Preuves en arithmétique virgule flottante

Jean-Michel Muller
CNRS - Laboratoire LIP

LIMOS Keynote – 29 septembre 2022

Arithmétique « virgule flottante »

- de très loin la + utilisée pour du calcul numérique ;
- trop souvent perçue comme un tas de recettes de cuisine ;
- de simples modèles tels que

  en l'absence d'overflow/underflow, la valeur calculée de $(a \top b)$ *vaut* $(a \top b) \cdot (1 + \delta), \quad |\delta| \leq 2^{-p},$

  (en base 2, mantisses de p bits et arrondi au plus près, $\top \in \{\pm, \times, \div\}$) sont très utiles, mais ne permettent pas de capter des comportements subtils, comme dans

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

  et beaucoup d'autres.

Propriétés souhaitables d'une arithmétique machine

- Critères de performance :
  - Vitesse : météo de demain en moins de 24 h ;
  - Précision : certaines prédictions physiques vérifiées avec erreur relative $\approx 10^{-15}$;
  - fiabilité : tout est construit au-dessus de l'arithmétique ;
- Critères technologiques
  - « taille » : surface de circuit, taille du code, consommation mémoire ;
  - Energie consommée : autonomie, coût carbone, chauffe des circuits ;
- Critères de coût humain
  - Portabilité et reproducibilité : les programmes mis au point sur un système doivent tourner sur un autre sans requérir des modifications longues et/ou complexes ; on doit pouvoir «rejouer» un calcul ;
  - Simplicité d'implantation et d'utilisation

Système virgule flottante

Paramètres : $\begin{cases} \text{base} & \beta \geq 2 \\ \text{précision} & p \geq 1 \\ \text{exposants extrèmes} & e_{\min}, e_{\max} \end{cases}$

Un nombre VF fini x est représenté par 2 entiers :

- mantisse entière : M, $|M| \leq \beta^p - 1$;
- exposant e, $e_{\min} \leq e \leq e_{\max}$.

tels que

$$x = M \times \beta^{e+1-p}$$

Si plusieurs choix, on prend e minimal sous ces contraintes. On appelle mantisse réelle, ou mantisse de x le nombre

$$m = M \times \beta^{1-p},$$

ce qui donne $x = m \times \beta^e$, avec $|m| < \beta$.

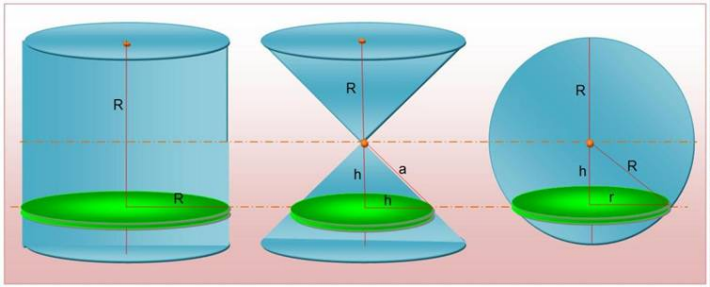
Les Mésopotamiens inventent les mantisses. . .

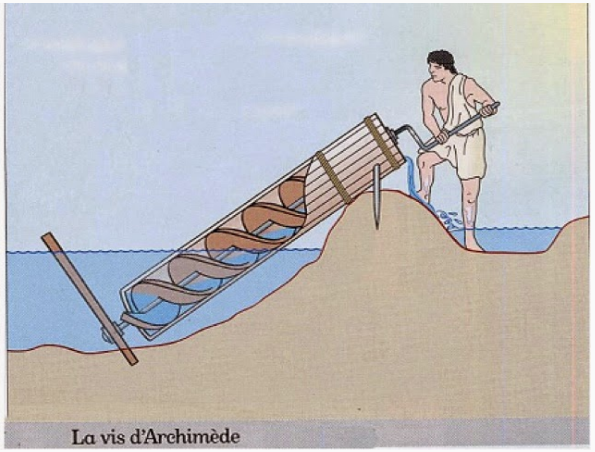
- actuel Irak, vers -2000 ;
- Système de base 60 (58 tables de multiplication à connaître !) ;
- pas de zéro « à la fin » : on manipule juste des mantisses (comme si dans notre système 25, 0.025 et 250 avaient la même représentation).

. . . et Archimède (-287 – -212) invente les exposants

- Traité l'Arénaire (compteur de sable : *arena* = sable en Latin) ;
- nombre de grains de sable qui pourraient remplir l'Univers ;
- notation exponentielle pour représenter les ordres de grandeur.

C'est Le génie scientifique de l'antiquité.





. . . et Archimède invente les exposants

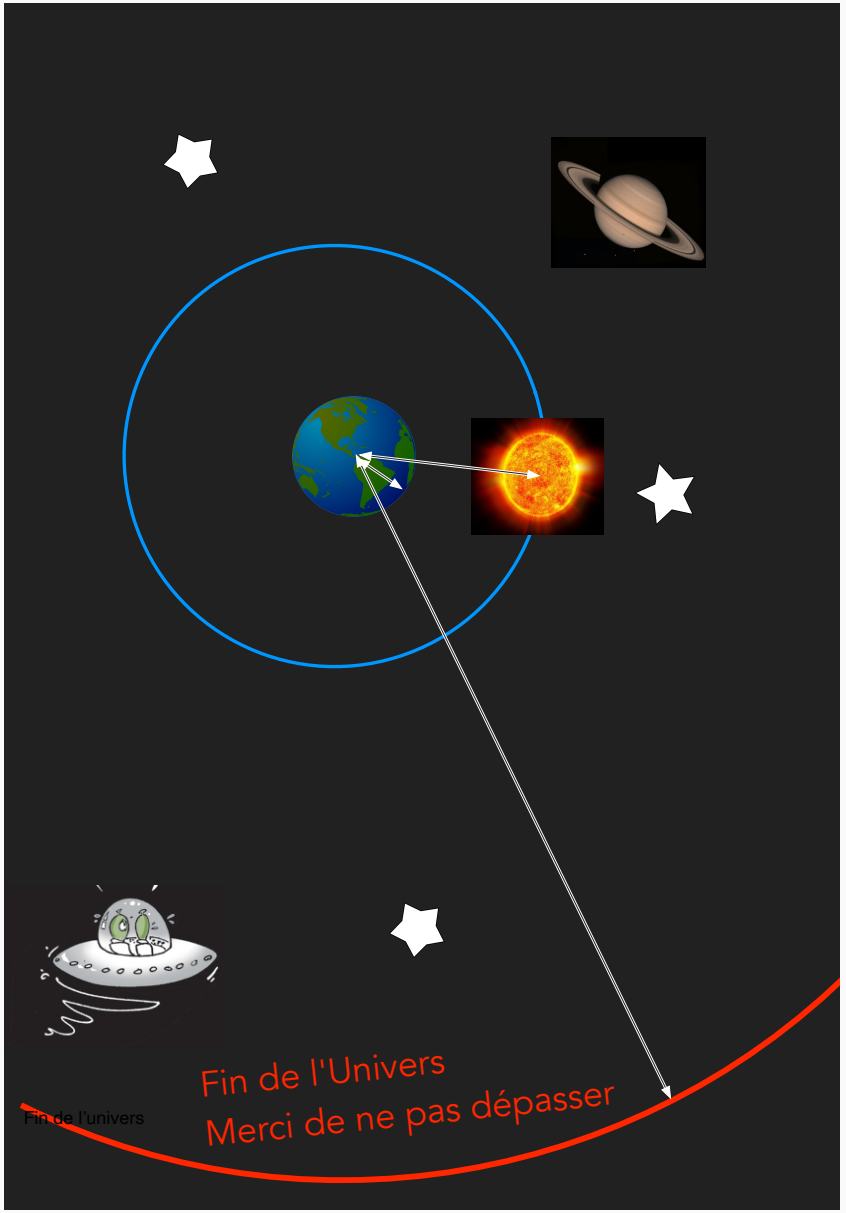
Hypothèse :

$$\frac{\text{rayon Univers}}{\text{distance Terre-Soleil}}$$

$$= \frac{\text{distance Terre-Soleil}}{\text{rayon Terre}}$$

Réponse d'Archimède :

on fait tenir 10^{63} grains de sable dans l'Univers.



