AIgorithmes de division pour microprocesseurs : illustration à l'aide du " Bug " du Pentium

Jean-Michel Muller

Algorithmes de division pour microprocesseurs : illustration à l'aide du « Bug » du Pentium

Jean-Michel Muller

CNRS — Laboratoire LIP
Ecole normale supérieure de Lyon
46, allée d'Italie
69364 Lyon cedex 07

RÉSUMÉ. *Dans cet article, on présente rapidement les algorithmes de division les plus employés sur les microprocesseurs actuels, et tout particulièrement l'algorithme utilisé par le Pentium d'Intel. On analyse en détail le problème qui a causé l'imprécision de la division du Pentium (qui a fait couler, il y a quelques mois, beaucoup de « salive électronique »), afin de permettre à l'utilisateur potentiel d'estimer, en fonction de ses applications, quels sont les problèmes qu'il peut encourir.*

ABSTRACT. *This paper briefly presents the division algorithms that are the most frequently implemented on microprocessors, especially the algorithm used by Intel's Pentium. The FDIV flaw of the Pentium is analysed, in order to help the users to estimate, depending on their applications, what are the potential risks.*

MOTS-CLÉS : *division, arithmétique « virgule flottante ».*
KEY WORDS : *division, floating-point arithmetic.*

1. Introduction

En octobre 1994, Thomas Nicely, du *Lynchburg College*, en Virginie (USA), a constaté que son ordinateur, dont le micro-processeur était un Pentium d'Intel, donnait un résultat erroné lors du calcul de

$$\frac{1.0}{824633702441}$$

opération	résultat Pentium	bon résultat
8391667/12582905	0.666869455...	0.666910145...
12845015/11010020	1.166619406...	1.16666591...
14909407/11010030	1.35411938...	1.35416588...

Table 1. *Quelques exemples découverts par V. Pratt*

il a averti la compagnie Intel, ainsi que d'autres utilisateurs par le biais des *newsgroups*. L'erreur constatée par T. Nicely était assez faible (le résultat avait environ 11 chiffres significatifs). Cependant, une fois l'alerte donnée, plusieurs possesseurs de machines à base de Pentium ont procédé à des tests intensifs, et des erreurs bien plus importantes ont été détectées. Le «pire cas» a été découvert par Tim Coe, de la compagnie *Vitesse Semiconductors* : le calcul de

$$\frac{4195835.0}{3145727.0}$$

donne 1.333739068902... sur un Pentium (les chiffres non significatifs sont soulignés), alors que le résultat correct est 1.333820449136... Tim Coe a donné de nombreux exemples. Il fut également l'un des premiers à cerner l'origine du problème. Parmi les exemples qu'il a donnés, on peut citer le petit programme suivant :

```
B = 4.1 - 1.1
A = 699306 xB
Q = A/B
```

qui donne 699263.3 au lieu de 699306. Attention, dans ce programme, la ligne « `B = 4.1 - 1.1` » ne peut pas être remplacée par « `B = 3` » (ceci provient du fait que, comme ni 4.1 ni 1.1 ne s'écrivent exactement en base 2, la soustraction $4.1 - 1.1$ donne un résultat très légèrement inférieur à 3, et c'est pour cette valeur particulière du dénominateur qu'apparaît une erreur de division).

Vaughan Pratt, de l'université de Stanford, s'est livré à une recherche exhaustive en simple précision. La table 1 donne quelques uns des exemples découverts.

Pratt a également mis en évidence le fait que beaucoup de problèmes apparaissent avec des opérandes proches de petits entiers. Par exemple le calcul de $\frac{4.999999}{14.999999}$ donne 0.33332922 au lieu de 0.33333329.

Après la publication de ces quelques exemples, une véritable « tempête électronique » s'est levée sur le réseau internet. On a vu beaucoup d'utilisateurs perdre toute mesure. Dans cet article, on analyse en détail le problème de la division du Pentium, afin que chacun puisse se faire son opinion, en fonction de ses applications et de ses besoins en précision.

2. Comment fait-on une division ?

Tout d'abord, lorsque les nombres manipulés sont des nombres *virgule flottante* (c'est-à-dire — sur une machine utilisant la base 2, ce qui sera le seul cas traité ici — des nombres écrits sous la forme $\pm m \times 2^e$, où m, appelé *mantisse* est un nombre réel compris entre 1 et 2, et où e, appelé *exposant* est un entier relatif), on doit en pratique diviser leurs mantisses et soustraire leurs exposants (après ceci, une étape de renormalisation est parfois nécessaire, mais elle est élémentaire et ne sera pas considérée ici). Notre problème est donc ramené au problème de la division de mantisses, c'est-à-dire de nombres compris entre 1 et 2. En particulier, toutes les erreurs rencontrées lors de divisions effectuées sur le Pentium seront invariantes par multiplication par une puissance positive ou négative de 2 : si le calcul de x/y est erroné, le calcul de $(2^5x)/(2^{-3}y)$ le sera également.
Les algorithmes de division employés par les constructeurs de machines se partagent en deux catégories :

— Il y a tout d'abord des algorithmes basés sur l'itération de Newton :

$$r_{n+1} = r_n \times (2 - a r_n)$$

qui, pour peu que r_0 soit bien choisi, converge vers $1/a$; ou sur l'itération de Goldschmidt [AND 67] :

$$\begin{aligned} x^{(0)} &= x \\ \epsilon^{(0)} &= 1 - y \\ x^{(i+1)} &= x^{(i)} \times \left(1 + \epsilon^{(i)}\right) \\ \epsilon^{(i+1)} &= \left(\epsilon^{(i)}\right)^2 \end{aligned}$$

pour laquelle $x^{(i)}$ converge vers x/y pour peu que y soit compris entre $1/2$ et 1. Ces deux itérations ne sont d'ailleurs pas très différentes : on passe de l'une à l'autre en posant $\epsilon_n = 1 - a r_n$ [FLY 70]. Elles nécessitent l'emploi d'un multiplicateur, mais leur convergence est quadratique, donc très rapide. Elles sont surtout utilisés sur les supercalculateurs.

— Il y a ensuite les algorithmes basés sur des additions et des décalages [ROB 58, ERC 94], qui sont de la famille de l'algorithme que l'on apprend à l'école. C'est cette dernière classe que nous allons étudier ici.

Pour étudier les algorithmes basés sur des additions et des décalages, prenons un exemple (illustré par la figure 1) : on cherche à diviser $x = 4125260$ par $y = 521274$ en utilisant la méthode usuelle.

Notons $x^{(0)} = x$. Comme $x^{(0)}$ est inférieur à y, on va ajouter un zéro à la fin de son écriture décimale (ce qui revient à le multiplier par 10), puis on va chercher le plus grand chiffre q_1 (pris entre 0 et 9) tel que $x^{(1)} = x^{(0)} - q_1 y$ soit compris entre 0 et y. La valeur q_1 constituera le premier chiffre fractionnaire du quotient. On procède de même par la suite : on construit une suite de la forme $x^{(i+1)} = 10x^{(i)} - q_{i+1}y$ où q_{i+1} est choisi de sorte que l'on ait toujours $0 \leq x^{(i+1)} < y$. Le résultat est alors le nombre $0.q_1q_2q_3q_4\ldots$

