Unités de calcul flottant

Arnaud Tisserand

LIRMM, CNRS–Univ. Montpellier 2
Équipe ARITH

ARCHI07, 19–23 mars, 2007, Boussens

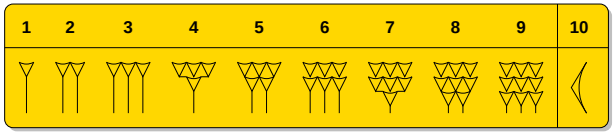
Plan

Partie I

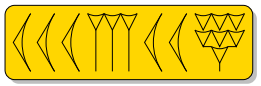
Introduction

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 :



$= 33 \times 60 + 27 = 2007$

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...

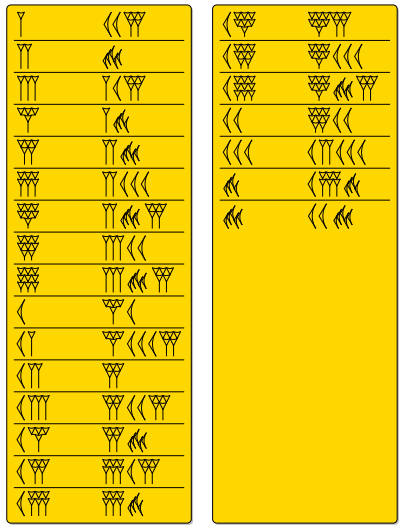


Illustration de la table de multiplication par 25 trouvée à Suse et datée du II^e millénaire av. J.-C (conservée au Musée du Louvre).

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.

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, crypto)
- support logiciel pour les calculs de base :
  - bibliothèques mathématiques de base (libm)
  - bibliothèques de fonctions élémentaires (sin, cos, exp, log, ...)
  - 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

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 ...), 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

Exemples de sujets de recherche dans l'équipe Arith au LIRMM :

- étude des propriétés des représentations des nombres et des opérations
- opérateurs arithmétiques pour les circuits intégrés numériques
- opérateurs de cryptographie
- unités flottantes et logarithmiques
- opérateurs arithmétiques matériels à basse consommation d'énergie
- arithmétique pour les corps finis
- liens entre la géométrie discrète et l'arithmétique
- liens entre la physique et l'arithmétique
- ...

Besoins dans les processeurs/architectures évolués

- Opérateurs arithmétiques capables de fonctionner à de hautes fréquences
- Opérateurs arithmétiques à basse consommation d'énergie pour les applications embarquées et les appareils portables
- Prendre en compte les nouvelles contraintes technologiques
- Ajout de nouvelles fonctionnalités de calcul, exemple : FMA (*fused multiply and add*), opérateurs de cryptographie, unités graphiques
- Sujets de recherche actuels sur les opérateurs arithmétiques matériels :
  - nouvelles fonctionnalités
  - basse consommation d'énergie
  - opérateurs reconfigurables
  - opérateurs pour la cryptographie/signature
  - ...

Systèmes de numération

Critères de classification :

- propriétés des nombres représentés
  - entiers $\mathbb{N}, \mathbb{Z}$
  - rationnels $\mathbb{Q}$
  - réels $\mathbb{R}$
  - ...
- propriétés du système de représentation
  - positionnel ou non
  - redondant ou non
  - précision fixe ou arbitraire
  - complétude (dans un ensemble fini)
  - ...

Un système de numération c'est (au moins) deux choses :

- un codage des données
- un ensemble de règles d'interprétation du codage

Représentations 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

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

Représentations des réels

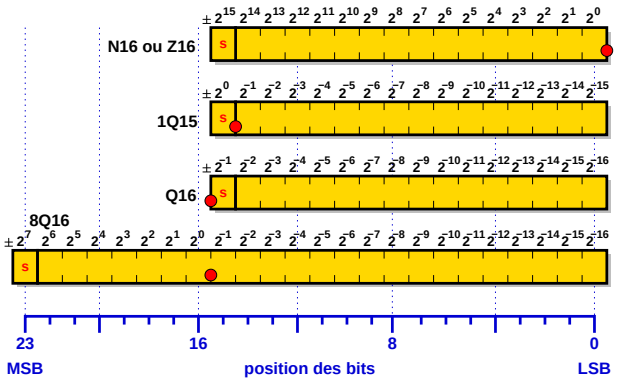
Représentations en machine ou dans les circuits :

- virgule fixe
- virgule flottante
- système logarithmique
- ...

Important : du fait de la précision fixe, on a une approximation des nombres réels.

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

Virgule 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.

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.

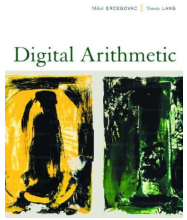
Applications en traitement du signal et en contrôle numérique. Il y avait même un projet européen pour concevoir un processeur avec des unités de calcul 32 bits en système logarithmique.

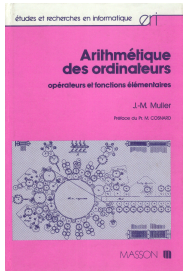
Pour en savoir plus...

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
- Livre collectif : *Calcul et arithmétique des ordinateurs (Traité IC2)*, Hermes Sciences, 2004
- Livre collectif : *Qualité des Calculs sur Ordinateur*, Masson, 1997
- Numéro spécial collectif de la revue *Réseaux et systèmes répartis, calculateurs parallèles* sur l'*arithmétique des ordinateurs*, Hermes, 2001
- Document de Jean Vuillemin : *Les Langages Numériques*, `http://www.di.ens.fr/~jv/HomePage/pdf/langnum.pdf`
- Un site web sur les bouliers : `http://www.ee.ryerson.ca:8080/~elf/abacus/history.html`

Partie II

Représentation flottante

Représentation virgule 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 ?

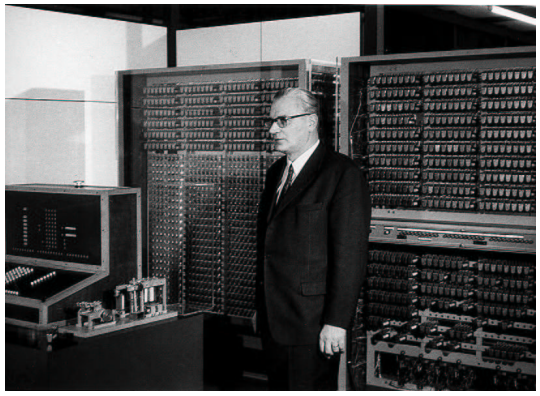
A l'époque, les constructeurs ne s'intéressent qu'aux formats de stockage des données et pas beaucoup aux propriétés mathématiques des unités de calcul...

Question histoire

De quand date le premier ordinateur avec des flottants ?

Réponse :

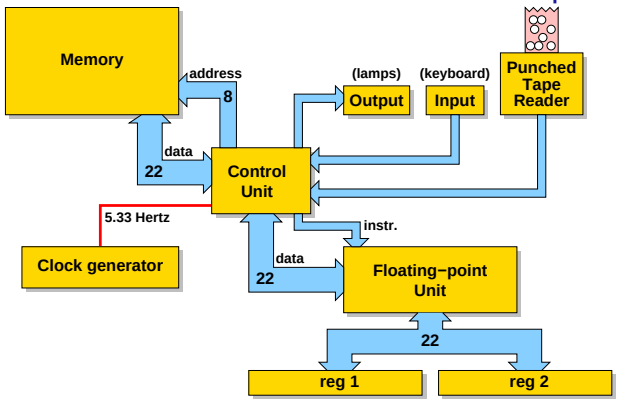
Le Z3 est créé en 1941 par Konrad Zuse (1910–1995) à Berlin.



photographie de la version reconstruite en 1961

Source : `http://www.epemag.com/zuse/`

Le Z3 : architecture et caractéristiques



Taille	5 m × 2 m × 0.8 m
Poids	≈ 1000 kg
Fréquence	5.33 Hz
Technologie	électrique à relais (num. : 600, mém. : 1400)
Consommation	≈ 4000 W

Le Z3 : format des données et unité flottante

Format flottant :

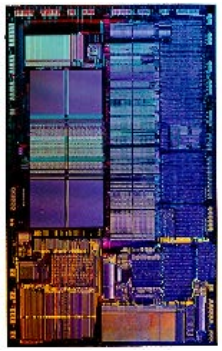
- 1 bit de signe
- 7 bits d'exposant représenté en complément à deux
  - exposant à -64 $\longrightarrow$ valeur 0
  - exposant à 63 $\longrightarrow$ infini
- 14 bits de mantisse + 1 bit implicite

Unité flottante :

- addition/soustraction en 3 cycles (0.6 s/op)
- multiplication en 16 cycles (3.0 s/op)
- division en 18 cycles (3.4 s/op)
- racine carrée (temps variable)

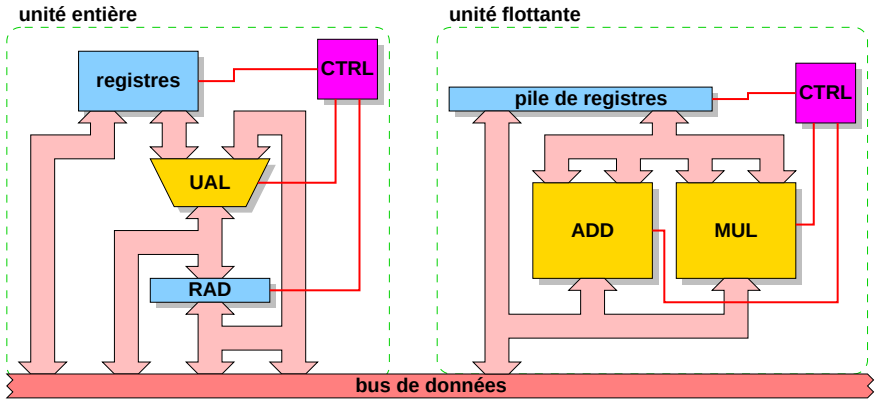
Autres instructions : lecture clavier, affichage, *load*, *store*.

