Fonctions élémentaires : algorithmes et implémentations efficaces pour l'arrondi correct en double précision

David Defour

THÈSE

présentée devant

L'ÉCOLE NORMALE SUPÉRIEURE DE LYON

pour obtenir le grade de

Docteur de l'École Normale Supérieure de Lyon
spécialité : Informatique
au titre de l'école doctorale MathIf

par **David DEFOUR**

FONCTIONS ÉLÉMENTAIRES : ALGORITHMES ET IMPLÉMENTATIONS EFFICACES POUR L'ARRONDI CORRECT EN DOUBLE PRÉCISION.

Présentée et soutenue publiquement le 9 septembre 2003

Après avis de : Monsieur Jean DELLA DORA
Monsieur Daniel LITAIZE

Devant la commission d'examen formée de :

Monsieur Florent de DINECHIN, *Membre/Directeur de thèse*
Monsieur Jean DELLA DORA, *Membre/Rapporteur*
Monsieur Daniel LITAIZE, *Membre/Rapporteur*
Monsieur Peter MARKSTEIN, *Membre*
Monsieur Jean-Michel MULLER, *Membre/Directeur de thèse*
Monsieur Paul ZIMMERMANN, *Membre*

Thèse préparée à l'École Normale Supérieure de Lyon
au sein du Laboratoire de l'Informatique du Parallélisme.

Table des matières

Remerciements

Je remercie Florent de Dinechin mon directeur de thèse, qui a toujours consacré le temps nécessaire pour m'encadrer, me diriger et me conseiller pendant ces trois années de thèse. Je n'oublie pas non plus, mon «deuxième» directeur de thèse Jean-Michel Muller, qui a parfaitement su jongler entre ses nombreuses charges administratives et mon encadrement de thèse. j'ai ainsi pu bénéficier d'un parfait duo d'encadrement, dont la complémentarité scientifique a permis de baliser l'avancée de ce travail jusqu'à son aboutissement.

Je tiens à remercier M.Della Dora et M. Litaize d'avoir accepté la tâche de rapporter cette thèse malgré les délais très courts que je leur ai imposés et de m'avoir permis, par leurs commentaires avisés, d'améliorer la qualité de ce document.

Je remercie M. Markstein et M. Zimmermann de m'avoir fait l'honneur d'être membre de mon jury. J'apprécie la qualité des nombreuses remarques et réflexions qu'ils ont pu faire sur ce travail.

J'ai une pensée particulière pour tous les membres et anciens membres d'Arenaire. Je remercie tout particulièrement Arnaud Tisserand avec qui j'ai passé de bons moments et dont j'admire les nombreuses qualités scientifiques et humaines. Je remercie également Marc Daumas pour m'avoir impliqué dans de nombreux événements ainsi que pour tous ses conseils, Nathalie pour avoir accepté de co-encadrer un stage au cours de ma troisième année de thèse, Pascal pour tous les petits services rendus ainsi que Brice pour m'avoir supporté dans ce bureau pendant cette thèse, Nicolas et Sylvie pour respectivement leur chaleureuse aide technique et scientifique, NW pour la bonne humeur permanente qu'il a su apporter dans l'équipe, Gilles et Claude-Pierre pour tous les petits moments de détente passés ensemble et enfin Catherine pour le travail dont elle a su me décharger sur le projet crlibm. Ce paragraphe est également l'occasion de réitérer l'admiration pour mes deux directeurs de thèse Florent et Jean-Michel.

Je remercie Samir et Stef d'avoir assisté à ma soutenance et Jean-Luc pour tout le chocolat suisse ainsi que la relecture de ce document.

Je ne peux clore cette partie sans remercier ma famille et plus particulièrement mes parents ainsi que ma future épouse Shereen, pour m'avoir soutenu et encouragé avec abnégation.

Exorde

L'introduction de la virgule flottante au sein des processeurs s'est accompagnée de certaines difficultés. Le format de représentation utilisé, la précision et le comportement des opérateurs étaient, en l'absence de consensus entre constructeurs, différents d'un processeur à un autre. Les concepteurs d'algorithmes numériques devaient alors, en fonction des caractéristiques de l'arithmétique en virgule flottante de chaque processeur, créer autant d'algorithmes et d'implémentations que de plates-formes ciblées.

La portabilité a été un réel problème jusqu'à l'adoption en 1985 de la norme IEEE-754 [47] spécifiant le format de représentation des nombres en virgule flottante ainsi que le comportement des opérateurs arithmétiques. Cette norme requiert, entre autre, ce que l'on appelle la propriété « d'arrondi correct » pour les opérations flottantes ($+$, $\times$, $/$, $\sqrt{}$). Cette propriété impose aux opérateurs en virgule flottante de rendre comme résultat l'arrondi du résultat exact suivant un mode choisi à l'avance (arrondi au plus près, vers 0,vers $+\infty$, vers $-\infty$). Cette norme est aujourd'hui respectée par tous les processeurs généralistes. La portabilité et la construction de preuves d'algorithmes numériques basés sur le strict respect de cette norme sont désormais possibles. En particulier les trois modes d'arrondi dirigés, vers 0, $+\infty$, $-\infty$, ont contribué au développement de l'arithmétique par intervalle.

Toutefois, cette norme a des limites. En effet, il est fréquent que les algorithmes numériques utilisent des opérations plus complexes que la simple addition ou la multiplication. Ils peuvent, par exemple, utiliser des fonctions élémentaires tels que sinus, cosinus, exponentielle, etc. Toute possibilité de construire des preuves de ces algorithmes, ou d'en estimer l'erreur, est alors anéantie. Ceci est dû à l'absence de spécification du comportement de ces fonctions par la norme IEEE-754.

La principale raison pour laquelle ces fonctions ne font toujours pas partie de la norme tient dans la définition même de ces fonctions : elles sont transcendantes. En général, le résultat d'une de ces fonctions pour une entrée exactement représentable ne peut être calculé exactement. Il est par conséquent difficile de déterminer la précision avec laquelle il faut effectuer les calculs pour garantir l'arrondi correct de ces fonctions. Ce problème porte le nom du « dilemme du fabricant de table ». Toutefois une solution à ce problème fut proposée en 1991 par Ziv [83]. Il a proposé une méthode où l'on augmente la précision du résultat jusqu'à être capable de fournir l'arrondi correct pour la précision désirée. Ce procédé porte le nom de «peau d'oignon» et il est prouvé qu'il s'arrête.

Cette stratégie est celle utilisée par les évaluations de fonctions élémentaires proposées au sein de la bibliothèque multiprécision MPFR [3]. Cependant, le surcoût lié aux opérateurs multiprécision est trop important pour en généraliser leurs utilisations lorsque l'on cible la double précision.

Pour la double précision, il existe la bibliothèque *MathLib* [2], développée par IBM. Cependant cette biliothèque ne fournit que l'arrondi au plus près, n'a pas de borne sur le temps

maximum d'évaluation et n'inclut pas de preuve de l'arrondi correct. De plus elle est très volumineuse en nombre de ligne de code et en ressource mémoire.

Pour borner le temps d'évaluation avec une stratégie en peau d'oignon, nous devons disposer pour un format de représentation donné et pour chaque fonction, d'une borne sur la précision maximum pour être capable de toujours rendre l'arrondi correct. Le travail de thèse de Lefèvre [59] fut de déterminer cette précision par une recherche exhaustive pour le format double précision. Les bornes trouvées montrent que les résultats intermédiaires doivent être calculés avec, dans le pire cas et selon le mode d'arrondi et la fonction, 100 à 150 bits de précision, pour garantir l'arrondi correct.

Le travail de cette thèse fut d'exploiter les résultats de Lefèvre sur les pires cas du dilemme du fabricant de table pour construire des algorithmes efficaces, portables, implémentés, et prouvés d'évaluation de fonctions élémentaires en double précision avec arrondi correct selon les quatre modes d'arrondi. Pour celà, nous avons choisi une évaluation en deux étapes : une étape rapide, basée sur les propriétés du format double précision de la norme IEEE-754, calculant juste la plupart du temps et une étape précise, basée sur des opérateurs multiprécision, retournant un résultat juste tout le temps.

Organisation

Le premier chapitre de cette thèse présentera la norme IEEE-754 sur la virgule flottante, les propriétés, définitions et limites liées à son utilisation. Nous exposons la raison pour laquelle les fonctions élémentaires n'ont pas été incluses dans cette norme à travers le dilemme du fabricant de table. Enfin, nous achevons ce chapitre par une liste de propositions que nous avons émises sur ce que devrait contenir une norme pour les fonctions élémentaires.

Le chapitre 2 est consacré aux méthodes utilisées pour évaluer les fonctions élémentaires. Après une brève présentation des méthodes d'évaluation matérielle à base de table et de nos contributions dans ce domaine, nous abordons les étapes qui composent l'évaluation logicielle. Ce sera l'occasion de présenter nos travaux sur la réduction d'argument en grande base.

Au cours du chapitre 3, nous aborderons les optimisations appliquées à la première phase de l'évaluation des fonctions élémentaires. Nous introduisons certains éléments relatifs aux processeurs en 2003, pour en cerner leurs caractéristiques et les exploiter de façon efficace. Nous présentons également les algorithmes permettant d'effectuer des additions et multiplications sans erreur d'arrondi, que nous complétons par ceux utilisés pour tester si l'arrondi correct est possible. Enfin, nous décrivons les résultats d'expériences pour mesurer l'impact que peut avoir la hiérarchie mémoire sur les performances des schémas d'évaluation des fonctions élémentaires.

Le chapitre 4 est consacré à la seconde phase de l'évaluation, et plus précisément à notre travail sur la bibliothèque multiprécision *Software Carry Save* (SCS). On y présente un état de l'art en matière de multiprécision, en classifiant les différents outils multiprécision selon le format de représentation utilisé en interne. On décrit les motivations et les choix faits dans la bibliothèque SCS.

Le chapitre 5 constitue la mise en pratique de tous les éléments rencontrés au cours des chapitres précédents. Nous décrivons, en utilisant comme exemple la fonction exponentielle, comment la bibliothèque *crlibm* en cours de développement fournit l'arrondi correct pour la double précision et les quatre modes d'arrondi.

CHAPITRE

1

- rendre possible la gestion des exceptions (racine carrée d'un nombre négatif),
- unicité de la représentation (à l'exception du zéro signé), pour faciliter les comparaisons,
- permettre un support pour l'arithmétique par intervalle avec les 2 modes d'arrondis dirigés ($\pm\infty$) plus une compatibilité pour le mode d'arrondi par troncature.

1.1.1.1 Représentations des nombres IEEE et précision

Les nombres du format IEEE-754 sont caractérisés par trois champs, une mantisse m, un exposant e et un signe s, la base de représentation étant fixée à 2. Ainsi, un nombre flottant x s'écrit de la façon suivante :

$$x = (-1)^s.2^e.m$$

où :

- L'**exposant** e est basé sur une représentation biaisée pour des raisons d'efficacité de l'implémentation matérielle. Cela signifie que la représentation de l'exposant e correspond à la valeur e plus un biais entier. Ce biais est égal à 127 pour la simple précision et à 1023 pour la double précision. Par exemple, l'exposant 0 est représenté par la valeur du biais. Cette représentation permet dans le cas de flottants positifs, de traiter les nombres flottants comme des entiers pour les comparaisons. La valeur de l'exposant évolue entre e_{min} et e_{max}.
- La **mantisse** m est un nombre en virgule fixe composé de n bits. Elle est comprise entre 0 et 2. Le nombre n de bits qui composent la mantisse est de 24 pour la simple précision et de 53 pour la double. Pour des raisons d'unicité de représentation des nombres flottants, la position de la virgule de la mantisse se situe entre le premier et le deuxième bit de la mantisse ($m = m_0, m_1 m_2 \dots m_{n-1}$), où :
  - Le **bit implicite** m_0 est le seul bit de la partie entière de la mantisse. Il n'est pas représenté pour les formats simple et double précision, car il est entièrement déterminé par la valeur de l'exposant. En revanche, il n'y a pas de bit implicite pour les formats étendus.
  - La **fraction (**$f = m_1 m_2 \dots m_n$**)** est la représentation interne de la mantisse.

La norme impose au minimum 2 formats : la simple précision codée sur 32 bits et la double précision codée sur 64 bits. Elle définit également une précision étendue pour ces deux formats. Le nombre de bits utilisés pour la simple et la double précision étendue est élargi, avec pour chacun un nombre de bit minimum pour les champs exposant et mantisse, le choix exact du nombre de bits utilisés étant laissé aux constructeurs. Le tableau 1.1 résume la longueur des différents champs pour chacun de ces formats.

Un exemple de la façon dont est traduite la représentation hexadécimal d'un nombre flottant double précision (ici `c0190000 00000000`) vers la valeur flottante qui lui correspond est donnée dans la figure 1.1.

1.1.1.2 Nombres IEEE et valeurs spéciales

Pour traiter des problèmes tels que des dépassements de capacité ou des exceptions, la norme impose certaines représentations comme les infinis ou les *NaN* (Not a Number). La façon de représenter ces quantités est donnée dans le tableau 1.2

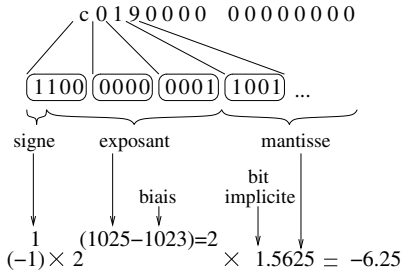


FIG. 1.1: *Conversion de la représentation hexadécimal d'un nombre double précision vers sa valeur flottante.*

précision	simple	simple étendue	double	double étendue
exposant max.	$+127$	$+1023$	$+1023$	$+16383$
exposant min.	-126	-1022	-1022	-16382
exposant biais	$+127$	$+1023$	$+1023$	$+16383$
signe (# bits)	1	1	1	1
exposant (# bits)	8	≥ 11	11	≥ 15
mantisse (# bits)	23	≥ 32	52	≥ 64
bit implicite	oui	non	oui	non
taille (# bits)	32	≥ 43	64	≥ 80

TAB. 1.1: *Taille des nombres*

Les normalisés. Les nombres normalisés sont les nombres dont le bit implicite de la mantisse vaut 1. Donc leur mantisse m satisfait la relation $1 \leq m < 2$ et leur exposant e est tel que $e_{min} \leq e \leq e_{max}$.

Les dénormalisés. Les nombres dénormalisés sont les nombres dont le bit implicite de la mantisse vaut 0. Ils ont pour but d'uniformiser la répartition des nombres représentables à proximité de la valeur 0 (Figure 1.2). Donc leur mantisse m satisfait la relation $0 \leq m < 1$ et leur exposant e est tel que $e = e_{min} - 1$.

Nom	Exposant	Fraction	Valeurs
Les normalisés	$e_{min} \leq e \leq e_{max}$	–	$1.f \times 2^e$
Les dénormalisés	$e = e_{min} - 1$	$f \neq 0$	$0.f \times 2^{e_{min}}$
Le Zéro signé	$e = e_{min} - 1$	$f = 0$	± 0
Not a Number	$e = e_{max} + 1$	$f \neq 0$	*NaN*
L'infini	$e = e_{max} + 1$	$f = 0$	$\pm\infty$

TAB. 1.2: *Représentation des valeurs spéciales de la norme IEEE 754*

Le Zéro signé. Zéro est représenté en mettant l'exposant et la mantisse à 0. Comme dans le cas de l'infini, Zéro est une quantité signée. Pour éviter les comportements imprévisibles de tests tel que «`if(x==0)`», la norme impose que le test de l'égalité $-0 = +0$ soit vrai.

Not a Number. Ce code sert à représenter le résultat d'une opération invalide comme $0/0$, $\sqrt{-1}$, $0 \times \infty$. Il permet de ne pas avoir à interrompre un calcul lors de telles opérations et de poursuivre le calcul. Il est représenté par *NaN*. Il existe deux types de *NaNs* : les *signaling NaN*(sNaN) et les *quiet NaN* (qNaN), chacun ayant plusieurs représentations machine (voir le tableau 1.3). De plus la norme impose que le test «`if(NaN==y)`» soit faux sauf si `y` est un *NaN*.

L'infini. Comme pour le NaN, l'infini est un code. Il permet de ne pas s'arrêter lorsqu'un dépassement de capacité ou une opération telle que $1/0$ sont rencontrés. Nous noterons que l'infini est une quantité signée, et qu'il existe des infinis 'exacts', comme $1/0$.

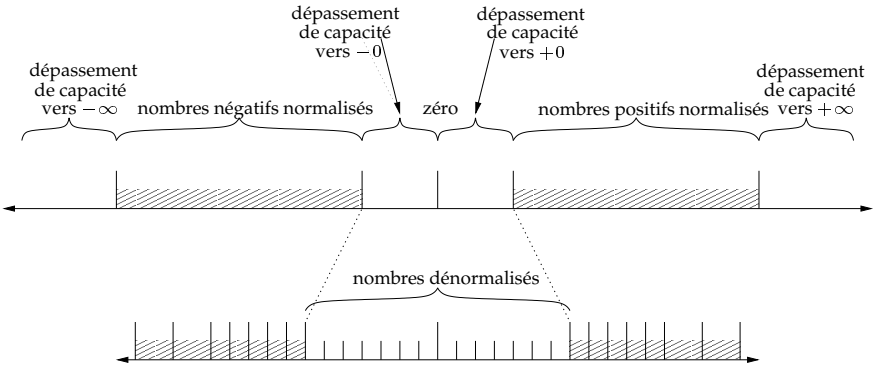


FIG. 1.2: *Ensemble des nombres représentable IEEE*

1.1.1.3 Les exceptions

Les exceptions constituent un élément important de la norme pour informer le système sur le comportement d'une opération et de son résultat.

La norme prévoit 5 drapeaux différents : dépassement de capacité vers l'infini, dépassement de capacité vers 0, division par 0, résultat inexact et opération invalide. Ces drapeaux sont dits *sticky* (collants) ce qui signifie qu'une fois les drapeaux levés, ils le restent jusqu'à ce qu'ils soient explicitement supprimés. De plus, l'implémentation doit pouvoir permettre à l'utilisateur de lire ou d'écrire ces drapeaux.

Dépassement de capacité vers 0. Cette exception parfois appelée *underflow* peut être produite de deux façons : soit lorsqu'une opération produit comme résultat un nombre dénormalisé, soit lorsque le résultat d'une opération est trop petit pour être représentable par un dénormalisé.

Le résultat d'une opération produisant un drapeau «dépassement de capacité vers 0» est soit zéro soit un dénormalisé.

Dépassement de capacité vers ∞. Cette exception aussi appelée *overflow* est le résultat d'une opération produisant un nombre trop grand pour être représentable.

Le résultat d'une opération produisant un drapeau «dépassement de capacité vers ∞», est $\pm\infty$.

Division par zéro. Cette exception est levée lorsque le diviseur est égal à zéro et que le dividende est un nombre différent de zéro. Ce drapeau est principalement utilisé pour faire la différence sur la source d'un infini.

Le résultat d'une opération produisant un drapeau «division par zéro» est $\pm\infty$.

Résultat inexact. Cette exception est levée lorsque le résultat d'une opération n'est pas exact car non représentable dans le format et que le résultat est arrondi. Il est également utilisé lorsqu'un résultat produisant un dépassement de capacité vers ∞ est produit mais que le drapeau correspondant n'est pas levé.

Le résultat d'une opération produisant un drapeau «résultat inexact» est le nombre arrondi ou la valeur ∞.

Opération invalide. Cette exception est levée lorsqu'une opération génère un *NaN*. Cependant, si le résultat est un *NaN* généré car l'une de ses entrées est un *NaN*, alors l'exception n'est pas levée.

Le résultat d'une opération produisant un drapeau «opération invalide» est une représentation de la valeur *NaN*.

1.1.1.4 Les conversions, comparaisons

Il est parfois nécessaire d'effectuer certaines conversions entre le format flottant et un autre format, ne serait-ce que pour l'affichage qui se fait généralement en base 10. Les conversions normalisées sont :

- conversion d'un nombre flottant en entier
- conversion d'un entier en flottant
- conversion d'une valeur flottante vers une valeur entière, avec le résultat dans un flottant
- conversion entre les formats flottants
- conversion entre les formats binaire et décimal.

La norme impose que l'opération de comparaison soit exacte et ne génère pas de dépassement de capacité. Les quatre situations normalisées sont : l'égalité, inférieur, supérieur et non-ordonné. La dernière comparaison, non-ordonné, renvoie vrai lorsqu'une comparaison impliquant un *NaN* est effectuée. Lors d'une comparaison, le signe du zéro ne doit pas être pris en compte.

1.1.1.5 Les arrondis

Le résultat r d'une opération arithmétique dont les opérandes sont des « nombres machine » n'est pas forcément exactement représentable par un élément de l'ensemble des nombres machine, il doit être arrondi. La norme IEEE 754 spécifie quatre modes d'arrondi :

- dirigé vers $+\infty$ (noté $\triangle(r)$) : on fournit le plus petit nombre machine supérieur ou égal à r ;

- dirigé vers $-\infty$ (noté $\bigtriangledown(r)$) : on fournit le plus grand nombre machine inférieur ou égal à r ;
- dirigé vers 0 (noté $\mathcal{Z}(r)$) : on fournit l'arrondi vers $+\infty$ si $r < 0$, et celui vers $-\infty$ si $r \geq 0$;
- au plus près (noté $\circ(r)$) : on fournit le nombre machine le plus proche de r (si r est le milieu de deux nombres machine consécutifs, alors on renvoie celui dont la mantisse est paire. Pour les raisons de ce choix le lecteur pourra consulter Knuth [57] page 236-237).

La norme requiert la propriété suivante sur les opérations arithmétiques :

Propriété 1 (Arrondi correct) *Soient x, y des nombres machine, $\diamond$ le mode d'arrondi choisi et $\bullet$ une des quatre opérations arithmétiques ($+$, $-$, $\times$, $\div$). Le résultat fourni lors du calcul de $(x \bullet y)$ doit être $\diamond(x \bullet y)$.*

Cette propriété est appelée propriété d'*arrondi correct*. Elle est également exigée par la norme pour la racine carrée.

A travers cette thèse, nous désignerons $+$, $-$,$\times$ les opérations arithmétiques usuelles et par $\oplus$, $\ominus$, $\otimes$ les opérations flottantes correspondantes en double précision avec arrondi au plus près.

L'arrondi correct est difficile à obtenir, aussi dans de nombreux cas, nous utiliserons la notion «d'arrondi fidèle» empruntée à Dekker [34], et complétée dans [25]. Cet arrondi a des conditions plus faibles :

Définition 1 (Arrondi fidèle) *L'arrondi fidèle d'un nombre réel correspond à l'un des deux nombres machine l'encadrant, et le nombre machine lui même en cas d'égalité.*

1.1.1.6 Mesure de l'erreur

Lors de l'arrondi une erreur est commise, l'amplitude de cette erreur peut être décrite de différentes façons. L'approximation par un nombre flottant x^* d'un nombre réel x peut se mesurer en erreur absolue, erreur relative ou en nombre d'ulps.

Définition 2 (Erreur absolue) *L'erreur absolue E_a est l'écart entre la valeur réelle x et sa représentation flottante x^* :*

$$E_a = |x - x^*|$$

Définition 3 (Erreur relative) *L'erreur relative E_r est la différence entre le nombre exact x et son approximation x^* divisé par le nombre exact x :*

$$E_r = \frac{|x - x^*|}{|x|} = \frac{E_a}{|x|}$$

Définition 4 (Ulp) *La fonction $ulp(x)$* (unit in the last place) *se définit de la façon suivante selon [12, 44]. Soient x^+, x, et x^- trois nombres représentables en machine consécutifs tels que $|x^-| < |x| < |x^+|$ alors $ulp(x)$ se définit à partir des différences $(x^+ - x)$ et $(x - x^-)$. Cependant ces deux quantités ne sont pas toujours identiques (par exemple si x est une puissance de 2) et pour lever les ambiguïtés on choisit la définition suivante :*

$$ulp(x) = \begin{cases} x^+ - x & si \quad x \geq 0 \quad et \quad x^+ \neq +\infty \\ 2ulp(\frac{x}{2}) & si \quad x \neq +\infty \quad mais \quad x^+ = +\infty \\ ulp(-x) & si \quad x < 0 \end{cases}$$

Nous avons vu qu'il existe trois modes d'arrondi proposés par la norme. Ainsi l'erreur commise lors d'un arrondi au plus près est bornée par $\frac{1}{2}ulp$ du résultat. Pour les modes d'arrondi dirigés, vers 0 ou vers $\pm\inf$, l'erreur maximale est bornée par $1\ ulp$ du résultat.

Définition 5 (ε_n) *Pour tout entier n, nous désignerons par ε_n une quantité α telle que :*

$$|\alpha| \leq 2^n$$

Remarques :

1. L'erreur commise en arrondi au plus près lors de la multiplication de deux flottants double précision a et b est telle que :

   $$a \otimes b = (a \times b).(1 + \varepsilon_{-53})$$

   ce qui est équivalent à $a \otimes b = (a \times b).(1 + \alpha)$ avec $|\alpha| \leq 2^{-53}$.
2. L'erreur commise en arrondi au plus près lors de l'addition de deux flottants double précision a et b est telle que :

   $$a \oplus b = (a + b).(1 + \varepsilon_{-53})$$

   ce qui est équivalent à $a \oplus b = (a + b).(1 + \alpha)$ avec $|\alpha| \leq 2^{-53}$.
3. On a également la propriété suivante : $\varepsilon_a.\varepsilon_b = \varepsilon_{a+b}$
4. L'utilisation de nombre en indice dans ce format permet de représenter facilement soit une erreur relative soit une erreur absolue et de les quantifier. Son défaut par rapport à la représentation utilisée par Higham [44] est de ne pas rendre les preuves portables d'un système de représentation des nombres à un autre.

1.1.2 Quelques valeurs caractéristiques

Le tableau 1.3 nous montre comment sont représentées en mémoire quelques valeurs caractéristiques de la norme. Toutefois une différence est possible entre deux représentations machine du même nombre flottant. Cette différence vient de l'ordre dans lequel sont rangés les mots en mémoire, dont nous détaillerons le principe en section 3.2.1.

1.1.3 Définitions et propriétés sur les nombres flottants

L'intérêt d'une norme est de pouvoir construire des preuves basées sur ses propriétés. Voici quelques résultats utiles :

Théorème 1 (Lemme de Sterbenz [78, 38]) *Si x et y sont des nombres flottants IEEE, et que $y/2 \leq x \leq 2y$ alors $x \ominus y$ est calculé exactement sans erreur d'arrondi.*

Propriété 2 («Continuité» des flottants) *Soit x un nombre flottant IEEE représentant le Zéro positif ou un dénormalisé positif ou un normalisé positif en simple ou double précision (avec bit implicite). Soit R_x l'entier correspondant à la representation en binaire de x.*

Alors $R_x + 1 = x + ulp(x)$ exactement.

Nom	Représentation hexadécimal	Valeur
Nb. normalisé Maximum > 0	`7feffffff ffffffff`	$+1.7976931348623157e^{+308}$
Nb. normalisé Minimum > 0	`00100000 00000000`	$+2.2250738585072014e^{-308}$
Nb. dénormalisé Maximum > 0	`000fffff ffffffff`	$+2.2250728585072009e^{-308}$
Nb. dénormalisé Minimum > 0	`00000000 00000001`	$+4.9406564584124654e^{-324}$
+1	`3ff00000 00000000`	$+1.0$
-1	`bff00000 00000000`	-1.0
+2	`40000000 00000000`	$+2.0$
+0	`00000000 00000000`	0.0
-0	`80000000 00000000`	-0.0
Quiet *NaN* avec plus grande fraction	`7fffffff ffffffff`	QNaN
Quiet *NaN* avec plus petite fraction	`7ff80000 00000000`	QNaN
Signaling *NaN* avec plus grande fraction	`7ff7ffff ffffffff`	SNaN
Signaling *NaN* avec plus petite fraction	`7ff00000 80000001`	SNaN
Infini positif	`7ff00000 00000000`	$+\infty$
Infini négatif	`fff00000 00000000`	$-\infty$

TAB. 1.3: *Quelques exemples de représentation de flottants double précision en machine.*

Exemples :
Supposons que x représente un flottant double précision positif. L'extension à la simple précision est immédiate. Le format de représentation de R_x utilisé dans la suite est le format hexadécimal. Des valeurs hexadécimales en double précision sont données en exemple dans le tableau 1.3.

1. Si x est un nombre flottant égal à $+0$. Alors R_x =`0x00000000 00000000` et R_x+1 =`0x00000000 00000001` qui représente effectivement le plus petit nombre dénormalisé positif, qui correspond au successeur de x.
2. Si x est un nombre flottant égal au plus grand nombre dénormalisé positif. Alors R_x =`0x000fffff ffffffff` et $R_x + 1$ =`0x00100000 00000000` qui représente effectivement le plus petit nombre normalisé positif, qui correspond au successeur de x.
3. Si x est un nombre flottant égal au plus grand nombre normalisé positif. Alors R_x =`0x7feffffff ffffffff` et $R_x + 1$ =`0x7ff00000 00000000` qui représente effectivement l'infini positif, qui correspond au successeur de x.

Remarques :

– Cette propriété lie un nombre flottant et son successeur. Elle peut être très utile pour modifier facilement l'arrondi d'un nombre flottant, sans en connaître de façon exacte sa valeur.
– Cette propriété n'est pas valable pour les formats étendus. Dans ces formats, l'unique bit de la partie entière de la mantisse n'est pas implicite et il est utilisé dans la représentation. Aussi si une retenue se propage dans la représentation de x du bit de poid faible jusqu'au bit représentant la partie entière, alors la représentation résultante n'est plus valide. Le bit de la partie entière n'est plus égal à 1.
– Cette propriété complète la relation [12, 38] entre l'ordre lexicographique de la représentation des flottants positifs et l'ordre sur ces mêmes flottants. Elle est le résultat de la volonté des concepteurs de la norme IEEE-754 de rendre le traitement de certaines opérations possible par les unités entières. En effet les processeurs disposent généralement de plus d'unités entières que d'unités flottantes. De plus les opérations d'incrémentation, de décrémentation, de comparaisons, etc sont effectuées plus facilement sur les unités entières.

Définition 6 (Nombres flottants IEEE) *L'ensemble des* nombres flottants IEEE *comprend l'ensemble des nombres représentables par la norme : les nombres normalisés, les dénormalisés, les deux zéro (± 0), les deux infinis ($\pm\infty$) et les deux* NaN *(sNaN, qNaN).*

Définition 7 (Nombres machine) *Nous appelons l'ensemble des* nombres machine, *l'ensemble composé des nombres IEEE normalisés et dénormalisés. Cet ensemble ne comprend pas les deux infinis ainsi que les* NaN.

1.1.4 Problèmes et limites de la norme

Nous avons vu que la norme est composée d'un ensemble de règles plus ou moins contraignantes. Il est donc rare de trouver des constructeurs qui prennent en charge la norme IEEE-754 entièrement en matériel. Le respect de la norme IEEE-754 s'articule autour d'un ensemble composé du processeur, du système d'exploitation, des bibliothèques ainsi que des compilateurs. La répartition des responsabilitées pour la conformité à la norme IEEE-754 entre chacun de ces éléments varie d'un système à un autre.

Un logiciel tel que PARANOIA de Karpinski [56] permet de détecter si un système donné respecte la norme. Ce logiciel est écrit en langage C, et teste si l'ensemble composé du langage, du compilateur, des bibliothèques, du système et du processeur est cohérent vis-à-vis de la norme IEEE-754.

La suite décrit quelques exemples de problèmes que l'on peut rencontrer avec des systèmes non-cohérents.

Doubles étendus

Comme le fait remarquer Goldberg [38], l'utilisation implicite des doubles étendus peut également engendrer des comportements inattendus. Il donne l'exemple suivant (qui ne fonctionne pas tel quel avec le compilateur gcc dont les optimisations masquent le problème) :

```
1 int main(){
2     double q;
3
4     q = 3.0/7.0;
```

```
5     if (q == 3.0/7.0) printf(''equal \n'');
6     else printf(''not equal \n'');
7     return 0;
8  }
```

Dans cet exemple, comme spécifié par la norme C99 [6], les constantes 3.0 et 7.0 ainsi déclarées sont codées par des flottants double précision. Mais sur une architecture avec des doubles étendus, comme les architectures x86 ou le 68000, l'opération $3.0/7.0$ est effectué avec des doubles étendus et le résultat ne sera converti que lors de son transfert vers la mémoire. Si l'un des deux résultats transite par la mémoire mais pas l'autre, alors l'égalité ne sera pas respectée.

Un problème similaire peut se produire lors de calculs aux abords des limites de codage des nombres flottants. Par conséquent, si une suite de calcul génère un dépassement de capacité vers l'infini, pour ensuite revenir dans le domaine de représentation des nombres IEEE sans transiter par la mémoire alors le résultat final ne conservera aucune trace de ce dépassement de capacité.

Pour éviter les effets de l'excès de précision apporté par ces registres, une solution est de forcer les données à transiter par la mémoire. Cette opération est réalisée par le compilateur. Avec le compilateur gcc, il faut utiliser le drapeau de compilation `-ffloat-store`.

Une autre solution est de définir correctement le registre de contrôle de précision et de mode d'arrondi. Sur un processeur de type x86 avec une distribution linux, il existe un fichier d'entête `fpu_control.h` qui définit les instructions à utiliser pour forcer le processeur à travailler sur des doubles. Le code suivant permet sur une architecture x86 de passer en mode 'arrondi au plus près, double précision'.

```
1  #include <fpu_control.h>
2  #ifndef __set_fpu_cw
3  #define __set_fpu_cw(cw) __asm__ ("fldcw %0" : : "m" (cw))
4  #endif
5  #ifndef _FPU_DBPREC
6  #define _FPU_DBPREC 0x027f
7  #endif
8
9  ...
10
11  __set_fpu_cw(0x027f);
12 /* Travail en mode double précision avec *
13  * arrondi au plus près                  */
14 ...
```

Le FMA

L'opération de Fused Multiply and Add ou *FMA*, introduite par IBM sur le RS/6000, permet d'effectuer $a + (b \times c)$ en une seule opération et avec un seul arrondi final. Il est très utile pour l'évaluation polynomiale ou l'approximation logicielle de la division ou de la racine carré. Cet opérateur est plus précis et plus rapide que la suite d'opérations composée d'une multiplication suivie d'une addition. En effet le FMA n'effectue qu'une seule étape d'arrondi. Cependant, en ne commettant qu'une seule erreur, cet opérateur n'est pas conforme à la norme IEEE-754. La norme impose que le résultat d'une opération flottante qui n'est pas représentable dans le format de destination soit arrondi, or le FMA conserve toute la précision entre la multiplication et l'addition. Le respect de la norme sur les architectures offrant le FMA, telles que les PowerPC

ou Itanium, est laissé au soin du compilateur. Néanmoins des discussions sont en cours pour l'inclure dans la future révision de la norme [7].

Les exceptions

Les architectures modernes proposent un registre spécial pour la gestion des exceptions flottantes. Il est en revanche laissé au soin du compilateur ou du système d'exploitation d'interpréter correctement ce registre.

Par exemple, sur le PowerPC, ce registre de 32 bits s'appelle 'Floating-Point Status and Control Register' (FPSCR). Il est mis à jour après chaque opération flottante, et contient des informations sur les exceptions IEEE, plus quelques autres facilitant leurs gestions, notamment celles relatives aux opérations invalides.

Pour accélérer les calculs, le PowerPC possède également un mode non respect de la norme (mode NI), dont la caractéristique principale est de ne pas prendre en charge les nombres dénormalisés.

1.2 Le dilemme du fabricant de tables

Nous avons vu en section 1.1.1.5 une définition de la propriété d'arrondi correct pour les opérations suivantes : $+$, $-$, $\times$, $\div$, $\sqrt{}$. Malheureusement, la norme ne spécifie rien au sujet de l'arrondi correct des fonctions élémentaires ($\sin$, $\cos$, $\tan$, $\exp\ldots$) principalement à cause du *dilemme du fabricant de tables*. Il n'est en effet pas possible d'évaluer « exactement » les fonctions transcendantales, le résultat est en général un nombre transcendant. On ne peut qu'en calculer une approximation, avec une certaine « précision intermédiaire ».

Il peut arriver parfois que le résultat soit trop proche d'un certain seuil S, pour que la précision de l'approximation ne permette pas de décider dans quelle direction l'arrondi doit se faire. Ce seuil est le milieu de deux nombres machine dans le cas de l'arrondi au plus près (voir la figure 1.3) et les nombres machine eux-mêmes dans le cas de l'arrondi vers $+\infty$, $-\infty$, 0. Donnons un exemple : supposons que l'on cherche à arrondir vers $-\infty$ l'exponentielle de $x = 1,12951$, dans un format virgule flottante en base 10 avec 6 chiffres de mantisse. La valeur exacte de e^x est $3,0941400000125...$, et en l'arrondissant on obtient $3,09414$. Mais si l'approximation calculée est $3,0941399999$, ce qui est une très bonne approximation, on a $3,09413$ en l'arrondissant, ce qui est inexact.

Nos machine actuelles (à l'exception des calculatrices de poche) travaillant en base 2, concentrons-nous sur cette base. Soient n le nombre de bits de mantisse du format virgule flottante considéré et m le nombre de «bits significatifs» du calcul intermédiaire (on veut dire par là, que l'erreur relative d'approximation est majorée par 2^{-m}). Le dilemme du fabricant de table se produit en base 2 :

– dans le cas de l'arrondi vers $+\infty$, $-\infty$, 0, lorsque le résultat est de la forme :

$$\overbrace{\underbrace{1.xxx...xx}_{n\ bits}111111...11}^{m\ bits}\ xxx...$$

ou :

$$\overbrace{\underbrace{1.xxx...xx}_{n\ bits}000000...00}^{m\ bits}\ xxx...$$

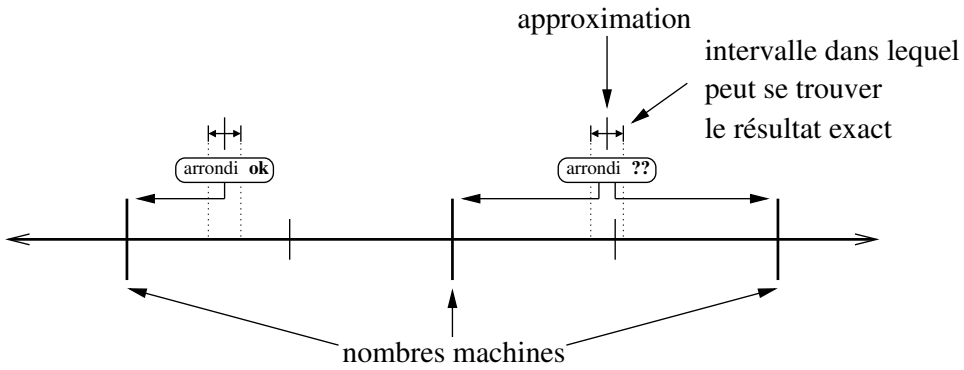


FIG. 1.3: *Dilemme du fabricant de table en arrondi au plus près.*

– dans le cas de l'arrondi au plus près, lorsque le résultat est de la forme :

$$\overbrace{1.\underbrace{xxx...xx}_{n\ bits}011111...11}^{m\ bits}\ xxx...$$

ou :

$$\overbrace{1.\underbrace{xxx...xx}_{n\ bits}100000...00}^{m\ bits}\ xxx...$$

Pour savoir avec quelle précision m il faut calculer les approximations intermédiaires, on peut trouver les « pires cas » du dilemme du fabricant de tables, c'est-à-dire pour une fonction, un domaine et un mode d'arrondi donnés, trouver le nombre virgule flottante dont l'image est la plus proche d'un seuil. Lefèvre a proposé dans sa thèse de doctorat [59] un algorithme de recherche de pires cas. Cet algorithme a permis de borner la précision des résultats intermédiaires nécessaires pour arrondir correctement les fonctions $\exp$, $\log$, 2^x, sur tout l'intervalle des flottants double précision, et pour les fonctions $\sin$, $\cos$, $\tan$, $\arctan$ sur un sous-intervalle centré en zéro des flottants double précision. Un exemple des résultats obtenus est donné dans le tableau 1.2 pour l'exponentielle.