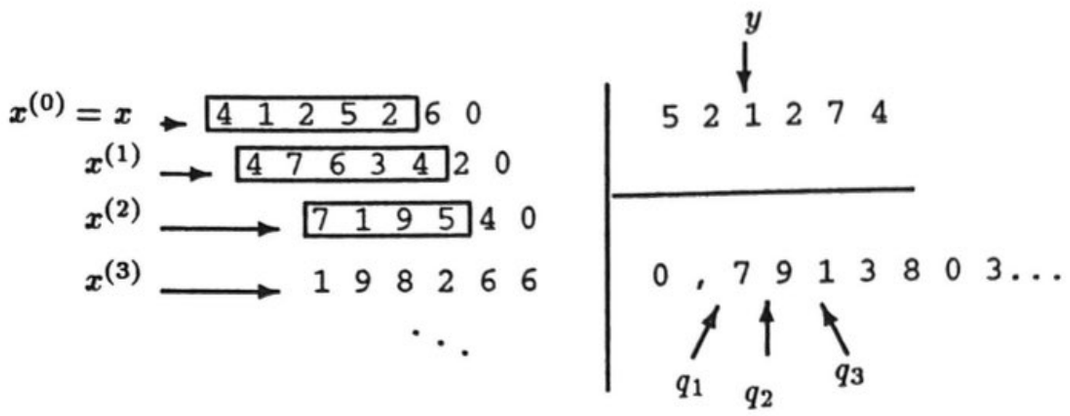

Figure 1. *Un exemple de division en base 10*

Sur une machine binaire, l'adaptation « naturelle » de ce procédé consiste à construire une itération

$$\begin{cases} x^{(0)} & = & x \\ x^{(i+1)} & = & 2x^{(i)} - q_{i+1}y \end{cases}$$

où les valeurs q_{i+1} sont choisies (égales à 0 ou 1) de sorte que $x^{(i+1)}$ reste compris entre 0 et y (au sens large : il n'est pas nécessaire d'interdire que $x^{(i+1)}$ puisse être égal à y pour garantir la convergence). On obtient ainsi un algorithme assez lent, car il faut effectuer n itérations pour obtenir n bits du quotient. Une autre solution est d'effectuer une itération de division dans une base qui est une puissance de 2 : en base 2^k, on obtiendra k bits du quotient à chaque itération, mais les itérations seront plus complexes. Dans le cas de l'algorithme de division du Pentium, la base choisie est 4 : on effectue donc une itération de la forme

$$\begin{cases} x^{(0)} & = & x \\ x^{(i+1)} & = & 4x^{(i)} - q_{i+1}y \end{cases} \qquad [1]$$

Il suffit alors de $n/2$ itérations pour obtenir n bits du quotient. Il est important de comprendre que ceci ne signifie pas forcément que les valeurs $x^{(i)}$ seront écrites en base 4. En fait, dans l'algorithme utilisé par le Pentium, trois systèmes de représentation des nombres sont utilisés conjointement : le diviseur y est représenté en base 2 « classique », les chiffres du quotient seront représentés en base 4 dans un système *redondant* d'écriture des nombres, et les valeurs $x^{(i)}$ sont représentées en base 2 dans le système « carry-save ». Tout ceci est expliqué dans les paragraphes suivants. Nous allons maintenant étudier plus en détail l'itération [1], afin de cerner les problèmes liés à son implantation en machine.

3. La division « SRT » en base 4

Nous allons maintenant rapidement présenter un algorithme permettant d'implanter de manière efficace l'itération [1]. Cet algorithme est une variante d'un algorithme

découvert indépendamment en 1958 par Sweeney, Robertson et Tocher [ROB 58, TOC 58], d'où le nom d'algorithme « SRT » qui lui est généralement donné. De nombreuses variantes de cet algorithme ont été proposées et implantées depuis [BOS 87, MCQ 93, COR 94, MON 92, EIS 93, ERC 85, ERC 83, FAN 89, FEN 95, MAN 93, SVO 63, TUN 68], et le lecteur pourra en trouver une présentation détaillée dans un récent livre d'Ercegovac et Lang [ERC 94].

3.1. *Le système « carry-save »*

Tout d'abord, afin d'accélérer la soustraction $4x^{(i)} - q_{i+1}y$ qui apparaît dans [1], les calculs sont faits en utilisant un système redondant d'écriture des nombres appelé « carry-save » [HWA 79, MUL 89, KOR 93]. Dans un tel système, un nombre est représenté en base deux avec des chiffres qui valent 0, 1 ou 2. Le chiffre c_i est représenté par 2 bits c_i^0 et c_i^1 dont la somme vaut c_i. L'intérêt d'un tel système est que l'on peut très rapidement additionner un nombre écrit dans le système binaire usuel à un nombre écrit en « carry-save », et obtenir le résultat en « carry-save ». La figure 2 montre comment s'effectue une telle addition : si on calcule $s = a + b$, où a est écrit en carry-save et b en notation binaire usuelle, on additionne sans propager de retenue les trois nombres suivants :

— le nombre constitué des bits a_i^0
— le nombre constitué des bits a_i^1
— le nombre b

les bits de « somme » constitueront les bits s_i^0, et les bits de « retenue », au lieu d'être propagés, seront conservés (d'où le nom de « carry-save ») et constitueront les bits s_i^1. Plus formellement, on calcule :

$$\begin{aligned} s_i^0 &= a_i^0 \oplus a_i^1 \oplus b_i \\ s_i^1 &= a_i^0 a_i^1 + a_i^0 b_i + a_i^1 b_i \end{aligned}$$

où « $\oplus$ » désigne le « ou exclusif ».

La figure 3 présente un additionneur séquentiel classique et un additionneur « carry-save ». L'addition carry-save est utilisée dans presque tous les circuits multiplicateurs. Les cellules rectangulaires sont des cellules « Full Adder ».

La conversion en écriture usuelle d'un nombre a représenté en « carry-save » se fait simplement en additionnant (avec propagation de retenue) le nombre constitué des chiffres a_i^0 au nombre constitué des chiffres a_i^1. Cette conversion peut être trop coûteuse dans certains cas : on peut alors préférer obtenir une écriture binaire d'une *valeur approchée* du nombre à convertir, en ne faisant porter l'addition que sur les premiers chiffres de poids fort. Ceci est illustré figure 4.

3.2. *Le choix des chiffres du quotient*

Nous venons de voir que grâce à l'utilisation de la notation « carry-save » pour représenter les itérés successifs $x^{(i)}$, la soustraction apparaissant dans l'itération