Processeur Intel 486 : caractéristiques



- processeur 32 bits
- coprocesseur arithmétique intégré
- commercialisation en 1989
- fréquences : 50, 33 et 25 MHz
- nombres de transistors : 1 200 000
- technologie : CMOS 0.8 ou 1.0 μm
- surface : 81 mm^2
- alimentation : 5 V
- boitier : 168 PGA
- pipeline : 5 étages
- cache L1 : 8 KB (4w SA, WT)
- UAL : 1

Processeur Intel 486 : architecture



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*

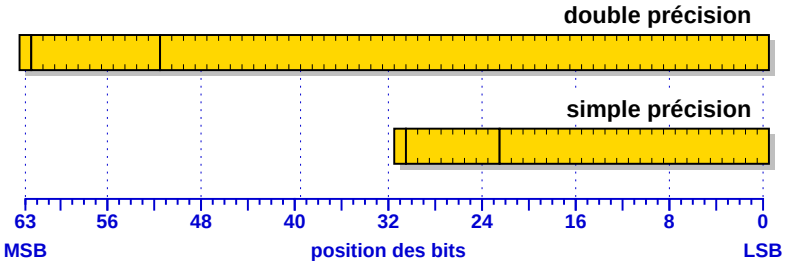
Objectifs de la norme IEEE-754

- 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 :

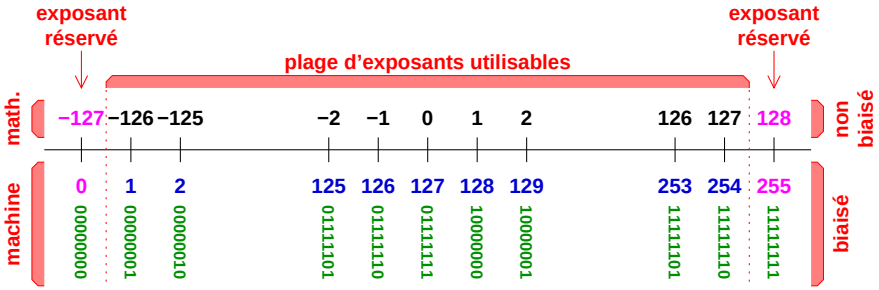
$$e_b = e + b$$

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

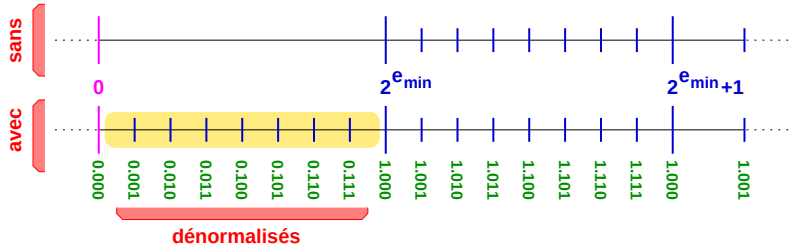
- Not a Number : `NaN`
  Permet de représenter le résultat d'une opération invalide telle que $0/0$, $\sqrt{-1}$ ou $0 \times \infty$. `NaN` est codé en utilisant le plus grand exposant possible et une fraction non-nulle. Les `NaN` se propagent dans les calculs.
  $$e = e_{max} + 1 \qquad \text{et} \qquad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

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 $e_b = 0$) :

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

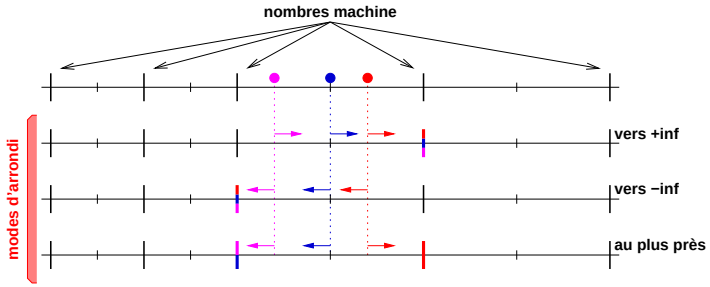
IEEE-754 : modes d'arrondis

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)



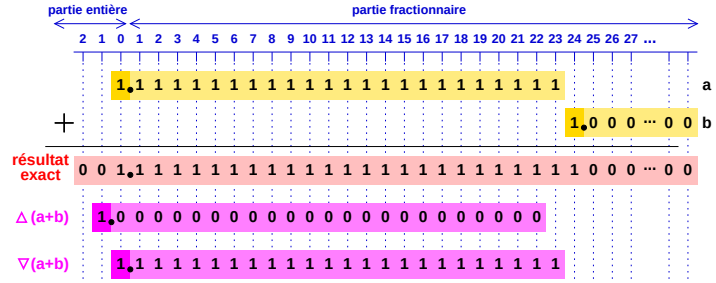
Propriété d'arrondi correct : soient a et b deux nombres machine, $\circledcirc$ une des opérations $(+, -, \times, \div)$ et $\diamond$ le mode d'arrondi choisi (parmi les 4 modes IEEE). Le résultat fourni lors du calcul de $(a \circledcirc b)$ doit être $\diamond(a \circledcirc_{th} b)$.

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

```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 = 2.00000000000000000000000000000000000000e+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 = 1.99999988079071044921875000000000000000e+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.

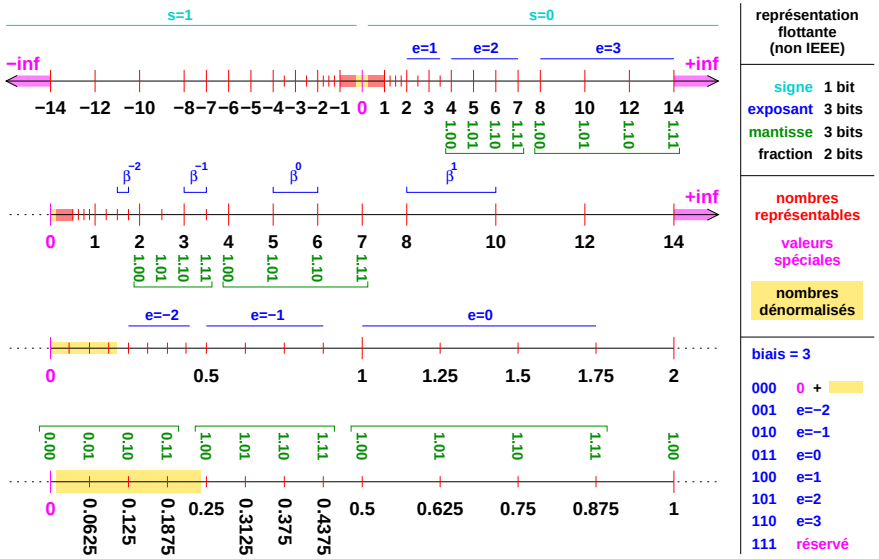
IEEE-754 : résumé

	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

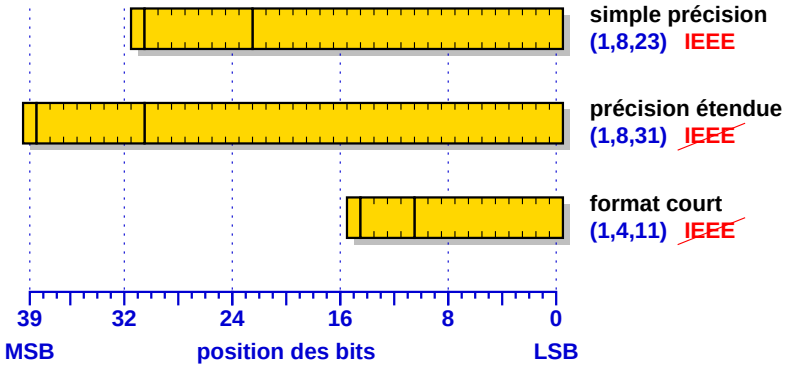
valeur	simple précision	double précision
+ grand normalisé > 0	$3.40282347 \times 10^{38}$	$1.7976931348623157 \times 10^{308}$
+ petit normalisé > 0	$1.17549435 \times 10^{-38}$	$2.2250738585072014 \times 10^{-308}$
+ grand dénormalisé > 0	$1.17549421 \times 10^{-38}$	$2.2250738585072009 \times 10^{-308}$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



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/`

Calcul flottant

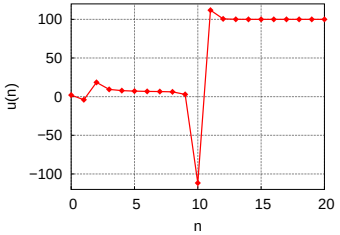
Calculer avec des flottants demande de faire attention à certains problèmes potentiellement dangereux...

Source principale des problèmes : approximation du fait de la précision finie.

Regardons quelques exemples.

Suite de J.-M. Muller

$$\begin{cases} u_0 &= 2 \\ u_1 &= -4 \\ u_{n+1} &= 111 - \frac{1130}{u_n} + \frac{3000}{u_n u_{n-1}} \end{cases}$$



n	u_n programme C	n	u_n programme C
0	2.000000000000000	9	2.913589954376221
1	-4.000000000000000	10	-111.709793090820312
2	18.500000000000000	11	111.898239135742188
3	9.378377914428711	12	100.661544799804688
4	7.801147460937500	13	100.040603637695312
5	7.154346466064453	14	100.002494812011719
6	6.805830478668213	15	100.000152587890625
7	6.578579425811768	16	100.000007629394531
8	6.235515594482422	17	100.000000000000000

Et pourtant, (u_n) converge théoriquement vers 6 !

Exemple de Shampine et Watts (1/2)

On cherche la solution de l'équation différentielle suivante :

$$\begin{cases} f'(x) &= 10(f(x) - x^2) \\ f(0) &= 0.02 \end{cases}$$

Résolution numérique par la méthode de Runge-Kutta d'ordre 4 (et 100000 points) sur un PentiumIV :

	valeur		
$f(x)$	prg. C (*float*)	prg. C (*double*)	exacte
$f(1)$	1.07524	1.22	1.22
$f(2)$	1018.31	4.42	4.42
$f(3)$	-1.4026e+07	9.66711	9.62
$f(4)$	4.97446e+11	-250.367	16.82
$f(5)$	-1.91225e+14	1.45778e+07	26.02
$f(6)$	-7.33462e+19	-1.22707e+11	37.22

Exemple de Shampine et Watts (2/2)

La solution de

$$\begin{cases} f'(x) &= 10(f(x)-x^2) \\ f(0) &= 0.02 \end{cases}$$

est

$$f(x) = \frac{1}{50} + \frac{1}{5}x + x^2$$

Celle de

$$\begin{cases} f'(x) &= 10(f(x)-x^2) \\ f(0) &= c \end{cases}$$

est

$$f(x) = \frac{1}{50} + \frac{1}{5}x + x^2 + (c - \frac{1}{50})e^{10x}$$

La solution dépend donc très fortement des conditions initiales.

Maple : *BUG # 1542*

Bug présent sur la version "`Maple 6.01, IBM INTEL NT, Jun 9 2000 Build ID 79514`".

entrée	résultat Maple	valeur théorique
1/21474836480;	`Error, division by zero`	1/21474836480
214748364810;	10	214748364810
21474836481.234;	1.234	21474836481.234
2147483648;	−2147483648	2147483648

Cause : problème dans la lecture des nombres entiers (*integer parsing*) : petits entiers avec un format du processeur et grands entiers avec un format interne, erreur dans le mode de sélection du format.

Source : `http://maple.bug-list.org/`

Maple : bug de la factorielle (version 7)

Entrée :

