Arithmétique des ordinateurs

Arnaud Tisserand
Arénaire INRIA LIP

École Jeunes Chercheurs en Algorithmique et Calcul Formel
Grenoble, 29 mars – 2 avril 2004

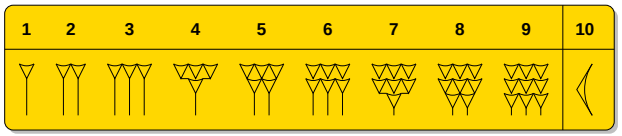
Plan de l'exposé

Partie 1

Introduction

L'arithmétique chez les Babyloniens

Utilisation d'un système de position (le premier de l'histoire) en base 60 avec les chiffres suivants (et la base auxiliaire 10) :



Exemple de codage :

$= 33 \times 60 + 24 = 2004$

Système de position en base β sur n chiffres (pour des entiers naturels) :

$$x = x_{n-1}x_{n-2}\cdots x_1x_0 = \sum_{i=0}^{n-1} x_i\,\beta^i$$

Oh, la belle table de multiplication. . .

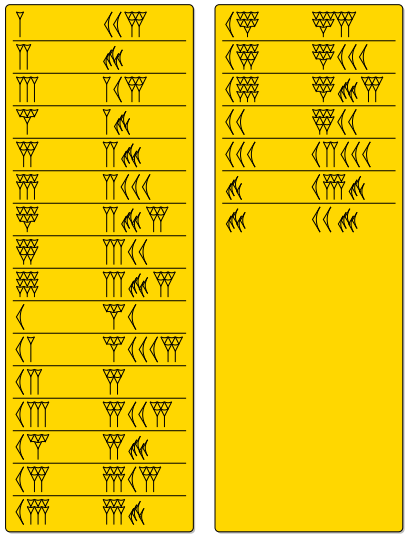
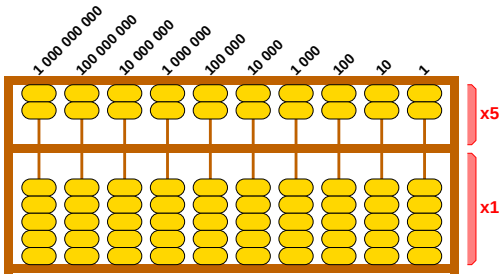


Illustration de la table de multiplication par 25 trouvée à Suse et datée du II^{e} millénaire av. J.-C (conservée au Musée du Louvre).

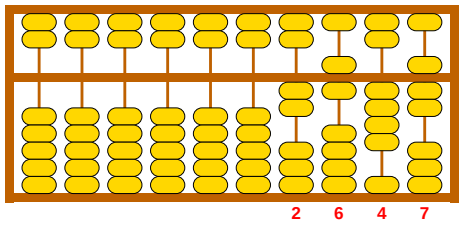
Remarque : seuls les produits par (1), 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 30, 40, 50 sont nécessaires sur les 59 possibles.

Le boulier chinois : position de repos (0)



- Les colonnes représentent les unités, les dizaines, les centaines, les milliers,. . . de la droite vers la gauche
- Les boules sont activées lorsqu'elles sont au plus proche de la barre du milieu, et désactivées sinon (0 est donc représenté ci-dessus)
- Les boules situées au dessus de la barre horizontale du milieu sont multipliées par 5 (celles de dessous par 1).

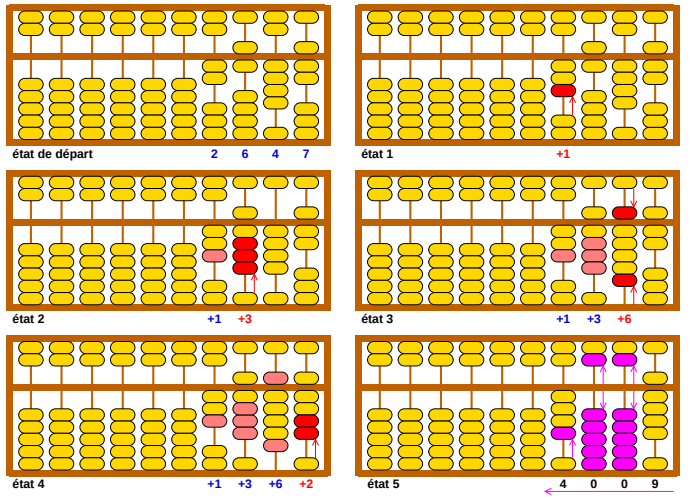
Le boulier chinois : écriture de 2647



La première illustration connue d'un boulier chinois dans un livre date de 1175. Il existe des dispositifs proches (tablettes avec des rainures sur lesquelles on déplace des petits cailloux) depuis bien plus longtemps.

Il existe des concours de rapidité de calcul sur des bouliers. Le japonais Yoshio Kogima a effectué correctement 50 divisions, avec des opérandes de 5 à 7 chiffres, en 1 min 18 s et 4 centièmes sur un boulier japonais (*soroban*) !

Le boulier chinois : 2647+1362



Arithmétique des ordinateurs

Étude et conception de "moyens" pour effectuer les calculs de base en machine.

- unités de calcul matérielles :
  - additionneur/soustracteur, multiplieur, diviseur, . . .
  - unités flottantes
  - opérateurs spécifiques (ex : filtres pour traitement du signal)
- support logiciel pour les calculs de base :
  - bibliothèques mathématiques de base (libm)
  - bibliothèques de fonctions élémentaires $(\sin, \cos, \exp, \log, \ldots)$
  - bibliothèques multi-précision
  - bibliothèques d'arithmétique d'intervalle
- validation de la qualité numérique :
  - test et/ou preuve de la précision de calculs
  - preuve du bon comportement des opérations (dépassements, . . .)

Arithmétique des ordinateurs (suite)

Les trois aspects fondamentaux de l'arithmétique des ordinateurs :

- Systèmes de représentation des nombres :
  entier, virgule fixe, virgule flottante, redondant, grande base, système logarithmique, système modulaire, corps finis. . .
- Algorithmes de calcul :
  addition–soustraction, multiplication, division, PGCD, racine carrée, fonctions élémentaires $(\sin, \cos, \exp, \log \ldots)$, opérateurs composites (ex : $1/\sqrt{(x^2+y^2)}$), opérateurs spécifiques (FIR, DCT, crypto), algorithmes numériques, preuves de programmes. . .
- Maîtriser les implantations :
  cibles logicielles et matérielles, support arithmétique dans les langages de programmation, validation, test, optimisation des performances (vitesse, mémoire, surface de circuit, temps réel, consommation d'énergie). . .

Arithmétique des ordinateurs (suite)

Liens avec :

- architecture des ordinateurs
- micro-électronique
- outils CAO de circuits
- algorithmes numériques
- calcul formel
- optimisation globale
- théorie des nombres
- preuves formelles
- . . .

Arithmétique des ordinateurs (suite)

Exemples de sujets de recherche dans l'équipe/projet Arénaire :

- bibliothèque de fonctions élémentaires avec arrondi correct
- bibliothèque d'arithmétique d'intervalle en multi-précision
- bibliothèque pour le support flottant dans les processeurs entiers
- opérateurs de cryptographie pour circuits FPGA
- unités flottantes et logarithmiques pour circuits FPGA
- opérateurs arithmétiques matériels à basse consommation d'énergie
- arithmétique corps finis pour bibliothèque d'algèbre linéaire
- preuves automatiques d'opérations arithmétiques

Système logarithmique

Un nombre est représenté par son signe et le logarithme de sa valeur absolue écrit en virgule fixe (le nombre 0 doit être représenté par un codage spécial).

Les opérations dans ce système s'effectuent en utilisant :

$$\begin{aligned}
\log_2(a \times b) &= \log_2 a + \log_2 b \\
\log_2(a \div b) &= \log_2 a - \log_2 b \\
\log_2(a \pm b) &= \log_2 a + \log_2(1 \pm 2^{\log_2 b - \log_2 a}) \\
\log_2(a^q) &= q \times \log_2 a
\end{aligned}$$

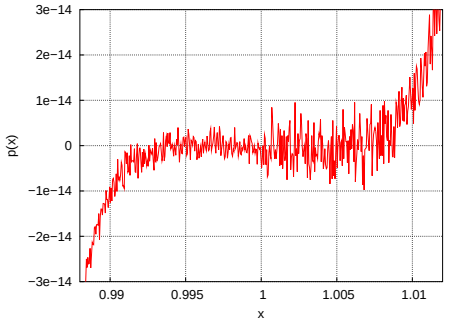
où les fonctions $\log_2(1 + 2^x)$ et $\log_2(1 - 2^x)$ sont tabulées ou approchées.

Applications en traitement du signal et en contrôle numérique. Il y avait même un projet européen pour concevoir un processeur avec des unités de calcul 32 bits en système logarithmique.

Petits problèmes numériques : un polynôme rebelle ?

Avec un petit programme C, calculons des valeurs (*float*) de $p(x)$ ci-dessous pour x entre 0.988 et 1.012, et traçons la courbe correspondante.

$$p(x) = x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1$$

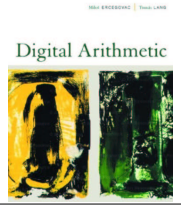


Pour en savoir plus. . .

