Représenter les entiers relatifs

Doc. 4,3

Objectif

Comprendre comment un ordinateur écrit un entier négatif avec seulement des 0 et des 1, et utiliser pour cela le complément à 2.

Q1. La mémoire d'un ordinateur ne contient que des 0 et des 1 : il n'y a pas de case pour le signe « − ». Proposer une idée pour écrire -5 sur 4 bits.

Réponse : L'idée la plus fréquente consiste à réserver un bit pour le signe : par exemple 0 pour « positif » et 1 pour « négatif », suivi de l'écriture binaire de 5. Sur 4 bits, cela donne 1101 : un bit de signe, puis 101. C'est l'idée étudiée dans la première partie. Toute autre proposition cohérente est acceptable à ce stade.

§1. Une fausse bonne idée : le bit de signe

Essayons l'idée la plus naturelle. Sur 4 bits, on réserve le bit de gauche au signe (0 pour un nombre positif, 1 pour un nombre négatif) et les 3 autres bits à la valeur absolue. Ainsi, $+3$ s'écrit 0011 et -3 s'écrit 1011.

Q2. Avec cette convention sur 4 bits :

1. Écrire $+5$ et -5.
2. Écrire $+0$ et -0. Que remarque-t-on ?
3. Poser l'addition binaire des écritures de $+5$ et de -3, en ne gardant que 4 bits. Quel nombre obtient-on ? Quel nombre devrait-on obtenir ?

Réponse :

1. $+5$ s'écrit 0101 et -5 s'écrit 1101.
2. $+0$ s'écrit 0000 et -0 s'écrit 1000 : le nombre zéro a deux écritures différentes. Il faudrait que la machine sache que ces deux codes représentent le même nombre, par exemple pour tester une égalité.
3. $0101 + 1011 = 10000$; en gardant 4 bits, il reste 0000, c'est-à-dire 0. Or $5 + (-3) = 2$, qui devrait s'écrire 0010. L'addition binaire que l'on sait poser ne fonctionne plus.

Cette convention aurait donc un coût : il faudrait un circuit électronique pour additionner les nombres positifs et un autre, plus compliqué, pour les nombres de signes différents. On cherche une écriture des entiers négatifs avec laquelle l'addition binaire ordinaire donne directement le bon résultat.

§2. L'idée du compteur kilométrique

Un vieux compteur kilométrique de voiture n'affiche que 4 chiffres. Lorsqu'il indique 9999 et que la voiture parcourt encore un kilomètre, il affiche 0000 : le chiffre 1 de 10000 n'a pas de place pour s'afficher. C'est le dépassement de capacité du Doc. 4,2.

Q3. Sur ce compteur à 4 chiffres, quel nombre faut-il ajouter à 0001 pour revenir à 0000 ? Quel nombre « joue le rôle » de -1 ? Et de -2 ?

Réponse : $0001 + 9999 = 10000$, qui s'affiche 0000. Sur ce compteur, ajouter 9999 revient à retirer 1 : 9999 joue le rôle de -1. De même, 9998 joue le rôle de -2, puisque $0002 + 9998$ s'affiche 0000.

Une case de 4 bits se comporte exactement comme ce compteur, mais en base 2. Après 1111 vient 0000, car $(1111)_2 + 1 = (10000)_2$ et le cinquième bit est perdu. On peut donc décider que l'écriture de -1 est celle qui, ajoutée à 0001, donne 0000.

Q4.
1. Quelle écriture sur 4 bits, ajoutée à 0001, donne 0000 (en perdant la retenue finale) ? Quel entier positif représente-t-elle habituellement ?
2. Même question pour l'écriture de -2, qui ajoutée à 0010 doit donner 0000.
3. En observant ces deux exemples, quel entier positif p faut-il écrire pour représenter un entier négatif $-x$ sur 4 bits ? Compléter le tableau.

entier	-1	-2	-3	-4	-5	-6	-7	-8
entier positif p								
écriture sur 4 bits								

Coup de pouce

Sur le compteur kilométrique, -1 était représenté par $10000 - 1$. Sur 4 bits, le rôle de 10000 est joué par $(10000)_2 = 2^4 = 16$.

Réponse :
1. 1111, car $0001 + 1111 = 10000$, qui donne 0000 sur 4 bits. Habituellement, 1111 représente 15.
2. 1110, car $0010 + 1110 = 10000$. Habituellement, 1110 représente 14.
3. On écrit $p = 16 - x$, c'est-à-dire $2^4 - x$:

entier	-1	-2	-3	-4	-5	-6	-7	-8
entier positif p	15	14	13	12	11	10	9	8
écriture sur 4 bits	1111	1110	1101	1100	1011	1010	1001	1000

Q5. Avec cette nouvelle convention, poser l'addition des écritures de 5 et de -3 sur 4 bits, en perdant la retenue finale. Le résultat est-il juste ?