```
> 1001!/1000!,1002!/1000!,1002!/1001!,1003!/1000!,1003!/1001!;

                              1, 1, 1, 1, 1
```

Comportement corrigé :

```
> 1001!/1000!,1002!/1000!,1002!/1001!,1003!/1000!,1003!/1001!;

            1001, 1003002, 1002, 1006011006, 1005006
```

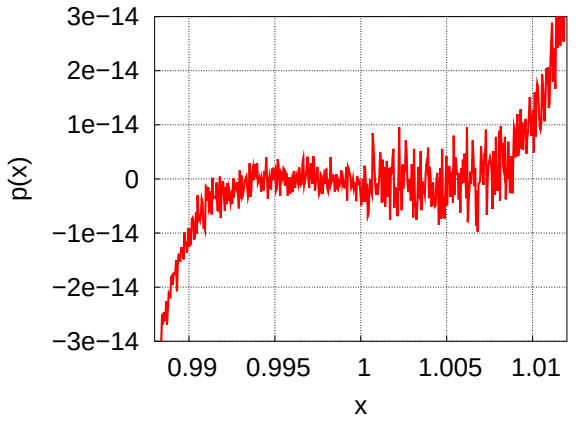
Il y a une erreur dans le code de la fonction factorielle de la version 7 de Maple (il y des correctifs possibles pour cette version).

Source : `Maple User Group Answers` (`http://www-math.math.rwth-aachen.de/MapleAnswers/`)

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.

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



Augmenter la précision : une stratégie fiable ?

Exemple de S. Rump :

$$f(a,b) = 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b}$$

Sur un ordinateur IBM 370, on a :

précision	valeur calculée
simple	1.172603
double	1.1726039400531
étendue	1.172603940053178

Mais ces résultats très proches sont totalement faux ! La bonne valeur est -0.827396059946821368141 16...

Petits problèmes numériques : $I_n = nI_{n-1} - \frac{1}{e}$ (1/4)

Problème : approcher numériquement $I_n = \int_0^1 x^n e^{-x} dx$ pour $n = 50$.
Méthode : utiliser la récurrence $I_n = nI_{n-1} - \frac{1}{e}$ avec $I_0 = 1 - \frac{1}{e}$

n	programme C	valeur exacte
1	0.2642411177	0.26424111765711535680895245
2	0.1606027941	0.16060279414278839202238114
3	0.1139289413	0.11392894125692285447161967
4	0.08783632386	0.08783632385624909629095493
5	0.07130217811	0.07130217810980315985925092
6	0.05993362749	0.05993362748737663755998179
7	0.05165595124	0.05165595124019414132434878
8	0.04536816875	0.04536816875011080899926653
9	0.04043407757	0.04043407757955495939787503
10	0.0364613345	0.03646133462410727238322656
20	-82.17689174	0.01835046769725620632614475
30	-8.961529476e+15	0.01224950295785890695334039
40	-2.756557975e+31	0.00919138801539721301175301
50	-1.027535737e+48	0.00735470679580018886393671

Petits problèmes numériques : $I_n = nI_{n-1} - \frac{1}{e}$ (2/4)

En intégrant par parties I_n, on a :

$$\begin{aligned} I_n &= \int_0^1 x^n e^{-x} dx \\ &= [x^n(-e^{-x})]_0^1 - \int_0^1 nx^{n-1}(-e^{-x})dx \\ &= (-e^{-1} - 0) + n\int_0^1 x^{n-1}e^{-x}dx \\ &= -\frac{1}{e} + nI_{n-1} \end{aligned}$$

Rappel :

$$\int f'(x)g(x)dx = f(x)g(x) - \int f(x)g'(x)dx$$

Petits problèmes numériques : $I_n = nI_{n-1} - \frac{1}{e}$ (3/4)

Question : pourquoi un tel comportement numérique ?

Réponse : en passant de I_{n-1} à I_n, l'erreur numérique (accumulation des erreurs de représentation et des premiers calculs) est multipliée par n. L'erreur numérique sur l'évaluation de I_n croit donc comme $n!$.

Rq : 50! = 30414093201713378043612608166064768844377641568960512000000000000

En transformant un peu la méthode, on a...

Petits problèmes numériques : $I_n = nI_{n-1} - \frac{1}{e}$ (4/4)

Récurrence rétrograde : $I_{n-1} = \frac{I_n + \frac{1}{e}}{n}$ en connaissant I_p avec $p >> n$.
Gain : en partant de I_p, l'erreur sur I_n sera proportionnelle à $\frac{50!}{p!}$.

n	$I_p = 0.5$	$I_p = 1.0$	$I_p = 12345.0$
60	0.5	1	12345
59	0.01446465735	0.02279799069	205.7561313
58	0.00648040845	0.006621651387	3.493627301
57	0.006454480166	0.006456915389	0.06657770245
56	0.006567261778	0.006567304501	0.007622055151
55	0.006686548267	0.00668654903	0.006705383863
54	0.006810290717	0.006810290731	0.006810633182
53	0.006938698739	0.006938698739	0.006938705081
52	0.007072040376	0.007072040376	0.007072040495
51	0.007210605414	0.007210605414	0.007210605417
50	0.007354706796	0.007354706796	0.007354706796

Rappel : valeur exacte 0.00735470679580018886393671...
Source : J.-M. Muller

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)

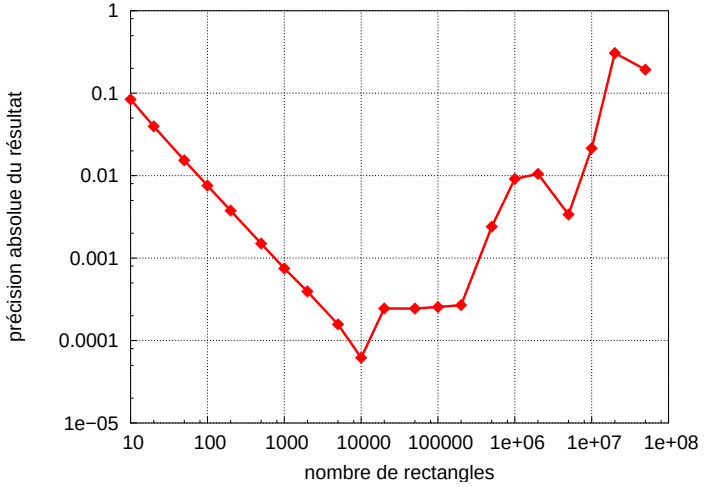
Grâce à la norme IEEE, en l'absence de dépassement de capacité (vers $+\infty$ ou vers 0) et de division par 0, avec x et y deux nombres machine on a :

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

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

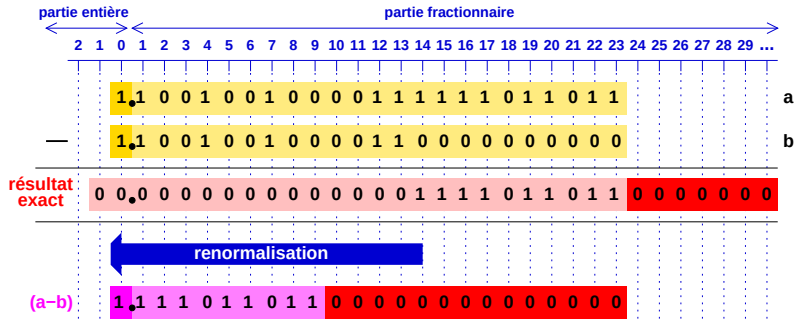
A chaque arrondi, il est possible de perdre un peu de précision, on parle d'erreur d'arrondi. Même si une opération isolée retourne le meilleur résultat possible (l'arrondi du résultat exact), une suite de calculs peut conduire à d'importantes erreurs du fait du cumul des erreurs d'arrondi. Précision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

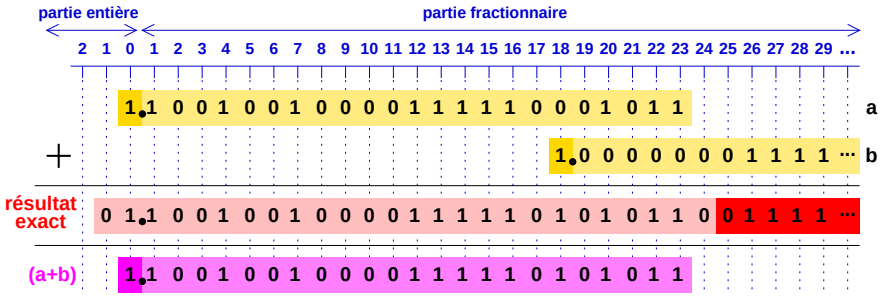
Se produit lors de la soustraction de deux nombres très proches.



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 :

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

Exemple du phénomène d'absorption

Calculer numériquement, pour de grandes valeurs de N, la somme :

$$\sum_{i=1}^{N} \frac{1}{i}$$

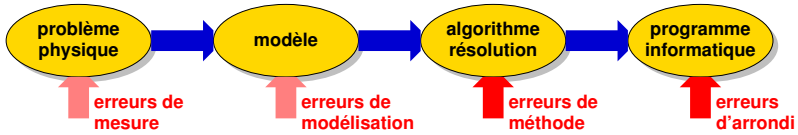
Résultats d'un programme C (flottant SP) sur un processeur Pentium4 :

ordre	N			
	100000	1000000	10000000	100000000
$1 \rightarrow N$	1.209085e+01	1.435736e+01	1.540368e+01	1.540368e+01
$N \rightarrow 1$	1.209015e+01	1.439265e+01	1.668603e+01	1.880792e+01
exacte	1.209015e+01	1.439273e+01	1.669531e+01	1.899790e+01

En rouge les chiffres faux.

Source : J.-M. Muller

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.

Des erreurs numériques qui causent des catastrophes

- Le 25 février 1991, à Dharan en Arabie Saoudite, un missile (anti-missile) Patriot américain loupe un missile Scud iraquien. Ceci provoqua la mort de 28 personnes et plus d'une centaine de blessés.
- Perte d'une plate-forme pétrolière en mer du nord, au large de la Norvège, le 23 août 1991 : coût estimé à 700 M$.
- En 1982 à la bourse de Vancouver, les calculs sur un indice nouveau, de valeur initiale 1000.0, sont tronqués (plutôt qu'arrondis). Après 22 mois la valeur de l'indice calculé est de 520 au lieu de 1098.892...
- Liens web :
  - `http://www.ima.umn.edu/~arnold/disasters/disasters.html`
  - `http://ta.twi.tudelft.nl/nw/users/vuik/wi211/disasters.html`
  - `http://www5.in.tum.de/~huckle/bugse.html`

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

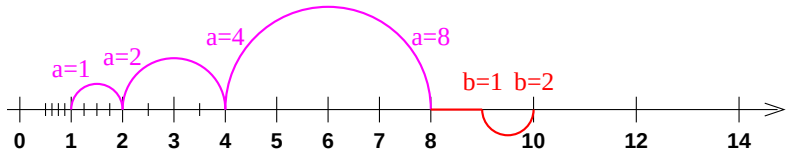
Que fait le programme suivant, dû à Gentleman [3] ?