Digital Arithmetic

Milos Ercegovac et Tomas Lang

2003
Morgan Kaufmann
ISBN : 1–55860–798–6



Arithmétique des ordinateurs

Jean–Michel Muller

1989
Masson
ISBN : 2–225–81689–1

Autres références

- Livre de G. Guitel : Histoire comparée des numérations écrites, Flammarion, 1975
- Livre de G. Ifrah : Histoire universelle des chiffres, Robert Lafond, 1994
- Document *Les Langages Numériques* de Jean Vuillemin : `http://www.di.ens.fr/~jv/HomePage/pdf/langnum.pdf`
- Numéro spécial collectif de la revue *Réseaux et systèmes répartis, calculateurs parallèles* sur l'*arithmétique des ordinateurs*, Hermes, 2001.
- Un site web sur les bouliers : `http://www.ee.ryerson.ca:8080/~elf/abacus/history.html`

Partie 2

Arithmétique flottante

Points abordés dans cette partie

- Problèmes historiques
- Représentation des nombres
- Comportement des opérations
- Propriétés des nombres flottants
- Problèmes de précision
- Exemples

Représentation flottante

Un nombre x est représenté en virgule flottante de base β par :

- son signe s_x (codage sur un bit : 0 pour x positif et 1 pour x négatif)
- son exposant e_x, un entier de k chiffres compris entre e_{min} et e_{max}
- sa mantisse m_x de $n+1$ chiffres

tels que

$$x = (-1)^{s_x} \times m_x \times \beta^{e_x}$$

avec

$$m_x = x_0 \,.\, x_1\, x_2\, x_3\, \cdots\, x_n$$

où $x_i \in \{0, 1, \ldots, \beta - 1\}$.

Pour des questions de précision, on exige que la mantisse soit normalisée, c'est-à-dire que son premier chiffre x_0 soit différent de 0. On a alors $m_x \in [1, \beta[$. Il faut alors un codage spécial pour le nombre 0.

Au début était le chaos. . .

Les représentations flottantes étaient pendant longtemps très différentes les unes des autres selon les constructeurs de processeurs.

machine	β	n	e_{min}	e_{max}
Cray 1	2	48	-8192	8191
	2	96	-8192	8191
DEC VAX	2	53	-1023	1023
	2	56	-127	127
HP 28 et 48G	10	12	-499	499
IBM 3090	16	6	-64	63
	16	14	-64	63
	16	28	-64	63

Problème : il n'était pas possible de faire raisonnablement des programmes et des bibliothèques numériques portables !

Autres exemples de problèmes

- IBM System/370 en FORTRAN on avait $\sqrt{-4} = 2$
- Sur certaines machines CDC et Cray on avait :

$$x + y \neq y + x$$

$$0.5 \times x \neq x/2.0$$

Avec ça, comment écrire des programmes numériquement corrects de façon simple ?

A l'époque, les constructeurs ne s'intéressent qu'aux formats de stockage des données et pas beaucoup aux propriétés mathématiques des unités de calcul. . .

Révolution de 1985 : la norme IEEE-754

Après de nombreuses années de travail, une norme fixe la représentation des données et le comportement des opérations de base en virgule flottante.

Cette norme fixe :

- les formats des données
- les valeurs spéciales
- les modes d'arrondi
- la précision des opérations de base
- les règles de conversion

En fait, il y a deux normes[1] :

- *ANSI/IEEE Standard for Binary Floating-Point Arithmetic* en 1985
- *ANSI/IEEE Standard for Radix-Independent Floating-Point Arithmetic* en 1987 (où $\beta = 2$ ou 10)

[1] ANSI : *American National Standards Institute*, IEEE : *Institute of Electrical and Electronics Engineers*

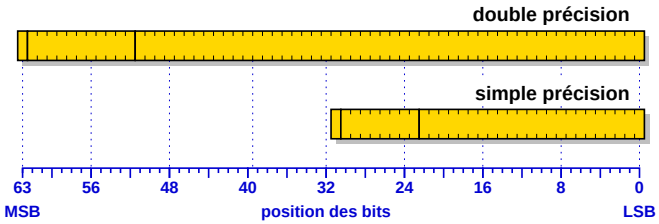
Objectifs de la norme IEEE-754

- permettre de faire des programmes portables
- rendre les programmes déterministes d'une machine à une autre
- conserver des propriétés mathématiques
- permettre/imposer l'arrondi correct
- permettre/imposer des conversions fiables
- faciliter la construction de preuves
- faciliter la gestion des exceptions
- faciliter les comparaisons (unicité de la représentation, sauf pour 0)
- permettre un support pour l'arithmétique d'intervalle

IEEE-754 : formats de base

En base $\beta = 2$, la normalisation de la mantisse implique que le premier bit est toujours un "1" qui n'est pas stocké physiquement, on parle de 1 implicite.

format	nombre de bits			
	total	signe	exposant	mantisse
double précision	64	1	11	$52 + 1$
simple précision	32	1	8	$23 + 1$



IEEE-754 : mantisse et fraction

La mantisse (normalisée) du nombre flottant x est représentée par $n+1$ bits :

$$m_x = 1 \,.\, \underbrace{x_1\ x_2\ x_3\ \cdots\ x_{n-1}\ x_n}_{f_x}$$

où les x_i sont des bits.

La partie fractionnaire de m_x est appelée fraction (de n bits) : f_x. On a alors :

$$m_x = 1 + f_x$$

On a aussi :

$$1 \leq m_x < 2$$

IEEE-754 : exposant

L'exposant e_x est un entier signé de k bits tel que :

$$e_{min} \leq e \leq e_{max}$$

Différentes représentations sont possibles : complément à 2, signe et magnitude, biaisée. C'est la représentation biaisée qui est choisie.

Ceci permet de faire les comparaisons entre flottants dans l'ordre lexicographique (sauf pour le signe s_x) et de représenter le nombre 0 avec $e_x = f_x = 0$.

L'exposant stocké physiquement est l'exposant biaisé e_b tel que :

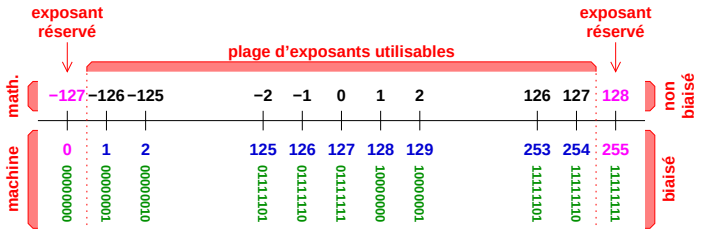
$$e_b = e + b$$

où b est le biais.

IEEE-754 : exposant (suite)

Les exposants non biaisés $e_{min}-1$ et $e_{max}+1$ (respectivement 0 et 2^k-1 en biaisé) sont réservés pour zéro, les dénormalisés et les valeurs spéciales.

format	taille k	biais b	non-biaisé e_{min}	non-biaisé e_{max}	biaisé e_{min}	biaisé e_{max}
SP	8	127 $(= 2^{8-1}-1)$	-126	127	1	254
DP	11	1023 $(= 2^{11-1}-1)$	-1022	1023	1	2046



IEEE-754 : zéro

Le nombre zéro est codé en mettant tous les bits de l'exposant et de la fraction à 0 dans le mot machine. Le bit de signe encore "libre" permet d'avoir deux représentations différentes du nombre zéro : -0 et $+0$.

Le fait que zéro soit signé est cohérent avec le fait qu'il y ait deux infinis distincts. On a alors :

$$\frac{1}{+0} = +\infty \qquad \text{et} \qquad \frac{1}{-0} = -\infty$$

La norme impose que le test $-0 = +0$ retourne la valeur vrai.

En simple précision, on a donc :

-0	1	00000000	00000000000000000000000
$+0$	0	00000000	00000000000000000000000

IEEE-754 : valeurs spéciales

- Les infinis : $-\infty$ et $+\infty$
  Ils sont codés en utilisant le plus grand exposant possible et une fraction nulle. L'infini est signé.

$$e = e_{max} + 1 \qquad \text{et} \qquad f_x = 0$$

- Not a Number : `NaN`
  Permet de représenter le résultat d'une opération invalide telle que $0/0$, $\sqrt{-1}$ ou $0 \times \infty$. `NaN` est codé en utilisant le plus grand exposant possible et une fraction non-nulle. Les `NaN` se propagent dans les calculs.

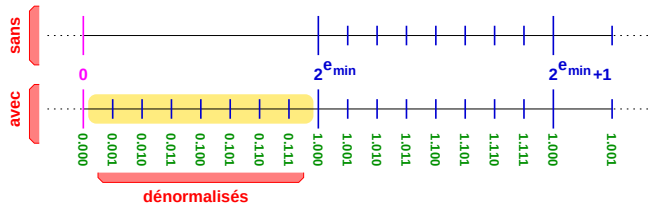
$$e = e_{max} + 1 \qquad \text{et} \qquad f_x \neq 0$$

En simple précision, on a donc :

$-\infty$	`1 11111111 00000000000000000000000`
$+\infty$	`0 11111111 00000000000000000000000`
`NaN`	`0 11111111 00000000000000000000100` (par exemple)

IEEE-754 : nombres dénormalisés

L'objectif des nombres dénormalisés est d'uniformiser la répartition des nombres représentables autour de 0. En effet, le 1 implicite dans la mantisse implique qu'il n'y a pas de nombre représentable entre 0 et $2^{e_{min}}$ alors qu'il y en a 2^n entre $2^{e_{min}}$ et $2^{e_{min}+1}$.



Les nombres dénormalisés s'écrivent (avec $e_b = 0$) :

$$x = (-1)^{s_x} \times (0.f_x) \times 2^{e_{min}}$$

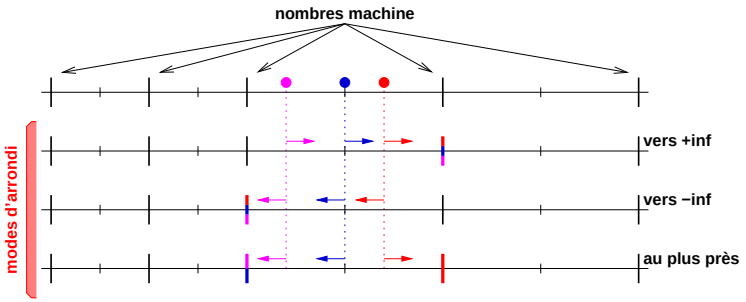
IEEE-754 : modes d'arrondis

Si a et b sont deux nombres exactement représentables en machine (ou nombres machine) alors le résultat d'une opération $r = a \circledcirc b$ n'est, en général, pas représentable en machine. Il faut arrondir le résultat. Par exemple, en base $\beta = 10$, le nombre $1/3$ n'est pas représentable avec un nombre fini de chiffres.

