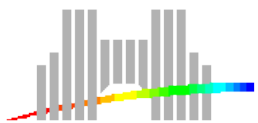
# Les fonctions élémentaires dans un PC : un exemple d'adéquation algorithme-architecture

Florent de Dinechin, projet Arénaire

ARCHI 09, Pleumeur-Bodou

# Plan

# Évaluation de fonctions

# Trois bouquins



J.-M. Muller.
*Elementary Functions, Algorithms and Implementation*.
Birkhäuser Boston, MA, 2nd edition, 2006.


P. Markstein.
*IA-64 and Elementary Functions : Speed and Precision*.
Hewlett-Packard Professional Books. Prentice-Hall, Englewood Cliffs, NJ, 2000.


M. Cornea, J. Harrison, and P. T. P. Tang.
*Scientific Computing on Itanium®-based Systems*.
Intel Press, Hillsboro, OR, 2002.

# Fonction ?

## Fonction continue et dérivable d'une variable réelle

- Fonction algébrique : polynôme, $\sqrt[3]{x}$, $\frac{1}{\sqrt{x}}$, ...
- Fonction élémentaire : sinus, exponentielle, logarithme
- Autre fonction ayant sa place dans une bibliothèque mathématique : erf, Gamma, ...
- Composée, toute fonction utile

# Fonction ?

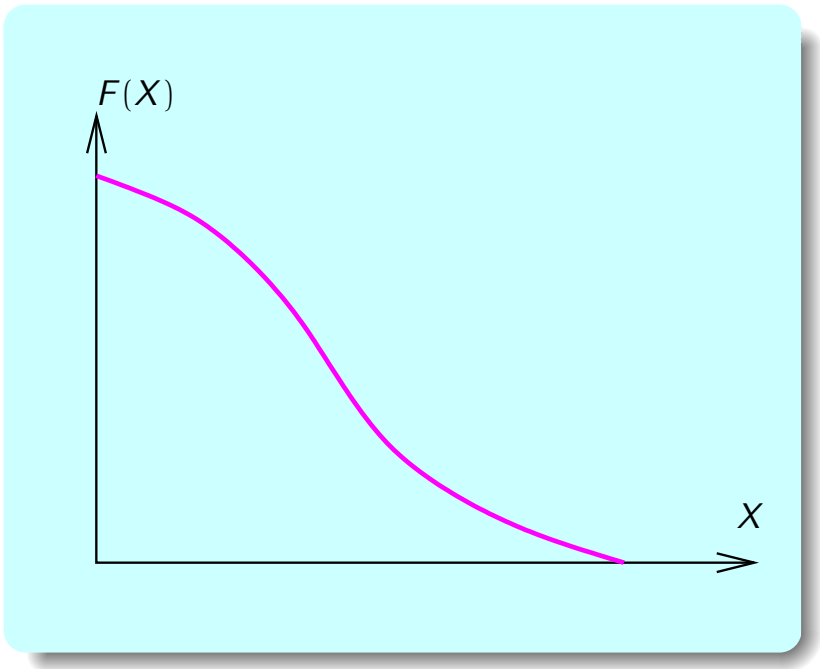
## Fonction continue et dérivable d'une variable réelle

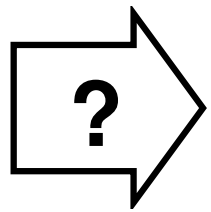
- Fonction algébrique : polynôme, $\sqrt[3]{x}$, $\frac{1}{\sqrt{x}}$, ...
- Fonction élémentaire : sinus, exponentielle, logarithme
- Autre fonction ayant sa place dans une bibliothèque mathématique : erf, Gamma, ...
- Composée, toute fonction utile

Bibliothèque mathématique standard : `libm`

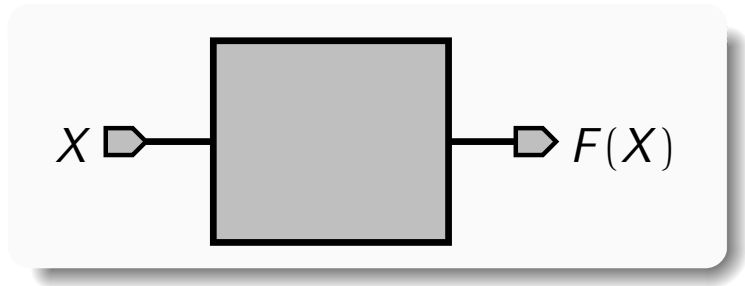
# D'une fonction à un opérateur qui la calcule

Une fonction



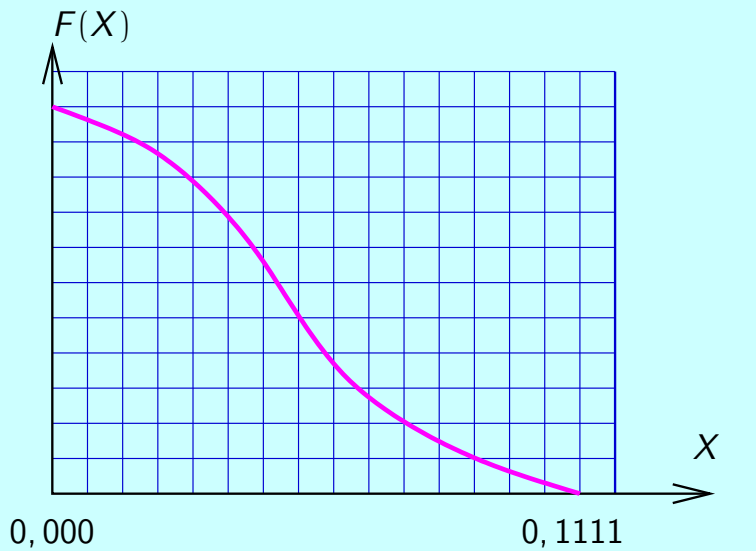




Un opérateur qui l'évalue



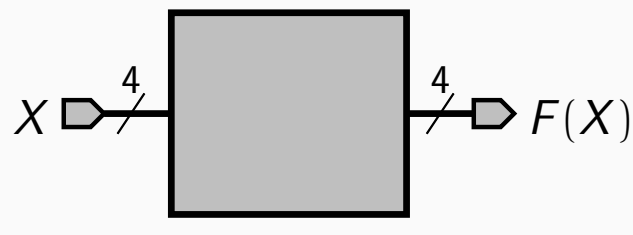
# D'une fonction à un opérateur qui la calcule

Une fonction




?

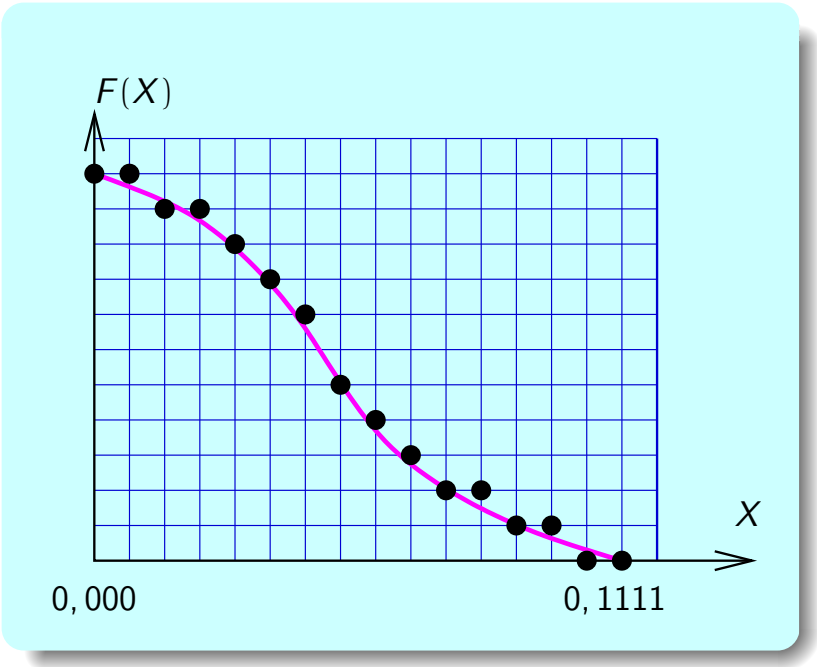
Un opérateur qui l'évalue




Représentation finie des nombres en entrée et en sortie

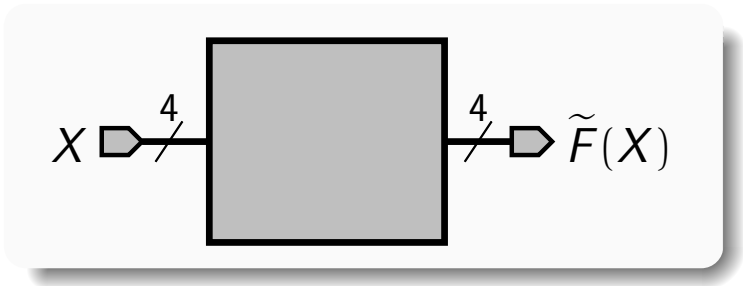
# D'une fonction à un opérateur qui la calcule

Une fonction




?

Un opérateur qui l'évalue




Représentation finie des nombres en entrée et en sortie

$\Longrightarrow$ le plus souvent, fonction approchée

# Que faut-il pour implémenter une fonction ?

# Que faut-il pour implémenter une fonction ?

Il faut pouvoir faire un peu de maths (analyse).

# Que faut-il pour implémenter une fonction ?

Il faut pouvoir faire un peu de maths (analyse).

- Continuité, dérivabilité à un certain ordre
  - Théorème de Taylor
  - Plus généralement, approximation polynomiale

# Que faut-il pour implémenter une fonction ?

Il faut pouvoir faire un peu de maths (analyse).

- Continuité, dérivabilité à un certain ordre
  - Théorème de Taylor
  - Plus généralement, approximation polynomiale

Briques de base :
opérateurs $+$, $\times$, et mémoire

# Que faut-il pour implémenter une fonction ?

Il faut pouvoir faire un peu de maths (analyse).

- Continuité, dérivabilité à un certain ordre
  - Théorème de Taylor
  - Plus généralement, approximation polynomiale

Briques de base :
opérateurs $+$, $\times$, et mémoire

- Identités mathématiques spécifiques
  - parité/imparité,
  - périodicité,
  - propriétés algébriques comme $\log(ab) = \log(a) + \log(b)$...

# Que faut-il pour implémenter une fonction ?

Il faut pouvoir faire un peu de maths (analyse).

- Continuité, dérivabilité à un certain ordre
  - Théorème de Taylor
  - Plus généralement, approximation polynomiale

Briques de base :
opérateurs $+$, $\times$, et mémoire

- Identités mathématiques spécifiques
  - parité/imparité,
  - périodicité,
  - propriétés algébriques comme $\log(ab) = \log(a) + \log(b)$...

Possibilité de réduction d'argument.

# Qualité numérique du résultat

Attention aux erreurs qui s'accumulent

# Qualité numérique du résultat

Attention aux erreurs qui s'accumulent

- erreurs d'approximation
  - approximation d'une fonction par un polynôme
  - suite qui converge vers une valeur
  - ...

# Qualité numérique du résultat

Attention aux erreurs qui s'accumulent

- erreurs d'approximation
  - approximation d'une fonction par un polynôme
  - suite qui converge vers une valeur
  - ...
- erreurs d'arrondi
  - dans la plupart des opérations
  - (discrétisation des résultats intermédiaires)

# Qualité numérique du résultat

Attention aux erreurs qui s'accumulent

- erreurs d'approximation
  - approximation d'une fonction par un polynôme
  - suite qui converge vers une valeur
  - ...
- erreurs d'arrondi
  - dans la plupart des opérations
  - (discrétisation des résultats intermédiaires)

On peut toujours réduire ces erreurs en calculant pour plus cher.

# Qualité numérique du résultat

Attention aux erreurs qui s'accumulent

- erreurs d'approximation
  - approximation d'une fonction par un polynôme
  - suite qui converge vers une valeur
  - ...
- erreurs d'arrondi
  - dans la plupart des opérations
  - (discrétisation des résultats intermédiaires)

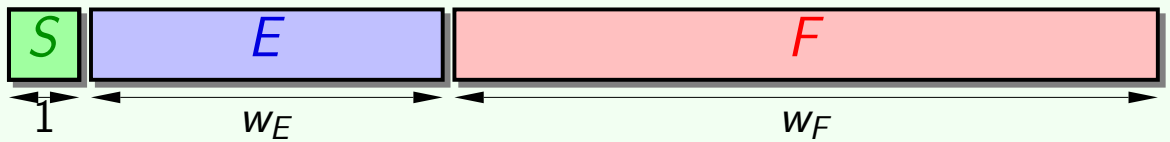
On peut toujours réduire ces erreurs en calculant pour plus cher.

Compromis précision / performance

# Les fonctions de la `libm`

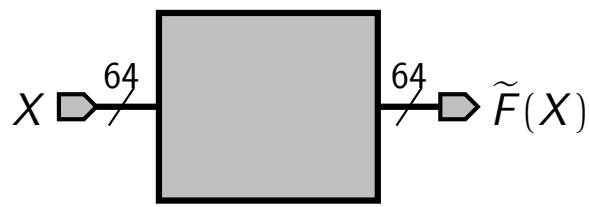
## Entrées et sorties flottantes

$$X = (-1)^S \cdot 2^E \cdot 1, F$$




(codes spéciaux pour $\pm 0$, $\pm\infty$, Not a Number)

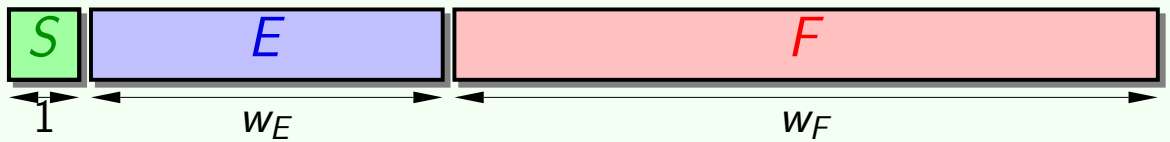
| format | $w_E$ | $w_F$ |
|---|---|---|
| simple précision (binary32) | 8 | 23 |
| double précision (binary64) | 11 | 52 |



# Les fonctions de la `libm`

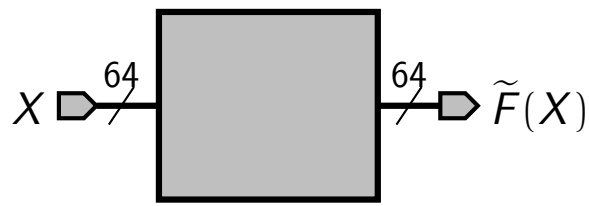
## Entrées et sorties flottantes

$$X = (-1)^S \cdot 2^E \cdot 1, F$$




(codes spéciaux pour $\pm 0$, $\pm\infty$, Not a Number)

| format | $w_E$ | $w_F$ |
|---|---|---|
| simple précision (binary32) | 8 | 23 |
| double précision (binary64) | 11 | 52 |

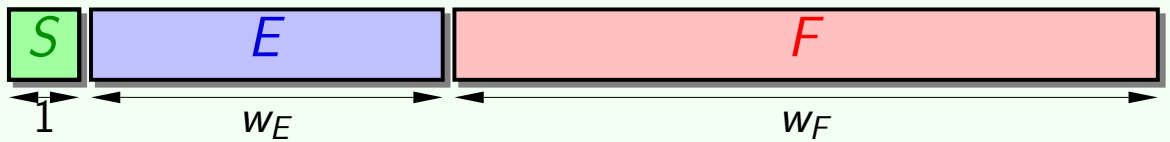



- Implémentation en logiciel ou microcode
  - quelques dizaines à quelques centaines de cycles

# Les fonctions de la `libm`

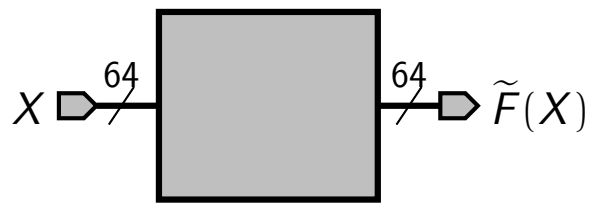
## Entrées et sorties flottantes

$$X = (-1)^S \cdot 2^E \cdot 1, F$$




(codes spéciaux pour $\pm 0$, $\pm\infty$, Not a Number)

| format | $w_E$ | $w_F$ |
|---|---|---|
| simple précision (binary32) | 8 | 23 |
| double précision (binary64) | 11 | 52 |

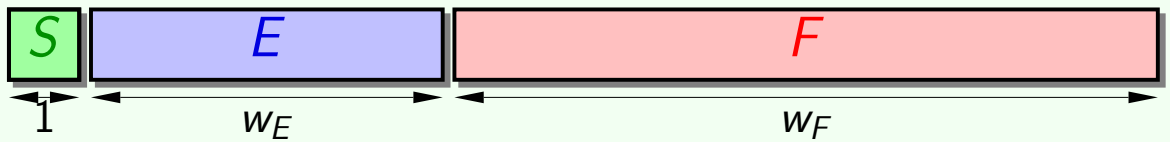



- Implémentation en logiciel ou microcode
  - quelques dizaines à quelques centaines de cycles
- La plupart des implémentation optimisent la latence

# Les fonctions de la `libm`

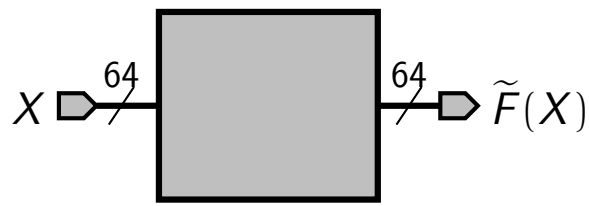
## Entrées et sorties flottantes

$$X = (-1)^S \cdot 2^E \cdot 1, F$$




(codes spéciaux pour $\pm 0$, $\pm\infty$, Not a Number)

| format | $w_E$ | $w_F$ |
|---|---|---|
| simple précision (binary32) | 8 | 23 |
| double précision (binary64) | 11 | 52 |




- Implémentation en logiciel ou microcode
  - quelques dizaines à quelques centaines de cycles
- La plupart des implémentation optimisent la latence
- Il existe des bibliothèques vectorielles (Intel MKL)

# L'historique

G. Paul and M. W. Wilson.
Should the elementary functions be incorporated into computer instruction sets ?
*ACM Transactions on Mathematical Software*, 2(2) :132–142, June 1976.