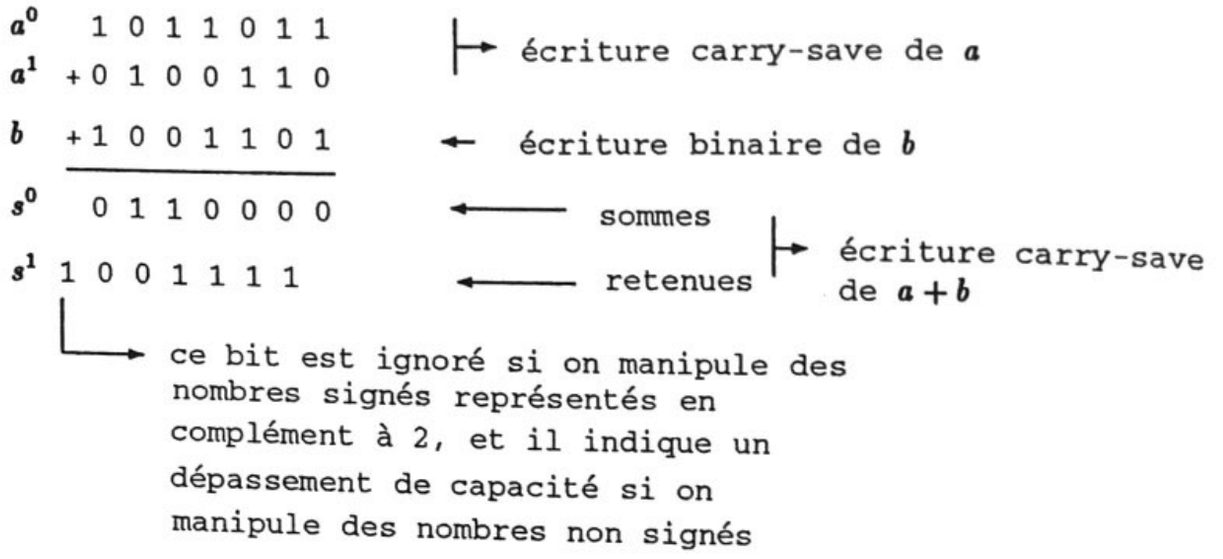

Figure 2. *Un exemple d'addition « carry-save »*

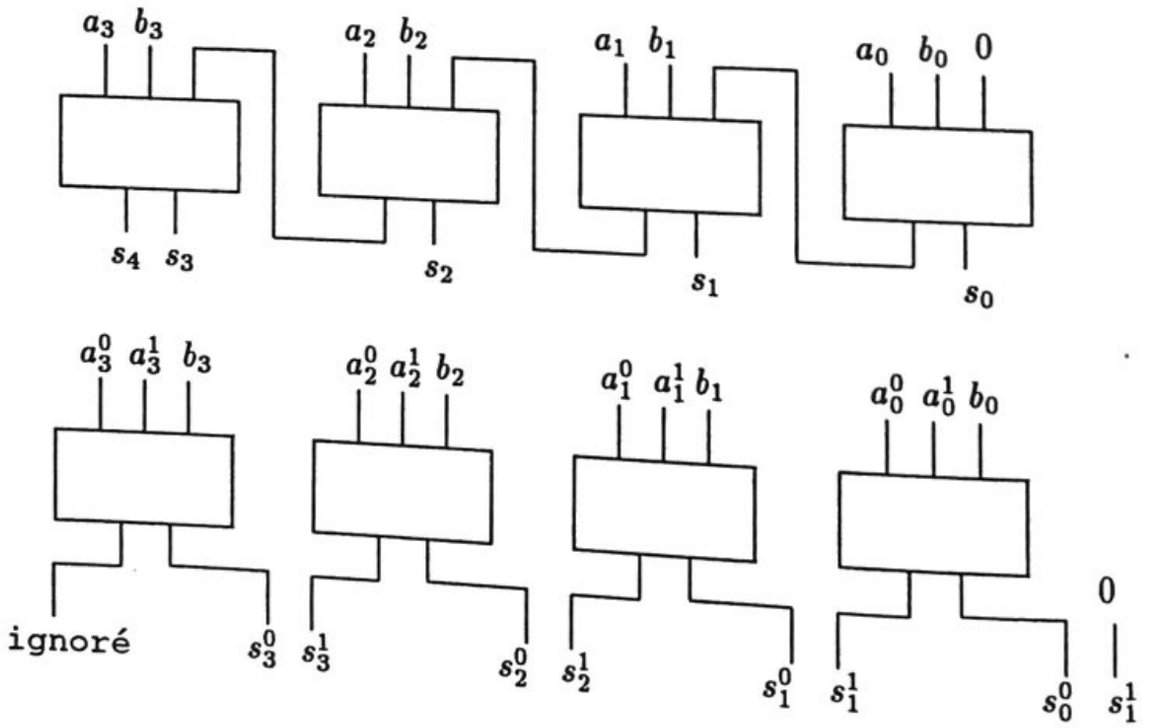

Figure 3. *Un additionneur séquentiel simple et un additionneur « carry-save ». Les deux additionneurs calculent $s = a + b$. Dans le premier cas, a, b et s sont représentés dans le système usuel — non redondant —, dans le deuxième cas, seul b est représenté dans le système usuel, et a et s sont en notation «carry-save». L'addition «carry-save» s'effectue très rapidement (le temps de traversée par le signal d'une cellule élémentaire d'addition), sans aucune propagation de retenue.*

```
  1 0.1 1 0 1 0 1 0 1 1 1 0 1 1 0 1 1      conversion exacte
+ 1 1.0 0 1 0 0 1 1 0 1 0 0 0 1 1 0 1      utilisant une addition
-------------------------------------
  0 1.1 1 1 1 1 1 1 0 0 0 1 1 0 1 0 0 0

  1 0.1 1 0 1 0 1 0 1 1 1 0 1 1 0 1 1      conversion approchée
+ 1 1.0 0 1 0 0 1 1 0 1 0 0 0 1 1 0 1      utilisant une
-------------------------------------      "petite addition"
1 [0 1.1 1 1 1 1 0]                        (ici sur 7 bits seulement)
```

Figure 4. *Conversion «approchée» d'un nombre carry-save*

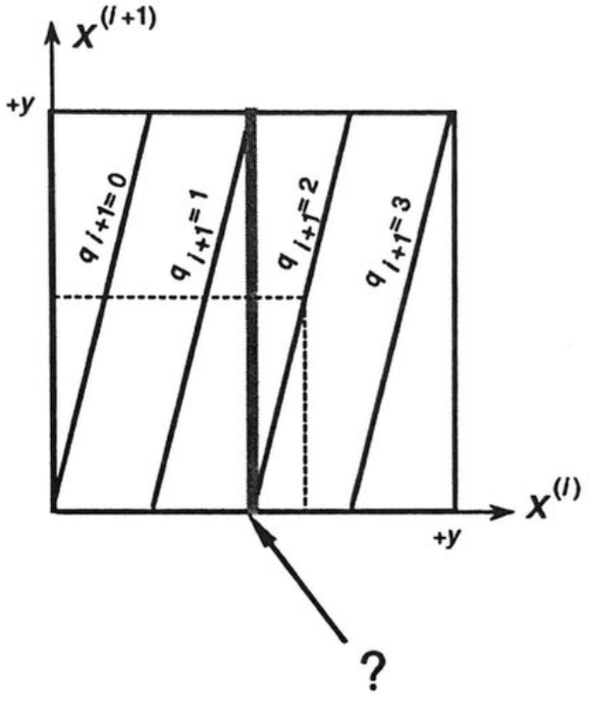

Figure 5. *Diagramme de Robertson de la division usuelle en base 4*

$x^{(i+1)} = 4x^{(i)} - q_{i+1}y$ peut s'effectuer très rapidement. Le problème restant est donc, à chaque itération, le choix de q_{i+1}.

Si on fixe la valeur de q_{i+1}, la fonction donnant $x^{(i+1)}$ en fonction de $x^{(i)}$ est affine. On a tracé figure 5 cette fonction pour q_{i+1} égal à 0, 1, 2 et 3. Le diagramme obtenu est appelé *Diagramme de Robertson* [ROB 58]. On voit immédiatement que, sauf pour les 3 valeurs particulières $x^{(i)} = y/4, y/2$ et $3y/4$, le choix de q_{i+1} qui garantit que $x^{(i+1)}$ sera bien compris (au sens large) entre 0 et y est *unique*. Ceci pose problème : il faut dans certains cas (lorsque $x^{(i)}$ sera au voisinage de ces valeurs particulières) tenir compte de *tous* les chiffres de $x^{(i)}$ pour décider de la valeur de q_{i+1} qui convient, ce qui rend une tabulation du choix de q_{i+1} impossible (par exemple, si $x^{(i)}$ se trouve dans la zone indiqué en grisé à la figure 5, il est impossible de savoir s'il faut prendre q_{i+1} égal à 1 ou à 2 au vu d'un petit nombre de chiffres de $x^{(i)}$ seulement : si $x^{(i)}$ est très légèrement inférieur à $y/2$ il faut choisir q_{i+1} égal à 1, tandis que si $x^{(i)}$ est très légèrement supérieur à $y/2$ il faut choisir q_{i+1} égal à 2).

