Arithmétique des Ordinateurs

Jean-Michel Muller

ARITHMÉTIQUE DES ORDINATEURS

Jean-Michel Muller
Directeur de Recherche au CNRS
Laboratoire LIP, ENS Lyon
jean-michel.muller@ens-lyon.fr

Résumé

L'arithmétique des ordinateurs s'intéresse à tous les aspects liés à l'implantation du calcul arithmétique sur ordinateur : systèmes de numération, algorithmes, circuits et programmes de calcul, erreurs d'arrondis, fiabilité... J'essaie dans cet article de donner un petit aperçu des diverses questions qui se posent actuellement dans ce domaine.

I Introduction

Concevoir un système arithmétique pour calculer sur ordinateur n'est pas une tâche aisée, car les diverses propriétés souhaitables d'un tel système ne sont pas toujours compatibles. On voudrait tout d'abord calculer *vite :* la simulation de certains phénomènes en dynamique des fluides fait appel à des systèmes capables de faire plusieurs milliers de milliards d'opérations arithmétiques par seconde. [1] Mais bien entendu calculer vite ne sert pas à grand chose si le résultat obtenu est complètement faux : la *précision* des calculs est, elle aussi, importante. Sans parler de certains "records" tels que le calcul du plus grand nombre de décimales [2] de π, certains domaines de la physique demandent une grande précision : les interféromètres LIRGO et VIRGO, qui ont été les premiers à détecter des ondes gravitationnelles, ont été capables de détecter des compressions relatives de l'espace-temps [3] de l'ordre de 10^{-22}. Nous devons donc être parfois capables d'effectuer des suites de calculs, qui sont parfois longues, dont l'erreur relative finale n'est pas supérieure à cela. Certains algorithmes de cryptanalyse demandent la manipulation de nombres de quelques centaines à quelques milliers de chiffres (voir Boudot *et al.* (2020)). La vitesse et la précision ne sont pas tout : d'autres contraintes, plus technologiques, sont elles-aussi

1. Voir par exemple https://irfu.cea.fr/Projets/coast_documents/communication/Simulation.pdf ainsi que https://cerfacs.fr/gallery-of-images/
2. Aux dernières nouvelles, on en serait à 10^{14} décimales.
3. Voir https://www.nature.com/articles/nature.2016.19361

importantes : il faut que les opérations arithmétiques *consomment peu d'énergie* (pour diverses raisons : pour des raisons environnementales, pour que les petits dispositifs, tels que les téléphones ou les calculatrices, aient une grande autonomie, mais également pour que les supercalculateurs ne brûlent pas), et que les circuits de calcul ne soient pas énormes. Finalement, pour des calculs mettant en jeu la sécurité de personnes ou de biens précieux (par exemple, ceux effectués par le calculateur embarqué d'un avion, ou ceux d'une ligne de métro automatique) il est important de pouvoir *prouver* le bon comportement du système. Un autre souhait qui émerge actuellement est celui de la *reproductibilité* des calculs : une personne effectuant exactement les mêmes calculs que vous mais dans un contexte différent (une autre machine, une autre version du système d'exploitation, voire les mêmes mais à un autre moment) doit obtenir exactement les mêmes résultats. Les principales raisons de ce souhait sont scientifiques et techniques (on veut pouvoir vérifier un résultat affirmé par une autre équipe) mais aussi légales (pour certaines applications, on veut pouvoir expliquer devant une commission d'enquête ou un tribunal le processus qui a conduit à une décision). Toutes ces propriétés ne peuvent pas être satisfaites en même temps, et les solutions ne seront donc pas les mêmes à l'intérieur d'un téléphone portable, d'un ordinateur de bureau, d'un supercalculateur ou d'une automobile.