- Sa réponse est : oui

# L'historique

G. Paul and M. W. Wilson.
Should the elementary functions be incorporated into computer instruction sets ?
*ACM Transactions on Mathematical Software*, 2(2) :132–142, June 1976.

- Sa réponse est : oui
- L'histoire lui a donné tort

# L'historique

G. Paul and M. W. Wilson.
Should the elementary functions be incorporated into computer instruction sets ?
*ACM Transactions on Mathematical Software*, 2(2) :132–142, June 1976.

- Sa réponse est : oui
- L'histoire lui a donné tort
  - Votre Pentium offre des instructions pour exp, log, sin, ...
  - Elles sont cablés en dur en microcode
  - Je sais en écrire de plus rapides et plus précises en soft
  - C'est vraiment du gâchis de silicium...
- Mais à l'époque
  - cela permettait un OS plus compact
  - cela garantissait une certaine qualité numérique

# Par parenthèse

*Should the elementary functions be incorporated into computer instruction sets ?*

- Dans les années 90 la bande à Flynn a posé la même question pour la division.

# Par parenthèse

*Should the elementary functions be incorporated into computer instruction sets ?*

- Dans les années 90 la bande à Flynn a posé la même question pour la division.
- Et ils ont répondu oui aussi.

# Par parenthèse

*Should the elementary functions be incorporated into computer instruction sets ?*

- Dans les années 90 la bande à Flynn a posé la même question pour la division.
- Et ils ont répondu oui aussi.
- Et l'histoire leur a donné tort aussi
  (on y reviendra)

# Par parenthèse

*Should the elementary functions be incorporated into computer instruction sets ?*

- Dans les années 90 la bande à Flynn a posé la même question pour la division.
- Et ils ont répondu oui aussi.
- Et l'histoire leur a donné tort aussi
  (on y reviendra)

Moi, maintenant quand je pose une question comme celles-ci, je réponds non.

# Comment est-ce possible d'aller plus vite en soft ?

C'est parce que le PC a profondément changé depuis 1976 ou 1986 (introduction du 8087)

- La mémoire est passée du Ko au Go
  - on se permet de nos jours de précalculer de grosses tables
- Le processeur est devenu superscalaire
  - Il peut faire plusieurs $+$ et $\times$ en parallèle,
  - parce que c'est généralement utile
  - (plus que des sinus ou des exp)

# Comment est-ce possible d'aller plus vite en soft ?

C'est parce que le PC a profondément changé depuis 1976 ou 1986 (introduction du 8087)

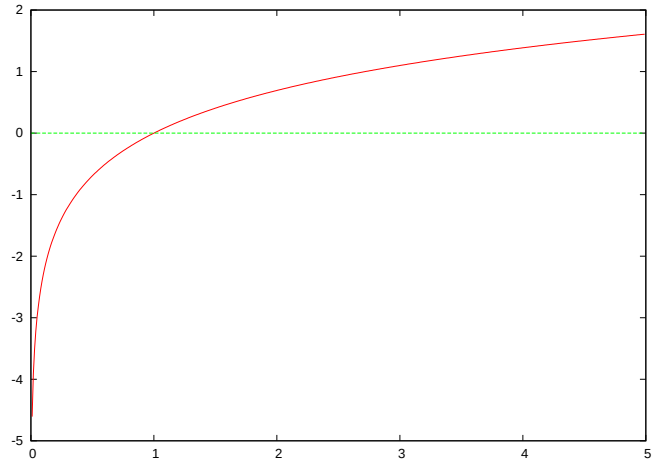
- La mémoire est passée du Ko au Go
  - on se permet de nos jours de précalculer de grosses tables
- Le processeur est devenu superscalaire
  - Il peut faire plusieurs $+$ et $\times$ en parallèle,
  - parce que c'est généralement utile
  - (plus que des sinus ou des exp)

Voyons un peu en détail les algorithmes modernes pour

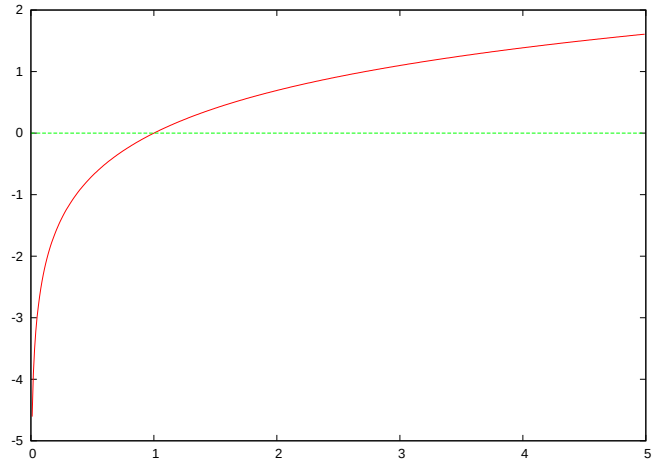
- Une fonction gentille : le logarithme népérien
- Une fonction méchante : le sinus
- (il existe des fonctions très méchantes)

# L'algorithmique sur deux exemples

# Une fonction gentille en détail : le logarithme

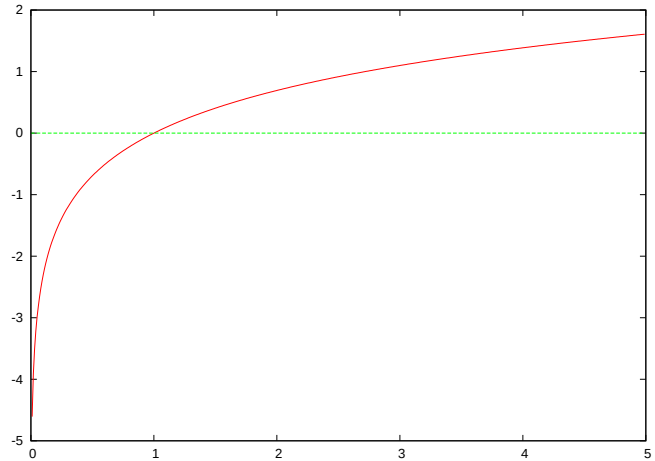


# Une fonction gentille en détail : le logarithme




Pourquoi elle est gentille ?

# Une fonction gentille en détail : le logarithme




Pourquoi elle est gentille ?

- Parce que $\ln(ab) = \ln(a) + \ln(b)$
- Parce que $\ln(1+\varepsilon) = \varepsilon - \frac{\varepsilon^2}{2} + \frac{\varepsilon^3}{3} \ldots$
- Parce qu'elle est strictement croissante
- Parce qu'elle ne fait pas de dépassement de capacité
  - Exemple : double-précision,
    $\text{maxFloat} \approx 1.798 \cdot 10^{308}$ $\quad \ln(\text{maxFloat}) < 710$
    $\text{minFloat} \approx 4.941 \cdot 10^{-324}$ $\quad \ln(\text{minFloat}) > -745$

# Démontage du format flottant

(Dans ce transparent et ceux qui suivent, je décris ce qu'il faut faire, pas encore comment on peut le faire)

- Test des cas exceptionnels (voir la norme IEEE-754-2008)

$$
\begin{array}{lcll}
X = \mathrm{NaN} & \longrightarrow & \ln(X) = \mathrm{NaN}, & \text{signaler Invalid} \\
X < 0 & \longrightarrow & \ln(X) = \mathrm{NaN}, & \text{signaler Invalid} \\
X = \pm 0 & \longrightarrow & \ln(X) = -\infty, & \text{signaler DivisionByZero}
\end{array}
$$

# Démontage du format flottant

(Dans ce transparent et ceux qui suivent, je décris ce qu'il faut faire, pas encore comment on peut le faire)

- Test des cas exceptionnels (voir la norme IEEE-754-2008)

$$\begin{array}{lcll} X = \mathrm{NaN} & \longrightarrow & \ln(X) = \mathrm{NaN}, & \text{signaler Invalid} \\ X < 0 & \longrightarrow & \ln(X) = \mathrm{NaN}, & \text{signaler Invalid} \\ X = \pm 0 & \longrightarrow & \ln(X) = -\infty, & \text{signaler DivisionByZero} \end{array}$$

- Première réduction d'argument
  - $X = 2^E \cdot 1, F$
  - Donc $\ln(X) = E \cdot \ln(2) + \ln(1, F)$
  - Pour recentrer la mantisse sur 1 et éviter un problème de précision autour de $X = 2$, on préfère

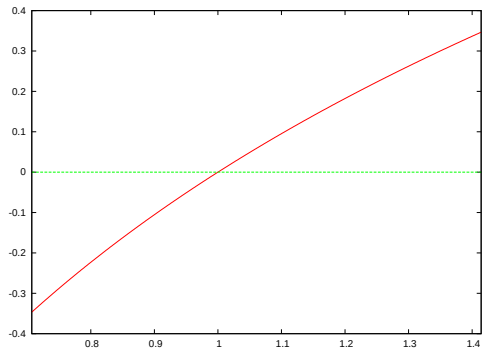
$$\begin{cases} E' = E & \text{et } Y = 1, F \quad \text{si} \quad m \leqslant \sqrt{2} \\ E' = E + 1 & \text{et } Y = \frac{1,F}{2} \quad \text{if} \quad m > \sqrt{2} \end{cases}$$

  - alors $\ln(X) = E' \cdot \ln(2) + \ln(Y)$
    - avec $\frac{\sqrt{2}}{2} \leqslant Y \leqslant \sqrt{2}$
    - (choisi pour que $\log(Y)$ soit centré autour de 0)

# Évaluation

On va donc calculer $\ln(X)$ comme $E' \cdot \ln(2) + \ln(Y)$

- $E' \cdot \ln(2)$ c'est un petit entier par une constante réelle
  - fastoche...
- Reste à calculer $\ln(Y)$ avec $\frac{\sqrt{2}}{2} \leqslant Y \leqslant \sqrt{2}$

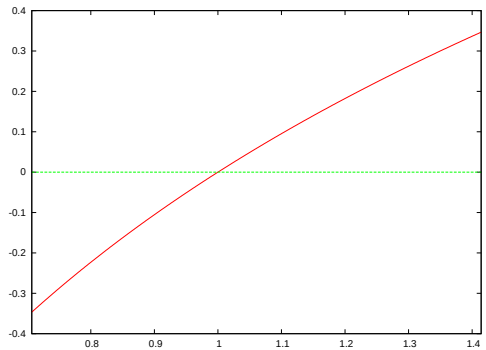



- Elle a l'air encore plus gentille comme cela. Approchons-là

# Évaluation

On va donc calculer $\ln(X)$ comme $E' \cdot \ln(2) + \ln(Y)$

- $E' \cdot \ln(2)$ c'est un petit entier par une constante réelle
  - fastoche...
- Reste à calculer $\ln(Y)$ avec $\frac{\sqrt{2}}{2} \leqslant Y \leqslant \sqrt{2}$




- Elle a l'air encore plus gentille comme cela.
  Approchons-là par un polynôme.
- Le couteau suisse à polynômes c'est Sollya
  (`http://sollya.gforge.inria.fr/`).
  Quel degré faut-il pour une approximation à $2^{-55}$ :
  ```
  > guessdegree(log(x), [1/sqrt(2);sqrt(2)], 1b-55);
  ```
  Sollya répond : degré 20.

# On le tient, notre algorithme ?

Un polynôme $p(x) = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n$ de degré $n$ s'évalue en $2n$ opérations par le schéma de Horner :

$$p(Y) = a_0 + Y \times (a_1 + Y \times (a_2 + Y \times (... + Y \times a_n))..)$$

En comptant 4 cycles l'opération flottante, cela nous fait dans les 160 cycles pour le polynôme, et 300 cycles en tout.

# On le tient, notre algorithme ?

Un polynôme $p(x) = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n$ de degré $n$ s'évalue en $2n$ opérations par le schéma de Horner :

$$p(Y) = a_0 + Y \times (a_1 + Y \times (a_2 + Y \times (... + Y \times a_n))..)$$

En comptant 4 cycles l'opération flottante, cela nous fait dans les 160 cycles pour le polynôme, et 300 cycles en tout.

C'est vendable. Peut-on mieux faire ?

# On le tient, notre algorithme ?

Un polynôme $p(x) = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n$ de degré $n$ s'évalue en $2n$ opérations par le schéma de Horner :

$$p(Y) = a_0 + Y \times (a_1 + Y \times (a_2 + Y \times (... + Y \times a_n))..)$$

En comptant 4 cycles l'opération flottante, cela nous fait dans les 160 cycles pour le polynôme, et 300 cycles en tout.

C'est vendable. Peut-on mieux faire ?

Oui, en brûlant de la mémoire.

# Seconde réduction d'argument à base de table

Donc on veut calculer $\ln(Y)$ avec $\frac{\sqrt{2}}{2} \leqslant Y \leqslant \sqrt{2}$

- Soit $i$ l'entier composé des $k$ bits de poids fort de $Y$.

# Seconde réduction d'argument à base de table

Donc on veut calculer $\ln(Y)$ avec $\frac{\sqrt{2}}{2} \leqslant Y \leqslant \sqrt{2}$

- Soit $i$ l'entier composé des $k$ bits de poids fort de $Y$.
- Lisons, dans une table indicée par $i$,

$$R[i] \approx \frac{1}{Y}$$

- Calculons

$$Z = Y \times R[i] - 1$$

# Seconde réduction d'argument à base de table

Donc on veut calculer $\ln(Y)$ avec $\frac{\sqrt{2}}{2} \leqslant Y \leqslant \sqrt{2}$

- Soit $i$ l'entier composé des $k$ bits de poids fort de $Y$.
- Lisons, dans une table indicée par $i$,

$$R[i] \approx \frac{1}{Y}$$

- Calculons

$$Z = Y \times R[i] - 1$$

- $Z$ est petit (on peut montrer que $|Z| < 2^{-k+1}$) et

$$\ln(Y) = \ln(1 + Z) - \ln(R[i])$$

# Seconde réduction d'argument à base de table

Donc on veut calculer $\ln(Y)$ avec $\frac{\sqrt{2}}{2} \leqslant Y \leqslant \sqrt{2}$

- Soit $i$ l'entier composé des $k$ bits de poids fort de $Y$.
- Lisons, dans une table indicée par $i$,

$$R[i] \approx \frac{1}{Y}$$

- Calculons

$$Z = Y \times R[i] - 1$$

- $Z$ est petit (on peut montrer que $|Z| < 2^{-k+1}$) et

$$\ln(Y) = \ln(1 + Z) - \ln(R[i])$$

- Précalculons et tabulons aussi, pour chaque valeur de $R[i]$,

$$L[i] = \ln(R[i])$$

- Et notre calcul devient

$$\ln(X) = E' \cdot \ln(2) - L[i] + \ln(1 + Z)$$

# Le log selon Tang

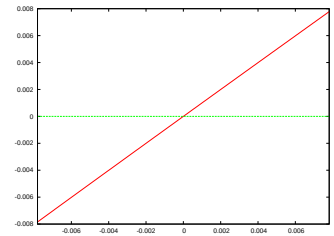
$$\ln(X) = E' \cdot \ln(2) - L[i] + \ln(1 + Z)$$

- Avec k=8 (une table à 256 entrées), on a $|Z| < 2^{-7}$

# Le log selon Tang

$$\ln(X) = E' \cdot \ln(2) - L[i] + \ln(1 + Z)$$

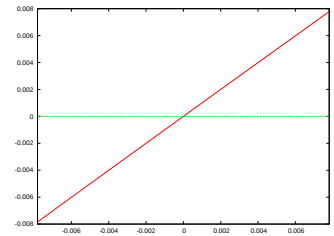
- Avec k=8 (une table à 256 entrées), on a $|Z| < 2^{-7}$
- La fonction est de plus en plus gentille.