Du tableau 1.2 on peut déduire la propriété suivante.

Propriété 3 (Arrondi correct de l'exponentielle) *Soit y la valeur exacte de l'exponentielle d'un nombre virgule flottante double précision x. Soit y^* une approximation de y telle que la distance entre y et y^* soit majorée par ε.*

- *pour $|x| \geq 2^{-30}$, si $\varepsilon \leq 2^{-53-59} = 2^{-112}$ pour chacun des quatre modes d'arrondi, arrondir y^* est équivalent à arrondir y ;*
- *pour $|x| < 2^{-30}$, si $\varepsilon \leq 2^{-53-104} = 2^{-157}$ alors arrondir y^* est équivalent à arrondir y.*

L'essentiel des travaux de cette thèse consiste à exploiter des théorèmes comme celui ci.

1.3 Vers une normalisation des fonctions mathématiques

Nous avons vu dans la section précédente le dilemme du fabricant de table, qui a constitué un obstacle majeur à l'inclusion des fonctions élémentaires dans la norme IEEE-754. Ce

Intervalle	pire cas (en binaire)
$[\ln(2^{-1074}), -2^{-30}]$	$\exp(-1.11101101001100011000111011111011011000100111111101010 \times 2^{-27})$ $= 1.11111111111111111111111111000010010110011100111000100 \quad 1 \quad 1^{59}0001\ldots \times 2^{-1}$
$[-2^{-30}, 0)$	$\exp(-1.00000000000000000000000000000000000000000000000000000001 \times 2^{-51})$ $= 1.11111111111111111111111111111111111111111111111111100 \quad 0 \quad 0^{100}1010\ldots \times 2^{-1}$
$(0, +2^{-30}]$	$\exp(1.11111111111111111111111111111111111111111111111111111111 \times 2^{-53})$ $= 1.00000000000000000000000000000000000000000000000000000000 \quad 1 \quad 1^{104}0101\ldots$
$[2^{-30}, \ln(2^{1024})]$	$\exp(1.01111111111111100111111111111111011100000000000100100 \times 2^{-32})$ $= 1.00000000000000000000000000000001011111111111111101000 \quad 0 \quad 0^{57}1101\ldots$ $\exp(1.10000000000000010111111111111111011011111111111011100 \times 2^{-32})$ $= 1.00000000000000000000000000000011000000000000000010111 \quad 1 \quad 1^{57}0010\ldots$ $\exp(1.10011110100111001011101111111101011000001000000001011 \times 2^{-31})$ $= 1.00000000000000000000000000000011001111010011100101111 \quad 1 \quad 0^{57}1010\ldots$ $\exp(110.00001111010100101111001101111010111011001111110100)$ $= 110101100.01010000101101000000100111001000101011101110 \quad 0 \quad 0^{57}1000\ldots$

TAB. 1.4: *Pires cas pour l'exponentielle et le format virgule flottante double précision.*

manque de norme a contribué au développement de bibliothèques mathématiques de piètre qualité. Néanmoins des résultats nouveaux et des bibliothèques mathématiques de meilleure qualité sont apparus depuis la création de la norme IEEE-754. Il est par conséquent légitime de s'intéresser à quelques aspects de ce que devrait contenir un standard sur les fonctions élémentaires. Les réflexions que nous avons menées dans ce domaine [32], sont motivées par les discussions tenues au sein du comité de révision de la norme IEEE-754.

Les suggestions faites dans cette section s'appuient sur :
- les suggestions faites dans les normes IEEE-754, IEEE-854 [47, 48, 19],
- les travaux de Kahan dans ce domaine [51, 52, 53, 54],
- les discussions en cours pour la révision de la norme IEEE-754 [7],
- les discussions faites à ce sujet par d'autres comités [5, 6],
- notre propre expérience dans le domaine des fonctions élémentaires,
- les récents progrès sur le problème du dilemme du fabricant de tables [61, 60, 77].

1.3.1 Propriétés souhaitables

Parmi les propriétés souhaitables pour les fonctions élémentaires nous rencontrons :

1. Arrondi correct pour tous les modes d'arrondi.
2. Préservation de l'intervalle de sortie : Par exemple, nous voulons que le sinus soit effectivement entre -1 et $+1$, que l'arctangente soit entre $-\pi/2$ et $+\pi/2$, etc. Si cette propriété n'est pas préservée alors des conséquences non négligeables peuvent apparaître. Ainsi $\tan(\arctan(x))$ peut conduire à une importante erreur, dans le cas où l'arrondi de $\tan(x)$ renvoie une valeur plus grande que $\pi/2$, même légèrement supérieure.
3. Borne sur l'erreur maximale commise. Cette borne doit être fournie dans la documentation des bibliothèques mathématiques.
4. Préservation de la monotonie.
5. Préservation de la symétrie.

6. Préservation de la direction de l'arrondi lorsqu'un arrondi dirigé est sélectionné, même si l'arrondi ne peut être satisfait.
7. Gestion correcte des exceptions et des nombres dénormalisés.
8. Renvoi d'un *NaN* chaque fois qu'une fonction ne peut être définie de façon unique en utilisant la continuité : par exemple $1^{\pm\infty}$ ou $\sin\infty$. Une exception peut être faite pour 0^0 qui n'est pas défini de façon unique mais pour lequel la convention $0^0 = 1$ présente l'avantage de préserver de nombreuses propriétés mathématiques.
9. Compatibilité avec d'autres normes, tels que ISO/IEC 10967 (Language Independent Arithmetic (LIA) et en particulier LIA-2, la deuxième des trois parties de LIA [5]) et les normes relatives aux langages telle que ISO/IEC 9899 pour le langage C [6].

Nous noterons que ce ne sont là que des propriétés souhaitables et que certaines ne sont pas compatibles entre elles. Par exemple, en simple ou double précision, l'arrondi correct de l'arctangente renvoie en arrondi au plus près une valeur plus grande que $\pi/2$. Par cet exemple nous observons que les propriétés 1 et 2 ne sont pas compatibles pour le cas de l'arctangente.

1.3.2 Les fonctions considérées

Les fonctions auxquelles nous nous attachons sont les fonctions incluses dans les bibliothèques mathématiques. Pour des raisons de compatibilité, nous avons également complété cette liste avec celles auxquelles s'attachent les autres normes, notamment C99 et LIA-2. Pour une définition mathématique de ces fonctions nous suggérons la lecture de [8] :

> $\log$, $\log_2$, $\log_{10}$, $\log_b x$, $\log(1+x)$, $\exp$, $\exp(x)-1$, 2^x, 10^x, $\sin$, $\cos$, $\tan$, $\cot$, $\sec$, $\csc$, $\arcsin$, $\arccos$, $\arctan$, $\arctan\frac{x}{y}$, arccot, arcsec, arccsc, x^y, $\sinh$, $\cosh$, $\tanh$, $\coth$, sech, csch, $\operatorname{arcsinh}$, $\operatorname{arccosh}$, $\operatorname{arctanh}$, $\operatorname{arccoth}$, $\operatorname{arcsech}$, $\operatorname{arccsch}$.
>
> La plupart de ce qui est énoncé ici peut s'appliquer aux fonctions spéciales (gamma, erf, erfc, fonction de Bessel, etc.) et à quelques fonctions algébriques telles que la racine carrée inverse, la racine cubique, ou l'hypoténuse.

1.3.3 Choix des critères de qualités

Nous avons vu qu'il existe différents critères souhaitables pour les fonctions élémentaires, dont certains sont mutuellement incompatibles. Nous proposons que les propriétés souhaitées soient choisies par l'intermédiaire d'options. Les options possibles sont `preserve-rounding`, `preserve-range` ou `preserve-symmetry`. Les arrondis dirigés sont principalement utilisés en arithmétique par intervalles. L'arrondi au plus près est quant à lui, utilisé pour obtenir des résultats les plus précis possibles et un comportement proche de la fonction mathématique, c'est pourquoi nous suggérons que le choix des options par défaut soit `preserve-range` pour l'arrondi au plus près et `preserve-rounding` pour les arrondis dirigés.

1.3.4 Les trois niveaux d'arrondi proposés

Nous suggérons trois niveaux de qualité pour l'arrondi du résultat, le niveau offert par la bibliothèque devant être clairement indiqué dans la bibliothèque. Ces niveaux correspondent à un compromis entre qualité et vitesse d'exécution. Par défaut, le niveau privilégiant la qualité doit être sélectionné (au contraire de ce qui est bien souvent fait dans les compilateurs).

Niveau 0 : Arrondi fidèle et erreur relative garantie
- En arrondi au plus près, la valeur renvoyée doit toujours être l'un des deux nombres flottants qui encadrent immédiatement le résultat exact.
- En arrondi vers $-\infty$, la valeur renvoyée doit toujours être inférieure ou égale au résultat exact.
- En arrondi vers $+\infty$, la valeur renvoyée doit toujours être supérieure ou égale au résultat exact.
- L'arrondi vers 0, doit se comporter comme l'arrondi vers $-\infty$ pour les valeurs positives et comme l'arrondi vers $+\infty$ pour les valeurs négatives.
- L'erreur maximale autorisée est de $1,5\ ulp$ pour les arrondis dirigés.
- Dans tous les cas où la fonction est monotone, l'implémentation doit également être monotone.
- En arrondi au plus près et en arrondi vers zéro, l'éventuelle symétrie de la fonction autour de 0 doit être préservée.

Niveau 1 : Arrondi correct sur un intervalle fixé
- L'implémentation doit satisfaire les critères de niveau 0.
- Il doit exister un intervalle, habituellement autour de zéro, où l'implémentation de la fonction doit être correctement arrondie. Nous suggérons, par exemple, que ce domaine doit au minimum contenir $[-2\pi, +2\pi]$ pour les fonctions $\sin$, $\cos$ et $\tan$; et $[-1, 1]$ pour exp, $cosh$, $sinh$, 2^x et 10^x.

Niveau 2 : Arrondi correct
- L'arrondi correct doit être fourni par l'implémentation sur tout l'intervalle de définition de la fonction.
- Nous suggérons l'utilisation du mode `preserve-range` lorsque l'intervalle de sortie est prioritaire. Dans ce cas, l'arrondi correct est fourni dans tous les cas ou cet arrondi est compatible avec l'intervalle de sortie.
- Dans le cas d'un arrondi dirigé, nous supposons que l'utilisateur a connaissance de l'incompatibilité avec la propriété de symétrie, donc le mode `preserve-symmetry` n'est pas disponible.
- *Remarque :* L'arrondi correct ne peut être incompatible avec la monotonie.

Nous noterons que l'erreur donnée par le niveau 0 peut être remplacée par une erreur plus générale. L'erreur relative doit être bornée par $(1/2 + \tau)$ ulp pour l'arrondi au plus près et par $(1 + \tau)$ ulp pour les autres modes d'arrondis. Pour le niveau 0, le seuil de tolérance τ ne peut pas excéder la valeur $1/2$, tandis que pour le niveau 2 ce seuil est fixé à 0.

Nous considérons le niveau 0 comme le niveau minimum acceptable. En effet, il existe déjà des bibliothèques [2, 3] satisfaisant ce niveau pour l'arrondi au plus près. De plus, les récentes avancées sur la recherche des pires cas [60, 61, 77] nous laissent entrevoir des méthodes à coût raisonnable pour quelques fonctions en simple et en double précision. Nous verrons dans le chapitre 5 une méthode ainsi qu'une étude sur le surcoût de ce niveau 2 pour la fonction $\exp$.

1.3.5 Les exceptions

Pour des raisons de concision, dans la suite, le signe sera symbolisée par $(-1)^s$ avec $s = \{0, 1\}$.

1.3.5.1 Valeurs non définissables par continuité

Lorsque la fonction considérée n'est pas définie mathématiquement, nous avons plusieurs possibilités. L'important est de rester cohérent dans les choix faits. Voici trois exemples typiques :
Exemple 1 : Le cas 0^0 est important. Comme nous l'avons mentionné plus haut, nous ne pouvons pas le définir en utilisant la continuité. En revanche, de nombreuses propriétés restent satisfaites si nous choisissons $0^0 = 1$ (ce qui est fréquemment adopté par convention par les mathématiciens). Kahan [53] a également suggéré de choisir $0^0 = 1$ ce qui a pour conséquence que $NaN^0 = 1$ alors que $0^{NaN} = NaN$.
Exemple 2 : Un autre exemple est celui de $\log(-0)$. D'un coté, comme le suggère Goldberg [38], nous pouvons voir -0 comme un petit nombre négatif qui a débordé vers zéro. Ce qui favorise le choix : $\log(-0) = NaN$. D'un autre coté, nous savons que la norme IEEE-754 impose que $+0 = -0$ ce qui conduit à avoir $x = y$ et $\log x \neq \log y$ lorsque $x = +0$ et $y = -0$.
Exemple 3 : Le cas $1^{\pm\inf}$ est similaire à 0^0. On peut construire des suites $u_n \to 1$ et $v_n \to +\infty$ telles que $u_n^{v_n}$ tend vers ce que l'on veut. A cet effet, Kahan [53] suggère $1^{\pm\infty}$= *NaN*, ce qui implique pour des raisons de cohérence $1^{NaN} = NaN$.

1.3.5.2 Les *NaN*s

Nous suggérons de façon à rester cohérent avec la norme IEEE-754 que toute fonction prenant un *NaN* en entrée retourne un *NaN* comme résultat. Une éventuelle exception peut être faite pour $NaN^0 = 1$ (si 0^0 est défini comme étant égal à 1).

De plus des *NaN* pourront être générés dans les cas suivants :
- $\log$, $\log_2$, $\log_{10}$, $\log_b$ d'un nombre négatif,
- $\log_b$ pour $b < 0$,
- $\log(1 + x)$ pour $x < -1$,
- x^y pour $x < 0$ et $y \notin \mathbb{N}$
- 1^{NaN} doit être un *NaN* si 1^∞ est un *NaN*,
- $\sin, \cos, \tan, \cot, \sec, \csc$ de $\pm\infty$.
- $\arcsin x$, $\arccos x$, $\operatorname{arctanh} x$ pour $x \notin [-1, +1]$,
- $\operatorname{arccosh} x$ pour $x < 1$, $\operatorname{arccoth} x$ pour $|x| \notin]-1, +1[$,
- $\operatorname{arcsech} x$ pour $x \notin [0, 1]$,
- $\arctan\left(\frac{x}{y}\right)$ pour $y = 0$,
- $\operatorname{arccot} 0$ est soit un *NaN* car arccot n'est pas définie en 0, soit $\operatorname{arccot}(-0) = -\pi/2$ et $\operatorname{arccot}(+0) = \pi/2$, ($\pi/2$ n'est pas une constante exactement représentable en machine, aussi le résultat n'est pas $\pi/2$, mais l'arrondi de $\pi/2$),
- $\operatorname{arcsec} x$ et $\operatorname{arccsc} x$ pour $x \in \{-1, 1\}$,

Nous noterons qu'il n'est pas envisageable de retourner un *NaN* lorsque le résultat mathématique est défini (par exemple le sinus d'un très grand nombre).

1.3.5.3 Les infinis

Deux types d'infini peuvent être produits :

L'infini comme le résultat d'un dépassement de capacité :
Il se produit lorsque le résultat mathématique est correctement défini mais que ce

résultat est de valeur absolue supérieure au plus grand nombre représentable.

L'infini exact :
Il peut être produit par les opérations suivantes :
- $\log_\beta(+0) = -\infty$ avec $\beta \in \{e, 2, 10, b(b > 0)\}$ et $\operatorname{arcsech}(+0) = +\infty$, avec le drapeau "division par zéro" levé. Il peut en être de même pour $log(-0)$, voir discussion de la section 1.3.5.1.
- $f((-1)^s 0) = (-1)^s \infty$ pour $f = \cot, \csc, \sinh, \cosh, \coth, \operatorname{csch}, \operatorname{arccsch}$
- $((-1)^s 0)^x = (-1)^s \infty$ avec $x < 0$, comme recommandé dans [53],
- $\operatorname{arctanh}((-1)^s) = (-1)^s \infty$, $\operatorname{arccoth}((-1)^s) = (-1)^s \infty$.
- $\log_\beta(+\infty) = +\infty$ avec $\beta \in \{e, 2, 10, b(b > 0)\}$,
- $\log(1 + \infty) = +\infty$,
- $\log(1 + x) = -\infty$ pour $x = -1$,
- $\beta^{+\infty} = +\infty$ avec $\beta \in \{e, 2, 10, b(b > 0)\}$,
- $\exp(+\infty) - 1 = +\infty$,
- $\operatorname{arcsinh}((-1)^s \infty) = (-1)^s \infty$,
- $\operatorname{arccosh}(+\infty) = +\infty$.

Nous noterons que les fonctions tangente, cotangente, sécante et cosécante ne renvoient jamais l'infini pour les précisions courantes. En effet il n'existe pas de nombre flottant simple ou double précision assez proche d'un multiple de $\pi/2$ [67].

1.3.5.4 Drapeau résultat inexact

Comme le souligne Muller dans [67], dont en voici un extrait, le résultat d'une fonction élémentaire peut être exact dans certain cas.

> It is difficult to know when functions such as x^y or $\log_b x$ give an exact result. However, using a theorem due to Lindemann, one can show that the sine, cosine, tangent, exponential, or arctangent of a nonzero finite machine number, or the logarithm of a finite machine number different from 1 is not a machine number, so that its computation is always inexact.

Nous pouvons compléter cet extrait sur les opérations exactes, par les autres fonctions les plus courantes :
Pour le logarithme et l'exponentielle en base β, tel que $\beta \in e, 2, 10, b(b > 0)$:

1. $\log_\beta(\pm 0) = -\infty$ ou *NaN* (cf. §1.3.5.1) et $\log_\beta(+\infty) = +\infty$;
2. $\log_\beta 1 = +0$ excepté pour $\bigtriangledown(\log_\beta 1) = -0$;
3. $\beta^{-\infty} = +0$ et $\beta^{+\infty} = +\infty$;
4. β^p, pour tout entier p tel que β^p est exactement représentable. En particulier :
   - $\beta^0 = 1$;
   - lorsque $\beta = 2$ alors $\log_2 2^p = p$ et $\exp_2 p = 2^p$;
   - lorsque $\beta = 10$ alors $\log_{10} 10^p = p$ et $\exp_{10} p = 10^p$;
   - lorsque $\beta = b$ alors $\log_b b^p = p$ et $\exp_b p = b^p$;
5. $\log_e(1 + x) = -\infty$ pour $x = -1$, 0 pour $x = 0$ et $+\infty$ pour $x = +\infty$;
6. $\exp_e(x) - 1 = +0$ pour $x = -\infty$, 0 pour $x = 0$ et $+\infty$ pour $x = +\infty$.

Pour les fonctions trigonométriques et leurs réciproques :

1. $f(\pm 0) = \pm 0$ pour $f = \sin, \tan, \arcsin, \arctan$;

2. $\cos 0 = 1$;
3. $\cot((-1)^s 0) = (-1)^s \infty$;
4. $\sec 0 = 1$;
5. $\csc((-1)^s 0) = (-1)^s \infty$;
6. $\arccos 1 = +0$;
7. $\arctan\left(\frac{(-1)^s 0}{y}\right) = (-1)^s \operatorname{sign}(y) 0$ pour $y \neq 0$;
8. $\operatorname{arccot}((-1)^s \infty) = (-1)^s 0$
9. $\operatorname{arcsec} 1 = +0$ excepté pour $\bigtriangledown(\operatorname{arcsec} 1) = -0$;
10. $\operatorname{arccsc}((-1)^s \infty) = (-1)^s 0$.

Pour les fonctions hyperboliques et leurs réciproques :

1. $f(\pm 0) = \pm 0$ pour $f = \sinh, \tanh, \operatorname{arcsinh}, \operatorname{arctanh}$;
2. $f((-1)^s \infty) = (-1)^s \infty$ pour $f = \sinh, \operatorname{arcsinh}$;
3. $\cosh 0 = 1$ et $\cosh(\pm\infty) = +\infty$;
4. $\tanh((-1)^s \infty) = (-1)^s 1$;
5. $\coth((-1)^s 0) = (-1)^s \infty$ et $\coth((-1)^s \infty) = (-1)^s 1$;
6. $\operatorname{sech} 0 = 1$ et $\operatorname{sech}(\pm\infty) = +0$;
7. $\operatorname{csch}((-1)^s 0) = (-1)^s \infty$ et $\operatorname{csch}((-1)^s \infty) = (-1)^s 0$;
8. $\operatorname{arccosh} 1 = +0$ et $\operatorname{arccosh}(+\infty) = +\infty$;
9. $\operatorname{arctanh}((-1)^s 1) = (-1)^s \infty$;
10. $\operatorname{arccoth}((-1)^s \infty) = (-1)^s 0$ et $\operatorname{arccoth}((-1)^s 1) = (-1)^s \infty$;
11. $\operatorname{arcsech} 1 = +0$, $\operatorname{arcsech}(+0) = +\infty$, $\operatorname{arcsech}(-0)$ est $+\infty$ ou *NaN* (voir discussion en 1.3.5.1 ;
12. $\operatorname{arccsch}((-1)^s \infty) = (-1)^s 0$ et $\operatorname{arccsch}((-1)^s 0) = (-1)^s \infty$;

1.3.6 Interactions avec les autres normes

À travers les différentes propositions que nous avons émises, nous avons essayé de rester cohérent avec les autres normes traitant des fonctions mathématiques (la norme LIA-2 [5] et la norme C99 [6]). En ne citant que ces normes, nous sommes loin de l'exhaustivité. En effet, chaque langage a sa norme et sa propre collection de critères à satisfaire pour l'évaluation des fonctions mathématiques.

Toutefois ces propositions diffèrent en quelques points :

Précision. La plupart des normes liées à des langages comme la norme C99, n'ont aucune spécification quant à la précision des fonctions élémentaires. En revanche, la norme LIA-2 traite de la précision que doivent fournir les évaluations des fonctions mathématiques. La précision varie entre $0.5 \times C$ et $1.5 \times C$ à $2 \times C$ où C est une constante dépendant du système de représentation. Il existe de plus plusieurs intervalles de précision définis pour des groupes de fonctions mathématiques. Dans le cas des fonctions trigonométriques cette spécification n'est requise que sur un sous-intervalle centré en zéro.

Propriétés mathématiques. Comme pour la précision, la norme C99 ne spécifie rien quant aux propriétés que doivent satisfaire les implémentations. Parmi toutes les propriétés que nous avons énoncées, la norme LIA-2 n'évoque que la monotonie. Elle impose cette propriété pour les fonctions trigonométriques sur un intervalle centré en zéro, dépendant du système de représentation ($[-2^{27}, +2^{27}]$ pour la double précision). Pour les autres fonctions mathématiques, la propriété de monotonie est imposée sur tout l'intervalle où les fonctions sont définies.

Les exceptions. Mis à part quelques désaccords portant principalement sur les questions que nous avons laissées ouvertes (voir discussion en 1.3.5.1), toutes les normes ont une approche identique sur la question des exceptions.

Remarques diverses. La norme LIA-2 impose des éléments qui semblent difficile à satisfaire dans un système à virgule flottante. A de nombreuses reprises, elle émet des conditions sur le résultat d'une fonction lorsque l'argument en entrée est un nombre irrationnel, donc non représentable en machine (par exemple e ou π).

Nous noterons également que la troisième partie de la norme LIA est consacrée aux fonctions complexes, alors que les propositions que nous avons soumises ne les concernent pas.

1.4 Conclusion

La norme IEEE-754 a constitué une révolution dans le monde de l'arithmétique virgule flottante. Elle a facilité l'échange des données entre processeurs, la portabilité du code et la construction de preuves. À travers le dilemme du fabricant de table nous comprenons aisément pourquoi les fonctions élémentaires n'ont pas été incluses dans cette norme. Mais depuis, de récents travaux dans la recherche des pires cas [60, 61, 67, 69, 77] pour les fonctions élémentaires nous laissent entrevoir une solution au dilemme du fabricant de table.

Il n'est donc pas illusoire d'espérer voir les fonctions élémentaires incluses dans une révision de la norme IEEE-754. Pour aller dans cette direction nous avons donc lancé la discussion en proposant une liste de propriétés souhaitables pour les fonctions élémentaires. Disposer d'une norme pour les fonctions élémentaires aurait alors les mêmes conséquences sur les calculs numériques que celle qu'a eu la norme IEEE-754 sur les opérations flottantes : principalement meilleure précision, déterminisme des algorithmes et possibilité de construction de preuves.

CHAPITRE

2

2.1.1 État de l'art

Les tables multipartites

Introduite en 1995 par Das Sarma et Matula [23] dans le cas de la fonction $1/x$, cette méthode a été généralisée en 1999 de façon indépendante par Schulte et Stine [79] et par Muller [68] puis améliorée en 2001 par Dinechin et Tisserand [27]. Elle consiste en une approximation au premier ordre des fonctions où les multiplications sont remplacées par des lectures de table. Elle permet d'obtenir l'arrondi fidèle (voir définition 1) pour des précisions d'environ 16 bits en utilisant des tables de quelque Koctets et de petits additionneurs.

Les autres méthodes à base de table et d'addition

Hassler et Tagaki [41] décomposent une approximation polynomiale d'une fonction en une somme de produits de bits du mot d'entrée. L'ensemble de ces produits partiels est décomposé de façon heuristique en sous-ensembles mémorisés dans des tables, et fusionnés à l'aide d'un additionneur multi-opérandes. Une autre méthode est donnée par Wong et Goto [82], où les additions sont effectuées avant et après les lectures de table. Bien que ces deux méthodes considèrent des termes d'ordre supérieur, il a été prouvé qu'elles conduisent à de plus grosses tables et à une architecture plus lente que les tables multipartites généralisées [27].

Les méthodes de plus grand ordre

La taille exponentielle des tables rend les méthodes précédentes non utilisables pour des précisions supérieures à 24 bits. En 2001, Bruguera, Piñeiro et Muller ont par conséquent présenté une architecture basée sur une approximation de second ordre en utilisant des multiplieurs et des unités d'élévation au carré [73]. Liddicoat et Flynn [64] ont également proposé en 2001, une architecture pour calculer l'inverse en utilisant plusieurs unités d'élévation à la puissance fonctionnant en parallèle, suivies d'un additionneur multi-opérande.

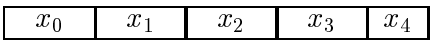
2.1.2 Une méthode à base de table avec de petits multiplieurs

La méthode présentée ici est une version intermédiaire entre les méthodes à base de lecture de tables et d'additions et les méthodes basées sur des multiplications. Elle a fait l'objet d'une publication [31].

2.1.2.1 Approximation mathématique

Supposons que l'on souhaite évaluer $f(x)$ avec $x \in [0,1[$ un nombre en virgule fixe sur $n = 4k + p$ bits. Donc $x = x_0 + x_1 2^{-k} + x_2 2^{-2k} + x_3 2^{-3k} + x_4 2^{-4k}$, et les x_i sont des nombres codés sur k bits appartenant à $[0,1[$, sauf pour x_4 qui lui est sur p bits, où $p < k$.

Nous avons :

$x =$ 

La méthode consiste à écrire le développement de Taylor à l'ordre 5 de f en x_0 et à ne conserver que les termes prépondérants pour le résultat final. Rappelons que la précision visée

pour le résultat est identique à la précision du nombre en entrée qui est de l'ordre de 2^{-4k-p}. Nous obtenons :

$$\begin{array}{rcll} f(x) & = & f(x_0) & (T_0) \\ & + & [x-x_0]f'(x_0) & (T_1) \\ & + & \frac{1}{2}[x-x_0]^2 f''(x_0) & (T_2) \\ & + & \frac{1}{6}[x-x_0]^3 f'''(x_0) & (T_3) \\ & + & \frac{1}{24}[x-x_0]^4 f^{(4)}(x_0) & (T_4) \\ & + & \frac{1}{120}[x-x_0]^5 f^{(5)}(x_0) & (T_5) \\ & + & \varepsilon_1 & \end{array} \tag{2.1}$$

où

$$\varepsilon_1 = \frac{1}{720}([x_1 2^{-k} + x_2 2^{-2k} + x_3 2^{-3k} + x_4 2^{-4k}]^6) f^{(6)(\theta)} < \frac{1}{720} 2^{-6k} \max |f^{(6)}(\theta)|$$

avec $\theta \in [0,1]$. En développant l'équation (2.1) et en ne gardant que les termes qui ont une influence sur le résultat, nous obtenons la méthode *multiplicative à base de tables* suivante :

$$\begin{array}{rcl} f(x) & = & A(x_0,x_1) + B(x_0,x_2) + C(x_0,x_3) + D(x_0,x_4) + \\ & & x_2 \times E(x_0,x_1) + x_3 2^{-k} \times E(x_0,x_1) + \varepsilon_f \end{array} \tag{2.2}$$

où

$$\begin{array}{rcl} A(x_0,x_1) & = & f(x_0) + x_1 2^{-k} f'(x_0) + \frac{1}{2} x_1^2 2^{-2k} f''(x_0) + \\ & & \frac{1}{6} x_1^3 2^{-3k} f'''(x_0) + \frac{1}{24} x_1^4 2^{-4k} f^{(4)}(x_0) + \\ & & \frac{1}{120} x_1^5 2^{-5k} f^{(5)}(x_0) + \frac{1}{2} x_1 (1/2) 2^{-5k} f''(x_0) + \\ B(x_0,x_2) & = & x_2 2^{-2k} f'(x_0) + \frac{1}{2} x_2^2 2^{-4k} f''(x_0) + (1/2) x_2 2^{-5k} f''(x_0) \\ C(x_0,x_3) & = & x_3 2^{-3k} f'(x_0) \\ D(x_0,x_4) & = & x_4 2^{-4k} f'(x_0) \\ E(x_0,x_1) & = & x_1 2^{-3k} f''(x_0) + \frac{1}{2} x_1^2 2^{-4k} f'''(x_0) + \\ & & \frac{1}{6} x_1^3 2^{-5k} f^{(4)}(x_0) + \frac{1}{2} x_1 (1/2) 2^{-5k} f'''(x_0) \\ \varepsilon_f & \leq & \frac{1}{720} 2^{-6k} \max |f^{(6)}| + \frac{1}{2} 2^{-5k} \max |f''| + \frac{1}{3} 2^{-6k} \max |f'''| + \\ & & \frac{1}{24} 2^{-6k} \max |f^{(4)}| + \frac{1}{120} 2^{-6k} \max |f^{(5)}| \\ & \leq & 2^{-5k} \left[\frac{1}{2} \max |f''| + \frac{1}{3} \max |f'''| + \right. \\ & & \left. \frac{1}{24} \max |f^{(4)}| + \frac{1}{120} \max |f^{(5)}| + \frac{1}{720} \max |f^{(6)}| \right] . \end{array}$$

L'évaluation de $f(x)$ peut être faite en effectuant seulement 2 «petites» multiplications et 5 additions, le tout avec une erreur inférieure à ε_f. Les valeurs $A(x_0,x_1)$, $B(x_0,x_2)$, $C(x_0,x_3)$, $D(x_0,x_4)$ et $E(x_0,x_1)$ sont pré-calculées pour la fonction cible et pour chaque argument possible. La taille des tables utilisées dépend donc de la fonction considérée.

Nous noterons que cette méthode exploite une propriété de la dérivée de la fonction. Les dérivées successives doivent en effet ne pas croître trop rapidement, pour rendre les termes absents des tables non significatifs par rapport aux autres.

L'architecture correspondant à cette méthode est présentée en figure 2.1.

2.1.2.2 Calcul de l'erreur d'arrondi

En plus de l'erreur mathématique d'approximation ε_f, cette architecture à d'autre sources d'erreurs dont il faut tenir compte :

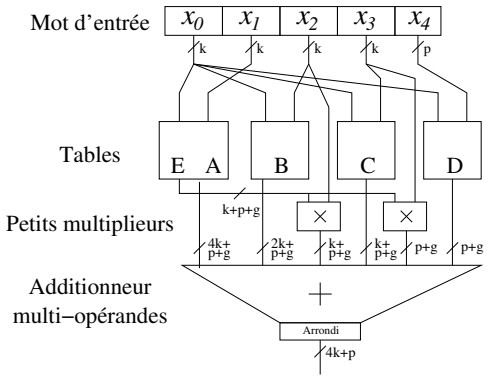


FIG. 2.1: *Architecture de la méthode à base de petite multiplication et de lecture de table.*

- Les tables doivent être remplies avec des valeurs en virgule fixe correspondant aux valeurs exactes arrondies à la précision cible.
- De façon similaire les résultats des multiplications doivent également être arrondis à la précision cible.
- Ces erreurs d'approximation, plus l'erreur mathématique nécessitent l'utilisation de g bits additionnels (ou bits de garde) dans les tables et à la sortie des multiplieurs (voir figure 2.1).
- Enfin le résultat final doit être arrondi à la précision cible après la dernière addition, avec une erreur d'au plus 1 ulp du résultat pour garantir l'arrondi fidèle.

Cette méthode nécessite les tailles de tables suivante :

$$\begin{aligned} Taille(A[x_0, x_1]) &\leq 2^{2.k}(4k + p + g) \\ Taille(B[x_0, x_2]) &\leq 2^{2.k}(2k + p + g) \\ Taille(C[x_0, x_3]) &\leq 2^{2.k}(k + p + g) \\ Taille(D[x_0, x_4]) &\leq 2^{p+k}(p + g) \\ Taille(E[x_0, x_1]) &\leq 2^{2.k}(k + p + g) \end{aligned}$$

Le multiplieur utilisé pour multiplier E par x_2 est de taille $(k) \times (k + p + g)$, et l'autre utilisé pour multiplier E par x_3 est de taille $(k) \times (p + g)$.

Le nombre de bits additionnels g correspond au plus petit nombre tel que l'arrondi fidèle est garanti pour tous les arguments en entrée.

Nous noterons que l'arrondi des résultats des multiplications, et l'arrondi final, peuvent être effectués par simple troncature (sans surcoût au niveau matériel), si un terme correcteur est ajouté aux valeurs utilisées pour remplir la table A.

2.1.2.3 Résultats

Nous avons réalisé un programme C simulant la méthode présentée ci-dessus. Il remplit les tables et détermine l'erreur totale par énumération, pour des valeurs croissantes du nombre de bit de garde, jusqu'à ce que l'arrondi fidèle soit obtenu.

Le tableau 2.1 présente les résultats pour différentes fonctions générées par le programme C. Les meilleurs résultats de tailles de tables obtenus pour les méthodes multipartites généralisées [27] sont également présents pour établir une comparaison et constituent les tailles de référence.

f	k	p	Taille de table	Nombre de bits corrects	Taille de référence
$\sin$ $[0, \pi/4[$	3	2	2768	14	3712
	4	3	15040	19	29440
	5	3	70528	23	138624
$\exp$ $[0, 1[$	3	2	3232	14	6272
	4	3	16256	19	56320
	5	4	82432	24	366080
$2^x - 1$ $[0, 1[$	3	2	3392	14	7168
	4	3	18048	19	56320
	5	4	89600	24	259584

TAB. 2.1: *Taille de tables et précision avec l'arrondi fidèle, pour des entrées variant entre 14 et 24 bits de précision.*

2.1.2.4 Conclusion et application aux FPGA et VLSI

La méthode que nous venons de présenter constitue une importante amélioration des méthodes multipartites lorsque nous visons une implémentation sur FPGA de dernière génération (Virtex II et Virtex II Pro). Ces FPGA disposent en effet de nombreux multiplieurs (18 bits $\times$ 18 bits $\rightarrow$ 35 bits), dont la précision est plus que suffisante pour celles visées par cette méthode. Ils permettent de réduire d'un facteur trois la taille de tables nécessaire sans surcoût additionnel.

Lorsque l'on vise une implémentation sur VLSI, le principal compromis de la méthode est d'échanger du silicium utilisé pour les multiplieurs par du silicium utilisé pour les tables. Une implémentation réelle sur VLSI serait légèrement différente de celle que l'on peut voir sur le schéma 2.1. Des modifications seraient apportées sur les multiplications ainsi que la façon d'effectuer la troncature dans les produits partiels de la multiplication.

2.2 Évaluation logicielle

Les méthodes à base de lecture de tables et de petits opérateurs, additions ou multiplications, sont efficaces seulement pour de petites précisions. Lorsque l'on souhaite évaluer ces mêmes fonctions pour des précisions plus importantes, alors il faut se tourner vers des algorithmes différents, qui sont souvent basés sur des approximations polynomiales.

Cependant ces approximations ne sont précises que sur de petits intervalles, généralement centrés sur 0. Un exemple est donné dans le tableau 2.2 pour la fonction exponentielle. Sur cet exemple on observe qu'il est presque impossible d'évaluer un nombre dont l'amplitude serait de 2^3, l'erreur relative serait de l'ordre de 2^{20} : il doit être réduit. Cette étape appelée *réduction d'argument* s'accompagne également (en général) en fin du processus d'évaluation d'une étape de *reconstruction*.

Nous présentons dans cette section chacune des étapes qui composent l'évaluation logicielle ainsi que notre contribution à la réduction d'argument en grande base. Pour la description d'autres schémas d'évaluation nous invitons le lecteur à consulter [21, 65, 67].

k	ε
-5	ε_{-27}
-4	ε_{-23}
-3	ε_{-19}
-2	ε_{-15}
-1	ε_{-10}
-0	ε_{-6}
1	ε_{0}
2	ε_{7}
3	ε_{20}

TAB. 2.2: *Précision donnée par l'erreur ε, de l'approximation de $\exp(x)$ sur l'intervalle $[-2^k, 2^k]$ par le polynôme $P(x)$ de Chebychev de degré 3 telle que $\exp(x) = P(x).(1+\varepsilon)$.*

2.2.1 Réduction d'argument

La réduction d'argument est la première étape dans l'évaluation logicielle des fonctions élémentaires. Les performances globales de l'algorithme d'évaluation, et la précision du résultat dépendent donc de la rapidité et de la précision de cette étape. En effet, les algorithmes utilisés pour évaluer les fonctions élémentaires et basés sur une évaluation polynomiale sont corrects si l'argument appartient à un petit intervalle, habituellement centré sur 0. Pour évaluer une fonction élémentaire $f(x)$ pour tout x, il est donc nécessaire de trouver une transformation permettant de déduire $f(x)$ à partir de $g(x^*)$, où :

- x^* est appelé l'*argument réduit* et est déduit de x,
- x^* appartient au domaine de convergence de l'algorithme implémenté pour l'évaluation de g.

