Rapport

Titre	Laboratoire 1 : nombre réel sur 100 bits
Professeur	Stéphane Gobron
Étudiants	Diego Antognini, Alexandre Perez, Sébastien Vaucher
Date	11 mars 2013

Représentation

Selon la consigne, nous avons représenté le signe du nombre sous 1 bit, l'exposant sous 13 bits puis la mantisse sur 86 bits réels (mais en réalité 87). Pour ce faire, nous avons utilisé des tableaux de booléens pour représenter les nombres binaires. Le bit le moins significatif se situe à l'indice de 0 du tableau. Pour le *d*, on le trouve avec la formule suivante : $2^{e-1}-1$. Dans certains cas (quand il s'agit d'un nombre dénormalisé[1]), il faut encore enlever 1 à ce nombre. Selon le cours, cette étape supplémentaire est effectuée partout, nous avons donc suivi cette convention.

La classe que nous avons réalisée pour cette représentation reçoit une chaîne de caractères en entrée. Nous regardons d'abord le signe et définissons le bit de signe. Ensuite, nous séparons la partie entière de la partie décimale (s'il y en a), puis nous effectuons les conversions en binaire à l'aide de tableau de *int*, donc chaque case contient un des chiffres de ce nombre. Nous utilisons cette technique car si nous utilisions un type existant, nous serions limités au nombre maximal qu'il pourrait gérer. Une fois les conversions effectuées, nous regardons de combien de case nous devons déplacer la virgule afin d'avoir un bit à 1 à sa gauche. Nous utilisons ce nombre (k) afin de générer la valeur de l'exposant, avec $e = k + d + 1$, le $+1$ vient du fait que nous faisons -2 pour tous les cas.

Nous remplissons ensuite la mantisse avec tous les bits se trouvant après la nouvelle position de la virgule.

Gestion des différentes exceptions

Selon la norme *IEEE 754-1985*, certains nombres doivent avoir une représentation spéciale :

- N'est pas un nombre (*NaN, Not a Number*) si $e = 2^{13}-1$ et $m \neq 0$, indépendamment du signe ;
- $\pm\infty$ si $e = 2^{13}-1$ et $m = 0$, le signe dépend de s ;
- ± 0, que nous simplifions par 0, si $e = 0$ et $m = 0$.

Les cas sont aussi gérés lors de l'utilisation des diverses opérations.

Addition ($+(a+b), -1\cdot(a+b)$)

L'addition de deux nombres de même signe se déroule en trois étapes : alignement des mantisses, addition des mantisses et enfin normalisation. Si les nombres sont de signes différents, on utilise la soustraction.

La première étape est l'alignement. Elle consiste à faire en sorte que les deux nombres aient le même exposant. Pour ce faire, on décale le nombre ayant le plus petit exposant vers la droite jusqu'à ce que son exposant soit le même que le nombre *a*. Il ne faut pas oublier de prendre en compte le bit caché lors du décalage.

La deuxième étape doit additionner les deux mantisses de manière classique, ici encore en prenant les bits cachés en compte.

[1] Un nombre dénormalisé est un nombre dont le bit caché vaut 0 car il est impossible d'exprimer ce nombre avec une mantisse normalisée.

Pour finir, on décale la mantisse résultante vers la gauche autant de fois que nécessaire afin d'avoir un bit à 1 dans l'emplacement du bit caché. Le bit caché étant implicite, il ne sera pas codé dans le résultat final.

Soustraction ($+a-b, -a+b$)

Pour commencer, on s'arrange pour avoir le cas de $+a-b$, où a représente le nombre le plus grand et changer les signes si nécessaire (avec $-a+b=-1\cdot(a-b)$). Ensuite, nous cherchons la différence des exposants (k_1), puis effectuons un décalage à droite de la mantisse b. Nous faisons ensuite la soustraction entre la mantisse de *a* et celle de *b*. Nous comptons le décalage nécessaire (k_2) à partir du bit le plus significatif pour avoir le premier bit à 1 (dans notre cas toujours 1) puis calculons l'exposant final avec $E=E_a+d-k_2$. Pour terminer, nous remplissons l'exposant et la mantisse avec les résultats précédents puis définissons le bit de signe en conséquence.

Dans le cas où a et b possèdent le même signe, on utilise l'addition.

Multiplication ($\pm a \cdot \pm b$)

Tout d'abord, on additionne les deux exposants et on soustrait le complément à deux de notre *d.* Puis on multiplie les deux mantisses avec leurs bits cachés. Finalement on normalise en décalant la mantisse et en réduisant notre exposant. Pour le signe du résultat, il nous suffit de faire un ou exclusif sur les signes des deux nombres.

Division ($\pm a / \pm b$)

Pour la division, on fait *a*(1/b)*. Pour ce faire on commence par initialiser un nombre à 1. Pour l'exposant on soustrait les deux exposants (de 1 et de b), et on ajoute le complément à deux de notre *d.* Puis on divise les deux mantisses avec leurs bits cachés. Ensuite, on normalise le résultat en décalant la mantisse. Finalement, on appelle la fonction de multiplication avec *a* et notre *1/b*. Pour le signe du résultat, il nous suffit de faire un ou exclusif sur les signes des deux nombres.

Approximation de π

Puisque les 4 opérations mathématiques de bases fonctionnent, nous pouvons effectuer l'approximation de π, mais avant il faut créer la fonction de puissance. La puissance est une suite de multiplications consécutives (excepté dans le cas où $n \leq 0$, néanmoins nous ne nous intéressons pas au cas négatif et ne gérons que le cas où *n* est positif).

Nous approximons π par la formule $\pi = \sum_{i=0}^{n}\left(\frac{4}{8i+1}-\frac{2}{8i+4}-\frac{1}{8i+5}-\frac{1}{8i+6}\right)\cdot\left(\frac{1}{16}\right)^i$.

Références

1. Représentation
    a. *IEEE Standard for Binary Floating-Point Arithmetic* 754-1985
    b. Wikipedia, IEEE 754-1985, http://en.wikipedia.org/wiki/IEEE_754-1985
2. Opérations binaires
    a. *Computer Organization & Design – The Hardware Software Interface*, 4ème édition, David A. Patterson et John L. Hennessy
    b. Cours *Aritmética en coma flotante* de Daniem Mozos, Marcos Sánchez-Elez, José Luis Risco, de l'université de Madrid, http://www.fdi.ucm.es/profesor/mozos/aec/aritm_pf.pdf
    c. Cours *Floating Point* ainsi que les sous-cours, de l'université du Maryland, http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Data/float.html
    d. University of Wisconsin-Madison - Chapter 7 - floating point arithmetic, http://pages.cs.wisc.edu/~smoler/x86text/lect.notes/arith.flpt.html