales d'un microprocesseur, à savoir celles qui permettent d'émuler une machine de Turing. Nous allons commencer à voir, dans ce chapitre, des notions « avancées », c'est-à-dire non indispensables en théorie.

Nous avons étudié au chapitre 15 les opérations sur les entiers naturels. Ceci est en théorie suffisant pour la programmation puisque, en théorie, les entiers relatifs, par exemple, sont émulables par des entiers naturels. Cette émulation conduit cependant à une implémentation des entiers relatifs qui n'est pas efficace. Les concepteurs des processeurs câblent donc des opérations sur les entiers relatifs.

Profitons de ce chapitre sur les entiers relatifs pour voir un complément sur les entiers naturels. Bien qu'employant la représentation binaire, les premiers concepteurs des ordinateurs voulaient être très proches de la représentation décimale. Ceci a conduit à un codage dit BCD. Celui-ci est toujours important pour les calculettes et les calculs financiers. Bien que d'une importance relative, nous l'étudierons dans ce chapitre.

17.1 Entiers relatifs

17.1.1 Représentation des entiers relatifs

Nous avons vu que la première étape pour travailler avec un ensemble (dénombrable) sur un ordinateur est de coder ses éléments. Voyons comment le faire pour les entiers relatifs. Il existe plusieurs façons de le faire, les unes plus naturelles, les autres plus efficaces.

17.1.1.1 Bit de signe et valeur absolue

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

– 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 naturelle de représentation présente trois inconvénients :

– Il n'y a pas compatibilité ascendante lorsqu'on décide, par exemple, de passer d'une représentation sur un octet à une représentation sur 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 doit distinguer quatre cas suivant le signe de chacun des deux termes.

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

17.1.1.2 Complément à un

Définition.- *Dans la représentation dite du* **complément à un**, *le bit de poids le plus fort 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 représentation).*

Exemple.- L'entier relatif -5 sera représenté sur un octet par `1111 1010b`, puisque 5 = `0000 0101b`.

Remarque.- Il n'y a toujours pas compatibilité ascendante avec cette représentation et zéro a toujours deux représentations.

17.1.1.3 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).

L'idée d'une représentation permettant de mettre facilement en place l'addition repose alors sur le fait qu'en arithmétique modulaire on a :

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

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

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

De ce fait, $r^{n+1} - b$ s'appelle le **complément à r** de b. Dans le cas du binaire, on parlera donc de **complément à deux**.

Remarque.- La dénomination n'est pas bien choisie puisqu'elle ne fait référence qu'à la base r mais pas au nombre maximal de chiffres n. Il vaudrait mieux parler de **complément à r sur n bits** mais la tradition est maintenant bien établie.

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 logique NOT bit à bit) ;
- on lui ajoute un.

Exemple.- 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 seule représentation et la soustraction se ramène à une addition (puisqu'on utilise tout simplement une arithmétique modulaire). Par contre il n'y a toujours pas compatibilité ascendante.

Remarques.- 1^o) Quelle est l'interprétation d'un octet lorsqu'il représente un entier ? C'est l'utilisateur qui choisit une des deux interprétations suivantes suivant le cas :

- S'il ne doit travailler qu'avec des entiers naturels, il choisit dans ce cas que tous les mots de n bits représentent des entiers naturels, variant de 0 à $2^n - 1$.
- S'il doit travailler à la fois avec des entiers positifs et des entiers négatifs, il choisit dans ce cas que tous les mots de n bits représentent des entiers relatifs variant de -2^{n-1} à $2^{n-1} - 1$. Le bit de poids le plus fort peut alors encore être considéré comme bit de signe car il est égal à 1 pour les entiers compris entre -2^{n-1} et -1 et à zéro pour ceux compris entre 0 et $2^{n-1} - 1$.

- 2^o) Si on procède deux fois successives à la formation du complément à deux, on retombe sur le nombre qu'on avait au départ. L'opération de complémentation à deux est donc *idempotente*.

17.1.2 Cas du microprocesseur 8086/8088

Le microprocesseur 8086/8088 possède un jeu d'instructions spécifiques pour travailler avec les entiers relatifs représentés en complément à deux sur huit ou seize bits.