Le problème est résolu en ajoutant un peu de « redondance » dans le choix des chiffres du quotient, c'est-à-dire en représentant ce quotient dans un *système redondant d'écriture des nombres* [AVI 61] : on utilisera toujours la base 4, mais avec les chiffres -2, -1, 0, 1 et 2. On peut facilement montrer que tout nombre inférieur ou égal à $2/3$ peut s'écrire en base 4 sous la forme $0.q_1q_2q_3\ldots$, avec des chiffres q_i compris entre -2 et 2 — le lecteur familiarisé avec les techniques de multiplication aura reconnu ici le recodage de Booth —. Par exemple (si on note $\bar{2} = -2$ et $\bar{1} = -1$), 1/3 s'écrit $0.1111111\ldots$, mais il s'écrit également $0.2\bar{2}\bar{2}\bar{2}\bar{2}\bar{2}\bar{2}\bar{2}\ldots$ ou $0.11\bar{2}\bar{2}\bar{2}\bar{2}\bar{2}\ldots$ Cette multiplicité des écritures possibles va nous permettre de choisir à chaque itération une valeur de q_{i+1} qui convienne au vu d'un petit nombre de chiffres seulement de $x^{(i)}$ et de y. Ce choix pourra alors être tabulé. Choisir q_{i+1} dans $\{-2,-1,0,1,2\}$ présente également un autre avantage : le calcul de $q_{i+1} \times y$ devient alors trivial.
La figure 6 représente le diagramme de Robertson de la division en base 4 avec des chiffres du quotient égaux à -2, -1, 0, 1 ou 2. La condition à respecter à chaque itération n'est plus « $0 \leq x^{(i)} < y$ », mais « $\left|x^{(i)}\right| \leq 2y/3$ ». Dans la zone représentée en grisé sur cette figure, le choix $q_{i+1} = 1$ et le choix $q_{i+1} = 2$ conviennent. L'algorithme obtenu est la variante à base 4 de l'algorithme « SRT ». On voit figure 6 les choix possibles suivant la valeur de $x^{(i)}$:

— si $-2y/3 \leq x^{(i)} < -5y/12$ on doit prendre $q_{i+1} = -2$;
— si $-5y/12 \leq x^{(i)} \leq -y/3$ on doit prendre $q_{i+1} = -2$ ou -1 ;
— si $-y/3 < x^{(i)} < -y/6$ on doit prendre $q_{i+1} = -1$;
— si $-y/6 \leq x^{(i)} \leq -y/12$ on doit prendre $q_{i+1} = -1$ ou 0 ;
— si $-y/12 < x^{(i)} < +y/12$ on doit prendre $q_{i+1} = 0$;
— si $+y/12 \leq x^{(i)} \leq +y/6$ on doit prendre $q_{i+1} = 0$ ou 1 ;
— si $+y/6 < x^{(i)} < y/3$ on doit prendre $q_{i+1} = 1$;
— si $+y/3 \leq x^{(i)} \leq +5y/12$ on doit prendre $q_{i+1} = 1$ ou 2 ;
— si $+5y/12 < x^{(i)} \leq +2y/3$ on doit prendre $q_{i+1} = 2$;
— les cas $x^{(i)} > +2y/3$ et $x^{(i)} < -2y/3$ sont impossibles par construction.

Ces divers choix sont illustrés par la figure 7, appelée *diagramme Reste-Diviseur* de l'algorithme. Ce diagramme a été publié par Intel dans un document électronique intitulé *Statistical Analysis of Floating Point Flaw in the Pentium* que l'on peut se procurer à l'adresse html `http://www.intel.com`. Dans ce diagramme, la « zone impossible » correspond aux valeurs de $x^{(i)}$ supérieures à $2y/3$, qui ne sont jamais atteintes par construction de l'algorithme. La conversion du quotient vers un système non redondant se fait de manière triviale : on soustrait le nombre constitué des valeurs absolues des chiffres négatifs au nombre constitué des chiffres positifs. Par exemple, la conversion de $1\bar{2}0\bar{1}1$ s'obtient en calculant $10001 - 02010$.

4. L'implantation du Pentium

Nous sommes maintenant à même de comprendre les détails de l'algorithme utilisé par le Pentium. L'algorithme utilisé est donc l'algorithme SRT, en base 4. Les restes

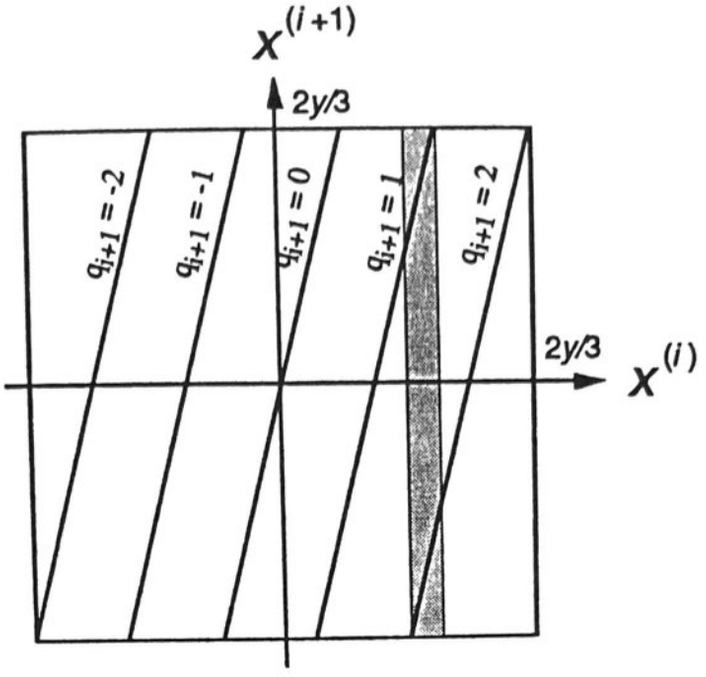

Figure 6. *Diagramme de Robertson de la division SRT en base 4*

partiels $x^{(i)}$ sont représentés en notation «carry-save», et le quotient est représenté en base 4, avec les chiffres -2, -1, 0, 1 et 2.

Le choix de q_{i+1} se fait à l'aide d'une table dont les entrées sont les 5 bits de poids forts de y, et 7 bits de $x^{(i)}$ obtenus par une conversion «approchée» de $x^{(i)}$, (voir figure 4), c'est-à-dire en ne propageant une retenue que sur les 7 positions les plus significatives de l'écriture *carry-save* de $x^{(i)}$. La figure 9 présente un agrandissement d'une zone du diagramme Reste-Diviseur. On voit sur cette figure le domaine où peut se trouver le point de coordonnées $(y, x^{(i)})$ sachant la valeur des 7 bits de $x^{(i)}$ et des 5 bits de y utilisés par la table.

On peut constater figure 8 que ces 7 bits de $x^{(i)}$ et ces 5 bits de y suffisent pour déterminer un chiffre correct du quotient, grâce au recouvrement des différentes zones de choix.

Dans le cas du Pentium, la table donnant q_{i+1} a été implantée à l'aide d'un PLA. Une erreur s'est glissée : 5 valeurs de la table sont fausses. Ces 5 valeurs sont indiquées figure 8, elles correspondent aux valeurs suivantes des 5 premiers bits de y : 1.0001, 1.0100, 1.0111, 1.1010 et 1.1101. Le parfait alignement (voir figure 8) des valeurs erronées laisse à penser que le problème ne résulte peut-être pas d'une mauvaise transcription des valeurs de la table : l'implanteur de l'algorithme a peut-être pensé que tous les points $(y, x^{(i)})$ correspondant à ces valeurs étaient dans la «zone impossible».

On peut voir figure 9 que pour une des valeurs, appelons-la y^*, du nombre constitué des 5 premiers bits de y posant problème, la probabilité d'erreur augmente lorsque la véritable valeur de y se rapproche de $y^* + 2^{-5}$. Ceci explique pourquoi les problèmes rencontrés en pratique interviennent lorsque y est constitué d'une des 5 valeurs «à risque» des 5 premiers bits suivie de plusieurs «1». *Une conséquence de ceci est que les valeurs du dénominateur pouvant conduire à une erreur sont une puissance de 2 (négative ou positive) multipliée par un nombre très légèrement inférieur à 3* (c'est-à-dire de la