Réponse : 5 s'écrit 0101 et -3 s'écrit 1101. $0101 + 1101 = 10010$, et il reste 0010 sur 4 bits, c'est-à-dire 2. Le résultat est juste : l'addition binaire ordinaire fonctionne, sans aucun circuit supplémentaire.

On écrit donc les entiers positifs de 0 à 7 comme d'habitude, de 0000 à 0111, et les entiers négatifs de -8 à -1 par les écritures de 1000 à 1111. On remarque que le bit de gauche, appelé **bit de poids fort**, vaut 0 pour les entiers positifs ou nuls et 1 pour les entiers négatifs. On retrouve l'idée du bit de signe, mais sans ses défauts : zéro n'a qu'une seule écriture, 0000.

À retenir

Sur n bits, on peut écrire les entiers relatifs compris entre -2^{n-1} et $2^{n-1} - 1$:

- un entier x positif ou nul ($0 \leq x \leq 2^{n-1} - 1$) s'écrit comme d'habitude ;
- un entier $-x$ négatif ($1 \leq x \leq 2^{n-1}$) s'écrit comme l'entier positif $2^n - x$.

Le bit de poids fort vaut 1 pour les entiers négatifs et 0 pour les autres. Sur 8 bits, on écrit les entiers de -128 à 127.

§3. Le complément à 2 : une méthode rapide

Calculer $2^n - x$ puis le convertir en binaire est un peu long. On observe sur le tableau de la partie précédente une méthode plus rapide, que l'on appelle le **complément à 2**.

Exemple résolu : écrire –5 sur 8 bits

1. On écrit la valeur absolue 5 sur 8 bits : 00000101.
2. On inverse tous les bits (les 0 deviennent des 1 et les 1 des 0). On obtient le **complément à 1** : 11111010.
3. On ajoute 1 : $11111010 + 1 = 11111011$.

On vérifie avec la règle de la partie précédente : $2^8 - 5 = 251$, et $251 = 128 + 64 + 32 + 16 + 8 + 2 + 1 = (11111011)_2$.

Q6. Écrire sur 8 bits, à l'aide du complément à 2 : -1, -56 et -128. Vérifier chaque résultat en calculant $2^8 - x$.

Réponse :
- -1 : 00000001, inversé 11111110, plus 1 : 11111111. Vérification : $256 - 1 = 255 = (11111111)_2$.
- -56 : $56 = 32 + 16 + 8$ s'écrit 00111000, inversé 11000111, plus 1 : 11001000. Vérification : $256 - 56 = 200 = 128 + 64 + 8$.
- -128 : 128 s'écrit 10000000, inversé 01111111, plus 1 : 10000000. Vérification : $256 - 128 = 128$. Le complément à 2 de 10000000 est lui-même : -128 n'a pas d'opposé sur 8 bits, puisque $+128$ ne s'y écrit pas.

Pour lire un octet qui commence par 1, on applique la même méthode : le complément à 2 d'un entier négatif donne l'écriture de sa valeur absolue.

Q7. Quels entiers relatifs les octets suivants représentent-ils ? 01111111, 11110000 et 10000001.

Réponse :
- 01111111 commence par 0 : c'est un entier positif, $64 + 32 + 16 + 8 + 4 + 2 + 1 = 127$.
- 11110000 commence par 1 : il est négatif. Inversé : 00001111, plus 1 : 00010000 = 16. Il représente -16.
- 10000001 commence par 1. Inversé : 01111110, plus 1 : 01111111 = 127. Il représente -127.

Q8. Un élève affirme que -5 s'écrit 10000101 sur 8 bits, « parce que le 1 de gauche indique le signe ». Expliquer son erreur en calculant la somme de son écriture et de celle de 5.

Réponse : Il utilise le bit de signe de la partie 1. Avec son écriture, $00000101 + 10000101 = 10001010$, qui n'est pas 00000000 : la somme de 5 et de son « opposé » ne donne pas 0. Avec le complément à 2, $00000101 + 11111011 = 100000000$, et il reste bien 00000000 sur 8 bits. Le bit de poids fort indique le signe, mais les autres bits ne sont pas ceux de la valeur absolue.

Q9. Écrire -5 sur 4 bits puis sur 8 bits. Pourquoi faut-il toujours préciser le nombre de bits avant de parler de complément à 2 ?

Réponse : Sur 4 bits, -5 s'écrit 1011 ; sur 8 bits, 11111011. Le même entier n'a pas la même écriture : on calcule $2^4 - 5 = 11$ dans un cas, $2^8 - 5 = 251$ dans l'autre. Inversement, l'écriture 1011 représente -5 sur 4 bits, mais 11 si on la lit comme l'octet 00001011. Une écriture binaire ne veut rien dire tant qu'on n'a pas précisé le nombre de bits et la convention utilisée.