```
#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)

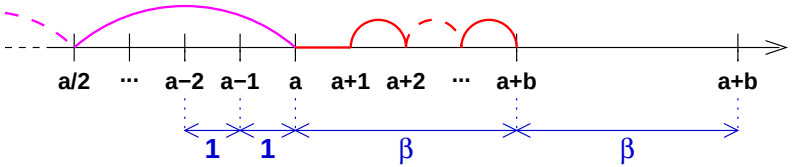
Dans le cas de notre représentation de test $(1, 3, 3)$, on a :



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	$((a+b)-a)-b = ((10.0)-a)-b = (2.0)-b = 0.0$

Un petit programme rigolo (fin)

Le programme de Gentleman retourne la base utilisée par les unités de calcul flottant du processeur (2 pour mon PC avec un Pentium IV). 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 $b = \beta$ la base interne de l'unité flottante

Partie III

Unités flottantes

Addition

Décomposition du problème en deux parties :

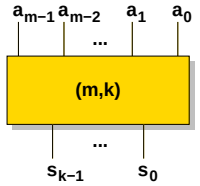
- addition des mantisses (addition d'entiers)
  - cellules de base
  - principaux types d'additionneurs
- gestion de la virgule
  - technique de base
  - optimisations
  - exemple de répartition des coûts

Cellules de base pour l'addition

En plus des portes logiques de base, nous allons utiliser des portes présentant une propriété arithmétique bien utile : le comptage de 1.

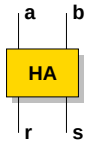
Un compteur (m, k) est une cellule, élémentaire ou non, qui compte le nombre de 1 présents sur ses m entrées et donne le résultat en numération simple de position sur k bits en sortie.

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



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

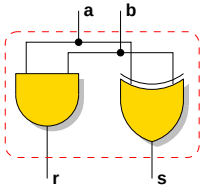
Équation arithmétique :

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

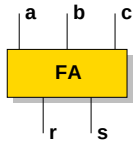
Équations logiques :

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

Réalisation pratique du HA :



La cellule FA



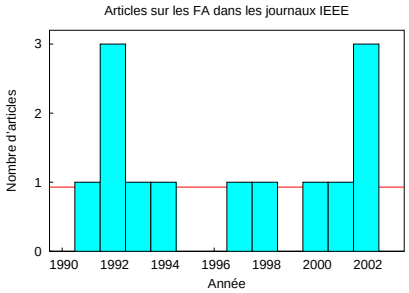
a	b	c	r	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Équation arithmétique :

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

Équations logiques :

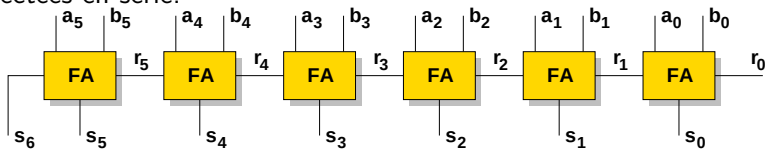
$$\begin{aligned} s &= a \oplus b \oplus c \\ r &= ab + ac + bc \end{aligned}$$



Il existe de nombreuses réalisations pratiques de la cellule FA.

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)[4].

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

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

Propagation et génération de retenue

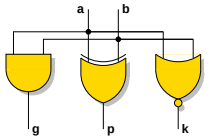
Autres façons de voir la table de vérité de la cellule FA :

a	b	r	s	remarque
0	0	0	c	génération
0	1	c	$\overline{c}$	propagation
1	0	c	$\overline{c}$	propagation
1	1	1	c	génération

a, b	r	s	remarque
$a = b$	a	c	génération
$a \neq b$	c	$\overline{c}$	propagation

Dans certaines références bibliographiques, on parle d'absorption (*kill*) dans le cas particulier de la génération d'une retenue sortante à 0 ($a = b = 0$).

$$\begin{aligned} p &= a \oplus b \\ g &= ab \\ k &= \overline{a}\overline{b} = \overline{a+b} \end{aligned}$$



Chaînes de retenues dans l'additionneur séquentiel

Du fait de la dépendance linéaire entre les cellules, il y a une grande variabilité du temps de calcul en fonction de la longueur de la plus grande chaîne de propagation de retenue.

- pire cas en n propagations (ex : $000000 + 111111$)
- meilleur cas en 0 propagation (ex : $111111 + 111111$)
- cas moyen ?

En 1946, A. Burks, H. Goldstine et J. Von Neumann présentent dans "*preliminary discusion of the logical design of an electronic instrument*" l'étude du temps de calcul moyen d'un additionneur séquentiel.
Soit $L(n)$ la *longueur moyenne de la plus grande chaîne de propagation de retenue* dans un additionneur séquentiel de n bits avec une distribution uniforme et équiprobable des entrées. On a :

$$L(n) \leq \log_2 n$$

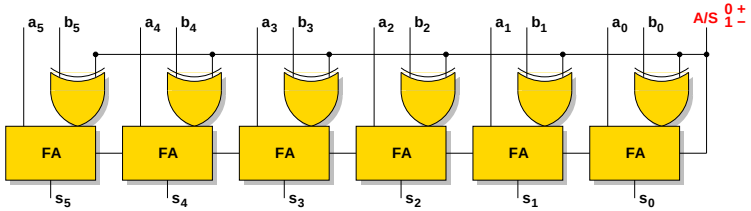
Soustraction

En complément à 2 on va faire :

$$A - B = A + (-B)$$

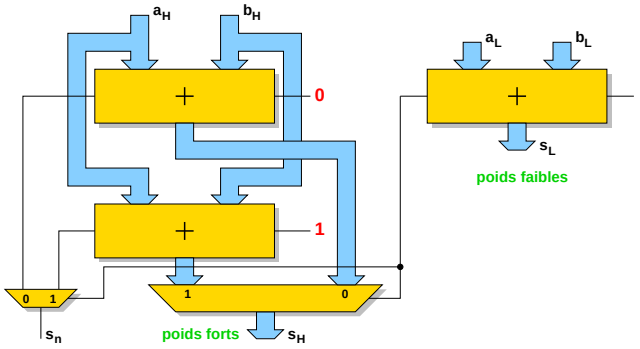
Pour avoir l'opposé de B :

$$\begin{aligned} B + \overline{B} &= \underbrace{111\ldots111}_{n \text{ bits}} = -2^{n-1} + \sum_{i=0}^{n-2} 2^i = -1 \\ -B &= \overline{B} + 1 \end{aligned}$$



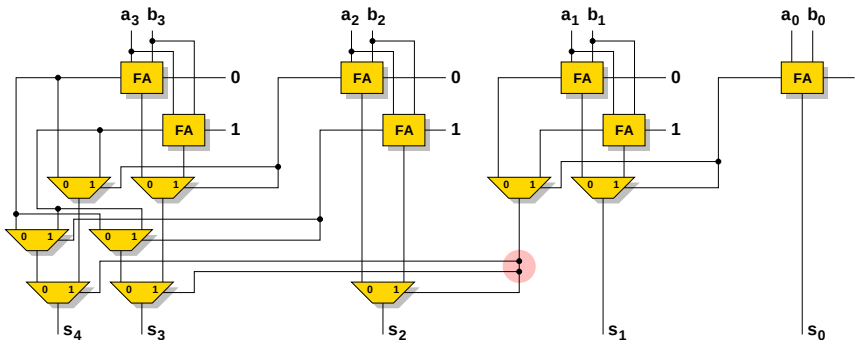
Additionneur à sélection de retenue

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 ⟶ délai en $O(\log n)$ mais...

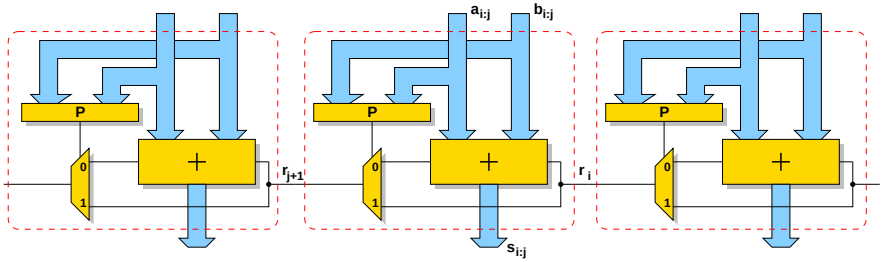
Il y a un petit problème de sortance



La sortance de certains nœuds n'est pas bornée. On peut ajouter des portes (inverseurs ou *buffers*) pour regénérer le signal mais cela augmente le délai car ces nœuds sont sur le chemin critique.

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

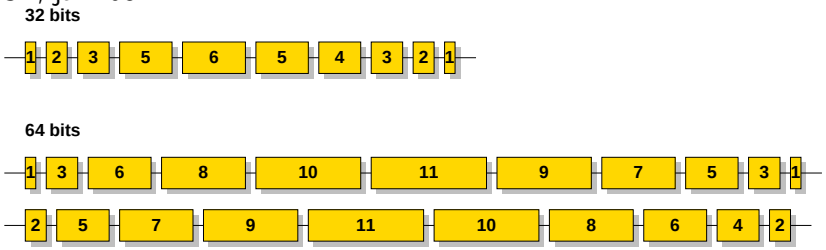
Le pire cas d'un additionneur de n bits découpé en blocs de k bits se produit lorsque l'on doit propager du bit 1 jusqu'au bit $n-2$ (générations de retenues aux bits 0 et $n-1$ et propagation au milieu). Soit τ_1 le temps de propagation sur un 1 bit et τ_2 le temps de saut d'un bloc de k bits ($\tau_2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

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

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

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.



