Nombres entiers

Représentation de l'information

Représentation d'un entier naturel

Un **entier naturel** est un nombre entier positif ou nul. Pour le représenter en mémoire, il faut choisir le nombre de bits à utiliser, ce qui détermine la fourchette des nombres représentables.

Principe. Un codage sur n bits permet de représenter tous les entiers naturels de 0 à $2^n - 1$.

Nombre de bits	Intervalle de codage	Plus grand entier
4	[0 ; 15]	$2^4 - 1 = 15$
8 (1 octet)	[0 ; 255]	$2^8 - 1 = 255$
16 (2 octets)	[0 ; 65 535]	$2^{16} - 1 = 65\,535$
32 (4 octets)	[0 ; 4 294 967 295]	$2^{32} - 1$

Exemples. $9 = 00001001_2$, $128 = 10000000_2$, $255 = 11111111_2$.

Remarque sur Python. Contrairement à la plupart des langages de programmation, Python gère les entiers avec une précision arbitraire : il n'y a pas de limite à la taille d'un entier en Python. C'est le langage qui adapte automatiquement la mémoire nécessaire.

```
1 print(2 ** 100)  # un nombre bien plus grand que 2^64
```

Résultat : 1267650600228229401496703205376

Représentation d'un entier relatif

Un **entier relatif** est un entier pouvant être négatif. Il faut donc trouver un codage qui permette de distinguer les nombres positifs des négatifs, tout en conservant les règles d'addition habituelles.

Problème de la représentation naïve

Le **bit de poids fort** est le bit le plus à gauche, celui qui correspond à la plus grande puissance de 2 ; le bit le plus à droite est le **bit de poids faible**. Une première idée serait d'utiliser le bit de poids fort comme marqueur du signe (0 pour positif, 1 pour négatif), les autres bits donnant la valeur absolue :

Notation naïve	Décimal
00000010	+2
10000010	−2

Cette représentation possède **deux inconvénients** :

— Le zéro possède deux représentations : 00000000 (soit +0) et 10000000 (soit −0). C'est une gêne mineure mais c'est un gaspillage.

— L'addition binaire usuelle donne un résultat **incorrect** lorsqu'un des nombres est négatif :

Addition en notation naïve	Décimal
00000011	3
+10000110	−6
= 10001001	−9

Le résultat attendu est −3, mais on obtient −9. Il faudrait un circuit spécial pour gérer la soustraction, ce qui complique inutilement le processeur.

Le complément à deux

L'astuce consiste à utiliser un codage appelé **complément à deux**. Cette représentation permet d'effectuer les opérations arithmétiques avec le même circuit que pour les entiers naturels.

Pour un entier relatif positif ou nul, la représentation est identique à celle d'un entier naturel, à condition que le bit de poids fort soit 0 (ce bit indique le signe : 0 pour positif, 1 pour négatif).

Intervalles de codage sur n bits en complément à deux :

Nombre de bits	Intervalle de codage
4	$[-8\ ;\ 7]$
8 (1 octet)	$[-128\ ;\ 127]$
16 (2 octets)	$[-32\,768\ ;\ 32\,767]$
32 (4 octets)	$[-2\,147\,483\,648\ ;\ 2\,147\,483\,647]$

D'une manière générale, sur n bits, on peut coder les entiers relatifs de -2^{n-1} à $2^{n-1}-1$.

Principe du complément à deux

Pour coder un **entier relatif négatif** sur n bits :

1. Écrire la valeur absolue du nombre en base 2 sur n bits.
2. **Inverser** tous les bits (les 0 deviennent des 1 et vice versa). C'est ce qu'on appelle le *complément à un*.
3. **Ajouter 1** au résultat (les dépassements sont ignorés).

Cette opération revient à calculer $2^n - |x|$.

Exemple. Codons −19 sur huit bits :

1. Valeur absolue en binaire : $19 = 00010011_2$
2. Complément à un (inversion) : 11101100
3. On ajoute 1 : 11101101

La représentation binaire de −19 sur huit bits est donc 11101101_2.

Vérification. $00010011 + 11101101 = 00000000$ (la retenue sortante est ignorée). C'est bien le comportement attendu : $19 + (-19) = 0$.