§4. Programmer le complément à 2

Les fonctions suivantes, écrites dans le Doc. 4,1, sont à recopier en tête du programme :

```python
def decompose(n: int) -> str:
    if n == 0:
        return "0"
    bits = ""
    while n > 0:
        bits = str(n % 2) + bits
        n = n // 2
    return bits

def octet(n: int) -> str:
    bits = decompose(n)
    while len(bits) < 8:
        bits = "0" + bits
    return bits

def recompose(binaire: str) -> int:
    n = 0
    for chiffre in binaire:
        n = 2 * n + int(chiffre)
    return n
```

Q10. Écrire la fonction `complement_a_1`, qui inverse tous les bits d'une chaîne :

```python
def complement_a_1(bits: str) -> str:
    """
    Retourne la chaîne obtenue en remplaçant chaque 0 de bits
    par 1 et chaque 1 par 0.

    >>> complement_a_1("00000101")
    '11111010'
    >>> complement_a_1("1011")
    '0100'
    """
```

Réponse :

```python
def complement_a_1(bits: str) -> str:
    resultat = ""
    for bit in bits:
        if bit == "0":
            resultat = resultat + "1"
        else:
            resultat = resultat + "0"
    return resultat
```

On parcourt la chaîne bit par bit, et on construit le résultat dans le même ordre, en ajoutant à droite.

Ajouter 1 à une chaîne binaire demande de propager des retenues : on le programmera dans le Doc. 4,4. Pour l'instant, on utilise la règle de la partie 2 : sur 8 bits, l'entier négatif $-x$ s'écrit comme l'entier positif $2^8 - x$.

Q11. Écrire la fonction `relatif_vers_octet`, qui doit utiliser la fonction `octet` :

```python
def relatif_vers_octet(n: int) -> str:
    """
    Retourne l'écriture sur 8 bits de l'entier relatif n,
    en complément à 2.
    Précondition : -128 <= n <= 127.

    >>> relatif_vers_octet(5)
    '00000101'
    >>> relatif_vers_octet(-5)
    '11111011'
    >>> relatif_vers_octet(-128)
    '10000000'
    """
```

Réponse :

```python
def relatif_vers_octet(n: int) -> str:
    assert -128 <= n <= 127, "n doit être compris entre -128 et 127"
    if n >= 0:
        return octet(n)
    else:
        return octet(2 ** 8 + n)
```

Si `n` vaut -5, alors `2 ** 8 + n` vaut $256 - 5 = 251$: on écrit bien $2^8 - x$ avec $x = 5$. Erreur fréquente : écrire `2 ** 8 - n`, qui donne 261 car `n` est déjà négatif.

Q12. Écrire la fonction `octet_vers_relatif`, qui fait le travail inverse. Choisir la méthode : on peut utiliser `recompose` et la règle $2^8 - x$, ou `complement_a_1`.

```python
def octet_vers_relatif(bits: str) -> int:
```

```
    """
    Retourne l'entier relatif écrit sur 8 bits par bits,
    en complément à 2.

    >>> octet_vers_relatif("01111111")
    127
    >>> octet_vers_relatif("11110000")
    -16
    >>> octet_vers_relatif("10000000")
    -128
    """
```

Coup de pouce

Commencer par regarder le bit de poids fort `bits[0]`. S'il vaut `"1"`, l'octet lu comme un entier positif vaut $p = 2^8 - x$: comment retrouver $-x$ à partir de p ?

Réponse :

```python
def octet_vers_relatif(bits: str) -> int:
    p = recompose(bits)
    if bits[0] == "1":
        return p - 2 ** 8
    else:
        return p
```

Si $p = 2^8 - x$, alors $-x = p - 2^8$. Pour `"11110000"`, $p = 240$ et $240 - 256 = -16$. Avec le complément à 1, on écrirait `-(recompose(complement_a_1(bits)) + 1)` dans le cas négatif.

§5. Bilan

À l'issue de cette activité, je sais…

- expliquer pourquoi un simple bit de signe ne convient pas (deux zéros, addition fausse) ;
- écrire un entier négatif sur n bits comme l'entier positif $2^n - x$;
- utiliser le complément à 2 (inverser les bits, puis ajouter 1) pour écrire un entier négatif ou pour lire un octet qui commence par 1 ;
- donner la plage des entiers relatifs écrits sur n bits, de -2^{n-1} à $2^{n-1} - 1$, et en particulier de -128 à 127 sur 8 bits ;
- expliquer pourquoi il faut toujours préciser le nombre de bits ;
- programmer le complément à 1 et les conversions entre un entier relatif et son octet.

Dans le Doc. 4,4, on programme l'addition de deux octets. On pourra alors ajouter 1 à un complément à 1, et vérifier que l'addition fonctionne aussi pour les entiers négatifs.