# Le log selon Tang

$$\ln(X) = E' \cdot \ln(2) - L[i] + \ln(1 + Z)$$

- Avec k=8 (une table à 256 entrées), on a $|Z| < 2^{-7}$
- La fonction est de plus en plus gentille.

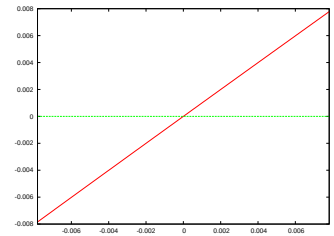


- Un polynôme de degré 6 fera l'affaire

# Le log selon Tang

$$\ln(X) = E' \cdot \ln(2) - L[i] + \ln(1 + Z)$$

- Avec k=8 (une table à 256 entrées), on a $|Z| < 2^{-7}$
- La fonction est de plus en plus gentille.

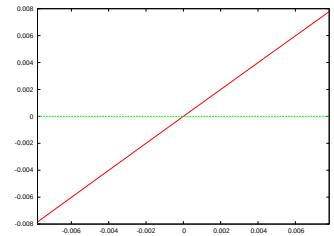


- Un polynôme de degré 6 fera l'affaire
- On a gagné 14× et 14+, au prix de
  - l'extraction de $i$
  - deux lectures de tables dans des tables à 256 entrées
  - le calcul de $Z = Y \times R[i] - 1$

# Le log selon Tang

$$\ln(X) = E' \cdot \ln(2) - L[i] + \ln(1 + Z)$$

- Avec k=8 (une table à 256 entrées), on a $|Z| < 2^{-7}$
- La fonction est de plus en plus gentille.



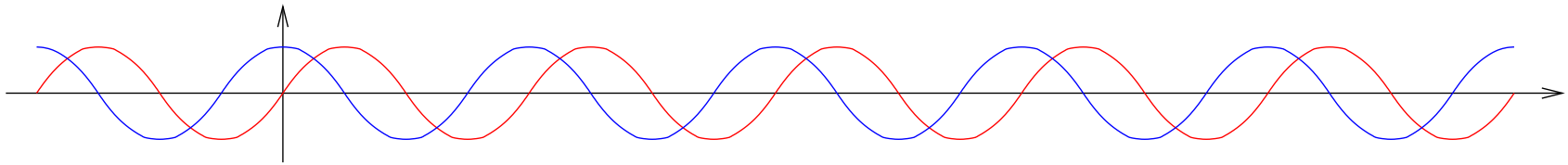
- Un polynôme de degré 6 fera l'affaire
- On a gagné 14× et 14+, au prix de
  - l'extraction de $i$
  - deux lectures de tables dans des tables à 256 entrées
  - le calcul de $Z = Y \times R[i] - 1$
- Remarque : on choisit $R[i]$ :
  - on peut le choisir tenant sur peu de bits (environ $k$ bits)
  - alors (TSVP) on peut calculer $Z$ exactement (sans arrondi)
  - Les deux réductions d'argument sont exactes

# Un peu d'adéquation tartine – confiture

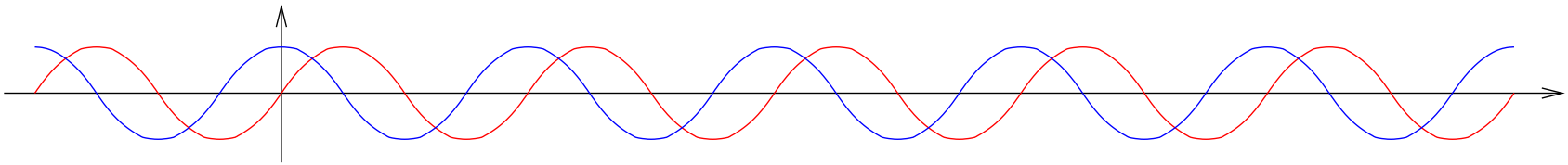
Deux manières de calculer $Z$ exactement, suivant votre processeur :

- Vous voulez calculer 24 bits de mantisse (flottant binary32) sur un processeur entier 32 bits
  - vous avez donc 8 bits de marge
  - si $R[i]$ tient sur 8 bits le produit $YR[i]$ tient sur 32 bits
- Vous voulez calculer 53 bits de mantisse sur un Pentium
  - vous pouvez utiliser du double-étendu (64 bits de mantisse)
  - vous avez donc 11 bits de marge

# Une fonction méchante en détail : le sinus

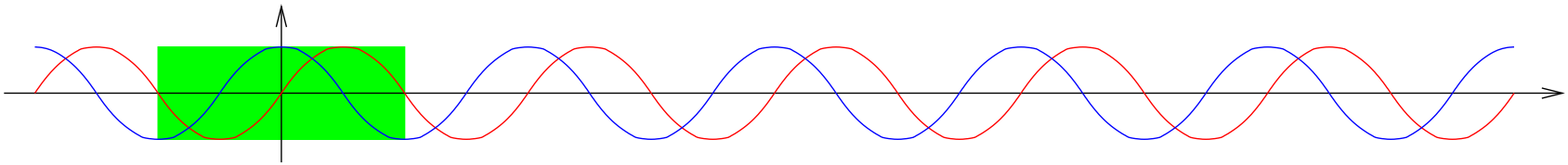


# Une fonction méchante en détail : le sinus



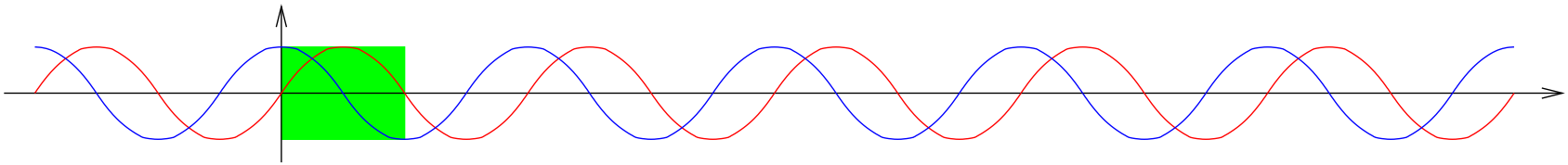
- Réduction d'argument par les identités trigonométriques

# Une fonction méchante en détail : le sinus



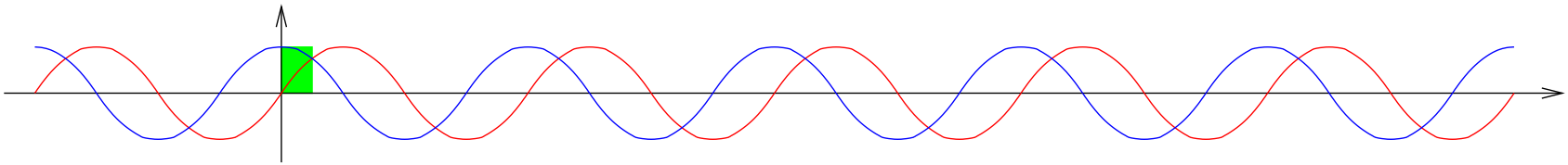
- Réduction d'argument par les identités trigonométriques
  - $\sin(X) = \sin(X + 2k\pi)$

# Une fonction méchante en détail : le sinus



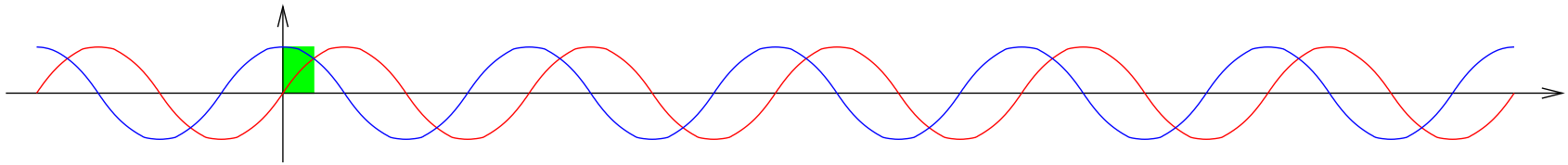
- Réduction d'argument par les identités trigonométriques
  - $\sin(X) = \sin(X + 2k\pi)$
  - $\sin(-X) = -\sin(X)$

# Une fonction méchante en détail : le sinus



- Réduction d'argument par les identités trigonométriques
  - $\sin(X) = \sin(X + 2k\pi)$
  - $\sin(-X) = -\sin(X)$
  - $\sin(\frac{\pi}{2} - X) = \cos(X)$

# Une fonction méchante en détail : le sinus

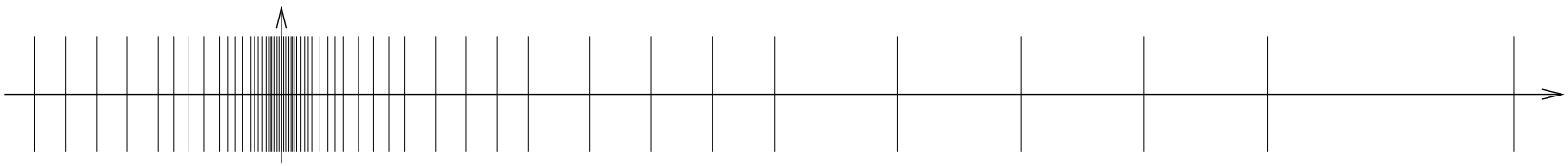


- Réduction d'argument par les identités trigonométriques
  - $\sin(X) = \sin(X + 2k\pi)$
  - $\sin(-X) = -\sin(X)$
  - $\sin(\frac{\pi}{2} - X) = \cos(X)$
  - ...
- ... et nous voilà dans $\left[-\frac{\pi}{4}, \frac{\pi}{4}\right]$, un petit voisinage de 0
  - avec une reconstruction sans calcul, c'est plus simple que le log
  - Yapuka faire une approximation polynomiale (ou un Taylor)

C'est plutôt gentil tout cela. Pourquoi elle est méchante ?

# Mais alors, pourquoi elle est méchante

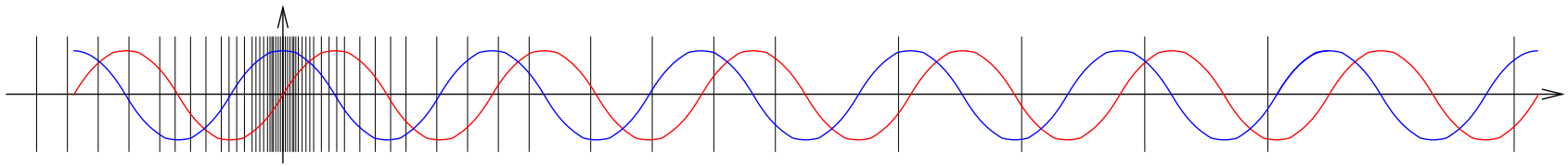
À cause de $\sin(1.00100100111101010001 \times 2^{42})$



- La virgule flottante est une représentation logarithmique en base 2

# Mais alors, pourquoi elle est méchante

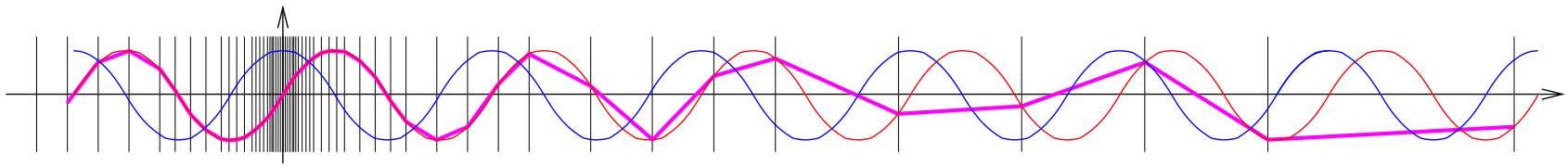
À cause de $\sin(1.00100100111101010001 \times 2^{42})$



- La virgule flottante est une représentation logarithmique en base 2
- Les sinus et cosinus de la `libm` prennent des arguments en radian
- donc leur période est $2\pi$

# Mais alors, pourquoi elle est méchante

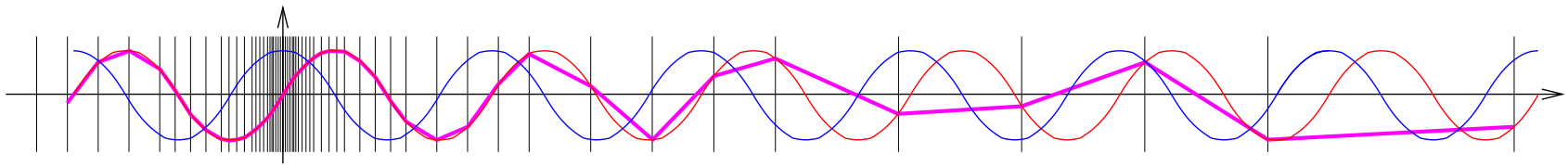
À cause de $\sin(1.00100100111101010001 \times 2^{42})$



- La virgule flottante est une représentation logarithmique en base 2
- Les sinus et cosinus de la `libm` prennent des arguments en radian
- donc leur période est $2\pi$
- Or 2 et $2\pi$ sont dans un rapport irrationnel

# Mais alors, pourquoi elle est méchante

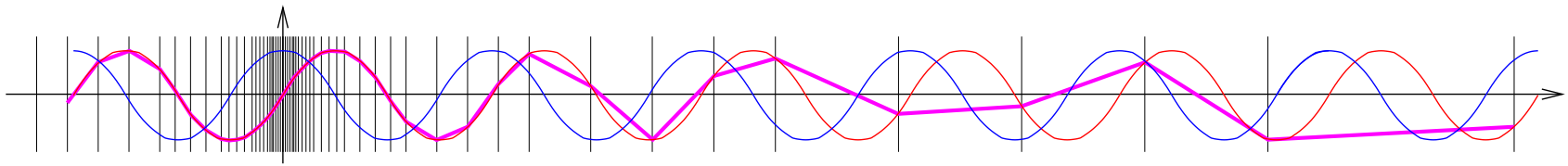
À cause de $\sin(1.00100100111101010001 \times 2^{42})$



- La virgule flottante est une représentation logarithmique en base 2
- Les sinus et cosinus de la `libm` prennent des arguments en radian
- donc leur période est $2\pi$
- Or 2 et $2\pi$ sont dans un rapport irrationnel
- Sur les grandes valeurs, sinus et cosinus ressemblent à des générateurs pseudo-aléatoires.