Additionneur à retenues anticipées

Jusqu'ici on a essayé de propager au plus vite les retenues. On va maintenant essayer de les calculer au plus vite.
L'étage de rang i reçoit une retenue entrante c_i égale à 1 dans les seuls cas suivants[5] :

- l'étage $i-1$ génère une retenue
  $\hookrightarrow g_{i-1} = 1$
- l'étage $i-1$ propage une retenue générée à l'étage $i-2$
  $\hookrightarrow p_{i-1} = g_{i-2} = 1$
- les étages $i-1$ et $i-2$ propagent une retenue générée à l'étage $i-3$
  $\hookrightarrow p_{i-1} = p_{i-2} = g_{i-3} = 1$

$\vdots$

- les étages de $i-1$ à 0 propagent la retenue entrante c_0 à 1
  $\hookrightarrow p_{i-1} = p_{i-2} = \ldots = p_1 = p_0 = c_0 = 1$

[5] Rappels : $p_i = a_i \oplus b_i$, $g_i = a_i b_i$ et $k_i = \overline{a_i + b_i}$.

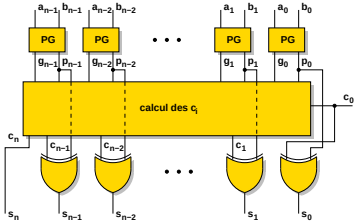
Additionneur à retenues anticipées

On peut donc calculer les retenues avec la relation[6] suivante :

$$c_i = g_{i-1} + p_{i-1}g_{i-2} + p_{i-1}p_{i-2}g_{i-3} + \ldots + p_{i-1}\cdots p_1 g_0 + p_{i-1}\cdots p_0 c_0$$

Le principe des additionneurs à retenues anticipées (*Carry Look Ahead*) consiste à évaluer de manière totalement parallèle :

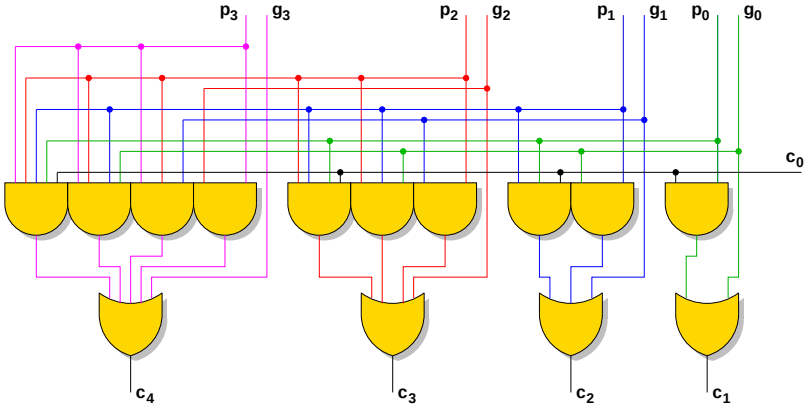
- les couples (g_i, p_i)
- les retenues c_i à l'aide de la relation ci-dessus
- les sommes $s_i = a_i \oplus b_i \oplus c_i = p_i \oplus c_i$



[6] qui se démontre par récurrence en utilisant $c_i = g_{i-1} + c_{i-1}p_{i-1}$.

Calcul des retenues pour un CLA 4 bits

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



Remarques sur les CLA

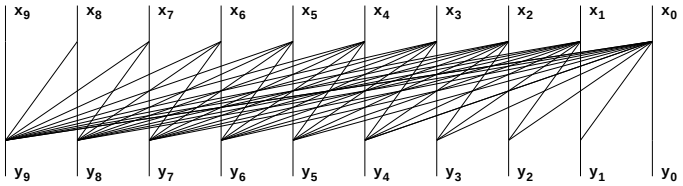
- Problème de sortance non bornée.
- Problème d'entrance[7] non bornée pour certaines portes.
- Additionneur théoriquement en temps constant mais logarithmique en pratique du fait de l'ajout de portes supplémentaires pour résoudre les problèmes d'entrance et de sortance.
- Les CLA à 4 bits servent souvent comme petites briques de base (élémentaires ou pas) pour concevoir de plus gros additionneurs (CLA ou autres).

[7] Nombre d'entrées limité dans une porte.

Résolution de problèmes par préfixe parallèle

Les n sorties $(y_{n-1}, y_{n-2}, \cdots, y_0)$ sont calculées à partir des n entrées $(x_{n-1}, x_{n-2}, \cdots, x_0)$ en utilisant un opérateur associatif $\square$ avec :

$$\begin{aligned} y_0 &= x_0 \\ y_1 &= x_1 \square x_0 \\ y_2 &= x_2 \square x_1 \square x_0 \\ &\vdots \\ y_{n-1} &= x_{n-1} \square x_{n-2} \square \cdots \square x_1 \square x_0 \end{aligned}$$



Addition par préfixe parallèle

Étape 1 : Calcul des générations et propagations des entrées :

$$g_i = a_i b_i \quad \text{et} \quad p_i = a_i \oplus b_i \qquad \forall i = 0, 1, \ldots, n-1$$

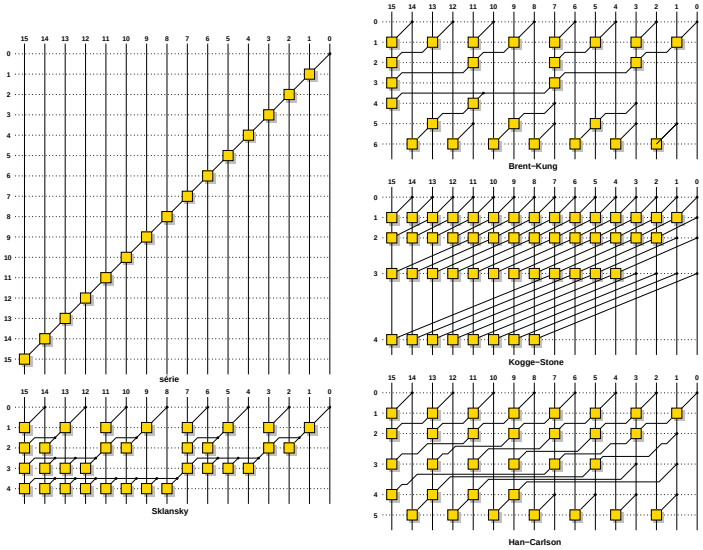
Étape 2 : Calcul des retenues c_i par préfixe parallèle à m niveaux :

$$\begin{aligned} (G^0_{i:i}, P^0_{i:i}) &= (g_i, p_i) \\ (G^l_{i:k}, P^l_{i:k}) &= (G^{l-1}_{i:j}, P^{l-1}_{i:j}) \square (G^{l-1}_{j:k}, P^{l-1}_{j:k}) \quad k \le j \le i \text{ et } l = 1, \ldots, m \\ &= (G^{l-1}_{i:j} + P^{l-1}_{i:j} G^{l-1}_{j:k}, P^{l-1}_{i:j} P^{l-1}_{j:k}) \\ c_{i+1} &= G^m_{i:0} + P^m_{i:0} c_0 \quad \forall i = 0, 1, \ldots, n-1 \end{aligned}$$

Étape 3 : Calcul des sommes :

$$s_i = p_i \oplus c_i \qquad \forall i = 0, 1, \ldots, n-1$$

Quelques additionneurs à préfixe parallèle



Addition en notation scientifique

Format : base 10, 4 chiffres de mantisse, arrondi au plus proche.

Opération :

$$1.601 \times 10^{-5} + 9.927 \times 10^{-4} = ?$$

$$\begin{array}{lll} & 0.0009927000\ldots & \text{poids } 10^{-4} \\ + & 0.0000160100\ldots & \text{poids } 10^{-5} \\ \hline & 0.0010087100\ldots & \text{poids } 10^{-3} \\ = & 1.0087100\ldots \times 10^{-3} & \\ = & 1.009 \times 10^{-3} & \end{array}$$

Addition flottante

Entrées :

$$x = (-1)^{s_x} \times m_x \times 2^{e_x} \quad \text{et} \quad y = (-1)^{s_y} \times m_y \times 2^{e_y}$$

Résultat :

$$r = x \oplus y \quad \text{où} \quad r = (-1)^{s_r} \times m_r \times 2^{e_r}$$

Principe :

Étape 1 : addition/soustraction mantisses, calcul exposant

$$m_r \neq m_x \pm m_y \quad \text{en fait} \quad m_r = f(m_x, m_y, e_x, e_y, s_x, s_y)$$

$$e_r = \max(e_x, e_y)$$

Étape 2 : normalisation mantisse

Étape 3 : arrondi

Étape 4 : traitement cas spéciaux

Addition/soustraction des mantisses

Étape 1.1 : différence des exposants

$$d = e_x - e_y$$

Étape 1.2 : échange des valeurs

$$x \leftrightarrow y \quad \text{si} \quad d < 0$$

Étape 1.3 : alignement des mantisses

$$m_y = m_y \times 2^{-d}$$

Étape 1.4 : calcul du type d'opération

$$op = + \quad \text{si} \quad (\oplus, s_x = s_y) \text{ ou } (\ominus, s_x \neq s_y)$$

$$op = - \quad \text{si} \quad (\oplus, s_x \neq s_y) \text{ ou } (\ominus, s_x = s_y)$$

Étape 1.5 : addition/soustraction

$$m_r = m_x \, op \, m_y$$

Étape 1.6 : correction du signe

$$m_r = -m_r \quad \text{si} \quad m_r < 0$$

Normalisation de la mantisse

Principe :

Étape 2.1 : détection du 1 de poids fort (MSO *most significant one*)

$$t = LOD(m_r)$$

Étape 2.2 : normalisation (et correction exposant)

$$m_r = m_r \times 2^t \quad \text{et} \quad e_r = e_r + t$$

Il y a 3 cas possibles :

rien à faire ($t = 0$)	débordement ($t = -1$)	cancellation ($t > 0$)
`1.10001...`	`1.10001...`	`1.10111...`
`+0.00101...`	`+1.00101...`	`-1.10100...`
`1.10110...`	`10.10110...`	`0.00111...`

LOD : *leading one detector*

Arrondi

Entrée : mantisse pas encore dans le bon format (trop de bits)

$$m_r = 1.\underbrace{xxxx\ldots xxxx}_{n \text{ bits}}\underbrace{xxxx\ldots xxxx}_{k \text{ bits}}$$

Sortie : mantisse dans le bon format

$$m'_r = 1.\underbrace{xxxx\ldots xxxx}_{n \text{ bits}}$$

Solution : utilisation des «3 bits de garde» pour arrondir, grosso modo on ajoute $+1$ à m'_r suivant `grs` et le signe de r.

$$\overbrace{xx\ xx\ldots xxxx}^{s}{}_{k \text{ bits}} \longrightarrow \underbrace{grs}_{3 \text{ bits}}$$

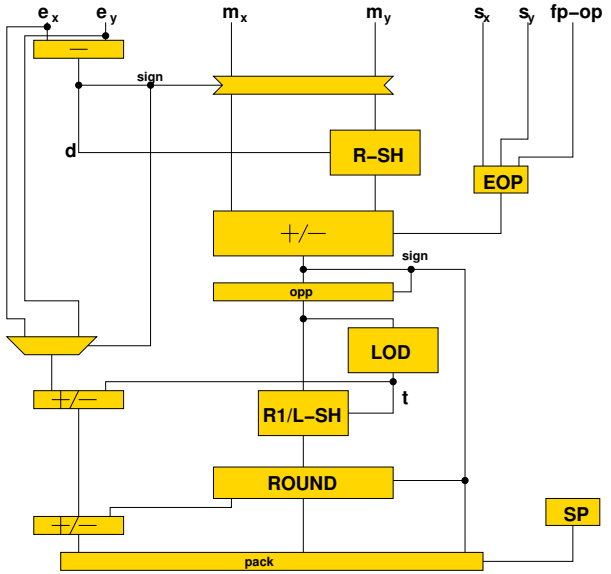
Cas spécial : l'arrondi propage une retenue qui nécessite une renormalisation de la mantisse.