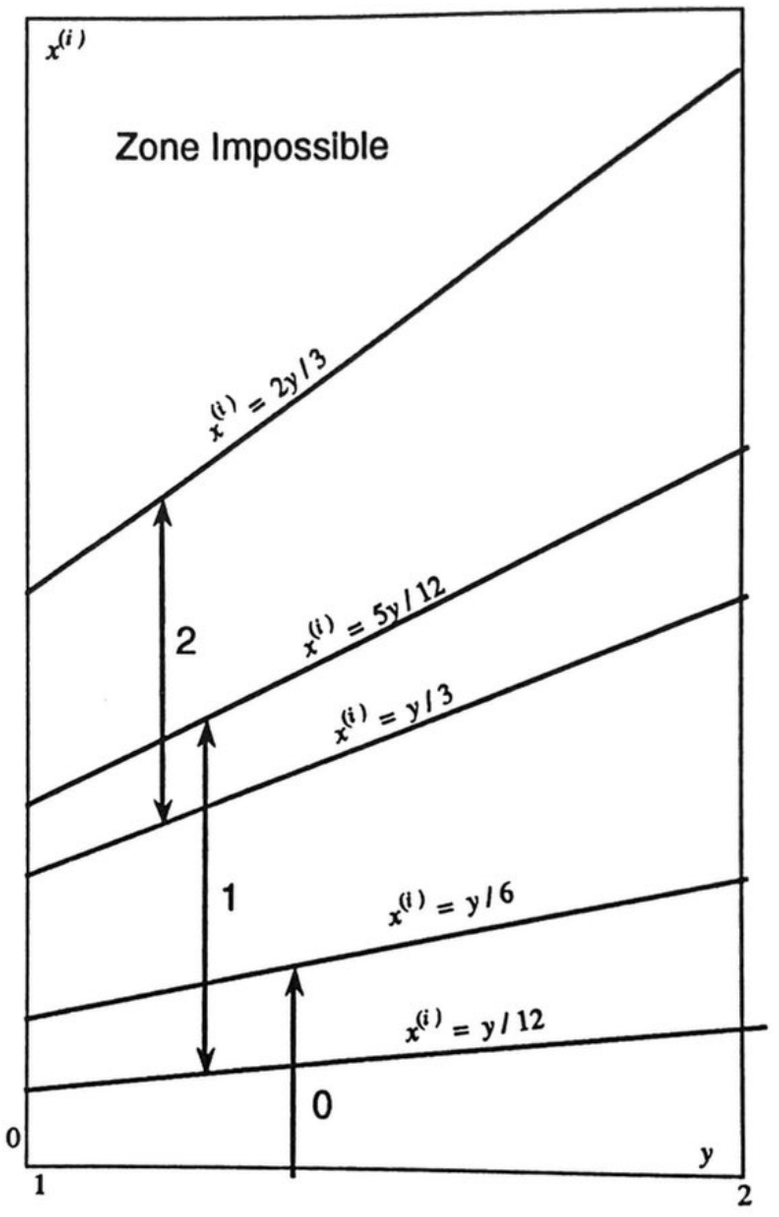

Figure 7. *Diagramme Reste-Diviseur de la division SRT en base 4*

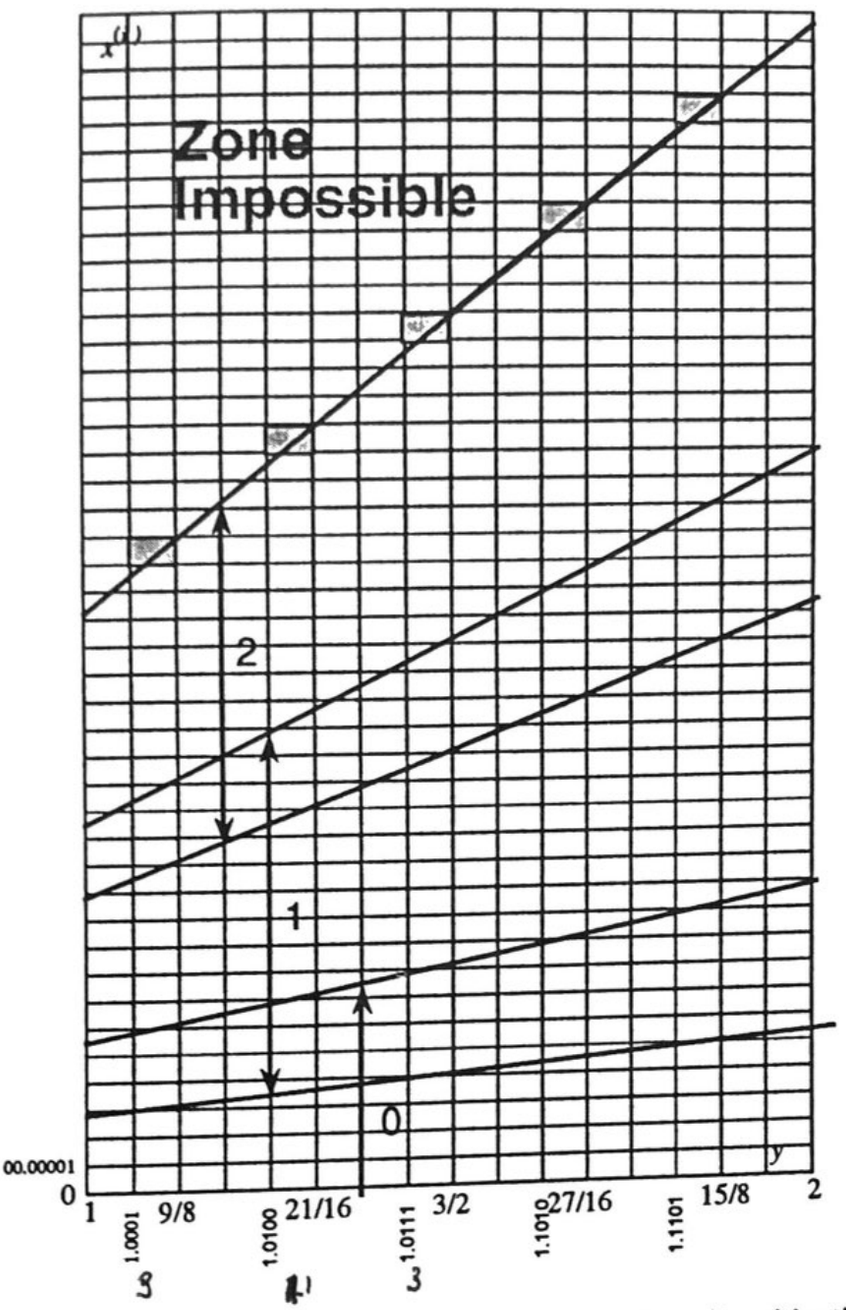

Figure 8. *Diagramme Reste-Diviseur faisant apparaître la «discrétisation» du choix et les 5 entrées erronées de la table. Seule la partie «* $x^{(i)} \geq 0$ *» du diagramme est présentée. Cette figure est tirée du document électronique d'Intel* Statistical Analysis of Floating Point Flaw in the Pentium

forme 1.0111111...xxx), *à 9* (c'est-à-dire de la forme 1.000111...xxx), *à 15* (c'est-à-dire de la forme 1.1101111...xxx), *à 21* (c'est-à-dire de la forme 1.0100111...xxx) *ou à 27* (c'est-à-dire de la forme 1.1010111...xxx). Dans l'exemple de T. Nicely présenté dans l'introduction :

$$\frac{1.0}{824633702441}$$

le dénominateur 824633702441 est égal à $2^{38} \times 2.99999993\ldots$, tandis que dans un des exemples de V. Pratt :

$$\frac{12845015}{11010020}$$

le dénominateur est égal à $2^{19} \times 20.999946\ldots$

5. Conséquences : fréquences et ordres de grandeur des erreurs

On va maintenant chercher à estimer quelles peuvent être les conséquences de l'erreur introduite dans la table sur des calculs. Tout d'abord, il a été observé par les premières personnes ayant détecté le problème (ce fut confirmé par les tests exhaustifs de Vaughan Pratt) que même dans les pires cas, l'erreur sur le résultat d'une division reste assez faible (en pratique, on a toujours au moins 3 à 4 chiffres significatifs en base 10). Ceci peut s'expliquer assez simplement : l'algorithme garantit que l'on aura toujours $\left|x^{(i)}\right| \leq 2y/3$, pour peu que $x^{(0)}$ satisfasse la même condition. Au départ, les opérandes x et y sont des *mantisses* de nombres virgule flottante, ils sont donc compris entre 1 et 2. Pour garantir la condition $\left|x^{(0)}\right| \leq 2y/3$ quel que soit y, il faut décaler x de deux positions vers la droite — c'est-à-dire le diviser par 4 — $x^{(0)}$ satisfait alors la relation :