On peut classer les réductions d'argument en deux classes : les réductions multiplicatives (par exemple le cas du logarithme), et les réductions additives (par exemple le cas des fonctions trigonométriques). En pratique, seule la réduction d'argument additive nécessite une attention particulière. Dans ce cas, x^* est égal à $x - kC$, où k est un entier et C un multiple entier de $\pi/4$ pour les fonctions trigonométriques, ou de $\ln(2)$ pour l'exponentielle.

2.2.1.1 Problème de précision

Une mauvaise réduction d'argument peut conduire à de catastrophiques problèmes de précision comme le montre K. C. Ng [69], lorsque l'argument d'entrée est proche d'un multiple de C. Il est facile de comprendre pourquoi une mauvaise réduction d'argument peut donner des résultats inexacts. La méthode naïve consiste à effectuer le calcul suivant en utilisant la précision machine :

$$\begin{array}{rcl} k & = & \lfloor \frac{x}{C} \rfloor \\ x^* & = & x - kC. \end{array} \tag{2.3}$$

Lorsque x est proche de kC, presque toute la précision, sinon toute, est perdue lors de la soustraction $x - kC$. C'est ce que l'on appelle *annulation* ou *cancellation*. Donnons un exemple :

si $C = \pi/2$ et $x = 584664.53$ la valeur correcte de x^* est $-0.0000000016757474\ldots$, et la valeur de k correspondante est 372209. Si cette soustraction est effectuée avec une arithmétique

en base 10 avec 8 chiffres (en arrondi au plus près, et en remplaçant $\pi/2$ par sa plus proche valeur représentable dans cette arithmétique), on obtient -0.027244229, donc tous les chiffres du résultat calculé sont faux.

Une première solution pour résoudre ce problème est d'utiliser de l'arithmétique multiprécision, mais le calcul s'en trouvera fortement ralenti et la précision nécessaire pour effectuer les calculs intermédiaires va dépendre de la précision et de l'intervalle donné.

Il faut donc analyser l'erreur commise lors de la réduction d'argument. Cela nécessite la connaissance du plus petit argument réduit possible atteignable sur un intervalle donné. Il existe un algorithme, que l'on doit à Kahan, pour déterminer cet argument. Une version écrite en langage C est disponible à l'adresse `http ://http.cs.berkeley.edu/~wkahan` et une version Maple du même programme est donnée dans [67], dont quelques résultats se trouvent dans le tableau 2.3. L'algorithme en question est basé sur la théorie des fractions continues.

Intervalle	C	Pires cas	Nombre de bits perdus
$]-2^{1024}, 2^{1024}[$	$\pi/2$	$6381956970095103 \times 2^{797}$	60.9
$]-2^{1024}, 2^{1024}[$	$\pi/4$	$6381956970095103 \times 2^{796}$	61.9
$]-2^{1024}, 2^{1024}[$	$\ln 2$	$5261692873635770 \times 2^{499}$	66.8
$]-1024, 1024[$	$\ln 2$	$7804143460206699 \times 2^{-51}$	57.5

TAB. 2.3: *Nombre le plus proche d'un multiple de C pour la réduction additive pour la double précision, et nombre maximum de bits annulés.*

2.2.1.2 Méthode de Cody et Waite

Dans les années 1980, Cody et Waite [17, 18] ont suggéré une méthode relativement efficace pour des arguments d'amplitude moyenne. La constante C est représentée comme la somme de plusieurs nombres flottants $C_1, C_2, ...C_i$. Ces nombres devront être choisis de façon à ce que kC_j soit exactement représentable par un nombre flottant pour $0 < j \leq i-1$ et pour toute valeur de k considérée. Ainsi au lieu d'évaluer $x - kC$, il suffit d'évaluer :

$$((x - kC_1) - kC_2)...$$

Cette méthode présente l'avantage d'évaluer rapidement une approximation de l'argument réduit. En revanche, il n'est pas possible d'utiliser un tel algorithme pour de grands arguments, le nombre de bits nécessaire pour stoker kC_i pouvant vite devenir supérieur au nombre de bits du format. Par exemple prennons $C = \pi$ et un flottant double précision $x = 2^{1000}$, alors $k = \lfloor \frac{2^{1000}}{\pi} \rfloor \approx 2^{1000}$, ce qui signifie qu'au moins 1000 bits seront nécessaires pour représenter k, et encore plus pour représenter les kC_i ! On est donc loin des 53 bits de précision du format double précision. De plus, la précision du résultat en sortie est limitée à la précision d'un nombre flottant, par conséquent il est difficile d'obtenir par cette méthode le résultat comme la somme de deux nombres flottants (nous verrons au chapitre 5 que, dans certain cas, l'argument réduit doit être sur 2 flottants).

2.2.1.3 Méthode de Payne et Hanek

Lorsque la méthode de Cody et Waite ne suffit pas, c'est-à-dire lorsque l'on désire avoir l'argument réduit avec une grande précision, ou que l'argument en entrée est grand, nous devons nous orienter vers d'autres méthodes. L'une des plus connues est celle proposée en 1983 par Payne et Hanek [72].

Nous supposons pour la suite que nous souhaitons effectuer une réduction d'argument pour les fonctions trigonométriques, avec $C = \pi/4$, et que le domaine de convergence des algorithmes contient $I = [0, \pi/4]$. L'extension de la méthode aux autres domaines de convergence ou fonctions (exponentielle) est simple.

Pour un argument en double précision x, nous voulons trouver l'argument réduit x^* avec p bits de précision et l'entier k, satisfaisant $x = kC + x^*$. Une fois x^* connu, il suffit de disposer de $(k \mod 8)$ pour évaluer $\sin(x)$ ou $\cos(x)$ à partir de x^*. Nous allons donc effectuer la suite d'opérations suivantes :

$$\begin{array}{rcl} y & = & x \times \frac{1}{C} \\ k & = & \lfloor y \rfloor \\ w & = & y - k \\ x^* & = & C \times w. \end{array} \tag{2.4}$$

Ces opérations ne peuvent pas être réalisées en précision machine. Représentons le flottant x par $x = X \times 2^e$, où X est un entier sur 53 bits, et $e \geq -1$ (dans le cas où $e < -1$, aucune réduction n'est nécessaire).

Si l'on se place dans le pire cas, alors l'argument x est dans l'intervalle $]-2^{1024}, 2^{1024}[$. De plus d'après le tableau 2.3, sur l'intervalle $]-2^{1024}, 2^{1024}[$, pour $C = \pi/4$ nous pouvons perdre jusqu'à 62 bits de précision. Il faut disposer de la valeur $\frac{1}{C}$ avec au minimum $1023 + 62 + p$ bits de précision pour effectuer l'opération $x \times \frac{1}{C}$, et garantir les p bits de précision du résultat.

En revanche cette multiplication ne doit pas être faite telle quelle. L'idée de Payne et Hanek [72] est de remarquer que seule une petite partie de $\frac{1}{C}$ est nécessaire pour obtenir le résultat qui nous intéresse dans l'opération $y = x \times \frac{1}{C}$.

En effet, si $x = X \times 2^e$ alors nous pouvons découper $\frac{1}{C}$ en trois parties g, m, d où :

- $g = G \times 2^{-e+3}$ contient les $(e-3)$ premiers bits de la partie fractionnaire de $\frac{4}{\pi}$, avec G entier,
- $m = M \times 2^{-e-p-115}$ contient les $p + 53 + 3 + 62$ bits suivants de $\frac{4}{\pi}$, avec M entier,
- $d = D \times 2^{-e-p-115-1}$ contient les bits suivants de $\frac{4}{\pi}$, où $0 < D < 1$.

On remarque en effet que $\frac{4}{\pi} \times x = (8 \times g + (m + d)2^{-e-p-115-1})x$. La valeur $8 \times g \times x$ est un multiple de 8, qui une fois multipliée par $\pi/4$ (voir Eq. 2.4), n'aura pas d'influence sur l'évaluation trigonométrique. De plus, en faisant varier le paramètre de précision p on peut rendre la quantité $D \times 2^{-e-p-115-1} \times x$ aussi petite que l'on souhaite. Par conséquent, seule la partie m de $\frac{4}{\pi}$ est pertinente pour la réduction d'argument.

Les différentes étapes de cet algorithme sont décrites dans le schéma 2.2.

2.2.1.4 Réduction en grande base

La méthode de Cody et Waite exposée en section 2.2.1.2 est efficace pour les petits arguments. Cependant lorsque l'on considère de grand arguments, elle se révèle trop imprécise et la solution est alors d'utiliser celle de Payne et Hanek, présentée en section 2.2.1.3. Cette méthode est certes précise, mais le nombre d'opérations nécessaires est élevé. Lorsque l'on consi-

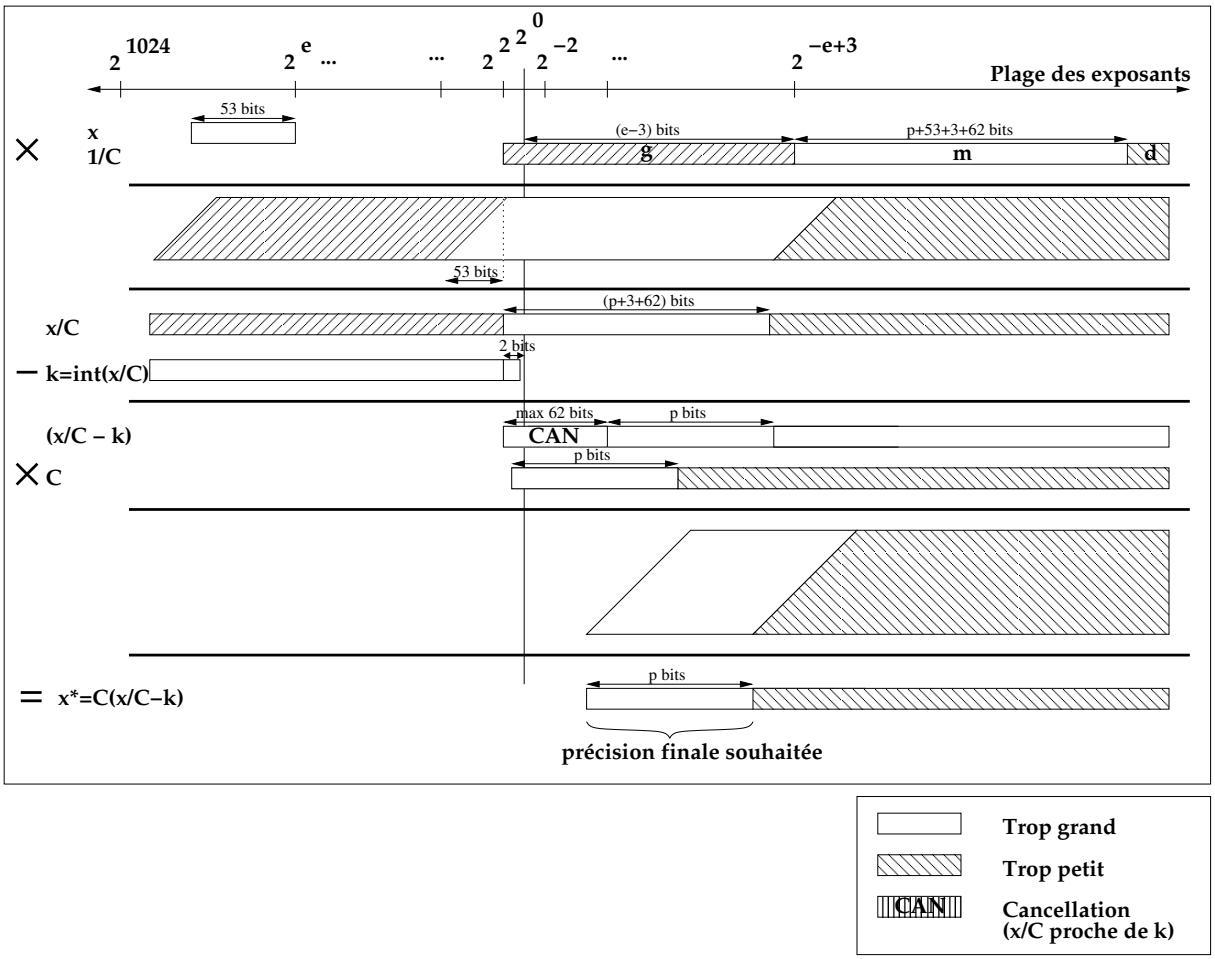


FIG. 2.2: *Algorithme de Payne et Hanek.*

dère des arguments de taille moyenne (jusqu'à $2^{63}-1$), une solution est de se tourner vers la lourde procédure de Payne et Hanek, ou d'utiliser celle que nous avons développée et décrite dans [33]. Nous avons appelé cette méthode *Réduction en grande base*. Nous supposons donc que l'argument à réduire x est compris entre 8 et $2^{63}-1$. Le graphique 2.3 résume les étapes de l'algorithme ci-dessous :

1. Définissons $I(x)$ comme l'entier le plus proche de x, et $F(X) = x - I(x)$. Notons que $F(x)$ est exactement représentable en double précision et calculé exactement en machine, et que pour $x \geq 2^{52}$, nous avons $F(x) = 0$ et $I(x) = x$. Par hypothèse $x \geq 8$, donc le dernier bit de mantisse de $F(x)$ a un poids supérieur ou égal à 2^{-49} ;

2. $I(x)$ est découpée en parties de 8 bits

$$\begin{cases} I_0(x) \text{ contenant les bits } 0 \text{ à } 7 \quad \text{de } I(x) \\ I_1(x) \text{ contenant les bits } 8 \text{ à } 15 \quad \text{de } I(x) \\ I_2(x) \text{ contenant les bits } 16 \text{ à } 23 \text{ de } I(x) \\ I_3(x) \text{ contenant les bits } 24 \text{ à } 31 \text{ de } I(x) \\ I_4(x) \text{ contenant les bits } 32 \text{ à } 39 \text{ de } I(x) \\ I_5(x) \text{ contenant les bits } 40 \text{ à } 47 \text{ de } I(x) \\ I_6(x) \text{ contenant les bits } 48 \text{ à } 55 \text{ de } I(x) \\ I_7(x) \text{ contenant les bits } 56 \text{ à } 63 \text{ de } I(x) \end{cases}$$

telle que :

$$I(x) = I_0(x) + 2^8 I_1(x) + 2^{16} I_2(x) + \ldots + 2^{56} I_7(x).$$

Notons au passage que $I(x)$ est représentable sur au plus 53 bits, par conséquent soit $I_7(x) = 0$, soit $I_0(x) = 0$. Il y a donc au plus 7 des $I_j(x) = 0$ qui sont non nuls.
Comme mentionné ci-dessus, notre but est de toujours fournir un résultat correct. L'utilisation de l'algorithme de recherche de pires cas [67] nous permet d'obtenir le nombre maximum de bits perdus sur l'intervalle considéré. Nous perdons au maximum 61 bits de précision dans notre cas : intervalle de départ $= [8, 2^{63} - 1]$, $C = \pi/2$ et double précision. Nous avons besoin de mémoriser $(I_i(x) \mod \pi/2)$ [1] avec au moins

	61	(possible annulation sur les bits de poids fort)
$+$	53	(bits de mantisse différents de zéro)
$+$	p	(bits de garde)
$=$	$114 + p$	bits.

Pour obtenir cette précision, tous les nombres $(I_i(x) \mod \pi/2)$, appartenant à $[-\frac{\pi}{4}, +\frac{\pi}{4}]$ sont tabulés comme la somme de trois nombres double précision.

$$\begin{cases} T_{hi}(i,w) \quad \text{contenant les bits de poids } 2^{-1} \text{ à } 2^{-49} \text{ de } \quad ((2^{8i}w) \mod \pi/2) \\ T_{med}(i,w) \text{ contenant les bits de poids } 2^{-50} \text{ à } 2^{-98} \text{ de } \quad ((2^{8i}w) \mod \pi/2) \\ T_{lo}(i,w) \quad \text{contenant les bits de poids } 2^{-99} \text{ à } 2^{-147} \text{ de } ((2^{8i}w) \mod \pi/2) \end{cases}$$

en arrondi au plus près, où w est un entier sur 8 bits. Notons que $T_{hi}(i,0) = T_{med}(i,0) = T_{lo}(i,0) = 0$. Les trois tables T_{hi}, T_{med} et T_{lo} sont adressées par 11 bits. La plus grande valeur possible de

$$\left| \left(\sum_{i=0}^{7} (I_i(x) \mod \pi/2) \right) + F(x) \right|$$

est bornée par $2\pi + \frac{1}{2}$, qui est inférieure à 8. Nous en déduisons que

$$S_{hi}(x) = \left(\sum_{i=0}^{7} T_{hi}(i, I_i(x)) \right) + F(x)$$

est un multiple de 2^{-49} et a une valeur absolue inférieure à 8. *S_{hi} est donc exactement représentable en double précision* (il est même représentable sur 52 bits). Donc, avec une arithmétique de type IEEE, il sera *calculé exactement*.

[1] défini tel que $-\pi/4 \leq (X \mod \pi/2) < \pi/4$.

3. Nous calculons
$$\begin{cases} S_{hi}(x) &= \left(\sum_{i=0}^{7} T_{hi}(i, I_i(x))\right) + F(x) \\ S_{med}(x) &= \sum_{i=0}^{7} T_{med}(i, I_i(x)) \\ S_{lo}(x) &= \sum_{i=0}^{7} T_{lo}(i, I_i(x)) \end{cases}$$
Ces trois sommes sont calculées exactement en arithmétique double précision, sans erreur d'arrondi.
Le nombre $S(x) = S_{hi}(x) + S_{med}(x) + S_{lo}(x)$ est égal à x moins un entier multiple de $\pi/2$ plus une petite erreur bornée par $8 \times 2^{-148} = 2^{-145}$, due à l'arrondi au plus proche des valeurs $(2^{8i}w) \mod \pi/2$ aux bits de poids 2^{-147}.
$S(x)$ n'est pas encore l'argument réduit car sa valeur absolue peut être supérieure à $\pi/4$. Nous avons donc besoin de soustraire ou d'additionner un multiple de $\pi/2$ à $S(x)$ pour obtenir le résultat final. Définissons $C_{hi}(k)$, pour $k = 1, 2, 3, 4$, comme le multiple de 2^{-49} le plus proche de $k\pi/2$. $C_{hi}(k)$ est exactement représentable en double précision. Définissons $C_{med}(k)$ comme le multiple de 2^{-98} le plus proche de $k\pi/2 - C_{hi}(k)$ et $C_{lo}(k)$ comme le nombre flottant double précision le plus proche de $k\pi/2 - C_{hi}(k) - C_{med}(k)$.

4. Nous procédons comme suit :
   - Si $|S_{hi}(x)| \leq \pi/4$ alors définissons
$$\begin{aligned} R_{hi}(x) &= S_{hi}(x) \\ R_{med}(x) &= S_{med}(x) \\ R_{lo}(x) &= S_{lo}(x) \end{aligned}$$
   - Sinon, soit k_x tel que $C_{hi}(k_x)$ est le plus proche de $|S_{hi}(x)|$. Nous calculons successivement :
     - Si $S_{hi}(x) > 0$
$$\begin{aligned} R_{hi}(x) &= S_{hi}(x) - C_{hi}(k_x) \\ R_{med}(x) &= S_{med}(x) - C_{med}(k_x) \\ R_{lo}(x) &= S_{lo}(x) - C_{lo}(k_x) \end{aligned}$$
     - Sinon,
$$\begin{aligned} R_{hi}(x) &= S_{hi}(x) + C_{hi}(k_x) \\ R_{med}(x) &= S_{med}(x) + C_{med}(k_x) \\ R_{lo}(x) &= S_{lo}(x) + C_{lo}(k_x) \end{aligned}$$
     $R_{hi}(x)$ et $R_{med}(x)$ sont exactement représentables en double précision. Ils sont donc calculés exactement.

   Le nombre $R(x) = R_{hi}(x) + R_{med}(x) + R_{lo}(x)$ est égal à x moins un entier multiple de $\pi/2$ plus une erreur bornée par $2^{-145} + 2^{-148}$.
   Cette étape est également utilisée (seule, sans l'étape précédente) pour réduire les petits arguments inférieurs à 8. L'argument réduit est représenté sous la forme d'une somme de trois nombres flottants double précision. Il nous faut donc maintenant réduire cet argument sous la forme de deux nombres flottants. Pour cela nous utiliserons l'algorithme Fast2Sum, décrit en section 3.2.2, qui additionne de façon exacte deux nombres flottants et renvoie le résultat sous la forme de deux nombres flottants.

5. Nous obtenons l'argument réduit final de la façon suivante. Soit p un paramètre entier utilisé pour définir la précision finale requise, alors :

– Si $R_{hi}(x) > 1/2^p$, alors nous calculons

$$(y_{hi}, y_{lo}) = \text{Fast2Sum}(R_{hi(x)}, R_{med(x)}).$$

Les 2 nombres flottants y_{hi} et y_{lo} contiennent l'argument réduit avec une erreur relative inférieure à $2^{-96+p} + 2^{-99+p}$;

– Si $R_{hi}(x) = 0$, alors nous calculons

$$(y_{hi}, y_{lo}) = \text{Fast2Sum}(R_{med(x)}, R_{lo(x)}).$$

Le plus petit argument réduit est supérieur à 2^{-62}, donc les 2 nombres flottants y_{hi} et y_{lo} contiennent l'argument réduit avec une erreur relative inférieure à

$$\frac{2^{-145} + 2^{-148}}{2^{-62}} \leq 2^{-82}.$$

– Si $0 < R_{hi(x)} \leq 2^{-p}$, alors nous calculons successivement

$$\begin{aligned} (y_{hi}, temp) &= \text{Fast2Sum}(R_{hi}(x), R_{med}(x)) \\ y_{lo} &= R_{lo}(x) + temp \end{aligned}$$

Les 2 nombres flottants y_{hi} et y_{lo} contiennent l'argument réduit avec une erreur absolue inférieure à 2^{-98} pour $0 < |y| \leq 2^{-p}$, et une erreur relative inférieure à 2^{-95+p} sinon.

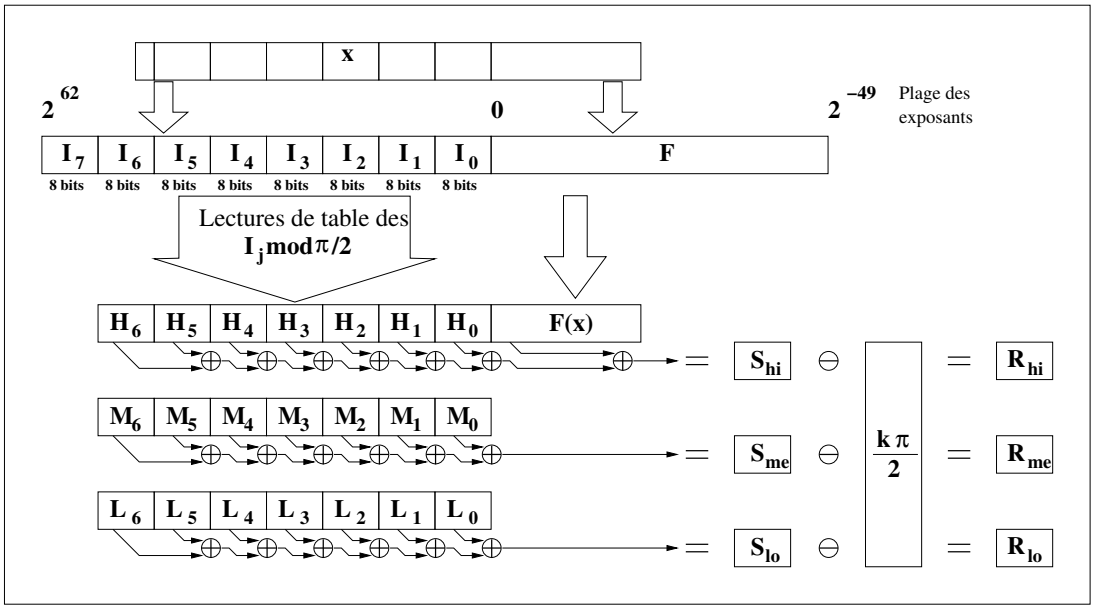


FIG. 2.3: *Réduction d'argument en grande base.*

Comparaisons

Si l'on compte le nombre N d'opérations flottantes double précision, on a pour $|x| < 2^{63}$, $N = 10 + 3\log_{256} x$, ce qui est équivalent à un nombre d'opérations compris entre 13 et 34 et $3\lceil \log_{256} x \rceil$ lectures de table.

Une variante de l'algorithme consiste à calculer dans une première étape S_{hi}, S_{med} et R_{hi}, R_{med}. Si au cours de la cinquième étape de l'algorithme on détecte que la précision n'est pas suffisante alors on calcule T_{lo} et R_{lo}. Cette petite modification permet de réduire le nombre d'opérations dans les cas fréquents (faible nombre de bits cancellés).

Dans cette section, nous avons comparé notre méthode avec d'autre algorithmes sur le même intervalle $[8, 2^{63} - 1]$, notamment la méthode de Payne et Hanek (voir section 2.2.1.3) et la réduction modulaire décrite dans [26]. Nous avons choisi l'implémentation de Sun Microsystems [69] pour la réduction de Payne et Hanek. La comparaison est composée des critères suivants : taille de table, nombre de lectures de table, nombre de multiplications/additions/-divisions flottantes.

	# Opérations élémentaires	# De lecture de table	Taille de table en Koctets
Notre algorithme	13 à 34	3/24	48
Payne & Hanek	55 à 103	2	0.14
Réduction d'argument modulaire	150	53	2

Les comparaisons montrent que l'algorithme présenté nécessite moins d'opérations pour des arguments de taille moyenne. En revanche, la méthode de Payne et Hanek et la réduction d'argument modulaire n'ont pas besoin de beaucoup de mémoire, mais nécessitent entre trois et quinze fois plus d'opérations flottantes. Cet algorithme constitue donc un excellent compromis entre taille de table et nombre d'opérations pour la réduction d'arguments de taille moyenne.

2.2.1.5 Réduction par table

Nous venons de voir les méthodes de réduction d'argument. Malheureusement, comme nous l'avons observé à travers l'exemple de l'exponentielle (tableau 2.2), il est souvent nécessaire d'effectuer une deuxième réduction d'argument pour obtenir un intervalle plus petit et où l'évaluation polynomiale sera assez précise. La solution la plus couramment utilisée est la mémorisation de quelques valeurs précalculées dans des tables [65, 67, 80, 81].

Supposons que x^* soit l'argument à réduire par la réduction par table. Cette réduction consiste à trouver un bon compromis entre méthodes à base de tables et approximations polynomiales. Pour ceci, il faut découper x^* en une partie X_1 constituée des k bits de poids fort, et une partie X_2 constituée des ℓ bits de poids faible (voir figure 2.4).

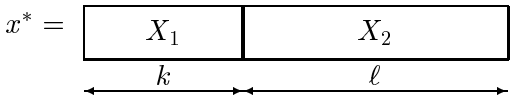


FIG. 2.4: *Découpage de x^* en deux mots de k et ℓ bits. Le mot X_1 servira à adresser une table et le mot X_2 sera l'argument d'une approximation polynomiale.*

Nous avons alors :

$$x^* = X_1 + X_2 2^{-k}, \text{ avec } 0 \leq X_1, X_2 < 1.$$

La partie X_1 est utilisée pour adresser une table et l'argument réduit X_2 est utilisé pour l'évaluation polynomiale. La précision en sortie utile pour mémoriser ces valeurs est limitée à 53 bits lorsque l'on considère la double précision. Cependant plusieurs solutions permettent d'augmenter cette précision :

- On utilise deux nombres flottants double précision. Malheureusement cette technique nécessite deux fois plus d'espace mémoire et au moins deux fois plus d'opérations pour l'évaluation de l'argument réduit.
- Une autre solution a été proposée par Gal [37]. Elle consiste à perturber localement les points à tabuler de façon à obtenir une valeur à tabuler pratiquement égale à un nombre machine. Ainsi, un nombre flottant est utilisé pour mémoriser la perturbation, et un autre pour le résultat proche d'un nombre machine minimisant l'erreur. Bien que le nombre de valeurs à mémoriser soit identique à la solution précédente, l'évaluation de la fonction s'en trouve souvent simplifiée.

Dans les deux cas, la façon de placer les mots en mémoire a également une importance. Nous verrons en section 3.1.3.1 comment les arranger de façon optimale.

2.2.2 Approximation polynomiale

Nous supposerons que dans le processus d'évaluation des fonctions élémentaires, la réduction d'argument a été effectuée. On se ramène désormais au problème d'évaluer une fonction f sur un intervalle d'assez petite taille. On supposera que cet intervalle est de la forme $[0, a]$. Dans cet intervalle, on désire approcher la fonction f par un polynôme.

2.2.2.1 Approximations de Taylor

La première idée qui vienne à l'esprit est d'utiliser un développement de Taylor de la fonction f. Ce développement peut être choisi soit en l'origine, soit en un point bien précis de l'intervalle.

Le développement de Taylor est basé sur le théorème suivant :

Théorème 2 (Taylor-Lagrange) *Soit f une fonction définie et continue sur $[a, b]$ ainsi que ses n premières dérivées successives, et telle que $f^{(n+1)}$ existe sur $]a, b[$. Alors il existe un point c, $c \in]a, b[$ tel que*

$$f(b) = f(a) + \frac{(b-a)}{1!} f'(a) + \ldots + \frac{(b-a)^n}{n!} f^{(n)}(a) + \frac{(b-a)^{(n+1)}}{(n+1)!} f^{(n+1)}(c).$$

Ce développement permet d'obtenir des valeurs approchées d'une fonction avec une précision aussi grande qu'on le désire car le reste d'ordre $n+1$ tend généralement vers 0 lorsque n augmente. Aussi une majoration de l'erreur commise en approchant le calcul de $f(x)$ par l'évaluation polynomiale de Taylor P_n de degré n est donnée par :

$$|f(x) - P_n(x)| \leq M \frac{|x|^{n+1}}{(n+1)!}$$

où M est une majoration de $|f^{(n+1)}|$ sur l'intervalle $]a, b[$.

fonction	développement de Taylor en 0
$\exp(x)$	$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!} + \ldots$
$\ln(1+x)$	$x - \frac{x^2}{2} + \frac{x^3}{3} + \ldots + (-1)^{(n+1)}\frac{x^n}{n} + \ldots$
$\sin(x)$	$x - \frac{x^3}{3!} + \frac{x^5}{5!} + \ldots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + \ldots$
$\cos(x)$	$1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \ldots + (-1)^n \frac{x^{2n}}{(2n)!} + \ldots$
$\sinh(x)$	$x + \frac{x^3}{3!} + \frac{x^5}{5!} + \ldots + \frac{x^{2n+1}}{(2n+1)!} + \ldots$
$\cosh(x)$	$1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \ldots + \frac{x^{2n}}{(2n)!} + \ldots$

TAB. 2.4: *Exemples de développement de Taylor en* 0.

L'intérêt de l'approximation par un polynôme de Taylor vient de la simplicité de ses coefficients (voir Tab. 2.4). Cette simplicité permet d'économiser des opérations (multiplications par 1 par exemple), où de les simplifier (multiplications par $1/2$).

2.2.2.2 Approximation de Chebychev

De bonnes approximations sont également obtenues par les polynômes de Chebychev. Le polynôme de Chebychev de degré n, T_n est donné par :

$$T_n(x) = \cos(n \arccos x)$$

ce qui donne :

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_2(x) &= 2x^2 - 1 \\ T_3(x) &= 4x^3 - 3x \\ &\ldots \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x) \quad n \geq 1. \end{aligned}$$

Ces polynômes sont orthogonaux pour le produit scalaire $< f, g >= \int_{-1}^{1} \frac{f(x)g(x)dx}{\sqrt{1-x^2}}$. En particulier nous avons :

$$\int_{-1}^{1} \frac{T_i(x)T_j(x)}{\sqrt{1-x^2}}dx = \begin{cases} 0 & \text{si} \quad i \neq j \\ \pi & \text{si} \quad i = j = 0 \\ \pi/2 & \text{sinon.} \end{cases}$$

Le polynôme T_n a la propriété d'avoir n zéros dans l'intervalle $[-1, 1]$, aux points

$$x = \cos\left(\frac{\pi\left(k - \frac{1}{2}\right)}{n}\right) \quad \text{pour} \quad k = 1, 2, \ldots, n.$$

Plus de détails sur les polynômes de Chebychev sont donnés dans [10]. Une implémentation en langage C et les détails de calcul de l'approximation de Chebychev sont également disponibles dans [36].

2.2.2.3 Approximation Minimax

L'approximation de Taylor n'est pas la meilleure solution, car ce n'est une meilleure approximation que localement (au voisinage du point où l'on développe) et non globalement (sur un intervalle).

On préfère calculer la meilleure approximation polynomiale *minimax* P_n^* de la fonction, définie par :

$$\max_{a \leq x \leq b} |f(x) - P_n^*(x)| = \min_{P_n} \left(\max_{a \leq x \leq b} |f(x) - P_n(x)| \right)$$

où P_n^* et P_n sont des polynômes de degré inférieur ou égal à n. L'existence d'un tel polynôme a été montrée par Chebychev, qui énonce que si $f(x)$ est continue sur l'intervalle $[a, b]$ alors il existe un unique polynôme minimax pour un n donné, et que l'erreur maximum est atteinte en au moins $n + 2$ points distincts de l'intervalle $[a, b]$, avec des signes alternés.

Le polynôme P_n^* minimise l'erreur maximale sur l'intervalle $[a, b]$ sur lequel on cherche à approcher la fonction. Cette approximation, appelée *minimax*, se calcule à l'aide d'un algorithme dû à Remez [74].

2.2.2.4 Construction d'une bonne approximation

Nous allons maintenant décrire en détails le processus pour obtenir les coefficients d'une approximation polynomiale ayant de bonnes propriétés.

Nous utiliserons le logiciel Maple et sa bibliothèque d'approximation *numapprox*. Cette dernière offre une fonction *minimax* proposant une approximation du polynôme minimax. La commande

```
> minimax(f(x),x=a..b,[p,q],w(x),'err');
```

donne la meilleure approximation rationnelle $Q^*_{\frac{p}{q}}$ dont le numérateur est de degré p, le dénominateur de degré q, de la fonction f sur l'intervalle $[a, b]$, avec une fonction de poids $w(x)$, où l'erreur maximum est obtenue en err, vérifiant :

$$\max_{x \in [a,b]} \left| w(x) \left(Q^*_{\frac{p}{q}}(x) - f(x) \right) \right| = \min_{Q_{\frac{p}{q}}} (\max_{x \in [a,b]} \left| w(x) \left(Q_{\frac{p}{q}}(x) - f(x) \right) \right|).$$

Nous ne considérons que les approximations polynomiales de la forme

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$$

d'une fonction $f(x)$ sur l'intervalle $I = [a, b]$. En pratique, plutôt qu'une approximation minimax standard, nous cherchons un polynôme satisfaisant certaines propriétés :

Pour obtenir le polynôme P_n^* nous devons préalablement étudier la fonction et les propriétés souhaitées pour le polynôme.

1. **Intervalle :**
   Il faut rechercher un intervalle où la fonction f s'approche facilement par un polynôme.
   Si la fonction est paire/impaire, il faut en préserver la parité.
   mise en pratique :
   Si l'on considère la fonction $\sin$ sur un intervalle $[a, b]$ avec $a.b \leq 0$, alors on sait que $\sin(x) = -\sin(x)$ et il suffit d'évaluer cette fonction sur un intervalle de la forme $[a, 0]$ ou $[0, b]$.

2. **Fonction :**
On souhaite simplifier l'évaluation du polynôme P_n^*, ce qui implique des modifications sur la fonction à approcher.
   - Si la fonction est paire/impaire, on souhaite rendre les coefficients de P_n^* de degré impair/pair nuls. Cela permet de simplifier l'évaluation polynomiale (il y a deux fois moins de coefficients).
   mise en pratique :
   Nous cherchons le polynôme $Q_{n'}^*(X)$ tel que $X = x^2$ pour $X \in [a^2, b^2]$ de degré n' vérifiant :
   - Si $f(x)$ est paire
   $P_n^*(x) = Q_{n'}^*(X)$ avec $n' = \lfloor n/2 \rfloor$ vérifiant

$$\max_{a^2 \leq X \leq b^2} w(\sqrt{X}) \left| f(\sqrt{X}) - Q_{n'}^*(X) \right| = \min_{Q_{n'}} \left(\max_{a^2 \leq X \leq b^2} w(\sqrt{X}) \left| f(\sqrt{X}) - Q_{n'}(X) \right| \right)$$

   Le code Maple correspondant est
   ```
   > Q := minimax(f(x), x=a^2..b^2, [floor(n/2),0], 1, 'err');
   > P := subs(x=x^2, Q);
   ```
   - Si $f(x)$ est impaire
   $P_n^*(x) = x.Q_{n'}^*(X)$ avec $n' = \lfloor (n-1)/2 \rfloor$ vérifiant

$$\max_{a^2 \leq X \leq b^2} \left(w(\sqrt{X}) \left| \frac{f(\sqrt{X})}{\sqrt{X}} - Q_{n'}^*(X) \right| \right) = \min_{Q_{n'}} \left(\max_{a^2 \leq X \leq b^2} \left(w(\sqrt{X} \left| \frac{f(\sqrt{X})}{\sqrt{X}} - Q_{n'}(X) \right| \right) \right)$$

   Le code Maple correspondant est
   ```
   > Q := minimax(f(x), x=a^2..b^2, [floor((n-1)/2),0], 1, 'err');
   > P := x * subs(x=x^2, Q);
   ```
   - On aimerait rendre les premiers coefficients du polynôme égaux à ceux du développement de Taylor et/ou exactement représentables. Ce qui permet de simplifier les calculs (multiplications par 1 ou $1/2$), mais également d'éviter l'utilisation d'une procédure spéciale pour les petits arguments. Prenons par exemple le polynôme $P(x) = 1.01 + 0.999x + .501x^2$ approchant la fonction exponentielle sur l'intervalle $[0, 2^{-5}]$. Aussi nous avons $P(0) = 1.01$ ce qui n'est pas une bonne approximation puisque $\exp(0) = 1$.
   mise en pratique :
   Soient a_0 et a_1 les coefficients de l'approximation polynomiale P_n^* que l'on souhaite fixer. On cherche donc une approximation de la forme

$$P_n^*(x) = a_0 + a_1 x + x^2 R_{(n-2)}^*(x)$$

   où $R_{(n-2)}^*$ est un polynôme de degré $(n-2)$.
   Donc, on cherche la meilleure approximation polynomiale $R_{(n-2)}^*$ qui vérifie :

$$\max_{a \leq x \leq b} w(x) \left| f(x) - \left(a_0 + a_1 x + x^2 R_{(n-2)}^*(x) \right) \right| = \min_{R_{(n-2)}} \left(\max_{a \leq x \leq b} w(x) \left| f(x) - \left(a_0 + a_1 x + x^2 R_{(n-2)}(x) \right) \right| \right)$$

équivalent à

$$\max_{a\leq x\leq b} \frac{w(x)}{x^2}\left|g(x) - R^*_{(n-2)}(x)\right| = \min_{R_{(n-2)}}\left(\max_{a\leq x\leq b} \frac{w(x)}{x^2}\left|g(x) - R_{(n-2)}(x)\right|\right)$$

avec $g(x) = \frac{f(x)-a_0-a_1.x}{x^2}$
Le code Maple correspondant est

```
> R := minimax((f(x) - a0 - a1*x)/x^2, x=a..b, [n,0], 1, 'err');
> P := a0 + a1 * x + R;
```

3. **Recherche de l'erreur à minimiser :**
   À chaque étape, nous avons cherché à minimiser l'erreur absolue. Avec la virgule flottante il est plus intéressant de minimiser l'erreur relative que l'erreur absolue.
   mise en pratique :
   Pour minimiser l'erreur relative il faut chercher la meilleure approximation polynomiale P_n de la fonction f avec une fonction de poids $w(x) = 1/f(x)$ telle que :

$$\max_{a\leq x\leq b} \frac{1}{f(x)}|f(x) - P^*_n(x)| = \min_{P_n}\left(\max_{a\leq x\leq b} \frac{1}{f(x)}|f(x) - P_n(x)|\right)$$

   Le code Maple correspondant est

```
> P := minimax(f(x), x=a..b, [n,0], 1/f(x), 'err');
```

4. **Recherche du degré du polynôme**
   La fonction `minimax` de Maple est programmée en utilisant l'algorithme de Remez. C'est un algorithme itératif lent. Par conséquent pour la recherche de la meilleure approximation polynomiale sur un intervalle pour une précision donnée, on pourra préférer dans un premier temps utiliser la fonction `chebpade`.

```
> chebpade(f(x), x=a..b, [p,q]);
```

   Cette fonction donne l'approximation rationnelle de Chebychev dont le numérateur est de degré p et le dénominateur de degré q de la fonction f sur l'intervalle $[a, b]$. La qualité de cette approximation est bonne. En effet, le polynôme obtenu lorsque $q = 0$ est proche du polynôme de meilleure approximation en terme d'erreur. Il existe une relation entre l'erreur de l'approximation de f par le polynôme de Chebychev P_T de degré n, et le polynôme minimax P_M de degré n donnée par [10, 62] :

$$||f - P_T||_\infty \leq \left(4 + \frac{4}{\pi^2}\ln(n)\right)||f - P_M||_\infty$$

   Ainsi le polynôme de Chebychev représente une excellente alternative au polynôme minimax dans la recherche du bon degré, dû à ses performances en terme de vitesse et de précision.