17.1.2.1 Calcul de l'opposé

Principe.- 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 négation bit à bit suivie d'une incrémentation. Les concepteurs du microprocesseur 8086 ont câblé cette opération de passage à l'opposé :

```
NEG
```

sous le nom de **négation** (*NEGation* en anglais) là où on aurait pu s'attendre à *opposition*.

Langage machine.- La négation est codée sur deux ou quatre octets par :

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

Exemple.- *Vérifions que l'opposé de 5 sur deux octets est bien* :

```
1111 1111 1111 1011b = FFFBh.
```

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

```
MOV AX,3000
MOV DS,AX
MOV AX,5
NEG AX
MOV [400],AX
RET
```

La quatrième instruction est codée en langage machine par :

```
| 1111 011 1 | 11 011 000 |
```

soit F7 D8h. La traduction de ce programme en langage machine est donc :

```
B8 00 30
8E D8
B8 05 00
F7 D8
A3 00 04
C3
```

Codons ce programme :

```
C:\COM>debug
-E CS:100 B8 00 30
-E CS:103 8E D8
-E CS:105 B8 05 00
-E CS:108 F7 D8
```

```
-E CS:10A A3 00 04
-E CS:10D C3
-R BX
BX 0000
:
-R CX
CX 0000
:0E
-N NEG.COM
-W
Ecriture de 0000E octets
-Q
```

Après exécution du programme `neg.com`, on vérifie que l'on a bien FF FBh à l'emplacement mémoire choisi :

```
C:\COM>debug
-D 3000:400
3000:0400 FB FF 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
3000:0410 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
3000:0420 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
3000:0430 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
3000:0440 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
3000:0450 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
3000:0470 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
3000:0470 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
-q
```

17.1.2.2 Addition et soustraction

Introduction.- Le complément à deux 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 : entiers naturels ou entiers relatifs ; il effectue l'addition et il interprète le résultat de la façon qu'il a choisie.

Pour la soustraction, on utilise le fait que :

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

que b soit positif ou négatif.

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

Le programme s'écrit de la façon suivante en langage symbolique, en utilisant la commutativité de l'addition :

```
MOV AX,3000
MOV DS,AX
MOV AX,12
NEG AX
ADD AX,25
MOV [400],AX
RET
```

La traduction de ce sous-programme en langage machine donne :

```
B8 00 30
8E D8
B8 12 00
F7 D8
05 25 00
A3 00 04
C3
```

Codons ce programme :

```
C:\COM>debug
-E CS:100 B8 00 30
-E CS:103 8E D8
-E CS:105 B8 12 00
-E CS:108 F7 D8
-E CS:10A 05 25 00
-E CS:10D A3 00 04
-E CS:110 C3
-R BX
BX 0000
:
-R CX
CX 0000
:011
-N ADDNEG.COM
-W
Ecriture de 00011 octets
-Q
```

Après exécution du programme `addneg.com`, on vérifie que l'on a bien 13h à l'emplacement mémoire choisi :

```
C:\COM>debug
-D 3000:400
3000:0400 13 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
3000:0410 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
3000:0420 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
3000:0430 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
3000:0440 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
3000:0450 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
3000:0470 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
3000:0470 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
-q
```

Incidence sur les indicateurs.- Nous allons voir l'intérêt de l'indicateur OF dans le cas d'une addition/soustraction sur les entiers relatifs, alors que celui-ci est affecté mais ne joue pas de rôle dans le cas des entiers naturels. En effet il faut conserver l'information sur le signe, qui peut être perdue dans deux cas :

- l'indicateur CF est mis à 1 sans retenue ajoutée au(x) bit(s) de signe,
- l'indicateur CF est mis à 0 mais une retenue a été ajoutée au bit de signe.