$$\frac{1}{4} \leq \left|x^{(0)}\right| < \frac{1}{2}$$

Cette condition garantit que $x^{(i)}$ ne se trouvera pas dans la zone « à risques» (c'est-à-dire dans la zone $x^{(i)} \simeq 2y/3$) avant quelques itérations. En conséquence, les premières valeurs q_{i+1} trouvées seront correctes, ce qui explique que même lorsqu'on tombe sur une erreur, le résultat obtenu est tout de même proche du résultat exact.
Un autre phénomène explique la relativement basse fréquence d'apparition d'erreurs : la zone « à risques » est répulsive. En effet, si $x^{(i)}$ est proche de $2y/3$, alors $x^{(i+1)}$ en sera 4 fois plus éloigné. En effet, notons $\epsilon = x^{(i)} - 2y/3$. Si q_{i+1} est évalué sans erreur (c'est-à-dire ici s'il vaut 2), alors $x^{(i+1)} - 2y/3$ vaudra $4x^{(i)} - 2y - 2y/3 = 4\epsilon$. On ne reste donc jamais longtemps dans cette zone « à risques ».
La table 2 donne le nombre d'erreurs trouvées lors du test exhaustif en simple précision effectué par Vaughan Pratt, ainsi qu'une estimation (due également à Pratt) du nombre d'erreurs en double précision. Dans ce décompte seules les mantisses différentes sont comptées. Pour avoir par exemple le nombre effectif de couples de réels simple précision (x, y) conduisant à une erreur lors du calcul de x/y, il faut multiplier le nombre donné par la table par 2^{16}.

Ces chiffres, ainsi que d'autres donnés par Pratt, Coe et divers expérimentateurs, ont été à l'origine des estimations très variées de probabilité d'erreur qui ont circulé

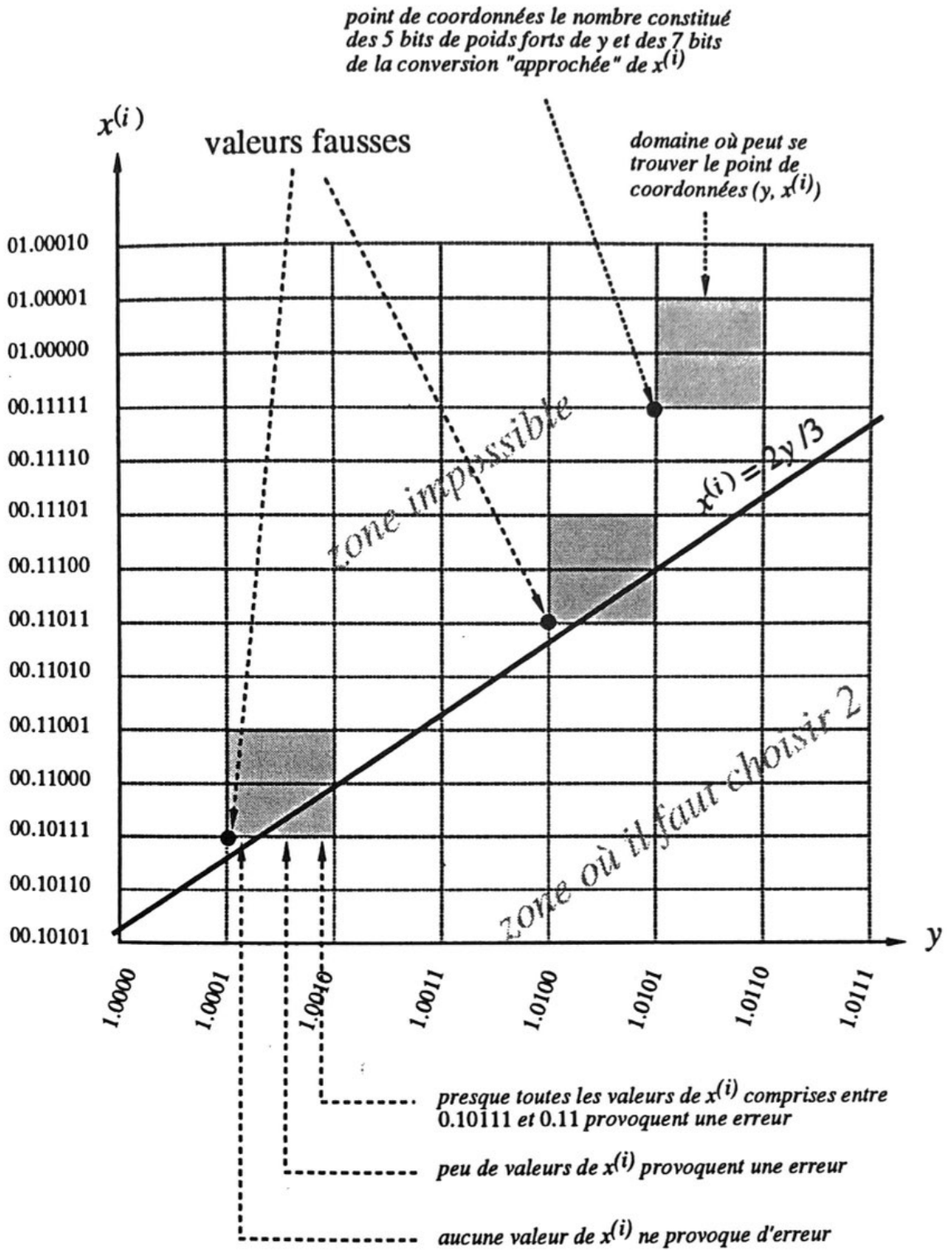

Figure 9. *Agrandissement d'une partie du diagramme Reste-Diviseur*

	nombre d'erreurs	nombre de valeurs	ratio
simple précision	1738	7.036×10^{13}	2.47×10^{-11}
double précision	2.266×10^{21}	2.028×10^{31}	1.11×10^{-10}

Table 2. *Nombre d'erreurs en simple précision et estimation du nombre d'erreurs en double précision par V. Pratt*

ces derniers temps (d'une erreur tous les 27000 ans à un erreur toutes les quelques minutes !). Si ces estimations sont très variées, c'est certes en partie parce que chacun a essayé dans cette affaire de défendre ses intérêts, mais surtout parce que personne ne sait dire ce qu'est un utilisateur « moyen » d'un microprocesseur : suivant vos applications vous ferez très souvent des divisions virgule flottante ou jamais, vos données seront très souvent dans la « zone à risques » ou jamais. Toujours suivant vos applications, le fait qu'une erreur puisse se glisser vers la 4ème décimale lors d'une division peut être sans conséquences ou avoir des conséquences importantes. On peut tout de même donner quelques informations objectives :

— Comme l'indique la deuxième ligne de la table 2, en double précision, si les réels apparaissant dans les calculs sont équiprobables, le chiffre d'un cas conduisant à une erreur sur 9 milliards donné par Intel est correct ;

— hélas, dans beaucoup de domaines (ex. calcul financier) des nombres proches de « nombres ronds » sont bien plus probables (que l'on pense par exemple à des sommes en francs et centimes légèrement « bruitées » par des erreurs d'arrondi). Comme on l'a vu les dénominateurs « dangereux » sont égaux à une puissance — positive ou négative — de 2 multipliée par un nombre très légèrement inférieur à 3, 9, 15, 21 ou 27. Ceci explique pourquoi la probabilité d'erreur augmente lorsque les données qu'on traite sont très proches de nombres entiers (songeons à l'exemple de 4.999999/14.999999 donné au début de cet article). En utilisant les résultats de ses tests, Vaughan Pratt a montré que si on s'intéresse au calcul de x/y, avec x et y compris entre 1 et 100 et distants de moins de $\frac{1}{100000}$ d'un entier (ce qui constitue tout de même un cas extrême !), alors la probabilité d'erreur monte à 1 sur 3000.

