REPRÉSENTATION DES ENTIERS RELATIFS

- ▷ Histoire de l'informatique
- ▷ Représentation des données
- ▷ Traitement des données
- ▷ Interactions entre l'homme et la machine sur le Web
- ▷ Architectures matérielles et systèmes d'exploitation
- ▷ Langages et programmation
- ▷ Algorithmique

1. Une fausse bonne idée

Partant de la représentation des entiers naturels, il est assez naturel d'envisager de coder un entier relatif en prenant l'entier naturel associé sa valeur absolue et en lui ajoutant un *bit de signe* avec la convention :

- ▷ si l'entier est positif ou nul, le bit de signe est `0` ;
- ▷ si l'entier est négatif, le bit de signe est `1`.

Une convention supplémentaire stipulerai que ce bit de signe est le bit de plus fort poids [1] dans le codage de l'entier. Ainsi, un entier relatif serait codé par un bit de signe suivi du codage de sa valeur absolue. Par exemple, sur 4 bits, -3 serait codé sous la forme `1011` : un bit de signe égal à `1` suivi du codage binaire `011` de 3. Notons sur N bits, il serait ainsi possible de coder pratiquement tous les entiers compris entre -2^{N-1} et $2^{N-1} - 1$.

Mais si ce codage est simple à mettre en œuvre, il présente un premier inconvénient : le zéro admet deux codages : `00...0` et `10...0`. Ce qui est clairement regrettable.
Un deuxième inconvénient, en relation avec les circuits électroniques réalisant les opérations arithmétiques, rend même cette solution inenvisageable. En effet, un *circuit additionneur* permet l'addition de deux entiers naturels codés sur un nombre de bits donné. En pratique, le circuit ne *voit* que des codes binaires. Aussi, additionner sur 4 bits les codes `0101` et `1011` renvoie le code `1110`. Avec les correspondances :

$$\begin{array}{lcr} \texttt{0101} & \longrightarrow & 5 \\ \texttt{1011} & \longrightarrow & -3 \\ \texttt{0000} & \longrightarrow & 0 \end{array}$$

le résultat obtenu n'est pas celui escompté.

Une solution à ce problème existe mais elle a un coût : construire un circuit spécifique pour effectuer les soustractions ! Une autre solution ne nécessite que l'utilisation d'un additionneur pour réaliser à la fois les additions et les soustractions !

1. Bit situé le plus à gauche.

2. Addition modulaire

Sur N bits, les entiers naturels qu'il est possible de coder sont les entiers compris entre 0 et $2^N - 1$ inclus. Soit x et y deux entiers compris entre 1 et $2^N - 1$. L'addition de leurs codes binaires sur N bits renvoie un entier compris entre 0 et $2^N - 1$.

- ▷ Si $x + y < 2^N$, le code binaire renvoyé est celui de $x + y$.
- ▷ Si $x + y \geqslant 2^N$, le code binaire renvoyé est celui de $x + y$ modulo 2^N.

Considérons à présent un entier x compris entre 1 et $2^N - 1$ et choisissons $y = 2^N - x$, entier encore compris entre 1 et $2^N - 1$. La somme $x + y$ est égale à 2^N. L'addition des codes binaires renvoie donc 2^N modulo 2^N, à savoir 0. On peut donc écrire :

$$x + y \equiv 0 \mod 2^N$$

Autrement dit, y correspond à l'opposé de x dans cette opération modulaire. Ce qui permet de construire des nombres négatifs pour lesquels l'addition des codes binaires renvoie un résultat conforme à l'arithmétique sur les décimaux. C'est le choix adopté pour coder les entiers relatifs sur N bits.

- ▷ Si x est compris entre 0 et $2^{N-1} - 1$, le codage binaire de x est celui de l'entier naturel x sur N bits.
- ▷ Si x est compris entre -2^{N-1} et -1, le codage binaire de x est celui de l'entier naturel $2^N - |x|$ sur N bits.

Ce qui mène au tableau de correspondances suivant.

$$\begin{array}{rcl} -2^{N-1} & \longleftrightarrow & \texttt{10...00} \\ -2^{N-1}+1 & \longleftrightarrow & \texttt{10...01} \\ \dots & \dots & \dots \\ -1 & \longleftrightarrow & \texttt{11...11} \\ 0 & \longleftrightarrow & \texttt{00...00} \\ 1 & \longleftrightarrow & \texttt{00...01} \\ \dots & \dots & \dots \\ 2^{N-1}-1 & \longleftrightarrow & \texttt{01...11} \end{array}$$

Dans ce contexte, le zéro admet un seul codage. Par ailleurs, le bit de plus fort poids est `0` pour le entiers positifs, `1` pour les entiers négatifs. On retrouve ainsi l'idée du *bit de signe* introduit dans le paragraphe précédent.
Ces résultats peuvent être présentés de manière visuelle sur un cercle comme sur la figure 1 qui illustre un codage sur $N = 4$ bits.