Considérons quatre exemples, dans le cas d'additions sur un octet pour simplifier :

a) 80h + 80h [= -128 + (-128) = -256]

```
  1000 0000
```

```
+ 1000 0000
-----------
1 0000 0000
```

Ici l'information du signe est perdue sans retenue ajoutée aux bits de signe. Le microprocesseur met les indicateurs CF et OF à 1.

b) 80h + 7Fh [= -128 + 127 = -1]

```
  1000 0000
+ 0111 1111
-----------
0 1111 1111
```

Ici tout se passe bien, il n'y a pas perte d'information. Les indicateurs CF et OF sont alors mis à 0.

c) 7Fh + 7Fh [= 127 + 127 = 254]

```
  0111 1111
+ 0111 1111
-----------
0 1111 1110
```

Ici il y pas perte d'information. L'indicateur CF est mis à 0 mais OF est mis à 1.

d) C0h + 7Fh [= -64 + 127 = 63]

```
  1100 0000
+ 0111 1111
-----------
1 0011 1111
```

Le résultat est correct mais l'indicateur CF est positionné. L'indicateur OF est alors également positionné pour que $CF \oplus OF = 0$.

17.1.2.3 Multiplication

Contrairement à l'addition et à la soustraction, la multiplication et la division ne se comportent pas de la même façon suivant qu'on considère que les opérandes sont des entiers naturels ou des entiers relatifs, d'où la nécessité d'introduire deux instructions nouvelles.

Langage symbolique.- Pour la multiplication, l'instruction est :

```
IMUL source
```

pour *Integer MULtiply*, c'est-à-dire multiplication sur les entiers relatifs, où `source` désigne un registre ou une case mémoire.

Sémantique.- Cette instruction permet de multiplier le contenu de l'accumulateur 8 bits AL (resp. 16 bits AX) par le contenu d'un registre ou d'une case mémoire 8 bits (resp. 16 bits), considérés comme des entiers relatifs. Le résultat, qui a une taille double, est placé dans AX (resp. AX et DX, ce dernier pour le mot de poids fort).

Langage machine.- Cette multiplication s'effectue grâce à une instruction codée sur deux ou quatre octets suivant le cas :

```
| 1111 011w | mod 101 r/m |         |         |
```

concernant `AX` si $w = 1$ et `AL` si $w = 0$.

Exercice corrigé.- *Traduire l'instruction* :

```
IMUL BL
```

en langage machine.

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

```
| 1111 0110 | 11 101 011 |
```

soit F6 EBh.

17.1.2.4 Division euclidienne

Langage symbolique.- L'instruction est :

```
IDIV source
```

(pour *Integer DIVision*), où le dividende de 16 bits (resp. 32 bits) est placé dans l'accumulateur `AX` (resp. `AX` et `DX`, ce dernier contenant le mot de poids fort) et le diviseur de 8 bits dans `source`, qui est un registre ou une case mémoire.

Sémantique.- Cette instruction permet d'effectuer la division euclidienne du dividende par le diviseur, le quotient exact est placé dans AL (ou AX) et le reste dans AH (ou DX).

Contrairement à la définition mathématique, la convention est que les reste est du même signe que le dividende. Ainsi dans le cas de la division « euclidienne » de ± 26 par ± 7, on a :

$$-26 = (-3) \times 7 - 5$$

$$-26 = 3 \times (-7) - 5$$

$$26 = 3 \times 7 + 5$$

$$26 = (-3) \times (-7) + 5$$

ce qui signifie que la division est d'abord non signée ($26 = 3 \times 7 + 5$) puis que les signes sont mis : au reste en fonction de celui du dividende, au quotient en fonction de ceux du dividende et du diviseur.

Langage machine.- Cette division euclidienne s'effectue grâce à l'instruction codée sur deux ou quatre octets suivant le cas :

```
| 1111 011w | mod 111 r/m |         |         |
```

Si $w = 0$, on a dividende = [AX], quotient = [AL] et reste = [AH].
Si $w = 1$, on a dividende = [DX,AX], quotient = [AX] et reste = [DX].

Exercice corrigé.- *Traduire l'instruction* :

```
IDIV BL
```

en langage machine.

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

```
| 1111 0110 | 11 111 011 |
```

soit F6 FBh.