# Mais alors, pourquoi elle est méchante

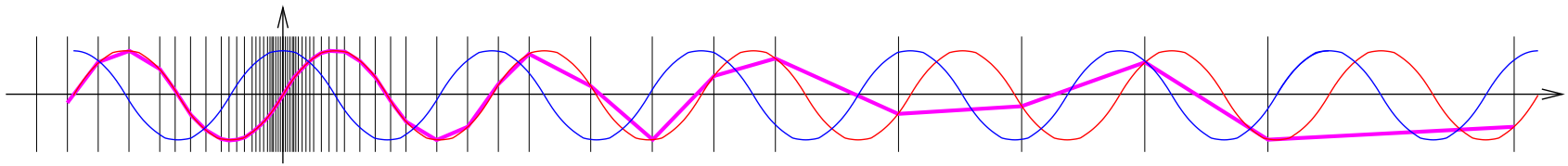
À cause de $\sin(1.00100100111101010001 \times 2^{42})$



- La virgule flottante est une représentation logarithmique en base 2
- Les sinus et cosinus de la `libm` prennent des arguments en radian
- donc leur période est $2\pi$
- Or 2 et $2\pi$ sont dans un rapport irrationnel
- Sur les grandes valeurs, sinus et cosinus ressemblent à des générateurs pseudo-aléatoires.
- Une solution (x87)
  - On ne garantit rien pour les grandes valeurs ($x > 2^{64}$ en x87)

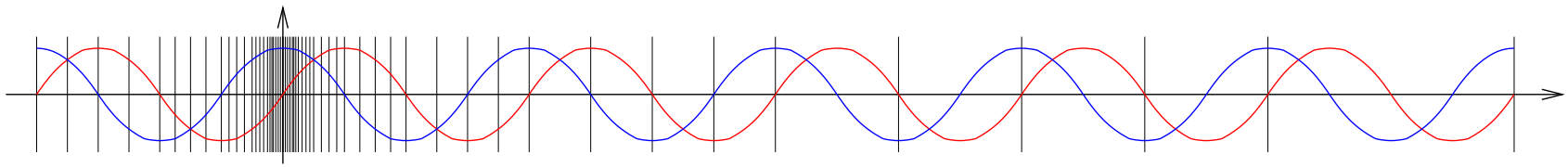
# Mais alors, pourquoi elle est méchante

À cause de $\sin(1.00100100111101010001 \times 2^{42})$



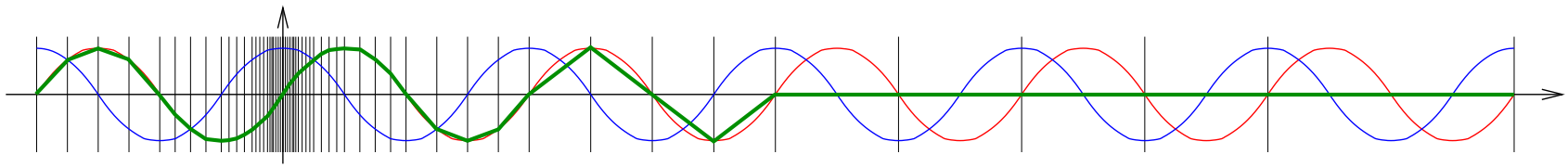
- La virgule flottante est une représentation logarithmique en base 2
- Les sinus et cosinus de la `libm` prennent des arguments en radian
- donc leur période est $2\pi$
- Or 2 et $2\pi$ sont dans un rapport irrationnel
- Sur les grandes valeurs, sinus et cosinus ressemblent à des générateurs pseudo-aléatoires.
- Une solution (x87)
  - On ne garantit rien pour les grandes valeurs ($x > 2^{64}$ en x87)
  - (un programme qui demande $\sin(x)$ avec $x > 2^{64}$ est sans doute dans les choux depuis longtemps)

# Une autre solution : $\sin(\pi x)$ et $\cos(\pi x)$



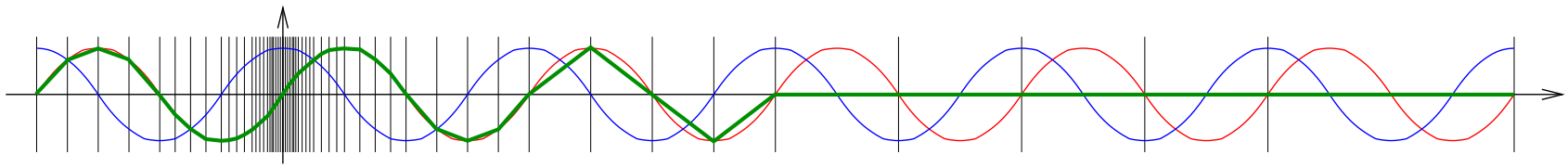
- À présent la période est 2

# Une autre solution : $\sin(\pi x)$ et $\cos(\pi x)$



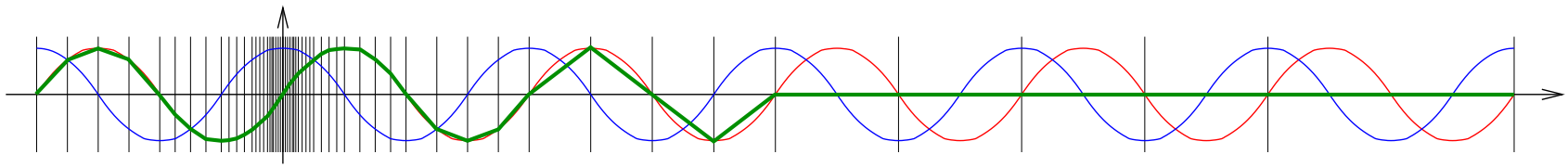
- À présent la période est 2
- Comportement pour les grands arguments :
  - pas plus informatif que le précédent
  - mais plus facile à calculer
  - et le résultat est exact (pas d'irrationnel)

# Une autre solution : $\sin(\pi x)$ et $\cos(\pi x)$



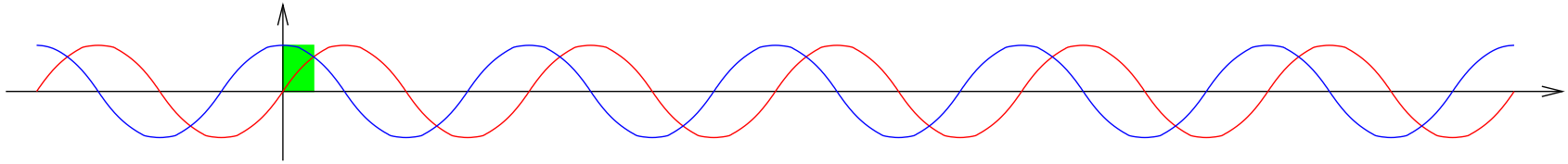
- À présent la période est 2
- Comportement pour les grands arguments :
  - pas plus informatif que le précédent
  - mais plus facile à calculer
  - et le résultat est exact (pas d'irrationnel)
- Remarque : $\pi x$ ou $\frac{\pi}{180}x$ c'est kif-kif
  - en prime, on a les arguments en degrés

# Une autre solution : $\sin(\pi x)$ et $\cos(\pi x)$



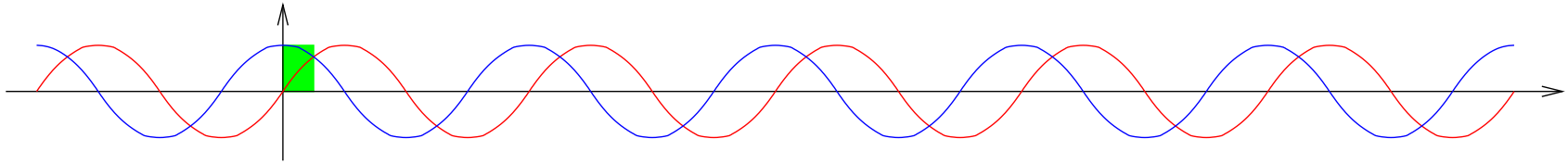
- À présent la période est 2
- Comportement pour les grands arguments :
  - pas plus informatif que le précédent
  - mais plus facile à calculer
  - et le résultat est exact (pas d'irrationnel)
- Remarque : $\pi x$ ou $\frac{\pi}{180}x$ c'est kif-kif
  - en prime, on a les arguments en degrés
- Sexe des anges : quelle doit être $\sin(\pi x)$ pour $x = \pm\infty$ ?
  - 0 protège les programmes qui étaient déjà dans les choux
  - NaN protège notre conscience mathématique

# Une troisième solution : au boulot



Goal : replace $X \in \mathbb{R}$ with angle $\alpha$ in $\left[0, \frac{\pi}{4}\right)$

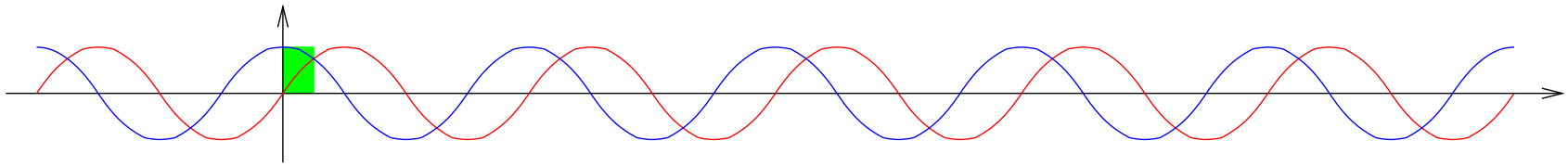
# Une troisième solution : au boulot



Goal : replace $X \in \mathbb{R}$ with angle $\alpha$ in $\left[0, \frac{\pi}{4}\right)$

- compute $K = X \times \frac{4}{\pi}$

# Une troisième solution : au boulot

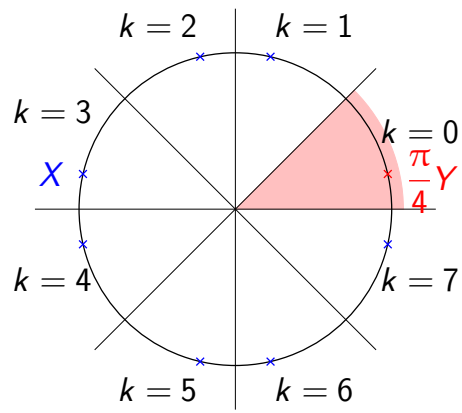


Goal : replace $X \in \mathbb{R}$ with angle $\alpha$ in $\left[0, \frac{\pi}{4}\right)$

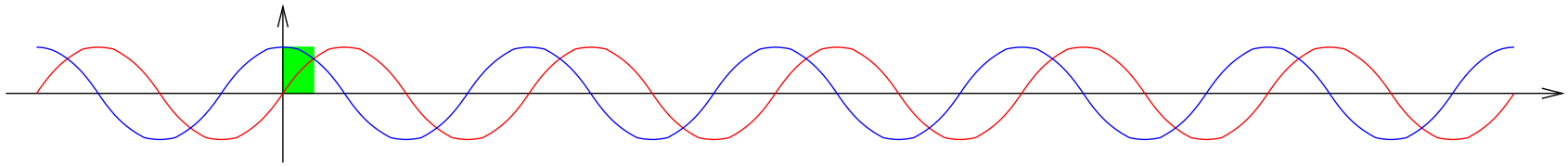
- compute $K = X \times \frac{4}{\pi}$
- integer part $\mathrm{int}(K)$ provides the octant $k$
  - keep only the 3 least significant bits
- fraction part $\mathrm{frac}(K)$ is used to build

$$Y = \begin{cases} \mathrm{frac}(K) & \text{when } k \text{ is even} \\ 1 - \mathrm{frac}(K) & \text{when } k \text{ is odd} \end{cases}$$

- then $\alpha = \frac{\pi}{4} Y$



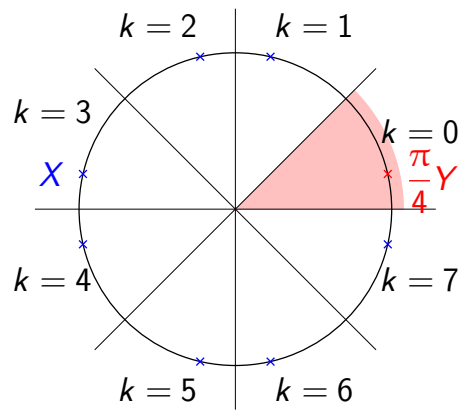
# Une troisième solution : au boulot



Goal : replace $X \in \mathbb{R}$ with angle $\alpha$ in $\left[0, \frac{\pi}{4}\right)$

- compute $K = X \times \frac{4}{\pi}$
- integer part $\mathrm{int}(K)$ provides the octant $k$
  - keep only the 3 least significant bits
- fraction part $\mathrm{frac}(K)$ is used to build

$$Y = \begin{cases} \mathrm{frac}(K) & \text{when } k \text{ is even} \\ 1 - \mathrm{frac}(K) & \text{when } k \text{ is odd} \end{cases}$$




- then $\alpha = \frac{\pi}{4} Y$
- $\sin X$ and $\cos X$ rebuilt out of $k$, $\sin\left(\frac{\pi}{4} Y\right)$, $\cos\left(\frac{\pi}{4} Y\right)$

# Combien elle coûte ?

Payne et Hanek :

- On doit calculer $K = X \times \frac{4}{\pi}$

# Combien elle coûte ?

Payne et Hanek :

- On doit calculer $K = X \times \frac{4}{\pi}$
- mis on va garder seulement les 3 derniers bits de la partie entière, et “assez de bits” de la partie fractionnaire
- n'utilisons que les bits de $\frac{4}{\pi}$ nécessaires pour obtenir cette information

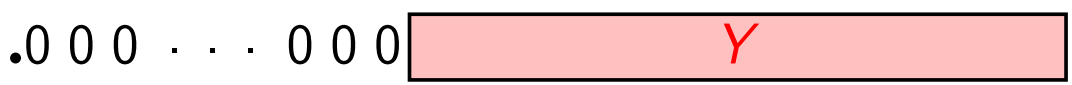
Réduction d'argument précise sur tout le domaine des flottants :

- Il faut stocker $2^{w_E-1} + 2w_F$ bits de $\frac{4}{\pi}$
- et en extraire $3w_F$ bits (pourquoi $3w_F$ ? TSVP)
  - binary32 : 180 bits stockés, 75 bits extraits
  - binary64 : 2200 bits stockés, 160 bits extraits
- puis les multiplier par la mantisse de $w_F$ bits

Tout cela parce que $\frac{4}{\pi}$ est irrationnel.

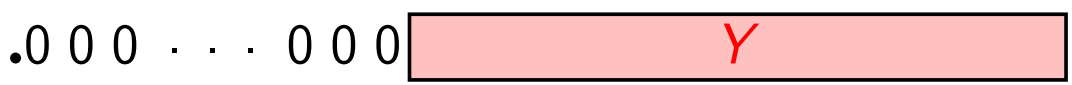
# Pourquoi $3w_F$ ?

- $Y = \mathrm{frac}(K)$ may come very close to 0
  - example : binary32, $X = 16367173 \cdot 2^{72}$
    we get $Y = 1{,}10380056... \cdot 2^{-29}$



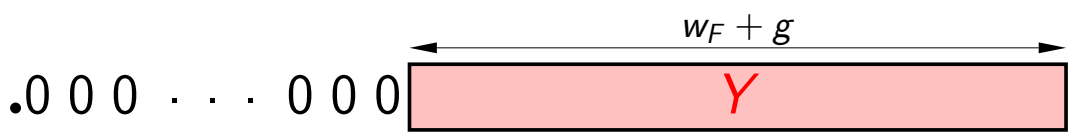
# Pourquoi $3w_F$ ?

- $Y = \mathrm{frac}(K)$ may come very close to 0
  - example : binary32, $X = 16367173 \cdot 2^{72}$
    we get $Y = 1{,}10380056... \cdot 2^{-29}$
- Then, $\sin\left(\frac{\pi}{4}Y\right) \approx \frac{\pi}{4}Y$ (Taylor), so will also be very small



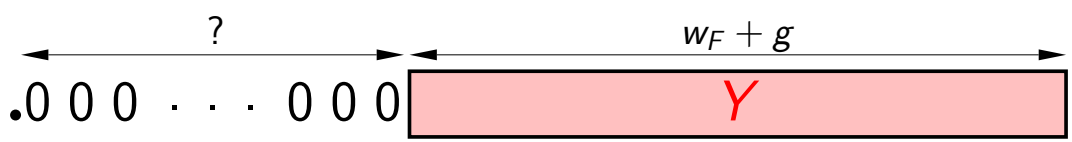
# Pourquoi $3w_F$ ?

- $Y = \mathrm{frac}(K)$ may come very close to 0
  - example : binary32, $X = 16367173 \cdot 2^{72}$
    we get $Y = 1{,}10380056... \cdot 2^{-29}$
- Then, $\sin\left(\frac{\pi}{4}Y\right) \approx \frac{\pi}{4}Y$ (Taylor), so will also be very small
- To compute the normalized mantissa of a floating-point sine, we need the $w_F + g$ first significant bits.