II La fiabilité n'a pas toujours été de mise

Nous avons tous pesté devant un ordinateur «planté», un téléphone qu'il faut redémarrer, une imprimante qui n'imprime que les fichiers des autres, ou un GPS qui nous ramène encore et encore devant la même route barrée. Il est très difficile de garantir qu'un programme ou un processeur sont exempts de « bugs », et les programmes et circuits arithmétiques n'échappent pas à cette fatalité. ll faut se méfier de l'intuition selon laquelle les algorithmes arithmétiques seraient «simples» et donc facile à implanter sans erreur : certes, les algorithmes d'addition, multiplication, division, que nous avons appris à l'école sont simples, mais si on veut de la performance, il faut souvent utiliser des algorithmes nettement plus complexes, ce qui inévitablement augmente la probabilité de laisser une erreur. Voici quelques exemples de «bugs» arithmétiques célèbres :

— Le processeur Pentium d'Intel, sorti en 1994, avait un algorithme de division faux (dans les pires cas, on n'avait que 3 chiffres significatifs). Par exemple, le calcul de $8391667/12582905$ donnait $0.666869\cdots$ au lieu de $0.666910\cdots$. L'analyse de ce «bug» est assez amusante, voir Muller (1995). La compagnie Intel a dû changer les processeurs défectueux… qu'elle a utilisé au Noël suivant pour offrir des porte-clés à ses cadres (cf Fig. 1) ;

— Sur certains ordinateurs des années 1970 (par exemple des Cray) on pouvait déclencher un overflow [4] en multipliant par 1 ;

— avec la version 6.0 de Maple (2000), en entrant 214748364810, vous obteniez 10 (si on remarque que 2147483648 est égal à 2^{32}, on peut intuiter qu'une mauvaise conversion base 10/base 2 se cache derrière ce problème). Avec la version 7.0 du même logiciel (2001), le calcul de $\frac{5001!}{5000!}$ donnait 1 au lieu de 5001 ;

4. https ://people.eecs.berkeley.edu/˜wkahan/ieee754status/754story.html

— avec les toutes premières versions d'Excel'2007 (le bug a été vite corrigé), le calcul de $65535 - 2^{-37}$, donnait 100000 (là encore, une mauvaise conversion base 10–base 2 semble être à l'origine du problème) ;
— plus près de nous, avec une calculette Casio FX 83-GT ou FX-92, calculez $11^6/13$, vous obtiendrez

$$\frac{156158413}{3600}\pi$$

qui est certes un résultat précis (la valeur affichée est effectivement très proche de $11^6/13$), mais très trompeur : la calculatrice nous fait croire qu'elle fait du calcul symbolique (comme le ferait par exemple le système de calcul formel Maple), et que le résultat affiché est exact. [5]

FIGURE 1 – Les porte-clés contenant des Pentium défectueux.

Parfois les bugs coûtent très cher. Nous avons déjà cité le cas du bug du Pentium, mais on peut aussi citer les cas suivants :

— en novembre 1998, à bord du navire de guerre américain *USS Yorktown*, un membre de l'équipage a par erreur tapé un «zéro» sur un clavier. Cela a entraîné une division par 0. Ce problème n'était pas prévu : il s'en est suivi une cascade d'erreurs qui a conduit à l'arrêt du système de propulsion ;
— le premier tir, en 1996, de la fusée Ariane 5 a été un échec à la suite d'une mauvaise interprétation d'un «overflow» (qu'il aurait suffi, à ce stade du vol, de tout bonnement ignorer [6]).

Mais même en l'absence de «bugs», certains problèmes sont intrinsèquement difficiles, et conduisent inévitablement à de larges erreurs. Laissez-moi vous conter une de mes mésaventures. Désirant sécuriser ma retraite, j'ai décidé il y a déjà longtemps de placer

$$e - 1 = 1.71828182845904523536028747135266249775724709 3\ldots$$

5. Cette calculatrice est très répandue, beaucoup d'élèves de lycée en possèdent une. Cela peut-être l'occasion de glisser un mot sur l'irrationalité de π.
6. https ://www-users.cse.umn.edu/~arnold/disasters/ariane5rep.html

euros. Je m'étais rendu à la *Société chaotique de banque*, où le banquier m'a présenté leur toute nouvelle formule :