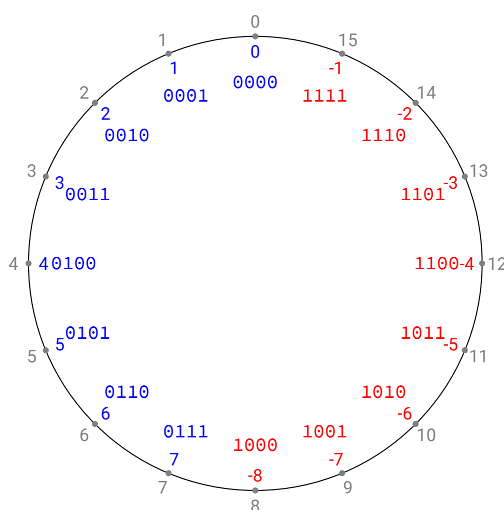

FIGURE 1

Illustrons notre propos avec l'exemple du premier paragraphe des codes binaires `0101` et `1011` de deux entiers relatifs dont la somme a pour code binaire `0000`.

▷ `0101` a un bit de signe égal à `0`; c'est le code d'un entier positif, à savoir 5.

$$\texttt{0101}_2 = 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5$$

▷ `1011` a un bit de signe égal à `1`; c'est le code d'un entier négatif, à savoir l'entier x tel que $2^4 - |x| = 11$. Or :

$$\texttt{1011}_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 11$$

D'où $|x| = 16 - 11$ puis $x = -5$.

La somme de ces entiers est bien nulle !

retenue	1	1	1	0		
		0	1	0	1	⟵ code de 5
+		1	0	1	1	⟵ code de -5
	1	0	0	0	0	⟵ code de 0

3. Complément à deux

L'algorithme précédent permet effectivement de coder sur un nombre de bits fixé des entiers naturels mais il n'est pas commode à utiliser. On lui préfère la technique du *complément à deux* (CP2). Avant de la détailler, observons les codages suivants sur $N = 4$ bits et, plus particulièrement, le passage du codage d'un entier positif à celui d'un entier négatif.

1	⟶	0001	1111	⟵	-1
3	⟶	0010	1110	⟵	-2
4	⟶	0011	1101	⟵	-3
		⋮	⋮		
7	⟶	0111	1001	⟵	-7
		↓ inversion `0` ↔ `1`	⟶ ↑ ajouter 1		

Ainsi, la technique du *complément à deux* s'applique en deux temps.

▷ Inversion des bits `0` et `1` dans le code de l'entier positif (c'est la phase dite de *complément à un*).

▷ Addition de `1` au code obtenu.

Appliquons cette technique pour coder l'entier -5 vu dans le paragraphe précédent, dont le codage binaire sur 4 bits est `1011`.

▷ On code d'abord l'entier positif 5 sur 4 bits : `0101`.

▷ On procède ensuite aux inversions (complément à un) : `1010`.

▷ On ajoute `1` : `1011`.

Appliquons cette technique au code obtenu `1011`.

▷ Inversions (complément à un) : `0100`.

▷ Ajout de `1` : `0101`.

On retrouve le code de 5. On dit que l'opération de *complément à deux* est involutive : pour tout entier n :

$$n \xrightarrow{CP2} (-n) \xrightarrow{CP2} n$$

Enfin, ajoutons que le complément à deux du code de 0 est le même code. Le code suivant illustre les opérations de complément à 1 puis de complément à 2.

```
def cp1(code):
    '''
    Renvoie le complément à 1 d'un code binaire
    Entrée : chaîne de caractères d'un code binaire
```

```python
    Sortie : chaîne de caractères de même longueur du complément à 1
    '''
    code_cp1 = ""
    for c in code:
        if c == '0':
            code_cp1 += '1'
        else:
            code_cp1 += '0'
    return code_cp1

def cp2(code):
    '''
    Renvoie le complément à 2 d'un code binaire
    Entrée : chaîne de caractères d'un code binaire
    Sortie : chaîne de caractères de même longueur du complément à 2
    '''
    taille = len(code)
    code_cp1 = cp1(code)
    retenue = 1
    code_cp2 = ""
    j = taille
    while j > 0:
        j -= 1
        if retenue == 1:
            if code_cp1[j] == '0':
                code_cp2 = '1' + code_cp2
                retenue = 0
            else:
                code_cp2 = '0' + code_cp2
        else:
            code_cp2 = code_cp1[j] + code_cp2
    return code_cp2
```

Pour conclure cette partie, donnons une justification rapide de la technique du complément à deux. Soit x un entier positif compris entre 0 et $2^{N-1} - 1$. Sa décomposition binaire peut s'écrire :

$$x = \sum_{i=0}^{N-1} b_i 2^i$$

avec $b_i \in \{0, 1\}$. Considérons à présent l'expression :

$$\begin{aligned} 2^N - 1 - x &= \sum_{i=0}^{N-1} 2^i - \sum_{i=0}^{N-1} b_i \times 2^i \\ &= \sum_{i=0}^{N-1} (1 - b_i) \times 2^i \end{aligned}$$

de sorte que :

$$2^N - x = 1 + \sum_{i=0}^{N-1} \underbrace{(1 - b_i)}_{\text{complément à 1}} \times 2^i$$

Ainsi, en posant :

$$CP2(x) = 2^N - x \xrightarrow{\text{donne la représentation binaire de}} (-x)$$

il vient :

$$CP2(CP2(x)) = 2^N - (2^N - x) \xrightarrow{\text{donne la représentation binaire de}} -(-x) = x$$