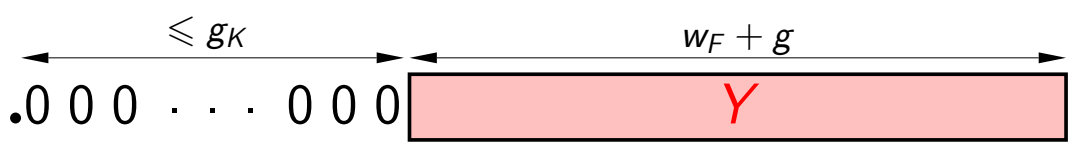
# Pourquoi $3w_F$ ?

- $Y = \mathrm{frac}(K)$ may come very close to 0
  - example : binary32, $X = 16367173 \cdot 2^{72}$
    we get $Y = 1{,}10380056... \cdot 2^{-29}$
- Then, $\sin\left(\frac{\pi}{4}Y\right) \approx \frac{\pi}{4}Y$ (Taylor), so will also be very small
- To compute the normalized mantissa of a floating-point sine, we need the $w_F + g$ first significant bits.
- How many leading zeroes may appear ?



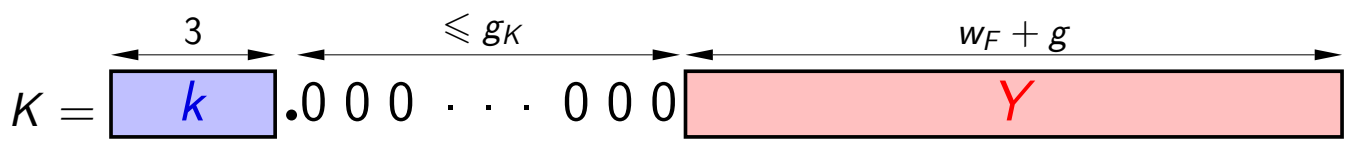
# Pourquoi $3w_F$ ?

- $Y = \mathrm{frac}(K)$ may come very close to 0
  - example : binary32, $X = 16367173 \cdot 2^{72}$
    we get $Y = 1{,}10380056... \cdot 2^{-29}$
- Then, $\sin\left(\frac{\pi}{4}Y\right) \approx \frac{\pi}{4}Y$ (Taylor), so will also be very small
- To compute the normalized mantissa of a floating-point sine, we need the $w_F + g$ first significant bits.
- How many leading zeroes may appear ?
  - Kahan/Douglas algorithm computes a bound $g_K$
    for a given floating-point format
  - rule of thumb : $g_k \approx w_F$ (the mantissa size)



# Pourquoi $3w_F$ ?

- $Y = \mathrm{frac}(K)$ may come very close to 0
  - example : binary32, $X = 16367173 \cdot 2^{72}$
    we get $Y = 1{,}10380056... \cdot 2^{-29}$
- Then, $\sin\left(\frac{\pi}{4}Y\right) \approx \frac{\pi}{4}Y$ (Taylor), so will also be very small
- To compute the normalized mantissa of a floating-point sine, we need the $w_F + g$ first significant bits.
- How many leading zeroes may appear ?
  - Kahan/Douglas algorithm computes a bound $g_K$
    for a given floating-point format
  - rule of thumb : $g_k \approx w_F$ (the mantissa size)
- therefore at least $3 + w_F + g_K + g$ correct bits of the product $K = X \times \frac{4}{\pi}$ need to be computed



# Accurate argument reduction (Payne and Hanek)

$$K = \frac{4}{\pi} \times X$$

# Accurate argument reduction (Payne and Hanek)

$$K = \frac{4}{\pi} \times 2^{E_X} \times 1, F_X$$

# Accurate argument reduction (Payne and Hanek)

$$K = \frac{4}{\pi} \times 2^{E_X} \times 1, F_X$$

# Accurate argument reduction (Payne and Hanek)

$$K = \frac{4}{\pi} \times 2^{E_X} \times 1, F_X$$

- Very large multiplication
  - on one side, $\frac{4}{\pi}$ shifted left by the exponent of $X$
    - (at most $2^{w_E-1}$ positions, where $w_E$ is the exponent size)
  - on the other side, the $w_F$-bit mantissa of $X$

# Accurate argument reduction (Payne and Hanek)

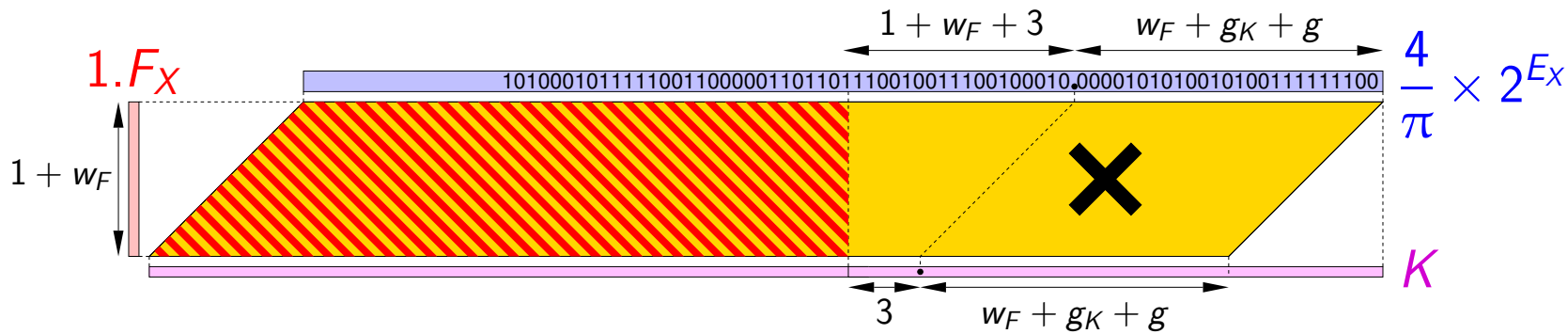
$$K = \frac{4}{\pi} \times 2^{E_X} \times 1, F_X$$

- Very large multiplication
  - on one side, $\frac{4}{\pi}$ shifted left by the exponent of $X$
    - (at most $2^{w_E-1}$ positions, where $w_E$ is the exponent size)
  - on the other side, the $w_F$-bit mantissa of $X$
- Fortunately we need to compute only
  - 3 bits to the left of the point (the octant $k$)
  - $w_F + g_K + g$ bits to the right of the point (frac($K$))

# Accurate argument reduction (Payne and Hanek)

$$K = \frac{4}{\pi} \times 2^{E_X} \times 1, F_X$$

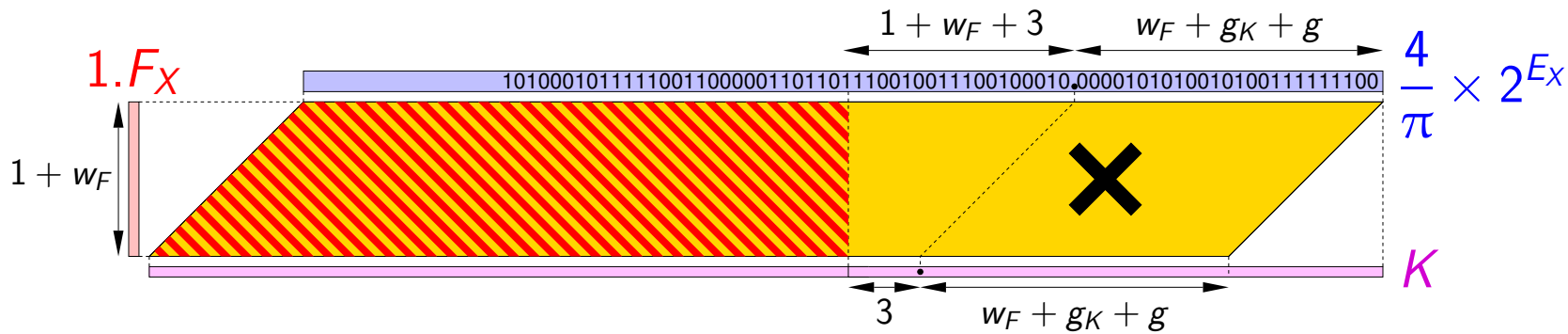
- Very large multiplication
  - on one side, $\frac{4}{\pi}$ shifted left by the exponent of $X$
    - (at most $2^{w_E-1}$ positions, where $w_E$ is the exponent size)
  - on the other side, the $w_F$-bit mantissa of $X$
- Fortunately we need to compute only
  - 3 bits to the left of the point (the octant $k$)
  - $w_F + g_K + g$ bits to the right of the point (frac($K$))
- no need to compute the left part of the multiplication



# Accurate argument reduction (Payne and Hanek)

$$K = \frac{4}{\pi} \times 2^{E_X} \times 1, F_X$$

- Very large multiplication
  - on one side, $\frac{4}{\pi}$ shifted left by the exponent of $X$
    - (at most $2^{w_E-1}$ positions, where $w_E$ is the exponent size)
  - on the other side, the $w_F$-bit mantissa of $X$
- Fortunately we need to compute only
  - 3 bits to the left of the point (the octant $k$)
  - $w_F + g_K + g$ bits to the right of the point (frac($K$))
- no need to compute the left part of the multiplication




- Need to store $2^{2^{w_E-1}} + 3 + w_F + g_K + g$ bits of $\frac{4}{\pi}$

# Ensuite c'est pareil que le log

- On peut évaluer sin et cos directement sur $[0, \pi/4]$
  - gros polynôme
- On peut faire une seconde réduction d'argument à base de table
  - $\alpha = a + b$
  - $a$ est composé des $k$ bits de poids fort de $\alpha$
  - $b$ est composé des bits de poids faible, et est donc petit
  - $\sin(a + b) = \sin a \cos b + \sin b \cos a$
- Dans CRLibm on bricole la première réduction d'argument pour qu'elle donne directement $a$ et $b$

# Conclusion de cette partie

On aura besoin, pour évaluer des fonctions élémentaires,

- de touiller des bits
  - extraire exposant et mantisse,
  - extraire les premiers bits de la mantisse (index),
  - extraire les bits qu'il faut de $4/\pi$
  - ...
  - et parfois le contraire pour réassembler le résultat

# Conclusion de cette partie

On aura besoin, pour évaluer des fonctions élémentaires,

- de touiller des bits
  - extraire exposant et mantisse,
  - extraire les premiers bits de la mantisse (index),
  - extraire les bits qu'il faut de $4/\pi$
  - ...
  - et parfois le contraire pour réassembler le résultat
- d'évaluer des polynômes en flottant

# Conclusion de cette partie

On aura besoin, pour évaluer des fonctions élémentaires,

- de touiller des bits
  - extraire exposant et mantisse,
  - extraire les premiers bits de la mantisse (index),
  - extraire les bits qu'il faut de $4/\pi$
  - ...
  - et parfois le contraire pour réassembler le résultat
- d'évaluer des polynômes en flottant
- et de calculer en précision plus grande que celle du résultat final

# Au fait

On pourrait faire des approx rationnelles ?
Bof car la division coûte 5 à 10 fois plus cher que l'addition.
Par expérience :

# Au fait

On pourrait faire des approx rationnelles ?
Bof car la division coûte 5 à 10 fois plus cher que l'addition.
Par expérience :

- Pour les fonctions gentilles, on ne diminue pas vraiment le nombre d'opérations, on remplace juste des multiplications par des divisions.

# Au fait

On pourrait faire des approx rationnelles ?
Bof car la division coûte 5 à 10 fois plus cher que l'addition.
Par expérience :

- Pour les fonctions gentilles, on ne diminue pas vraiment le nombre d'opérations, on remplace juste des multiplications par des divisions.
- Quand on diminue vraiment le nombre d'opérations (par exemple asymptote verticale), il vaut mieux mettre la division dans une réduction d'argument bien choisie.

# L'arithmétique de mon PC

# On n'est pas aidé

Ouvrons un manuel de programmation pour un langage populaire...
Types de base du C :

- `char` (abbréviation de *character*, un terme typographique)
  - un entier signé sur 8 bits

# On n'est pas aidé

Ouvrons un manuel de programmation pour un langage populaire...
Types de base du C :

- `char` (abbréviation de *character*, un terme typographique)
  - un entier signé sur 8 bits (donc pas vraiment un caractère)

# On n'est pas aidé

Ouvrons un manuel de programmation pour un langage populaire...
Types de base du C :

- `char` (abbréviation de *character*, un terme typographique)
  - un entier signé sur 8 bits (donc pas vraiment un caractère)
  - Non seulement on peut additionner des caractères ☺

# On n'est pas aidé

Ouvrons un manuel de programmation pour un langage populaire...
Types de base du C :

- `char` (abbréviation de *character*, un terme typographique)
  - un entier signé sur 8 bits (donc pas vraiment un caractère)
  - Non seulement on peut additionner des caractères 2
  - mais en plus, 127+1 = -128

# On n'est pas aidé

Ouvrons un manuel de programmation pour un langage populaire...
Types de base du C :

- `char` (abbréviation de *character*, un terme typographique)
  - un entier signé sur 8 bits (donc pas vraiment un caractère)
  - Non seulement on peut additionner des caractères 7 0
  - mais en plus, 127+1 = -128
  - d'ailleurs il y a des `unsigned char`

# On n'est pas aidé

Ouvrons un manuel de programmation pour un langage populaire...
Types de base du C :

- `char` (abbréviation de *character*, un terme typographique)
  - un entier signé sur 8 bits (donc pas vraiment un caractère)
  - Non seulement on peut additionner des caractères 2
  - mais en plus, 127+1 = -128
  - d'ailleurs il y a des `unsigned char`,
  - logiquement caractérisés par 255+1 = 0 ...

# On n'est pas aidé

Ouvrons un manuel de programmation pour un langage populaire...
Types de base du C :

- `char` (abbréviation de *character*, un terme typographique)
  - un entier signé sur 8 bits (donc pas vraiment un caractère)
  - Non seulement on peut additionner des caractères 2 0
  - mais en plus, 127+1 = -128
  - d'ailleurs il y a des `unsigned char`,
  - logiquement caractérisés par 255+1 = 0 ...
- `int` (abbréviation de *integer*, un terme mathématique)
  - Un entier signé sur 8, 16, 32 ou 64 bits

# On n'est pas aidé

Ouvrons un manuel de programmation pour un langage populaire...
Types de base du C :

- `char` (abbréviation de *character*, un terme typographique)
  - un entier signé sur 8 bits (donc pas vraiment un caractère)
  - Non seulement on peut additionner des caractères 2
  - mais en plus, 127+1 = -128
  - d'ailleurs il y a des `unsigned char`,
  - logiquement caractérisés par 255+1 = 0 ...
- `int` (abbréviation de *integer*, un terme mathématique)
  - Un entier signé sur 8, 16, 32 ou 64 bits
  - (donc pas vraiment un entier)

# On n'est pas aidé

Ouvrons un manuel de programmation pour un langage populaire...
Types de base du C :

- `char` (abbréviation de *character*, un terme typographique)
  - un entier signé sur 8 bits (donc pas vraiment un caractère)
  - Non seulement on peut additionner des caractères 2
  - mais en plus, 127+1 = -128
  - d'ailleurs il y a des `unsigned char`,
  - logiquement caractérisés par 255+1 = 0 ...
- `int` (abbréviation de *integer*, un terme mathématique)
  - Un entier signé sur 8, 16, 32 ou 64 bits
  - (donc pas vraiment un entier)
- `float` (un terme informatique, enfin)
  - un flottant 32 bits

# On n'est pas aidé

Ouvrons un manuel de programmation pour un langage populaire...
Types de base du C :

- `char` (abbréviation de *character*, un terme typographique)
  - un entier signé sur 8 bits (donc pas vraiment un caractère)
  - Non seulement on peut additionner des caractères @
  - mais en plus, 127+1 = -128
  - d'ailleurs il y a des `unsigned char`,
  - logiquement caractérisés par 255+1 = 0 ...
- `int` (abbréviation de *integer*, un terme mathématique)
  - Un entier signé sur 8, 16, 32 ou 64 bits
  - (donc pas vraiment un entier)
- `float` (un terme informatique, enfin)
  - un flottant 32 bits
- `double` (un terme rien du tout)
  - un flottant 64 bits
  - pour les flottants 80 bits essayez `long double` (ha ha)

# Blague à part

Les normes (C99, Fortran 2002, IEEE-754) définissent les choses de plus en plus précisément.