— bien que ceci ait sans doute peu d'influence sur les probabilités d'erreur, il est bon de savoir que, même avec des données numériques « brassées » (c'est-à-dire après de grandes quantités de calcul), les mantisses ne sont pas équiprobables. Leur répartition est plus proche d'une répartition logarithmique (d'ailleurs une répartition logarithmique des mantisses se conserve par multiplication, ce qui n'est pas le cas d'une répartition uniforme). Ce fait a été noté pour la première fois par Hamming [HAM 70]. On peut le visualiser en faisant tourner le petit programme suivant (T est un tableau de 100000 réels) :

```
Pour i = 1 ...100000 faire T[i] = random
Pour i = 1 ...5 faire
    pour j = 1 ...100000 faire
       T[j] = T[random] + T[random]
```

On voit figure 10 que la répartition obtenue est très proche de la répartition logarithmique prédite par Hamming, et assez distante d'une répartition uniforme.

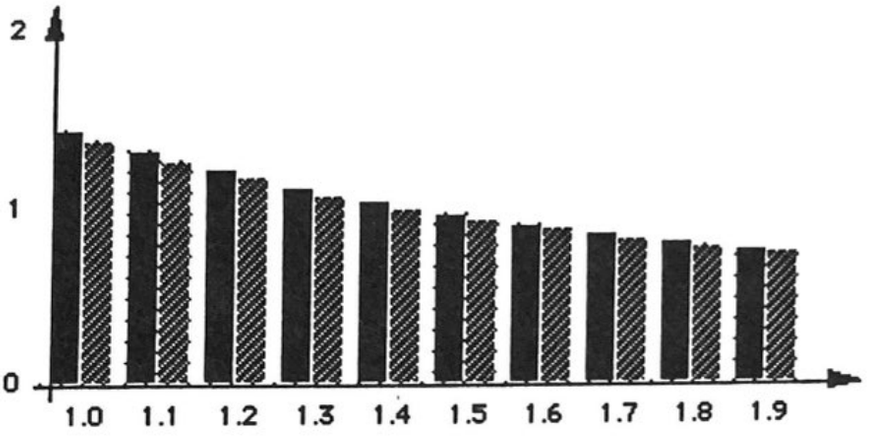

Figure 10. *Répartition des mantisses des nombres après exécution du programme (en hachuré), comparée à la répartition logarithmique prédite par Hamming (en noir)*

Il n'est pas suffisant de s'intéresser uniquement aux probabilités d'apparition des erreurs : c'est mettre dans le même paquet une erreur apparaissant à la quatrième décimale et une erreur apparaissant après la dixième. La répartition des erreurs suivant leur *ordre de grandeur* est plus importante. En examinant la liste des cas mis en évidence par V. Pratt, on s'aperçoit que les grandes erreurs sont bien moins probables que les petites. Ceci s'explique assez simplement : supposons que nous calculions x/y, où y est une des valeurs pouvant conduire à une erreur. Nous sommes à l'étape i de l'algorithme, on peut avoir l'apparition d'une erreur :

— soit tout de suite (i.e. lors du choix de q_{i+1}) si $x^{(i)}$ est dans la «zone à risques», c'est-à-dire si $x^{(i)}$ est très proche de $2y/3$;

— soit à l'étape suivante (i.e. lors du choix de q_{i+2}) si $x^{(i)}$ est dans un des antécédents de la «zone à risques». Il y a 4 antécédents de cette zone, et ils sont de taille 4 fois inférieure à celle de cette zone ;

— soit en deux coups (i.e. lors du choix de q_{i+3}) si $x^{(i)}$ est dans un des antécédents des antécédents de la «zone à risques». Il y a 16 antécédents des antécédents de cette zone, et ils sont de taille 16 fois inférieure à celle de cette zone ;

— etc.

La figure 11 représente la « zone à risques » (sa taille est considérablement exagérée) ainsi que ses antécédents en 1 et 2 coups. On voit qu'en prenant les antécédents des antécédents des antécédents, etc. on va progressivement remplir le diagramme : ceci montre qu'avoir une erreur au bout de plusieurs itérations est bien plus probable que d'avoir une erreur tout de suite.

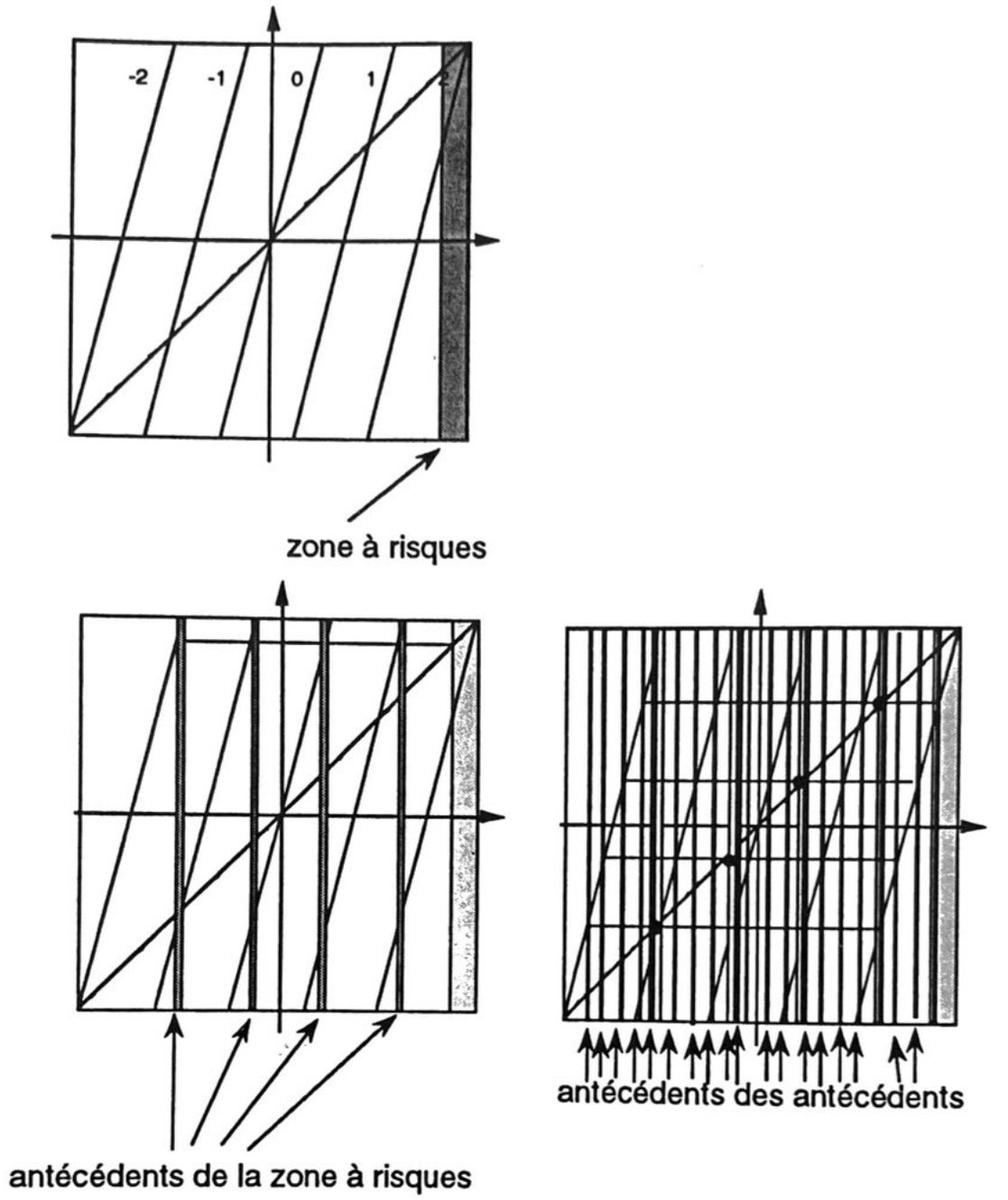

Figure 11. *La zone à risques (on a considérablement exagéré sa taille), ainsi que ses antécédents en 1 et 2 itérations*

6. Correction

Des versions corrigées du Pentium ont maintenant été produites, et la société Intel remplace les microprocesseurs comportant le «bug». Il est également possible de faire assez simplement des corrections logicielles. La méthode qui suit est due à T. Coe et P. Tang, elle est basée sur le fait que si y est un dénominateur «à risque», alors $15y/16$ ne l'est pas.

