Michael F. Cowlishaw, Eric M. Schwarz, Ronald M. Smith, Charles F. Webb,

A decimal floating-point specification

Sylvain Collange

27 janvier 2006

L'article étudié [1] présente une proposition de norme décrivant des formats de représentation de nombres virgule flottante en base 10. Nous verrons tout d'abord les raisons qui amènent à vouloir utiliser un système flottant décimal, et quelles exigences un tel système devra respecter, puis étudierons la représentation des flottants proposée, et enfin tenterons d'aborder les choix possibles pour une implantation matérielle utilisant ce format.

1 Objectifs

1.1 Pourquoi le système décimal ?

Historiquement, la base 2 s'est imposée pour la grande majorité des formats virgule flottante. Elle a pour avantage la simplicité et l'efficacité d'implémentation. Cependant, elle n'est pas adaptée au domaine financier, où les données utilisent le système décimal, et où les pertes de précision ne peuvent pas être tolérées. De plus, les données numériques présentes dans les bases de données existantes ne sont que très rarement codées dans un format virgule flottante binaire, nécessitant des conversions lors des accès, ce qui peut être très pénalisant lorsque les calculs à effectuer sont simples.

1.2 Spécificités du système décimal

Normalisation

Les nombres flottants binaires de la norme IEEE 754 doivent être représentés si possible sous forme normalisée, c'est-à-dire avec une mantisse de la forme $1,f$. Cela offre l'avantage de pouvoir comparer facilement deux flottants, et de ne pas avoir à stocker le 1 initial dans la représentation mémoire.

Dans le système décimal, le 1 implicite dans la représentation n'est pas possible, obligeant à stocker tous les chiffres de la mantisse. De plus, ne pas normaliser les nombres offre la possibilité de retenir le nombre de chiffres significatifs des nombres manipulés, en alignant toujours le dernier chiffre significatif avec le chiffre de poids faible de la mantisse.

Ainsi, dans un système à cinq chiffres, 15.0 sera représenté comme 0.0150×10^4.

L'arithmétique non-normalisée est cependant différente de l'arithmétique d'ordre de grandeur utilisée dans les calculs effectués à la main et

mettant en jeu des mesures, notamment en physique et en chimie. Dans le système d'ordre de grandeur, on écrira par exemple 1.20 pour une mesure dont l'intervalle de confiance est de 0.01. Le résultat de l'opération $1.20+0.003$ sera 1.20 : le nombre 0.003 étant inférieur à l'erreur de mesure, il est négligé.

À l'inverse, dans le système de l'arithmétique non-normalisée, il n'est pas intéressant de limiter la précision de manière arbitraire, conduisant par exemple à renvoyer 1.0 plutôt que 1.1 pour le calcul de $((1.0+0.04)+0.04)+0.02$. Les opérations se font donc de manière exacte lorsque c'est possible, l'arrondi n'intervenant que lorsque la mantisse exacte est trop grande pour être représentée [5]. De ce fait, on a l'assurance que si un flottant n'est pas normalisé (et que son exposant n'est pas l'exposant minimal), il représente un nombre exact.

Un avantage important de cette représentation non-normalisée est que, lorsque les exposants sont identiques, les opérations $+, -$ et $\times$ se comportent comme des opérations en virgule fixe, aux dépassements de capacité près. On peut donc utiliser le calcul flottant décimal pour émuler le calcul en virgule fixe.

1.3 Exigences logicielles

L'arithmétique décimale a été et reste très présente dans les langages de programmations tournés vers la gestion.

Les types de données présents dans ces langages sont généralement basés soit sur des formats virgule fixe à 31 ou 32 chiffres, soit sur des formats virgule flottante en précision arbitraire.

Les auteurs de l'article se proposent donc de définir un format de virgule flottante utilisant la base 10, pouvant être utilisé par les langages et bibliothèques existants.

2 Représentation

2.1 Mantisse

Trois possibilités sont proposées pour le stockage de la mantisse.