17.2 Le décimal codé binaire

17.2.1 Notion

Un entier naturel peut être représenté de différentes façons : dans un *système de représentation additif*, 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*), mieux adaptée au matériel, et la base seize (*nombres hexadécimaux*), associée à la base deux mais en plus lisible par un être humain.

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

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 `i8080` conserve encore des instructions pour ce type de représentation, pour des raisons de compatibilité ascendante avec le tout premier microprocesseur, le `i4004`. Rappelons que celui-ci a 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 restants, de `1010b` *à* `1111b`*, n'ont pas de signification particulière en DCB.*

La plus petite unité de mémoire adressable par le microprocesseur `i8080` est l'octet et non le quartet, d'où l'intérêt du *DCB compacté*.

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 (sur le quartet de poids faible).*

17.2.2 Addition en DCB

Principe.- Pour travailler plus facilement sur la représentation décimale, on représente les nombres en DCB compacté. On exécute l'addition de façon habituelle mais on doit corriger le résultat obtenu. On utilise pour cela l'instruction :

```
DAA
```

(pour l'anglais *Decimal Adjust after Addition*) qui doit être effectuée immédiatement après l'instruction d'addition. Cette instruction, entièrement héritée du 8080, n'agit que sur l'accumulateur d'un octet, c'est-à-dire sur AL.

Langage machine.- L'instruction `DAA` est codée sur un octet par `0010 0111b`, soit 27h.

Exemple 1.- *Écrivons un programme machine qui additionne 12 à 63 et place le résultat à l'emplacement mémoire* 1000 :400.

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

```
MOV AX,3000
MOV DS,AX
MOV AL,12
ADD AL,63
DAA
MOV [400],AL
RET
```

On s'aperçoit que l'initialisation des nombres en DCB est facile pour l'utilisateur : il suffit d'entrer les entiers 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 de l'affichage (bien qu'en hexadécimal) est très lisible pour l'utilisateur.

La traduction de ce sous-programme en langage machine donne :

```
B8 00 30
8E D8
B0 12
04 63
27
A2 00 04
C3
```

Codons ce programme :

```
C:\COM>debug
-E CS:100 B8 00 30
-E CS:103 8E D8
-E CS:105 B0 12
-E CS:107 04 63
-E CS:109 27
-E CS:10A A2 00 04
-E CS:10D C3
-R BX
BX 0000
```

```
:
-R CX
CX 0000
:0E
-N DAA.COM
-W
Ecriture de 0000E octets
-Q
```

Après exécution du programme `daa.com`, on vérifie que l'on a bien 75h à l'emplacement mémoire choisi :

```
C:\COM>debug
-D 3000:400
3000:0400 75 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 u...............
3000:0410 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
3000:0420 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
3000:0430 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
3000:0440 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
3000:0450 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
3000:0470 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
3000:0470 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
-q
```

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

17.2.3 Soustraction en DCB

Principe.- Comme pour l'addition, en DCB compacté, on exécute la soustraction de façon habituelle mais on doit corriger le résultat obtenu, grâce à l'instruction :

`DAS`

(pour l'anglais *Decimal Adjust after Substraction*) qui doit être effectuée immédiatement après l'instruction de soustraction. Cette instruction n'agit que sur l'accumulateur d'un octet, c'est-à-dire sur AL.

Langage machine.- L'instruction `DAS` est codée sur un octet par `0010 1111b`, soit 2Fh.

17.2.4 Cas de la multiplication et de la division

Comme nous l'avons déjà dit, la multiplication et la division ne sont pas implémentées sur tous les microprocesseurs, car elles exigent un câblage complexe. En particulier, elles ne le sont pas sur le tout premier microprocesseur, le `i4004`. On ne trouve donc rien sur celles-ci en BCD compacté sur le `i8080`.

17.3 Historique

17.3.1 Blaise Pascal et le complément à neuf

Rappelons que Blaise Pascal a conçu l'une des toutes premières calculatrices mécaniques. Il ne pouvait effectuer que des additions sur des entiers naturels de 5, 6 ou huit chiffres décimaux (selon la version). Mathématicien expérimenté, il pensa donc à utiliser le complément à neuf pour effectuer des soustractions.