— tester si les premiers bits de mantisse de y sont de la forme 1.0001, 1.0100, 1.0111, 1.1010 ou 1.1101 ;
— si c'est le cas, calculer $\frac{(15/16)\times x}{(15/16)\times y}$, sinon calculer x/y.

Conclusion

En résumé, si en effectuant avec un Pentium des divisions de grosses erreurs sont très peu probables (et même dans un tel cas, le résultat a toujours au moins 3 à 4 chiffres significatifs), de très petites erreurs peuvent, elles, apparaître assez fréquemment. Il me parait clair que ceux qui effectuent du calcul numérique intensif, ou qui ont des applications où de petits entiers « bruités » par les erreurs d'arrondis apparaissent souvent, ou encore ceux qui travaillent dans des domaines où les conséquences d'une erreur relative de l'ordre de 10^{-4} sur une division peuvent être graves doivent se méfier. Dans les autres domaines, il semble qu'il n'y ait rien à craindre et qu'un peu de modération soit de mise : on a vu dans des *newsgroups* des utilisateurs dont la seule application était le traitement de textes faire preuve d'une fureur plus qu'exagérée.

De plus, si vos applications sont susceptibles d'être *très* sensibles à l'erreur de la division du Pentium, elle sont certainement *au moins un peu* sensibles au cumul des erreurs d'arrondi, et il peut s'avérer nécessaire de les valider numériquement. Des outils existent maintenant pour cela, et plusieurs équipes en France travaillent dans ce domaine (entre autres au MASI, à l'IRISA, au Centre Spatial de Toulouse du CNES, au CERFACS et au LIP). Même avec un processeur sans « bug » les conséquences de l'erreur d'arrondi peuvent être énormes. Un rapport du *General Accounting Office* américain cite le cas d'un missile *Patriot* qui, pendant la guerre du golfe, n'a pas intercepté sa cible à la suite d'accumulation d'erreurs d'arrondis, causant la mort de 28 personnes.

7. Bibliographie

[AND 67] ANDERSON S.F., EARLE J.G., GOLDSCHMIDT R.E., et POWERS D.M., The IBM 360/370 model 91 floating-point execution unit, *IBM Journ. of Res. and Dev.*, Janvier 1967. Réédité dans E.E. Swartzlander, Computer Arithmetic, Vol. 1, IEEE Computer Society Press Tutorial, 1990.

[AVI 61] AVIZIENIS A., Signed-digit number representations for fast parallel arithmetic, *IRE Transactions on electronic computers*, 10, p. 389–400, 1961. Réédité dans E.E. Swartzlander, Computer Arithmetic, Vol. 2, IEEE Computer Society Press Tutorial, 1990.

[BOS 87] BOSE B.K., PATTERSON D.A., PEI L., et TAYLOR G.S., Fast multiply and divide for a VLSI floating-point unit, In *8th IEEE Symposium on Computer Arithmetic*, IEEE Computer Society Press, 1987. Réédité dans E.E. Swartzlander, Computer Arithmetic, Vol. 2, IEEE Computer Society Press Tutorial, 1990.

[COR 94] CORTADELLA J. et LANG T., High-radix division and square-root with speculation, *IEEE Transactions on Computers*, 43 n° 8, p. 919–931, Août 1994.

[EIS 93] EISIG D., ROTSTAIN J., et KOREN I., The design of a 64-bit integer multiplier/divider unit, In M.J. Irwin, E.E. Swartzlander et G. Jullien, editeurs, *11th Symposium on Computer Arithmetic*, p. 171–178, Windsor, Canada, Juin 1993. IEEE Computer Society Press.

[ERC 83] ERCEGOVAC M.D., A higher-radix division with simple selection of quotient digits, In *6th Symposium on Computer Arithmetic*, p. 94–98, Aarhus, Danemark, Juin 1983. IEEE Computer Society Press.

[ERC 85] ERCEGOVAC M.D. et LANG T., A division algorithm with prediction of quotient digits, In *7th IEEE Symposium on Computer Arithmetic*, Urbana, USA, Juin 1985. IEEE Computer Society Press.

[ERC 94] ERCEGOVAC M.D. et LANG T., *Division and Square Root, Digit-Recurrence Algorithms and Implementations*, Kluwer Academic Publishers, 1994.

[FAN 89] FANDRIANTO J., Algorithms for high-speed shared radix 8 division and radix 8 square-root, In *9th IEEE Symposium on Computer Arithmetic*, Santa Monica, USA, Sept. 1989. IEEE Computer Society Press.

[FEN 95] FENWICK P., High-radix division with approximate quotient-digit estimation, *JUCS*, 1 n° 1, Janvier 1995.

[FLY 70] FLYNN M.J., On division by functional iteration. *IEEE Transactions on Computers*, C-19 n° 8, p. 702–706, Août 1970, Réédité dans E.E. Swartzlander, Computer Arithmetic, Vol. 1, IEEE Computer Society Press Tutorial, 1990.

[HAM 70] HAMMING R.W., On the distribution of numbers. *Bell Systems Technical Journal*, 49, p. 1609–1625, 1970, Réédité dans E.E. Swartzlander, Computer Arithmetic, Vol. 1, IEEE Computer Society Press Tutorial, 1990.

[HWA 79] HWANG K., *Computer Arithmetic Principles, Architecture and design*, Wiley & Sons Inc, 1979.

[KOR 93] KOREN I., *Computer Arithmetic Algorithms*, Prentice-Hall, 1993.

[MAN 93] MANDELBAUM D.M., Some results on a SRT type division scheme, *IEEE Transactions on Computers*, 42 n° 1, p. 102–106, Janvier 1993.

[MCQ 93] MCQUILLAN S.E., MCCANNY J.V., et HAMILL R., New algorithms and VLSI architectures for SRT division and square root, In M.J. Irwin E.E. Swartzlander et G. Jullien, editeurs, *11th Symposium on Computer Arithmetic*, p. 80–86, Windsor, Canada, Juin 1993, IEEE Computer Society Press.

[MON 92] MONTUSCHI P. et CIMINIERA L., Design of a radix 4 division unit with simple selection table, *IEEE Transactions on Computers*, 41, n° 12 p. 1606–1611, Décembre 1992.

[MUL 89] MULLER J.M., *Arithmétique des Ordinateurs*, Masson, Paris, 1989.

[ROB 58] ROBERTSON J.E., A new class of digital division methods, *IRE Transactions on Electronic Computers*, EC-7, p. 218–222, 1958. Réédité dans E.E. Swartzlander, Computer Arithmetic, Vol. 1, IEEE Computer Society Press Tutorial, 1990.

[SVO 63] SVOBODA A., An algorithm for division, *Inf. Process. Mach.*, 9, p. 25–32, 1963. Réédité dans E.E. Swartzlander, Computer Arithmetic, Vol. 1, IEEE Computer Society Press Tutorial, 1990.

[TOC 58] TOCHER K.D., Techniques of multiplication and division for automatic binary divider, *Quarterly journal of mechanics and applied mathematics*, 11, n° 3 p. 364–384, 1958.

[TUN 68] TUNG C., A division algorithm for signed-digit arithmetic, *IEEE Transactions on Computers*, C-17, 1968.

Article reçu le 10 février 1995.
Version révisée le 16 juin 1995.
Rédacteur responsable : Philippe Besnard.

Jean-Michel Muller *est chargé de recherche CNRS au Laboratoire de l'informatique du parallélisme (Ecole normale supérieure de Lyon), où il dirige le groupe « Silico-algorithmes et arithmétique des ordinateurs ». Son principal domaine d'intérêt est l'arithmétique des ordinateurs, qui recouvre l'étude des systèmes de numération utilisables en informatique, et des algorithmes et architectures permettant d'évaluer les fonctions arithmétiques et élémentaires.*