5. **Calcul de l'erreur**
   Une fois le polynôme obtenu, il faut arrondir les coefficients pour les représenter en machine. C'est de cette étape que provient la plus grande partie de l'erreur mathématique d'approximation. Aussi il est nécessaire de recalculer l'erreur d'approximation en utilisant la fonction `infnorm` de Maple :

```
> infnorm(f(x), x=a..b, 'xmax');
```

Cette fonction calcule la norme infinie de la fonction f sur l'intervalle $[a, b]$. Le point où la fonction f est maximum en valeur absolue est placé dans `xmax`.
Donc le nombre de bits perdus par l'approximation de f par le polynôme P est obtenu en cherchant la norme infinie de la fonction $(f(x) - P_n)/f(x)$. Le code Maple correspondant est :

```
> err_b10 := infnorm(abs((f(x)-P)/f(x)), x=a..b, 'xmax'));
> log(err_b10)/log(2.);
```

Toutefois, il est discutable de vouloir obtenir le meilleur polynôme d'approximation polynomiale, pour ensuite en tronquer les coefficients. Brisebarre et Muller [16] ont travaillé sur ce sujet, et proposé une méthode pour obtenir le meilleur polynôme d'approximation polynomiale avec une contrainte sur le nombre de bits utilisés pour coder chacun des coefficients du polynôme.

CHAPITRE

3

tière, les processeurs peuvent être classés en deux catégories :

- Les processeurs tels que Alpha, Pentium III, Athlon et PowerPC qui ont une unité dédiée à la multiplication entière. Elle peut donc être effectuée en seulement quelques cycles.
- Les processeurs Pentium 4, les Itanium I et II et l'UltraSPARC qui délèguent la multiplication entière au multiplieur flottant. La latence est donc plus élevée, ce qui est dû principalement au coût de transfert des opérandes entiers vers les registres flottants.

Les pipelines entiers et flottants sont «hermétiques» au sein des processeurs. Aussi les conversions, ou transferts d'un registre flottant vers entier ou vice-versa sont des opérations très couteuses. Parfois il n'existe même pas d'instruction assembleur permettant de les effectuer. Par exemple le jeu d'instruction SPARC V8 ne dispose pas d'instruction pour transférer le contenu d'un registre entier vers un registre flottant. Nous verrons par la suite comment effectuer efficacement ces opérations.

Le nombre plus important d'unités entières et leur faible latence par rapport aux unités flottantes, nous invitent à les utiliser pour les comparaisons et les opérations de décalages.

Nous noterons également que l'opérateur de division flottante, qui est parfois utilisé pour la réduction d'argument [80], a une latence bien supérieure à celle des autres opérateurs. Il est en effet assez rare de dédier du matériel spécifique à la division, qui est plutôt effectuée par des itérations de Newton ou de Goldschmidt.

Processeur	Cycle	Entier			Flottant double précision			
	(ns)	Nb. Unités	$a \pm b$ L/T	$a \times b$ L/T	Nb. Unités	$a \pm b$ L/T	$a \times b$ L/T	$a \div b$ L/T
Alpha 21264	1.6	4	1/1	7/1	2	4/1	4/1	15/12
AMD K6-III	2.2	6	1/1	2-3/2-3	3	2/2	2/2	20/17
Athlon XP	0.57	3	1/1	4-8/3-7	3	4/1	4/1	20/17
Pentium III	1.0	3	1/1	4/1	1	3/1	5/2	32/32
Pentium 4	0.4	3	.5/.5	14/3	2	5/1	7/2	38/38
Itanium IA-64	1.25	4	1/1	18/1	4	5/1	5/1	>35/>5
PowerPC 750	2.5	2	1/1	2-5/1	1	3/1	4/2	31/31
Sun UltraSparc IIi	2.27	2	1/1	20/20	2	3/1	3/1	22/22
Sun UltraSparc III	0.95	4	3/1	6-7/5-6	2	4/1	4/1	20/17

TAB. 3.1: *Latences (L) et Débits (T) en cycle d'horloge des ALU et FPU de processeurs en 2003.*

3.1.2 Mémoire

3.1.2.1 Hiérarchie mémoire

Les performances des mémoires constituent le principal goulot d'étranglement des systèmes actuels. La méthode classique pour réduire l'impact de la latence mémoire est d'utiliser un ou plusieurs niveaux de cache très rapide. Les caches fonctionnent en exploitant le principe de *localité* : si une donnée est référencée une fois, alors la probabilité que ce même élément ou un élément proche dans l'espace d'adressage (*localité spatiale*) soit accédé dans un futur proche (*localité temporelle*) est élevée. La localité est donc utilisée en mémorisant les données référencées récemment dans une petite mémoire très rapide proche du cœur du processeur, appelée *cache* .

Ces caches sont *unifiés* dans le cas où le code et les données sont stockées dans la même structure matérielle et *séparés* dans le cas contraire. Nous noterons qu'à la compilation les

constantes entières sont généralement considérées comme faisant partie des instructions, alors que les flottants et les tableaux accédés de façon aléatoire sont considérés comme des données.

La tendance actuelle est d'avoir un cache de niveau 1 (L1) séparé entre données et instructions, et unifié pour les autres niveaux (L2, L3, etc). C'est le cas de tous les processeurs mentionnés dans le tableau 3.2.

3.1.2.2 Organisation du cache

Les caches sont divisés en *lignes*, chaque ligne étant composée d'un groupe d'instructions ou de données consécutives dans l'espace d'adressage. Il existe plusieurs techniques pour effectuer le placement des lignes à l'intérieur des caches (voir figure 3.1). Un cache est dit *à correspondance directe* si l'emplacement de chaque ligne dans le cache est imposé ; typiquement la place dépend d'un *index* constitué d'un groupe de bits consécutifs de l'adresse de la ligne. À l'opposé, un cache est *complètement associatif* si une ligne peut aller n'importe où dans le cache. Un compromis entre ces deux méthodes est le cache *partiellement associatif* où une ligne peut aller n'importe où dans un sous-ensemble de lignes du cache. Le nombre n de sous-ensemble détermine le nombre de *voies* du cache. Ces caches sont également appelés caches partiellement associatifs à n-voies. Ils sont les plus courants pour les caches de données de niveau 1, comme on peut le noter dans le tableau 3.2 (la seule exception est l'UltraSparc IIi avec un cache de données L1 à correspondance directe).

Lorsqu'un accès mémoire est effectué dans le cache et que la donnée ne s'y trouve pas, alors elle doit être chargée depuis un niveau inférieur et placée dans une des n différentes places possibles. La nouvelle donnée prend alors la place d'une autre donnée. Il faut donc utiliser une technique de remplacement pour décider quelle donnée remplacer. Ces techniques sont LRU (least recently used), pseudo-LRU (la plus courante), LRA (least recently allocated), FIFO (first in first out) ou remplacement aléatoire.

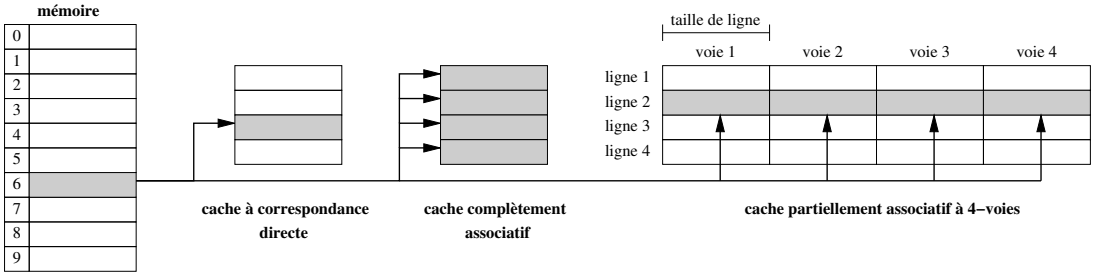


FIG. 3.1: *Le placement des lignes à l'intérieur des différents types de cache.*

3.1.2.3 Accès mémoire

Deux types d'adresses coexistent au sein d'un processeur : les adresses virtuelles utilisées par les instructions mémoires des langages machines et les adresses physiques qui font références aux adresses matérielles ou adresses en mémoire physique. Pour plus de détails sur ces mécanismes nous invitons le lecteur à consulter [49] et [50].

Les adresses virtuelles doivent être converties en adresses physiques pour pouvoir accéder réellement aux données. Cette traduction d'adresse est réalisée par une unité spéciale appelée

Processeur	Cycle (ns)	Cache de donnée L1			Cache L2			Taille de page
		Taille (Ko)	Latence	Nb voies	Taille (Ko)	Latence	Nb voies	
Alpha 21264	1.6	64	3	2	<16MB	12		8Ko
AMD K6-III	2.2	32	2	2	256		4	4Ko,4Mo
Athlon XP	0.57	64	3	2	<8MB	16	1-2	4Ko,2Mo,4Mo
Pentium III	1.0	16	3	4	512	10		4Ko,2Mo,4Mo
Pentium 4	0.4	8	2	4	512	7		4Ko
Itanium IA-64	1.25	16	3	4	96	7	6	4Ko-4Go
Itanium II	1.0	16	2	4	256	6	8	4Ko-4Go
PowerPC 750 (G3)	2.5	32	2	8	<1MB	13-20		4Ko-256Mo
PowerPC 7455 (G4)	1.0	32	2	8	256	9		4Ko-4Go
Sun UltraSparc IIi	2.27	16	2	1	<2MB	6		8Ko-4Mo
Sun UltraSparc III	0.95	64	2	4	<8MB	15	1	8Ko-4Mo

TAB. 3.2: *Principales propriétés des caches de processeurs actuels. Cette table donne la taille des caches de données, la latence d'un chargement entier et le nombre de voies tels qu'ils sont sonnés dans les documents constructeurs.*

MMU (Memory Management Unit). Cette unité traduit l'adresse virtuelle en adresse physique en lui associant la page ou le segment qui lui correspond. La pagination divise l'espace d'adressage du processus en pages de taille fixe. La traduction se fait donc par l'intermédiaire d'un catalogue de pages. De façon à accélérer la traduction, le matériel maintient un petit cache appelé TLB (Translation Look-aside Buffer), qui fait partie de la MMU. Sa fonction est de traduire rapidement les adresses virtuelles en adresses physiques. Comme pour le cache de niveau 1, le TLB est sujet aux techniques de remplacement dont la plus utilisée est la technique pseudo-LRU.

Par conséquent les performances mémoires du système sont étroitement liées aux performances du TLB dont le nombre d'entrées est relativement faible (32 pour la famille des Pentium et plus généralement entre 30 et 100). De plus, lorsqu'une page ne se trouve plus référencée dans le TLB, le temps requis pour charger la donnée de l'endroit où elle se trouve jusqu'au cache de plus haut niveau peut être relativement long, de 1000 cycles si la page est en mémoire, jusqu'à 10^6 (et plus) cycles dans le cas d'un défaut de page, qui est traité par le système d'exploitation.

3.1.3 Conséquences au niveau logiciel

3.1.3.1 Implémentation efficace des tableaux

Les tableaux unidimensionnels ne posent pas de problèmes particuliers. Les éléments d'un tel tableau sont mémorisés de manière continue dans l'espace d'adressage. En revanche, lorsque l'on considère des tableaux multidimensionnels, il faut tenir compte de la façon dont les éléments de ce tableau vont être accédés de façon à augmenter la probabilité de trouver l'élément recherché en mémoire cache. Le but est de minimiser la différence d'adresse entre deux éléments accédés dans une période de temps très courte.

Par exemple, pour les tableaux bidimensionnels (figure 3.2), que l'on utilisera pour l'évaluation des fonctions élémentaires, le langage définit si le premier index définit la ligne où la colonne. En langage C, le premier élément définit la ligne et le deuxième la colonne. Cet ordre est inversé en FORTRAN.

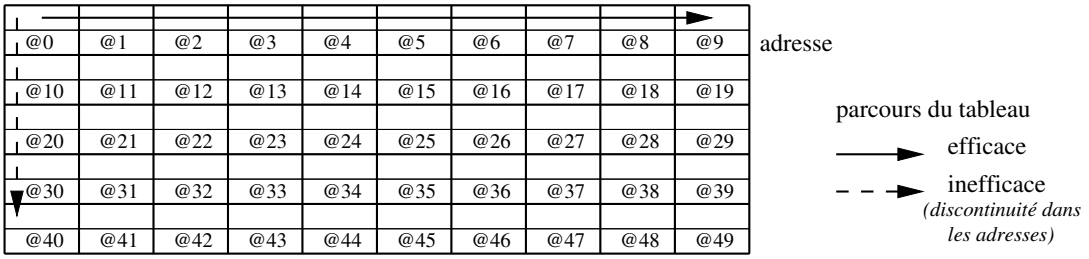


FIG. 3.2: *Les différents parcours d'un tableau à deux dimensions.*

Prenons l'exemple de deux itérations imbriqués parcourant l'une les lignes et l'autre les colonnes d'un tableau. Le code suivant est alors préférable en langage C :

Listing 3.1: *Accès efficace à un tableau en langage C*

```
1 for(i=0; i<N; i++)
2   for(j=0; j<N; j++)
3     A[i][j] = B[i][j] + C[i][j] * D;
```

alors que le code à choisir en FORTRAN est le suivant :

Listing 3.2: *Accès efficace à un tableau en langage FORTRAN*

```
1 DO j=1,N
2   DO i=1,N
3     A(i,j) =B(i,j) + C(i,j) * D;
4   ENDDO
5 ENDDO
```

Nous noterons également que sur cet exemple particulier, les performances décroissent au fur et à mesure que N se rapproche de la taille d'une ligne du cache (plusieurs lignes de tableau par ligne de cache), et cessent de décroître lorsque N dépasse la taille d'une ligne du cache.

3.1.3.2 Alignement mémoire

La mémoire est un élément central pour obtenir de bonnes performances. Nous avons vu que la mémoire n'est pas un élément «continu», mais qu'elle est divisée à plusieurs niveaux, niveau de la page, de la ligne de cache, d'un mot mémoire, de l'octet. Par conséquent, lorsque l'on stocke des éléments en mémoire, il faut s'assurer à tous les niveaux que ces éléments ne sont pas à cheval sur une de ces frontières. Cette technique s'appelle *l'alignement mémoire*, et est plus ou moins bien prise en charge par le compilateur.

Par exemple sur une architecture de type Pentium, lorsque l'on accède à une donnée mémorisée sur 64 bits (par exemple un flottant double précision), et que cette donnée n'est pas alignée sur une frontière de 8 octets alors le surcoût lié à cette perte d'alignement coûte 3 cycles supplémentaires, et beaucoup plus si l'on franchit une frontière de ligne de cache.

Une conséquence directe se trouve dans l'utilisation des structures, composées par exemple d'un flottant double précision et d'un flottant simple précision. Une solution connue sous le nom de *padding*, ou remplissage, consiste à rajouter un élément vide dont la taille permet l'alignement mémoire et ainsi de ne plus être à cheval sur une des frontières. Par exemple, l'option de compilation `-malign-double` de gcc permet de forcer l'alignement mémoire des flottants double précision.

3.2 Opérations exactes sur les flottants

Le résultat d'une addition ou d'une multiplication flottante n'est pas forcément un nombre machine, il doit être arrondi. Cependant, il est parfois nécessaire d'effectuer ces opérations de façon exacte. Nous allons dans cette section détailler les différents algorithmes permettant d'exécuter ces opérations de façon exacte. De plus, nous allons mettre en pratique les différentes remarques faites au cours des sections 3.1 et 3.1.3, pour en améliorer l'efficacité.

3.2.1 Codage d'un flottant double précision

Un nombre flottant double précision est codé sur 64 bits, soit deux fois plus qu'un entier codé généralement sur 32 bits. Pour certaines opérations comme les comparaisons, les additions, les décalages et les masques, il est avantageux d'utiliser la faible latence des unités entières. Toutefois il faut obtenir la partie haute et la partie basse d'un flottant double précision. Nous appelons la partie haute celle contenant le signe, l'exposant ainsi que les 20 bits de poids fort de la mantisse. La partie basse est celle qui contient les 32 bits de poids faible de la mantisse.

L'ordre dans lequel sont rangés en mémoire les deux mots de 32 bits varie selon le type d'architecture. Une architecture est dite *Little Endian* si les bits de poids faible du nombre sont stockés en mémoire à la plus petite adresse. Les processeurs x86 utilisent cette représentation. À l'inverse, une architecture où les bits de poids fort sont stockés en mémoire à la plus petite adresse est dite *Big Endian*. Les processeurs PowerPC sont de ce type.

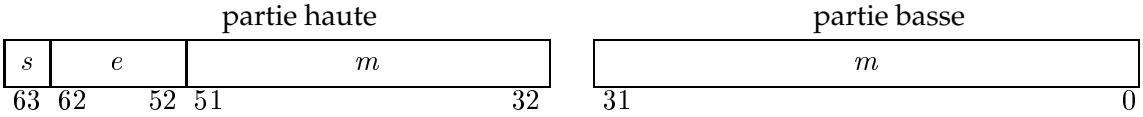


FIG. 3.3: *Stockage des flottants double précision en machine*

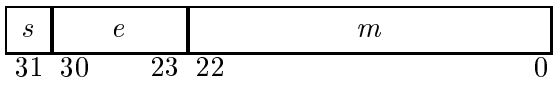


FIG. 3.4: *Stockage des flottants simple précision en machine*

Dans le format 'Little Endian', la façon d'accéder respectivement aux octets 1, 2, 3, 4... se fait de la même façon quel que soit le format : on accède en premier aux octets de poids faibles. Cette représentation est avantageuse pour les routines multiprécision où l'accès aux bits de poids faibles doit se faire en premier. En revanche, dans le cas d'une représentation 'Big Endian', l'accès aux informations de signe et d'exposant permet de tester rapidement les caractéristiques d'un nombre sans avoir à disposer d'une information supplémentaire sur le format (simple précision, double précision,...). De plus dans ce format, les nombres sont stockés de la même façon qu'ils sont imprimés, ce qui rend les conversions binaires vers décimal particulièrement efficaces.

Le code suivant permet de récupérer la partie haute et la partie basse d'un flottant x en fonction du format de représentation de la machine. Ce code ne fonctionne que sur les systèmes où les entiers (type C99 «`int`») sont représentés sur 32 bits.

Listing 3.3: *Extraction de la partie haute et basse d'un flottant double précision*

```
/* LITTLE_ENDIAN/BIG_ENDIAN sont définis par l'utilisateur ou  */
/* automatiquement par des outils tels que autoconf/automake.  */

#ifdef LITTLE_ENDIAN
#define HI(x) *(1+(int*)&x)
#define LO(x) *(int*)&x
#elif BIG_ENDIAN
#define HI(x) *(int*)&x
#define LO(x) *(1+(int*)&x)
#endif
```

Le principal inconvénient de ce code, est qu'il ne permet pas de *load-to-store forwarding* [42] à l'intérieur du pipeline. Cette technique, aussi appelée court-circuit permet dans le cas où une instruction attend le résultat d'une instruction précédente, d'envoyer directement le résultat produit par l'unité d'exécution à la deuxième instruction sans transiter par les registres ou la mémoire. Cette technique implémentée dans tous les processeurs généralistes, permet dans le cas de dépendances de données d'économiser les cycles d'écriture et de lecture du résultat. Cependant avec ce code nous ne pouvons utiliser cette technique car x est un flottant double précision sur 8 octets et le résultat renvoyé est un entier sur 4 octets, nous sommes donc obligés de travailler sur les adresses mémoires.

3.2.2 L'addition exacte

Les algorithmes suivants permettent d'effectuer des additions flottantes de manière à disposer de plus de précision qu'avec les flottants double précision du format IEEE-754. Le principe est de représenter le résultat d'une opération flottante (addition ou multiplication) par la somme de deux flottants. Il existe cependant pour chacun de ces algorithmes des limitations que nous mentionnerons à chaque fois.

Listing 3.4: *Fast2SumCond*

```
#define Fast2SumCond(s, r, a, b) \
        {double z, _a=a, _b=b;   \
         s = _a + _b;            \
         if ((HI(_a)&0x7FF00000) > (HI(_b)&0x7FF00000)){  \
           z = s - _a;           \
           r = _b - z;           \
         }else {                 \
           z = s - _b;           \
           r = _a - z;           \
         }}
```

Cet algorithme nécessite 3 opérations flottantes, 2 masques et 1 test sur les entiers.

Théorème 3 (Algorithme Fast2Sum [57]) *Si a et b sont des nombres flottants, alors la méthode Fast2SumCond calcule deux nombres flottants s et r, tels que $s + r = a + b$ avec $s = a \oplus b$.*

La preuve de cet algorithme est publiée dans [78] et dans [38]. Les conditions de cet algorithme ont été améliorées par Boldo [13] qui a prouvé à l'aide de l'assistant de preuve Coq, que seul un test sur les exposants est nécessaire pour l'algorithme Fast2Sum. Une façon simple et rapide de le réaliser est d'utiliser le mot machine (32 bits) contenant l'information sur l'exposant, ce qui correspond à la partie haute. Une définition de la fonction `HI` est donnée en section 3.2.1.

Voici quelques remarques concernant ce code, et notamment un début d'explication sur les raisons pour lesquelles nous préférons effectuer le test avec des entiers et non avec des flottants (test équivalent à `if ((abs(a)) > abs(b))`).

- Grâce à Boldo [13], il n'est pas nécessaire de tester le flottant complet, mais seulement l'exposant.
- Pour le coût d'un masque (en général un seul cycle) nous supprimons l'information de signe et obtenons l'exposant.
- Il y a plusieurs unités capables d'effectuer ce masque dans les processeurs. Il peut donc être effectué en parallèle pour a et b.
- Les tests sur les entiers sont plus rapides que sur les flottants.
- Nous effectuons du calcul flottant avant et après ce test. Les unités entières effectuent moins de calcul que les unités flottantes. Donc ce test répartit la charge de travail entre les deux.
- Les valeurs de a et b sont disponibles dès le début. Ces opérations entières peuvent être lancées, sous réserve des contraintes sur les ports disponibles, en même temps que l'addition flottante `s=a+b`.
- Nous sommes en présence d'un `if...then...else` avec seulement 2 instructions dans chaque corps. Cette opération peut être parfaitement prédicatée sur les processeurs le permettant, et il n'y a alors pas de rupture de pipeline.
- Les trois instructions flottantes sont dépendantes, ces instructions ne pourront pas être pipelinées (les latences de chaque instruction s'additionnent).
- L'inconvénient est qu'il faut transférer le contenu de deux registres flottants (a, b) vers des registres entiers. Cette opération s'effectue en passant par la mémoire.

Si nous possédons une connaissance a priori sur la valeur des exposants de a et b (l'exposant de a est supérieur ou égal à l'exposant de b) alors l'algorithme précédent peut se ramener à la séquence d'instructions suivante :

Listing 3.5: *Fast2Sum*

```
#define Fast2Sum(s, r, a, b)      \
        {double z, _a=a, _b=b;    \
         s = _a + _b;             \
         z = s - _a;              \
         r = _b - z; }
```

Le coût de cet algorithme est désormais de 3 opérations flottantes.

Pour la somme exacte de 3 nombres flottants nous disposons de l'algorithme *Fast3Sum*.

Théorème 4 (Algorithme Fast3Sum [66]) *Si a, b et c sont des nombres flottants triés par exposant décroissant, alors la méthode suivante calcule trois nombres flottants $r1$, $r2$ et $r3$, tels que $r1+r2+r3 = a+b+c$ exactement, avec $|r1| \geq |r2| \geq |r3|$ et $r1 = a \oplus (b \oplus c)$.*

Listing 3.6: *Fast3Sum*

```
#define Fast3Sum(r1, r2, r3, a, b, c) \
        {double u, v, w;              \
         Fast2Sum(u, v, b, c);        \
         Fast2Sum(r1, w, a, u);       \
         Fast2Sum(r2, r3, w, v); }
```

Cet algorithme a été prouvé en Coq dans [66], et nécessite seulement 9 additions flottantes.

3.2.3 La multiplication exacte

Nous allons maintenant considérer le cas de la multiplication avec plus de précision qu'une simple multiplication flottante.

Théorème 5 (double multiplication [34, 57]) *Soient a et b deux nombres flottants, et $p \geq 2$ la taille de leurs mantisses. Soit $c = 2^{\lceil \frac{p}{2} \rceil} + 1$. La méthode décrite ci-après calcule deux nombres flottants $r1$ et $r2$ tels que $r1 + r2 = a \times b$ avec $r1 = a \otimes b$ dans le cas $p = 53$ (double précision) :*

Listing 3.7: *DekkerCond*

```
void inline DekkerCond(double *r1, double *r2, double a, double b){
  double two_m53 = 1.1102230246251565404e-16;  /* 0x3CA00000, 0x00000000 */
  double two_53  = 9007199254740992.;          /* 0x43400000, 0x00000000 */
  double c       = 134217729.;                 /* 0x41A00000, 0x02000000 */
  double u, up, u1, u2, v, vp, v1, v2, r1, r2;

  if ((HI(a)&0x7FF00000)>0x7C900000) u = a*two_m53;
  else                               u = a;
  if ((HI(b)&0x7FF00000)>0x7C900000) v = b*two_m53;
  else                               v = b;

  up = u*c;         vp = v*c;
  u1 = (u-up)+up;   v1 = (v-vp)+vp;
  u2 = u-u1;        v2 = v-v1;

  *r1 = u*v;
  *r2 = (((u1*v1-*r1)+(u1*v2))+(u2*v1))+(u2*v2)

  if ((HI(a)&0x7FF00000)>0x7C900000) {*r1 *= two_53; *r2 *= two_53;}
  if ((HI(b)&0x7FF00000)>0x7C900000) {*r1 *= two_53; *r2 *= two_53;}
}
```

Nous devons tester a et b avant et après le cœur de l'algorithme pour éviter les possibles dépassements de capacité lors de la multiplication par c. Le coût global de cet algorithme dans le pire cas est de 4 tests entiers, 10 additions flottantes et de 13 multiplications flottantes.

Au coeur de cet algorithme, nous noterons que u_1 et v_1 contiennent respectivement les 27 bits de poids fort de a et b. Si nous savons que a et b sont tous deux inférieurs à 2^{970} alors nous pouvons supprimer les tests de début et de fin. Cette information supplémentaire permet de réduire le coût de cet algorithme à 10 additions flottantes et 7 multiplications flottantes.

Listing 3.8: *Dekker*

```
void inline Dekker(double *r1, double *r2, double u, double v){
  double c       = 134217729.;                 /* 0x41A00000, 0x02000000 */
  double up, u1, u2, vp, v1, v2;

  up = u*c;         vp = v*c;
  u1 = (u-up)+up;   v1 = (v-vp)+vp;
  u2 = u-u1;        v2 = v-v1;

  *r1 = u*v;
  *r2 = (((u1*v1-*r1)+(u1*v2))+(u2*v1))+(u2*v2)
}
```

3.2.4 Conversions d'un flottant vers un entier

Il existe deux solutions pour effectuer une conversion d'un flottant vers un entier.

Solution 1

La première solution qui nous vient à l'esprit correspond au code suivant :

Listing 3.9: *Solution 1*

```
1 double x;
2 int i;
3 i=x;
```

La norme C impose que la conversion d'un flottant vers un entier se fasse par troncature. Cette conversion consiste à prendre la partie entière d'un flottant en utilisant le mode d'arrondi vers 0. Voici ce qui se passe dans le processeur :

1. Sauvegarder le mode d'arrondi courant
2. Passer en mode d'arrondi vers 0
3. Charger le flottant double précision et le sauvegarder dans un registre en tant qu'entier.
4. Restaurer le mode d'arrondi initial.

Cette conversion est lente car elle nécessite sur la plupart des processeurs d'attendre que les instructions déjà présentes dans le pipeline se terminent.

Nous noterons qu'il existe des architectures comme l'Itanium sur lesquelles ce code a un meilleur comportement. Sur ce processeur, il n'y a pas un seul registre de statuts (le registre contrôlant la précision, le mode d'arrondi, la gestion des exceptions, ...), mais quatre ! Ces registres sont utilisés en spécifiant pour chaque instruction flottante quel registre de statuts et par conséquent quel mode d'arrondi utiliser. Il n'y a pas donc pas rupture de pipeline [39].

Solution 2

Nous utiliserons un autre algorithme bien connu [9] qui ne casse pas le flot d'instruction du pipeline. Pour cela nous utilisons le mode d'arrondi au plus près qui est le mode d'arrondi que nous utiliserons tout au long de cette thèse. En revanche, cette conversion flottant vers entier pourra différer d'au plus 1 avec la version précédente utilisant la troncature. Cet algorithme place dans `i` l'entier le plus proche du flottant double précision `d`. L'astuce de cet algorithme consiste à ajouter $2^{52} + 2^{51}$ au flottant à convertir pour placer la partie entière dans la partie basse du flottant double précision. Nous utilisons la valeur $2^{52} + 2^{51}$ et non pas simplement 2^{52}, car la valeur 2^{51} sert à contenir les propagations de retenues dans le cas où le nombre à convertir est négatif.

Listing 3.10: *Solution 2*

```
1 #define DOUBLE2INT(i, d) \
2   {double t=(d+6755399441055744.0); i=LO(t);}
```

Le résultat est cohérent si le flottant double précision f est tel que $|f| < 2^{32}$ dans le cas d'une conversion vers un entier non signé et $|f| < 2^{31}$ dans le cas d'une conversion vers un entier signé.

3.3 Test de l'arrondi

Lors de l'évaluation d'une fonction élémentaire f, nous devons tester efficacement si le résultat obtenu peut être arrondi correctement. Cette opération revient à détecter si l'on est en présence d'un cas difficile à arrondir correspondant à une suite consécutive de 0 ou de 1 après le $53^{\text{ème}}$ bit (voir la partie sur le dilemme du fabricant de table en section 1.2).

Dans les schémas d'évaluation nous commettons des erreurs «mathématiques» d'approximation, mais également des erreurs d'arrondi inhérentes à chaque opération flottante. Aussi après une étude fine de l'algorithme utilisé pour l'évaluation, nous disposons d'une borne δ correspondant à l'erreur relative finale. L'objectif est de détecter si le résultat final représenté par la somme de deux nombres flottants double précision x_{hi} et x_{lo}, peut être correctement arrondi en connaissant δ.

$$f(x) = (x\_hi + x\_lo).(1 + \delta) \tag{3.1}$$

Pour effectuer ces tests nous avons développé quatre algorithmes pour chacun des quatre modes d'arrondi de la norme IEEE-754.

Nous rappelons que $\oplus$, $\ominus$ et $\otimes$ représentent respectivement les opérations machine en mode d'arrondi au plus près des opérations mathématiques $+$, $-$, $\times$.

3.3.1 Arrondi au plus près

Le code suivant permet, pour le coût de 4 additions flottantes, 1 multiplication flottante et 1 test flottant, de détecter si l'arrondi au plus près de x_{hi} et x_{lo} est possible avec une erreur relative inférieure à δ. Nous noterons par err le flottant double précision tel que $err = 1+\delta\times 2^{55}$, ce qui correspond à la plus petite valeur telle que $(\frac{1}{2} - \delta.2^{53}).(1 + \varepsilon_{-53}).err > \frac{1}{2}$, comme nous le verrons dans l'équation 3.5.

Listing 3.11: *Test si l'arrondi est possible en arrondi au plus près*

```
1 double A, B;
2
3 Fast2Sum(A, B, x_hi , x_lo)
4 if (A == (A + B * err))
5   return A;
6 else{
7   /* Cas difficile */
8   ...
9 }
```

Les conditions pour utiliser cette méthode sont les suivantes :

- $|x\_hi| \geq |x\_lo|$ condition exigée par le Fast2sum
- x_hi et x_lo ne représentent pas une valeur parmi $(sNan, qNan)$.

Preuve :

Si x_hi ou x_lo représente $\pm\infty$ alors on vérifie de tête que le test est vrai et le résultat retourné également.

L'algorithme Fast2Sum nous garantit que $A + B = x\_hi + x\_lo$ avec $A = x\_hi \oplus x\_lo$. De plus les opérations sont effectuées avec l'arrondi au plus près, donc

$$|B| \leq \frac{1}{2} ulp(A) \tag{3.2}$$

Ainsi l'équation 3.1 est équivalente à

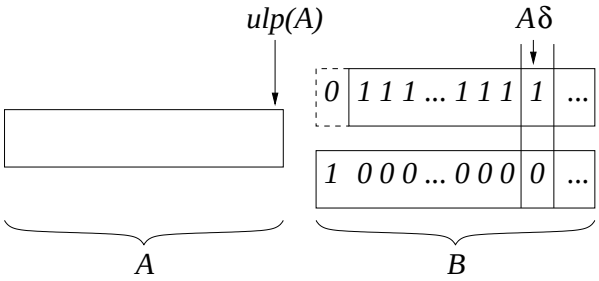
$$f(x) = (A + B).(1 + \delta)$$

Montrons que le test $(A == A \oplus (B \otimes err))$ est faux si et seulement si on est dans un cas difficile à arrondir, en arrondi au plus près.

En arithmétique double précision IEEE avec arrondi au plus près, on sait que $A \neq A \oplus X$ ssi $|\frac{1}{2}ulp(A)| < |X|$. Donc $(A \neq A \oplus (B \otimes err))$ ssi

$$|\frac{1}{2}ulp(A)| < |B \otimes err| \tag{3.3}$$

Comme nous l'avons vu en section 1.2 page 11, on est dans un cas difficile à arrondir ssi l'on est dans le cas de figure suivant



Nous noterons que les deux variables A et B peuvent être de signe opposé. De façon plus formelle, on est dans un cas difficile ssi B vérifie l'équation suivante :

$$|(\frac{1}{2} - \delta.2^{53}).ulp(A)| \leq |B| \leq |\frac{1}{2}.ulp(A)| \tag{3.4}$$

Montrons que le test $(A == A \oplus (B \otimes err))$ est faux ssi nous sommes dans un cas difficile.

La première partie de l'équation 3.4 ($|(\frac{1}{2} - \delta.2^{53}).ulp(A)| \leq |B|$) implique que

$$\begin{array}{rcl} |(\frac{1}{2} - \delta.2^{53}).ulp(A)| & \leq & |B| \\ |((\frac{1}{2} - \delta.2^{53}).ulp(A)) \otimes (1 + \delta.2^{55})| & \leq & |B \otimes err| \\ |((\frac{1}{2} - \delta.2^{53}).ulp(A)).(1 + \delta.2^{55}).(1 + \varepsilon_{-53})| & \leq & |B \otimes err| \\ |((\frac{1}{2} - \delta.2^{53}).(1 + \delta.2^{55}).(1 + \varepsilon_{-53})).ulp(A)| & \leq & |B \otimes err| \end{array} \tag{3.5}$$

Or :

$$\frac{1}{2} < |(\frac{1}{2} - \delta.2^{53}).(1 + \delta.2^{55}).(1 + \varepsilon_{-54})|$$

donc

$$|\frac{1}{2}ulp(A)| < |B \otimes err|$$

La seconde partie de l'équation 3.4 ($|B| \leq |\frac{1}{2}.ulp(A)|$), implique que $|\frac{1}{2}ulp(A)| < |B \otimes err|$ si $|B| = |\frac{1}{2}.ulp(A)|$.

Ce qui donne $A \oplus (B \otimes err) > A \oplus \frac{1}{2}ulp(A) \neq A$. Nous venons donc de montrer que le test $(A == A \oplus (B \otimes err))$ est faux si nous sommes dans un cas difficile à arrondir.

A l'inverse, si nous ne sommes pas dans un cas difficile à arrondir alors par définition de l'arrondi au plus près $0 \leq |B| < |(\frac{1}{2} - \delta.2^{53}).ulp(A)|$, équivalent à

$$0 \leq |B \otimes err| < \frac{1}{2}ulp(A)$$

Ce qui donne $A \oplus (B \otimes err) = A$. Nous venons donc de montrer que le test $(A == A \oplus (B \otimes err))$ est vrai si nous sommes dans un cas facile à arrondir.

En conclusion, le test $(A == A \oplus (B \otimes err))$ est faux si et seulement si nous sommes dans un cas difficile à arrondir, en arrondi au plus près.

Nous noterons que l'utilisation du FMA pour l'opération $(A + (B \times err))$ est possible, car il est sans conséquence sur le comportement de l'algorithme.

3.3.2 Arrondi vers $+\infty$

De façon similaire à ce qui a été proposé dans la section précédente, nous allons décrire la méthode permettant de détecter si l'arrondi vers $+\infty$ de x_{hi} et x_{lo} est possible avec une erreur relative inférieure à δ. Nous noterons par err l'entier 32 bits tel que $err = |\log_2(\delta)| \times 2^{20}$.