La norme propose 4 modes d'arrondi :

- arrondi vers $+\infty$ (ou par excès), noté $\triangle(x)$: retourne le plus petit nombre machine supérieur ou égal au résultat exact x
- arrondi vers $-\infty$ (ou par défaut), noté $\nabla(x)$: retourne le plus grand nombre machine inférieur ou égal au résultat exact x
- arrondi vers 0, noté $\mathcal{Z}(x)$: retourne $\triangle(x)$ pour les nombres négatifs et $\nabla(x)$ pour les positifs
- arrondi au plus près, noté $\circ(x)$: retourne le nombre machine le plus proche du résultat exact x (celui dont la mantisse se termine par un 0 pour le milieu de nombres machine consécutifs, on parle d'arrondi pair)

IEEE-754 : modes d'arrondis (suite)



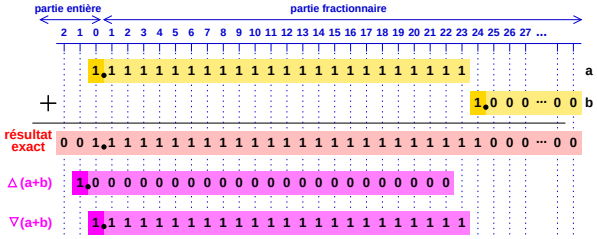
Propriété d'arrondi correct : soient a et b deux nombres machine, $\circledcirc$ une des opérations $(+, -, \times, \div)$ et $\diamond$ le mode d'arrondi choisi (parmi les 4 modes IEEE). Le résultat fourni lors du calcul de $(a \circledcirc b)$ doit être $\diamond(a \circledcirc_{th} b)$.

Le résultat retourné doit être celui obtenu par un calcul avec une précision infinie, puis arrondi. On a la même exigence pour la racine-carrée.

IEEE-754 : modes d'arrondi en C

```
#include <stdio.h>
#include "util-ieee.h"
#include <fenv.h> /* gestion de l'environnement flottant */

int main () {
  cfloat a, b, sup, sdown;

  /* a = 2 - 2^(-23)     et     b = 2^(-24)*/
  a.s.sig = 0;   a.s.exp = 0+127;     a.s.man = (2<<22)-1;
  b.s.sig = 0;   b.s.exp = -24+127;  b.s.man = 0;
  cfloat_print("               a",a);   cfloat_print("               b",b);

  fesetround(FE_UPWARD); /* passe en arrondi vers le haut */
  sup.f = a.f + b.f;
  cfloat_print("  (a+b) rnd up",sup);

  fesetround(FE_DOWNWARD); /* passe en arrondi vers le bas */
  sdown.f = a.f + b.f;
  cfloat_print("(a+b) rnd down",sdown);

  return 0; }
```


IEEE-754 : modes d'arrondi en C (suite)




```
  (a+b) rnd up = 2.00000000000000000000000000000000000000000e+00
  (a+b) rnd up = 0 128(    1)          0
  (a+b) rnd up = 0 128(    1) 1.00000000000000000000000
  (a+b) rnd up = 0x40000000

(a+b) rnd down = 1.99999988079071044921875000000000000000000e+00
(a+b) rnd down = 0 127(    0)    8388607
(a+b) rnd down = 0 127(    0) 1.11111111111111111111111
(a+b) rnd down = 0x3fffffff
```


IEEE-754 : conversions

Les conversions normalisées sont :

- flottant vers entier
- entier vers flottant
- flottant vers entier stocké dans un flottant
- entre flottants de différents formats
- entre formats binaire et décimal

Propriétés des doubles conversions imposées par la norme :

- binaire $x \longrightarrow$ décimal $x_{(10)} \longrightarrow$ binaire $x_{(2)}$:
  on retrouve le nombre initial, c'est-à-dire $x = x_{(2)}$, si $x_{(10)}$ a au moins 9 chiffres décimaux pour x en simple précision (17 en DP)
- décimal $y \longrightarrow$ binaire $y_{(2)} \longrightarrow$ décimal $y_{(10)}$:
  on retrouve le nombre initial, c'est-à-dire $y = y_{(10)}$, si y a au plus 6 chiffres décimaux et que $y_{(2)}$ en simple précision est converti en $y_{(10)}$ avec 6 chiffres décimaux (15 en DP)

IEEE-754 : comparaisons

La norme impose que l'opération de comparaison soit exacte et ne gérée pas de dépassement de capacité.

Les comparaisons normalisées sont :

- égalité
- supérieur
- inférieur
- non-ordonné

Lors de ces comparaisons le signe de zéro n'est pas pris en compte.

Dans le cas de comparaisons impliquant un `NaN`, la comparaison retourne faux. Sauf dans le cas de l'égalité : si $x =$ `NaN` alors $x = x$ retourne faux[2] et $x \neq x$ retourne vrai.

[2] Ce qui permet de tester si une valeur est un NaN.

IEEE-754 : drapeaux ou exceptions

La norme précise qu'aucun calcul ne doit entraver le bon fonctionnement de la machine. Un mécanisme de drapeaux permet d'informer le système sur le comportement des opérations. La norme prévoit 5 drapeaux :

- opération invalide : le résultat par défaut est `NaN`
- dépassement de capacité vers ∞ (*overflow*) : le résultat est soit $\pm\infty$ soit le plus grand nombre représentable (en valeur absolue) suivant le signe du résultat exact et du mode d'arrondi
- depassement de capacité vers 0 (*underflow*) : le résultat est soit ± 0 soit un dénormalisé
- division par zéro : le résultat est $\pm\infty$
- résultat inexact : levé lorsque que le résultat d'une opération n'est pas exact

Ces drapeaux, une fois levés, le restent pendant tout le calcul jusqu'à une remise à zéro volontaire (*sticky flags*). Ils peuvent être lus et écrits par l'utilisateur.

IEEE-754 : dynamique de la représentation

La dynamique d'une représentation est le rapport entre le plus grand nombre et le plus petit nombre représentables et strictement positifs.

- en virgule fixe sur n bits, on a :

$$D_{fixe} = \frac{2^n - 1}{1}$$

- en virgule flottante (e sur k bits et m sur $n+1$ bits), on a :

$$D_{flottant} = \frac{m_{max} \times 2^{e_{max}}}{m_{min} \times 2^{e_{min}}} = \frac{(2 - 2^{1-n}) \times 2^{2^{k-1}-1}}{(2^{1-n}) \times 2^{-2^{k-1}+2}}$$

Pour 32 bits on a : $D_{fixe} = 4.29 \times 10^9$ et $D_{flottant} = 2.43 \times 10^{85}$.

IEEE-754 : autres formats

format	taille totale	taille exposant	taille mantisse
simple étendu	≥ 32	≥ 11	≥ 24
double étendu	≥ 64	≥ 15	≥ 43
double étendu PC	80	15	64
quad (Sun)	128	15	113

Dans les formats étendus, il n'y a pas de bit implicite pour la mantisse, il doit être stocké dans le mot machine.

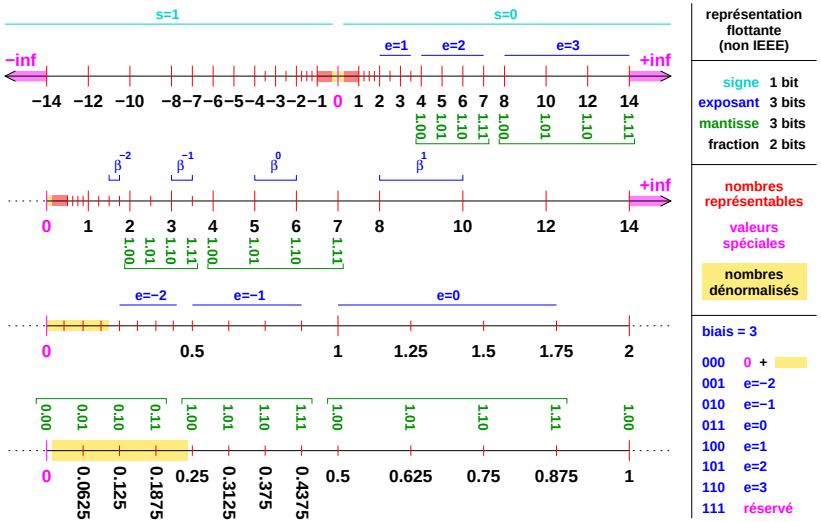
IEEE-754 : résumé

	exposant	fraction	valeur
normalisés	$e_{min} \leq e \leq e_{max}$	$f \geq 0$	$\pm(1.f) \times 2^e$
dénormalisés	$e = e_{min} - 1$	$f > 0$	$\pm(0.f) \times 2^{e_{min}}$
zéro (signé)	$e = e_{min} - 1$	$f = 0$	± 0
infinis	$e = e_{max} + 1$	$f = 0$	$\pm\infty$
Not a Number	$e = e_{max} + 1$	$f > 0$	`NaN`

format	# bits total	k	n	e_{min}	e_{max}	b
simple précision	32	8	23	-126	127	127
double précision	64	11	52	-1022	1023	1023

