1 Représentation machine d'un entier

1.1 Problématique

L'ordinateur représente toutes les données sous formes de "**mots**" binaires de taille fixe (ex : 8, 16, 32, 64 bits). La question se pose alors de savoir **comment représenter les entiers relatifs sous cette forme de taille limitée** (un octet pour représenter un entier par exemple).

1.2 Représentation des entiers positifs

La représentation d'une entier positif correspond tout simplement à sa traduction en binaire.

> Avec un mot de n bits, on peut écrire 2^n nombres, donc on peut représenter tous les entiers positifs ou nul de **0** à ($\mathbf{2^n - 1}$).

Exemple : sur un octet, on peut représenter les entiers de 0 à 255 ($= 2^8 - 1$).

0	00000000
1	00000001
2	00000010
...	...
254	11111110
255	11111111

Le nombre de bits n nécessaires à l'écriture d'un entier positif N est donné par son logarithme (en base 2).
Plus précisément, le nombre de bits n est l'arrondi entier supérieur du logarithme de N.

Voici quelques valeurs particulières pour illustrer cela :

N	$\log_2(N)$	n	écriture
1	0	1	1
2	**1**	2	10
3	1.58	2	11
4	**2**	3	100
5	2.32	3	101
7	2.81	3	111
8	**3**	4	1000
9	3.17	4	1001
15	3.91	4	1111
16	**4**	5	10000
17	4.09	5	10001
255	7.99	8	11111111
256	**8**	9	100000000
257	8.01	9	100000001

> La somme de 2 nombres de n bits s'écrit aussi sur n bits ou $(n + 1)$ bits s'il y a une retenue.

! ALERTE ! Cela peut conduire à des erreurs dans un programme si on prévoit de faire des additions mais qu'on oublie que la machine a un nombre limité de bits pour représenter les entiers : la retenue peut être perdue !

Exemple en se limitant à 4 bits : Calcul à effectuer (ici en notation décimale) : $10 + 12 = 22$

1	0	0	0		(retenues)
	1	0	1	0	(10)
+	1	1	0	0	(12)
1	0	1	1	0	(22)

Représentation en machine : 1 0 1 0 + 1 1 0 0 = [1] 0 1 1 0
En perdant la retenue (le 5ème bit), la machine va interpréter le résultat comme étant 0110, soit 6 (au lieu de 22 : c'est gênant...) !

Le produit de 2 nombres de nombre de n bits s'écrit sur $2n$ bits.

Il faut donc aussi prévoir dans les programmes de limiter la taille des nombres à manipuler à la moitié de la taille maxi si on compte faire des multiplications.

Remarque : Python peut manipuler des nombres entiers de taille arbitraire (avec des « artifices » internes au langage).

1.3 Première tentative pour les nombres relatifs

Pour représenter les nombres relatifs (positifs ou négatifs) en binaire, une première idée simple peut venir à l'esprit : utiliser un bit de signe avant de coder la valeur absolue du nombre (par exemple 0 pour les positifs, et 1 pour les négatifs).

On aurait ainsi pour un codage sur 8 bits (1 bit de signe et 7 bits pour la valeur absolue) :
$+23_{10} = 00010111_2$
$-23_{10} = 10010111_2$

Un inconvénient est que 0 peut se coder de 2 façons : 0000 0000 (0 positif) ou 1000 0000 (0 négatif).
Mais le plus gênant est pour effectuer des additions par la suite :

Exemple : $23 + (-23) = 0$:

			1		1	1	1		(retenues)
	0	0	0	1	0	1	1	1	(+23)
+	1	0	0	1	0	1	1	1	(−23)
	1	0	1	0	1	1	1	0	(−46)
	-		32		8	4	2		(-46)

qui s'interprète comme $-(32+8+4+2)_{10} = -46 \cdots$ au lieu de $\cdots\ 0$!

L'additionneur (full adder) qu'on a vu dans le chap. P1-4 ne fonctionnerait donc pas pour l'addition des nombres relatifs.

2 Représentation des nombres relatifs en complément à 2^n

2.1 Le complément à 10^n (abrégé en complément à 10)

Une solution originale s'est imposée. Voyons d'abord le principe en base décimale.
Pour représenter un nombre négatif en se limitant à 8 chiffres, on peut décider de le représenter par le complément à 10^8 de sa valeur absolue.

Exemple : le nombre -243 se représente alors par 1 0000 0000 − 243 = 9999 9757
Que l'on peut aussi écrire : 9999 9999 − 243 + 1 = 9999 9756 + 1 = 9999 9757
Pour avoir le complément à 10, on prend le complément à 9 de chaque chiffre, et on ajoute 1.

Avec cette représentation un nombre négatif commence toujours par 9.

La magie s'opère lorsqu'on fait l'addition de deux nombres, en se souvenant bien qu'on est limité à 8 chiffres : une éventuelle retenue de débordement disparaît.

Exemples :

(1)	1	1	1	1	1	1	1		(retenues)
	0	0	0	0	0	2	4	3	(+243)
+	9	9	9	9	9	7	5	7	(−243)
(1)	0	0	0	0	0	0	0	0	(0)

(1)	1	1	1	1	1		1		(retenues)
	0	0	0	0	0	5	2	7	(+527)
+	9	9	9	9	9	7	5	7	(−243)
(1)	0	0	0	0	0	2	8	4	(+284 : positif, commence par un 0)

							1		(retenues)
	0	0	0	0	0	1	2	7	(+127)
+	9	9	9	9	9	7	5	7	(−243)
	9	9	9	9	9	8	8	4	(−116 : complément à 10^8, négatif, commence par un 9)

2.2 Le complément à 2 (2^n)

En binaire, on fait exactement la même chose avec le complément à 2^n, abrégé habituellement en « complément à 2 », (n est le nombre de bits pour représenter le nombre).

> La méthode pour coder un nombre négatif est donc de prendre le complément à 2 de sa valeur absolue. Pour cela :
>
> 1. Prendre le complément à 1 de chaque bit (⇒ inverser les bits).
> 2. Ajouter 1.

Exemple : représenter le nombre $-(94)_{10}$ en binaire en complément à 2 :

1. Conversion de 94_{10} en binaire :
   $(94)_{10} = 64 + 16 + 8 + 4 + 2 = 2^6 + 2^4 + 2^3 + 2^2 + 2^1 = (0101\ 1110)_2$
2. Complément à 1 (inversion des bits) : 0101 1110 ⇀↽ 1010 0001
3. On ajoute 1 : 1010 0001 + 1 = **1010 0010** (représentation finale de $-(94)_{10}$)

Exercice : Calculer la somme $-(94)_{10}$ et $+(33)_{10}$ en binaire en complément à 2.

		1							(retenues)
	1	0	1	0	0	0	1	0	(−94)
+	0	0	1	0	0	0	0	1	(+33)
	1	1	0	0	0	0	1	1	(−?)

Le résultat est un nombre négatif puisqu'il commence par un 1.
Il reste à décoder sa valeur absolue en prenant son complément à 2.

1. Complément à 1 : 1100 0011 ⇀↽ 0011 1100
2. On ajoute 1 : 0011 1100 + 1 = 0011 1101
3. Conversion en décimal :
   $(0011\ 1101)_2 = 2^5 + 2^4 + 2^3 + 2^2 + 2^0 = 32 + 16 + 8 + 4 + 1 = (61)_{10}$

Finalement on trouve bien : $-94 + 33 = -61$.