La particularité de cette méthode est d'utiliser le mode d'arrondi par défaut du processeur : l'arrondi au plus près. En effet c'est le mode d'arrondi minimisant l'erreur mathématique. De plus changer le mode d'arrondi machine est coûteux, il serait donc regrettable et coûteux en temps d'avoir à changer le mode d'arrondi du processeur juste pour arrondir le résultat final.

Listing 3.12: *Test si l'arrondi est correct en arrondi vers $+\infty$*

```
1  union {long long int l; double d;} A;
2  double B;
3  int    exp_A;
4
5  Fast2Sum(A.d, B, x_hi, x_lo)
6
7  exp_A = (HI(A.d) & 0x7FF00000) - err;
8
9  if ((HI(B)&0x7FF00000) > exp_A){
10   /* On est capable d'arrondir */
11   if (HI(B) > 0)
12     A.l += 1;
13
14   return A;
15 }else {
16   /* Cas difficile */
17   ...
18 }
```

Le coût de cet algorithme est de 4 additions flottantes, 1 multiplication flottante, 2 tests entiers, 3 masques et 1 addition entière. Malgré le nombre plus important d'opérations de cet algorithme par rapport à celui de l'arrondi au plus près, son chemin critique est tel que dans certain cas il peut s'avérer plus rapide.

Les conditions pour utiliser cette méthode sont les suivantes :

- $|A| \geq 2^{-1022}/\delta$ de façon à ce que A ne représente pas un dénormalisé. Si cette condition ne peut pas être vérifiée, il faut alors utiliser une astuce similaire à celle utilisée dans `DekkerCond`. La solution consiste à tester l'exposant de A qui, s'il est inférieur à une certaine valeur conduira à une multiplication flottante sur A et B par une constante, puissance de deux. Cette multiplication ramènera les valeurs de A et de B dans la plage des

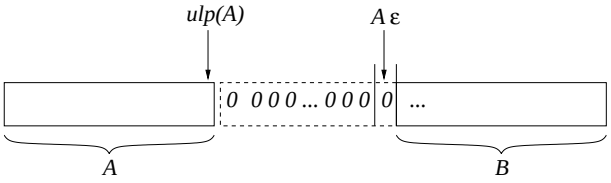
nombres normalisés et permettra d'effectuer les tests sur les entiers. Enfin, il ne faudra pas oublier de multiplier le résultat final par l'inverse de la constante utilisée.
- $|x\_hi| \geq |x\_lo|$ condition exigée par le Fast2sum,
- x_hi et x_lo ne représentent pas une valeur parmi $(sNan, qNan, +Inf, -Inf)$.
- `long long int` représente un entier sur 64 bits.

Preuve :
Dans ce programme la variable `err` de type `int` correspond au nombre de bits corrects du résultat, multiplié par 2^{20}. Cette multiplication par 2^{20} est justifiée par la position de l'exposant dans la représentation IEEE des flottants double précision (voir la figure 3.3, page 46).

L'algorithme Fast2sum nous garantit que $A + B = x\_hi + x\_lo$ avec $A = x\_hi \oplus x\_lo$ et $|B| \leq \frac{1}{2}\text{ulp}(A)$.
Ainsi l'équation (3.1) est équivalente à

$$f(x) = (A + B).(1 + r) \quad \text{avec} \quad |r| \leq \delta \leq 2^{-54}$$

De par la définition de l'arrondi dirigé vers $+\infty$, et $|B| \leq \frac{1}{2}\text{ulp}(A)$, on est dans un cas difficile à arrondir ssi on est dans le cas de figure suivant :



De façon plus formelle, on est dans un cas difficile ssi B vérifie l'équation suivante :

$$0 \leq |B| \leq \delta.2^{53}.ulp(A) \tag{3.6}$$

La ligne 7 place dans exp_A (l'exposant de A)$-|\log_2(\delta)|$. Cette opération est valide à la condition que A ne représente pas un dénormalisé, un NaN ou un infini.

L'expression $((HI(B)\&0x7FF00000) > exp\_A)$ teste si l'exposant de B est supérieur à celui de $A - |\log_2(\delta)|$. Ce test est vrai si A et B ne sont pas des valeurs spéciales et $|B| \geq \delta.2^{53}.ulp(A)$. Donc, si l'équation (3.6) est vraie alors ce test est vrai.

De par la définition de la fonction `HI`, $HI(x) < 0$ si et seulement si $x < 0$ où x est un flottant double précision ne représentant pas une valeur spéciale.

Nous avons donc deux cas :
- Si $B < 0$, l'arrondi au plus près utilisé dans le Fast2sum nous garantit que A correspond à l'arrondi vers $+\infty$ de $x\_hi + x\_lo$.
- Si $B > 0$ alors il nous faut ajouter 1 ulp à A pour disposer de l'arrondi vers $+\infty$. Nous utilisons pour cela la propriété 2, page 7 sur la «continuité» des flottants.

3.3.3 Arrondi vers $-\infty$

Le code suivant permet de savoir si l'arrondi vers $-\infty$ est possible. Son principe est similaire au test vers $+\infty$, par conséquent la preuve de cet algorithme est omise.

Listing 3.13: *Test si l'arrondi est correct en arrondi vers* $-\infty$

```
1 int     err = |log2(δ)| × 2^20 ;
```

```
union {long long int l; double d;} A;
double B;
int    exp_A;

Fast2Sum(A.d, B, x_hi, x_lo)

exp_A = (HI(A.d) & 0x7FF00000) - err;

if ((HI(B)&0x7FF00000) > exp_A){
  /* On est capable d'arrondir */
  if (HI(B) < 0)
    A.l -= 1;

  return A;
}else {
  /* Cas difficile */
  ...
}
```

3.3.4 Arrondi vers 0

Le code suivant permet de savoir si l'arrondi vers 0 est possible. Son principe est similaire au test vers $+\infty$, par conséquent la preuve de cet algorithme est omise.

Listing 3.14: *Test si l'arrondi est correct en arrondi vers* 0

```
int    err = |log2(δ)| × 2^20 ;
union {long long int l; double d;} A;
double B;
int    exp_A;

Fast2Sum(A.d, B, x_hi, x_lo)

exp_A = (HI(A.d) & 0x7FF00000) - err;

if ((HI(B)&0x7FF00000) > exp_A){
  /* On est capable d'arrondir */
  if (HI(A.d)^HI(B) < 0){
    /* Si A et B sont de signes différents */
    A.l -= 1-((HI(A.d)>>31)<<1);
  }
  return A;
}else {
  /* Cas difficile */
  ...
}
```

3.4 Compromis entre taille de table, degré du polynôme et fréquence d'utilisation de la fonction

Nous avons vu dans les sections précédentes quelques opérateurs de base (addition et multiplication exactes, test de l'arrondi...) que nous allons utiliser pour l'évaluation des fonctions élémentaires. Cependant il nous reste un autre élément à étudier : l'impact de la hiérarchie mémoire sur l'évaluation des fonctions élémentaires.

Comme nous pouvons l'observer à travers plusieurs articles [65, 67, 80, 81] et au cours de la section 2.2.1.5, de nombreuses méthodes d'évaluation de fonctions élémentaires utilisent

des algorithmes basés sur la tabulation de données. Malheureusement aucun de ces articles ne donne une estimation claire de l'impact des tables sur le coût global de ces algorithmes. L'argument le plus souvent avancé est qu'il n'existe aucune constance dans les caractéristiques des unités d'exécutions et encore moins sur l'implémentation des caches.

Pour ces raisons le choix de la taille de table par rapport au degré des polynômes d'approximation est laissé comme un paramètre non défini ou non justifié. Le but de cette section est de montrer que choisir des tables de taille inférieure à 4 Koctets conduit à de bonnes performances quelle que soit la fréquence d'utilisation de la fonction. Nous utiliserons pour cela les éléments présentés en section 3.1 pour cette étude.

3.4.1 Objectifs de l'étude

Nous savons que les performances de l'évaluation des fonctions élémentaires sont le résultat d'un compromis entre précision et vitesse d'exécution. Pour cette étude, nous souhaitons nous attacher seulement à la rapidité des évaluations, tout en gardant comme objectif l'arrondi correct pour la double précision. Pour obtenir l'arrondi correct, il faut effectuer les calculs en interne avec une plus grande précision. Une solution est d'utiliser les doubles étendus (plus de 64 bits de précision) lorsqu'ils sont disponibles. Néanmoins si nous visons la portabilité, nous devons nous conformer à la norme IEEE-754 et n'utiliser que les flottants double précision. Nous avons donc fixé arbitrairement la précision finale requise pour l'évaluation à environ 70 bits pour ne pas biaiser les résultats par l'utilisation d'arithmétiques non disponibles sur tous les processeurs. Cette précision correspond également à la précision que nous souhaitons atteindre dans la première phase de l'évaluation des fonctions élémentaires avec arrondi correct comme nous le verrons au chapitre 5.

Nous avons eu l'occasion de voir au cours de la section 2.2 que la façon la plus courante d'évaluer les fonctions élémentaires en logiciel est d'effectuer deux réductions d'argument consécutives suivies d'une évaluation polynomiale de l'argument réduit. La première réduction d'argument est basée sur les propriétés mathématiques de la fonction (par exemple $\sin(2k\pi + x) = \sin(x)$). La deuxième réduction d'argument utilise des données tabulées. Par conséquent, les paramètres de ce schéma d'évaluation sont le degré du polynôme et la taille de la table utilisée pour atteindre la précision définie préalablement. Choisir un polynôme de haut degré a pour conséquence de ne nécessiter qu'une petite taille de table, mais augmente le nombre d'instructions flottantes nécessaires à son évaluation. À l'inverse une table de grande taille utilisée pour la réduction d'argument conduit à une évaluation polynomiale de petite taille mais augmente l'utilisation des ressources mémoires.

Il n'est a priori pas possible de déterminer le meilleur couple (degré du polynôme/taille de table) sans effectuer des tests exhaustifs pour chaque fonction. Cette étape de test implique un fort investissement pour le développeur soucieux d'obtenir de bonnes performances pour l'implémentation de fonctions élémentaires. L'objectif de cette étude est de simplifier cette étape en effectuant des tests exhaustifs représentatifs d'un grand nombre de fonctions élémentaires, dans différents cas de figure : petite table/grand polynôme, grosse table/petit polynôme et fréquence variable d'appel à une fonction.

3.4.1.1 Fréquences d'utilisation de la fonction

Le nombre d'appels à l'évaluation d'une fonction détermine la probabilité avec laquelle les données et les instructions utilisées par l'évaluation vont être présentes dans les caches de plus

haut niveau. Si des appels fréquents à une même fonction sont effectués, alors cette probabilité est élevée et nous pouvons donc nous permettre d'utiliser plus d'accès à des données et moins d'instructions. À l'inverse, si cette fonction est peu appelée, alors cette probabilité est faible, et il est plus efficace d'utiliser une évaluation utilisant peu d'accès à des tables.

Une solution pour maintenir de bonnes performances quelle que soit la fréquence d'utilisation d'une fonction est de construire une bibliothèque optimisée en fonction du nombre d'appels prévu. Cette solution est celle qui a été retenue par Sun avec la bibliothèque *libmvec* et les différents schémas d'évaluation qui la composent (mvec1, mvec10, mvec100 ...). Cependant, elle est relativement complexe car elle reporte le problème vers l'utilisateur qui doit disposer de suffisamment d'information sur son programme mais également vers le concepteur des bibliothèques en multipliant son travail : il doit en effet écrire plusieurs programmes d'évaluation pour une même fonction.

Une autre solution est de trouver un bon compromis entre fréquence d'appels et schéma d'évaluation. Il reste toutefois à estimer le surcoût engendré par ce choix, ce que nous ferons dans la suite.

3.4.1.2 Degré du polynôme d'approximation

L'évaluation polynomiale utilise des multiplications et additions flottantes. Les performances qui lui sont associées dépendent des caractéristiques des unités flottantes des processeurs cibles (voir tableau 3.1). Par exemple, voici deux processeurs et l'influence qu'ils ont sur le schéma d'évaluation polynomiale :

- Sur un Pentium 4 avec une latence de 5 pour l'addition flottante et de 7 pour la multiplication, l'évaluation par un schéma de Horner d'un polynôme de degré n coûte approximativement $12.n$ cycles. Une autre solution est de diviser l'évaluation polynomiale en deux sous-groupes sans dépendance de données entre chaque groupe. Avec ce schéma l'évaluation polynomiale sur le même Pentium 4 ne coûte plus que $12\lceil n/2 \rceil + 13$ cycles.
- Un autre exemple peut être pris avec le cas de l'Itanium qui dispose de 2 FMA. Sur cette architecture, la latence de la meilleure évaluation polynomiale (celle faite à la main, et différente de Horner) a une complexité en $Cst.\lfloor \log_2 n \rfloor + Cst$ où n est le degré du polynôme [39].

3.4.1.3 Taille de table pour la réduction d'argument

En observant les caractéristiques des caches mentionnées en section 3.1.2 et plus particulièrement le tableau 3.2, nous pouvons en déduire qu'une implémentation des fonctions élémentaires performante en termes de gestion mémoire sur le plus grand nombre de systèmes doit par ordre de priorité :

1. Utiliser un petit nombre de pages. Un défaut de page peut coûter jusqu'à 10^6 cycles d'horloges.
2. Conserver les données utilisées par l'évaluation dans le cache de plus haut niveau. Le chargement d'une donnée qui n'est pas dans le cache L1, coûte au minimum une vingtaine de cycles.
3. Utiliser un multiple de la taille des pages. L'application doit faire la meilleure utilisation de la mémoire qui lui est allouée, de façon à réduire la fragmentation interne des pages.

4. Utiliser moins que la taille d'une voie du cache, et regrouper le tout dans une structure continue en mémoire. Le but est de limiter les problèmes de remplacement de données dans le cache.

Nous en déduisons la proposition suivante :

Proposition
Pour exhiber de bonnes performances, une implémentation efficace d'une évaluation de fonctions élémentaires doit utiliser une quantité de données inférieure au minimum entre la taille d'une page et la taille d'une voie du cache.

Après observations des données présentées dans le tableau 3.2, nous pouvons quantifier cette proposition. En effet, la plupart des processeurs disposent de 4 Koctets par voie dans les caches de données, et des pages de 4 Koctets pour les systèmes Linux et HP-UX à quelques exceptions près (par exemple, un Itanium avec une Linux Debian dispose de pages de 16 Koctets par défaut). Par conséquent les évaluations de fonctions élémentaires basées sur des algorithmes utilisant moins de 4 Koctets de données donneront les meilleurs résultats sur un grand nombre de processeurs et de systèmes.

3.4.2 Expérimentations

3.4.2.1 Description des expérimentations

Pour quantifier l'impact des caches sur l'évaluation des fonctions élémentaires, nous avons choisi trois fonctions typiques : le logarithme car cette fonction nécessite un polynôme de haut degré pour être bien approchée ; le sinus et le cosinus car ces deux dernières fonctions sont connues pour bien s'approcher par un polynôme de petit degré. La difficulté d'implémenter réellement les fonctions élémentaires nous a poussé à ne nous focaliser que sur ces deux groupes de fonctions. Néanmoins leurs caractéristiques opposées les rendent représentatives d'une grande variété de fonctions élémentaires (exponentielle, tangente, ...). Dans les deux cas nous faisons varier uniquement le degré du polynôme et la taille de table utilisée pour la réduction d'argument. Le degré du polynôme est le plus petit qui permette d'atteindre la précision désirée avec la taille de table choisie. Les algorithmes utilisés pour ces deux fonctions sont décrit en détails dans [28].

Le compromis entre le coût de la réduction d'argument et le degré du polynôme est donné Table 3.3 pour le logarithme et Table 3.4 pour les deux fonctions trigonométriques. Toutefois, les données présentes dans ces deux tables n'incluent pas les constantes utilisées dans le programme ainsi que les coefficients du polynôme, car elles deviennent vite négligeable en taille. Les polynômes ont été choisis pour obtenir au minimum 65 bits de précision pour le logarithme et 70 bits de précision pour les fonctions trigonométriques.

Pour les tests, 24 algorithmes et implémentations différentes ont été écrites, ce qui correspond à 12 programmes par fonction. Les programmes ont été écrits en langage C et compilés avec *gcc-3*, niveau 2 d'optimisation. Les temps d'exécution, en termes de latence, ont été mesurés par des instructions assembleurs spécifiques à chaque processeur. Les instructions de mesure de temps comprennent les primitives de synchronisation avec des instructions telles que CPUID pour les architectures x86 ainsi qu'un calibrage pour supprimer le surcoût lié à la mesure de temps elle-même. Les temps donnés sont donc des nombres de cycles d'horloge

dans le cas du Pentium III ou de l'Itanium ou dans une unité multiple d'un cycle d'horloge dans le cas du PowerPC G4.

Ces fonctions ont été testées sur des valeurs générées aléatoirement. Les tests sont composés d'appels consécutifs à une même fonction, comme c'est le cas dans certaines applications [45]. Nous avons limité le nombre d'appels consécutifs à une fonction à 300, les résultats pour des valeurs supérieures n'apportant pas d'informations supplémentaires. Avant chaque test, les caches sont vidés de façon à inclure les *démarrages à froid*, comme dans le cas d'une application réelle. Les caractéristiques des architectures testées sont les suivantes :

- Une station Pentium III avec une distribution Linux Debian 3.0, 4-Koctets de taille de page, gcc version 3.2.2.
- Une station HP i2000 basé sur un processeur Itanium avec une distribution Linux Debian 3.0, 16-Koctets de taille de page, gcc version 3.2.1.
- Une station Imac basé sur un processeur Power PC G4 avec OS X, 4-Koctets taille de page, gcc version 1151, basé sur gcc version 3.1.

Polynôme		Taille de	k
degré	précision	table en octet	
20	65	48	2
15	66	96	3
12	67	176	4
10	68	368	5
8	65	720	6
7	66	1456	7
6	65	2896	8
6	71	5792	9
5	67	11584	10
5	73	23168	11
4	66	46336	12
4	71	92688	13

TAB. 3.3: *Compromis entre la taille de table utilisée pour la réduction d'argument et le degré du polynôme utilisé pour atteindre* 65 *bits de précision sur l'intervalle* $[-2^{-k}, 2^{-k}]$ *pour la fonction* $\log(1+x)$.

3.4.2.2 Résultats expérimentaux

Seuls les appels fréquents comptent

Résultats pour des appels occasionnels à une fonction

Si l'on ne considère que quelques appels à une même fonction dans les figures du tableau 3.5, on peut conclure que le temps d'exécution n'est absolument pas corrélé à la taille de table utilisée par la réduction d'argument ni au degré de polynôme. Ce qui signifie que le temps d'exécution est dominé par le coût du 'démarrage à froid' lié aux nombreux défauts de cache. Cette propriété est observée pour toutes les fonctions sur tous les processeurs. Donc, pour une utilisation occasionnelle des fonctions élémentaires, ni le degré, ni la taille de table ont une influence mesurable sur le temps d'exécution total.

Sinus poly.		Cosinus poly.		Taille de table	k
deg.	précision	deg.	précision	en octet	
8	75	8	70	128	2
7	73	7	69	256	3
7	82	7	78	512	4
6	78	6	74	1024	5
5	72	5	69	2048	6
5	81	5	77	4096	7
4	71	4	67	8192	8
4	76	4	73	16384	9
4	82	4	79	32768	10
3	70	3	68	65536	11
3	75	3	72	131072	12
3	80	3	77	262144	13

TAB. 3.4: *Compromis entre la taille de table utilisée pour la réduction d'argument et le degré du polynôme utilisé pour atteindre* 70 *bits de précision sur l'intervalle* $[-2^{-k}, 2^{-k}]$ *pour les fonctions* cos *et* sin.

Résultats pour des appels fréquents à une fonction

La différence entre les méthodes utilisant des polynômes de petit ou grand degré peut être observée sur la figure 3.5, dans le cas de l'évaluation polynomiale du logarithme. Sur cette figure, on observe 14% de différence entre le temps mis par la meilleure méthode et celle basée sur la table de plus petite taille. Cette différence passe à 37% lorsque l'on considère la méthode basée sur la plus grande table. Donc la taille de table et le degré du polynôme sont des paramètres à prendre en considération lorsque l'on souhaite faire de nombreux appels à une fonction.

Bénéfice de l'évaluation polynomiale

Différences entre architectures

Les différences entre les architectures testées ne semblent pas évidentes, car sur chaque architecture testée la taille de page était de 4 KB.

L'Itanium présente de bons résultats sur les évaluations polynomiales (Figure 3.5) car parmi les processeurs testés, c'est le seul disposant de 2 FMA, et donc capable de démarrer 2 multiplications-additions à chaque cycle. C'est une des principales raisons pour laquelle ce processeur a un excellent comportement pour les évaluations polynomiales et que les meilleurs résultats sont obtenus pour une table de 720 Ko et non pour 2.8 Ko.Toutefois, en choisissant 2.8 Koctets de table, le surcoût engendré par ce choix par rapport au meilleur choix n'est que de 20 cycles. De plus, comme nous l'observons sur la figure correspondant du tableau 3.5, ce surcoût tend à diminuer avec le nombre d'appels. Nous notons également que la grande taille des caches et le grand nombre de voies de l'Itanium font que le surcoût dû aux grandes tables est moins important pour ce processeur que pour les autres.

Différences entre fonctions

Sur chaque architecture testée, nous observons que la différence de performance est moins importante pour le logarithme que pour les fonctions trigonométriques. Cette propriété est liée à

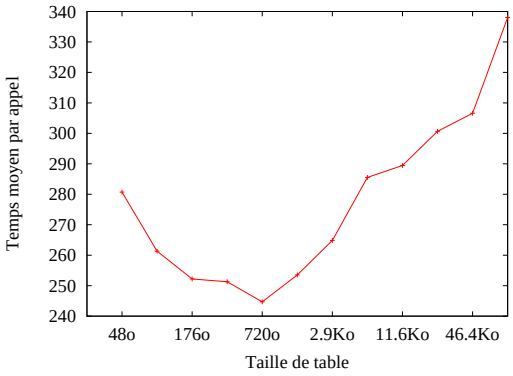


FIG. 3.5: *Temps moyen pour un appel à la fonction logarithme sur un processeur Itanium (en cycle d'horloge).*

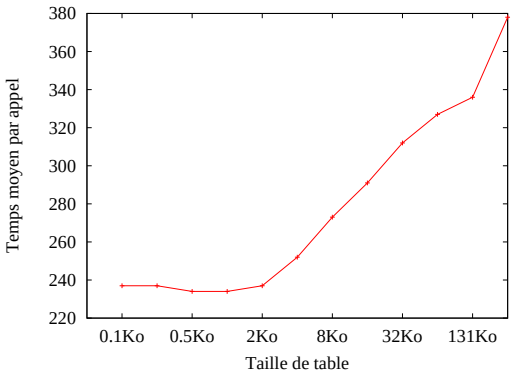


FIG. 3.6: *Temps moyen pour un appel à la fonction sinus sur un processeur PowerPC G4 (en unité arbitraire).*

la plus grande variabilité du degré du polynôme en fonction de la taille de table, pour la fonction logarithme que pour les fonctions trigonométriques. Ce qui signifie que pour de nombreux appels à une fonction, le nombre d'opérations flottantes nécessaire à l'évaluation polynomiale a un impact mesurable sur le temps d'exécution. Toutefois cet impact est moins important que celui dû aux nombreux défauts de caches rencontrés avec les méthodes basées sur des grandes tables (Figure 3.6). Les fonctions sinus/cosinus sont des fonctions qui s'approchent par un polynôme de petit degré contrairement à la fonction logarithme. L'implémentation des autres fonctions élémentaires ($\exp$, $\tan$, $\arctan$...) est similaire aux deux fonctions testées. Par conséquent, il devrait en être de même de leur comportement.

Le coût des défauts de caches

Impacts des défauts de caches

Sur chaque figure du tableau 3.5, le temps d'exécution des méthodes à base de petites tables est plus régulier que celui des méthodes à base de grandes tables. Ce phénomène est dû au nombre plus important de défauts de cache rencontrés par les méthodes à base de grandes

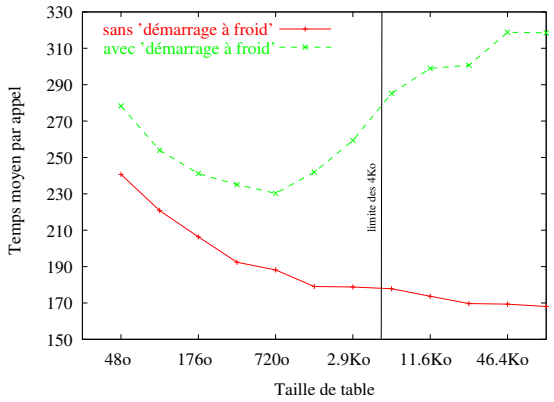


FIG. 3.7: *Temps moyen pour un appel à la fonction logarithme sur un processeur Pentium III. Le temps mesuré correspond au temps mis lorsque les données sont déjà présentes dans le cache avant l'appel de la fonction (graphique 'sans démarrage à froid'), et lorsque les données ne sont pas dans le cache (graphique 'avec démarrage à froid').*

tables. Ce nombre est plus important lorsque la table ne tient plus dans le cache de plus haut niveau. Ainsi, pour un nombre important d'appels à une fonction, les méthodes avec de petites tables sont à privilégier car elles présentent des résultats plus réguliers.

L'impact de la limite des 4 Koctets est plus visible sur la figure 3.6 qui présente de bonnes performances pour les méthodes utilisant des tables ayant une taille strictement inférieure à 4 Koctets. Le «strictement» vient du fait que dans la mesure de la taille de table utilisée (tableaux 3.3 et 3.4), nous ne considérons pas les variables flottantes qui occupent quelques dizaines d'octets par méthode.

Impact du démarrage à froid

La figure 3.7 compare le temps d'exécution pour 300 appels consécutifs à la fonction logarithme avec et sans démarrage à froid. La figure sans démarrage à froid peut être interprété comme le résultat d'un appel à une fonction dont les entrées ne sont pas corrélées. La figure correspondant à une application réelle (données plus ou moins corrélées, nombre d'appels plus ou moins fréquent ...) se trouvera quelque part entre les deux courbes avec une forme identique. Ainsi la meilleure méthode indépendamment de la fréquence d'appel sera celle utilisant moins de 4 Koctets de table, et 0.72 Koctets dans ce cas particulier (Pentium III, fonction $\log$).

3.4.3 Conclusion : 4 Koctets

Le choix de la taille de table optimale est fréquemment laissé comme un paramètre ne pouvant être maîtrisé. Toutefois à travers cette étude nous observons que les meilleures performances sont obtenues si la quantité de données utilisées pour la réduction d'argument ne dépasse pas le minimum entre la taille d'une voie du cache de plus haut niveau où sont stockés les flottants et la taille d'une page du système.

Les bibliothèques mathématiques que nous souhaitons construire sont destinées à s'exécuter sur un grand nombre d'architectures, où la taille moyenne d'une voie du cache est de 4 Koctets (Tab. 3.2). Comme nous l'avons constaté, l'efficacité de ces bibliothèques peut être améliorée (jusqu'à 37% sur nos comparaisons selon la fonction et la fréquence d'appel), simplement en optimisant la réduction d'argument des fonctions fréquemment appelées ensemble

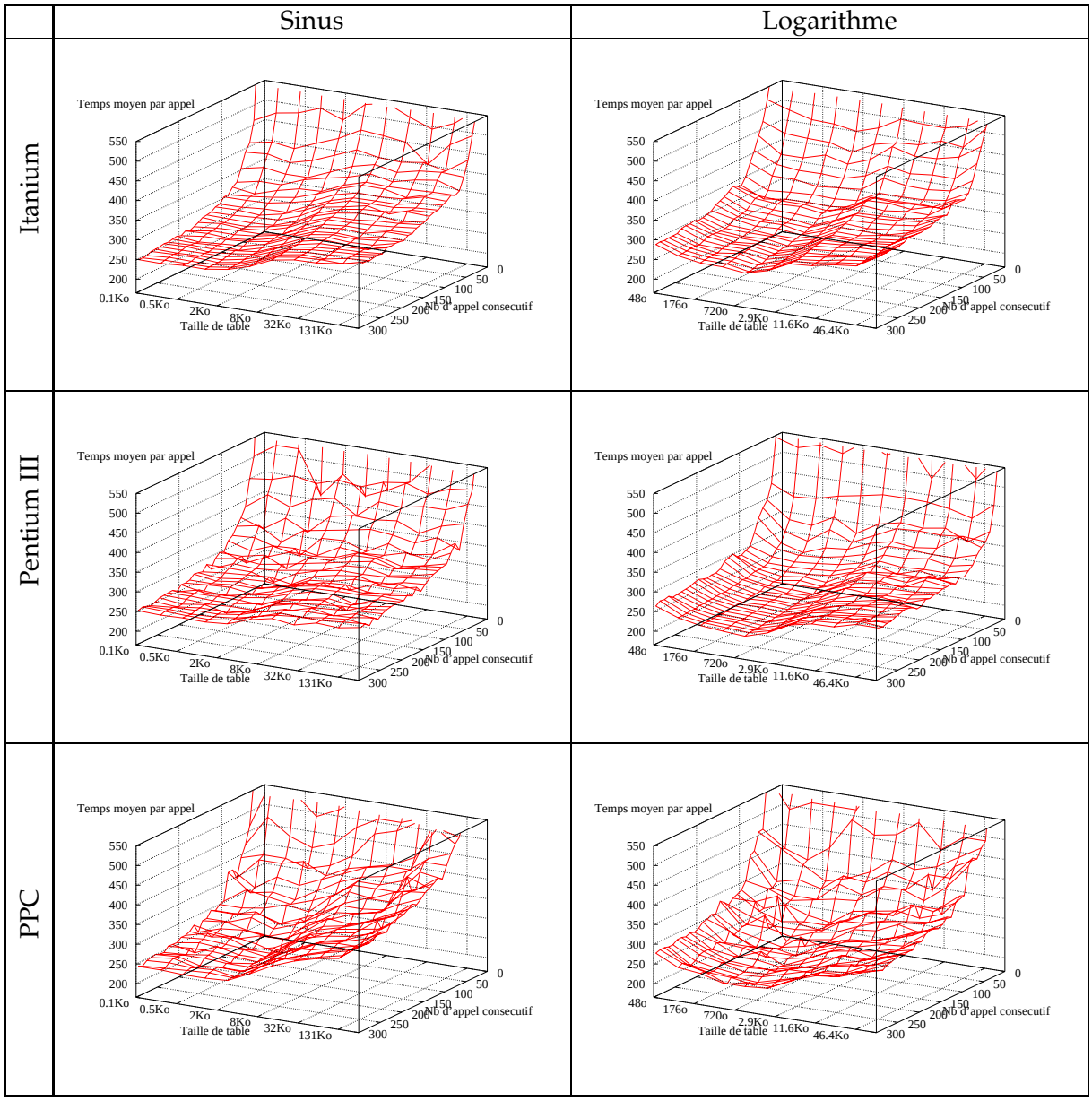
(par exemple sinus et cosinus) de façon à ce que cette étape ne prenne pas plus de 4 Koctets de mémoire. Cette borne de 4 Koctets est loin d'être optimale pour tous les systèmes. En contrepartie, elle garantit une certaine homogénéité des performances sur une majorité de systèmes, pour un grand nombre de fonctions à évaluer et pour une fréquence d'utilisation variable.

3.5 Conclusion

Les techniques d'optimisation numérique ont fait l'objet de nombreuses recherches. Aussi dans la course au dernier bit, nous en utilisons certaines déjà bien connues comme la gestion mémoire ou le déroulement de boucle, mais de nouvelles doivent être découvertes et exploitées.

Au cours de ce chapitre nous avons présenté des algorithmes déjà connus comme l'addition et la multiplication exactes. Nous avons également proposé une implémentation efficace des tests de l'arrondi correct pour les quatre modes d'arrondi IEEE, basée sur des opérations en arrondi au près. Enfin, nous finalisons cette étude par la présentation de résultats de tests portant sur la taille de la table à utiliser pour la réduction d'argument dans le processus d'évaluation des fonctions élémentaires.

Nous sommes encore loin d'avoir exploré l'ensemble des répercussions des caractéristiques des processeurs sur les algorithmes d'évaluation des fonctions élémentaires. L'apparition de nouveaux opérateurs encore peu exploités comme l'approximation de l'inverse, le FMA ou les instructions multimedia nous démontre qu'il reste de nombreuses optimisations à découvrir dans ce domaine.



TAB. 3.5: *Temps moyen pris pour 1 appel à la fonction sinus/logarithme (en cycle d'horloge pour Itanium, Pentium III, et dans une unité arbitraire pour le PPC).*

CHAPITRE

4

rempli nos exigences concernant les opérateurs multiprécision.

4.2 Les formats de représentation

La meilleure façon d'introduire nos travaux dans le domaine des bibliothèques multiprécision est de présenter des bibliothèques MP existant sur le marché, classées selon leurs représentations internes des nombres MP. Nous appellerons *chiffre* la brique de base utilisée pour la représentation interne des nombres. Le choix des chiffres est dirigé par les algorithmes les utilisant. Nous discuterons également des avantages et inconvénients respectifs de chaque format.

4.2.1 Somme de nombres flottants

Les bibliothèques MP *double double* de Briggs [15], *quad doubles* de Bailey [43] et les *expansions* flottantes de Daumas [24], utilisent une somme non évaluée de plusieurs nombres flottants double précision pour la représentation interne des nombres MP. Les opérations arithmétiques sur ce format sont basées uniquement sur les opérations flottantes. Ce choix est motivé par le nombre croissant d'unités flottantes (FPU) proposées par les processeurs modernes. Par exemple dans les dernières architectures 64 bits comme l'UltraSparc [46] et l'Itanium [76], les multiplications flottantes sont plus rapides que leurs homologues entiers (voir tableau 3.1 page 42). De plus les processeurs superscalaires ont plusieurs de ces unités, et la tendance a été de se focaliser plus fortement sur le calcul flottant pour inclure de plus en plus de ces unités dans les processeurs. Néanmoins il n'est pas clair que cette tendance se poursuive, comme le prouve une récente comparaison des processeurs dans [20].

Un avantage d'un tel format est de rendre triviales les conversions entre le format MP et le format IEEE-754. En revanche, les additions et multiplications sont relativement complexes à cause des erreurs d'arrondi inhérentes aux nombres flottants. Ces opérations nécessitent des algorithmes sophistiqués devant être prouvés avec attention, et parfois l'utilisation de prouveurs automatiques [13].

4.2.2 Représentation en grande base «dense»

Ce format utilise une base standard des nombres MP, où les chiffres sont des nombres machines. Ils peuvent être des nombres flottants comme dans la bibliothèque MPFUN de Bailey [11], ou des entiers comme dans le cas de la bibliothèque GNU Multi Précision (GMP) [1], audessus de laquelle sont construites plusieurs autres bibliothèques (voir MPFR [3] ou Arithmos [22]). Dans GMP les chiffres sont les entiers du format natif de la machine (actuellement 32 ou 64 bits). Ainsi la base de représentation des nombres est 2^{32} ou 2^{64}, d'où le nom *grande base*.

Les opérations arithmétiques sur ce format utilisent des algorithmes similaires à ceux que l'on apprend à l'école en base 10. La différence est la base sur laquelle ces opérations sont effectuées qui correspond à la largeur d'un mot machine. Pour les même raisons que pour les algorithmes en base 10, le résultat de ces opérations arithmétiques sur les chiffres n'est pas représentable par un mot machine. Des arrondis se produisent dans le cas de flottants et des dépassements de capacité dans le cas d'entiers. De ce fait, les algorithmes doivent être écrits pour gérer les phénomènes d'arrondi ou de débordement. En général, cela est traduit au niveau algorithmique par l'ajout d'une étape de *propagation de retenue* avec des conséquences importantes sur le temps d'exécution des algorithmes MP. Dans ce format les conversions entre format MP et flottant double précision sont parfois coûteuses.

4.2.3 Représentation en grande base «creuse»

Dans un tel format les nombres MP sont codés comme un vecteur d'entiers ou de nombres flottants, que nous appellerons également *chiffres*. Néanmoins, pour éviter des propagations de retenue, ce format garantit qu'aucune précision n'est perdue lors d'additions ou de multiplications de deux chiffres. Pour cela les chiffres n'utilisent pas toute la précision disponible dans les formats machine utilisés. Certains bits sont réservés et destinés à contenir les propagations de retenues intermédiaires. Nous appellerons ce format *Carry-Save*, nom dérivé d'une idée similaire exploitée par les opérateurs matériels [58, 70].

L'histoire de ce format est intéressante. Il a été utilisé en premier dans la bibliothèque MP de Brent [14] basée sur des entiers. Ses motivations semblaient être reliées à des problèmes de portabilité de code. En effet, bien que les processeurs offrent depuis longtemps des instructions machine facilitant la tâche des logiciels multiprécisions (tels que *add-with-carry*), nombre de ces instructions ne pouvaient et ne peuvent toujours pas être accédées à partir d'un langage de haut niveau tel que C. Comme nous le verrons par la suite, le format Carry-Save s'abstrait de ce problème en utilisant seulement des instructions arithmétiques simples, et par conséquent portables.

La bibliothèque MP de Brent a été peu à peu remplacée par GMP, qui offre une réponse différente au problème de la portabilité. Dans GMP, le cœur de chaque boucle des additions et multiplications est écrit en assembleur pour la plupart des architectures existantes, ce qui lui permet de tirer le meilleur parti des instructions machine non disponibles en langage de haut niveau.

Néanmoins, l'idée de Brent a refait surface récemment. Il semble que cette idée ait été reprise par Ziv dans la bibliothèque mathématique d'IBM avec arrondi correct [2], en utilisant des flottants comme brique de base. Malheureusement, il n'existe à notre connaissance aucun article à ce sujet. Indépendamment, nous avons développé la représentation SCS présentée en section 4.4 qui utilise indifféremment les entiers ou les flottants. Nos motivations étaient la portabilité, mais également l'efficacité. Nous avons constaté que le format MP Carry-Save permettait de ne pas avoir à se soucier des propagations de retenue ce qui rendait les algorithmes simples, et exploitait ainsi de façon naturelle l'important parallélisme pouvant être traité par les processeurs modernes.

4.3 Présentation des algorithmes MP standards

Dans le reste, nous noterons m le nombre de bits utilisés pour mémoriser un chiffre dans le format MP considéré. Dans GMP, les chiffres sont des entiers machines et par conséquent $m = 32$ ou $m = 64$. Dans la bibliothèque de Bailey il s'agit de flottants, donc $m = 24$ ou $m = 53$. Nous noterons que ces représentations utilisent une base de type 2^m, alors que certains produits commerciaux utilisent une puissance de 10 essentiellement pour des raisons historiques.

4.3.1 L'addition et la propagation de retenue

Avec des nombres entiers, l'algorithme d'addition le plus simple est semblable à celui appris à l'école pour ajouter deux nombres en base 10. Il consiste à additionner des paires de chiffres de poids identique. La somme de deux chiffres dans une certaine base peut déborder ou créer une retenue, ce qui signifie que le résultat est plus grand que la base utilisée. Dans ce cas, le résultat doit être découpé en deux chiffres : le premier qui est le résultat modulo la

base, et le deuxième qui correspond à la *retenue* et qui doit être additionné avec les chiffres immédiatement à gauche.