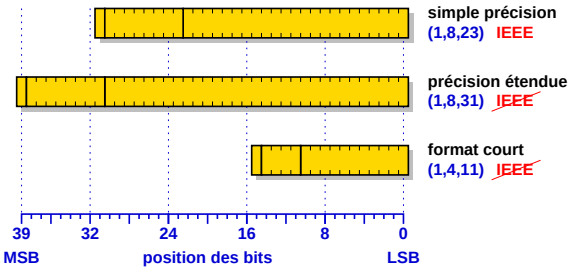
valeur	simple précision	double précision
+ grand normalisé > 0	$3.40282347 \times 10^{38}$	$1.7976931348623157 \times 10^{308}$
+ petit normalisé > 0	$1.17549435 \times 10^{-38}$	$2.2250738585072014 \times 10^{-308}$
+ grand dénormalisé > 0	$1.17549421 \times 10^{-38}$	$2.2250738585072009 \times 10^{-308}$
+ petit dénormalisé > 0	$1.40129846 \times 10^{-45}$	$4.9406564584124654 \times 10^{-324}$

Exemple de représentation simplifiée



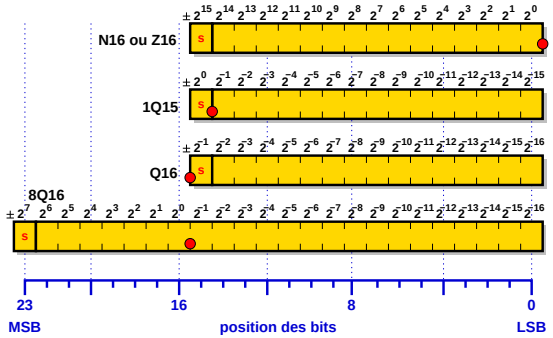
Représentations flottantes non-standard

Exemple du processeur DSP (*digital signal processor*) SHARC 21160 d'Analog Devices où il y a plusieurs formats flottants non-standard (et seulement un compatible IEEE-754 simple précision).



Alternative aux flottants : la virgule fixe

Dans bon nombre de processeurs DSP, on trouve un support matériel très efficace (en vitesse et consommation d'énergie) pour le calcul sur les *réels* à l'aide de multiples formats en virgule fixe (16, 24 ou 32 bits).



Mais la qualité numérique et la portablilité sont moindres qu'en IEEE-754

Références sur les flottants IEEE-754

- Documentation de W. Kahan (le "père" de la norme)
  `http://www.cs.berkeley.edu/~wkahan/ieee754status/`
- Documentation générale et scripts Java pour faire des conversions
  `http://babbage.cs.qc.edu/courses/cs341/IEEE-754references.html`
- Version web de l'article de D. Goldberg dans *ACM Computing Surveys*
  `http://docs.sun.com/source/806-3568/ncg_goldberg.html`
- Documentation du Centre Charles Hermite (Nancy)
  `http://cch.loria.fr/documentation/IEEE754/`

Multiplication-addition fusionnée (FMA)

Le FMA (pour *fused multiply and add*) existe depuis de nombreuses années dans les DSP et arrive de plus en plus dans les processeurs généralistes. Cette opération effectue le calcul suivant en une seule instruction (au lieu de 2 sans unité FMA).

$$r = (a + b \times c)$$

Pour le moment le comportement de cette opération n'est pas normalisé en IEEE-754. En pratique les processeurs généralistes qui possède une unité FMA retournent le meilleur résultat possible : l'arrondi du résultat théorique (arrondi correct de $(a + b \times c)$) en effectuant un seul arrondi.

Exemple d'utilisation : évaluation de polynômes

$$p(x) = p_0 + (p_1 + (p_2 + (p_3 + p_4\ x)\ x)\ x)\ x$$

Exemples de propriétés vraies en machine

L'addition flottante $\oplus$ et la multiplication flottante $\otimes$ sont commutatives mais pas associatives (dépassements de capacité par exemple). La multiplication flottante n'est pas distributive par rapport à l'addition.

Si aucun dépassement de capacité vers l'infini ou vers zéro ne se produit pendant les calculs, les propriétés suivantes sont vérifiées avec des nombres flottants et des opérations flottantes IEEE.

$$\begin{aligned} x \oplus 0 &= x \ominus 0 = x \\ x \otimes 1 &= x \oslash 1 = x \\ x \otimes -1 &= x \oslash -1 = -x \\ 2 \otimes x &= x \oplus x = 2x \\ 0.5 \otimes x &= x \oslash 2 = x/2 \end{aligned}$$

$$\begin{aligned} \circ(\sqrt{x}) &\geq 0 \quad \text{si} \quad x \geq 0 \\ \circ(\sqrt{-0}) &= -0 \end{aligned}$$

Exemples de propriétés vraies en machine (suite)

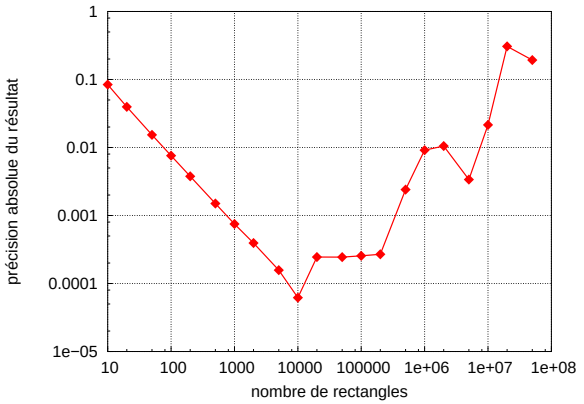
Grâce à la norme IEEE, en l'absence de dépassement de capacité (vers $+\infty$ ou vers 0) et de division par 0, avec x et y deux nombres machine on a :

$$-1 \quad \leq \quad \frac{x}{\sqrt{x^2 + y^2}} \quad \leq \quad 1$$

même après 5 erreurs d'arrondi.

Problème d'intégration numérique

À l'aide de la méthode des rectangles, essayons de calculer $\int_1^2 \frac{dx}{x}$ en SP (le résultat théorique est $\ln(2)$), et ce pour plusieurs nombres de rectangles :



Précision et erreur d'arrondi

A chaque arrondi, il est possible de perdre un peu de précision, on parle d'erreur d'arrondi. Même si une opération isolée retourne le meilleur résultat possible (l'arrondi du résultat exact), une suite de calculs peut conduire à d'importantes erreurs du fait du cumul des erreurs d'arrondi.

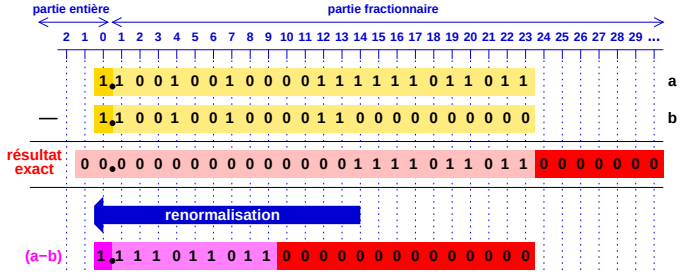
Précision en nombre de bits justes ($p_{bits} = -\log_2(|err_{absolue}|)$) :

$$\begin{aligned} a &= 1.110\,000\,000 = 1.75 \\ a' &= 1.101\,111\,111 = 1.748046875 \\ |a - a'| &= 0.000\,000\,001 = 0.00193125 \\ p_{bits} &= 9 \end{aligned}$$

Si a est la valeur exacte, alors a' représente a avec un 1 bit faux (et pas 8). En effet, $|a - a'| = 2^{-9}$, où 2^{-9} est le poids du dernier bit de a.

Phénomène de cancellation (ou élimination)

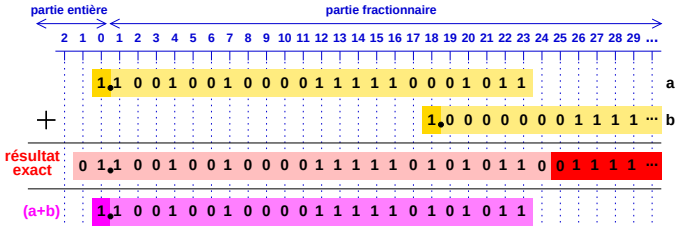
Se produit lors de la soustraction de deux nombres très proches.



L'opération $(a - b)$ est "exacte" car les opérandes sont supposés exactes. Mais si les opérandes sont elles-mêmes des résultats de calculs avec des erreurs d'arrondi, les 0 ajoutés à droite (partie rouge foncé) sont faux. La cancellation est dite catastrophique quand il n'y a presque plus de chiffres significatifs.

Phénomène d'absorption

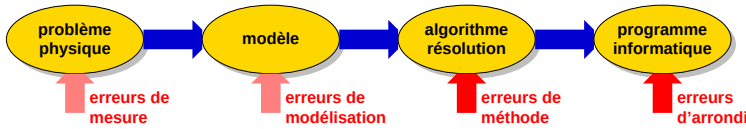
Se produit lors de l'addition de deux nombres ayant des ordres de grandeur très différents où l'on peut "perdre" le plus petit.



On parle même d'absorption catastrophique dans certains cas. Exemple : en simple précision, avec $a = 2^{30}$ et $b = 2^{-30}$ on a alors :

$$a \oplus b = 2^{30} \quad \text{et donc} \quad (a \oplus b) \ominus a = 0$$

Les différentes sources d'erreur



À notre niveau, on peut "seulement" travailler pour proposer des algorithmes numériques avec de plus petites erreurs de méthode et des implantations logicielles qui maîtrisent mieux les problèmes d'arrondi.

En pratique, pour obtenir de bons résultats, il faut concevoir des algorithmes en prenant en compte dès le début les erreurs d'arrondi.

Explosion du vol Ariane 501

Le 4 juin 1996, lors de son premier vol, la fusée européenne Ariane 5 explose 30 secondes après son décollage causant la perte de la fusée et de son chargement estimé à 500 M$.