Traitement des cas spéciaux

addition	$-\infty$	$x \in \mathbb{F}^*_-$	0^-	0^+	$x \in \mathbb{F}^*_+$	$+\infty$	NaN
$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	NaN	NaN
$y \in \mathbb{F}^*_-$	$-\infty$	$x+y$ ou $-\infty$	y	y	$x+y$	$+\infty$	NaN
0^-	$-\infty$	x	0^-	0^+	x	$+\infty$	NaN
0^+	$-\infty$	x	0^+	0^+	x	$+\infty$	NaN
$y \in \mathbb{F}^*_+$	$-\infty$	$x+y$	y	y	$x+y$ ou $+\infty$	$+\infty$	NaN
$+\infty$	NaN	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

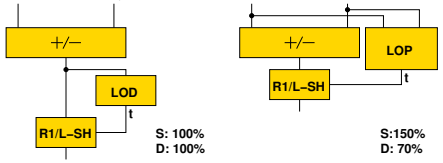
L'implantation pratique est simple car c'est juste quelques portes logiques.

Schéma fonctionnel



Optimisations

- Séparer les cas "impossibles simultanément"
  Exemple : *Close/Far Path*, si $d > 1$ alors pas de cancellation possible
- Anticiper certaines valeurs
  Exemple : Commencer à calculer t en même temps que l'addition/soustraction des mantisses



- Spéculer sur les résultats les plus probables (latence variable)

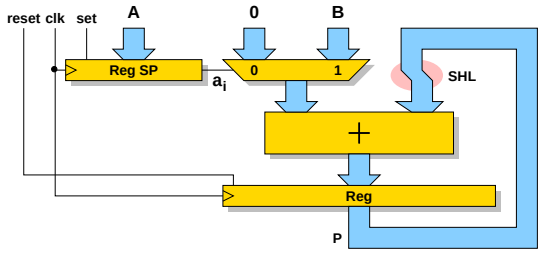
Problème : les optimisations entrainent une augmentation significative de la surface

Multiplication (et extensions)

Décomposition du problème en deux parties :

- multiplication des mantisses
- gestion de la virgule (simple)

Implantation matérielle du multiplieur par additions–décalages



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

Multiplieur ou multiplicateur ?

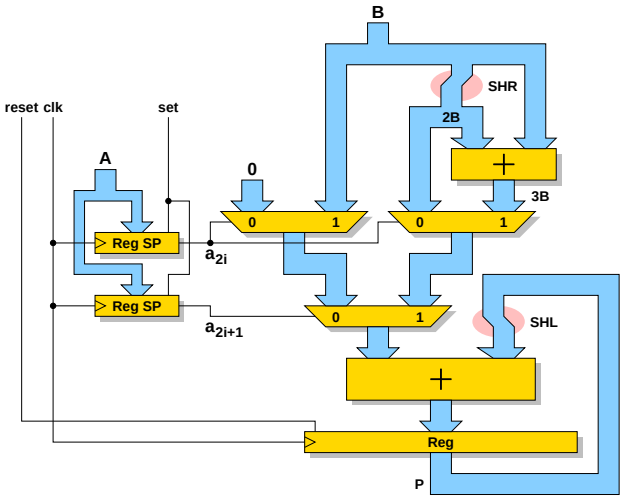
Dans "Le petit Larousse 2003" on trouve :

MULTIPLIEUR n.m. INFORM. Organe d'un calculateur analogique ou numérique permettant d'effectuer le produit de deux nombres.

MULTIPLICATEUR n.m. ARITHM. Nombre par lequel on multiplie un autre appelé *multiplicande*. (Dans 3 fois 4, égal à $4 + 4 + 4$, 3 est le multiplicateur.)

Accélération de la multiplication série–parallèle

Idée : utiliser des chiffres du multiplicateur dans une base plus grande.



Recodage de Booth

Idée de Booth en 1951 : augmenter le nombre de 0 dans le multiplicateur en le recodant avec l'ensemble de chiffres signés $\{-1 = \overline{1}, 0, 1\}$.

Utilisation de l'identité : $2^{i+k} + 2^{i+k-1} + 2^{i+k-2} + \cdots + 2^i = 2^{i+k+1} - 2^i$

Exemple : le nombre 60 peut s'écrire $00111100 = 0100\overline{1}00$.

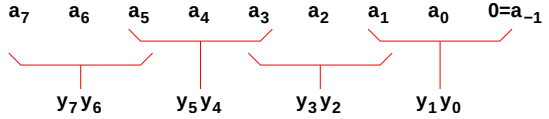
Cette méthode permet de transformer les chaînes de 1 par une écriture avec plus de 0.

Mais dans certains cas on peut faire apparaître plus de 1 dans la chaîne recodée que dans la chaîne initiale !
Exemple : la chaîne 01010101 se recode par Booth en $\overline{1}1\overline{1}1\overline{1}1\overline{1}1$.

$\Longrightarrow$ utilisation du recodage de Booth modifié

Recodage de Booth modifié

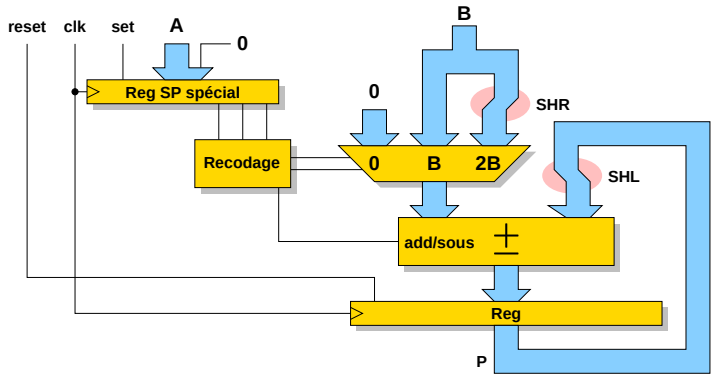
Idée : ne pas recoder les 1 isolés mais seulement les chaînes de 1.



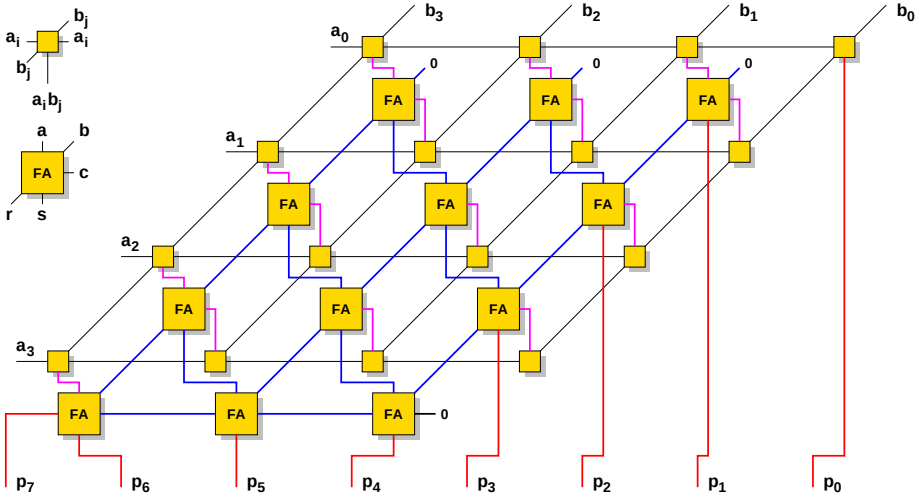
a_i	a_{i-1}	a_{i-2}	y_i	y_{i-1}	signification	opération
0	0	0	0	0	chaîne de 0	$+0$
0	0	1	0	1	fin chaîne de 1	$+B$
0	1	0	0	1	1 isolé	$+B$
0	1	1	1	0	fin chaîne de 1	$+2B$
1	0	0	$\overline{1}$	0	début chaîne de 1	$-2B$
1	0	1	1	$\overline{1}$	0 isolé	$-B$
1	1	0	0	$\overline{1}$	début chaîne de 1	$-B$
1	1	1	0	0	milieu chaîne de 1	$+0$

Avantage : faire un produit de 2 nombres de n bits en $\lfloor n/2 \rfloor + 1$ additions–décalages au plus.

Multiplieur série–parallèle avec recodage modifié de Booth



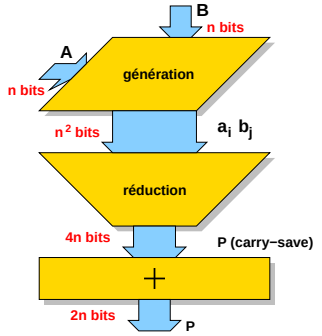
Multiplieur cellulaire de Braun



Temps de calcul en $O(n)$.

Multiplieurs arborescents

1. Génération parallèle des produits partiels $a_i b_j$ avec ou sans recodage de Booth
   ↪ temps en $O(1)$ (sortance a_i, b_j $O(\log n)$)
2. Calcul de la somme en *carry-save* des produits partiels à l'aide d'un arbre de réduction
   ↪ temps en $O(\log n)$
3. Addition finale des 2 vecteurs de bits de la somme *carry-save* avec un additionneur rapide
   ↪ temps en $O(\log n)$



On a donc une structure permettant la multiplication en un temps de l'ordre de $O(\log n)$.