L'algorithme naïf présente deux inconvénients majeurs. Le premier est qu'il faut être capable de calculer la représentation sur deux chiffres, de la somme de deux chiffres et d'une retenue. Pour être efficace, cette opération doit utiliser l'instruction assembleur *add-with-carry*, ce que fait GMP qui dispose de routines avec cette instruction pour chaque architecture. Le second problème est lié au caractère séquentiel de l'opération d'addition qui s'effectue de droite à gauche à cause des propagations de retenue. Nous montrerons par la suite que ce problème limite l'utilisation de toutes les capacités offertes par les processeurs modernes.

4.3.2 La multiplication

L'algorithme de multiplication le plus simple est décrit sur la figure 4.1. Il est semblable à celui appris à l'école pour la multiplication en base 10. Il existe également d'autres algorithmes offrant une meilleure complexité asymptotique, par exemple la multiplication rapide de Karatsuba [55, 57]. Toutefois, ils ne sont généralement intéressants que pour des précisions supérieures à celle que nous considérons.

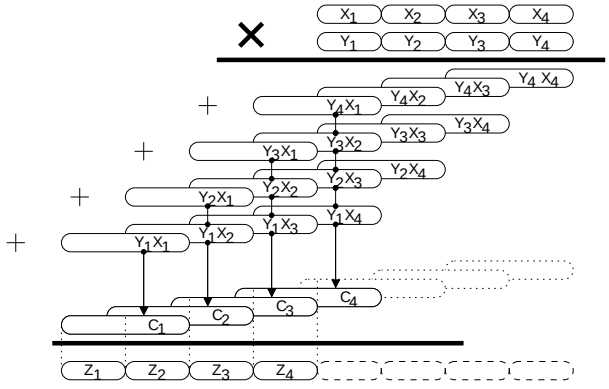


FIG. 4.1: *Multiplication MP*

La figure 4.1 représente les deux nombres X et Y que l'on souhaite multiplier. Ils sont respectivement composés de n chiffres x_i et y_i (avec $n = 4$ sur la figure) avec chaque chiffre codé sur m bits de précision.

L'algorithme de multiplication inclut trois étapes. En premier, un tableau de produits partiels $x_i y_j$ codés sur $2m$ bits est calculé. Ensuite ces produits partiels doivent être sommés verticalement et le résultat exprimé en chiffres. Il y a au plus n produits partiels dans une colonne, la somme c_i de chaque colonne est un nombre d'au plus $2m + \lceil \log_2 n \rceil$ bits. Finalement, les colonnes de somme c_i sont décomposées en nombres de seulement m bits. Cette opération est effectuée itérativement en sommant de droite à gauche les chiffres de poids équivalent. Les m bits de poids faible du résultat sont mémorisés et la partie haute est utilisée comme une retenue pour l'itération suivante.

Cette représentation exhibe énormément de parallélisme. En effet tous les produits partiels peuvent être calculés en parallèle et il en est de même pour les colonnes. Le choix de la représentation interne des chiffres est crucial car il détermine la quantité d'instructions séquentielles.

Il est clair que chaque implémentation qui utilise comme chiffre des nombres machines (entiers ou flottants) ne peut représenter avec un seul mot machine les produit partiels, et encore moins la colonne de somme. Des astuces algorithmiques sont donc nécessaires pour obtenir le résultat représenté sur deux ou plusieurs chiffres. Ces astuces impliquent en retour des dépendances entre les opérations inhibant le parallélisme. Une solution, utilisée par GMP, est de calculer ligne après ligne les produits partiels, en convertissant chaque somme en un nombre MP. Cette démarche algorithmique fonctionne comme une propagation de retenue en grande base, ce qui introduit donc de la séquentialité.

4.3.3 Utilisation de nombres flottants en multiprécision

Le même type de discussion peut être tenu lorsque le format de représentation des nombres multiprécision est basé sur les flottants. La somme et le produit peuvent engendrer un dépassement de capacité et une perte de précision lors de l'arrondi du résultat imposé par la norme IEEE-754. Néanmoins, il existe des algorithmes, qui sous certaines conditions peuvent représenter la somme ou le produit de deux nombres flottants de manière exacte sur deux nombres flottants [57]. Parmi ces algorithmes, nous avons détaillé l'addition et la multiplication exacte en section 3.2, dont les coûts sont de 3 additions flottantes, 2 masques et 1 test entier pour l'addition et de 13 multiplications flottantes, 10 additions flottantes et 4 tests entiers pour la multiplication. Le caractère portable des algorithmes basés sur la norme IEEE-754 constitue la principale motivation pour utiliser cette représentation. Malheureusement, ces algorithmes s'avèrent souvent coûteux en terme de ressources utilisées et impliquent de nombreuses dépendances de données.

4.4 Multiprécision en représentation 'Software Carry-Save'

Cette section décrit les caractéristiques du format SCS que nous avons proposé. L'idée principale est de garantir que tous les calculs intermédiaires des algorithmes MP sont exacts. En d'autres termes, le résultat d'un calcul intermédiaire doit tenir sur un mot machine sans erreur d'arrondi ou sans débordement.

Si nous représentons par M la précision en bits offerte par le format machine considéré, par m la précision utilisée pour coder un chiffre ($m \leq M$) et par n le nombre de chiffres internes au format MP, alors une condition suffisante est :

$$M \geq 2m + \lceil log_2(n) \rceil \tag{4.1}$$

Cette équation exprime le compromis du format SCS entre nombre de chiffres et précision interne utilisée par ces chiffres. Nous constatons que ce format nécessite plus de chiffres qu'une représentation dite dense pour une précision donnée. Ainsi lorsque qu'un nombre MP est stocké en mémoire, plus de la moitié des bits sont connus comme étant nuls. Il faut également effectuer plus d'opérations atomiques dans le format SCS pour la même précision. Néanmoins, cet inconvénient est largement compensé par l'avantage de ne plus avoir de gestion de retenue, ce qui simplifie les opérations.

Nos motivations étaient d'exprimer les algorithmes d'addition et de multiplication avec plus de parallélisme entre les opérations atomiques qu'il n'était possible avec le format de représentation utilisé par GMP. Cette motivation est identique à celle que l'on retrouve dans la

Mots Machine	M	m	n	précision max.
IEEE double précision	52+1 bits	25 bits	7 termes	175 bits
IEEE double précision	52+1 bits	24 bits	31 termes	744 bits
Double étendue (x86)	64 bits	29 bits	63 termes	1827 bits
Double étendue (x86)	64 bits	28 bits	255 termes	7140 bits
Entier 64 bits	64 bits	30 bits	15 termes	450 bits
Entier 64 bits	64 bits	29 bits	63 termes	1827 bits

TAB. 4.1: *Quelques exemples de précision atteignable avec le format SCS.*

représentation "Carry-Save" utilisée en arithmétique des ordinateurs bien que la base considérée soit plus petite. En supprimant les propagations de retenue, ou plus exactement en les repoussant, cette technique supprime également les dépendances de données liées à cette étape.

4.4.1 Compromis sur la précision

La précision globale est de $m \times n$ bits et quelques exemples de valeurs sont données dans le tableau 4.1. La précision correspond à la précision maximale pouvant être atteinte pour un m donné, mais il est possible d'obtenir une précision inférieure en utilisant un nombre n de chiffres plus petit. Par exemple, comme nous le verrons au chapitre 5, la bibliothèque mathématique avec arrondi correct utilise une précision de 240 bits ($m = 30$, $n = 8$).

Nous noterons que Brent [14] impose une condition plus faible. Il impose que 8 produits partiels puissent être sommés sans erreur d'arrondi. Par conséquent, il doit toujours effectuer une propagation de retenue toutes les 8 multiplications, avec un format permettant d'avoir une précision arbitraire. En revanche, notre approche est plus efficace, mais ne permet pas de disposer d'une précision arbitraire.

4.4.2 L'addition et la multiplication MP utilisées par SCS

L'algorithme de multiplication utilisé par SCS est très simple. Nous calculons en premier la colonne de somme sans se soucier des possibles débordements/arrondis. Nous terminons par une propagation de retenue.

Calculer la colonne de somme sans débordement et sans arrondi est la partie centrale de l'algorithme. Cette partie s'exprime facilement de façon portable, efficace et avec beaucoup de parallélisme en langage de haut-niveau. La propagation de retenue finale est la seule partie séquentielle de l'algorithme, mais il n'y a pas de pénalité en terme de temps d'exécution à exprimer cette partie en langage de haut niveau. La propagation de retenue finale est sur plusieurs bits, et il n'existe pas d'instructions assembleur comparables à l'instruction "add-with-carry" pour les retenues sur 1 bit. En arithmétique entière, nous découpons un nombre machine en m bits plus une retenue en utilisant des masques et des décalages. Lorsqu'il s'agit d'arithmétique IEEE-754 alors nous utilisons des multiplications par une puissance de deux (opérations exactes) et des soustractions.

Les additions MP sont composées d'additions de paires de chiffres de poids identique et d'une propagation de retenue. Elles ne sont par conséquent pas plus rapides que les additions MP standard et sont même plus lentes si elles sont exprimées en langage de haut niveau. Néan-

moins, le format permet l'addition de plusieurs nombres avec une seule propagation de retenue à la fin, mais il nous reste à trouver une utilisation pratique de cette fonctionnalité.

Finalement, il est aussi possible d'effectuer des *FMA* (une addition et une multiplication en une seule opération), ce qui permet d'accélérer les évaluations polynomiales.

4.5 Implémentation machine de la bibliothèque SCS

4.5.1 Implémentation initiale

Nous avons implémenté l'idée SCS dans une première version de notre bibliothèque [29]. Cette version préliminaire utilisait une représentation interne basée indifféremment sur des entiers ou des flottants. Les algorithmes utilisés étaient semblables avec une légère différence sur la façon de décomposer un mot machine entre un chiffre et une retenue lors de la propagation de retenue.

Dans la première version, des routines entières et flottantes étaient utilisées, avec une précision finale de 200 bits et avec un effort d'optimisation modéré. En effet le code était petit et simple, ce qui permettait d'obtenir des optimisations de base telles que le déroulement de boucle (ce qui devrait être laissé au compilateur dans les versions futures). Dans de rares cas, l'assembleur généré par le compilateur a été vérifié de façon à modifier le code C pour aider l'ordonnancement effectué par le compilateur.

4.5.2 Analyse et améliorations

4.5.2.1 Les performances entières dépassent les flottantes

En analysant les premiers résultats de l'implémentation initiale, nous avons observé que l'implémentation entière était pratiquement toujours plus rapide que l'implémentation flottante. Cette constatation est cohérente avec l'intuition que l'on peut se faire à la vue des performances respectives des unités entières et flottantes (voir section 3.1.1). En effet les processeurs superscalaires offrent plus d'unités de calcul entier que flottant (bien que toutes ces unités ne soient pas capables d'exécuter des multiplications). La seule exception est la famille des UltraSPARC I et II, sur laquelle la multiplication entière 64 bits est particulièrement inefficace.

Nous avons par conséquent concentré nos efforts sur l'implémentation entière de SCS, mais ce choix pourra être revu en fonction des évolutions technologiques futures.

4.5.2.2 L'utilisation d'une arithmétique mixte 32 et 64 bits

Une autre amélioration que nous avons inclue dans SCS fut l'utilisation d'une arithmétique entière sur 32 ou 64 bits de la façon suivante :

- Les chiffres MP sont stockés sur des nombres de 32 bits où seuls quelques bits sont réservés pour les retenues. Nous avons ainsi résolu le problème majeur de la version initiale [29] concernant les mouvements mémoire qui étaient relativement inefficaces.
- L'addition utilise une arithmétique 32 bits.
- Dans la multiplication MP, les produits partiels sont des produits de deux nombres de 32 bits, ce qui correspond à un nombre sur 64 bits. La colonne de somme nécessite donc l'utilisation d'une arithmétique sur 64 bits. Ceci peut s'exprimer en langage C de la façon suivante : un nombre codé sur 64 bits de type «`long long int`» est multiplié à un nombre

codé sur 32 bits de type «`unsigned int`». Lors de cette multiplication, le deuxième chiffre sur 32 bits est promu en entier sur 64 bits. Cette technique permet d'effectuer des multiplications efficaces de deux mots de 32 bits et de récupérer le résultat dans un mot de 64 bits.
Pour la famille des UltraSPARC (détectée au moment de la compilation) nous utilisons une conversion vers les flottants de l'un des deux multiplicandes.

Comme nos expériences vont le montrer, cela fonctionne particulièrement bien sur les processeurs actuels qui possèdent une arithmétique 64 bits ou qui offrent les instructions permettant de stocker sur deux mots de 32 bits le résultat codé sur 64 bits d'un produit de deux mots sur 32 bits. Le reste est laissé au compilateur, ce qui est trivial (promotion en 64 bits d'un entier codé sur 32 bits, et l'addition 64 bits utilisant l'instruction *add-with-carry*).

4.5.3 Implémentation de SCS

Cette section donne les détails de l'implémentation actuelle de SCS. La principale différence avec la précédente présentation est que nous utilisons une représentation SCS pour les mantisses d'un nombre flottant MP au format SCS. Pour clarifier une possible confusion, les nombres représentés sont des nombres flottants MP, mais les chiffres qui les composent sont toujours des entiers machines. Cette implémentation est similaire au format virgule flottante de `mpf` et `mpfr` de GMP.

Une motivation du format SCS décrit ci-dessous est d'être compatible avec le standard IEEE-754, sur la gestion des cas exceptionnels et des arrondis lors des conversions. Ces caractéristiques sont indispensables à l'utilisation de *scslib* pour la deuxième partie de l'évaluation des fonctions élémentaires au sein de la bibliothèque d'évaluation de fonctions élémentaires avec arrondi correct *crlibm* (voir chapitre 5).

4.5.3.1 Le format

Un nombre MP est représenté dans le format proposé comme une structure R de type *Software Carry-Save* (SCS), comme décrit sur la figure 4.2. Il est composé des champs suivants :

R.digits[n***]*** Une table de n entiers machine avec m bits de précision (ce paramètre peut être choisi au moment de la compilation) ;

R.index Un entier stockant l'index du premier chiffre dans la plage des nombres représentables, comme décrit sur la figure 4.2. Ce champ joue le rôle de l'exposant.

R.sign L'information de signe (1 si positif -1 si negatif).

R.exception Un nombre flottant IEEE-754 permettant de gérer les cas exceptionnels (± 0, $\pm\infty$, Not-A-Number). Ce champ est défini au moment de la conversion d'un nombre flottant IEEE à 1.0 si le nombre n'est pas une exception et à l'exception elle-même sinon. Chaque opération effectuée sur les nombres MP est effectuée sur ce champ.

Ainsi la valeur x de la représentation R est dans ce cas :

$$x = R.sign \times \sum_{j=0}^{n-1} R.digits[j] \times 2^{m(R.index-j)} \tag{4.2}$$

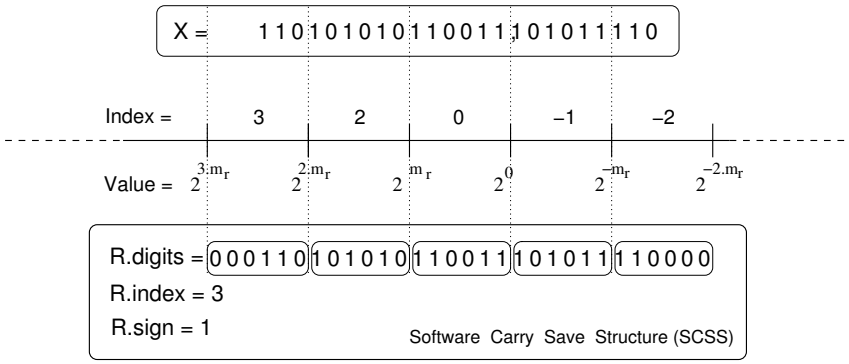


FIG. 4.2: *Le format proposé*

Dans la suite de ce chapitre, nous appellerons un nombre SCS *normalisé* un nombre SCS où tous les bits de retenue (les bits de M à m du champs $R.digits$) sont à zéro. Un nombre SCS où ces bits ne sont pas à zéro est dit *non-normalisé*.

Nous imposons à tous les chiffres d'être alignés sur une frontière multiple de m sur l'échelle des exposants (voir figure 4.2). Cela correspond dans l'équation (4.2) à un exposant multiple de m. C'est un choix d'implémentation : une autre solution aurait été d'utiliser un entier arbitraire comme exposant, mais dans ce cas, l'addition MP aurait été plus complexe car des décalages entre chiffres du même nombre MP auraient été nécessaires. Avec cette implémentation ils doivent également être décalés si leurs index sont différents, mais cette opération s'effectue seulement sur l'indice de tableau de $R.digits$.

Ce choix a un impact sur la précision. Comme dans la plupart des formats flottants, nous imposons au chiffre de poids fort d'être différent de zéro, pour assurer l'unicité de la représentation pour les nombres SCS normalisés. Par conséquent, ce premier chiffre peut avoir dans le pire cas seulement 1 bit d'information sur m bits disponibles. Cette propriété implique que la précision minimum garantie est de $m-1$ bits inférieure à celle annoncée dans le tableau 4.1. Par exemple, pour l'évaluation des fonctions élémentaires en double précision avec arrondi correct nous avons choisi $m = 30$ et $n = 8$, et la précision garantie est de 211 bits.

Pour un champ index codé sur 32 bits, la plage d'exposants utilisables est grande comparée à celle utilisée dans le format double précision, qui n'utilise que 11 bits pour coder l'exposant. Cette fonctionnalité est très importante car elle permet de garantir que l'ensemble des nombres représentables par SCS inclut l'ensemble des nombres flottants définis par le format IEEE-754. Cette ensemble est également composé des *nombres dénormalisés* du moment que la taille de la mantisse SCS est supérieure à celle du format IEEE-754 (dans le cas contraire, une bibliothèque multiprécision n'est pas utile). Cette grande plage d'exposants constitue donc un atout par rapport aux bibliothèques basées sur les flottants telles que celle de Bailey ou celle de Daumas qui sont limitées aux plages d'exposants accessibles par les flottants double précision $(2^{-1074}, 2^{1024})$. Nous évitons pratiquement tous les possibles dépassements de capacité qui se produisent en utilisation courante sur les calculs intermédiaires.

D'autres cas exceptionnels (± 0, $\pm\infty$, Not-A-Number) sont stockés comme leurs représentations en flottant double précision défini par la norme IEEE-754. Nous laissons ainsi la gestion des exceptions au système, de façon à assurer les propriétés suivantes :

Propriété 4 *Si la mantisse SCS dispose de plus de précision qu'une mantisse d'un nombre flottant*

double précision (53 bits), et si l'association compilateur/architecture respecte la norme IEEE-754 pour ce qui est de la gestion des cas exceptionnels alors :

- *l'ensemble des nombres flottants IEEE-754 double précision est un sous-ensemble des nombres représentables par le format SCS,*
- *les conversions entre SCS et format IEEE-754 double précision gèrent les dénormalisés ainsi que les exceptions,*
- *Toutes les opérations MP fournies par la bibliothèque SCS gèrent les exceptions comme en arithmétique IEEE.*

4.5.3.2 Opérations fournies par la bibliothèque SCS

Autant que possible la bibliothèque développée utilise une syntaxe similaire à celle utilisée par GMP.

- Fonctions de conversion et d'initialisation :
  - Conversion d'un flottant double précision en nombre SCS :
    `void scs_set_d(scs_ptr, double)`
  - Conversion d'un entier signé en nombre SCS :
    `void scs_set_si(scs_ptr, signed int)`
  - Conversion d'un nombre SCS en flottant double-précision, avec arrondi au plus près, vers $-\infty$, vers ∞ ou vers 0 :
    `void scs_get_d(double*, const scs_ptr)`
    `void scs_get_d_minf(double*, const scs_ptr)`
    `void scs_get_d_pinf(double*, const scs_ptr)`
    `void scs_get_d_zero(double*, const scs_ptr)`
  - Créer un nombre SCS égal à zéro :
    `void scs_zero(scs_ptr)`
  - Créer un nombre SCS aléatoire (sans garantie sur la loi aléatoire utilisée). L'entier définit la pages des exposants possibles (exposant compris entre - l'entier et + l'entier) :
    `void scs_rand(scs_ptr, int)`
  - Créer une copie d'un nombre SCS :
    `void scs_set(scs_ptr, const scs_ptr)`
  - Imprimer la structure d'un nombre SCS :
    `void scs_get_std(const scs_ptr)`
- Fonctions arithmétiques
  - Addition et soustraction retournant un résultat normalisé :
    `void scs_add(scs_ptr result, scs_ptr x, scs_ptr y)`
    `void scs_sub(scs_ptr result, scs_ptr x, scs_ptr y)`
  - Multiplication et carré :
    `void scs_mul(scs_ptr result, scs_ptr x, scs_ptr y)`
    `void scs_square(scs_ptr result, scs_ptr x)`
  - Multiplication par un entier non-signé :
    `void scs_mul_ui(scs_ptr, unsigned int)`
  - Inverse et division :
    `void scs_inv(scs_ptr result, scs_ptr x)`
    `void scs_div(scs_ptr result, scs_ptr x, scs_ptr y)`

D'autres fonctions devraient être incluses dans de prochaines versions de la bibliothèque (*FMA*, addition sans normalisation, renormalisation seule).

De plus, la bibliothèque fournit une interface C++ permettant d'utiliser le format SCS de façon transparente. Le surcoût de ces traducteurs C++ est très dépendant du couple processeur/compilateur et est mesuré dans la suite.

4.5.3.3 Considérations spécifiques à l'implémentation

Pour des raisons de portabilité, l'implémentation utilise du C99 ANSI combiné à une version récente du compilateur `gcc`. Nous n'avons pas pu trouver un cas où un compilateur fourni par les fabricants de processeur (Intel ou Sun) donnait des différences de performance significatives par rapport à `gcc`, ce qui est probablement la conséquence de la simplicité de notre code.

Lorsque nous cherchons à optimiser les performances, nous avons observé que le même code qui était efficace sur un processeur pouvait avoir un comportement inverse sur un autre. D'habitude, cette différence vient des caractéristiques du processeur lui-même, l'UltraSparc II et la faiblesse de sa multiplication entière en sont un exemple. Le processeur n'est pas le seul fautif et le compilateur doit parfois être remis en cause. Un exemple est la promotion d'un entier de 32 bits vers un entier de 64 bits dans l'algorithme de multiplication. Dans ces deux cas nous avons essayé de changer le style de programmation de façon à ce que le code se comporte bien sur tous les processeurs, ce qui n'était parfois pas possible. Dans ces rares cas, en plus d'une version générique, nous avons développé plusieurs versions astucieuses de l'opération critique en fonction des processeurs. Ces versions sont sélectionnées automatiquement à la compilation grâce aux outils GNU `autoconf/automake`.

Plus surprenant, nous avons été très déçus par les capacités des compilateurs, et plus précisément sur une technique connue depuis de longues années : le déroulement de boucles. Notre code est constitué de nombreuses petites boucles `for` dont la taille est connue au moment de la compilation. Ce cas correspond à la configuration idéale pour le déroulement de boucles. Des options existent pour utiliser cette optimisation. Malheureusement, laisser le compilateur dérouler les boucles par lui-même conduit à de mauvaises performances, même comparées à la version non déroulée. Dérouler les boucles à la main ne prend que quelques minutes, nous l'avons donc fait pour la version que nous utilisons pour évaluer les fonctions élémentaires ($m = 30$, $n = 8$), ainsi que pour d'autres valeurs caractéristiques. Cette technique augmente peu la taille du code, et conduit parfois à diviser le temps d'exécution par deux. Bien sûr cela n'est pas satisfaisant, et nous ne voulons pas le faire pour toutes les valeurs possibles de n, ni étudier les compromis liés à l'architecture des processeurs au fur et à mesure de l'augmentation de n. Nous espérons que les futures versions des compilateurs auront un meilleur comportement.

4.6 Résultats et mesures de temps

Il est très difficile de comparer les performances des bibliothèques MP. Le coût d'une multiplication ou addition machine est très différent d'un processeur à un autre. De plus le coût dépend fortement du contexte d'exécution de ces instructions dans les processeurs d'aujourd'hui munis de profonds pipelines, d'unités d'exécutions dans le désordre, et d'une hiérarchie de cache. Même la définition d'une opération multiprécision est dépendante de la machine (32 ou 64 bits, disponibilité ou non du FMA). Malgré tous ces problèmes, nous avons testé la bibliothèque sur les configurations suivantes :

– *Pentium III* avec Debian GNU/Linux, gcc-2.95, gcc-3.0, gcc-3.2
– *UltraSPARC IIi* avec SunOS 5.8 et gcc-2.95
– *Pentium 4* avec Debian GNU/Linux, gcc-2.95, gcc-3.0, gcc-3.2
– *PowerPC G4* avec MacOS 10.2 et gcc-2.95
– *Itanium* avec Debian GNU/Linux, gcc-2.95, gcc-3.0, gcc-3.2

Lorsqu'il était disponible, nous avons utilisé `gcc-3.x` qui donnait des résultats légèrement meilleurs que `gcc-2.95`. Les résultats pour les autres compilateurs testés (Sun et Intel) ne sont pas présents dans cette étude car ils ne fournissaient pas de nettes améliorations.

4.6.1 La mesure du temps de bibliothèques MP

Les graphiques suivants comparent les performances de notre bibliothèque avec une précision garantie de 211 bits avec deux autres bibliothèques travaillant sur une précision similaire : la bibliothèque quad-double de Bailey [43], et celle de Ziv [2]. Nous avons également utilisé la bibliothèque de référence en matière de multiprécision qu'est GMP [1] (plus précisément MPF, la version flottante de GMP). Chaque résultat est obtenu en mesurant le temps d'exécution sur 10^3 valeurs aléatoires (les mêmes valeurs étant utilisées pour chaque bibliothèque). Les exposants utilisés pour les tests étaient construits de telle sorte qu'il existait pratiquement toujours un chevauchement des mantisses, pour rendre les tests sur l'addition réalistes. De plus pour minimiser l'impact des interruptions systèmes, plusieurs tests ont été effectués en ne gardant que le meilleur temps à chaque fois. L'impact des caches de données a été réduit en pré-chargant les caches avec les données utilisées.

Nous avons mesuré le temps d'exécution de la multiplication, de l'addition et des conversions entre format MP et flottant double précision. Nous avons également testé toutes ces bibliothèques avec l'évaluation de la fonction logarithme avec arrondi correct pour la double précision.

Pour des raisons de clarté du document, nous avons normalisé les temps par rapport au temps SCS pour chaque fonction et chaque architecture testées. Les barres représentent donc un temps relatif et une barre absente signifie qu'une erreur s'est produite au moment de la compilation ou de l'exécution.

4.6.2 Commentaires

Un premier coup d'œil sur ces graphiques nous montre que la supériorité des performances de SCS sur les autres bibliothèques semble s'accroître avec chaque génération de nouveau processeur. La principale raison est probablement liée aux pipelines qui deviennent de plus en plus profonds, qui sont par conséquent mieux remplis par une bibliothèque exhibant plus d'opérations indépendantes. En effet nous avons vu que les dépendances entre opérations pouvent créer des trous dans les pipelines lorsqu'une instruction attend le résultat d'une instruction précédente. Lorsque l'on y regarde de plus près on remarque que les bibliothèques qui souffrent le plus des effets de ces profonds pipelines sont la bibliothèque quad-double de Bailey et celle d'IBM. En revanche la bibliothèque GMP 4.1 prend en compte les caractéristiques des pipelines, comme l'indiquent les commentaires dans le code source [1].

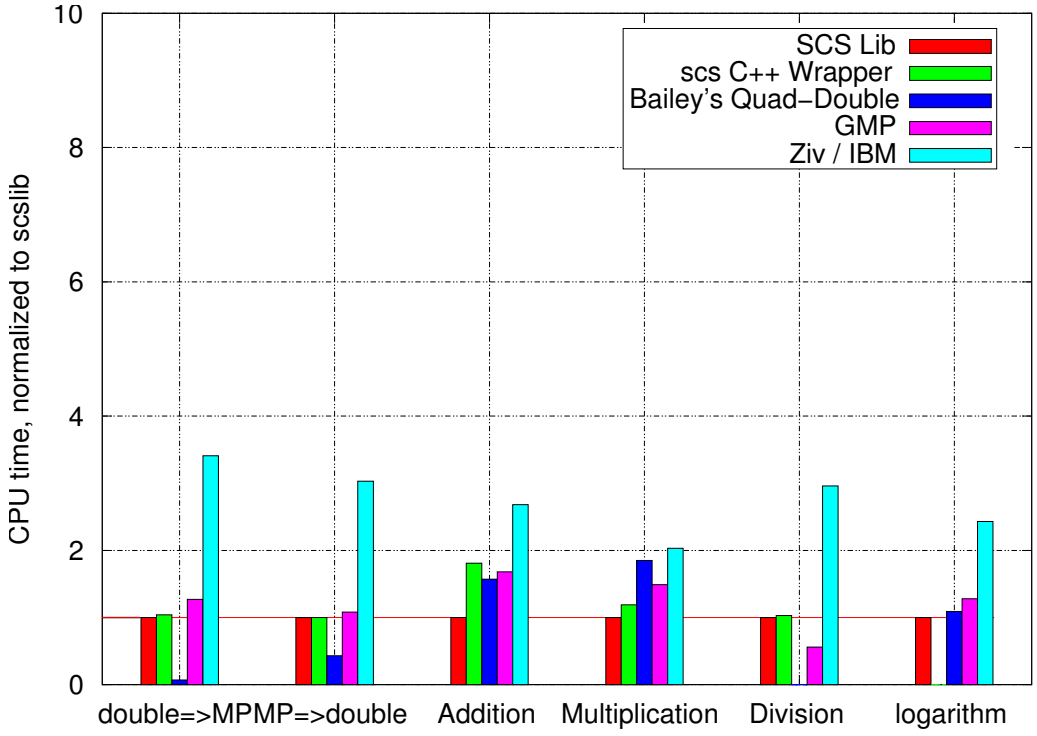


FIG. 4.3: *Comparaison sur un Pentium III*

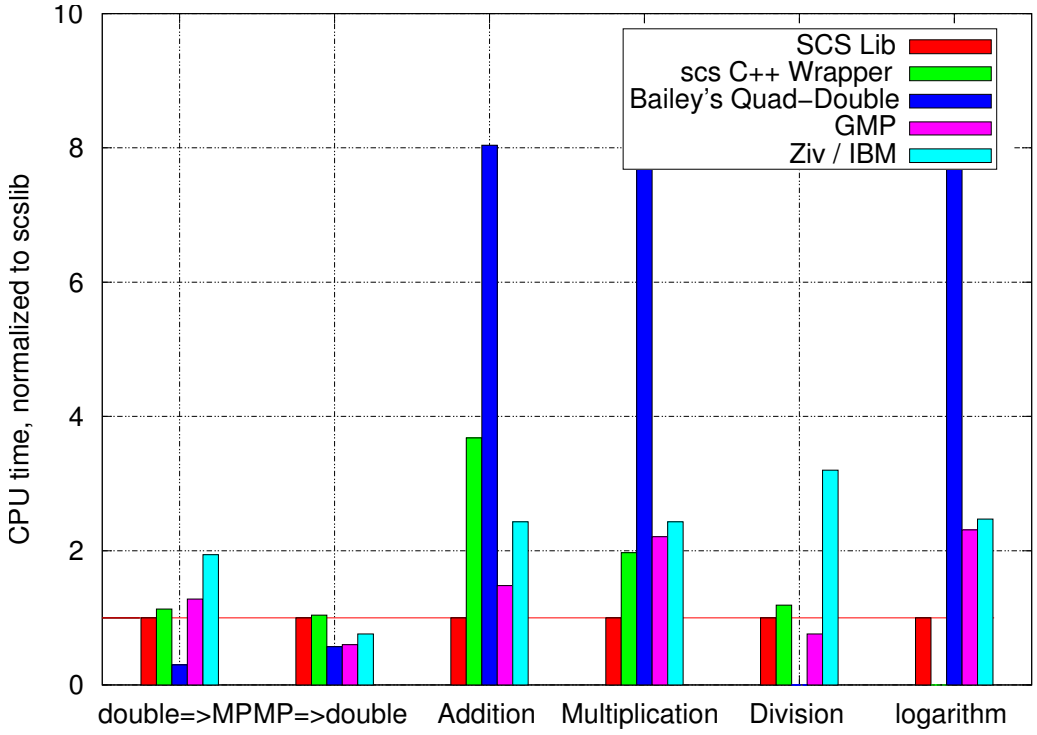


FIG. 4.4: *Comparaison sur un UltraSPARC IIi*

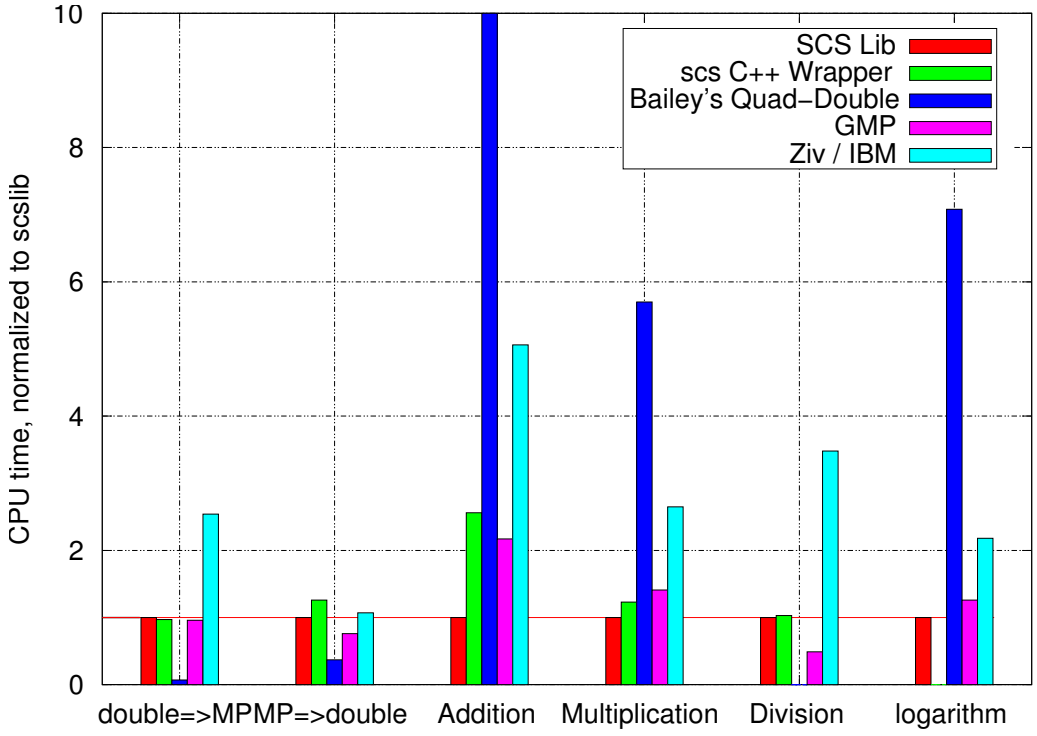


FIG. 4.5: *Comparaison sur un Pentium 4*

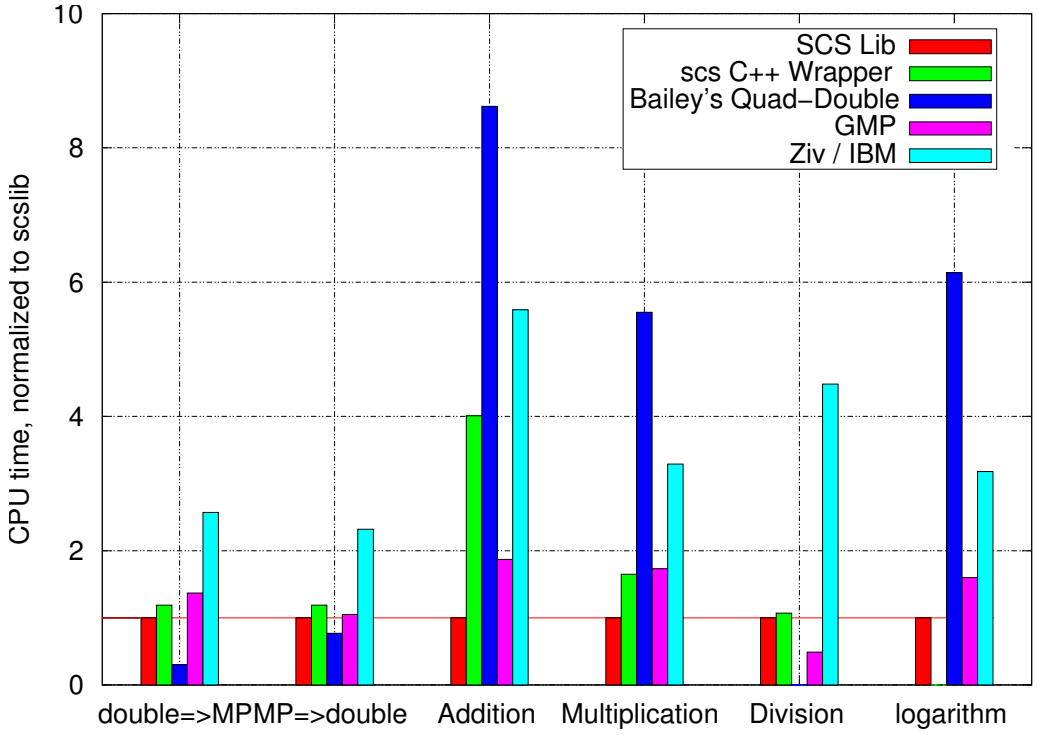


FIG. 4.6: *Comparaison sur un PowerPC G4*

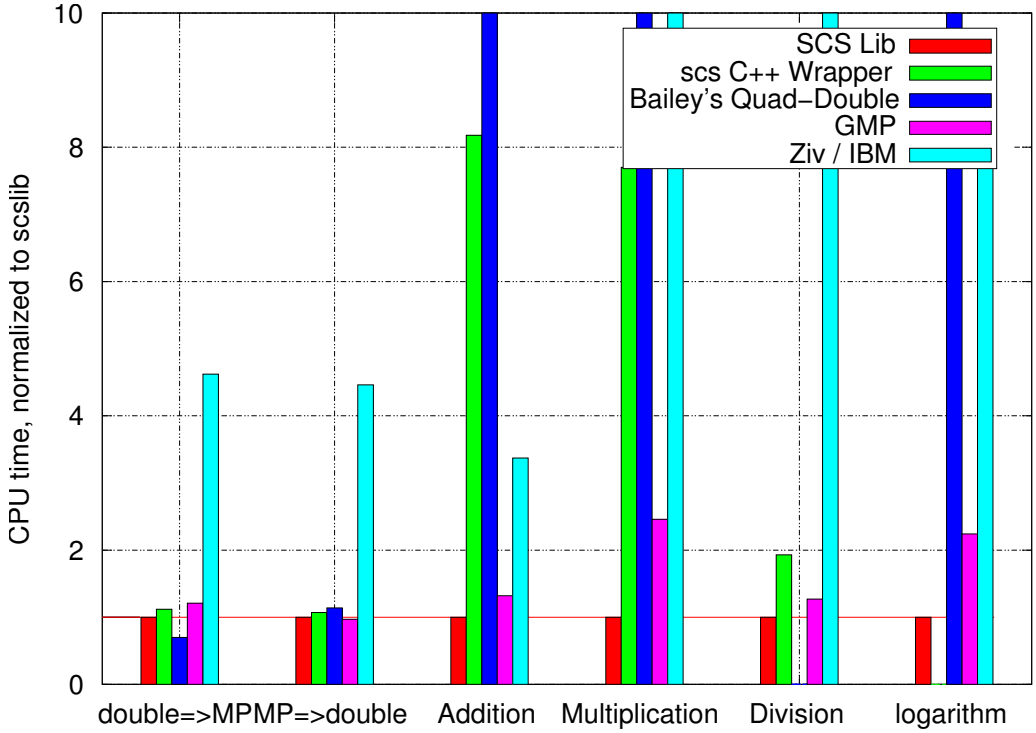


FIG. 4.7: *Comparaison sur un Itanium*

4.6.2.1 Les conversions

Il n'y a aucune surprise dans la mesure du temps de conversion d'un flottant double précision de/vers un nombre MP. SCS et GMP, les deux bibliothèques basées sur des entiers, ont des performances comparables, alors que la bibliothèque quad-double nous montre la simplicité de ses conversions. Cette différence entre ces deux classes d'algorithmes reflète probablement la présence d'instructions ou de matériel dédiés à la conversion d'un entier machine de/vers un flottant machine. En revanche les mauvais résultats de la bibliothèque quad-double sur Itanium par rapport aux autres architectures restent encore inexpliqués.