- **Le système binaire** est déjà utilisé par des formats existants basés sur une implantation logicielle [2]. Il offre l'avantage d'une densité de stockage optimale et se prête bien aux multiplications. Cependant, les divisions par des puissances de 10 nécessaires à l'alignement des mantisses lors des additions et aux normalisations ont un coût en matériel rédhibitoire. Ce système n'est donc pas envisageable pour une implantation matérielle.
- **Le décimal codé binaire (BCD)** représente chaque chiffre décimal par quatre bits. Il permet une implantation relativement efficace tant en logiciel qu'en matériel. Il présente cependant un surcoût en mémoire de 20,4% par rapport au stockage optimal.
- **Le codage de Chen-Ho** est une manière plus compacte que le BCD pour représenter les décimaux, en stockant trois chiffres décimaux sur 10 bits (0,34% de plus que l'optimal). La conversion entre un code de Chen-Ho et la représentation BCD correspondante peut se réaliser par 2 à 3 étages de portes logiques en matériel, ou par une recherche dans une table en logiciel. Le surcoût est donc faible, voire négligeable.

En plus des trois possibilités évoquées dans l'article étudié, d'autres représentations permettent une implantation plus efficace pour du matériel moderne. Il s'agit de systèmes redondants qui permettent d'éviter les propagations de retenues lors des additions, de manière analogue à la représentation *carry-save* en binaire [3].

Pour les multiplications, une représentation dite d'Avizienis utilisant des chiffres signés, équilibrés autour de 0 (par exemple dans l'intervalle $[-5; +5]$) permet également de simplifier la génération des produits partiels [4].

La principale contrainte que cela impose est que les comparaisons, devenues coûteuses, soient effectuées par un circuit spécifique. Cependant, ce problème se pose également dés lors qu'on utilise une arithmétique non normalisée et un codage de type Chen-Ho pour la mantisse. Une autre facteur à considérer est la complexité supplémentaire induite par ce système sur les circuits d'arrondis, du fait qu'ils doivent effectuer des comparaisons.

Bien que redondant, un système d'Avizienis en base 10 ne nécessite pas forcément un espace de stockage excessif : en effet, un système où les chiffres peuvent prendre 12 valeurs peut se coder avec 4 bits par chiffre de manière analogue au BCD, voire avec 11 bits par groupe de 3 chiffres si nécessaire.

Même si ce système n'apporte aucun avantage à une implantation logicielle, il n'ajoute pas non plus de complexité particulière : les chiffres signés peuvent se manipuler de la même façon que les chiffres BCD.

Un réel inconvénient d'un tel système est qu'il est assez éloigné du calcul tel qu'effectué à la main, et est donc d'utilisation peu intuitive : par exemple, un nombre de quatre chiffres, chacun dans $[-6; +5]$, sera compris entre -6666 et 5555.

En tout état de cause, la question des systèmes redondants n'est pas abordée dans l'article, et c'est la représentation de Chen-Ho qui a été retenue.

2.2 Exposant

Le choix du format de l'exposant est moins critique, étant donné qu'il reste relativement court et que seules des additions, soustractions et comparaisons lui sont appliquées.

Une base décimale ne présente pas d'intéret particulier, sinon de détecter plus simplement les dépassements de capacité si l'on impose un intervalle du type $[-999; +999]$ à l'exposant.

Un système complément à deux ou un système biaisé sont également envisageables. En effet, un exposant biaisé n'offre pas comme en base deux la possibilité de comparer deux flottants sur la base de leur représentation binaire, du fait des choix du codage de la mantisse et de la non-normalisation. Cependant, c'est ce format qui a tout de même été retenu, en raison de sa similarité avec le codage de l'exposant des flottants IEEE-754 en base 2.

Un choix qui a été fait dans l'article est de limiter l'intervalle des exposants possibles de façon à restreindre l'ensemble des nombres (strictement positifs) représentables à un intervalle de la forme $[1.00\ldots0 \times 10^{-999}; 9.99\ldots9 \times 10^{+999}]$. Cela permet de simplifier les procédures de formatage pour l'affichage des nombres et la validation des nombres entrés par l'utilisateur, en plus de donner à l'utilisateur l'impression d'un système entièrement décimal.

