PF1 – DM

14 novembre 2013

Ce DM n'est pas obligatoire mais vous pouvez me le rendre si vous souhaitez une correction. C'est uniquement pour ceux qui pensent n'avoir pas bien compris la représentation des nombres en machine et qui souhaite s'entraîner un peu.

1 Rappels de cours

1.1 Previously in PF1

Dans la partie précédente du cours, nous avons étudié des systèmes *d'écriture* des nombres entiers. Nous avons vu les systèmes additifs (égyptiens, chiffres romains) et les systèmes à positions (le système qu'on utilise actuellement en base 10). On a vu que le choix de la base 10 était avant tout culturel et que n'importe quelle base strictement plus grande que 1 conviendrait. Nous avons donc étudié comment passer d'une base à l'autre, comment calculer (additions, multiplications) dans une base différente etc.

Nous avons ensuite généralisé cela pour écrire les nombres à virgules. En base b, on s'autorise les fractions $1/b$, $1/b^2$, ..., $1/b^i$ ... Par exemple, $1/2 = (0.1)_2 = (0.5)_{10}$ mais $2/3 = (0.2)_3$ n'est pas représentable finiment en base 10.

Nous n'avons cependant pas parlé des nombres négatifs mais cela ne change pas grand chose dans l'écriture des nombres. Quelque soit la base, si on veut parler d'un nombre négatif, on ajoute le signe $-$ devant, comme on le faisait en base 10.

1.2 Représentation des entiers en machine

Tout ce qui précéde est un premier pas vers la représentation des nombres en machine mais ce n'est pas suffisant. En effet, comment indiquer à l'ordinateur que 1101 est un nombre codé sur 4 bits et non pas deux nombres 11 et 01 codés sur 2 bits chacun ? Comment calculer efficacement et rapidement avec des nombres entiers en machine ?

La solution adoptée a été de choisir une représentation des nombres sur un nombre fixé de bits. Bien sûr, on ne peut pas représenter *tous* les nombres mais sur 32 bits, on peut en représenter quelques milliards ce qui est suffisant pour la plupart des programmes qu'on utilise. Et si cela ne suffit pas, on utilisera 64 bits. Si cela ne suffit toujours pas, il faudra utiliser des librairies spécialisées dans le calcul exact (par exemple GMP) mais on perd énormément en efficacité. On dispose donc de plusieurs types d'entier en Java, chacun codant les entiers sur un nombre de bit différents. On choisira celui qui convient le mieux selon l'application (pas besoin de `long` (64 bits) si vos entiers représentent des notes entre 0 et 20). On veut que ces types représentent aussi des nombres négatifs. Pour cela, on propose un *codage* des nombres, qu'on appelle le **complément à deux sur n bits** : ça ressemble à du binaire mais ce n'en est pas tout à fait.

Avec le complément à deux sur n bits, on peut représenter tous les nombres de -2^{n-1} à $2^{n-1} - 1$ avec n bits :

- Si $0 \leq k < 2^{n-1}$ et que $k = (a_m \dots a_0)_2$ en binaire, alors son complément à deux sur n bits est $0 \dots 0 a_m \dots a_0$ (on rajoute des 0 pour écrire k sur n bits).
- Si $-2^{n-1} \leq k < 0$, alors le complément à deux de k est l'écriture binaire du nombre $2^n - k$

Moralité : le complément à deux d'un nombre positif est son écriture binaire (avec éventuellement des 0 devant).

Étant donné un nombre écrit en complément à deux sur n bits, on peut savoir facilement son signe : si le bit de gauche est 1 alors il est négatif, sinon il est positif.

Pour calculer le complément à deux d'un nombre négatif $-m$, on peut soit utiliser la définition ci-dessus, soit effectuer les opérations suivantes :

- écrire m en binaire sur n bits (c'est important de rajouter les 0 devant)
- inverser chaque bit
- ajouter 1

Étant donné les n bits d'un complément à deux, pour retrouver le nombre qu'il code on peut faire :

- si le bit de gauche est 0, c'est un nombre positif. Donc on convertit le code comme si c'était du binaire
- si le bit de gauche est 1, c'est un nombre négatif. On commence par calculer son opposé, donc on aura le code binaire. Pour calculer l'opposé d'un nombre négatif, il suffit de faire les opérations inverses :
  - Enlever 1
  - Inverser les bits

2 Quelques exercices

Pour les exercices suivants, n'hésitez pas à vérifier vos calculs à la calculette, mais essayez toujours de les faire sans.

Exercice 1. Lire la correction du contrôle numéro 2 disponible à l'adresse `http://www.math.jussieu.fr/~capelli/enseignements/cc/cc2corr.pdf` et essayer de comprendre ses erreurs.

Exercice 2. [D'un nombre à son complément à deux] Donnez le code en complément à deux sur 8 bits des nombres suivants :

1. $(10011)_2$
2. $-(5C)_{16}$
3. 42
4. $-(45)_7$

Exercice 3. [Du complément à deux vers la base 10] Dans cet exercice, les nombres sont représentés en complément à deux. Écrivez-les en binaire et en décimal :

1. 1111
2. 0111
3. 10011110
4. 01100011

Exercice 4. [overflow]

1. Écrivez le complément à deux sur 8 bits de 120
2. Écrivez le complément à deux sur 8 bits de 53
3. Donnez le complément à deux sur 8 bits de l'addition de ces deux représentations (ce que ferait l'ordinateur si vous additionniez un `byte` valant 120 et un `byte` valant 53). Quel nombre est représenté par ce complément à deux sur 8 bits ? Pourquoi n'est-ce pas 173 ?

Exercice 5. [Conversion vers les flottants]

1. Que valent, en binaire puis en décimal, les flottants suivants :
   (a) 1 10000001 10101000000000000000000000000000
   (b) 0 01110011 10110000000000000000000000000000
2. Donnez la représentation en flottant des nombres suivants (vous avez le droit de mettre trois petits points pour indiquer les derniers 0) :
   (a) $(111.10101)_2$
   (b) -4.75