4.6.2.2 Analyse des performances des opérations MP

Concernant les opérations arithmétiques, les biliothèques GMP et SCS ont un net avantage sur les bibliothèques basées sur une représentation flottante. Dans la suite, nous nous concentrerons donc sur GMP. Plusieurs effets contribuent à expliquer les différences de performance observées entre SCS et GMP.

1. La bibliothèque SCS (comme celles d'IBM et de Bailey) a une précision fixée sélectionnée au moment de la compilation, tandis que GMP utilise une précision arbitraire. Cela signifie que SCS utilise des boucles de tailles fixes, alors que GMP doit gérer des boucles de tailles arbitraires. De plus dans SCS les boucles ont été déroulées à la main.
2. SCS utilise moins de propagation de retenues, et effectue donc moins de travail par chiffre.
3. GMP utilise de l'assembleur, ainsi que des instructions multimedia lorsque cela est possible, par exemple dans le cas du Pentium 4.

4. GMP a besoin de moins de chiffres pour une précision donnée.
5. SCS exhibe du parallélisme.

4.6.2.3 Les bénéfices de la simplicité pour l'addition

Les algorithmes retenus pour SCS et GMP présentent des complexités semblables, ainsi que la même quantité de dépendances de données. Par conséquent elles devraient avoir les mêmes performances. Néanmoins, le coût de la gestion des boucles (décrémenter un index, le comparer à zéro, effectuer un branchement) dépasse le coût de calcul (un *add-with-carry*). La seule raison pouvant expliquer la rapidité de SCS par rapport à GMP dans ce cas est que dans SCS nous avons déroulé manuellement les boucles. Encore une fois, il n'est pas illusoire de croire qu'une telle technique sera prise en charge par le compilateur dans un futur proche.

4.6.2.4 Le bénéfice du parallélisme pour la multiplication

Pour la multiplication, les résultats sont plus intéressants. Sur les architectures ne pouvant lancer qu'une seule multiplication par cycle (toutes sauf l'Itanium), l'avantage de SCS sur GMP pour la multiplication est semblable à celui existant pour l'addition. En revanche, comme le montrent les résultats, l'Itanium exploite très bien l'important parallélisme exposé par le format SCS. Sur cette architecture, capable de lancer deux multiplications par cycle, les rapports de performances de SCS par rapport à GMP pour la multiplication sont deux fois ceux de l'addition. Quant aux résultats sur le SPARC, ils s'expliquent par les mauvaises performances de la multiplication entière utilisée par GMP.

4.6.2.5 Applications : division et logarithme

Concernant la division, les algorithmes utilisées par SCS et GMP sont complètement différents : la division SCS est basée sur des itérations de Newton-Raphson, tandis que GMP utilise un algorithme basé sur une récurrence sur les chiffres [58, 70]. Les résultats de la division nous montrent que la division SCS est améliorable du point de vue algorithmique. Nous n'avons pas cherché à améliorer les performances de la division principalement car cet opérateur n'est pas utilisé pour l'évaluation des fonctions élémentaires.

Finalement, les performances du logarithme sont proches des performances de la multiplication. En effet le cœur de l'algorithme utilisé pour le logarithme utilise des multiplications. Le logarithme est une des applications typiques de cette bibliothèque, ce qui justifie l'importance d'exploiter le parallélisme des multiplications MP.

4.7 Conclusion et travaux futurs

Au cours de ce chapitre nous avons présenté le format *Software Carry-Save* pour la multiprécision, et discuté de son implémentation sur les processeurs modernes en utilisant les caractéristiques des compilateurs. L'idée principale est de réduire le nombre de dépassements de capacité, d'erreurs d'arrondi, et de retenues générées par les opérations machine. Les principaux avantages de cette approche sont les suivants :

– Utiliser indifféremment des entiers ou des flottants double précision, ce qui permet d'écrire les algorithmes dans un langage de haut niveau de façon portable.

– Les algorithmes sont simples, donc efficaces et compacts.
– En étant intrinsèquement parallèles, ces algorithmes sont parfaits pour les profonds pipelines et les multiples unités de calcul des processeurs superscalaires.

Les expériences effectuées sur la variété de processeurs nous confortent dans cette opinion. Cette bibliothèque a par conséquent fait l'objet de plusieurs publications [29], [30]. Elle est par ailleurs disponible sous licence LGPL à l'adresse `www.ens-lyon.fr/LIP/Arenaire/`.

Les performances de cette bibliothèque dépassent celle des bibliothèques les plus compétitives, y compris la bibliothèque de référence dans le domaine qui est GMP, sur un large panel de processeurs et pour des précisions comprises entre 100 et 500 bits. De plus l'effort de programmation pour obtenir ces résultats est sans commune mesure avec ceux développés par GMP. Il est à noter que les développeurs de GMP sont conscients de ce que leur bibliothèque est trop séquencielle sur les processeurs modernes. Dans les versions récentes, certaines des fonctions écrites en assembleur font deux propagations de retenue en parallèle, ce qui complexifie encore plus la bibliothèque. Bien entendu, notre but n'est pas de remplacer GMP, mais d'offrir une alternative pour des petites précisions, et surtout de ne pas avoir à dépendre d'une autre bibliothèque pour l'évaluation des fonctions élémentaires au sein de la bibliothèque *crlibm*.

Perspectives

Bien que de telles prédictions puissent sembler hasardeuses, nous pouvons penser que les générations futures de processeurs pourront exploiter encore plus de parallélisme. Cela pourra prendre la forme d'un pipeline plus profond, bien que la limite pratique n'est pas loin d'être atteinte (voir discussion dans [71]). Nous pensons également que les futurs processeurs seront capables de lancer plus d'une multiplication par cycle, soit dans un style proche de celui de l'Itanium soit par l'intermédiaire d'unités multimedia.

Dans ce cas, l'approche SCS se révèlera particulièrement intéressante. En utilisant la variante de Brent (où les bits de retenues imposent une propagation tous les 2^{M-2m} multiplications), il semble envisageable de pouvoir développer une bibliothèque multiprécision en précision arbitraire dans le style de GMP. Toutefois, les compromis qu'implique cette approche restent à étudier.

CHAPITRE

5

luer les fonctions élémentaires en deux étapes. Ces bornes nous permettent également de borner le temps d'exécution dans le pire cas. Le schéma adopté pour chaque fonction élémentaire est d'utiliser conjointement une évaluation rapide qui rend un résultat juste la plupart du temps et une évaluation précise basée sur la multiprécision, qui rend un résultat juste tout le temps.

Un argument probabiliste est que pour chaque bit supplémentaire de précision obtenu pour le résultat final de la première étape, le nombre de cas difficiles à arrondir est divisé par deux. Il existe donc un compromis entre précision de la première étape et performance. Ce compromis est dépendant des fonctions à évaluer, et est destiné à fournir un temps d'exécution en moyenne comparable aux autres schémas d'évaluation.

Comme pour la bibliothèque multiprécision SCS, nous avons programmé cette bibliothèque en C ANSI. Nous avons rendu le code portable en nous basant sur le format IEEE-754 double précision et en utilisant les opérations correspondantes. En particulier nous avons exclu l'utilisation de matériel spécifique comme le FMA ou la double précision étendue.

Autant que possible nous avons préféré disposer d'une implémentation simple, quitte à perdre sur le plan des performances. Cette simplicité du code est nécessaire pour construire la preuve de l'algorithme mais également du programme correspondant. Nous minimisons ainsi la probabilité de commettre des erreurs dans la preuve. Comme nous le verrons par la suite avec l'exemple de l'exponentielle, établir une preuve du bon comportement d'un programme d'évaluation d'une fonction élémentaire est un travail fastidieux (30 pages).

Dans la suite de ce chapitre nous présenterons le programme complet de l'évaluation de l'exponentielle tel qu'il est présent dans la bibliothèque *crlibm* regroupant les évaluations des fonctions élémentaires déjà réalisées. Nous espérons avoir décrit la méthode utilisée avec suffisamment de détails et d'explications sur les choix réalisés, pour permettre au lecteur de cerner les problématiques du processus de création d'un programme d'évaluation de fonction élémentaire avec arrondi correct *prouvé*.

5.2 L'exponentielle

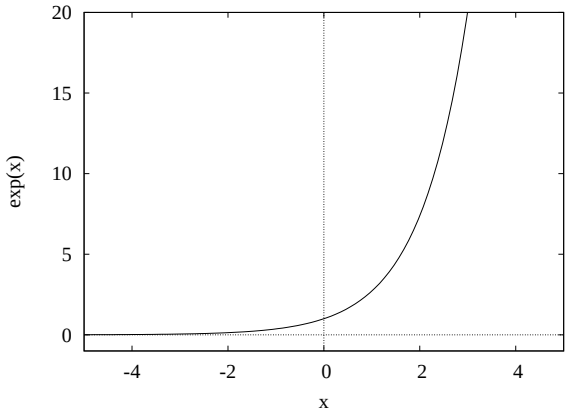


FIG. 5.1: *Fonction exponentielle*

5.3 Présentation de la méthode

L'évaluation de l'exponentielle s'effectue en deux étapes. La *phase rapide* calcule une approximation juste jusqu'à 68 bits. À la suite de cette étape un test de précision est effectué. Ce test détermine si l'appel à la *phase précise*, basée sur les opérateurs multiprécision SCS, est nécessaire.

Les constantes incluses dans le code sont données en hexadécimal. Pour des raisons de concision, seules les valeurs au format «big endian» sont données dans le code. Les valeurs décimales correspondantes sont mises en commentaires.

5.4 Phase rapide

Voici le schéma général adopté pour la première partie de l'évaluation de l'exponentielle :

1. **Réduction d'argument «mathématique»**
   Nous souhaitons évaluer $\exp(x)$. On évalue l'argument réduit

   $$(r\_hi + r\_lo) \in [-\ln(2)/2, +\ln(2)/2]$$

   tel que :

   $$x \simeq k.\ln(2) + (r\_hi + r\_lo)$$

   donc $\exp(x) = \exp(r\_hi + r\_lo).2^k$ avec $(r\_hi + r\_lo) \in [-\ln(2)/2, +\ln(2)/2]$
2. **Réduction d'argument par table**
   Soient $index\_flt$ le flottant conenant les bits compris entre 2^{-1} et 2^{-8} de $(r\_hi + r\_lo)$ nous cherchons

   $$\exp(r\_hi + r\_lo) = \exp(index\_flt) \times \exp(rp\_hi + rp\_lo)$$

   où $(rp\_hi + rp\_lo) = (r\_hi + r\_lo) - index\_flt$ tel que $(rp\_hi + rp\_lo) \in [-2^{-9}, +2^{-9}]$ et $\exp(index\_flt) \simeq (ex\_hi + ex\_lo)$ est obtenu par une lecture de table.
3. **Évaluation polynomiale**
   Nous évaluons le polynôme P_r de degré 3 tel que :

   $$\exp(rp\_hi + rp\_lo) \approx 1 + (rp\_hi + rp\_lo) + \frac{1}{2}.(rp\_hi + rp\_lo)^2 + (rp\_hi + rp\_lo)^3.(P\_r)$$

   avec $P\_r = c_0 + c_1.rp\_hi + c_2.rp\_hi^2 + c_3.rp\_hi^3$ et $rp\_hi \in [-2^{-9}, +2^{-9}]$
4. **Reconstruction**

   $$\begin{aligned} \exp(x) &= 2^k.(ex\_hi + ex\_lo). \\ &\quad (1 + (rp\_hi + rp\_lo) + \tfrac{1}{2}.(rp\_hi + rp\_lo)^2 + (rp\_hi + rp\_lo)^3.P\_r).(1 + \varepsilon_{-70}) \end{aligned}$$

5.5 Gestion des cas spéciaux

5.5.1 Lever les drapeaux

La norme prévoit la levée de drapeaux et une gestion des exceptions pour les 4 opérations de base ($+$, $\times$, $\div$, $\sqrt{}$). Comme nous l'avons soulevé au cours des discussions sur la normalisation des fonctions élémentaires, introduire une gestion similaire des exceptions pour les fonctions élémentaires est légitime.

Notre code est destiné à être portable, aussi nous utilisons du C standard. Nous ne pouvons donc gérer les exceptions manuellement en assembleur. Nous laissons l'ensemble composé du processeur, du compilateur et du système générer les exceptions et drapeaux grâce aux instructions suivantes :

- **underflow** : la multiplication $\pm smallest \times smallest$ où $smallest$ correspond au plus petit nombre dénormalisé représentable,
- **overflow** : la multiplication $\pm largest \times largest$ où $largest$ correspond au plus grand nombre normalisé représentable,
- **division par zéro** : la division $\pm 1.0/0.0$,
- **résultat inexact** : l'addition $(x + smallest) - smallest$ où x est le résultat et $smallest$ le plus petit nombre dénormalisé représentable,
- **opération invalide** : la division $\pm 0.0/0.0$.

Il semblerait que sur certaines configurations (ex. FreeBSD) les exceptions déclenchent des erreurs par défaut.

5.5.2 Dépassements de capacité

Dans le reste, nous considérerons des nombres dans l'intervalle $[u\_bound, o\_bound]$, où u_bound et o_bound sont définis de la façon suivante :

$$u\_bound = \triangle\left(\ln\left(\left(1-2^{-53}\right).2^{-1075}\right)\right) = -745.1332\ldots$$

$$o\_bound = \triangledown\left(\ln\left(\left(1-2^{-53}\right).2^{1024}\right)\right) = 709.7827\ldots$$

En mode d'arrondi au plus près, l'exponentielle d'un nombre plus grand que o_bound est représentée par un dépassement de capacité (overflow), tandis que l'exponentielle d'un nombre plus petit que u_bound sera arrondi à 0, et lèvera le drapeau inexact.

Cependant, d'autres dépassement de capacité peuvent apparaître dans deux cas que nous devons éviter :

- Un calcul intermédiaire peut générer un «overflow/underflow» qui se propagera jusqu'au résultat, bien que ce dernier soit représentable par un nombre IEEE double précision.
- En arithmétique IEEE-754, lorsqu'un résultat est compris entre 2^{-1023} et 2^{-1074}, une exception de type underflow est générée pour signaler une perte drastique de précision.

Au cours de la description de l'algorithme nous montrerons comment éviter de tels problèmes.

5.5.3 En arrondi au plus près

Listing 5.1: *Gestion des cas spéciaux en arrondi au plus près*

```
1  static const union{int i[2]; double d;}
2  #ifdef BIG_ENDIAN
3   _largest     = {0x7fefffff, 0xffffffff},
4   _smallest    = {0x00000000, 0x00000001},
5   _u_bound     = {0xC0874910, 0xD52D3052}, /* -7.45133219101941222107e+02 */
6   _o_bound     = {0x40862E42, 0xFEFA39F0}; /*  7.09782712893384086783e+02 */
7  #else
8   ...
9  #endif
10 #define largest     _largest.d
11 #define smallest    _smallest.d
12 #define u_bound     _u_bound.d
13 #define o_bound     _o_bound.d
14
15 /* Variables temporaires */
16 unsigned int hx;
17
18 hx   = HI(x);
19 hx &= 0x7fffffff;
20
21 /* Filtre les cas spéciaux */
22 if (hx >= 0x40862E42){
23   if (hx >= 0x7ff00000){
24     if (((hx&0x000fffff)|LO(x))!=0)
25       return x+x;                                  /* NaN */
26     else return ((HI(x)&0x80000000)==0)? x:0.0; /* exp(+/-inf) = inf,0 */
27   }
28   if (x > o_bound) return largest*largest;     /* overflow  */
29   if (x < u_bound) return smallest*smallest;   /* underflow */
30 }
31
32
33 if (hx <= 0x3C900000)                          /* if (hx <= 2^(-54)) */
34   return ((hx == 0)&&(LO(x) == 0))? 1.: 1.+smallest;
```

◇ *Preuve.*

ligne 18	Place dans hx la partie haute de x. (cf. prog. 3.3)
ligne 19	Supprime l'information de signe dans hx, pour simplifier les tests sur les cas spéciaux.
ligne 22	Test équivalent à $if(\|x\| >= 709.7822265625)$. Ce test est vrai si $x > u\_bound$, $x < o\_bound$, x représente l'infini ou un NaN. (Voir tableau 1.3 page 8 sur la représentation des nombres flottants). Ce test est réalisé sur les entiers pour une question de rapidité.
ligne (23-25)	Teste si x représente l'infini ou un NaN et renvoie les valeurs correspondantes ($+\infty$ ou 0 exact).
ligne 28	Si le compilateur a correctement traduit le flottant alors $o\_bound = 390207173010335/549755813888$. Si $x > o\_bound$ alors $\exp(x) = +\inf$. La multiplication $largest * largest$ laisse au processeur le soin de créer un overflow et de lever les drapeaux correspondants.
ligne 29	Si le compilateur a correctement traduit le flottant alors $u\_bound = -3277130554578985/4398046511104$. Si $x < u\_bound$ alors $\exp(x) = +0$. La multiplication $smallest * smallest$ est effectuée de façon à renvoyer 0 et de lever le drapeau inexact.

ligne 33 Test équivalent à $if(|x| \leq 2^{-54})$. Ce test est réalisé sur les entiers pour une question de rapidité et est valide car $x \notin \{NaN, \infty\}$. De plus ce test permet de traiter les cas où x est un dénormalisé. Nous avons donc la propriété suivante :

<1> $|x| > 2^{-54}$ et $x \notin \{NaN, \infty\}$ •

En effet, en arrondi au plus près si $|x| \leq 2^{-54}$ alors $\exp(x) = 1.0$. Ce test nous garantit qu'aucun dénormalisé ne sera rencontré dans la suite des calculs.

□

5.5.4 En arrondi vers $+\infty$

Listing 5.2: *Gestion des cas spéciaux en arrondi vers* $+\infty$

```
1  static const union{int i[2]; double d;}
2  #ifdef BIG_ENDIAN
3   _largest     = {0x7fefffff, 0xffffffff},
4   _smallest    = {0x00000000, 0x00000001},
5   _u_bound     = {0xC0874910, 0xD52D3052}, /* -7.45133219101941222107e+02 */
6   _o_bound     = {0x40862E42, 0xFEFA39F0}, /*  7.09782712893384086783e+02 */
7   _two_m52_56 = {0x3CB10000, 0x00000000}; /*  2.35922392732845764840e-16 */
8  #else
9    ...
10 #endif
11 #define largest     _largest.d
12 #define smallest    _smallest.d
13 #define u_bound     _u_bound.d
14 #define o_bound     _o_bound.d
15 #define two_m52_56 _two_m52_56.d
16
17
18 /* Variables temporaires */
19 unsigned int hx;
20
21 hx   = HI(x);
22 hx &= 0x7fffffff;
23
24 /* Filtre les cas spéciaux */
25 if (hx >= 0x40862E42){
26   if (hx >= 0x7ff00000){
27     if (((hx&0x000fffff)|LO(x))!=0)
28       return x+x;                                   /* NaN */
29     else return ((HI(x)&0x80000000)==0)? x:0.0;    /* exp(+/-inf) = inf,0 */
30   }
31   if (x > o_bound) return largest*largest;         /* overflow   */
32   if (x < u_bound) return smallest*(1.0+smallest);/* 2^(-1074) */
33 }
34
35 if (hx < 0x3CA00000){                              /* if (hx <= 2^(-53)) */
36   if ((hx == 0)&&(LO(x) == 0))
37     return 1.;                                    /* exp(0)=1. */
38   if (HI(x) < 0)
39     return 1. + smallest;                         /* 1 and inexact */
40   else
41     return 1. + two_m52_56;                       /* 1 + 2^(-52) and inexact */
```

```
42 }
```

◇ *Preuve.* Ce code est identique à celui utilisé pour l'arrondi au plus près à l'exception de :
- Lorsque $x < u\_bound$, en arrondi vers $+\infty$, nous devons rendre le plus petit nombre représentable (2^{-1074}) avec le drapeau inexact levé.
- Lorsque $|x| < 2^{-53}$, en arrondi vers $+\infty$, nous devons rendre 1.0 si x négatif avec le drapeau inexact levé ou $1 + 2^{-52}$ avec le drapeau inexact levé si x positif.

□

5.5.5 En arrondi vers $-\infty$

Listing 5.3: *Gestion des cas spéciaux en arrondi vers* $-\infty$

```
1  static const union{int i[2]; double d;}
2  #ifdef BIG_ENDIAN
3   _largest     = {0x7fefffff, 0xffffffff},
4   _smallest    = {0x00000000, 0x00000001},
5   _u_bound     = {0xC0874910, 0xD52D3052}, /* -7.45133219101941222107e+02 */
6   _o_bound     = {0x40862E42, 0xFEFA39F0}, /*  7.09782712893384086783e+02 */
7   _two_m53_56 = {0x3CA20000, 0x00000000}; /*  1.24900090270330110798e-16 */
8  #else
9   ...
10 #endif
11 #define largest     _largest.d
12 #define smallest    _smallest.d
13 #define u_bound     _u_bound.d
14 #define o_bound     _o_bound.d
15 #define two_m53_56 _two_m53_56.d
16
17 /* Variables temporaires */
18 unsigned int hx;
19
20 hx   = HI(x);
21 hx &= 0x7fffffff;
22
23 /* Filtre les cas spéciaux */
24 if (hx >= 0x40862E42){
25   if (hx >= 0x7ff00000){
26     if (((hx&0x000fffff)|LO(x))!=0)
27       return x+x;                                  /* NaN */
28     else return ((HI(x)&0x80000000)==0)? x:0.0;  /* exp(+/-inf) = inf,0 */
29   }
30   if (x > o_bound) return largest*(1.0+smallest);/* (1-2^(-53))*2^1024  */
31   if (x < u_bound) return smallest*smallest; /* underflow */
32 }
33
34 if (hx < 0x3CA00000){                           /*  if (hx <= 2^(-53))    */
35   if ((hx == 0)&&(LO(x) == 0))
36     return 1.;                                   /* exp(0)=1. */
37   if (HI(x) < 0)
38     return 1. - two_m53_56;                      /* 1-2^(-53) and inexact */
39   else
40     return 1. + smallest;                        /* 1 and inexact */
41 }
```

◇ *Preuve.* Ce code est identique à celui utilisé pour l'arrondi au plus près à l'exception de :

- Lorsque $x > o\_bound$, en arrondi vers $-\infty$, nous devons rendre le plus grand nombre représentable ($(1-2^{-53}).2^{1024}$) avec le drapeau inexact levé.
- Lorsque $|x| < 2^{-53}$, en arrondi vers $-\infty$, nous devons rendre 1.0 si $x = 0$ exactement, $1.0 - 2^{-53}$ si x négatif avec le drapeau inexact levé ou 1.0 avec le drapeau inexact levé si x positif.

□

5.5.6 En arrondi vers 0

La fonction $\exp(x)$ est continue, croissante et non-négative pour tout x, donc l'arrondi dirigé vers 0 est équivalent à l'arrondi vers $-\infty$.

5.6 La réduction d'argument

5.6.1 Présentation

La réduction d'argument typique pour l'exponentielle utilise la propriété :

$$e^{a+b} = e^a e^b.$$

5.6.2 Première réduction

Le but de cette première réduction est de remplacer l'argument d'entrée $x \in [u\_bound, o\_bound]$ par deux nombres flottants r_hi, r_lo et un entier k tels que :

$$x = k.\ln(2) + (r\_hi + r\_lo).(1+\varepsilon)$$

avec $|r\_hi + r\_lo| < \frac{1}{2}\ln(2)$.

Cette réduction de type additif peut générer une perte de précision importante si x est très proche d'un multiple de $\ln(2)$ (voir section 2.2.1.1). En utilisant la méthode de Kahan basée sur les fractions continues (voir Muller [67] pp 154) on calcule les pires cas pour la réduction d'argument résumés dans le tableau 5.2.

Intervalle	Pire cas	Nombre de bits perdus
$]2^{1024}, 2^{1024}[$	$5261692873635770 \times 2^{499}$	$66,8$
$[-1024, 1024]$	$7804143460206699 \times 2^{-51}$	$57,5$

TAB. 5.2: *Pire cas correspondant au nombre le plus proche d'un multiple de* $\ln 2$ *pour la réduction additive pour l'exponentielle. Le nombre de bits perdus est également indiqué.*

L'intervalle $[u\_bound, o\_bound]$ sur lequel la fonction exponentielle est évaluée est inclus dans $[-1024, 1024]$. Donc au plus 58 bits peuvent être annulés lors de la soustraction du plus proche multiple de $\ln(2)$ au nombre d'entrée x. Cependant cette perte de précision n'a pas d'influence sur le schéma d'évaluation de l'exponentielle. En effet, seul l'erreur absolue, et non l'erreur relative, va peser sur le précision du résultat final.

Théorème 6 *La suite d'opérations du programme 5.4 calcule les deux flottants double précision r_hi, r_lo et l'entier k tels que :*

$$r\_hi + r\_lo = (x - k \times \ln 2) + \varepsilon_{-91}$$

avec k l'entier le plus proche de $x/\ln 2$.

Listing 5.4: *Première réduction*

```
42 static const union{int i[2]; double d;}
43 #ifdef BIG_ENDIAN
44  _ln2_hi      = {0x3FE62E42, 0xFEFA3800}, /*  6.93147180559890330187e-01 */
45  _ln2_lo      = {0x3D2EF357, 0x93C76730}, /*  5.49792301870837115524e-14 */
46  _inv_ln2     = {0x3FF71547, 0x6533245F}; /*  1.44269504088896338700e+00 */
47 #else
48  ...
49 #endif
50 #define ln2_hi      _ln2_hi.d
51 #define ln2_lo      _ln2_lo.d
52 #define inv_ln2     _inv_ln2.d
53
54 /* Variables temporaires */
55 double r_hi, r_lo, rp_hi, rp_lo;
56 double u, tmp;
57 int k;
58
59 /* Arrondi au plus près */
60 DOUBLE2INT(k, x * inv_ln2)
61
62 if (k != 0){
63   /* r_hi+r_lo =  x - (ln2_hi + ln2_lo)*k */
64   rp_hi = x-ln2_hi*k;
65   rp_lo =  -ln2_lo*k;
66   Fast2SumCond(r_hi, r_lo, rp_hi, rp_lo);
67 }else {
68   r_hi = x;  r_lo = 0.;
69 }
```

◇ *Preuve.*

ligne (44-45)

<2> Par construction : $ln2\_hi + ln2\_lo = \ln(2)(1 + \varepsilon_{-102})$ •

<3> $|ln2\_hi| \leq 2^0 \quad |ln2\_lo| \leq 2^{-44}$ •

<4> $ln2\_hi$ est exactement représentable avec 42 bits de précision •

ligne 60 Place dans k l'entier le plus proche de $\circ(x * inv\_ln2)$. On se sert de la propriété de DOUBLE2INT qui convertit un flottant en entier avec un arrondi au plus près (programme 3.10, page 50).
De plus k vérifie la propriété suivante :

<5> $\lfloor x \otimes inv\_ln2 \rfloor \leq k \leq \lceil x \otimes inv\_ln2 \rceil$ •

On a vu à la section 5.5.2 que $-745.1332\ldots < x < 709.7827\ldots$, Donc :

<6> $-1075 \leq k \leq 1025$ et $|k|$ est un entier s'écrivant sur au plus 11 bits •

ligne 64 Les propriétés <4> et <6> nous donnent :

<7> $ln2\_hi \otimes k = ln2\_hi \times k$ exactement •

Par la propriété <5> on a :

$$|(x \otimes inv\_ln2 - 1) \times ln2\_hi| \leq |k \times ln2\_hi| \leq |(x \otimes inv\_ln2 + 1) \times ln2\_hi|$$

De plus, sans perte de généralité on suppose que $|x| \geq |ln2\_hi|$, dans le cas contraire $k = 0$ et aucune réduction n'est nécessaire. On a ainsi $|(x \times inv\_ln2 + 1) \times ln2\_hi| \leq |2.x|$ et $|(x \times inv\_ln2 - 1) \times ln2\_hi| \geq |\frac{x}{2}|$, donc :

$$|x/2| \leq |k \times ln2\_hi| \leq |2.x|$$

Donc par le théorème de Sterbenz (théorème 1, page 7), on a

$$x \ominus (ln2\_hi \otimes k) = x - (ln2\_hi \otimes k)$$

Si l'on combine ce résultat avec la propriété <7> on obtient :

<8> $x \ominus (ln2\_hi \otimes k) = x - (ln2\_hi \times k)$ exactement •

ligne 65 Par les propriétés <3> et <6> on a :

<9> $|rp\_lo| \leq 2^{-33}$,
$rp\_lo = -(ln2\_lo \times k) + \varepsilon_{-91}$ •

ligne 66 Nous utilisons l'algorithme Fast2Sum conditionnel (avec tests sur les entrées), car $x - ln2\_hi \times k$ peut être nul (dû aux 58 bits de «cancellation»). L'algorithme Fast2Sum conditionnel (programme 3.4, page 47) nous garantit que

$$r\_hi + r\_lo = (x \ominus (ln2\_hi \otimes k)) + (-ln2\_lo \otimes k)$$

En utilisant les propriétés <8> et <9> on obtient :

<10> $r\_hi + r\_lo = (x - ln2\_hi \times k) + (-ln2\_me \times k) + \varepsilon_{-91}$ •

ligne 68 Si $k = 0$ alors aucune soustraction n'est nécessaire, donc $r\_hi + r\_lo = x$ exactement.

□

A l'issue de la première réduction on a :

$$\exp(x) = 2^k . \exp(r\_hi + r\_lo + \varepsilon_{-91})$$

5.6.3 Deuxième réduction

Le nombre $(r\_hi + r\_lo)$ est toujours trop grand pour être utilisé dans une évaluation polynomiale. Par conséquent, une seconde réduction d'argument est nécessaire. Cette réduction

est basée sur la propriété additive de l'exponentielle $e^{a+b} = e^a e^b$ et consiste à tabuler certaines valeurs de l'exponentielle.

Soient $index\_flt$ les bits compris entre 2^{-1} et $2^{-\ell}$ de $(r\_hi + r\_lo)$, alors nous avons :

$$\begin{aligned} \exp(r\_hi + r\_lo) &= \exp(index\_flt).\exp(r\_hi + r\_lo - index\_flt) \\ &\approx (ex\_hi + ex\_lo).\exp(rp\_hi + rp\_lo) \end{aligned}$$

où ex_hi et ex_lo sont les flottants double précision contenus dans une table adressée par $index\_flt$, tels que $ex\_hi + ex\_lo \approx \exp(index\_flt)$. L'argument après réduction sera également composé de deux flottants double précision rp_hi et rp_lo tels que

$$rp\_hi + rp\_lo = r\_hi + r\_lo - index\_flt$$

exactement.

Les tests sur la mémoire cache (section 3.4) nous ont montré que la taille *optimale* de table pour la réduction d'argument est de 4 Koctets. Si l'on souhaite tabuler ces valeurs et conserver une précision suffisante (environ 70 bits), il faut 2 flottants double précision (16 octets) par valeur.

Une précision de 70 bits peut être atteinte en utilisant un flottant double précision (53 bits) et un flottant simple précision (24 bits). Cependant cette technique a deux inconvénients majeurs. Les flottants simple précision ont une dynamique de représentation des nombres plus faible que les flottants double précision, aussi les valeurs mémorisées sont susceptibles de ne pas être représentables par un flottant simple précision. De plus chaque donnée nécessitera 96 bits par valeur, et cette taille n'est pas une puissance de 2, ce qui peut poser problème pour l'alignement mémoire (voir section 3.1.3).

Si ℓ est le paramètre tel que $[-2^{-\ell-1}, 2^{-\ell-1}]$ soit l'intervalle après réduction, nous souhaitons que :

$$\lceil \ln 2.2^{\ell} \rceil 16 \leq (2^{12} = 4096).$$

Avec $\ell = 8$ nous avons $\lceil \ln(2).2^8 \rceil 16$ octets $= 2848$ octets, et l'intervalle d'évaluation est réduit à $[-2^{-9}, 2^{-9}]$. À l'issue de cette étape de réduction $|rp\_hi + rp\_lo| \leq 2^{-9}$.
La séquence d'opérations correspondant à cette deuxième réduction est :

Listing 5.5: *Deuxième réduction*

```
70 /* Définition des constantes */
71 static const union{int i[2]; double d;}
72 #ifdef BIG_ENDIAN
73  _two_44_43  = {0x42B80000, 0x00000000}; /*  26388279066624.  */
74 #else
75  ...
76 #endif
77 #define two_44_43  _two_44_43.d
78 #define bias       89;
79
80 /* Variables temporaires */
81 double ex_hi, ex_lo, index_flt;
82 int index;
83
84 /* Arrondi au plus près */
85 index_flt = (r_hi + two_44_43);
86 index     = LO(index_flt);
87 index_flt -= two_44_43;
88 index     += bias;
89 r_hi      -= index_flt;
90
91 /* Normalisation du résultat */
```

```
92 Fast2Sum(rp_hi, rp_lo, r_hi, r_lo)
93
94 /* Lecture de table */
95 ex_hi = tab_exp[index][0];
96 ex_lo = tab_exp[index][1];
```

◇ *Preuve.*

ligne 73 La constante $two\_44\_43 = 2^{44} + 2^{43}$ sert à utiliser le mode d'arrondi de la machine pour disposer des $\ell = 8$ premiers bits de $r\_hi + r\_lo$.

ligne 78 En langage C, les tableaux sont addressés par des index positifs. On considère aussi bien des valeurs positives que négatives pour $index$, nous devons donc disposer d'un biais égal à $178/2 = 89$ où $178 = (\ln(2) \times 2^8) + 1$.

ligne (85, 89) Cette séquence d'opérations est similaire à celle utilisée dans DOUBLE2INT (programme 3.10, page 50). Elle permet de placer dans la variable $index$ les bits compris entre 2^{-1} et 2^{-8}, moins la valeur du biais. Elle place également dans $index\_flt$ la valeur flottante correspondant aux 8 premiers bits de r_hi. En ligne 89 nous avons :

<11> $r\_hi = r\_hi - index\_flt$ exactement •

ligne 92 L'algorithme Fast2Sum nous garantit :

<12> $|rp\_hi| \leq 2^{-9}$ et $|rp\_lo| \leq 2^{-62}$,
$rp\_hi + rp\_lo = r\_hi + r\_lo$ exactement •

ligne 95, 96 On effectue la lecture de table des 2 valeurs de l'exponentielle. La structure de ce tableau est faite de telle sorte qu'au maximum un seul défaut de cache peut se produire pour les deux lectures de table (section 3.1.3).
Par construction de la table tab_exp on a :

<13> $|ex\_hi| \leq 2^{+1}$ et $|ex\_lo| \leq 2^{-52}$
$ex\_hi + ex\_lo = \exp(index\_flt).(1 + \varepsilon_{-105=-52-53})$ •

□

À l'issue de la deuxième réduction on a :

$$\exp(x) = 2^k.(ex\_hi + ex\_lo).\exp(rp\_hi + rp\_lo + \varepsilon_{-91}).(1 + \varepsilon_{-105})$$

5.7 Évaluation polynomiale

Soit $r = (rp\_hi + rp\_lo)$, Il nous faut évaluer $\exp(r)$ avec $r \in [-2^{-9}, 2^{-9}]$. Cette évaluation est effectuée par un polynôme construit en utilisant les différentes remarques de la section 2.2.2. Nous allons donc évaluer $f(r) = (\exp(r) - 1 - r - \frac{r^2}{2})/r^3$ par le polynôme de degré 3 suivant :

$$P(r) = c_0 + c_1 r + c_2 r^2 + c_3 r^3$$

où

- $c_0 = 6004799503160629/36028797018963968 \leq 2^{-2}$
- $c_1 = 750599937895079/18014398509481984 \leq 2^{-4}$
- $c_2 = 300240009245077/36028797018963968 \leq 2^{-6}$
- $c_3 = 3202560062254639/2305843009213693952 \leq 2^{-9}$

avec c_0, c_1, c_2, c_3 exactement représentables par des flottants double précision.
En utilisant la fonction `infnorm` de Maple nous obtenons l'erreur suivante :

<14> $\exp(r) = (1 + r + \frac{1}{2}r^2 + r^3.P(r)).(1 + \varepsilon_{-78})$ pour $r \in [-2^{-9}, 2^{-9}]$ •

Pour des raisons d'efficacité, nous n'allons pas évaluer $P(rp\_hi + rp\_lo)$ mais $P(rp\_hi)$. L'erreur correspondant à cette approximation est :

$$\begin{aligned} P(rp\_hi + rp\_lo) - P(rp\_hi) &= c_1.rp\_lo + \\ &\quad c_2.(rp\_lo^2 + 2.rp\_hi.rp\_lo) + \\ &\quad c_3.(rp\_lo^3 + 3.rp\_hi^2.rp\_lo + 3.rp\_hi.rp\_lo^2) \quad \text{<12>} \\ &\leq \varepsilon_{-66} + \varepsilon_{-74} \end{aligned}$$

La propriété <14> devient :

<15> $\exp(r) = (1 + r + \frac{1}{2}r^2 + r^3.P(rp\_hi)) + \varepsilon_{-78} + \varepsilon_{-93}$ pour $r \in [-2^{-9}, 2^{-9}]$ •

$P\_r = P(rp\_hi)$ est évalué par la séquence d'opérations suivante :

Listing 5.6: *Évaluation polynomiale*

```
97  static const union{int i[2]; double d;}
98  #ifdef BIG_ENDIAN
99   _c0          = {0x3FC55555, 0x55555535}, /*  1.66666666666665769236e-01 */
100  _c1          = {0x3FA55555, 0x55555538}, /*  4.16666666666664631257e-02 */
101  _c2          = {0x3F811111, 0x31931950}, /*  8.33334279438858738230e-03 */
102  _c3          = {0x3F56C16C, 0x3DC3DC5E}; /*  1.38889030804716772510e-03 */
103 #else
104  ...
105 #endif
106 #define c0          _c0.d
107 #define c1          _c1.d
108 #define c2          _c2.d
109 #define c3          _c3.d
110 double P_r;
111
112 P_r = (c_0 + rp_hi * (c_1 + rp_hi * (c_2 + (rp_hi * c_3))));
```

◇ *Preuve.*
Nous avons :

- $P_0 = c\_3 \otimes rp\_hi$ donc $|P_0| \leq 2^{-18}$ et $P_0 = (c\_3 \times rp\_hi) + \varepsilon_{-71}$
- $P_1 = c\_2 \oplus P_0$ donc $|P_1| \leq 2^{-6}$ et $P_1 = (c\_2 + P_0) + \varepsilon_{-59}$
- $P_2 = P_1 \otimes rp\_hi$ donc $|P_2| \leq 2^{-15}$ et $P_2 = (P_1 \times rp\_hi) + \varepsilon_{-68}$
- $P_3 = c\_1 \oplus P_2$ donc $|P_3| \leq 2^{-4}$ et $P_3 = (c\_1 + P_2) + \varepsilon_{-57}$
- $P_4 = P_3 \otimes rp\_hi$ donc $|P_4| \leq 2^{-13}$ et $P_4 = (P_3 \times rp\_hi) + \varepsilon_{-66}$
- $P_5 = c\_0 \oplus P_4$ donc $|P_5| \leq 2^{-2}$ et $P_5 = (c\_0 + P_4) + \varepsilon_{-55}$

En combinant toutes ces erreurs nous obtenons :

<16> $|P\_r| \leq 2^{-2}$
$P\_r = (c\_0 + rp\_hi \times (c\_1 + rp\_hi \times (c\_2 + (rp\_hi \times c\_3)))) + \varepsilon_{-54} + \varepsilon_{-64}$
$P\_r = P(rp\_hi) + \varepsilon_{-54} + \varepsilon_{-64}$ •

Par les propriétés <15> et <16> :

<17> $\exp(r) = (1 + r + \frac{1}{2}r^2 + r^3.P\_r) + \varepsilon_{-78} + \varepsilon_{-80}$ pour $r \in [-2^{-9}, 2^{-9}]$ •

□

À l'issue de l'évaluation polynomiale on a :

$$\exp(x) = 2^k.(ex\_hi + ex\_lo).(1 + r + \frac{1}{2}r^2 + r^3.P\_r + \varepsilon_{-78} + \varepsilon_{-80} + \varepsilon_{-91}).(1 + \varepsilon_{-105})$$

5.8 Reconstruction

Au cours des différentes étapes nous avons obtenu les résultats suivants :
- k, r_hi et r_lo par la première réduction additive ;
- ex_hi, ex_lo, rp_hi et rp_lo par la deuxième réduction par table ;
- P_r par l'évaluation polynomiale.