Leonardo Torres y Quevedo (1852–1936)

- version électromécanique (à relais) de la machine analytique de Babbage ;
- première proposition d'une arithmétique VF ;

L'article de Torres « Essais sur l'Automatique »

- En Français, *Revue Générale des Sciences*, 15 novembre 1915 ;
- contient le paragraphe :

Parfois aussi, pour ne pas avoir à écrire beaucoup de zéros, on écrit les quantités sous la forme $n \times 10^m$.

Nous pourrons simplifier beaucoup cette écriture en établissant arbitrairement ces trois règles très simples :

I. n aura toujours le même nombre de chiffres (six par exemple).

II. Le premier chiffre de n sera de l'ordre des dixièmes, le second des centièmes, etc.

III. On écrira chaque quantité sous cette forme: n, m.

Ainsi, au lieu de 2435,27 et de 0,00000341862, on écrira respectivement 243527 ; 4 et 341862 ; — 5.

Je n'ai pas indiqué de limite pour la valeur de l'exposant, mais il est évident que, dans tous les calculs usuels, il sera plus petit que cent, de sorte que, dans ce système, on écrira toutes les quan-

Konrad Zuse (1910–1995) et le Z3 (1941)

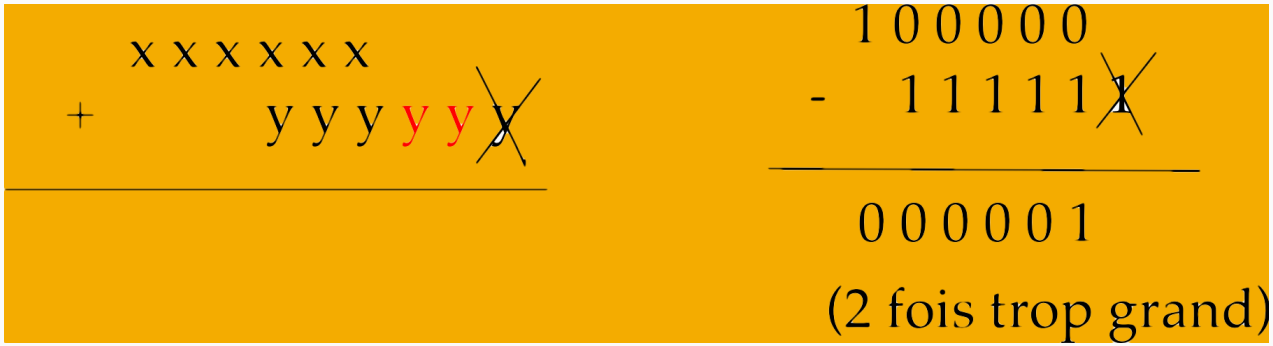
Zuse posant devant une reconstruction du Z3

- Z3 : Arithmétique VF de base 2, nombres sur 22 bits :
  - mantisses de 14 bits ;
  - exposants de 7 bits ;
  - 1 bit de signe ;
- représentations spéciales pour $\pm\infty$ et résultats indéterminés, plus de 40 ans avant IEEE 754 ;
- contrairement aux Z1 (1936–1938) et Z2 (1938), a été complètement opérationnel.

Années 50 et 60 : la VF se généralise. . . mais c'est la pagaille

- Base : 2, 4, 8, 10, 16. . . il y a même eu une machine russe de base 3. Des études de Brent et Cody trancheront ;
- seuils d'over/underflow très différents ;
- pas la même manière de gérer $1/0$, $0/0$, $\sqrt{-1}$, etc. ;
- spécification floue des opérations : on raisonne en « bits de garde »

Alignement lorsqu'on calcule $x + y$ avec $e_x > e_y$:



`(1 - x)` gagnait à être remplacé par `(0.5 - x) + 0.5`

Quelques exemples (Kahan)

- Quand seule la vitesse compte : sur les Crays, l'overflow était calculé à partir des exposants des entrées, en parallèle avec le calcul effectif du produit des mantisses

→ `1 * x` peut faire un overflow ;

- sur les mêmes, seuls 12 bits de x étaient examinés pour détecter une division par 0 lors du calcul y/x

→ `if (x = 0) then z := 17.0 else z := y/x`
pouvait provoquer une erreur « division par zéro »...
mais comme le multiplieur aussi ne regardait que 12 bits pour décider qu'une opérande est nulle,

`if (1.0 * x = 0) then z := 17.0 else z := y/x`

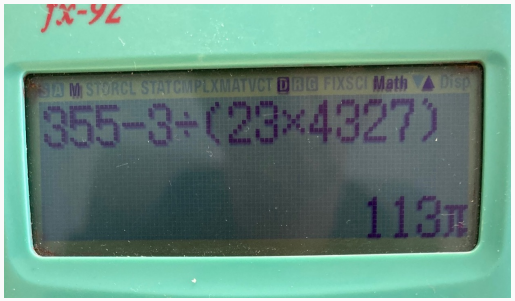
→ plus de problème.

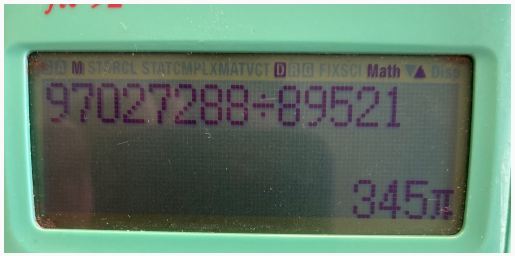
Quelques étrangetés plus récentes

- Maple, version 6.0 (2000). Entrez 214748364810, vous obtiendrez 10.
- Excel'2007 (premières versions), calculez $65535 - 2^{-37}$, vous obtiendrez 100000 ;
- si vos enfants ont une calculette Casio FX 83-GT ou FX-92, calculez $11^6/13$, vous obtiendrez

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

Vous en voulez plus ?