# Blague à part

Les normes (C99, Fortran 2002, IEEE-754) définissent les choses de plus en plus précisément.
Mais je suis pas là pour vous casser les pieds avec des normes.

Toutefois je vais (pour la première fois) utiliser le vocabulaire suivant (de IEEE-754-2008) :

- pour un flottant simple précision je dirai binary32
- pour un flottant double précision je dirai binary64

# Opérations sur ces types de base supportées en matériel

- Entiers (32 ou 64 bits)
  - addition/soustraction, opérations logiques, décalages : en un cycle
  - multiplication en quelques cycles, pipelinée
    - (parfois prise en charge par le pipeline flottant)
  - parfois division/modulo en encore plus de cycles, pas pipelinée

# Opérations sur ces types de base supportées en matériel

- Entiers (32 ou 64 bits)
  - addition/soustraction, opérations logiques, décalages : en un cycle
  - multiplication en quelques cycles, pipelinée
    - (parfois prise en charge par le pipeline flottant)
  - parfois division/modulo en encore plus de cycles, pas pipelinée
- Virgule flottante (32 ou 64 bits)
  - addition en 3-5 cycles, pipelinés
  - multiplication, idem
  - Parfois, *Fused Multiply and Add* en 4 à 6 cycles
  - Parfois, division, racine carrée en qq dizaines de cycles, pas pipelinée
  - parfois (IA32), fonctions élémentaires en microcode, qq centaines de cycles, pas pipelinées

# FMA, l'opérateur couteau suisse

*(Fused Multiply-and-Add)*

- Calcule $\mathrm{RN}(a \times b + c)$

# FMA, l'opérateur couteau suisse

*(Fused Multiply-and-Add)*

- Calcule $\mathrm{RN}(a \times b + c)$
- Deux opérations par instruction (et pour le prix d'une) : plus rapide

# FMA, l'opérateur couteau suisse

*(Fused Multiply-and-Add)*

- Calcule $\mathrm{RN}(a \times b + c)$
- Deux opérations par instruction (et pour le prix d'une) : plus rapide
- Un seul arrondi : plus précis

# FMA, l'opérateur couteau suisse

*(Fused Multiply-and-Add)*

- Calcule $\mathrm{RN}(a \times b + c)$
- Deux opérations par instruction (et pour le prix d'une) : plus rapide
- Un seul arrondi : plus précis
- Permet bien sûr de calculer $a + c$ et $a \times b$

# FMA, l'opérateur couteau suisse

*(Fused Multiply-and-Add)*

- Calcule $\mathrm{RN}(a \times b + c)$
- Deux opérations par instruction (et pour le prix d'une) : plus rapide
- Un seul arrondi : plus précis
- Permet bien sûr de calculer $a + c$ et $a \times b$
- Permet de calculer $a/b$ et $\sqrt{x}$ en quelques itérations
  - latence comparable à un diviseur matériel
  - plus grande flexibilité (choix entre latence et débit par ex.)

# FMA, l'opérateur couteau suisse

*(Fused Multiply-and-Add)*

- Calcule $\mathrm{RN}(a \times b + c)$
- Deux opérations par instruction (et pour le prix d'une) : plus rapide
- Un seul arrondi : plus précis
- Permet bien sûr de calculer $a + c$ et $a \times b$
- Permet de calculer $a/b$ et $\sqrt{x}$ en quelques itérations
  - latence comparable à un diviseur matériel
  - plus grande flexibilité (choix entre latence et débit par ex.)
- Mais : va casser quelques propriétés mathématiques espérées :
  - $\sqrt{a^2 + b^2}$ devient asymétrique
  - `if` $b^2 \geqslant 4ac$ `then` (...) $\sqrt{b^2 - 4ac}$

# FMA, l'opérateur couteau suisse

*(Fused Multiply-and-Add)*

- Calcule $\mathrm{RN}(a \times b + c)$
- Deux opérations par instruction (et pour le prix d'une) : plus rapide
- Un seul arrondi : plus précis
- Permet bien sûr de calculer $a + c$ et $a \times b$
- Permet de calculer $a/b$ et $\sqrt{x}$ en quelques itérations
  - latence comparable à un diviseur matériel
  - plus grande flexibilité (choix entre latence et débit par ex.)
- Mais : va casser quelques propriétés mathématiques espérées :
  - $\sqrt{a^2 + b^2}$ devient asymétrique
  - `if` $b^2 \geqslant 4ac$ `then` (...) $\sqrt{b^2 - 4ac}$

IEEE-754-2008 standardise le FMA.

# Le flottant IA32 (AMD ou Intel) en 2009

- unité historique x87
  - formats binary80, binary64 et binary32
  - Pour les opérations, arrondi en binary32 ou binary64 de la mantisse mais avec l'exposant de binary80.
  - transferts flottant/entier coûteux (par la mémoire)

# Le flottant IA32 (AMD ou Intel) en 2009

- unité historique x87
  - formats binary80, binary64 et binary32
  - Pour les opérations, arrondi en binary32 ou binary64 de la mantisse mais avec l'exposant de binary80.
  - transferts flottant/entier coûteux (par la mémoire)
- de plus en plus obsolète, utilisée uniquement pour le binary80.
  - entre autres pour les fonctions élémentaires :
    - calculs intermédiaires en binary80
    - bonne précision finale en binary64

# Le flottant IA32 (AMD ou Intel) en 2009

- unité historique x87
  - formats binary80, binary64 et binary32
  - Pour les opérations, arrondi en binary32 ou binary64 de la mantisse mais avec l'exposant de binary80.
  - transferts flottant/entier coûteux (par la mémoire)
- de plus en plus obsolète, utilisée uniquement pour le binary80.
  - entre autres pour les fonctions élémentaires :
    - calculs intermédiaires en binary80
    - bonne précision finale en binary64
- unité SSE2 (et suivantes)
  - calcul SIMD en parallèle sur 2 binary64 ou 4 binary32
  - vrai arrondi propre, support matériel des sous-normaux
  - Flottant : ADD, SUB, MUL, DIV
  - Plus des instructions entières et logiques (AND OR XOR, shifts) sur les mêmes registres XMM
  - Plus des instructions de chargement/déplacements/conversions

# IA32 dans les détails

## Intel LIBM design

- Architecture
  - IA-32 instructions set
    - General Purpose
      - 32-bit GP registers
    - x87 FPU
      - 80-bit FP registers

| General Purpose | |
|---|---|
| Arithmetic | ADD, SUB |
| Logical | OR, AND, XOR |
| Shifts | SHL, SHR, SAR |
| Tests | CMP, TEST |
| Loads | MOV |

| x87 FPU | |
|---|---|
| Load | FLD |
| Arithmetic | FADD, FMUL |
| Remainder | FPREM1 |

Transparent extrait de la présentation de Nikita Astafiev à RNC7 (2006)

`http://rnc7.loria.fr/`

# IA32 dans les détails

## Intel LIBM design

- Architecture
  - IA-32 instructions set
    - General Purpose
    - x87 FPU
    - SSE/SSE2/SSE3…
      - 128-bit registers
      - 2 double FP
      - 4 single FP

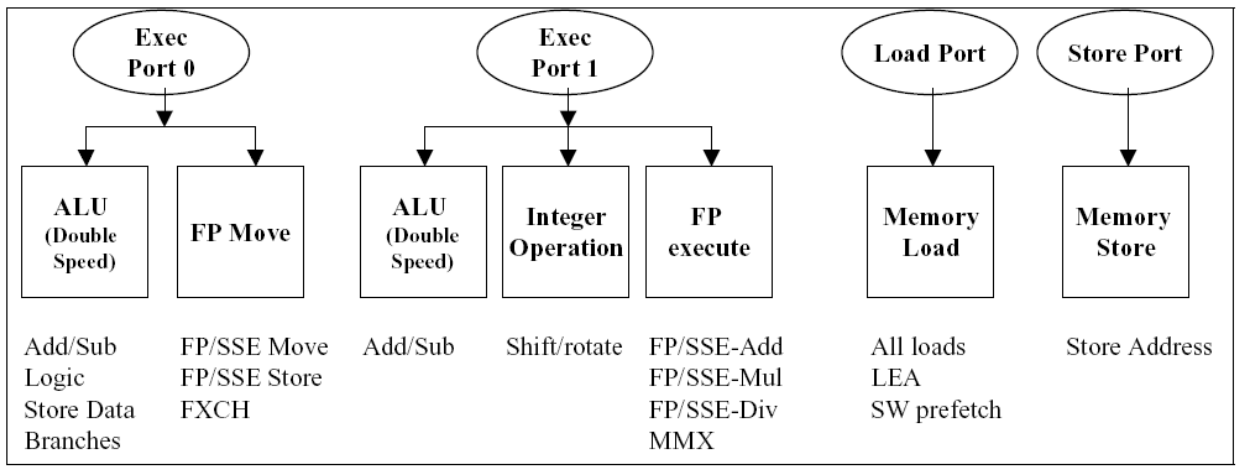
| SSE/SSE2/SSE3 | Double Precision | Single Precision |
|---|---|---|
| FP SIMD/scalar add, subtract | ADDPD/SD SUBPD/SD | ADDPS/SS SUBPS/SS |
| FP SIMD/scalar multiply, divide | MULPD/SD DIVPD/SD | MULPS/SS DIVPS/SS |
| Logical | ORPD, ANDPD, XORPD | ORPS, ANDPS, XORPS |
| Load one element<br>Load 128 bit | MOVSD<br>MOVAPD | MOVSS<br>MOVAPS |
| Pack | MOVDDUP | |
| Pack / unpack | PSHUFD | |
| Transfers to / from GPR | MOVD, PEXTRW, PINSRW | |
| Format conversions | CVTSS2SD, CVTSD2SS | |
| Logical shifts | PSRLQ, PSLLQ, PSRLD, PSLLD | |

Transparent extrait de la présentation de Nikita Astafiev à RNC7 (2006)

`http://rnc7.loria.fr/`

## Intel LIBM design

- Architecture
  - IA-32 instructions set
    - General Purpose
    - x87 FPU
    - SSE/SSE2/SSE3…
  - Instruction-level parallelism




Transparent extrait de la présentation de Nikita Astafiev à RNC7 (2006)

`http://rnc7.loria.fr/`

# Le flottant IA32 (AMD ou Intel) après 2009

Dans les cartons :

- Un produit scalaire introduit par Intel (SSE4), avec arrondis intermédiaires.
- Le FMA arrive avec des extensions AMD (SSE5) et Intel (AVX) incompatibles
- Et de plus en plus d'instructions d'emballage/déballage qui accéléreront la gestion des cas exceptionnels.

# La famille Power/PowerPC

Les Macs jusqu'à récemment, mais aussi les serveurs IBM dont les supercalculateurs en places 1, 4 et 5 du Top500 actuel

- Deux unités FMA par cœur
  - perf crête 4 FLOP binary64 par cycle
  - donc mieux que SSE2

# La famille Power/PowerPC

Les Macs jusqu'à récemment, mais aussi les serveurs IBM dont les supercalculateurs en places 1, 4 et 5 du Top500 actuel

- Deux unités FMA par cœur
  - perf crête 4 FLOP binary64 par cycle
  - donc mieux que SSE2
- Pas de diviseur

# La famille Power/PowerPC

Les Macs jusqu'à récemment, mais aussi les serveurs IBM
dont les supercalculateurs en places 1, 4 et 5 du Top500 actuel

- Deux unités FMA par cœur
  - perf crête 4 FLOP binary64 par cycle
  - donc mieux que SSE2
- Pas de diviseur
- Parenthèse : unité flottante décimale dans le Power6
  - ils ne savent vraiment pas quoi faire de leurs transistors
  - Intel pense qu'il vaut mieux le laisser au soft
  - Devinez à qui l'histoire donnera raison ?

# La famille Power/PowerPC

Les Macs jusqu'à récemment, mais aussi les serveurs IBM
dont les supercalculateurs en places 1, 4 et 5 du Top500 actuel

- Deux unités FMA par cœur
  - perf crête 4 FLOP binary64 par cycle
  - donc mieux que SSE2
- Pas de diviseur
- Parenthèse : unité flottante décimale dans le Power6
  - ils ne savent vraiment pas quoi faire de leurs transistors
  - Intel pense qu'il vaut mieux le laisser au soft
  - Devinez à qui l'histoire donnera raison ?

Les 8-9 SPU des PowerXCell sont aussi à base de FMA
(mais sont-ce des vrais ?)

# La famille IA64 (aka Itanium)

le beurre, l'argent du beurre, et une usine à gaz autour...

- Deux FMA binary82

  (le meilleur de IA32, et le meilleur de Power)

# La famille IA64 (aka Itanium)

le beurre, l'argent du beurre, et une usine à gaz autour...

- Deux FMA binary82
  (le meilleur de IA32, et le meilleur de Power)
- capables de binary64 et binary32 si on veut.
- 4 registres d'état flottant, choisis instruction par instruction
  - mélange d'arrondis et de précision sans surcoût
  - sur les autres architectures, un changement du registre d'état demande de vider le pipeline (10-100 cycles)

# La famille IA64 (aka Itanium)

le beurre, l'argent du beurre, et une usine à gaz autour...

- Deux FMA binary82
  (le meilleur de IA32, et le meilleur de Power)
- capables de binary64 et binary32 si on veut.
- 4 registres d'état flottant, choisis instruction par instruction
  - mélange d'arrondis et de précision sans surcoût
  - sur les autres architectures, un changement du registre d'état demande de vider le pipeline (10-100 cycles)
- Format de registre à 17 bits d'exposant

# La famille IA64 (aka Itanium)

le beurre, l'argent du beurre, et une usine à gaz autour...

- Deux FMA binary82
  (le meilleur de IA32, et le meilleur de Power)
- capables de binary64 et binary32 si on veut.
- 4 registres d'état flottant, choisis instruction par instruction
  - mélange d'arrondis et de précision sans surcoût
  - sur les autres architectures, un changement du registre d'état demande de vider le pipeline (10-100 cycles)
- Format de registre à 17 bits d'exposant
- Des instructions d'emballage/déballage/classification des flottants
  - Il restera très peu de touillage de bits à faire

# La famille IA64 (aka Itanium)

le beurre, l'argent du beurre, et une usine à gaz autour...