Cependant, ce choix a été remis en cause dans une version ultérieure de la proposition de norme, probablement pour permettre un plus grand intervalle des exposants possibles.

2.3 Valeurs spéciales

De la même façon que dans pour les flottants IEEE-754 en base 2, les infinis et les NaN utilisent des valeurs réservées de l'exposant.

La principale différence est que tout les chiffres de la mantisse étant conservés, le zéro ne constitue pas une valeur spéciale mais est simplement représenté par une mantisse nulle. Cela a pour conséquence importante que plusieurs représentations de zéro sont possibles, avec différents exposants. Allié à de l'arithmétique flottante sans normalisation, cela permet de garder la trace de l'ordre de grandeur d'un zéro, s'il apparaît à la suite d'une soustraction.

2.4 Formats proposés

Pour permettre de répondre aux exigences logicielles, il doit être possible de manipuler des nombres d'au moins 31 chiffres.

Deux formats sont donc proposés (fig. 1). L'ordre adopté pour le stockage des éléments est le même que celui des flottants binaire IEEE-754, à savoir un bit de signe suivi de l'exposant puis de la mantisse.

FIG. 1 – Taille, nombre de chiffres décimaux, taille et intervalle des exposants dans la proposition initiale

Taille	Chiffres	Bits d'exposant	e_{min}	e_{max}
64 bits	15	11	-985	$+999$
128 bits	33	15	-9967	$+9999$

Dans la version actuelle de la spécification, un format sur 32 bits a été ajouté et des modifications ont été opérés sur les deux autres formats (fig. 2).

L'intervalle des exposants de l'ancienne version était choisi de manière à ce que l'ensemble des nombres représentables corresponde exactement à celui des nombres pouvant s'écrire dans un format fixé (tous les nombres s'écrivant sous la forme $(-1)^s \times d_0.d_1 d_2 \dots d_{p-1} \times 10^{e_0 e_1 \dots e_{q-1}}$, avec $d_i, e_i \in [0,9]$). Dans la nouvelle révision, l'intervalle possible des exposants a été ajusté de façon à équilibrer l'intervalle par rapport à la fonction inverse, de manière analogue aux flottants IEEE-754.

3 Implantation matérielle

Plusieurs circuits d'addition et de multiplication d'entiers décimaux ont été proposés [3][4], mais il ne semble pas exister de publication concernant des circuits d'arithmétique flottante décimale sur du matériel moderne.

FIG. 2 – Taille, nombre de chiffres décimaux, taille et intervalle des exposants dans la spécification actuelle

Taille	Chiffres	Bits d'exposant	e_{min}	e_{max}
32 bits	7	8	-95	$+96$
64 bits	16	10	-383	$+384$
128 bits	34	14	-6143	$+6144$

Cependant, des circuits de calcul flottant décimal ne devraient que peu différer de leurs équivalents binaires, en dehors des additions et multiplications de mantisses.

Je me suis donc intéressé à la réalisation d'un additionneur flottant décimal simple, et à la question de savoir quelle représentation interne des nombres utiliser.

L'organisation générale du circuit est globalement identique à celle d'un additionneur flottant binaire, comprenant une étape de comparaison/échange, une étape de décalage et d'addition/soustraction des mantisses puis une étape d'arrondi et normalisation. (fig. 3).

Dans un premier temps, j'ai écrit un additionneur BCD naïf avec propagation de retenues. Sa latence linéaire en la taille des entrées rend cependant son utilisation inenvisageable pour additionner des mantisses de 33 bits (fig. 4).

L'additionneur suivant est basé sur un additionneur de Svoboda. Il s'agit d'un circuit sans propagation de retenues utilisant un système d'Avizienis dont les chiffres sont dans l'intervalle $[-6,6]$ [4]. Bien que nettement plus volumineux que le circuit naïf, il offre une latence indépendante de la taille des entrées (fig. 4).