William Kahan

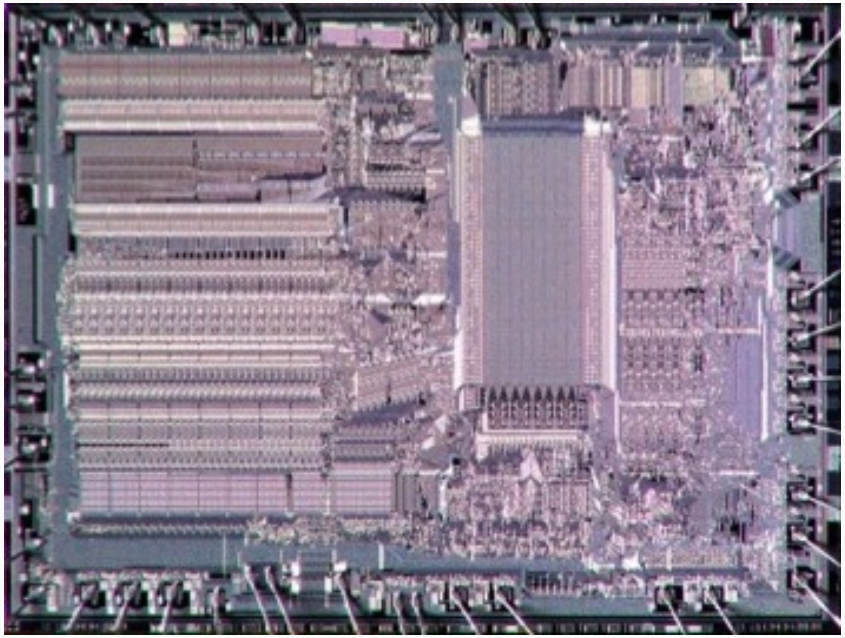
- PhD, Univ. Toronto, 1958 ;
- à programmé à peu près toutes les machines de l'époque ;
- a contribué à la conception de la calculatrice HP35 (1972) ;
- arithmétique VF du 8087 ;
- en parallèle (1977), 1ères discussions autour du futur standard IEEE 754.

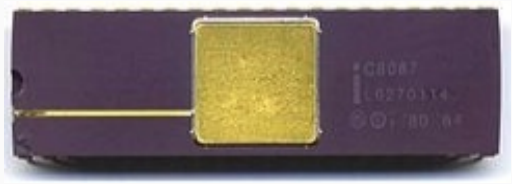
La HP35 : l'objet qui a « tué » la règle à calcul

Encore beaucoup d'informations à

`http://www.eecs.berkeley.edu/~wkahan`

Le 8087 d'Intel (1980)





- coprocesseur du 8086 ;
- $ 200 environ ;
- fonctions élémentaires en VF ;
- première « presque » implantation de ce qui sera 5 ans + tard IEEE 754 ;
- entrée de la VF rapide et propre dans le monde du personal computing ;
- 5 MHz.

La norme IEEE 754-1985

- choix de la base 2, de formats (à l'époque juste 32 et 64 bits) ;
- deux idées fortes :
  - système clos : même les opérations « illicites » $(1/0\,;\,\sqrt{-5})$ fournissent un résultat, qui doit pouvoir être réutilisable en entrée ;
  - arrondi correct : une fonction d'arrondi $\circ$ étant choisie, le calcul en machine de $a \star b$ donne

$$\circ(a \star b)$$

- a été révisée en 2008 et 2019.

Juste quelques mots sur les exceptions dans la norme

- philosophie par défaut : le calcul doit toujours continuer ;
- deux infinis, et deux zéros. Règles intuitives : $1/(+0) = +\infty$, $5 + (-\infty) = -\infty \ldots$;
- tout de même un truc bizarre : $\sqrt{-0} = -0$;
- Not a Number (NaN) : résultat de $\sqrt{-5}$, $(\pm 0)/(\pm 0)$, $(\pm\infty)/(\pm\infty)$, $(\pm 0) \times (\pm\infty)$, NaN $+3$, etc.

Arrondi correct

En général, la somme, le produit, etc. de deux nombres VF n'est pas un nombre VF $\rightarrow$ nécessité de l'arrondir.

Définition 1 (Arrondi correct)

Fonction d'arrondi $x \mapsto \circ(x)$ parmi :

- RN (x) : au plus près (défaut) s'il y en a deux :
  - *round ties to even :* celui dont la mantisse entière est paire ;
  - *round ties to away :* (2008 – recomm. en base 10 seulement) celui de plus grande valeur absolue.
- RU (x) : vers $+\infty$.
- RD (x) : vers $-\infty$.
- RZ (x) : vers zéro.

Une opération dont les entrées sont des nombres VF doit retourner ce qu'on obtiendrait en arrondissant le résultat exact.

Arrondi correct

IEEE-754-1985 : Arrondi correct pour $+$, $-$, $\times$, $\div$, $\sqrt{}$ et certaines conversions. Avantages :

- si le résultat d'une opération est exactement représentable, on l'obtient ;
- si on n'utilise que $+$, $-$, $\times$, $\div$ et $\sqrt{}$, et si l'ordre des opérations ne change pas, l'arithmétique est déterministe : on peut élaborer des algorithmes et des preuves qui utilisent ces spécifications ;
- avec RN, en l'absence d'underflow/overflow, l'erreur relative d'une opération est $\leq u := 2^{-p}$ ("modèle standard") ;
- précision et portabilité améliorées.

IEEE-754-2008 : l'arrondi correct est recommandé (mais pas exigé) pour les principales fonctions mathématiques (cos, exp, log, ...)

Erreur de l'addition VF (Møller, Knuth, Dekker)

Premier résultat : représentabilité. $\mathrm{RN}(x) = x$ arrondi au plus près.

Lemme 1

Soient a et b deux nombres VF. Soient

$$s = RN(a + b)$$

et

$$r = (a + b) - s.$$

s'il n'y a pas de dépassement de capacité en calculant s, alors r est un nombre VF.

Erreur de l'addition VF (Møller, Knuth, Dekker)

Proof without loss of generality, assume $|a| \geq |b|$

① s is "the" FPN nearest $a+b$ → it is closer to $a+b$ than a is

→ $|s - (a+b)| \leq |a - (a+b)|$

therefore $|r| \leq |b|$

② denote $a = M_a \cdot \beta^{e_a - p + 1}$; $b = M_b \cdot \beta^{e_b - p + 1}$

with $|M_a|, |M_b| \leq \beta^p - 1$ and $e_a \geq e_b$.

$a+b$ multiple of $\beta^{e_b - p + 1}$ ⇒ s and r multiple of $\beta^{e_b - p + 1}$ too

⇒ $\exists\ R \in \mathbb{Z}$ s.t. $r = R \cdot \beta^{e_b - p + 1}$

But $|r| \leq |b| \Rightarrow |R| \leq |M_b| \leq \beta^p - 1$

→ r is a FPN !

Obtenir r : l'algorithme fast2sum (Dekker)

Théorème 1 (Fast2Sum (Dekker))

(base ≤ 3) Soient a et b des nombres VF vérifiant $|a| \geq |b|$. Algorithme suivant : s et r t.q.

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

Algorithme 1 (FastTwoSum)

$s \leftarrow RN(a + b)$

$z \leftarrow RN(s - a)$

$r \leftarrow RN(b - z)$

Programme C 1

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

Se méfier des compilateurs « optimisants ».

Algorithme TwoSum (Møller-Knuth)

- pas besoin de comparer a et b ;
- 6 opérations au lieu de 3 $\rightarrow$ moins cher qu'une mauvaise prédiction de branchement en comparant a et b.