- la première année, mon capital est multiplié par 1, et on me retire 1 euro pour frais de gestion… pas terrible ;
- la deuxième année, mon capital est multiplié par 2, et on me retire 1 euro pour frais de gestion… c'est mieux ;
- la troisième année, mon capital est multiplié par 3, et on me retire 1 euro pour frais de gestion… c'est encore mieux ;
- …
- la 25ème année, mon capital est multiplié par 25, et on me retire 1 euro pour frais de gestion ;

et je peux retirer mon argent au bout de 25 ans. Est-ce intéressant ? En sortant de la banque, pour en avoir le coeur net, j'ai fait un rapide calcul sur ma calculette, qui m'a annoncé que je disposerais de -747895876335 € (gloups…environ 20 fois la dette des USA). Le même calcul fait sur mon ordinateur de bureau était plus rassurant, et me prédisait $+1201807247$€. J'ai donc accepté l'offre. Ma déconvenue fut forte 25 ans après de constater que mon avoir n'était que d'environ 4 centimes ! L'explication du problème est assez simple : on calcule des termes successifs de la suite

$$\left\{ \begin{array}{rcl} u_0 & = & e-1 \\ u_n & = & n \cdot u_{n-1} - 1 \end{array} \right. \tag{1}$$

pour laquelle il est assez facile de vérifier par récurrence que

$$u_n = \frac{1}{n+1} + \frac{1}{(n+1)(n+2)} + \frac{1}{(n+1)(n+2)(n+3)} + \cdots$$

Quelle que soit la précision de l'arithmétique utilisée, u_0 est forcément représenté avec une petite erreur d'arrondi. Lors des itérations successives, cette erreur va être multipliée par 1, puis par 2, puis par 3, etc., de sorte que l'erreur sur u_n est de l'ordre de $n!$ fois l'erreur initiale. Il est donc normal d'obtenir n'importe quoi lorsqu'on calcule u_{25}. Il peut être amusant de constater que, pour la même raison, u_{25} peut se calculer très précisément en effectuant la récurrence (1) «à l'envers». Si on prend comme valeur (complètement fausse !) de u_{50} le nombre 42, et si on calcule successivement des approximations de u_{49}, u_{48}, etc. en utilisant

$$u_{n-1} = \frac{u_n + 1}{n} \tag{2}$$

on obtient alors une excellente approximation de u_{25}.

Cet exemple est bien entendu un peu artificiel, mais on en construit un tout-à-fait similaire en essayant de calculer, pour n grand, des valeurs de

$$I_n = \int_0^1 x^n e^{-x} dx,$$

en utilisant la relation (obtenue en intégrant par parties) $I_n = nI_{n-1} - 1/e$ et en partant de $I_0 = 1 - 1/e$. On trouve facilement que $u_n = eI_n$.

Une autre cause fréquente d'erreurs informatiques est la mauvaise qualité des spécifications. La sonde *Mars Climate Orbiter* était censée orbiter autour de Mars pour analyser son climat. Elle s'est écrasée sur la planète dès son arrivée, en 1999 : une partie de l'équipe qui a conçu les logiciels pensait que les unités de mesure utilisées étaient celles du système métrique, et l'autre partie croyait que c'était celles du système anglo-saxon.

III L'Arithmétique virgule flottante

L'arithmétique à «virgule flottante» (VF) est de loin le système le plus utilisé pour représenter des nombres réels sur ordinateur. C'est le seul qui permette d'avoir des performances suffisantes pour effecter en temps raisonnable les énormes calculs requis par la météo ou le calcul d'un profil d'avion, pour simuler l'écoulement dans une turbine, etc. L'arithmétique VF est souvent perçue comme étant juste une «approximation floue» de l'arithmétique réelle, et pourtant, comme nous allons le voir, elle est aussi une structure (au sens mathématique) parfaitement définie, sur laquelle on peut construire des algorithmes et prouver des théorèmes.

Si on se donne une *base* β ($\beta \in \mathbb{N}$, $\beta \geq 2$), une *précision* p ($p \in \mathbb{N}$, $p \geq 2$), et des *exposants extrémaux* $e_{\min}$ et $e_{\max}$ ($(e_{\min}, e_{\max}) \in \mathbb{Z}^2, e_{\min} < 0 < e_{\max}$) un nombre VF est un nombre de la forme

$$x = M \cdot \beta^{e-p+1}, \tag{3}$$

où $(M, e) \in \mathbb{Z}^2$, $|M| \leq \beta^p - 1$ (ce qui signifie que M s'écrit en base β sur au plus p chiffres), et $e_{\min} \leq e \leq e_{\max}$. Comme certains nombres peuvent avoir plusieurs représentations de la forme (3) satisfaisant les contraintes sur M et e, on demande que $|M|$ soit le plus grand possible. Ceci implique $|M| \geq \beta^{p-1}$, sauf dans le cas $e = e_{\min}$. Le nombre M est appelé *mantisse entière* de x et le nombre e est *l'exposant* de x.