Après deux semaines d'enquête, un problème est trouvé dans le système de référence inertiel. La vitesse horizontale de la fusée par rapport au sol était calculée sur des flottants 64 bits. Dans le programme du calculateur de bord, il y avait une conversion de cette valeur flottante 64 bits vers un entier signé 16 bits. Malheureusement, rien n'était fait pour tester que cette conversion était bien possible mathématiquement (sans dépassement de capacité). . .

Un petit programme rigolo

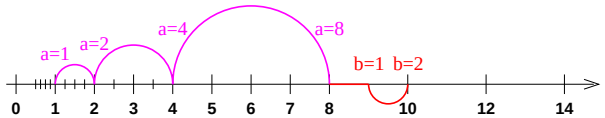
Que fait le programme suivant, dû à Gentleman [3] ?

```
#include <stdio.h>

int main() {
  float a = 1.0, b = 1.0;

  while ( (((a + 1.0) - a) - 1.0) == 0.0)
     a = 2.0 * a;

  while ( (((a + b) - a) - b) != 0.0)
    b = b + 1.0;

  printf("Gentleman : %f \n", b);

  return 0;
}
```

[3] W. M. Gentleman, More on algorithms that reveal properties of floating point arithmetic units, *Communications of the ACM*, vol. 17, n. 5, 1974.

Un petit programme rigolo (suite)

Dans le cas de notre représentation de test $(1, 3, 3)$, on a :

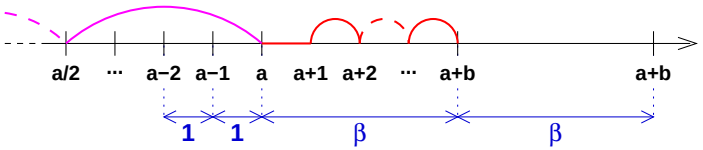


a	b	tests effectués
1.0	1.0	$((a+1.0)-a)-1.0 = ((2.0)-a)-1.0 = (1.0)-1.0 = 0$
2.0	1.0	$((a+1.0)-a)-1.0 = ((3.0)-a)-1.0 = (1.0)-1.0 = 0$
4.0	1.0	$((a+1.0)-a)-1.0 = ((5.0)-a)-1.0 = (1.0)-1.0 = 0$
8.0	1.0	$((a+1.0)-a)-1.0 = ((\underbrace{8.0}_{round(9)})-a)-1.0 = (0.0)-1.0 = -1.0$
8.0	1.0	$((a+b)-a)-b = ((\underbrace{8.0}_{round(9)})-a)-b = (0.0)-b = -1.0$
8.0	2.0	$((a+b)-a)-b = ((10.0)-a)-1.0 = (2.0)-b = 0.0$

Un petit programme rigolo (fin)

Le programme de Gentleman retourne la base utilisée par les unités de calcul flottant du processeur (2 dans le cas de mon PC avec un Pentium III).

Pourquoi ?



- première boucle : recherche de a représentable tel que $a+1$ ne soit pas représentable
- deuxième boucle : recherche de b représentable tel que $a+b$ soit représentable
- le résultat est $\beta = b$ la base interne de l'unité flottante

Partie 3

Addition et représentations redondantes

Points abordés dans cette partie

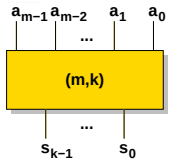
- Addition classique
- Addition rapide
- Représentations redondantes
- Addition redondante

Cellules de base pour l'addition

En plus des portes logiques classiques (et, ou. . .), nous allons utiliser des portes présentant une propriété arithmétique bien utile : le comptage de 1.

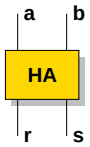
Un compteur (m, k) est une cellule, élémentaire ou non, qui compte le nombre de 1 présents sur ses m entrées et donne le résultat en numération simple de position sur k bits en sortie.

$$\sum_{i=0}^{m-1} a_i = \sum_{j=0}^{k-1} s_j 2^j$$



La cellule demi-additionneur (*half-adder* ou HA) est un compteur (2,2) tandis que la cellule d'addition complète (*full-adder* ou FA) est un compteur (3,2). Ces deux portes sont largement utilisées dans les opérateurs arithmétiques.

La cellule HA



a	b	r	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

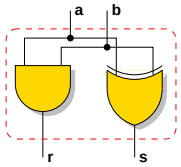
Équation arithmétique :

$$2r + s = a + b$$

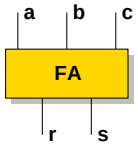
Équations logiques :

$$\begin{aligned} s &= a \oplus b \\ r &= ab \end{aligned}$$

Réalisation pratique du HA :



La cellule FA



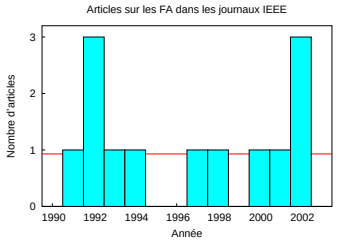
a	b	c	r	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Équation arithmétique :

$$2r + s = a + b + c$$

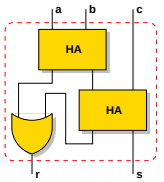
Équations logiques :

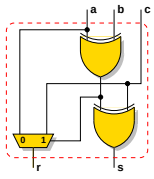
$$\begin{aligned} s &= a \oplus b \oplus c \\ r &= ab + ac + bc \end{aligned}$$

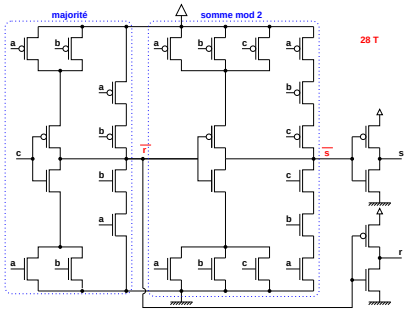


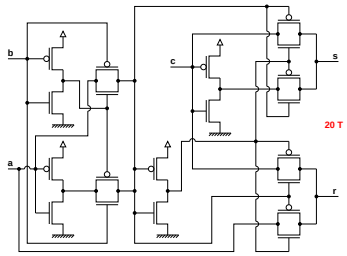
Il existe de nombreuses réalisations pratiques de la cellule FA.

Quelques implantations possibles de la cellule FA



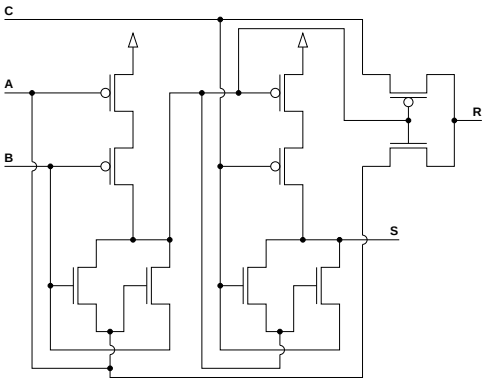






La cellule FA du jour

Solution en 10 transistors[4] :

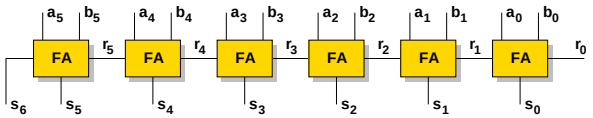


A	B	C	R	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0_w	1
0	1	1	1	0
1	0	0	0_w	1
1	0	1	1_w	0
1	1	0	1	0
1	1	1	1_w	1_w

[4] H. T. Bui, Y. Wang et Y. Jiang. *Design and analysis of low-power 10-transistor full adders using novel XOR–XNOR gates*. IEEE Trans. CaS, jan. 2002.

Additionneur séquentiel

C'est l'additionneur le plus simple. Il est composé de n cellules FA connectées en série.



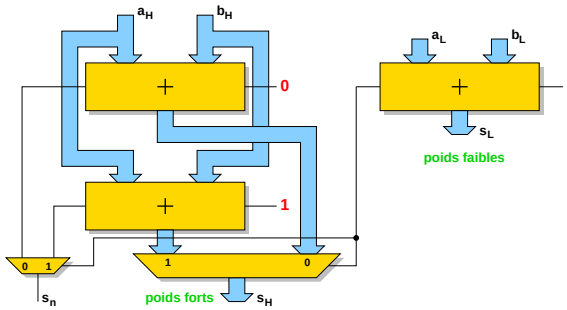
Dans la littérature, on le trouve sous le nom de *Ripple-Carry Adder* (RCA) ou parfois de *Carry-Propagate Adder* (CPA)[5].

	complexité
délai	$O(n)$
surface	$O(n)$

[5] Attention : CPA peut aussi désigner un additionneur non redondant (propage mais ne conserve pas).

Additionneur à sélection de retenue

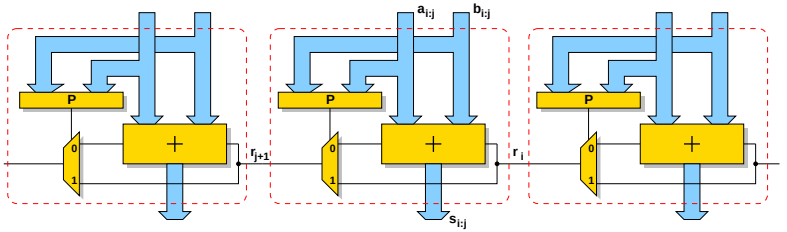
Idée : couper en deux et calculer le bloc des poids forts avec les deux retenues entrantes possibles et sélectionner la bonne sortie avec la retenue sortante des poids faibles (*Carry-Select Adder* ou *Conditional-Sum Adder*).



Version récursive $\longrightarrow$ délai en $O(\log n)$ (mais sortance non bornée).

Additionneur à retenue bondissante

Idée : découper en blocs où chaque bloc permet de détecter rapidement une propagation sur tout le bloc (*Carry-Skip Adder*).