Algorithme 2 (TwoSum)

$$
\begin{aligned}
s &\leftarrow RN(a+b)\\
a' &\leftarrow RN(s-b)\\
b' &\leftarrow RN(s-a')\\
\delta_a &\leftarrow RN(a-a')\\
\delta_b &\leftarrow RN(b-b')\\
r &\leftarrow RN(\delta_a+\delta_b)
\end{aligned}
$$

Knuth : $\forall\beta$, en absence d'underflow et d'overflow $a+b=s+r$, et s est le nombre VF le plus proche de $a+b$.

Boldo et al : (preuve formelle) en base 2, marche même si underflow.

Preuves formelles (en Coq) d'algorithmes similaires très pratiques :
`http://lipforge.ens-lyon.fr/www/pff/Fast2Sum.html`.

TwoSum est optimal

Supposons qu'un algorithme vérifie :

- pas de tests, ni d'instructions min/max ;
- seulement des additions/soustractions arrondies au + près : à l'étape i, on calcule $\mathrm{RN}(u+v)$ ou $\mathrm{RN}(u-v)$, où u et v sont des variables d'entrée ou des valeurs précédemment calculées.

Si cet algorithme retourne toujours les mêmes résultats que 2Sum, alors il nécessite au moins 6 *additions/soustractions (i.e., autant que 2Sum).*

- **preuve :** most inelegant proof award ;
- 480756 algorithmes avec 5 opérations (après suppression des symétries les plus triviales) ;
- chacun d'entre eux essayé avec 2 valeurs d'entrée bien choisies.

TwoSum et Fast2Sum sont « Robustes »

2Sum avec arrondis quelconques : $\circ_1, \circ_2, \ldots, \circ_6$ fonctions d'arrondi $\in \{\,\mathrm{RU}\,,\,\mathrm{RD}\,,\,\mathrm{RZ}\,,\,\mathrm{RN}\,\}$. Base 2 et précision p.

(1) $s \leftarrow \circ_1(a+b)$
(2) $a' \leftarrow \circ_2(s-b)$
(3) $b' \leftarrow \circ_3(s-a')$
(4) $\delta_a \leftarrow \circ_4(a-a')$
(5) $\delta_b \leftarrow \circ_5(b-b')$
(6) $t \leftarrow \circ_6(\delta_a+\delta_b)$

Théorème 2 (Boldo, Graillat, M.)

En l'absence d'overflow, si $p \geq 4$, s et t vérifient

$$|(a+b)-(s+t)| < 2^{-2p+2} \cdot |s|$$

De plus si $e_s - e_b \leq p-1$ alors t est un des 2 nombres VF qui encadrent $(a+b)-s$.

Théorème 3 (Boldo, Graillat, M.)

Si $|a| < \Omega := 2^{e_{max}}(2-2^{1-p})$ et s'il n'y a pas d'overflow à la ligne (1), il n'y en a pas aux lignes (2) à (6).

Exemple : calcul de $x_1 + x_2 + \cdots + x_n$

Ogita, Rump, Oishi :

Algorithme 3

$s \leftarrow x_1$

$e \leftarrow 0$

for $i = 2$ *to* n ***do***

$(s, e_i) \leftarrow \mathit{2Sum}(s, x_i)$

$e \leftarrow \mathit{RN}(e + e_i)$

end for

return $\mathit{RN}(s + e)$

$\rightarrow$ variante de l'algorithme naïf, où à chaque pas on accumule les erreurs des additions pour les rajouter à la fin.

Et les produits ?

- FMA : *fused multiply-add* (fma), calcule $\mathrm{RN}(ab + c)$. Introduit dans l'IBM RS6000, puis IBM PowerPC et Intel Itanium, partout depuis. Spécifié depuis IEEE 754-2008
- si a et b sont des nombres VF, alors $r = ab - \mathrm{RN}(ab)$ est un nombre VF ;
- obtenu par l'algorithme TwoMultFMA $\begin{cases} p &= \mathrm{RN}(ab) \\ r &= \mathrm{RN}(ab - p) \end{cases}$

  $\rightarrow$ 2 opérations seulement, donne

$$p + r = ab.$$

$ad - bc$ «naïf» avec un fma

- on a envie de calculer $w = \mathrm{RN}(bc)$;
- puis (avec un fma), $\hat{x} = \mathrm{RN}(ad - w)$.

peut être catastrophique (bien pire que de faire $\mathrm{RN}(\mathrm{RN}(ad) - \mathrm{RN}(bc))$). En effet, considérons le cas :

- $a = b$, et $c = d$;
- ad n'est pas exactement représentable en VF : $w = \mathrm{RN}(ad) \neq ad$.

On aura :

- la valeur exacte de $x = ad - bc$ est nulle ;
- $\hat{x} \neq 0$,

$\rightarrow$ l'erreur relative $|\hat{x} - x|/|x|$ est infinie.

$ad - bc$ précis avec un fma (base 2)

Algorithme de Kahan pour $x = ad - bc$:

$w \leftarrow \mathrm{RN}(bc)$
$e \leftarrow \mathrm{RN}(w - bc)$
$f \leftarrow \mathrm{RN}(ad - w)$
$\hat{x} \leftarrow \mathrm{RN}(f + e)$

$\mathbf{u} = 2^{-p}$

- avec le «modèle standard» :

$$|\hat{x} - x| \leq J|x|$$

où $J = 2\mathbf{u} + \mathbf{u}^2 + (\mathbf{u} + \mathbf{u}^2)\mathbf{u}\frac{|bc|}{|x|}$ $\rightarrow$ précis tant que $\mathbf{u}|bc| \not\gg |x|$

- en utilisant les propriétés de RN (Jeannerod, Louvet, M., 2011)

$$|\hat{x} - x| \leq 2\mathbf{u}|x|$$

asymptotiquement optimal.

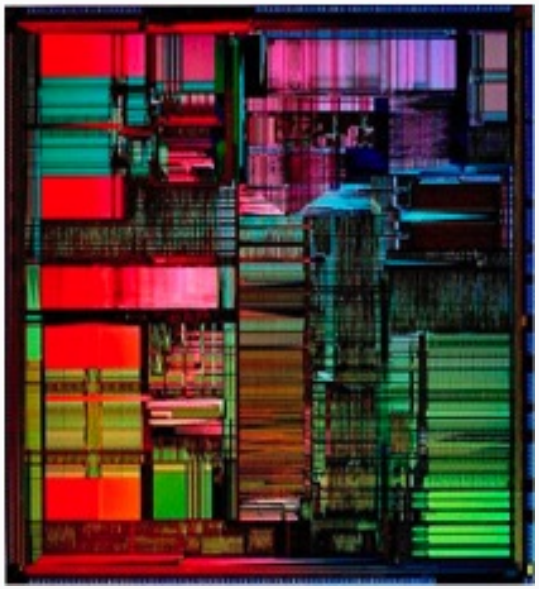
$\rightarrow$ $\times$ et $\div$ complexes.

Arithmétique (presque) bien spécifiée

Dès 1985, tout est prêt pour prouver rigoureusement les algorithmes utilisés, sauf qu'à l'époque...

- les ingénieurs/scientifiques n'en éprouvent pas vraiment le besoin : ils font de la simulation tranquilles au sol ;
- les outils de preuve formelle ne sont pas encore prêts ;
- culte du secret sur les algorithmes utilisés ;
- aucune spécification des fonctions transcendantes simples : exp, log, sin, cos, etc.
- ... et puis chez Intel, Motorola, etc. il y avait à ce moment là un côté « tonton bricoleur » sympathique mais dangereux.