La base choisie en pratique est presque toujours 2. Les calculatrices de poche et le système Maple utilisent la base 10. Une machine Russe, le Setun,[7] construite à la fin des années 1950, utilisait la base 3.

Cette représentation découle de la *notation scientifique* utilisée massivement depuis le 19ème siècle par les physiciens et les ingénieurs. On peut faire remonter cette dernière notation à très loin, puisque le système babylonien «savant» (de base 60) représentait les nombres par leur mantisse entière (de sorte que 5 et 5×60^2 avaient la même représentation), et puisque une notation exponentielle des grands nombres avait déjà été mise au point par Archimède dans son traité *l'Arénaire* (voir par exemple Hirshfeld (2009)) pour compter le nombre de grains de sable que l'on pourrait disposer dans une sphère de diamètre la taille de l'Univers. Nicolas Chuquet semble être le premier à avoir considéré des exposants pouvant être négatifs ou nuls (Flegg *et al.* (1985)). Descartes semble être à l'origine de notre notation exponentielle moderne (a^3 pour $a \times a \times a$), et c'est probablement pour cela qu'on pouvait il y a quelques années lire sur un site américain visiblement très informé que *la notation scientifique a été inventée par Descartes puis améliorée par Archimède !* La représentation virgule flottante «moderne» semble avoir été inventée par Leonardo Torres y Quevedo et Konrad Zuse (Ceruzzi (1981)).

7. Voir `https://link.springer.com/content/pdf/10.1007/978-3-642-22816-2_10.pdf`

La somme, le produit, le quotient de deux nombres virgule flottante n'est en général pas exactement représentable en virgule flottante. Il faut donc l'arrondir. Jusqu'aux années 1980, la seule chose qu'on savait était que le résultat fourni par la machine était «proche» du résultat exact, et des machines différentes pouvaient avoir des comportements très différents, ce qui rendait la mise au point des programmes très difficile. Un effort important de standardisation, sous l'impulsion de W. Kahan (Kahan (1981)), professeur à UC Berkeley, a permis de mettre un peu d'ordre dans cela. Le Standard IEEE 754, publié en 1985 et révisé en 2008 et 2019, impose la base 2, spécifie plusieurs formats et, surtout, définit l'exigence *d'arrondi correct* : des *fonctions d'arrondi* (qui sont des fonctions de $\mathbb{R}$ vers l'ensemble des nombres VF augmenté de $-\infty$ et $+\infty$) sont définies, et une fois que l'utilisateur a choisi une fonction d'arrondi $\circ$, chaque fois que l'on effectue une opération de la forme $a \top b$ (où a et b sont des nombres VF, et $\top \in \{+, -, \times, \div\}$), le résultat calculé par l'ordinateur doit être $\circ(a \top b)$. La même exigence est formulée pour la racine carrée. La fonction d'arrondi par défaut (celle que l'on obtient si on ne fait aucun choix) est *l'arrondi au plus près à arbitrage pair* (*round to nearest ties to even*), RN, défini comme suit : si t est le milieu exact de 2 nombres VF consécutifs, alors $\mathrm{RN}(t)$ est celui de ceux deux nombres dont la mantisse entière est paire, sinon $\mathrm{RN}(t)$ est le nombre VF le plus proche de t. Cette exigence d'arrondi correct, ainsi que la spécification de ce que doit faire l'ordinateur quand on demande à calculer $1/0$, $\sqrt{-5}$, etc., rend l'arithmétique complètement déterministe–tout au moins tant qu'on se contente d'utiliser les opérations arithmétiques et la racine carrée. On peut alors élaborer des algorithmes et des preuves qui utilisent cette propriété. Un exemple simple (et très utile en pratique !) est l'algorithme Fast2Sum, présenté par le théorème suivant.