Questions :

- blocs de même taille ?
- taille optimale des blocs ?

Cas des blocs de même taille

Le pire cas d'un additionneur de n bits découpé en blocs de k bits se produit lorsque l'on doit propager du bit 1 jusqu'au bit $n-2$ (générations de retenues aux bits 0 et $n-1$ et propagation au milieu). Soit τ_1 le temps de propagation sur un 1 bit et τ_2 le temps de saut d'un bloc de k bits ($\tau_2 < \tau_1$).

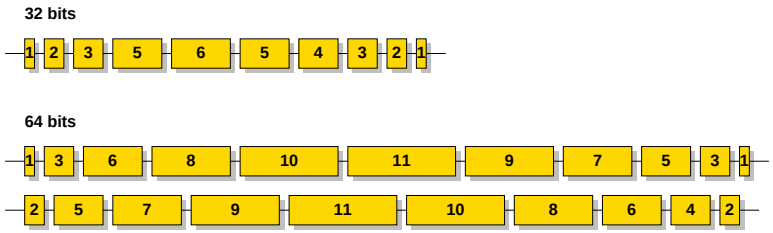
$$\begin{aligned} T(k) &= 2(k-1)\tau_1 + (\frac{n}{k} - 2)\tau_2 \\ T'(k) &= 2\tau_1 - \frac{n\tau_2}{k^2} \\ T''(k) &= \frac{2n\tau_2}{k^3} \end{aligned}$$

$$k_{opt} = \sqrt{\frac{n\tau_2}{2\tau_1}} \quad \longrightarrow \quad T(k_{opt}) = O(\sqrt{n})$$

Cas des blocs de tailles variables

Le problème est compliqué dans le cas général, mais de nombreuses solutions (heuristiques) ont été proposées pour les cas se présentant en pratique.

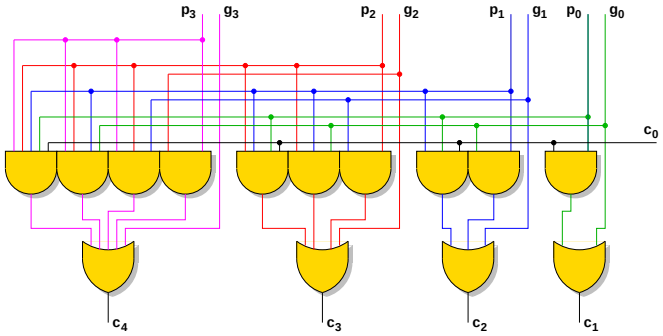
Exemple de solutions dans *A Simple Strategy for Optimized Design of One-Level Carry-Skip Adders*. M. Alioto et G. Palumbo. IEEE Trans. CaS I, jan. 03.



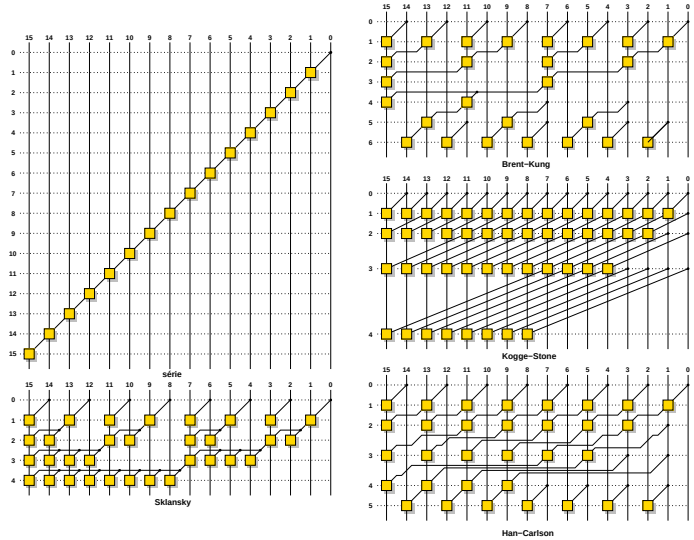
Calcul des retenues pour un CLA 4 bits

CLA = *carry lookahead adder*

$$c_1 = g_0 + p_0 c_0 \qquad c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$$
$$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0 \qquad c_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$$



Quelques additionneurs à préfixe parallèle



Représentations redondantes

Pour aller encore plus vite, on va "tricher" en conservant les retenues. Cela n'a un sens que si l'on effectue plusieurs additions successivement.

En 1961, Avizienis a suggéré de représenter les nombres en base β par l'ensemble de chiffres $\{-\alpha, -\alpha+1, \ldots, 0, \ldots, \alpha-1, \alpha\}$ au lieu des chiffres de $\{0, 1, 2, \ldots, \beta-1\}$ avec $\alpha \leq \beta - 1$.

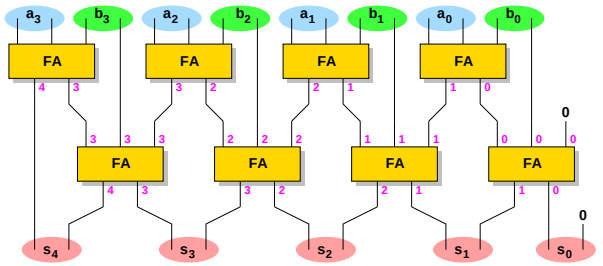
Dans ce système, si $2\alpha + 1 > \beta$ certains nombres ont plusieurs écritures possibles. Par exemple, le nombre 2345 (dans le système usuel) peut s'écrire en base 10 avec l'ensemble de chiffres $\{-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5\}$ selon les codages 2345, 235(-5) ou 24(-5)(-5). On dit alors que le système est redondant.

L'intérêt des systèmes de représentation redondants est qu'il existe dans ces systèmes des algorithmes permettant d'effectuer des additions de façon totalement parallèle, c'est-à-dire sans propagation de retenues.

Additionneur *carry-save*

On représente le nombre A en base 2 avec les chiffres $a_i \in \{0, 1, 2\}$ sur deux fils tels que $a_i = a_{i,c} + a_{i,s}$ où $a_{i,c} \in \{0, 1\}$ et $a_{i,s} \in \{0, 1\}$.

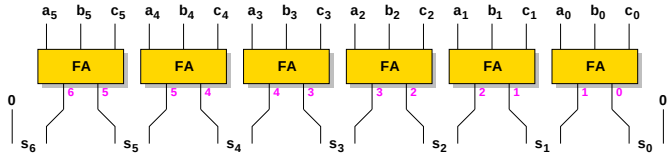
$$A = \sum_{i=0}^{n-1} a_i 2^i = \sum_{i=0}^{n-1} (a_{i,c} + a_{i,s}) 2^i$$



Toutes les sommes sont obtenues dans le délai de 2 cellules FA.

Addition *carry-save* de $k \geq 3$ nombres standards

Soient A, B et C trois nombres en notation non-redondante. On obtient leur somme S en représentation *carry-save* avec l'opérateur suivant ($k = 3$).



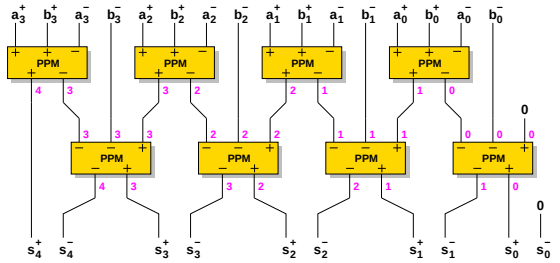
Plus généralement, un arbre *carry-save* à h niveaux permet de réduire au plus $n(h)$ entrées non-redondantes selon le tableau ci-dessous. On a $n(h) = \lfloor 3n(h-1)/2 \rfloor$ et $n(0) = 2$.

h	1	2	3	4	5	6	7	8	9	10	11
$n(h)$	3	4	6	9	13	19	28	42	63	94	141

Additionneur *borrow-save*

On représente le nombre A en base 2 avec les chiffres $a_i \in \{-1, 0, 1\}$ sur deux fils tels $a_i = a_i^+ - a_i^-$ où $a_i^+ \in \{0, 1\}$ et $a_i^- \in \{0, 1\}$.

$$A = \sum_{i=0}^{n-1} a_i 2^i = \sum_{i=0}^{n-1} (a_i^+ - a_i^-) 2^i$$



Toutes les sommes sont obtenues dans le délai de 2 cellules PPM.

Cellule PPM

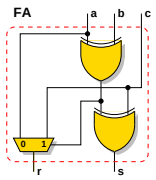
a^+	b^+	c^-	r^+	s^-
0	0	0	0	0
0	0	1	0	1
0	1	0	1	1
0	1	1	0	0
1	0	0	1	1
1	0	1	0	0
1	1	0	1	0
1	1	1	1	1

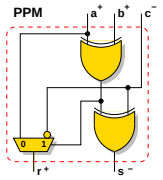
Équation arithmétique :

$$2r^+ - s^- = a^+ + b^+ - c^-$$

Équations logiques :

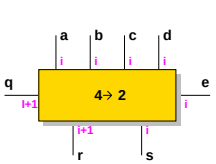
$$\begin{aligned} s &= a^+ \oplus b^+ \oplus c^- \\ r &= a^+ b^+ + a^+ \overline{c^-} + b^+ \overline{c^-} \end{aligned}$$

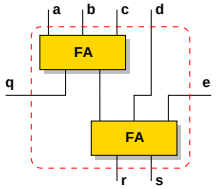


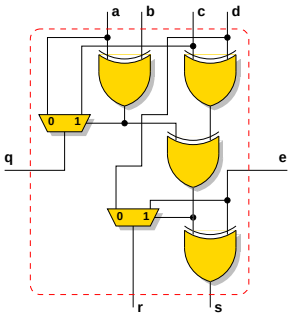


Cellule 4 donne 2

Équation arithmétique : $a + b + c + d + e = 2(r + q) + s$

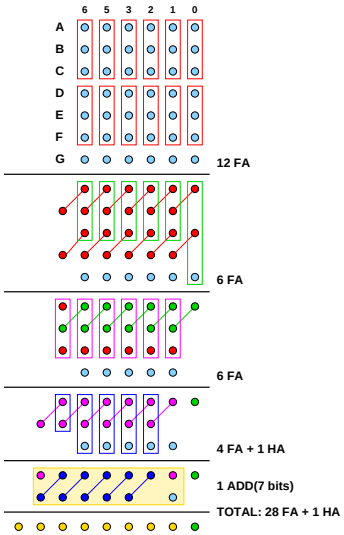




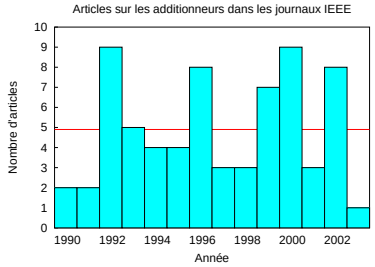


Intérêt : faire des arbres plus réguliers qu'avec la cellule FA (3 donne 2).