Automne 1994 : le « bug » du Pentium



- Thomas Nicely (Lynchburg Univ.) : constante de Brun

$$\left(\frac{1}{3}+\frac{1}{5}\right)+\left(\frac{1}{5}+\frac{1}{7}\right)+\left(\frac{1}{11}+\frac{1}{13}\right)+\cdots$$

(couples de nombres 1ers jumeaux).

- résultats pas en accord avec les précédents. Dans un tel cas on soupçonne :
  1. le programme ;
  2. les résultats précédents ;
  3. le compilateur ;
  4. en dernier recours le processeur.
- le Pentium donnait un résultat incorrect pour 1/824633702441 (824633702441 et 824633702443 sont jumeaux).

Automne 1994 : le « bug » du Pentium

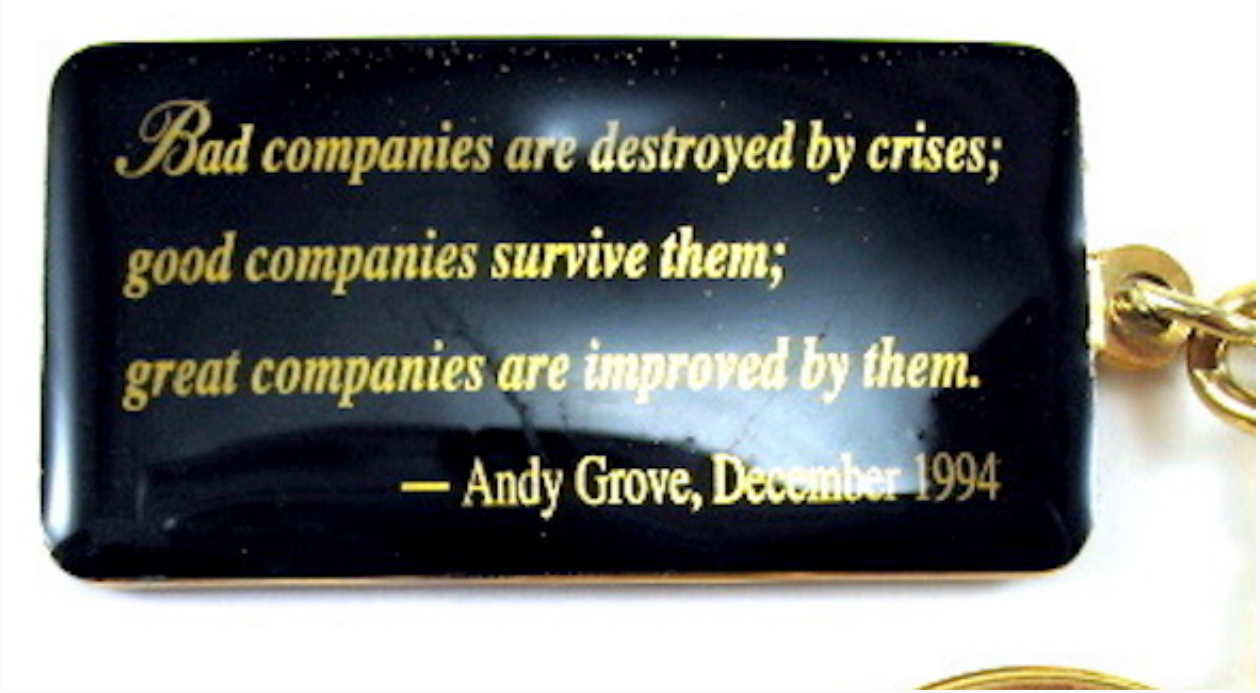
- erreur dans l'algorithme de division (SRT de base 4) ;
- nombreux quotients faux. Pire cas : 4195835.0/3145727.0 donne 1.33373906802 au lieu de 1.3338204491 ;
- Intel a dû remplacer les Pentium défectueux (coût : peut-être 400M$) ;
- la vraie perte a été en termes d'image de marque.

Après ceci : vrai changement de stratégie

- fin du secret sur les algorithmes VF : division et racine carrée de l'Itanium publiées dans les actes d'Arith14 (1999) ;
- preuve formelle : Intel embauche Harrison, AMD embauche Russinoff.

Que sont les Pentium devenus ?

Que sont les Pentium devenus ?



Arithmétique «double mot»

- Fast2Sum, 2Sum et 2MultFMA renvoient une somme non évaluée de deux nombres VF ;
- idée : manipuler de telles sommes non évaluées pour faire des calculs plus précis dans des parties critiques d'un programme.

$\rightarrow$ Arithmétique double-double» ou «double mot». Avatar le + récent : «pair arithmetic» de Lange et Rump (2020)

Définition 4

Un nombre «double mot» (DW : double-word) x est une somme non évaluée $x_h + x_\ell$ de deux nombres VF x_h et x_ℓ tels que

$$x_h = \mathrm{RN}(x).$$

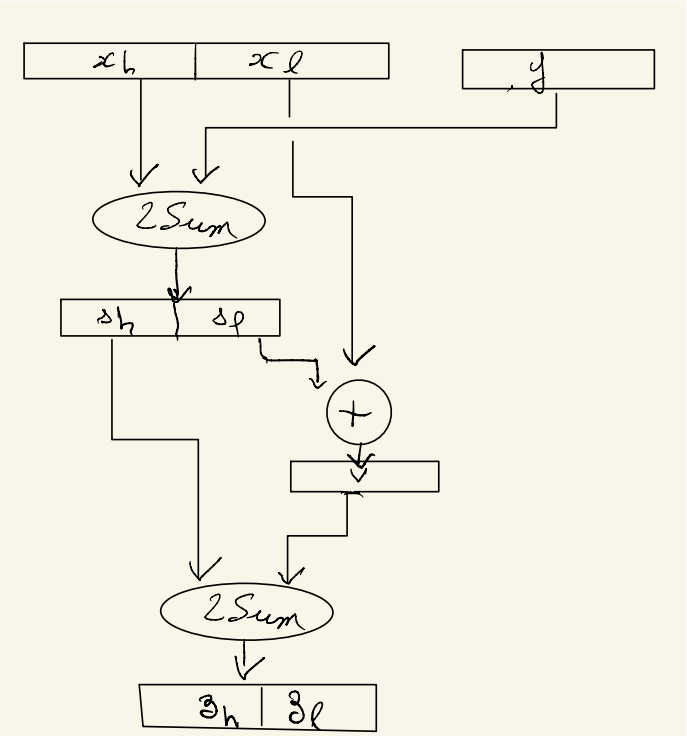
Par la suite : base 2, précision p.

DW+FP

- Bibliothèque QD de Bailey (1999) ;
- nombre DW $x = x_h + x_\ell$ plus nombre VF $y \to$ nombre DW z ;
- mesure d'erreur $u = 2^{-p}$.

DWPlusFP

1: $(s_h, s_\ell) \leftarrow \mathrm{2Sum}(x_h, y)$
2: $v \leftarrow \mathrm{RN}(x_\ell + s_\ell)$
3: $(z_h, z_\ell) \leftarrow \mathrm{Fast2Sum}(s_h, v)$
4: **return** (z_h, z_ℓ)



Théorème 5

L'erreur relative

$$\left|\frac{(z_h + z_\ell) - (x + y)}{x + y}\right|$$

de DWPlusFP est majorée par $2 \cdot u^2$.