L'addition fonctionne

Vérifions que le complément à deux corrige le problème de la représentation naïve :

Addition en complément à deux	Décimal
00000011	3
+11111010	−6
= 11111101	−3

Vérifions : 11111101_2. Le bit de poids fort est 1, c'est donc un nombre négatif. Pour retrouver sa valeur absolue, on applique à nouveau le complément à deux : inversion 00000010, puis +1 = 00000011 = 3. Le résultat est bien −3.

Astuce de conversion rapide

Pour trouver de tête le complément à deux, on parcourt le nombre **de droite à gauche** en laissant inchangés les bits jusqu'au premier 1 (compris), puis on inverse tous les bits suivants.

Exemple avec 20 **:** 00010100

1. On garde la partie droite jusqu'au premier 1 : `...100`
2. On inverse tout le reste : 11101100
3. −20 s'écrit donc 11101100_2

En appliquant une deuxième fois cette astuce, on retrouve le nombre de départ.

Comment reconnaître le signe

En complément à deux, le bit de poids fort indique le signe :

— Si le bit de poids fort est **0**, le nombre est **positif** (ou nul) : on lit directement sa valeur.

— Si le bit de poids fort est **1**, le nombre est **négatif** : pour connaître sa valeur absolue, on applique le complément à deux.

Exemple. Que vaut 11101101_2 en décimal (sur huit bits) ?

Le bit de poids fort est 1, donc c'est un nombre négatif. Appliquons le complément à deux :

— Inversion : 00010010

— +1 : 00010011 = 19

Donc $11101101_2 = -19_{10}$.

Cas particulier. $10000000_2 = -128$ (sur huit bits), c'est-à-dire -2^{n-1} pour $n = 8$. Son complément à deux est 10000000 lui-même : lu comme un entier naturel, il vaut bien 128, la valeur absolue attendue. La particularité est que +128 n'est pas représentable sur huit bits en complément à deux.

Dépassement de capacité

Lorsqu'un calcul produit un résultat hors de l'intervalle de codage, on parle de **dépassement de capacité** (*overflow*). Par exemple, sur huit bits en complément à deux, 127 + 1 devrait donner 128, mais :

$$01111111 + 00000001 = 10000000$$

Or $10000000_2 = -128$ en complément à deux. Le résultat « tourne en rond » : on passe de la plus grande valeur positive à la plus petite valeur négative. C'est un piège classique en programmation (dans les langages à entiers de taille fixe comme C ou Java).

Exercices

Exercice 1

1. Coder les entiers relatifs suivants en complément à deux sur huit bits : 42, −1, −56, −128.
2. Pour chaque nombre négatif, vérifier le résultat en l'additionnant avec la représentation de sa valeur absolue : le résultat doit être 0, la retenue sortante étant ignorée (pour −128, lire 10000000 comme l'entier naturel 128).

Exercice 2 Que valent, en base dix, les entiers relatifs suivants (codés en complément à deux) ?

1. 01101100_2 (sur huit bits)
2. 11101101_2 (sur huit bits)
3. 1010101010101010_2 (sur seize bits)

Exercice 3

1. Sur huit bits, combien de nombres négatifs peut-on coder ?
2. Pourquoi y a-t-il un négatif de plus que de positifs ?
3. Quel est l'entier le plus petit codable sur 32 bits en complément à deux ?

Exercice 4 Effectuer les additions suivantes en complément à deux sur huit bits. Indiquer si un dépassement de capacité se produit.

1. 01001010 + 00110101
2. 01100100 + 01011010
3. 11110000 + 11100000

Exercice 5 Écrire une fonction Python `complement_a_deux(n, bits=8)` qui prend un entier relatif `n` et renvoie sa représentation en complément à deux sous forme de chaîne de caractères binaire de longueur `bits`.

Exemples d'appels attendus :

```
print(complement_a_deux(19))    # affiche 00010011
print(complement_a_deux(-19))   # affiche 11101101
print(complement_a_deux(-1))    # affiche 11111111
print(complement_a_deux(0))     # affiche 00000000
```