Théorème 1 (Fast2Sum (Dekker))**.** *Supposons $\beta \leq 3$. Soient a et b des nombres VF vérifiant $|a| \geq |b|$. L'algorithme suivant calcule deux nombres VF s et r tels que*

— *$s + r = a + b$ exactement ;*
— *s est «le» nombre VF le plus proche de $a + b$.*

Algorithme 1 (FastTwoSum)**.**
$s \leftarrow RN(a + b)$
$z \leftarrow RN(s - a)$
$r \leftarrow RN(b - z)$

L'algorithme FastTwoSum est implanté par le programme C suivant :

```
s = a+b;
z = s-a;
r = b-z;
```

On peut ainsi en trois opérations calculer l'erreur (le terme r de l'algorithme FastTwoSum) d'une addition virgule flottante. Il est possible également de calculer l'erreur d'une multiplication virgule flottante. Pour ceci, nous devons utiliser l'opérateur arithmétique FMA (spécifié depuis la version 2008 du standard IEEE 754, et implanté sur tous les processeurs courants) :

$$\mathrm{FMA}(a, b, c) = \mathrm{RN}(ab + c),$$

Et l'erreur commise en effectuant la multiplication virgule flottante $t = \mathrm{RN}(ab)$ est tout simplement $\mathrm{FMA}(a, b, -t)$. Ces petits algorithmes de calcul de l'erreur des opérations arithmétiques

élémentaires permettent de mettre au point des algorithmes numériques précis, où l'on parvient à compenser (partiellement) les erreurs d'arrondis. De nombreux algorithmes de calcul de sommes, de produits scalaires, de normes, etc. font appel à ces techniques, comme le décrit Rump (2012). L'arithmétique complexe utilise également fréquemment ces techniques, voir par exemple Brent *et al.* (2007); Jeannerod *et al.* (2017).

On peut ainsi, par exemple, représenter des nombres avec une grande précision comme somme de deux nombres VF (une «partie haute» et une «partie basse») et mettre au point des algorithmes de manipulation de telles sommes. Une des principales difficultés que l'on rencontre est que les preuves de ces algorithmes deviennent très longues, et donc sujettes à doutes (peu de gens les lisent). Il faut souvent faire appel à des techniques de «preuve formelle» pour les valider (Boldo et Melquiond (2017); Muller et Rideau (2022)).

L'implantation très précise des fonctions mathématiques de base (sinus, cosinus, exponentielle) n'est pas un problème simple et fait toujours l'objet de nombreux travaux. On souhaiterait garantir l'arrondi correct de ces fonctions, ce qui demande la résolution d'un problème appelé *dilemme du fabricant de tables* (voir Boldo *et al.* (2023a)) et qui, très grossièrement, consiste si on s'intéresse à la fonction f, à déterminer quel est le nombre VF x tel que $f(x)$ est le plus proche du milieu exact de deux nombres VF consécutifs. Les mathématiques sous-jacentes à ce problème ne sont pas simples, voir par exemple Brisebarre *et al.* (2017). La meilleure bibliothèque de fonctions mathématiques, actuellement, est construite dans le cadre du projet CORE-MATH, porté par Paul Zimmermann au Loria. [8] Voir Sibidanov *et al.* (2022).

Terminons cette partie avec un exemple classique mais amusant, dérivé d'une idée de Malcolm (1972). Sur une arithmétique VF avec arrondi correct, l'algorithme suivant calcule la base β utilisée par votre processeur pour représenter les nombres VF (soit en général 2 ; mais le même algorithme transposé sur une calculatrice pourra donner 10). Saurez-vous trouver pourquoi ?