Additionneurs multi-opérandes : réduction en points



La recherche sur les additionneurs



- 1994 : NEC, CMOS 0.4 μm et 3.3 V
  ALU basée sur un CLA 32 bits à 500 MHz.
- 2002 : Intel, CMOS 0.13 μm et 1.5 V
  ALU basée sur un Han-Carlson 32 bits à 5 GHz.

Partie 4

Algorithmes de division

Points abordés dans cette partie

- Division simple par additions–décalages
- Division SRT
- Méthode de Newton
- Extensions au calcul de la racine carrée
- Fréquence d'utilisation de la division en machine

Division à la main

On cherche à trouver le quotient q de la division du dividende x par le diviseur d (le reste est r). On a : $x = q \times d + r$.

Exemple : calcul de $123/234$, (résultat exact $\frac{123}{234} = 0.5256410256\ldots$).

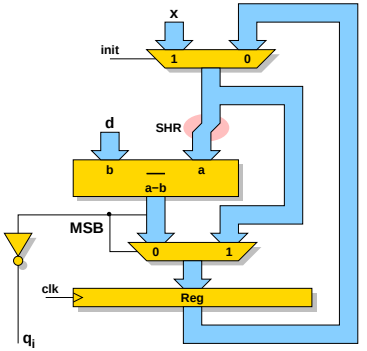
1	2	3	0				2	3	4	
		6	0	0						
		1	3	2	0		.	5	2	5	6	4	···
			1	5	0	0							
				9	6	0							
					2	4							

1	x	234	= 234
2	x	234	= 468
3	x	234	= 702
4	x	234	= 936
5	x	234	= 1170
6	x	234	= 1404
7	x	234	= 1638
8	x	234	= 1872
9	x	234	= 2106
10	x	234	= 2340

On effectue donc l'itération : $x^{(i+1)} = 10x^{(i)} - q_{i+1}d$

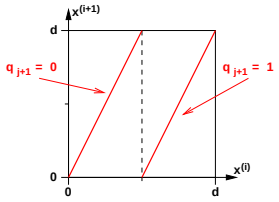
Division restaurante

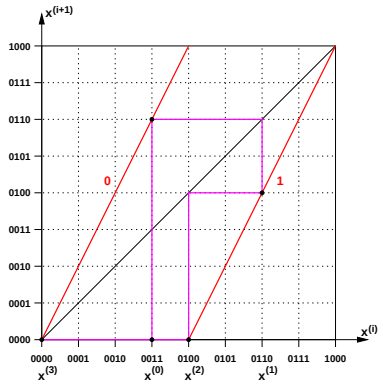
```
1  for i from 1 to n do
2      x ⟵ 2x
3      x ⟵ x − d
4      if x ≥ 0 then
5          q_i ⟵ 1
6      else
7          q_i ⟵ 0
8          x ⟵ x + d
```



Le caractère *restaurant* de cet algorithme vient de l'annulation de l'effet de la ligne 3 par la ligne 8 dans certains cas.

Diagramme de Robertson de la division restaurante





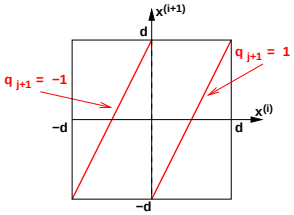
x = 11
d = 1000

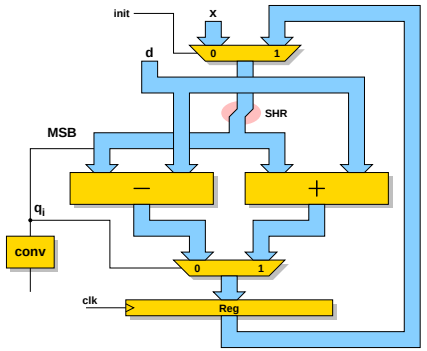
$x^{(0)} = 0011$	$q_0 = 0$
$x^{(1)} = 0110$	$q_1 = 1$
$x^{(2)} = 0100$	$q_2 = 1$
$x^{(3)} = 0000$	$q_3 = 0$

$\frac{3}{8} = 0.375$

Division non restaurante

```
1  for i from 1 to n do
2      x ⟵ 2x
3      if x ≥ 0 then
4          x ⟵ x − d
5          q_i ⟵ 1
6      else
7          x ⟵ x + d
8          q_i ⟵ −1
```



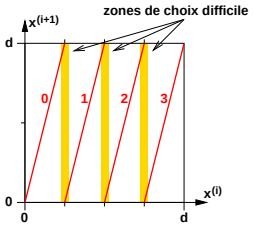


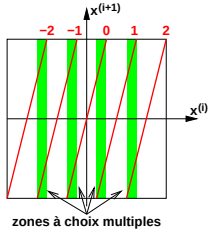
La conversion de $\{-1, 1\}$ vers $\{0, 1\}$ peut se faire à la volée (MSDF) avec un petit opérateur.

Comment aller plus vite ?

Idée : faire des itérations avec des q_i dans une base plus grande.

Problème : faire des comparaisons précises avec plusieurs multiples du diviseur.





Solution : utiliser une représentation redondante pour le quotient.
↪ faire des comparaisons approchées

Division SRT

La méthode SRT proposée par Sweeney, Robertson et Tocher en 1958 est basée sur :

- une représentation redondante des chiffres du quotient en base β (pour simplifier le choix des q_i).
- une représentation redondante des restes partiels (pour accélérer la soustraction, en *borrow-save* par exemple).
- une table permettant de déduire q_{i+1} à partir de quelques bits de poids forts de d et de $x^{(i)}$ (après conversion en non-redondant).

1 $d' \longleftarrow \mathrm{trunc}(d, k_d, \mathrm{MSB})$
2 **for** i from 1 **to** $n/\log_2\beta$ **do**
3 $\quad x'^{(i)} \longleftarrow \mathrm{trunc}(x^{(i)}, k_x, \mathrm{MSB})$
4 $\quad q_{i+1} \longleftarrow \mathrm{T}(d', \mathrm{conv}(x'^{(i)}))$
5 $\quad x^{(i+1)} \longleftarrow \beta x^{(i)} - q_{i+1} \times d$

Architecture générale d'un diviseur SRT

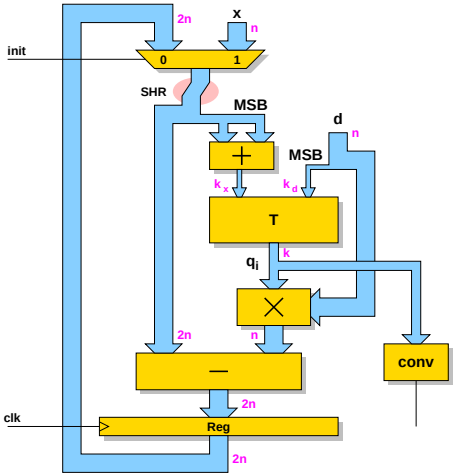
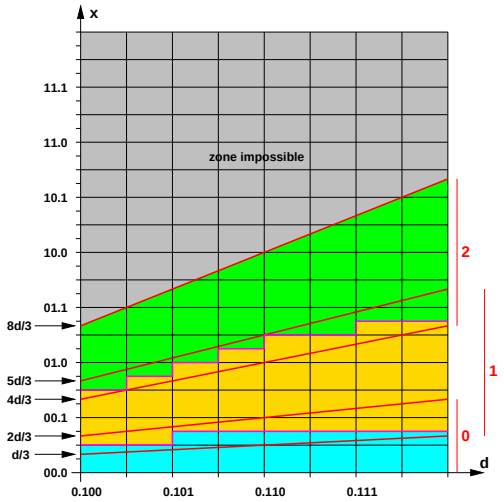


Table pour un diviseur SRT

Diagramme reste-diviseur :
- base $\beta = 4$
- $q_i \in \{-2, -1, 0, 1, 2\}$
- entrée :
  - d sur 3 bits
  - x sur 5 bits

Remarque : Le diagramme est symétrique par rapport à l'axe d.



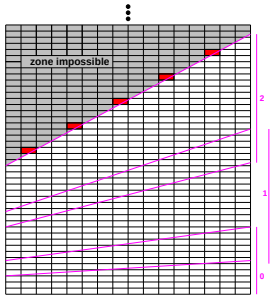
Le bug de la division du Pentium[6]

Exemple de problème en 1994 :

$$\begin{aligned}\frac{4195835.0}{3145727.0} &= 1.333739068902\ldots \quad \text{sur le pentium}\\ &= 1.333820449136\ldots \quad \text{exact}\end{aligned}$$

Diviseur du Pentium : SRT base 4, chiffres du quotient $\{-2,-1,0,1,2\}$, restes partiels en *carry-save*, table avec 5 bits d'entrée pour d et 7 bits pour $x^{(i)}$.