La borne ne peut pas être améliorée, elle est asymptotiquement optimale :

Pour $x_h = 1$, $x_\ell = (2^p - 1) \cdot 2^{-2p}$, $y = -\frac{1}{2} \cdot (1 - 2^{-p})$ l'erreur relative obtenue est

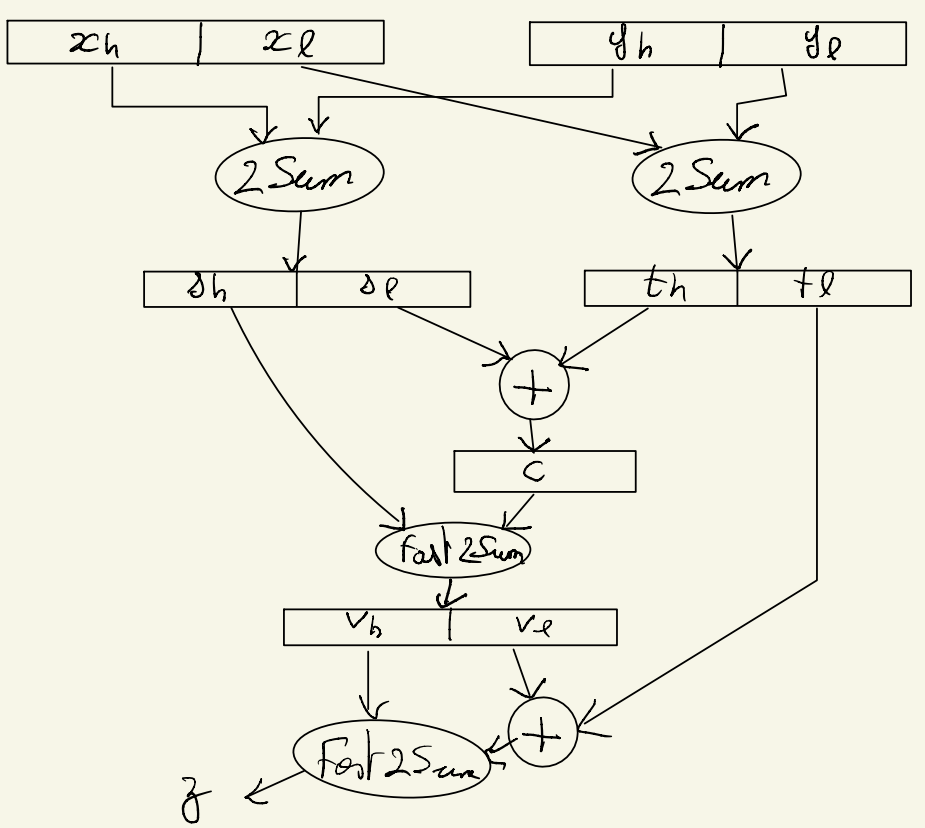
$$\frac{2u^2}{1 + 3u - 2u^2}.$$

DW+DW : “version précise”

Somme de deux nombres DW. Il existe un algorithme “quick & dirty” mais son erreur relative est non bornée.

DWPlusDW

1: $(s_h, s_\ell) \leftarrow \mathrm{2Sum}(x_h, y_h)$
2: $(t_h, t_\ell) \leftarrow \mathrm{2Sum}(x_\ell, y_\ell)$
3: $c \leftarrow \mathrm{RN}(s_\ell + t_h)$
4: $(v_h, v_\ell) \leftarrow \mathrm{Fast2Sum}(s_h, c)$
5: $w \leftarrow \mathrm{RN}(t_\ell + v_\ell)$
6: $(z_h, z_\ell) \leftarrow \mathrm{Fast2Sum}(v_h, w)$
7: **return** (z_h, z_ℓ)



DW+DW : “version précise”

On trouve (la preuve est très fastidieuse) :

Théorème 6 (Joldes, M., Popescu 2017)

Si $p \geq 3$, l'erreur relative de DWPlusDW is majorée par

$$\frac{3u^2}{1-4u} = 3u^2 + 12u^3 + 48u^4 + \cdots, \tag{1}$$

DW+DW : “version précise”

Donc le théorème donne une borne d'erreur $3u^2/(1-4u) \simeq 3u^2 \ldots$
Ce théorème a une histoire intéressante :

- les auteurs du papier où l'algorithme a été publié annonçaient sans preuve une borne d'erreur $2u^2$ (en double précision) ;
- en essayant sans succès de prouver cette borne, on trouve un exemple où l'erreur est $\approx 2.25u^2$;
- on a finalement prouvé le théorème (avec la borne $3u^2/(1-4u)$), et on a voulu le formaliser en Coq ;
- ceci nous a conduit à trouver une (petite) erreur dans notre preuve. . .

DW+DW : “version précise”

- heureusement l'erreur a vite été corrigée ;
- cependant, le gap entre la plus grande erreur trouvée ($\approx 2.25u^2$) et la borne ($\approx 3u^2$) nous chagrinait, donc on a passé des mois à essayer d'améliorer le théorème...
- mais bien sûr c'était impossible : c'était sur l'exemple de mauvais cas qu'il aurait fallu passer du temps !
- on a finalement trouvé qu'avec

$$\begin{aligned} x_h &= 1 \\ x_\ell &= u - u^2 \\ y_h &= -\tfrac{1}{2} + \tfrac{u}{2} \\ y_\ell &= -\tfrac{u^2}{2} + u^3. \end{aligned}$$

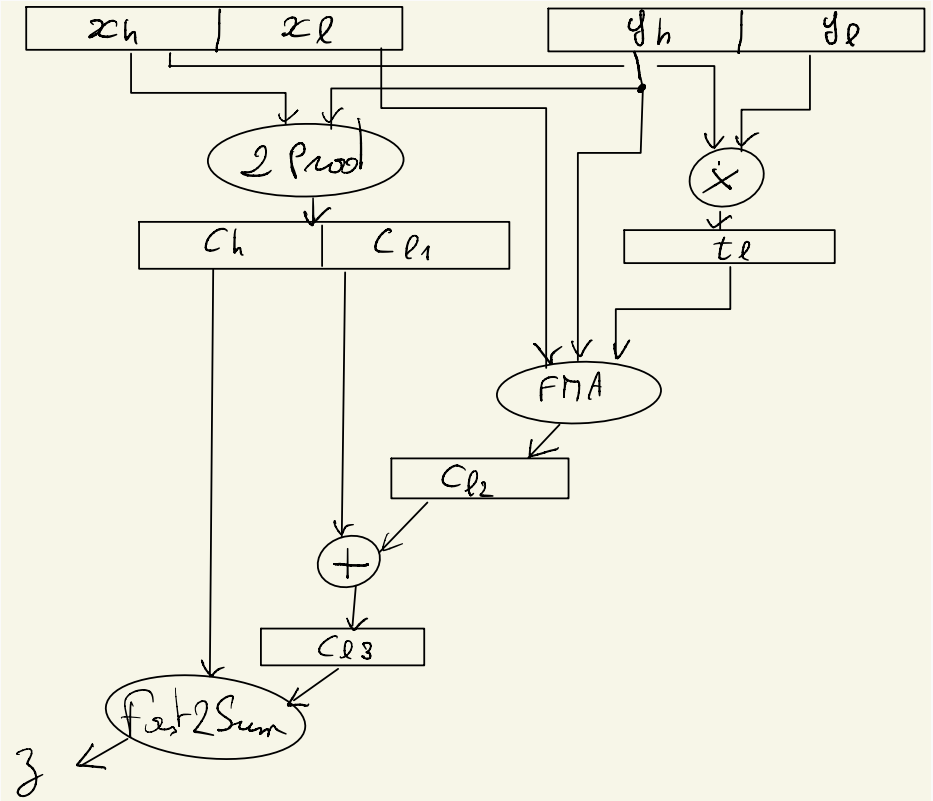
l'erreur relative commise est $\frac{3u^2-2u^3}{1+3u-3u^2+2u^3}$. Avec $p = 53$ (binary64/double précision), donne une erreur $2.99999999999999877875\cdots \times u^2$.