Algorithme 2.
$A \leftarrow 1.0$
$B \leftarrow 1.0$
while $RN\,(\,RN\,(A + 1.0) - A) = 1.0$ ***do***
  $A \leftarrow\ RN\,(2 \times A)$
end while
while $RN\,(\,RN\,(A + B) - A) \neq B$ ***do***
  $B \leftarrow\ RN\,(B + 1.0)$
end while
return B

(petit rappel : pour implanter cet algorithme sur votre machine il suffit d'omettre les «RN» puisque lorsque vous écrivez par exemple `a+b` dans un programme, ce qui est effectivement calculé est $\mathrm{RN}\,(a + b)$)

8. https ://core-math.gitlabpages.inria.fr

IV Réapprenons l'addition

Terminons ce petit tour en retournant à l'école : *comment faire une addition ?* Supposons que l'on veuille additionner deux entiers x et y de n chiffres, écrits en base 2

$$(x_{n-1}x_{n-2}x_{n-3}\cdots x_0) \quad \text{et} \quad (y_{n-1}y_{n-2}y_{n-3}\cdots y_0),$$

ce qui signifie que les x_i et les y_i valent 0 ou 1, et que

$$x = \sum_{i=0}^{n-1} x_i 2^i \quad \text{et} \quad y = \sum_{i=0}^{n-1} y_i 2^i.$$

Comment fait-on une addition ? L'algorithme élémentaire n'est pas très différent de celui de base 10 que nous avons appris à l'école. Nous partons d'une «retenue initiale» $c_0 = 0$ (nous verrons bientôt pourquoi j'introduis une retenue dès le premier chiffre). Nous calculons tout d'abord $x_0+y_0+c_0$. Si cette somme vaut 0 ou 1 elle sera le chiffre de droite s_0 de $x+y$, et nous propagerons vers l'étape suivante une retenue nulle c_1. Sinon, nous choisissons $s_0 = x_0 + y_0 + c_0 - 2$, et nous propagerons une retenue $c_1 = 1$. À l'étape suivante, nous calculons $x_1 + y_1 + c_1$, et nous prenons une décision similaire à la précédente selon que cette somme est inférieure ou égale à 1 ou pas. Bref, nous calculons itérativement la somme $s = s_n s_{n-1} s_{n-2} \cdots s_0$ comme suit :

$$\begin{array}{l} c_0 = 0, \\ \text{pour } i = 0, \ldots, n-1 \\ \qquad\qquad c_{i+1} = \begin{cases} 0 & \text{si } x_i + y_i + c_i \in \{0,1\} \\ 1 & \text{sinon} \end{cases} \\ \qquad\qquad s_i = x_i + y_i + c_i - 2c_{i+1} \\ s_n = c_n. \end{array} \tag{4}$$

Lorsqu'on effectue l'addition en utilisant (4), pour calculer s_n nous avons besoin de c_{n-1}. Mais le calcul de c_{n-1} requiert de connaître c_{n-2}. Et on ne peut calculer c_{n-2} que lorsqu'on connait c_{n-3}, et ainsi de suite. Le processus d'addition décrit par (4) est *intrinsèquement séquentiel*, et conduit à un temps de calcul qui *croît linéairement avec* n. On doit effectuer l'addition «de la droite vers la gauche». Lorsque j'effectue une addition «à la main» cela ne me pose pas de problème : je ne sais de toute façon calculer que séquentiellement. Il en va tout autrement d'un circuit intégré, dont tous les transistors peuvent fonctionner en même temps, et qui en principe serait capable d'effectuer simultanément des calculs portant sur tous les chiffres de x et y. Une des solutions pour accélérer le calcul est la suivante. Supposons pour simplifier que n est une puissance de 2, et que nous savons déjà additionner rapidement des nombres de $n/2$ chiffres. Pour effectuer l'addition

```
  10010110010110111100010111100011010100101101001 01
+ 01101001001110011010010111100101010010111101101 10
```

la première solution qui vient à l'esprit est de couper chacun des nombres x et y en deux paquets de $n/2$ bits :

```
10010110010110111110001011        11000110101001011010 0101

01101001001110011010 0101         11100101010010111110110110
```

d'additionner les paquets de droite, puis lorsqu'on connait la retenue sortante de cette addition, de l'utiliser comme retenue entrante pour additionner les paquets de gauche. En faisant ainsi, si on appelle T_n le temps de l'addition de nombres de n chiffres, on a $T_n = 2T_{n/2}$: le temps de calcul reste proportionnel à n, et on n'a rien gagné. Mais puisque la retenue sortante de l'addition des paquets de droite ne peut rendre que 2 valeurs possibles (0 ou 1), on peut travailler sur les deux hypothèses en parallèle. Dupliquons les paquets de gauche, et décidons, *dès le début du calcul, sans attendre, de faire en parallèle trois additions de nombres de $n/2$ bits* : celle des paquets de droite, celle des paquets de gauche en supposant une retenue entrante nulle, et celle des paquets de gauche en supposant une retenue entrante égale à 1 :

```
                                  0
  10010110010110111110001011        11000110101001011010 0101
+ 01101001001110011010 0101       + 11100101010010111110110110
```

```
                                  1
  10010110010110111110001011
+ 01101001001110011010 0101
```