Cependant, l'intégration d'un additionneur de ce type dans un additionneur virgule flottante soulève des problèmes supplémentaires, étant donné que la comparaison de deux nombres devient une opération com-

plexe. Des comparaisons étant nécessaires pour effectuer l'arrondi, il n'est pas possible de conclure dans un sens ou dans l'autre sans avoir comparé également les circuits d'arrondi. Il est probable que ce système n'offre pas d'avantage significatif par rapport à un système basé sur un arbre de retenues pour un circuit d'addition flottante.

4 Conclusion

La spécification de virgule flottante décimale présentée dans l'article étudié a été proposée pour inclusion dans la révision de la norme IEEE-754. Cette proposition fait actuellement l'objet de discussions, et a été déjà légèrement modifiée, notamment en ce qui concerne les tailles respectives de la mantisse et de l'exposant. Ces modifications vont dans le sens d'un système plus rigoureux, au détriment de l'aspect intuitif du système décimal.

Plusieurs implantations logicielles de cette spécification sont d'ores et déjà disponibles, notamment une bibliothèque en C pour laquelle une inclusion dans le langage C a été proposée et qui est déjà intégrée dans le compilateur *GCC* [6]. D'autres bibliothèques compatibles existent pour Java[7], C++, Eiffel et Python.

Néanmoins, on peut noter que la grande majorité des études et réalisations concernant l'arithmétique décimale sont effectués par IBM, et que ce domaine ne semble pas soulever véritablement l'intérêt d'autres sociétés et organismes. De ce fait, il parait peu probable que le système décimal soit massivement utilisé en dehors des *mainframes* destinés au calcul financier dans un futur proche.

Cependant, une normalisation de l'arithmétique décimale ne peut qu'avoir un effet positif, en fournissant une base commune aux futures implantations logicielles et matérielles.

Références

[1] Michael F. Cowlishaw, Eric M. Schwarz, Ronald M. Smith et Charles F. Webb. A Decimal Floating-Point Specification. *15th IEEE Symposium on Computer Arithmetic*, 2001.

[2] ISO/IEC 23270 :2003 – C# Language Specification. – 4.1.7 The decimal type

[3] Hooman Nikmehr, Braden Phillipsa et Cheng-Chew Lima. A decimal carry-free adder. *Proceedings of SPIE – Volume 5649*, 2005

[4] Mark A. Erle, Eric M. Schwarz et Michael J. Schulte. Decimal Multiplication With Efficient Partial Product Generation. *17th IEEE Symposium on Computer Arithmetic*, 2005.

[5] IBM Corporation, Frequently asked questions about decimal arithmetic – `http://www2.hursley.ibm.com/decimal/decifaq.html`

[6] IBM Corporation, The `decnumber` package – `http://www.alphaworks.ibm.com/tech/decNumber`

[7] IBM Corporation, Decimal arithmetic for Java – `http://www2.hursley.ibm.com/decimalj/`

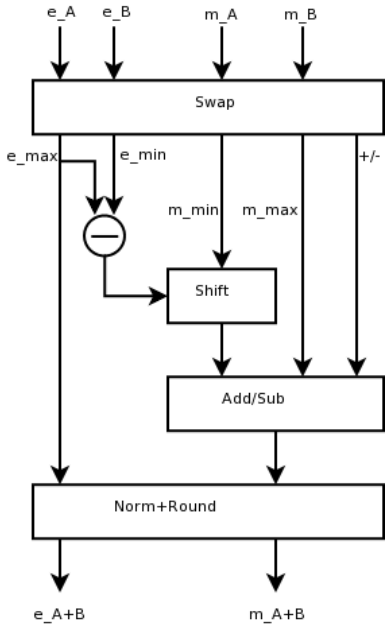

FIG. 3 – Un circuit d'addition flottante générique

	Surface (slices)		Latence (ns)	
	15 chiffres	33 chiffres	15 chiffres	33 chiffres
BCD naïf	106	232	100	190
Svoboda	316	707	30	33

FIG. 4 – Résultats en surface et en latence des additionneurs décimaux sur un FPGA Virtex, pour différentes précisions