Problème : 5 cases fausses dans la table



[6] RR 95-06 de J.-M. Muller sur ce sujet à `http://www.ens-lyon.fr/LIP/`.

Extensions à la racine carrée

On peut calculer des racines carrées avec un algorithme à additions-décalages proche de celui pour la divison.

Exemple : on cherche $r = \sqrt{c}$ avec $x^{(0)} = c - 1$ et

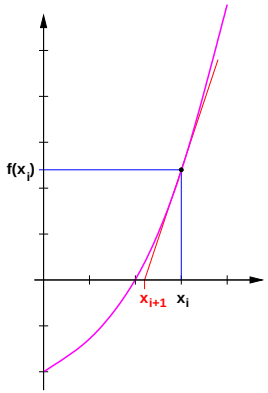
1 **for** i from 1 **to** $n/\log_2 \beta$ **do**
2 $\quad x'^{(i)} \longleftarrow \mathrm{trunc}(x^{(i)}, k_x, \mathrm{MSB})$
3 $\quad r'^{(i)} \longleftarrow \mathrm{trunc}(r^{(i)}, k_r, \mathrm{MSB})$
4 $\quad r_{i+1} \longleftarrow \mathrm{T}(\mathrm{conv}(r'^{(i)}), \mathrm{conv}(x'^{(i)}))$
5 $\quad r^{(i+1)} \longleftarrow r^{(i)} + r_{i+1}\beta^{-i-1}$
6 $\quad x^{(i+1)} \longleftarrow \beta x^{(i)} - 2r^{(i)}r_{i+1} - r_{i+1}^2\beta^{-i-1}$

Méthode de Newton

On cherche une racine de l'équation $f(x) = 0$ (où f est supposée continûment dérivable) en utilisant la suite (x_i) définie par :

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Si x_0 est suffisament proche d'une racine simple α de f, alors la suite (x_i) converge quadratiquement vers α (le nombre de chiffres significatifs double à chaque étape).



$f(x) = \dfrac{x^2}{2} - 2$

i	0	1	2	3	4
x_i	3	2.166666667	2.006410256	2.000010240	2.000000000

Méthode de Newton pour la division

Pour trouver le quotient $q = a/d$ on va procéder en 2 étapes :

1. calcul de $t = 1/d$ à l'aide de la fonction $f(x) = \frac{1}{x} - d$. On doit donc calculer l'itération suivante :

$$\begin{aligned}x_{i+1} &= x_i - \frac{\frac{1}{x_i} - d}{-\frac{1}{{x_i}^2}}\\ &= x_i + x_i - d{x_i}^2\\ &= x_i(2 - dx_i)\end{aligned}$$

   Le coût de chaque itération est de 2 multiplications et 1 addition.

   La valeur x_0 est obtenue par une lecture dans une petite table.

2. calcul de $q = t \times a$

Exemple de l'Itanium

Utilisation du multiplieur–accumulateur flottant sur des registres de 82 bits. Initialisation de la méthode de Newton par une lecture de table qui donne une approximation de $1/d$ à 8.886 bits près.

Exemple d'algorithme pour la simple précision (mantisse de 23 bits) :

1 $y_0 \longleftarrow \mathrm{T}(d)$
2 $q_0 \longleftarrow (a \times y_0)_{rn}$
3 $e_0 \longleftarrow (1 - b \times y_0)_{rn}$
4 $q_1 \longleftarrow (q_0 + e_0 \times q_0)_{rn}$
5 $e_1 \longleftarrow (e_0 \times e_0)_{rn}$
6 $q_2 \longleftarrow (q_1 + e_1 q_1)_{rn}$
7 $e_2 \longleftarrow (e_1 \times e_1)_{rn}$
8 $q_3 \longleftarrow (q_2 + e_2 \times q_2)_{rn}$
9 $q'_3 \longleftarrow \mathrm{round}(q_3)$

Méthode de Newton pour la racine carrée

Première idée : utiliser Newton avec $f(x) = x^2 - c$, on a alors :

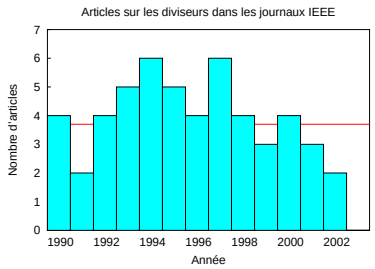
$$x_{i+1} = x_i - \frac{{x_i}^2 - c}{2x_i} = x_i - \frac{x_i}{2} + \frac{c}{2x_i} = \frac{1}{2}\left(x_i + \frac{c}{x_i}\right)$$

Seconde idée : utiliser Newton avec la fonction $f(x) = \frac{1}{x^2} - c$ qui a pour solution $\frac{1}{\sqrt{c}}$ (ensuite on multiplie par c pour avoir $\sqrt{c}$).

$$x_{i+1} = x_i - \frac{\frac{1}{x^2} - c}{-\frac{2}{{x_i}^3}} = x_i + \frac{x_i - c{x_i}^3}{2} = \frac{x_i}{2}\left(3 - c{x_i}^2\right)$$

Chaque itération fait intervenir 3 multiplications et une addition.

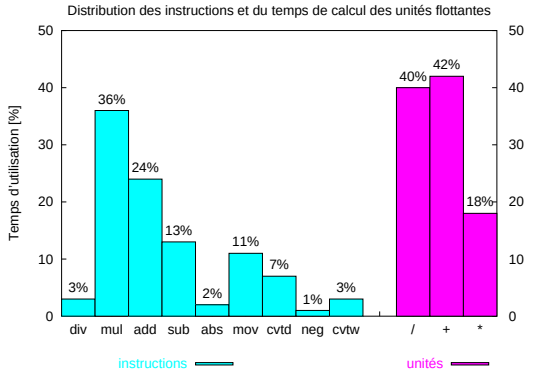
La recherche sur les diviseurs



- SRT base 4 dans le Pentium ou SRT base 8 dans l'Ultra (3 étages de base 2 par itération).
- Newton dans Itanium, Pentium 4 (table pour initialisation + routine logicielle).

La division, une opération peu utilisée ? Oui, mais. . .

Rapport technique[7] de S. Oberman et M. Flynn : "*Design issues in floating-point division*", CSL-TR-94-647 Stanford University.



[7] SPECfp92 sur DECstation avec un MIPS R3000 (latences : 2c add, 5c mul, 19c div), compil. O3.

Pour en savoir plus. . .

Division and Square Root : Digit-Recurrence Algorithms and Implementations

Milos Ercegovac et Tomas Lang

1994
Kluwer
ISBN : 0–7923–9438–0


Division Algorithms and Implementations

Division Algorithms and Implementations

Stuart Oberman et Micheal Flynn

1997
IEEE Transactions on Computers
Vol. 46, No. 8, pp 833–854

Fin

Questions ?

Pour me contacter :

- arnaud.tisserand@ens-lyon.fr
- http://perso.ens-lyon.fr/arnaud.tisserand/
- Laboratoire LIP. ENS Lyon. 46 allée d'Italie. F-69364 Lyon cedex 07.

Merci.

Annexes

Représentation des entiers relatifs

Il existe différentes représentations possibles pour les entiers signés :

- signe et magnitude (valeur absolue)

$$A = (s_a a_{n-2} \dots a_1 a_0) = (-1)^{s_a} \times \sum_{i=0}^{n-2} a_i 2^i$$

- complément à (la base) deux

$$A = (a_{n-1} a_{n-2} \dots a_1 a_0) = -a_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

- biaisée (souvent $B = 2^{n-1} - 1$)

$$A = A_{math} + B$$

- . . .

Représentation des entiers relatifs (suite)

entier	représentations		
	signe/magnitude	complément 2	biaisée (B=7)
−8	—	1000	—
−7	1111	1001	0000
−6	1110	1010	0001
−5	1101	1011	0010
−4	1100	1100	0011
−3	1011	1101	0100
−2	1010	1110	0101
−1	1001	1111	0110
0	0000	0000	0111
1	0001	0001	1000
2	0010	0010	1001
3	0011	0011	1010
4	0100	0100	1011
5	0101	0101	1100
6	0110	0110	1101
7	0111	0111	1110
8	—	—	1111

Trace de l'exécution du programme d'intégration

```
      10 8.410365e-02
      20 3.964663e-02
      50 1.533222e-02
     100 7.583203e-03
     200 3.769578e-03
     500 1.501383e-03
    1000 7.505436e-04
    2000 3.938694e-04
    5000 1.571794e-04
   10000 6.193113e-05
   20000 2.453346e-04
   50000 2.440791e-04
  100000 2.556424e-04
  200000 2.688150e-04
  500000 2.406953e-03
 1000000 9.144427e-03
 2000000 1.050532e-02
 5000000 3.374992e-03
10000000 2.150589e-02
20000000 3.068528e-01
50000000 1.931472e-01
```


Trace de l'exécution du programme de Gentleman

```
a = 1.000000   b = 1.000000
--- boucle1-----------------------------------------
64 itération(s)

                    a = 18446744073709551616.000000
              a + 1.0 = 18446744073709551616.000000
        (a + 1.0) - a = 0.000000
((a + 1.0) - a) - 1.0 = -1.000000

--- boucle2-----------------------------------------
                b = 1.000000
            a + b = 18446744073709551616.000000
      (a + b) - a = 0.000000
((a + b) - a) - b = -1.000000
1 itération(s)

                b = 2.000000
            a + b = 18446744073709551616.000000
      (a + b) - a = 2.000000
((a + b) - a) - b = 0.000000

Gentleman : 2.000000
```