Au bout d'un temps $T_{n/2}$, ces trois additions sont terminées, et au vu de la retenue sortante de l'addition des paquets de droite, on peut choisir lequel des deux résultats des additions des paquets de gauche est le bon :

```
                                  0
  10010110010110111110001011        11000110101001011010 0101
+ 01101001001110011010 0101       + 11100101010010111110110110
    yyyyyyyyyyyyyyyyyyyyyyyy        1xxxxxxxxxxxxxxxxxxxxxxxxxx
```

```
                                  1
  10010110010110111110001011
+ 01101001001110011010 0101
    zzzzzzzzzzzzzzzzzzzzzzzz
```

Le temps de calcul T_n devient alors égal à $T_{n/2} + C$ où C est un terme constant (qui est le temps—tout petit—requis par les circuits logiques utilisés pour choisir entre les deux paquets de gauche). On en déduit facilement que T_n devient proportionnel à $\log(n)$. En pratique le gain est considérable dès que n est un peu grand, et dans les processeurs, des solutions semblables à celle-ci (ou d'autres solutions à temps logarithmique, voir Knowles (2001)) sont utilisées pour

manipuler des nombres de 32 bits ou plus.

En utilisant un théorème de Winograd, on peut montrer que dans nos systèmes usuels de numération (par exemple, base 2 et chiffres 0 ou 1, ou base 10 et chiffres compris entre 0 et 9), sous des hypothèses raisonnables (essentiellement, le nombre d'entrées d'une «porte logique» est borné), un circuit ne peut pas additionner ou multiplier des nombres de n chiffres en un temps meilleur que logarithmique en n. Mais on peut tout de même aller plus vite en changeant notre manière de représenter les nombres. Cela nécessite cependant de renoncer à l'*unicité* de la représentation, ce qui ne va pas sans poser d'autres problèmes (les comparaisons par exemple deviennent nettement plus difficiles, la consommation de mémoire sera supérieure). Par exemple Avizienis (1961) a proposé de représenter les nombres en base 10, avec des chiffres allant de -6 à $+6$. Dans ce système de numération, certains nombres ont plusieurs représentations possibles : il est dit *redondant*. Le lecteur constatera aisément que l'algorithme ci-dessous permet à un circuit de calculer en «temps constant» (i.e., indépendant de la taille n des entrées) la somme de deux nombres x et y écrits dans ce système (le chiffre s_i se choisit au vu d'une «fenêtre» de 2 chiffres seulement de x et de y).

Algorithme 3 (Addition d'Avizienis)**.**

1. *Calculer pour* $i = 0 \dots n-1$ *:*

$$\begin{array}{rcl} t_{i+1} & = & \left\{ \begin{array}{rll} -1 & si & x_i + y_i \leq -6 \\ 0 & si & -5 \leq x_i + y_i \leq 5 \\ 1 & si & x_i + y_i \geq 6 \end{array} \right. \\ w_i & = & x_i + y_i - 10 t_{i+1} \end{array}$$

2. *Calculer pour* $i = 0 \dots n$ *:* $s_i = w_i + t_i$*, avec* $w_n = t_0 = 0$*.*

Des systèmes de numération redondants un peu analogues (en base 2) sont utilisés à l'intérieur de certains circuits de multiplication et division, pour accélérer les nombreuses additions que ces opérations nécessitent. Le résultat final est converti dans un système usuel, de sorte que cette utilisation d'un système redondant en interne est complètement transparente pour l'utilisateur.