Génération des produits partiels

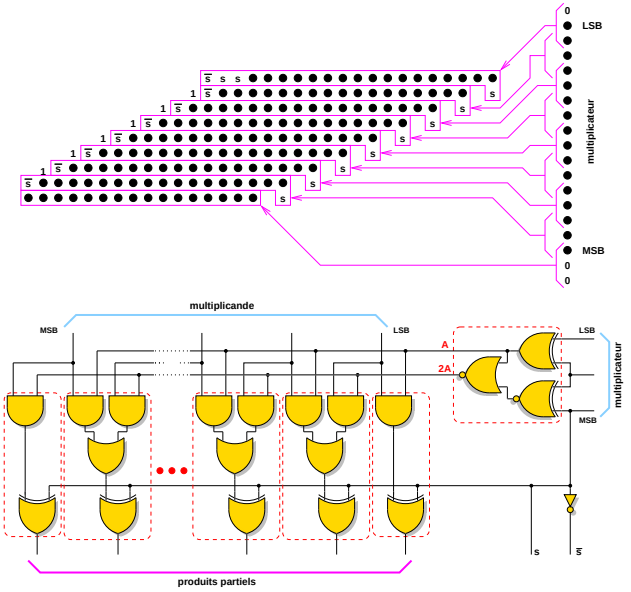
Dans le produit $P = A \times B$ où A et B sont des entiers de n bits, la génération des produits partiels nécessite n^2 cellules AND (ou $n^2 - 2n - 2$ AND et $2n - 2$ NAND dans le cas d'entrées en complément à deux).

Tous les produits partiels $a_i b_j$ peuvent être obtenus en même temps modulo le problème de sortance sur les entrées a_i et b_j.

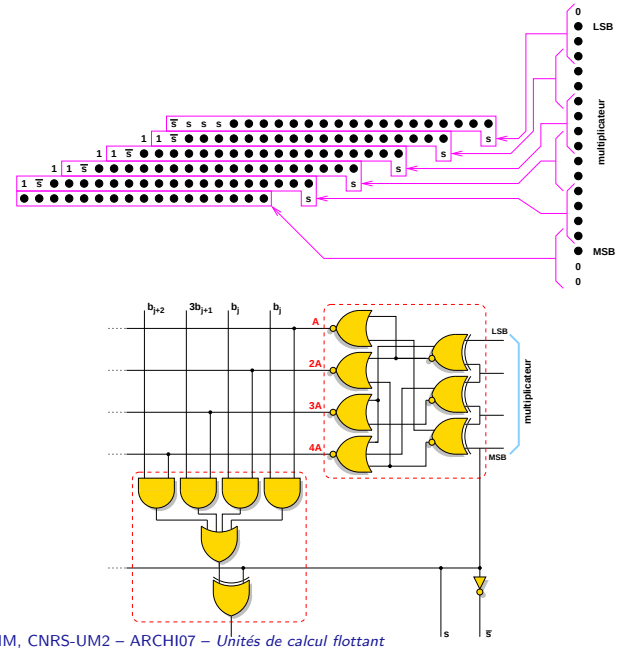
On peut utiliser un recodage de Booth modifié pour diminuer le nombre de produits partiels. Les solutions les plus courantes en terme de recodage de Booth modifié dans les multiplieurs arborescents sont :

- Booth 2 : recodage des a_i en base 4 avec l'ensemble de chiffres $\{0, \pm 1, \pm 2\}$.
- Booth 3 : recodage des a_i en base 8 avec l'ensemble de chiffres $\{0, \pm 1, \pm 2, \pm 3, \pm 4\}$.

Génération des produits partiels : Booth 2



Génération des produits partiels : Booth 3



Gain du recodage de Booth

Cas du recodage de Booth 2 :

- Vitesse : le gain d'un étage ou deux sur l'arbre de réduction des produits partiels est plus ou moins compensé par l'étage de recodage et génération des produits partiels beaucoup plus complexe.
- Surface : véritable gain (30 %) car on utilise les cellules de recodage pour régénérer le signal pour éviter les problèmes de sortance (chose qu'il faut faire sans le recodage avec des *buffers* supplémentaires) et les cellules de générations se réalisent bien en CMOS.

Cas du recodage de Booth 3 : Rarement utilisé en pratique car les étages de génération des produits partiels deviennent beaucoup trop complexes (nombreuses portes et dont certaines avec 3 ou 4 entrées).

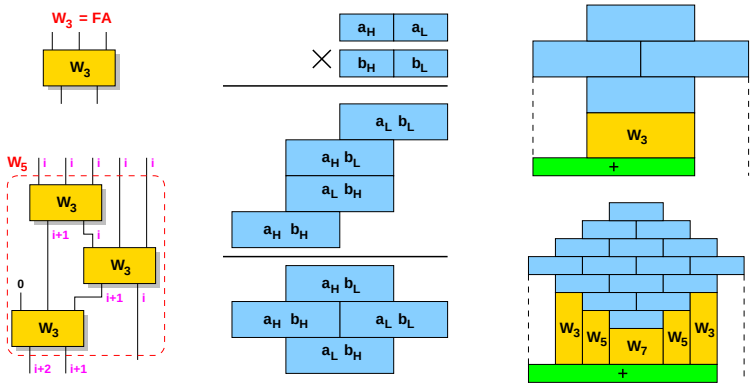
Arbres de réduction des produits partiels

Il faut maintenant calculer la somme *carry-save* des $n/2+1$ produits partiels. Pour cela, différents types d'arbres de réduction sont possibles :

- À base de cellules FA :
  - arbres de Wallace
  - arbres de Dadda
  - algorithmes de réduction rapide
- À base de cellules "4 donne 2"
- À base de "grands" compteurs

Arbres de Wallace

Les arbres de Wallace[8] sont des compteurs à p entrées ($\lceil log_2 p \rceil$ sorties). Un arbre de Wallace à $2^{p+1}-1$ se construit facilement à partir d'arbres de Wallace à 2^p-1 entrées (un arbre de Wallace à 3 entrées est une cellule FA).

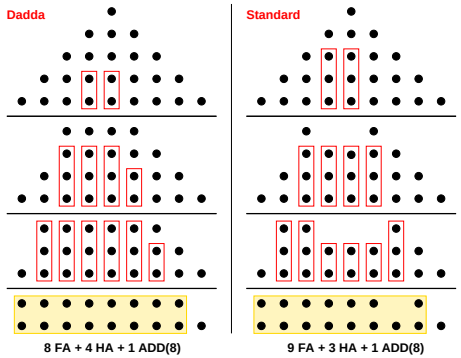


[8] C.S. Wallace. *A suggestion for a fast multiplier*. IEEE Transactions on Computers, février 1964.

Arbres de Dadda

Idée : faire le minimum de réduction à chaque étape pour gagner un niveau dans la suite $n(h) = \lfloor 3n(h-1)/2 \rfloor$ 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



Gain en surface pour les grands multiplieurs. Exemple : $n = 12$ bits $\Rightarrow$ 11% de portes en moins qu'avec un arbre de Wallace.

Algorithmes pour la génération automatique d'arbres de réduction optimisés

Il existe des généralisations et améliorations de la méthode de Dadda pour différents types de compteurs et utilisant des modèles de délai complexes (ex : bits de retenue plus rapides que ceux de somme).

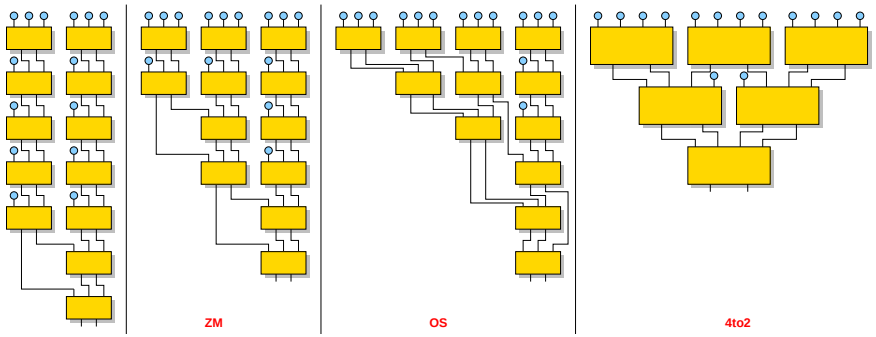
Par exemple, la méthode TDM[9] permet de gagner jusqu'a 30% en vitesse et 15% en surface par rapport à une solution utilisant un arbre de cellules "4 donne 2".

La génération automatique de multiplieurs est un problème très compliqué et fait intervenir différents aspects (souvent antagonistes) : vitesse, surface, régularité topologique, consommation d'énergie, besoins en cellules spéciales...

[9] V. Oklobdzija, D. Villeger et S. Liu. *A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach*.IEEE Transactions on Computers, mars 1996.

Problèmes topologiques au niveau cellules

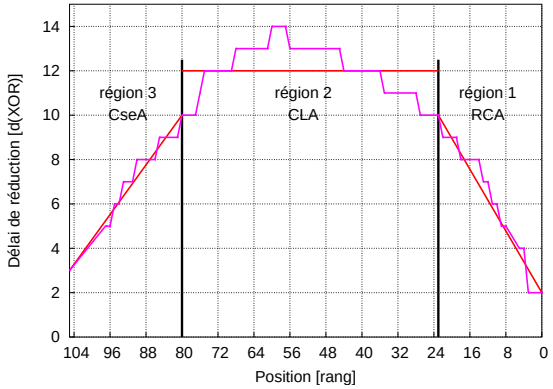
Quelle est la topologie la plus efficace et la plus simple à gérer ?



Réduction de 14 bits (sans mentionner les retenues latérales).

Addition finale pour multiplieurs

L'addition finale dans un multiplieur arborescent (assimilation des retenues) peut se faire avec un additionneur rapide. Mais de petites optimisations sont possibles en utilisant les temps d'arrivées relatifs des bits de la somme *carry-save* après réduction (adaptation du type d'additionneur suivant les régions).