- Deux FMA binary82
  (le meilleur de IA32, et le meilleur de Power)
- capables de binary64 et binary32 si on veut.
- 4 registres d'état flottant, choisis instruction par instruction
  - mélange d'arrondis et de précision sans surcoût
  - sur les autres architectures, un changement du registre d'état demande de vider le pipeline (10-100 cycles)
- Format de registre à 17 bits d'exposant
- Des instructions d'emballage/déballage/classification des flottants
  - Il restera très peu de touillage de bits à faire
- Une floppée de registres

Facteur 5 à 10 en nombre de cycles par rapport au Pentium

# Ya pas que la fréquence dans la vie

Perf du log optimisé par Intel :

- Pentium 4 : 159 cycles
- Itanium-2 : dans les 30 cycles

Perf d'un log portable écrit en C et compilé par gcc :

- Pentium-4 : dans les 300 cycles
- AMD ou Pentium3/Core : dans les 150 cycles

# Portabilité contre performance

- Plus petit dénominateur commun matériel :
  - entiers 32 bits signés ou non
  - flottants simple et double précision
  - 4 opérations, fussent-elles en logiciel

# Portabilité contre performance

- Plus petit dénominateur commun matériel :
  - entiers 32 bits signés ou non
  - flottants simple et double précision
  - 4 opérations, fussent-elles en logiciel
- Les possibilités plus exotiques (FMA, doubles étendus, vecteurs) sont mal supportées par les langages/compilateurs
  - même quand elles sont omniprésentes depuis la nuit des temps (*add-with-carry*)
  - même quand elles sont faciles à émuler (entiers 64 bits)

# Portabilité contre performance

- Plus petit dénominateur commun matériel :
  - entiers 32 bits signés ou non
  - flottants simple et double précision
  - 4 opérations, fussent-elles en logiciel
- Les possibilités plus exotiques (FMA, doubles étendus, vecteurs) sont mal supportées par les langages/compilateurs
  - même quand elles sont omniprésentes depuis la nuit des temps (*add-with-carry*)
  - même quand elles sont faciles à émuler (entiers 64 bits)
- Cela a tendance à s'arranger... Mais en 2009,
  - Intel paye 20 personnes à plein temps à porter la `libm` d'un processeur Intel au processeur Intel suivant
  - du code efficace pour des fonctions élémentaires ressemble toujours à de l'assembleur

# Autres cibles technologiques

- FPGA
- GPU

# Quelques touillages de bits

## Implementation tricks
## Branch collapsing

**Branch collapsing via bit-mask**

```
if (x > b)
    y = y + a
```

$$
\begin{aligned}
m_a &= (b - x) >> 31 \\
a_1 &= m_a \;\&\; a \\
y &= y + a_1
\end{aligned}
$$

Transparent extrait de la présentation de Nikita Astafiev à RNC7 (2006)

`http://rnc7.loria.fr/`

## Implementation tricks
## Branch collapsing

**Branch collapsing via condition shrinking**

Let a > b

```
if (x > a)
        goto overflow
if (x < b)
        goto underflow
…
<main path>
…
```

```
if ((unsigned) (x-b) > a-b )
        goto special
…
<main path>
…
```

Transparent extrait de la présentation de Nikita Astafiev à RNC7 (2006)

`http://rnc7.loria.fr/`

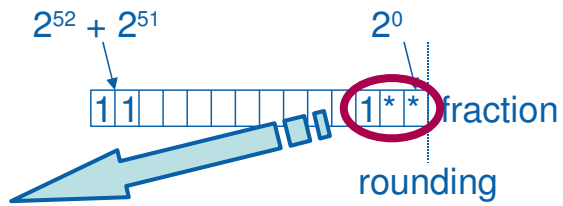
## Implementation tricks
## Fast conversions

**Fast conversion to integer via Right Shifter**

Let double $|x| < 2^{51}$

| | | | |
|---|---|---|---|
| cvtsd2si | y, x | addsd | x, $2^{52} + 2^{51}$ |
| cvtsi2sd | r, y | subsd | x, $2^{52} + 2^{51}$ |

Result is represented in double FP format




movd index, x

Il faut 2 bits à 1 pour que cela marche avec les nombres négatifs

Transparent extrait de la présentation de Nikita Astafiev à RNC7 (2006)

`http://rnc7.loria.fr/`

## Implementation tricks

**Fast Single to Double conversion**

Let single x > 0

$x = 2^{(E - 127)} \cdot \text{significand}$

```
cvtss2sd    y, x
```

```
psllq       x, 52 – 23
paddd       x, 0x3800000000000000
```

$y = 2^{(E - 1023)} \cdot \text{significand}$

**Fast integer ABS**

N = 32 or 64

```
S = x >> (N – 1)
```

$S = \begin{cases} 000..0, & \text{if } x >= 0 \\ 111..1, & \text{if } x < 0 \end{cases}$

```
y = (x + S) xor S
```

$y = |x|$

Transparent extrait de la présentation de Nikita Astafiev à RNC7 (2006)

`http://rnc7.loria.fr/`

# Polynômes

# Ce qu'il nous faut

1. Trouver un bon polynôme
2. L'évaluer vite
3. Obtenir un résultat précis
4. Borner l'erreur commise

# Trouver un bon polynôme

Refaisons de la pub pour Sollya...

# L'évaluer vite (1) : en SSE2

Cristina Anderson, Nikita Astafiev, Shane Story.
Accurate Math Functions on the Intel IA-32 Architecture : A Performance-Driven Design
In *Real Numbers and Computers*, 2006

## Exemple : un polynôme de degré 5

$$\begin{aligned} p(x) &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 \\ &= (a_0 + a_2 x^2 + a_4 x^4) \; + \; x(a_1 + a_3 x^2 + a_5 x^4) \end{aligned}$$

$$\text{Evaluation SIMD de } \left\{ \begin{array}{lllll} a_0 & + & x^2 \times (a_2 & + & a_4 \cdot x^2) \\ a_1 & + & x^2 \times (a_3 & + & a_5 \cdot x^2) \end{array} \right.$$

# L'évaluer vite (1) : en SSE2

Cristina Anderson, Nikita Astafiev, Shane Story.
Accurate Math Functions on the Intel IA-32 Architecture : A Performance-Driven Design
In *Real Numbers and Computers*, 2006

## Exemple : un polynôme de degré 5

$$\begin{aligned} p(x) &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 \\ &= (a_0 + a_2 x^2 + a_4 x^4) \; + \; x(a_1 + a_3 x^2 + a_5 x^4) \end{aligned}$$

$$\text{Evaluation SIMD de } \left\{ \begin{array}{lllll} a_0 & + & x^2 \times (a_2 & + & a_4 \cdot x^2) \\ a_1 & + & x^2 \times (a_3 & + & a_5 \cdot x^2) \end{array} \right.$$

Rem : Quand gcc sera-t-il capable de générer du code SIMD ?

# L'évaluer vite (1) : en SSE2

Cristina Anderson, Nikita Astafiev, Shane Story.
Accurate Math Functions on the Intel IA-32 Architecture : A Performance-Driven Design
In *Real Numbers and Computers*, 2006

## Exemple : un polynôme de degré 5

$$\begin{aligned} p(x) &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 \\ &= (a_0 + a_2 x^2 + a_4 x^4) \; + \; x(a_1 + a_3 x^2 + a_5 x^4) \end{aligned}$$

$$\text{Evaluation SIMD de } \left\{ \begin{array}{lllll} a_0 & + & x^2 \times (a_2 & + & a_4 \cdot x^2) \\ a_1 & + & x^2 \times (a_3 & + & a_5 \cdot x^2) \end{array} \right.$$

Rem : Quand gcc sera-t-il capable de générer du code SIMD ?
Le dernier slide d'Astafiev dit aussi :

*Future Development : Code in C*

# L'évaluer vite (2) : avec deux FMA

## Variations autour du schéma de Estrin

```
cycle  0     z2 = z*z;                p67 = c6 + z*c7;
cycle  1     p45 = c4 + z*c5;         p23 = c2 + z*c3;
cycle  2     p01 = logr + z*c1;
cycle  3
cycle  4     z4  = z2*z2;
cycle  5     p47 = p45 + z2*p67;
cycle  6     p03 = p01 + z2*p23;

cycle 10     p07 = p03 + z4*p47;

cycle 14     log = p07 + E*log2;
```

Harrison a écrit une moulinette qui optimise la latence

M. Cornea, J. Harrison, and P. T. P. Tang.
*Scientific Computing on Itanium®-based Systems*.
Intel Press, Hillsboro, OR, 2002.

# L'évaluer vite (3) : sur le ST200

Processeur VLIW 4 voies, 2 multiplieurs 32 bits

- implémentation de la division flottante par approx de $1/x$
  - deux fois plus rapide qu'en utilisant l'instruction de division du processeur
  - cf *should the division be incorporated into computer instruction sets ?*
- implémentation de la racine carrée flottante

Claude-Pierre Jeannerod, Hervé Knochel, Christophe Monat, Guillaume Revy, and Gilles Villard
A new binary floating-point division algorithm and its software implementation on the ST231 processor
To appear in *Arith*, 2009

# L'évaluer vite (4) : nombre d'opérations

- Les schémas précédents augmentent le nombre d'opération !
  - Pour une implémentation vectorielle / optimisée en débit, on se rapprochera de Horner.

# L'évaluer vite (4) : nombre d'opérations

- Les schémas précédents augmentent le nombre d'opération !
  - Pour une implémentation vectorielle / optimisée en débit, on se rapprochera de Horner.
- Il existe des préformage du polyôme qui diminuent le nombre d'opérations (cf Knuth)
  - mais ils changent les valeurs des coeffs
  - la qualité numérique peut s'en ressentir

# Obtenir un résultat précis

- Revenons à Horner :

$$p(x) = a_0 \ + \ x \times q(x)$$

- Si $x$ est petit devant $a_0$ (après réduction d'argument)
  - Exemple : exponentielle $e^x \approx 1 + x \cdot p(x)$
  - Le calcul en binary64 (53 bits de mantisse) de $q(x)$ donne une erreur p.ex. bornée par $2^{-50}$
  - La contribution de cette erreur à l'erreur finale est bornée par $2^{-50-k}$
  - Pour typiquement $k = 8$, l'erreur totale est dominée par l'erreur d'arrondi de l'addition finale : très bonne précision
- Sinon : arithmétique double-binary64

# Borner l'erreur commise

Et maintenant, une autre page de publicité :

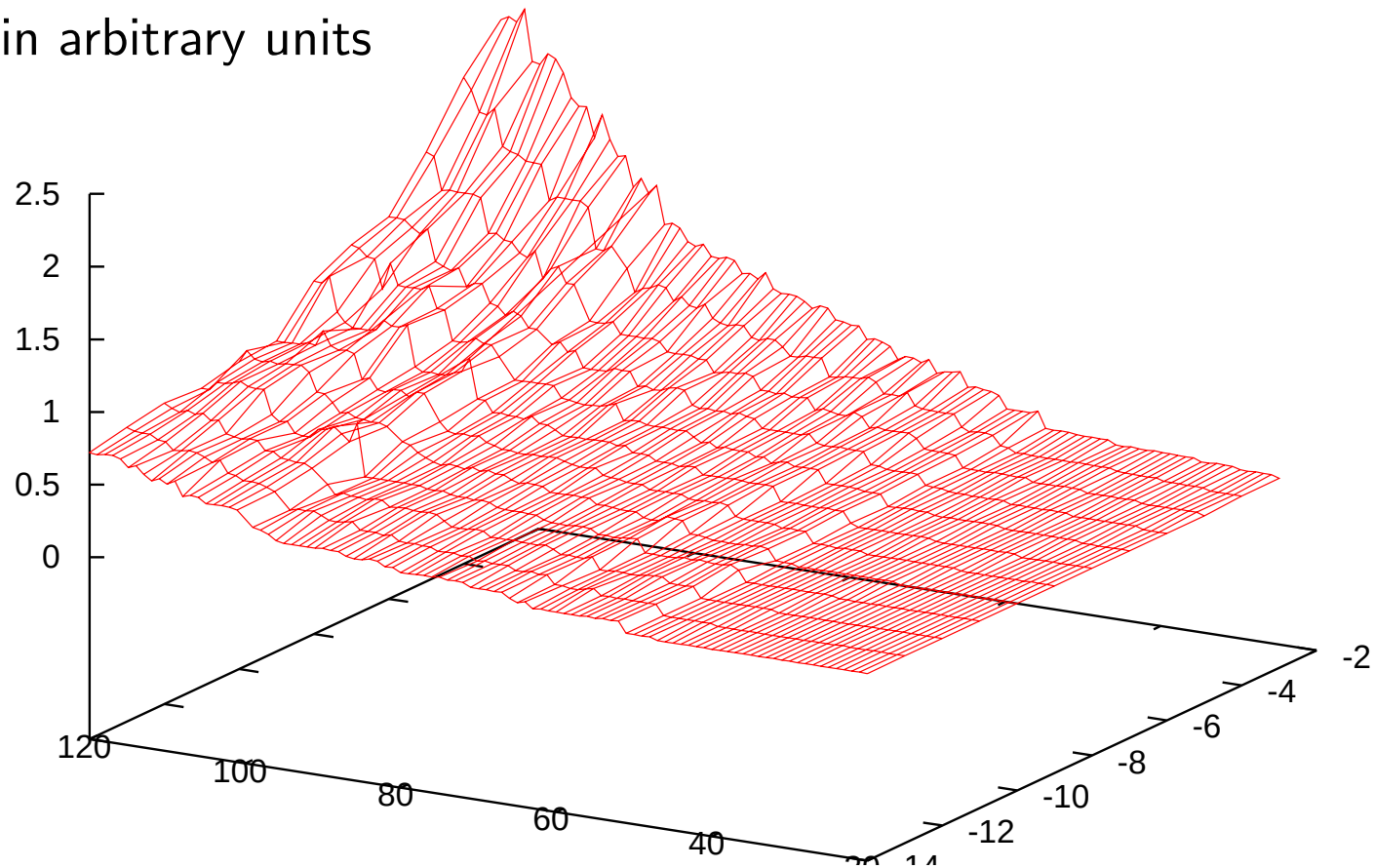
## Gappa

- Génération automatique de preuves de propriétés arithmétiques
- Thèse de Guillaume Melquiond chez Arénaire
- Quelques slogans (non validés par Guillaume)
  - La preuve formelle enfin à la portée des masses ignorantes
  - L'outil qui est à Coq ce que C est à l'assembleur

`http://lipforge.ens-lyon.fr/www/gappa/`

# Conclusion : script and conquer

- $\log(1+x)$
- Reduced $x$ in interval of size $2^{-13}$ to $2^{-1}$
- Accuracy between 20 and 120 bits
- 1203 implementations, all formally checked in Gappa
- Timings in arbitrary units



# Bonus 1 : la vraie vie

# From clean math to messy code, 1 : the math

The math for `crlibm` sine :

1. Range reduction
   - define $(k, y)$ such that $x = k\frac{\pi}{256} + y$ with $|y| \leqslant \frac{\pi}{512}$
   - Let $k' = k \mod 128$ and $a = k'\frac{\pi}{256}$ (quadrant symmetries and periodicity obtained by integer arithmetic on $k$)
2. $\sin(x)$ and $\cos(x)$ will be one of

$$\begin{aligned} \sin(a+y) &= \sin(a)\cos(y) + \cos(a)\sin(y) \\ \cos(a+y) &= \cos(a)\cos(y) - \sin(a)\sin(y) \end{aligned}$$

3. Read $\sin(a)$ and $\cos(a)$ from a table indexed by $k'$
4. Compute $\sin(y)$ and $\cos(y)$ as small polynomials

# From clean math to messy code, 2 : the mess

The implementation :

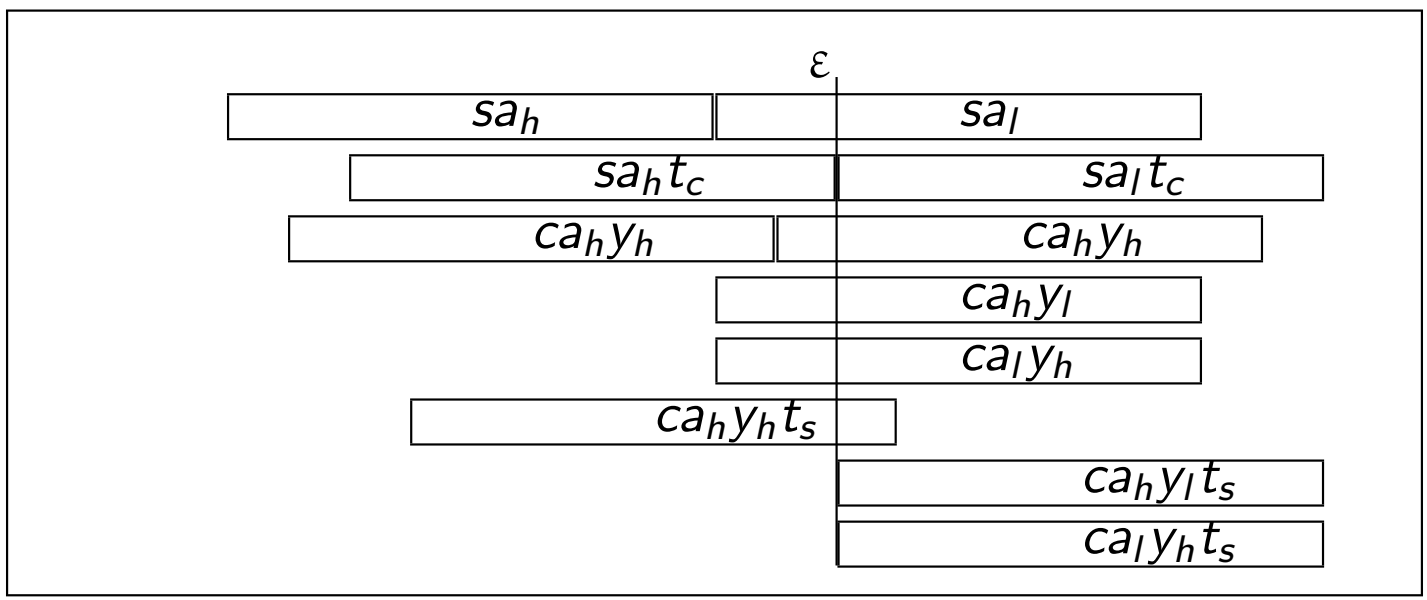
1. Range reduction : compute $(k, y)$ such that $x \approx k\frac{\pi}{256} + y$
   - $y$ computed as a double-double $y_h + y_l$,
   - $|y_h + y_l| \leqslant \frac{\pi}{512} + \varepsilon$
   - Let $a = (k \mod 128)\frac{\pi}{256}$
2. Read the double-doubles $sa_h + sa_l \approx \sin(a)$ and $ca_h + ca_l \approx \cos(a)$ from a table (2KB)
3. Compute $\sin(y)$ and $\cos(y)$ as small polynomials :
   - $\cos(y) \approx 1 + tc$
   - $\sin(y) \approx y(1 + ts)$
   - $t_s = P_s(y_h^2)$ and $tc = P_c(y_h^2)$ evaluated in double precision
4. Finally

$$\begin{aligned} \sin(a+y) &= \sin(a)\cos(y) + \cos(a)\sin(y) \\ &\approx (sa_h + sa_l)(1 + tc) + (ca_h + ca_l)(y_h + y_l)(1 + t_s) \end{aligned}$$