DW $\times$ DW

- Produit $z = (z_h, z_\ell)$ de deux nombres DW $x = (x_h, x_\ell)$ et $y = (y_h, y_\ell)$;
- plusieurs algorithmes $\rightarrow$ compromis vitesse/précision. On donne juste l'un d'eux.

DWTimesDW

1: $(c_h, c_{\ell 1}) \leftarrow 2\mathrm{Prod}(x_h, y_h)$
2: $t_\ell \leftarrow \mathrm{RN}(x_h \cdot y_\ell)$
3: $c_{\ell 2} \leftarrow \mathrm{RN}(t_\ell + x_\ell y_h)$
4: $c_{\ell 3} \leftarrow \mathrm{RN}(c_{\ell 1} + c_{\ell 2})$
5: $(z_h, z_\ell) \leftarrow \mathrm{Fast2Sum}(c_h, c_{\ell 3})$
6: **return** (z_h, z_ℓ)



Théorème 7 (M., Rideau 2022)

Si $p \geq 5$, l'erreur relative de DWTimesDW est majorée par

$$\frac{5u^2}{(1+u)^2} < 5u^2.$$

et ce théorème aussi a une histoire intéressante !

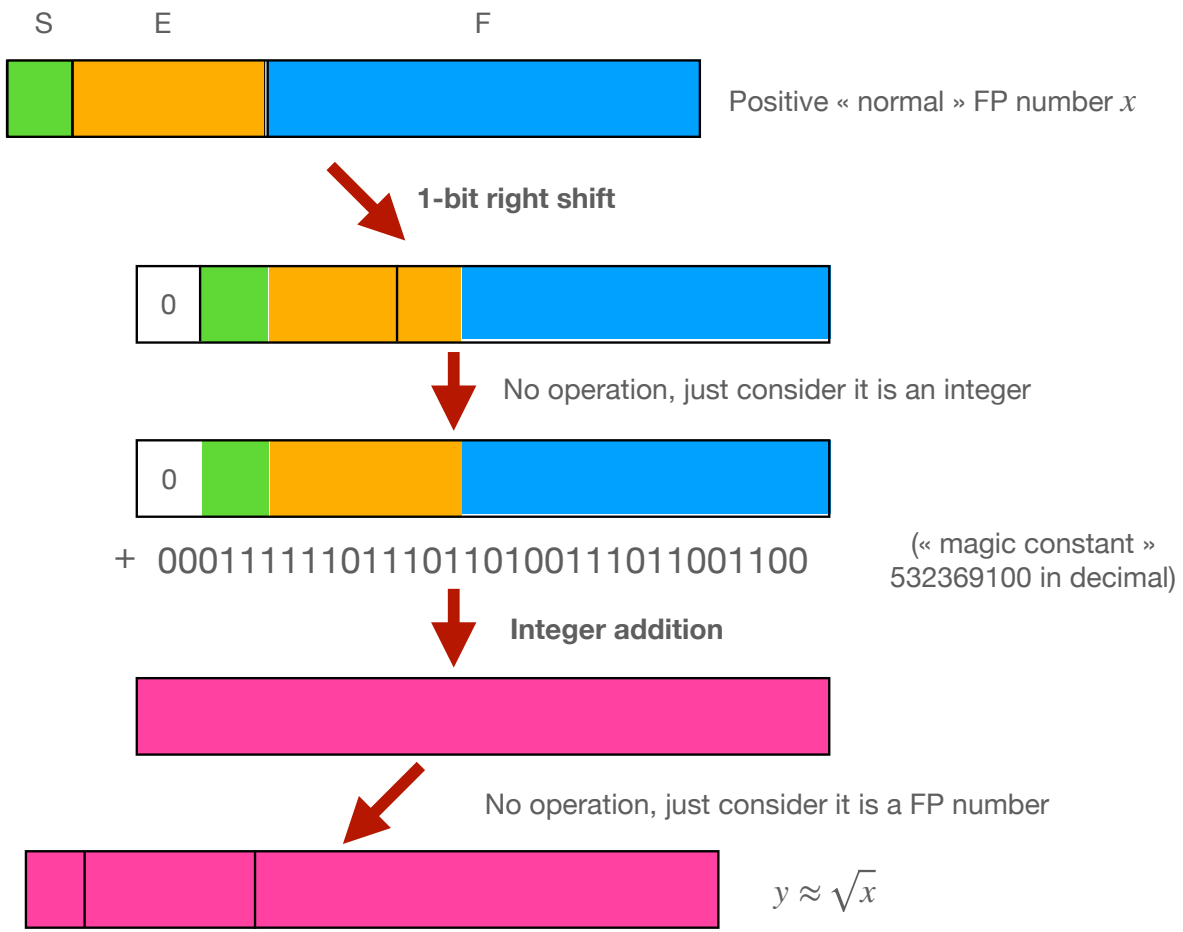
- borne initiale $6u^2$;
- à nouveau, formalisation en Coq. . . et il s'avéra que la preuve était basée sur un lemme faux.

DW $\times$ DW

- après quelques nuits de mauvais sommeil, solution se passant du lemme. . . qui a amélioré la borne !
- pas de preuve d'optimalité asymptotique, mais on a des exemples avec erreur $> 4.98u^2$;
- sans l'erreur initiale on n'aurait jamais trouvé la meilleure borne.

Conclusion : cette classe d'algorithmes gagne beaucoup à être prouvés formellement.

Et si ce que je veux c'est calculer très vite et comme un cochon ?