Multiplication flottante

Entrées :

$$x = (-1)^{s_x} \times m_x \times 2^{e_x} \quad \text{et} \quad y = (-1)^{s_y} \times m_y \times 2^{e_y}$$

Résultat :

$$r = x \otimes y \quad \text{où} \quad r = (-1)^{s_r} \times m_r \times 2^{e_r}$$

Principe :

- Étape 1 : multiplication mantisses, calcul exposant

$$m_r = m_x \times m_y$$

$$e_r = e_x + e_y$$

- Étape 2 : normalisation mantisse

$$[1,2) \times [1,2) = [1,4)$$

- Étape 3 : arrondi (proche addition)
- Étape 4 : traitement cas spéciaux

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

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

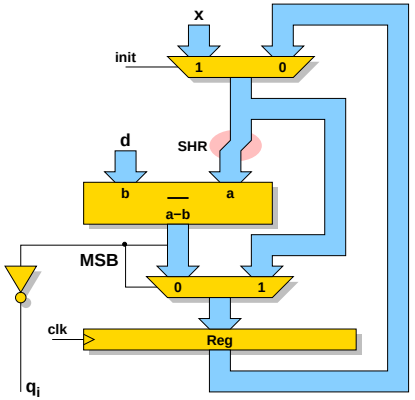
Division

Décomposition du problème en deux parties :

- division des mantisses
- gestion de la virgule (assez simple)

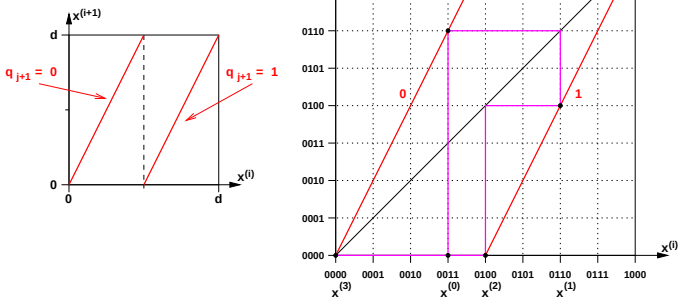
Division restaurante

```
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



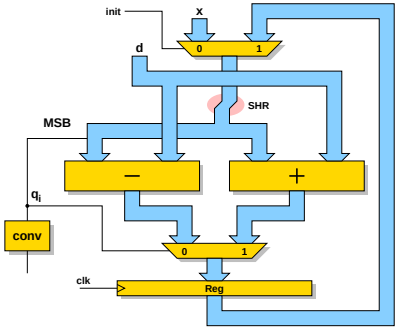
x = 11
d = 1000

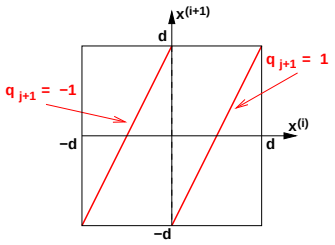
$x^{(0)} = 0011$	$q_0 = 0$
$x^{(1)} = 0110$	$q_1 = 1$
$x^{(2)} = 0100$	$q_2 = 1$
$x^{(3)} = 0000$	$q_3 = 0$

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

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
```



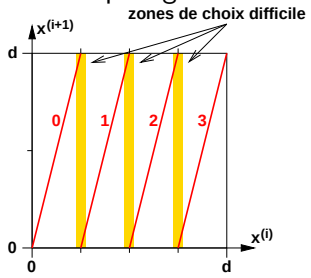


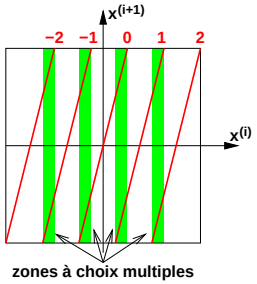
La conversion de $\{-1, 1\}$ vers $\{0, 1\}$ peut se faire à la volée (MSDF) avec un petit opérateur.

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.
$\hookrightarrow$ 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

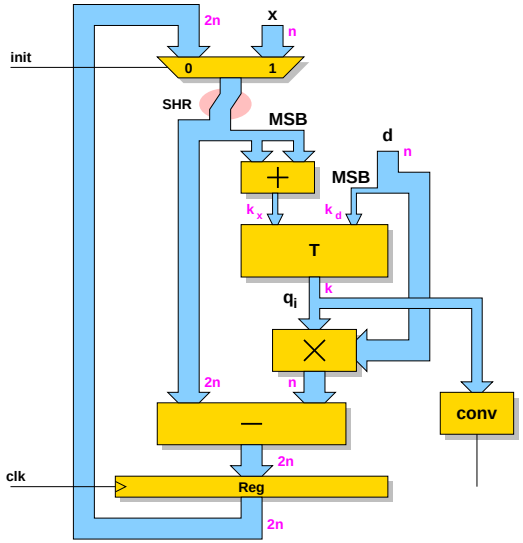
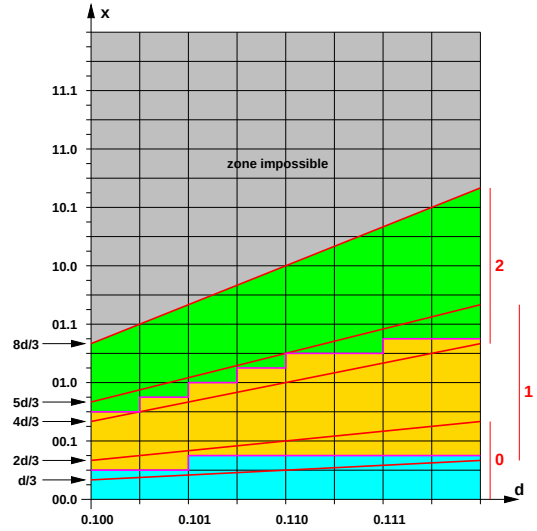


Table pour un diviseur SRT

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.



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

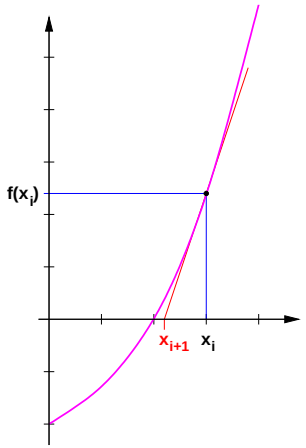
1 **for** i from 1 **to** $n/\log_2 \beta$ **do**
2 $x'^{(i)} \longleftarrow \mathrm{trunc}(x^{(i)}, k_x, \mathrm{MSB})$
3 $r'^{(i)} \longleftarrow \mathrm{trunc}(r^{(i)}, k_r, \mathrm{MSB})$
4 $r_{i+1} \longleftarrow \mathrm{T}(\mathrm{conv}(r'^{(i)}), \mathrm{conv}(x'^{(i)}))$
5 $r^{(i+1)} \longleftarrow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

Méthode de Newton

On cherche une racine de l'équation $f(x) = 0$ (où f est supposée continument 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 :

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

   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.

Division flottante

Entrées :

$$x = (-1)^{s_x} \times m_x \times 2^{e_x} \quad \text{et} \quad y = (-1)^{s_y} \times m_y \times 2^{e_y}$$

Résultat :

$$q = x \oslash y \quad \text{où} \quad q = (-1)^{s_q} \times m_q \times 2^{e_q}$$

Principe :

Étape 1 : division mantisses, calcul exposant

$$m_q = m_x / m_y$$

$$e_q = e_x - e_y$$

Étape 2 : normalisation mantisse

$$[1,2]/[1,2] = [1/2, 2]$$

Étape 3 : arrondi (proche addition)

Étape 4 : traitement cas spéciaux

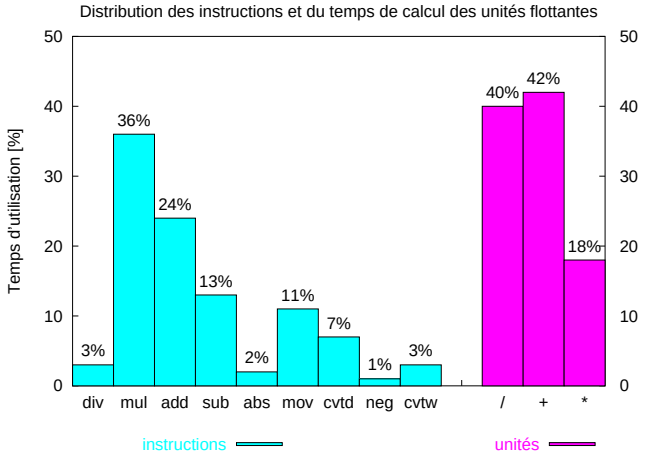
Mais...

L'arrondi est plus ou moins complexe/coûteux suivant l'algorithme utilisé pour la division des mantisses.

- Algorithme SRT : production du reste $\Longrightarrow$ bits de garde $\Longrightarrow$ arrondi simple
- Algorithme à la Newton : pas d'info sur le signe du reste $\Longrightarrow$ faire $x - q \times y$ (au moins partiellement) pour avoir le signe du reste

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

Rapport technique[10] de S. Oberman et M. Flynn : "*Design issues in floating-point division*", CSL-TR-94-647 Stanford University.



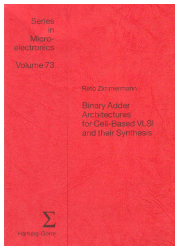
[10] SPECfp92 sur DECstation avec un MIPS R3000 (latences : 2c add, 5c mul, 19c div), compil. O3.

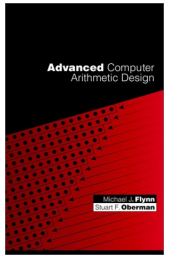
Pour en savoir plus. . .

Binary Adder Architectures for Cell-Based VLSI and their Synthesis

Reto Zimmermann

1998
Hartung–Gorre
ISBN : 3–89649–289–6





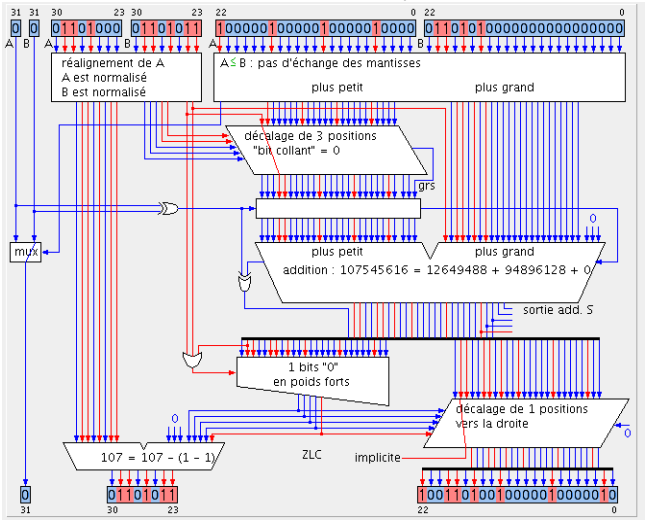
Advanced Computer Arithmetic Design

Micheal Flynn et Stuart Obermann

2001
Wiley–Interscience
ISBN : 0–471–41209–0

Un super site web. . .

`http://tima-cmp.imag.fr/~guyot/Cours/Oparithm/francais/Op_Ar2.htm`
(recherche google sur "Alain Guyot op art")



Évolutions de l'arithmétique flottante

- Révision de la norme IEEE-754
- Unités en quadruple précision (128 bits)
- Représentation décimale
- Primitives plus complexes
- Unités reconfigurables
- . . .

Fin, des questions ?

Contact:

- `mailto:arnaud.tisserand@lirmm.fr`
- `http://www.lirmm.fr/~tisseran`
- Equipe/projet Arith
- Laboratoire LIRMM, CNRS–Univ. Montpellier 2
  161 rue Ada. F-34392 Montpellier cedex 5. France

Merci

Join us for ARITH18 `http://www.lirmm.fr/arith18`

18th IEEE Symposium on Computer Arithmetic
Montpellier, France, June 25-27, 2007

Submission deadline	October 15th, 2006
Acceptance notification	February 2007
Final version deadline	March 2007