5. Develop and sum up cleverly

# Develop and sum up cleverly

$$(sa_h + sa_l)(1 + tc) \;+\; (ca_h + ca_l)(y_h + y_l)(1 + t_s)$$



# From clean math to messy code, 3 : the code

```
1  yh2 = yh*yh ;
2  ts = yh2 * (s3 + yh2*(s5 + yh2*s7));
3  tc = yh2 * (c2 + yh2*(c4 + yh2*c6 ));
4  Mul12(&cahyh_h,&cahyh_l, cah, yh);
5  Add12(thi, tlo, sah,cahyh_h);
6  tlo = tc*sah+(ts*cahyh_h+(sal+(tlo+(cahyh_l+(cal*yh +
       cah*yl)))));
7  Add12(*psh,*psl,  thi, tlo);
```

# A bestiary of errors

- Ideal reduced argument is $y$ approximated to $y_h + yl$

# A bestiary of errors

- Ideal reduced argument is $y$ approximated to $y_h + yl$
- then, $y_l$ is dropped for most of the computation

# A bestiary of errors

- Ideal reduced argument is $y$ approximated to $y_h + yl$
- then, $y_l$ is dropped for most of the computation
- then, rounding error in computing `yh2 = yh * yh`

# A bestiary of errors

- Ideal reduced argument is $y$ approximated to $y_h + yl$
- then, $y_l$ is dropped for most of the computation
- then, rounding error in computing `yh2 = yh * yh`
- then, polynomials of `yh2` (many rounding errors)

# A bestiary of errors

- Ideal reduced argument is $y$ approximated to $y_h + yl$
- then, $y_l$ is dropped for most of the computation
- then, rounding error in computing `yh2 = yh * yh`
- then, polynomials of `yh2` (many rounding errors)
- (these polynomials were themselves approximations)

# A bestiary of errors

- Ideal reduced argument is $y$ approximated to $y_h + yl$
- then, $y_l$ is dropped for most of the computation
- then, rounding error in computing `yh2 = yh * yh`
- then, polynomials of `yh2` (many rounding errors)
- (these polynomials were themselves approximations)
- then some terms are neglected in the final sum

# A bestiary of errors

- Ideal reduced argument is $y$ approximated to $y_h + yl$
- then, $y_l$ is dropped for most of the computation
- then, rounding error in computing `yh2 = yh * yh`
- then, polynomials of `yh2` (many rounding errors)
- (these polynomials were themselves approximations)
- then some terms are neglected in the final sum
- and again many rounding errors in this sum

# A bestiary of errors

- Ideal reduced argument is $y$ approximated to $y_h + yl$
- then, $y_l$ is dropped for most of the computation
- then, rounding error in computing `yh2 = yh * yh`
- then, polynomials of `yh2` (many rounding errors)
- (these polynomials were themselves approximations)
- then some terms are neglected in the final sum
- and again many rounding errors in this sum

What is the error of what with regard to what ?

# Information missing from the code

All this works because we have a lot of implicit knowledge :

- The previous magnitude graph
  - Is it reliable ? The code works only if it is accurate

# Information missing from the code

All this works because we have a lot of implicit knowledge :

- The previous magnitude graph
  - Is it reliable ? The code works only if it is accurate
- Mathematical identities like

$$\sin(a+y) = \sin(a)\cos(y) + \cos(a)\sin(y)$$

or

$$\sin(a)^2 + cos(a)^2 = 1$$

# Information missing from the code

All this works because we have a lot of implicit knowledge :

- The previous magnitude graph
  - Is it reliable ? The code works only if it is accurate
- Mathematical identities like

$$\sin(a+y) = \sin(a)\cos(y) + \cos(a)\sin(y)$$

or

$$\sin(a)^2 + cos(a)^2 = 1$$

- Implicit knowledge related to double-double arithmetic

# Information missing from the code

All this works because we have a lot of implicit knowledge :

- The previous magnitude graph
  - Is it reliable ? The code works only if it is accurate
- Mathematical identities like

$$\sin(a+y) = \sin(a)\cos(y) + \cos(a)\sin(y)$$

or

$$\sin(a)^2 + cos(a)^2 = 1$$

- Implicit knowledge related to double-double arithmetic

No automatic tool tool will invent this knowledge.

# Information missing from the code

All this works because we have a lot of implicit knowledge :

- The previous magnitude graph
  - Is it reliable ? The code works only if it is accurate
- Mathematical identities like

$$\sin(a+y) = \sin(a)\cos(y) + \cos(a)\sin(y)$$

or

$$\sin(a)^2 + cos(a)^2 = 1$$

- Implicit knowledge related to double-double arithmetic

No automatic tool tool will invent this knowledge.

Proof assistants require you to make it explicit.

# Bonus 2 : des fonctions élémentaires pour FPGA

# C'est tout pareil

- $X = 2^E \cdot 1, F$
- Donc $\log(X) = E \cdot \log(2) + \log(1, F)$
- Pour recentrer la mantisse sur 1 et éviter un problème de précision autour de $X = 2$, on préfère

$$\begin{cases} E' = E & \text{et } Y = 1, F \quad \text{si} \quad m \leqslant \sqrt{2} \\ E' = E + 1 & \text{et } Y = \frac{1,F}{2} \quad \text{if} \quad m > \sqrt{2} \end{cases}$$

- alors $\log(X) = E' \cdot \log(2) + \log(Y)$
  - avec $\frac{\sqrt{2}}{2} \leqslant Y \leqslant \sqrt{2}$
  - (choisi pour que $\log(Y)$ soit centré autour de 0)

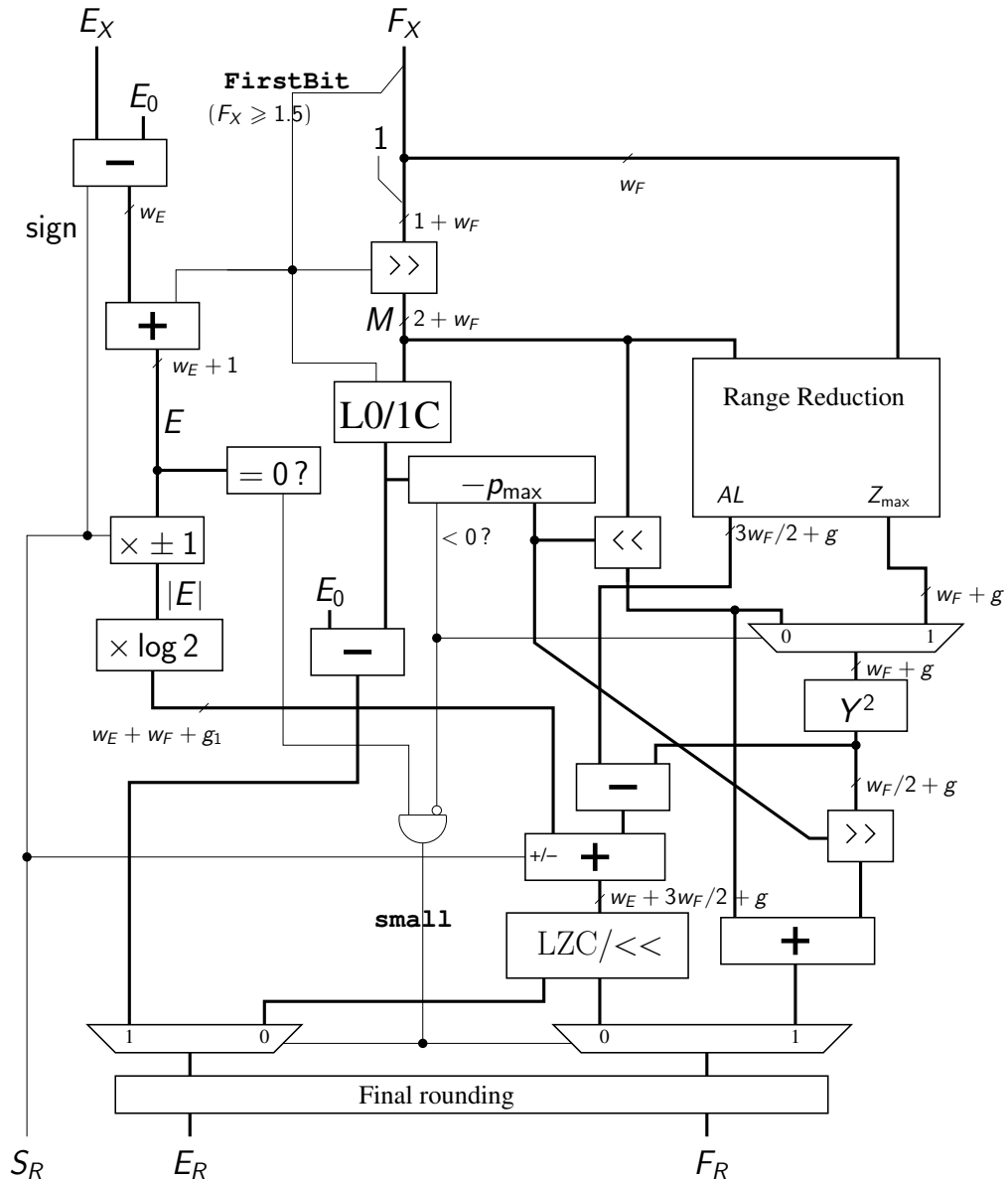
# C'est tout pareil

- $X = 2^E \cdot 1, F$
- Donc $\log(X) = E \cdot \log(2) + \log(1, F)$
- Pour recentrer la mantisse sur 1 et éviter un problème de précision autour de $X = 2$, on préfère

$$\begin{cases} E' = E & \text{et } Y = 1, F \quad \text{si} \quad m \leqslant 1.5 \\ E' = E + 1 & \text{et } Y = \frac{1,F}{2} \quad \text{if} \quad m > 1.5 \end{cases}$$

- alors $\log(X) = E' \cdot \log(2) + \log(Y)$
  - avec $0.75 \leqslant Y \leqslant 1.5$
  - (choisi pour que le test se ramène à un bit)

# Overview of the logarithm



# Evaluation de $\log(Y)$

- méthodes générales à base de tables (12-24 bits)
  - approximation de n'importe quelle fonction gentille
  - approximation polynomiale par morceaux, degré 1 à 4
  - évaluation parallèle
  - précision au plus juste
  - tables et additions, et parfois petits multiplieurs

# Evaluation de $\log(Y)$

- méthodes générales à base de tables (12-24 bits)
  - approximation de n'importe quelle fonction gentille
  - approximation polynomiale par morceaux, degré 1 à 4
  - évaluation parallèle
  - précision au plus juste
  - tables et additions, et parfois petits multiplieurs
- récurrences générales à la CORDIC
  - à adapter au contexte

# A universal iteration for exp and log

Let $(x_i)$ and $(l_i)$ be two given sequences of reals such that

$$\forall i, e^{l_i} = x_i$$

We will define two new sequences $(x_i')$ and $(l_i')$ as follows :
$l_0'$ and $x_0'$ are such that $x_0' = e^{l_0'}$, and

$$\forall i > 0 \begin{cases} l_{i+1}' &= l_i + l_i' \\ x_{i+1}' &= x_i \times x_i' \end{cases} \quad (1)$$

This iteration maintains the invariant

$$x_i' = e^{l_i'}$$

(since $x_0' = e^{l_0'}$ and $x_{i+1} = x_i x_i' = e^{l_i} e^{l_i'} = e^{l_i + l_i'} = e^{l_{i+1}'}$).

# A universal iteration for exp and log

$$\forall i \quad e^{l_i} = x_i$$

$$\forall i \quad x'_i = e^{l'_i}$$

$$\forall i > 0 \begin{cases} l'_{i+1} & = & l_i + l'_i \\ x'_{i+1} & = & x_i \times x'_i \end{cases}$$

Common idea :

- Compute this double iteration
- $(x_i)$ and $(l_i)$ are read from tables
- Keep the multiplications cheap by tabulating small $x_i$ (sometimes mere digits)
- The corresponding $l_i$ may be large because addition is cheaper

# A universal iteration for the exp

$$\forall i > 0 \begin{cases} l'_{i+1} &= l_i + l'_i \\ x'_{i+1} &= x_i \times x'_i \end{cases}$$

If $l$ is given and we want to compute $e^l$

- Start with $(l'_0, x'_0) = (0, 1)$
- Chose $(l_i, x_i)$ such that the corresponding sequence $(l'_i, x'_i)$ converges to $(l, x = e^l)$.

# A universal iteration for the log

$$\forall i > 0 \begin{cases} l'_{i+1} &= l_i + l'_i \\ x'_{i+1} &= x_i \times x'_i \end{cases}$$

If $x$ is given and we want to compute $\log(x)$

- $x'_0 = x$
- Chose the $x_i$ such that $(x'_i)$ converges to 1 for increasing $i$.
- For some $i = N$, $x'_N$ is so close to 1 that $l'_N = \log(x'_N) \approx x'_N - 1$
- This $l'_N$ bootstraps the computation of the other $l'i$ for decreasing $i$
- ... until $\log(x) = l'_0$

# Iterative range reduction

- Main argument reduction :

$$\log(X) = \log(Y_0) + E \cdot \log 2 \quad \text{with} \quad Y_0 \in [0.75, 1.5).$$

- First iteration : Read from a table indexed by 5 MSB bits of $Y_0$
  - an approximation $\widetilde{Y_0^{-1}}$ to $Y_0$
  - an accurate value $L_0$ of $\log(\widetilde{Y_0^{-1}})$

$$\log(Y_0) = \log(Y_0 \times \widetilde{Y_0^{-1}}) - \log(\widetilde{Y_0^{-1}}) = \log(1 + Z_1) - L_0$$

  We may then show that (for a well constructed table)

$$Y_1 = Y_0 \times \widetilde{Y_0^{-1}} = 1 + Z_1 \quad \text{with} \quad 0 \leqslant Z_1 < 2^{-4}$$

- Following iterations : proceed for $\log(1 + Z_1)$, with one difference : no need for an inverse table, the approximation to the inverse is given by

$$\widetilde{Y_1^{-1}} \approx 1 - Z_1$$

# Single precision

```
   Y0 : 1.00110011001100110011001100110
   Z1 :       00110011001100110011000010
   Z2 :          01010101100110011000010000 1101
   Z3 :             01101011010010101110111 0110
   Z4 :                100110100000110110110010
 Z4Sq :                                0101110010
LogY4 :                100110100000110001000000
   L0 :  .00101011011111101000000011010110101 0101
   L1 :       00101000001100100101001111110010 0101
   L2 :          01001000000101000100011110011 0110
   L3 :             01011000000000111100100000 1001
LogY0 :  .00101110101011001001111111110010000 1001
```

# Résultats

Area (in Virtex-II slices) and delay (in ns)

| Exponential | | | | |
|---|---|---|---|---|
| Format | This work | | Table-based | |
| $(w_E, w_F)$ | Area | Delay | Area | Delay |
| (7, 16) | **472** | **118** | 480 | 69 |
| (8, 23) | **728** | **123** | 948 | 85 |
| (9, 38) | **1242** | **175** | – | – |
| (11, 52) | **2045** | **229** | – | – |

| Logarithm | | | | |
|---|---|---|---|---|
| Format | This work | | Table-based | |
| $(w_E, w_F)$ | Area | Delay | Area | Delay |
| (7, 16) | **556** | **70** | 627 | 56 |
| (8, 23) | **881** | **88** | 1368 | 69 |
| (9, 38) | **1893** | **149** | – | – |
| (11, 52) | **3146** | **182** | – | – |

# Perspectives

- Use embedded multipliers $\longrightarrow$ polynomials
- Mixed approach