V Pour en savoir plus

J'espère vous avoir persuadé que même l'addition et la multiplication sont encore (un peu) du domaine de la recherche. Le lecteur qui souhaite en savoir plus sur l'arithmétique virgule flottante pourra consulter Boldo *et al.* (2023b). Les algorithmes utilisés en matériel pour effectuer des opérations arithmétiques sont décrits dans plusieurs ouvrages (il suffit de taper «computer arithmetic» ou «digital arithmetic» sur Google), mais j'ai un petit faible pour Ercegovac et Lang (2004). Sur l'utilisation de techniques de preuve formelle pour valider des algorithmes critiques, le livre de Boldo et Melquiond (2017) est un incontournable. La personne désirant suivre les travaux de la communauté française d'arithmétique des ordinateurs peut s'inscrire au groupe de travail «Arith» du GDR IM sur le site `https://mygdr.hosted.lip6.fr/`.

VI Bibliographie

AVIZIENIS, A. (1961). Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, 10:389–400.

BOLDO, S., BRISEBARRE, N. et MULLER, J.-M. (2023a). Le dilemme du fabricant de tables. *La Recherche*, (572).

BOLDO, S., JEANNEROD, C.-P., MELQUIOND, G. et MULLER, J.-M. (2023b). Floating-point arithmetic. *Acta Numerica*, 32:203–290.

BOLDO, S. et MELQUIOND, G. (2017). *Computer Arithmetic and Formal Proofs*. ISTE Press - Elsevier.

BOUDOT, F., GAUDRY, P., GUILLEVIC, A., HENINGER, N., THOMÉ, E. et ZIMMERMANN, P. (2020). Nouveaux records de factorisation et de calcul de logarithme discret. *Techniques de l'Ingénieur*, 12-2020:1–10.

BRENT, R., PERCIVAL, C. et ZIMMERMANN, P. (2007). Error bounds on complex floating-point multiplication. *Mathematics of Computation*, 76:1469–1481.

BRISEBARRE, N., HANROT, G. et ROBERT, O. (2017). Exponential sums and correctly-rounded functions. *IEEE Transactions on Computers*, 66(12):2044–2057.

CERUZZI, P. E. (1981). The early computers of Konrad Zuse, 1935 to 1945. *Annals of the History of Computing*, 3(3):241–262.

ERCEGOVAC, M. D. et LANG, T. (2004). *Digital Arithmetic*. Morgan Kaufmann Publishers, San Francisco, CA.

FLEGG, G., HAY, C. et MOSS, B. (1985). *Nicolas Chuquet, Renaissance Mathematician, A study with extensive translation of Chuquet's mathematical manuscript completed in 1484*. Springer Dordrecht.

HIRSHFELD, A. (2009). *Eureka Man, The life and legacy of Archimedes*. Walker & Company.

JEANNEROD, C.-P., KORNERUP, P., LOUVET, N. et MULLER, J.-M. (2017). Error bounds on complex floating-point multiplication with an FMA. *Mathematics of Computation*, 86:881–898.

KAHAN, W. (1981). Why do we need a floating-point arithmetic standard ? Rapport technique, Computer Science, UC Berkeley. `http://www.cs.berkeley.edu/~wkahan/ieee754status/why-ieee.pdf`.

KNOWLES, S. (2001). A family of adders. *In Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001*, pages 277–281.

MALCOLM, M. A. (1972). Algorithms to reveal properties of floating-point arithmetic. *Communications of the ACM*, 15(11):949–951.

MULLER, J.-M. (1995). Algorithmes de division pour microprocesseurs : illustration à l'aide du " Bug " du Pentium. *Technique et Science Informatiques*, 14(8):1031–1049. https ://hal.science/hal-04143937/file/MullerPentiumBug95.pdf.

MULLER, J.-M. et RIDEAU, L. (2022). Formalization of double-word arithmetic, and comments on "Tight and rigorous error bounds for basic building blocks of double-word arithmetic". *ACM Transactions on Mathematical Software*, 48(2):1–24.

RUMP, S. M. (2012). Error estimation of floating-point summation and dot product. *BIT Numerical Mathematics*, 52(1):201–220.

SIBIDANOV, A., ZIMMERMANN, P. et GLONDU, S. (2022). The CORE-MATH project. *In 29th IEEE Symposium on Computer Arithmetic*. `https://hal.inria.fr/hal-03721525`.