A similar trick first appears in
The game Quake III Arena

Techniques «bit-manipulation»

- jeu quake III, 1999 ;
- (très) faible précision, très rapide, software ;
- représentation Binary32 (= simple précision) de x :

  - signe S_x : 1 bit, exposant «biaisé» E_x : 8 bits, «fraction» F_x : 23 bits, t.q.

$$x = (-1)^{S_x} \cdot 2^{E_x - 127} \cdot \left(1 + 2^{-23} \cdot F_x\right).$$

  - la même chaîne binaire si on l'interprète comme un entier en complément à 2, représente le nombre

$$I_x = (1 - 2S_x) \cdot 2^{31} + \left(2^{23} \cdot E_x + F_x\right).$$

Techniques «bit-manipulation» pour $\sqrt{x}$

Rappel :

$$x = (-1)^{S_x} \cdot 2^{E_x - 127} \cdot (1 + 2^{-23} \cdot F_x) = (-1)^{S_x} \cdot 2^{e_x} \cdot (1 + f_x).$$

- Si $e_x = E_x - 127$ est pair (i.e., E_x est impair), on utilise :

$$\sqrt{(1 + f_x) \cdot 2^{e_x}} \approx \left(1 + \frac{f_x}{2}\right) \cdot 2^{e_x/2}, \tag{2}$$

- si e_x est impair (i.e., E_x est pair), on utilise :

$$\begin{aligned} \sqrt{(1 + f_x) \cdot 2^{e_x}} &= \sqrt{4 + \epsilon_x} \cdot 2^{\frac{e_x - 1}{2}} \\ &\approx \left(2 + \tfrac{\epsilon_x}{4}\right) \cdot 2^{\frac{e_x - 1}{2}} \\ &= \left(\tfrac{3}{2} + \tfrac{f_x}{2}\right) \cdot 2^{\frac{e_x - 1}{2}}, \end{aligned} \tag{3}$$

(Série de Taylor pour $\sqrt{4 + \epsilon_x}$ en $\epsilon_x = 0$, où $\epsilon_x = 2f_x - 2$)

Techniques «bit-manipulation» pour $\sqrt{x}$

$$x = (-1)^{S_x} \cdot 2^{E_x - 127} \cdot \left(1 + 2^{-23} \cdot F_x\right) = (-1)^{S_x} \cdot 2^{e_x} \cdot (1 + f_x).$$

- E_x impair $\rightarrow \left(1 + \frac{f_x}{2}\right) \cdot 2^{\frac{e_x}{2}}$,

$$(1 + F_y \cdot 2^{-23}) \cdot 2^{E_y - 127} \approx (1 + F_x \cdot 2^{-24}) \cdot 2^{\frac{E_x - 127}{2}}$$
$$\Rightarrow E_y = \frac{E_x + 127}{2} \text{ and } F_y = \left\lfloor \frac{F_x}{2} \right\rfloor$$

- E_x pair $\rightarrow \left(\frac{3}{2} + \frac{f_x}{2}\right) \cdot 2^{\frac{e_x - 1}{2}}$.

$$(1 + F_y \cdot 2^{-23}) \cdot 2^{E_y - 127} \approx \left(\tfrac{3}{2} + F_x \cdot 2^{-24}\right) \cdot 2^{\frac{E_x - 128}{2}}$$
$$\Rightarrow E_y = \frac{E_x + 127}{2} - \frac{1}{2} \text{ and } F_y = 2^{22} + \left\lfloor \frac{F_x}{2} \right\rfloor$$

Dans les 2 cas :

$$I_y = \left\lfloor \frac{I_x}{2} \right\rfloor + 127 \cdot 2^{22}$$

Techniques «bit-manipulation» pour $\sqrt{x}$

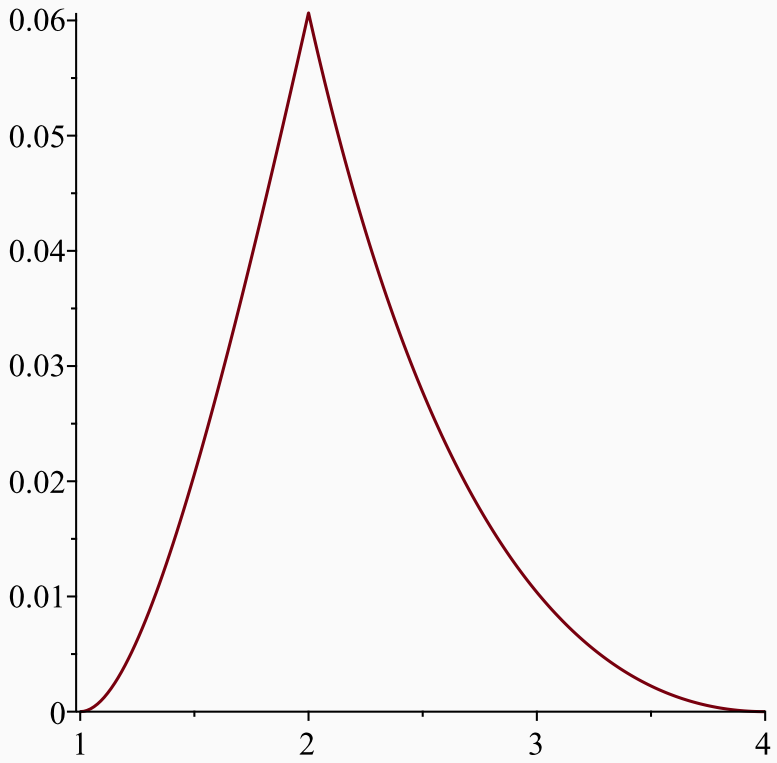


Figure 1 – $(\text{approx} - \sqrt{x})/\sqrt{x}$.

- rapide mais imprécis ;
- toujours $\geq \sqrt{x} \rightarrow$ remplacer $127 \cdot 2^{22}$ par une plus petite valeur ?

$\sqrt{x}$ avec une meilleure «constante magique»

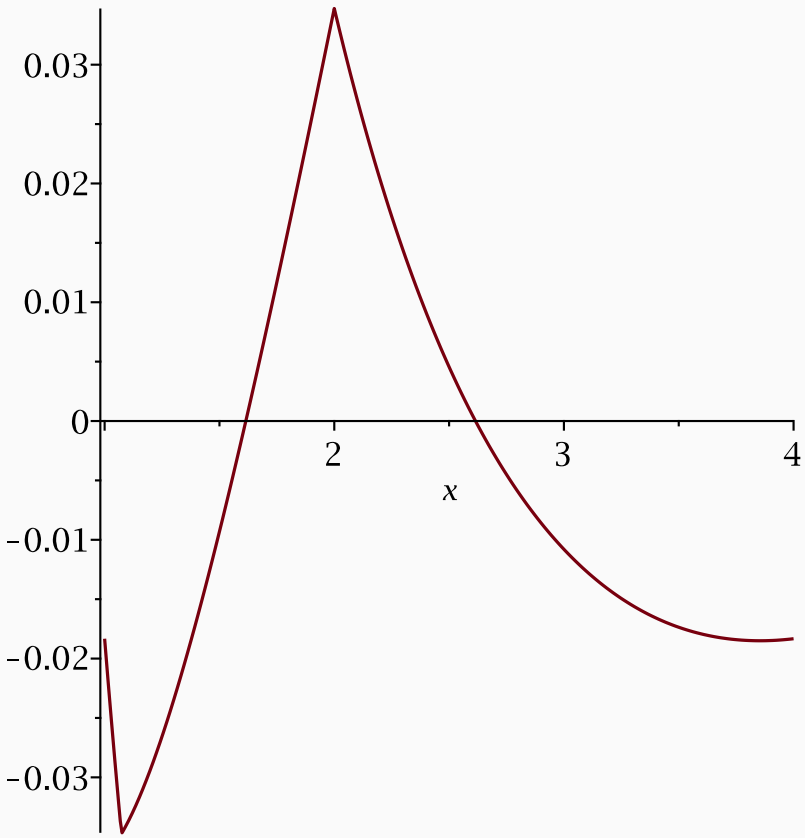


Figure 2 – (approx $-\sqrt{x}$)$/\sqrt{x}$ avec $127 \cdot 2^{22} = 532676608$ remplacé par 532369100.