L'étape de reconstruction consiste à fusionner tous ces résultats de façon à obtenir $\exp(x)$. Cette étape de reconstruction est basée sur la formule mathématique suivante :

$$\exp(x) = 2^k.(ex\_hi + ex\_lo).(1 + (rp\_hi + rp\_lo) + \frac{1}{2}.(rp\_hi + rp\_lo)^2 + (rp\_hi + rp\_lo)^3.P\_r)$$

Cependant, les termes de cette équation qui sont trop petits par rapport aux termes dominants ne sont pas pris en compte. Donc nous approchons :

$$Rec = (ex\_hi + ex\_lo).(1 + (rp\_hi + rp\_lo) + \frac{1}{2}.(rp\_hi + rp\_lo)^2 + (rp\_hi + rp\_lo)^3.P\_r)$$

par

$$Rec^* = ex\_hi \times (1 + rp\_hi + rp\_lo + \frac{1}{2}(rp\_hi)^2 + P\_r \times (rp\_hi)^3) + ex\_lo \times (1 + rp\_hi + \frac{1}{2}(rp\_hi)^2)$$

L'erreur correspondante est donnée par :

$$\begin{array}{l} Rec - Rec^* = \\ (ex\_hi + ex\_lo).(rp\_hi.rp\_lo + \frac{1}{2}rp\_lo^2) + \\ ex\_hi.rp\_lo.(3.rp\_hi^2 + 3.rp\_hi.rp\_lo + rp\_lo^2) + \\ ex\_lo.rp\_hi.(3.rp\_lo^2 + 3.rp\_hi.rp\_lo + rp\_hi^2) + ex\_lo.rp\_lo^3 + \\ ex\_lo.rp\_lo \\ \leq 2^{-75} \end{array}$$

Ce qui nous conduit à la propriété suivante :

<18> L'erreur commise lors de l'approximation de Rec par Rec^* est

$$Rec = Rec^* + \varepsilon_{-75}$$

•

L'ordre dans lequel sont exécutées les instructions est choisi pour minimiser l'erreur commise. Les différents termes et termes intermédiaires avec leur ordre de grandeur sont montrés en figure 5.2, page 101.

Listing 5.7: *Reconstruction*

```
113 double R1, R2, R3_hi, R3_lo, R4, R5_hi, R5_lo, R6, R7, R8, R9, R10, R11, crp_hi;
114
115
116 R1 = rp_hi * rp_hi;
117
118 crp_hi = R1 * rp_hi;
119 /* Correspond à R1 /= 2; */
120 HI(R1) = HI(R1)-0x00100000;
121
122 R2 =  P_r * crp_hi;
123
124 Dekker(R3_hi, R3_lo, ex_hi, rp_hi);
125 R4 = ex_hi * rp_lo;
126
127 Dekker(R5_hi, R5_lo, ex_hi, R1);
128 R6 = R4 + (ex_lo * (R1 + rp_hi));
129
130 R7  = ex_hi * R2;
131 R7 += (R6 + R5_lo) + (R3_lo + ex_lo);
132
133 Fast2Sum(R9, R8, R7, R5_hi);
134
135 Fast2Sum(R10, tmp, R3_hi, R9);
136 R8 += tmp;
137
138 Fast2Sum(R11, tmp, ex_hi, R10);
139 R8 += tmp;
140
141 Fast2Sum(R11, R8, R11, R8);
```

◇ *Preuve.*

ligne 116 $|rp\_hi| \leq 2^{-9}$ donc :

<19> $|R1| \leq 2^{-18}$,
$R1 = (rp\_hi)^2.(1 + \varepsilon_{-53})$ •

ligne 118 En utilisant la propriété <19> et $|rp\_hi| \leq 2^{-9}$ on a :

<20> $|crp\_hi| \leq 2^{-27}$,
$crp\_hi = (rp\_hi)^3.(1 + \varepsilon_{-52})$ •

ligne 120 Cette opération consiste en une division par 2, en soustrayant 1 à l'exposant. Elle est exacte et valide si et seulement si $R1$ n'est pas un dénormalisé, ce qui est le cas (Propriété <1> $|x| \geq 2^{-54}$, plus tableau 5.2 montrant que l'on a au maximum 58 bits de cancellation).

<21> $|R1| \leq 2^{-19}$,
$R1 = \frac{1}{2}(rp\_hi)^2.(1 + \varepsilon_{-53})$ •

ligne 122 En utilisant la propriété <16> et <20>

<22> $|R2| \leq 2^{-29}$,
$R2 = P\_r \times (crp\_hi).(1 + \varepsilon_{-53})$,
$R2 = P\_r \times (rp\_hi)^3.(1 + \varepsilon_{-51})$ •

ligne 124 En utilisant l'algorithme de Dekker (programme 3.7, page 49) et les propriétés <12> et <13> on a :

<23> $|R3\_hi| \leq 2^{-8}$ et $|R3\_lo| \leq 2^{-61}$,
$R3\_hi + R3\_lo = ex\_hi \times rp\_hi$ exactement •

ligne 125 En utilisant les propriétés <12> et <13> :

<24> $|R4| \leq 2^{-61}$,
$R4 = ex\_hi \times rp\_lo.(1 + \varepsilon_{-53})$ •

ligne 127 En utilisant l'algorithme de Dekker et les propriétés <13> et <21> on a :

<25> $|R5\_hi| \leq 2^{-18}$ et $|R5\_lo| \leq 2^{-71}$,
$(R5\_hi + R5\_lo) = ex\_hi \times R1$ exactement,
$(R5\_hi + R5\_lo) = (ex\_hi \times \frac{1}{2}(rp\_hi)^2).(1 + \varepsilon_{-53})$ •

ligne 128 Par les propriétés <13> et <21> :
$|R1 + rp\_hi| \leq 2^{-8}$,
$R1 \oplus rp\_hi = R1 + rp\_hi + \varepsilon_{-61}$,
$|ex\_lo \times (R1 + rp\_hi)| \leq 2^{-60}$,
$ex\_lo \otimes (R1 \oplus rp\_hi) = ex\_lo \times (\frac{1}{2}(rp\_hi)^2 + rp\_hi) + \varepsilon_{-112}$.
qui combiné à la propriété <24> nous donne :

<26> $|R6| \leq 2^{-59}$,
$R6 = R4 + \left(ex\_lo \times \left(\frac{1}{2}(rp\_hi)^2 + rp\_hi\right) + \varepsilon_{-112}\right) + \varepsilon_{-112}$
$R6 = (ex\_hi \times rp\_lo) + \left(ex\_lo \times \left(\frac{1}{2}(rp\_hi)^2 + rp\_hi\right)\right) + \varepsilon_{-111} + \varepsilon_{-114}$ •

ligne 130 En utilisant les propriétés <13> et <22> on a :

<27> $|R7| \leq 2^{-28}$,
$R7 = (ex\_hi \times R2).(1 + \varepsilon_{-53})$,
$R7 = (ex\_hi \times P\_r \times (rp\_hi)^3) + \varepsilon_{-79} + \varepsilon_{-80})$ •

ligne 131 Par les propriétés <25> et <26> on a :

<28> $|R6 + R5\_lo| \leq 2^{-58}$,
$R6 \oplus R5\_lo = R6 + R5\_lo + \varepsilon_{-111}$ ou
$R6 \oplus R5\_lo = (ex\_hi \times rp\_lo) + (ex\_lo \times (\frac{1}{2}(rp\_hi)^2 + rp\_hi)) + R5\_lo + \varepsilon_{-109}$
•

Par les propriétés <13> et <23> on a :

<29> $|R3\_lo + ex\_lo| \leq 2^{-51}$,
$R3\_lo \oplus ex\_lo = R3\_lo + ex\_lo + \varepsilon_{-104}$ •

Par les propriétés <28> et <29> on a :
$|R6 + R5\_lo + R3\_lo + ex\_lo| \leq 2^{-50}$ et
$(R6 \oplus R5\_lo) \oplus (R3\_lo \oplus ex\_lo) = (R6 \oplus R5\_lo) + (R3\_lo \oplus ex\_lo) + \varepsilon_{-103}$.
qui combiné à la propriété <27> donne :

<30> $|R7| \leq 2^{-27}$,

$$\begin{aligned} R7 &= (ex\_hi \times P\_r \times (rp\_hi)^3) + \varepsilon_{-79} + \varepsilon_{-80}) + \\ &\quad ((ex\_hi \times rp\_lo) + (ex\_lo \times (\tfrac{1}{2}(rp\_hi)^2 + rp\_hi)) + R5\_lo + \varepsilon_{-109}) + \\ &\quad (R3\_lo + ex\_lo + \varepsilon_{-104}) \\ &= ex\_hi \times (rp\_lo + P\_r \times (rp\_hi)^3) + \\ &\quad ex\_lo \times (1 + rp\_hi + \tfrac{1}{2}(rp\_hi)^2) + \\ &\quad R5\_lo + R3\_lo + \varepsilon_{-78} \end{aligned}$$

•

ligne 133 L'algorithme Fast2Sum nous garantit :

<31> $|R9| \leq 2^{-17}$ et $|R8| \leq 2^{-70}$,
$R7 + R5\_hi = R9 + R8$ exactement •

ligne 135 L'algorithme Fast2Sum nous garantit :

<32> $|R10| \leq 2^{-7}$ et $|tmp| \leq 2^{-60}$,
$R3\_hi + R9 = R10 + tmp$ exactement •

ligne 136 En utilisant les propriétés <31> et <32> :

<33> $|R8 + tmp| \leq 2^{-59}$,
$R8 = R8 + tmp + \varepsilon_{-112}$ •

ligne 138 L'algorithme Fast2Sum nous garantit :

<34> $|R11| \leq \sqrt{2} + 2^{-7}$ et $|tmp| \leq 2^{-52}$,
$R11 + tmp = ex\_hi + R10$ exactement •

ligne 139 En utilisant les propriétés <33> et <34> :

<35> $|R8 + tmp| \leq 2^{-51}$,
$R8 = R8 + tmp + \varepsilon_{-104}$ •

ligne 141 Ce dernier Fast2Sum sert à utiliser le mode d'arrondi par défaut du processeur pour cumuler les deux résultats. L'algorithme Fast2Sum nous garantit :

<36> $|R11| \leq 2^1$ et $|R8| \leq 2^{-52}$,
$R11 + R8 = R11 + R8$ exactement •

□

On a donc :

$$\begin{aligned}
R11 + R8 &= R11 + tmp + R8 + \varepsilon_{-104} && \text{<35>}\\
&= ex_hi + R10 + R8 + \varepsilon_{-104} && \text{<34>}\\
&= ex_hi + R10 + tmp + R8 + \varepsilon_{-104} + \varepsilon_{-112} && \text{<33>}\\
&= ex_hi + R3_hi + R9 + R8 + \varepsilon_{-103} && \text{<32>}\\
&= ex_hi + R3_hi + R7 + R5_hi + \varepsilon_{-103} && \text{<31>}\\
&= ex_hi + R3_hi + R5_hi +\\
&\quad ex_hi \times (rp_lo + P_r \times (rp_hi)^3) +\\
&\quad ex_lo \times (1 + rp_hi + \tfrac{1}{2}(rp_hi)^2) +\\
&\quad R5_lo + R3_lo + \varepsilon_{-78} + \varepsilon_{-103} && \text{<30>}\\
&= (R3_hi + R3_lo) + (R5_hi + R5_lo) +\\
&\quad ex_hi \times (1 + rp_lo + P_r \times (rp_hi)^3) +\\
&\quad ex_lo \times (1 + rp_hi + \tfrac{1}{2}(rp_hi)^2) + \varepsilon_{-78} + \varepsilon_{-103}\\
&= (R3_hi + R3_lo) +\\
&\quad ex_hi \times (1 + rp_lo + \tfrac{1}{2}(rp_hi)^2 + P_r \times (rp_hi)^3) +\\
&\quad ex_lo \times (1 + rp_hi + \tfrac{1}{2}(rp_hi)^2) + \varepsilon_{-71} + \varepsilon_{-77} && \text{<25>}\\
&= ex_hi \times (1 + rp_hi + rp_lo + \tfrac{1}{2}(rp_hi)^2 + P_r \times (rp_hi)^3) +\\
&\quad ex_lo \times (1 + rp_hi + \tfrac{1}{2}(rp_hi)^2) +\\
&\quad \varepsilon_{-71} + \varepsilon_{-77} && \text{<23>}
\end{aligned}$$

Par la propriété <18> nous obtenons :

$$\begin{aligned}
R11 + R8 &= (ex_hi + ex_lo).(1 + r + \tfrac{1}{2}.r^2 + r^3.P_r)\\
&\quad +\varepsilon_{-71} + \varepsilon_{-74}
\end{aligned}$$

Par construction des valeurs $(ex\_hi + ex\_lo)$, nous avons $|ex\_hi + ex\_lo| \leq \sqrt{2}$ donc,

<37> $|R11 + R8| < 2^1$
et
$\exp(x) = 2^k.(R11 + R8 + \varepsilon_{-71} + \varepsilon_{-74} + \varepsilon_{-91}).(1 + \varepsilon_{-105})$ •

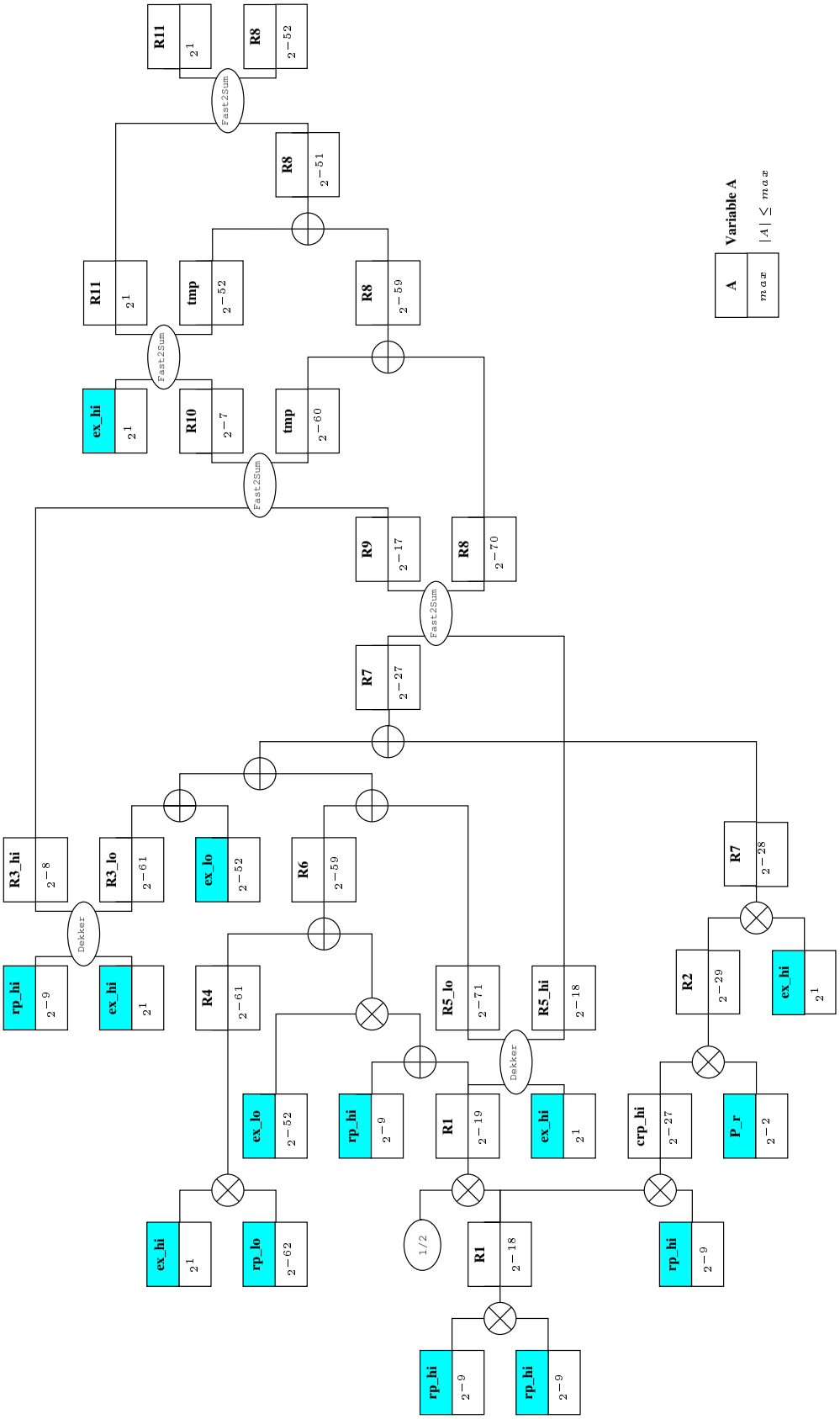


FIG. 5.2: *Étape de reconstruction*

5.9 Test si l'arrondi est possible

Nous devons maintenant arrondir le résultat correctement et effectuer la multiplication par 2^k, où k est un entier. Cette multiplication est exacte aussi nous avons :

$$\begin{aligned} \exp(x) &= \circ\left(2^k.(R11 + R8 + \varepsilon_{-71} + \varepsilon_{-74} + \varepsilon_{-91}).(1 + \varepsilon_{-105})\right) \\ &= 2^k.\circ\left((R11 + R8 + \varepsilon_{-71} + \varepsilon_{-74} + \varepsilon_{-91}).(1 + \varepsilon_{-105})\right) \end{aligned}$$

dans le cas ou le résultat ne représente pas un dénormalisé. En effet, si le résultat final est représenté par un dénormalisé, alors la précision du résultat est inférieure aux 53 bits d'un nombre normalisé. Prennons un exemple. Soit $a = 1.75$ un nombre flottant exactement représentable en double précision (nous avons $\circ(a) = a$). Multiplions ce nombre par 2^{-1074}, le résultat exacte est 1.75×2^{-1074} qui est différent du résultat arrondi $\circ(1.75 \times 2^{-1074}) = 2^{-1073}$.

Par conséquent, le cas des dénormalisés doit être traité à part, ce que nous ferrons dans les programmes suivants.

5.9.1 En arrondi au plus près

Listing 5.8: *Test si l'arrondi est possible en arrondi au plus près*

```
142 static const union{int i[2]; double d;}
143 #ifdef BIG_ENDIAN
144  _two1000     = {0x7E700000, 0x00000000}, /*  1.07150860718626732095e301   */
145  _twom1000    = {0x01700000, 0x00000000}, /*  9.33263618503218878990e-302  */
146  _errn        = {0x3FF00020, 0x00000000}, /*  1.00003051800000000000e0     */
147  _twom75      = {0x3B400000, 0x00000000}; /*  2.64697796016968855958e-23   */
148 #else
149  ...
150 #endif
151 #define two1000     _two1000.d
152 #define twom1000    _twom1000.d
153 #define errn        _errn.d
154 #define twom75      _twom75.d
155
156 int errd         = 73400320;                    /* 70 * 2^20 */
157
158 double R11_new, R11_err, R13, st2mem;
159 int    exp_R11;
160
161 union {int i[2]; long long int l; double d;} R12;
162
163
164 /* Résultat = (R11 + R8) */
165 if (R11 == (R11 + R8 * errn)){
166   if (k > -1020){
167     if (k < 1020){
168       HI(R11) += (k<<20);
169       return R11;
170     }else {
171       /* On est proche de + Inf */
172       HI(R11) += ((k-1000)<<20);
173       return R11*two1000;
174     }
175   }else {
176     /* On est dans les dénormalisés */
177     HI(R11_new) = HI(R11)+((k+1000)<<20);
178     LO(R11_new) = LO(R11);
179     R12.d       = R11_new * twom1000;
```

```
180
181     HI(st2mem) = R12.i[HI_ENDIAN];
182     LO(st2mem) = R12.i[LO_ENDIAN];
183
184     R11_err -= st2mem * two1000;
185     HI(R13) = HI(R11_err) & 0x7fffffff;
186     LO(R13) = LO(R11_err);
187
188     if (R13 == two_m75){
189       exp_R11 = (HI(R11) & 0x7ff00000) - errd;
190       if ((HI(R8) & 0x7ff00000) < exp_R11){
191         /* Arrondi difficile ! */
192         sn_exp(x);
193       }
194       /* Le terme d'erreur est exactement 1/2 ulp */
195       if ((HI(R11_err) > 0) && (HI(R8) > 0)) R12.l +=1;
196       else
197       if ((HI(R11_err) < 0) && (HI(R8) < 0)) R12.l -=1;
198     }
199     return R12.d;
200   }
201 }else {
202   /* Cas difficile */
203   sn_exp(x);
204 }
```

◇ *Preuve.*

ligne 165 Ce test permet de savoir si l'on est capable d'arrondir correctement. Pour plus d'explications sur ce test, nous renvoyons le lecteur à la section 3.3.
En utilisant la propriété <37> nous avons :

$$|(R11 + R8) \times 2^k - \exp(x)| \leq 2^{-70}$$

où $x \in [A, B]$. C'est pourquoi $errn = 1 + 2^{-70} \times 2^{55} = 1 + 2^{-15}$. Si ce test est vrai alors nous sommes capables d'arrondir correctement, sinon il faut faire appel à la multiprécision.

ligne 166-174 On est capable d'arrondir, il faut maintenant effectuer la multiplication $R11 \times 2^k$ de façon exacte. Pour une question de performance nous effectuerons une addition à l'exposant de $R11$. Toutefois pour que cette opération soit valide et exacte, il ne faut créer ni un dénormalisé ni un infini. C'est pourquoi nous effectuons un test sur la valeur de k.
$(R11 + R8) > 2^{-2}$ donc $2^k.(R11 + R8)$ ne créera pas de dénormalisé si $k > -1020$.
$(R11+R8) < 2^3$ donc $2^k.(R11+R8)$ ne créera pas de dépassement de capacité si $k < 1020$. Nous avons donc $R11 = R11 \oplus R8$
Dans le cas ou nous allons éventuellement rendre comme résultat un overflow, alors nous rendons la valeur de k plus petite, en lui soustrayant 1000 de façon à ne pas générer de cas exceptionnel lors de la multiplication par k. Ce résultat est multiplié exactement (puissance de 2) par le nombre flottant $twom1000 = 2^{-1000}$, et laisse à l'ensemble système, compilateur et processeur la prise en charge des overflows.

ligne 177 Le résultat est potentiellement un dénormalisé. Il faut mettre en place une procédure spéciale pour tester si l'arrondi s'est fait correctement, et si nous avons été capable d'arrondir.

$$R11 = R11 \times 2^{k+1000}$$

ligne 179 Nous obtenons le résultat en arrondi au plus près.

$$R12 = R11 \otimes 2^{-1000} = R11 \times 2^{-1000} + \varepsilon_{-1075}$$

Le terme d'erreur ε_{-1075} provient de la possible troncature lorsque le résultat appartient aux dénormalisés.

ligne 181,182 La plupart des processeurs ne gèrent par les dénormalisés en «matériel», ils sont traités lors de leur écriture en mémoire. Cela signifie qu'un nombre mémorisé dans un registre peut potentiellement représenter un dénormalisé et contenir toute la précision. Donc ces lignes empèchent le compilateur de faire des optimisations «dangereuses», et empêchent également d'avoir trop de précision en forçant $R12$ à transiter par la mémoire. Ces lignes pourraient être supprimées par exemple avec l'utilisation du compilateur gcc et du drapeau `-ffloat-store` (voir discussion en section 1.1.4, page 9). Toutefois ce drapeau force chaque opération flottante à transiter par la mémoire, et a pour conséquence de détériorer les performances du code résultant. Donc la solution pour conserver de bonnes performances est de forcer «manuellement» le passage par la mémoire, pour avoir une gestion des dénormalisés en accord avec la norme IEEE-754.

ligne 184 Appelons

$$R11\_err = R11 \ominus R12 \otimes 2^{1000}$$

Par le lemme de Sterbenz, et par le fait qu'une multiplication par une puissance de 2 est exacte nous avons :

$$R11\_err = R11 - R12 \times 2^{1000}$$

A l'issue de cette opération le terme $R11\_err$ correspond à l'erreur commise lors de la multiplication de $R11$ par 2^{-1000} (ligne 179).

ligne 185,186 Supprime l'information de signe de $R11\_err$

$$R13 = |R11\_err|$$

ligne 188 Teste si $R13$ est exactement égal à l'erreur absolue (qui est également l'erreur relative) commise lors de l'arrondi, en arrondi au plus près, dans les dénormalisés, multiplié par 2^{1000} :

$$2^{-1075} \times 2^{1000} = 2^{-75}$$

Montrons que si le terme d'erreur est strictement inférieur à $1/2ulp(R12)$ alors $R12$ représente bien l'arrondi correct de $R11 + R8$.
Si $|R11\_err| < \frac{1}{2}ulp(R12)$ alors
$|R11\_err| \leq \frac{1}{2}ulp(R12) - ulp(R11)$ et $|R8| \leq \frac{1}{2}ulp(R11)$ donc :

$$|R11\_err + R8| \leq \frac{1}{2}ulp(R12) - ulp(R11) + \frac{1}{2}ulp(R11)$$

donc :

$$|R11\_err + R8| < \frac{1}{2}ulp(R12)$$

Aussi $R12$ représente bien l'arrondi correct de $R11 + R8$ si $|R11\_err| < \frac{1}{2}ulp(R12)$.
En revanche si $|R11\_err| = \frac{1}{2}ulp(R12) = 2^{-75}$, lors de la multiplication $R11 \times 2^{-1000}$, le résultat est arrondi de façon paire/impaire. Dans ce cas $R12$ peut ne pas correspondre à l'arrondi au plus près de $R11+R8$, il nous faut corriger le résultat.

- Si $R8 > 0$, et $R11\_err = -\frac{1}{2}ulp(R12)$, alors l'arrondi pair/impair s'est fait du bon coté.
- Si $R8 > 0$, et $R11\_err = \frac{1}{2}ulp(R12)$, alors l'arrondi pair/impair ne s'est pas fait du bon coté. Il faut corriger en ajoutant $1ulp$ à $R12$ (figure 5.3, cas a)
- Si $R8 < 0$, et $R11\_err = -\frac{1}{2}ulp(R12)$, alors l'arrondi pair/impair ne s'est pas fait du bon coté. Il faut corriger en soustrayant $1ulp$ à $R12$ (figure 5.3, cas b).
- Si $R8 < 0$, et $R11\_err = \frac{1}{2}ulp(R12)$, alors l'arrondi pair/impair s'est fait du bon coté.

ligne 190 Lorsque qu'il existe une suite de 0 ou 1 consécutif, à cheval entre $R11$ et $R8$ alors le test de la ligne 165 ne détecte pas un éventuel problème. Ce problème se manifeste avec les dénormalisés, lorsque $R11\_err$ est proche de $\frac{1}{2}ulp(R12)$. Nous devons donc tester si dans ce cas (dénormalisé, $|R11\_err| = \frac{1}{2}ulp(R12)$) nous disposons d'assez de précision pour arrondir le résultat. Nous utilisons un test similaire à celui utilisé pour les arrondis dirigés. En effet les cas problématiques sont lorsque $\frac{1}{2}ulp(R12) - 2^{-70}.R11 \leq |R11\_err + R8| \leq \frac{1}{2}ulp(R12) + 2^{-70}.R11$.

ligne 195,197 Teste si nous sommes en présence d'un des cas énoncés précédemment, et corrige le résultat en ajoutant ou soustrayant $1ulp$ comme expliqué dans le cas de la propriété 2, page 7.

□

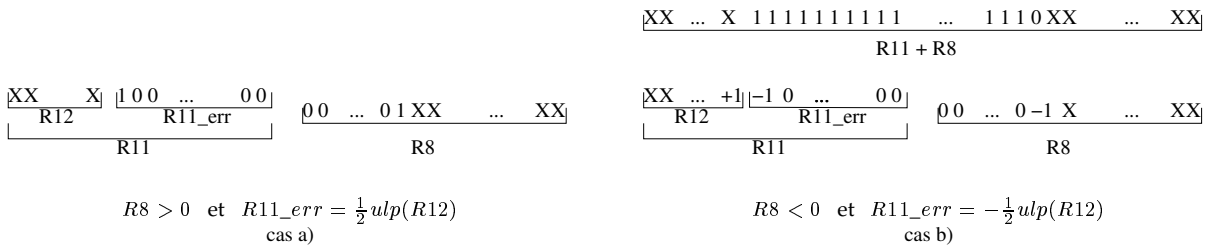


FIG. 5.3: *Description des problèmes liés à l'arrondi d'un résultat dénormalisé*

5.9.2 En arrondi vers $+\infty$

Listing 5.9: *Test si l'arrondi est possible vers* $+\infty$

```
205 static const union{int i[2]; double d;}
206 #ifdef BIG_ENDIAN
207  _two1000     = {0x7E700000, 0x00000000}, /*  1.0715086071862673209 5e301  */
208  _twom1000    = {0x01700000, 0x00000000}, /*  9.3326361850321887899 0e-302 */
209 #else
210  ...
211 #endif
212 #define two1000     _two1000.d
213 #define twom1000    _twom1000.d
214
215
216 int errd         = 73400320;                    /* 70 * 2^20 */
217
218 int    exp_R11;
219 union {int i[2]; long long int l; double d;} R12;
220
221 /* Résultat = (R11 + R8) */
222
223 exp_R11 = (HI(R11) & 0x7ff00000) - errd;
224
225 if ((HI(R8) & 0x7ff00000) > exp_R11){
226   /* On est capable d'arrondir */
227   if (k > -1020){
228     if (k < 1020){
229       HI(R11) += (k<<20);
230     }else {
231       /* On est proche de + Inf */
232       HI(R11) += ((k-1000)<<20);
233       R11 *= two1000;
234     }
235     if (HI(R8) > 0){
236       R12.d  = R11;
237       R12.l += 1;
238       R11    = R12.d;
239     }
240     return R11;
241   }else {
242     /* On est dans les dénormalisés */
243     HI(R11) += ((k+1000)<<20);
244     R12.d    = R11 * twom1000;
245
246     HI(st2mem) = R12.i[HI_ENDIAN] ;
247     LO(st2mem) = R12.i[LO_ENDIAN]
248
```

```
249     R11 -= st2mem * two1000;
250     if ((HI(R11) > 0)||((HI(R11) == 0)&&(HI(R8) > 0))) R12.l += 1;
251
252     return R12.d;
253   }
254 }else {
255   /* Cas difficile */
256   su_exp(x);
257 }
```

◇ *Preuve.*

Le code pour tester si l'arrondi correct est possible en arrondi vers $+\infty$ est proche de celui utilisé pour l'arrondi au plus près.

ligne 225 Ce test correspond à celui décrit dans le programme le programme 3.12 page 53. Ce test est valide même si le résultat final est un dénormalisé.

ligne 235 Nous ajoutons $1ulp$ au résultat si $R8$ est positif.

ligne 249 Comme dans le cas de l'arrondi au plus près, la quantité $R11$ représente l'erreur commise lors de l'arrondi de l'opération $R11 * twom1000$ en ligne 244.

ligne 250 Ce test détecte si l'erreur lors de l'arrondi en ligne 244 est strictement positif ou si elle est égal à zéro et si $R8$ est strictement positif. Si nous sommes dans l'un de ces deux cas, par définition de l'arrondi vers $+\infty$, il faut ajouter $1ulp$ au résultat.

□

5.9.3 En arrondi vers $-\infty$

Listing 5.10: *Test si l'arrondi est possible en arrondi vers* $-\infty$

```
144 static const union{int i[2]; double d;}
145 #ifdef BIG_ENDIAN
146  _two1000     = {0x7E700000, 0x00000000}, /*  1.0715086071862673209 5e301   */
147  _twom1000    = {0x01700000, 0x00000000}, /*  9.3326361850321887899 0e-302 */
148 #else
149  ...
150 #endif
151 #define two1000     _two1000.d
152 #define twom1000    _twom1000.d
153
154 int errd         = 73400320;                    /* 70 * 2^20 */
155
156 int    exp_R11;
157 union {int i[2]; long long int l; double d} R12;
158
159 /* Résultat = (R11 + R8) */
160
161 exp_R11 = (HI(R11) & 0x7ff00000) - errd;
162
163 if ((HI(R8) & 0x7ff00000) > exp_R11){
164   /* On est capable d'arrondir */
165   if (k > -1020){
166     if (k < 1020){
167       HI(R11) += (k<<20);
```

```
168       }else {
169         /* On est proche de + Inf */
170         HI(R11) += ((k-1000)<<20);
171         R11 *= two1000;
172       }
173       if (HI(R8) > 0){
174         R12.d  = R11;
175         R12.l += 1;
176         R11    = R12.d;
177       }
178       return R11;
179     }else {
180       /* On est dans les dénormalisés */
181       HI(R11) += ((k+1000)<<20);
182       R12.d    = R11 * twom1000;
183
184       HI(st2mem) = R12.i[HI_ENDIAN];
185       LO(st2mem) = R12.i[LO_ENDIAN];
186
187       R11 -= st2mem * two1000;
188       if ((HI(R11) < 0)||((HI(R11) == 0)&&(HI(R8) < 0))) R12.l -= 1;
189
190       return R12.d;
191     }
192   }else {
193     /* Cas difficile */
194     su_exp(x);
195   }
```

◇ *Preuve.* Ce code pour tester si l'arrondi correct est possible en arrondi vers $-\infty$ est presque identique au précédent et fonctionne pour les même raisons. □

5.9.4 En arrondi vers 0

Le code pour déterminer si l'on est capable d'arrondi vers 0 est identique à celui utilisé pour l'arrondi vers $-\infty$ car la fonction $\exp(x)$ est croissante et positive pour tout x.

5.10 Phase précise

Lorsque l'évaluation précédente a échoué, alors l'arrondi du résultat est difficile à obtenir. Nous devons utiliser des méthodes plus précises :

- sn_exp pour l'arrondi au plus près,
- su_exp pour l'arrondi vers $+\infty$,
- sd_exp pour l'arrondi vers $-\infty$,

Pour cela nous utilisons la bibliothèque SCS (voir chapitre 4) avec 30 bits de précision par chiffre et 8 chiffres par vecteur. La précision garantie par ce format est de 210 bits au minimum bien qu'il n'existe aucune preuve du bon comportement de ces opérateurs. Toutefois la preuve de l'arrondi correct de l'exponentielle n'utilise que les propriétés suivantes qui sont faciles à vérifier et/ou à satisfaire :

Propriété 5 *(Addition) Nous noterons par $a \boxplus b$ l'opération multiprécision effectuant l'addition entre a et b avec au minimum 211 bits de précision dans le résultat. A l'image de l'addition flottante double précision, l'addition SCS peut conduire à une cancellation. Nous avons :*

$$a + b = (a \boxplus b).(1 + \varepsilon_{-210})$$

Propriété 6 *(Multiplication) Nous noterons par $a \boxtimes b$ l'opération multiprécision effectuant la multiplication entre a et b avec au minimum 207 bits de précision dans le résultat. Cette opération ne produit pas de cancellation.*

$$a \times b = (a \boxtimes b).(1 + \varepsilon_{-207})$$

5.10.1 Description de l'algorithme

Voici l'algorithme utilisé pour la deuxième partie de l'évaluation :

1. **Pas de gestion des cas spéciaux**
   Pour cette partie en multiprécision, nous supposons que tous les tests sur la gestion des cas spéciaux a déjà été effectuée.
2. **Réduction d'argument «mathématique»**
   On calcule l'argument réduit r et l'entier k tel que :
   $$r = \frac{x - k.\ln(2)}{512}$$
   avec $\frac{-\ln(2)}{1024} \leq r \leq \frac{-\ln(2)}{1024}$
   de façon que
   $$\exp(x) = \exp(r)^{512} \times 2^k$$
3. **Évaluation polynomiale**
   Nous évaluons le polynôme $P(r)$ de degré 11 tel que :
   $$\exp(r) = (1 + r + P(r)).(1 + \varepsilon_{-179})$$
4. **Mise à la puissance**
   $$\exp(r)^{512} = \left(\left(\left(\left(\left(\left(\left(\left(\exp(r)^2\right)^2\right)^2\right)^2\right)^2\right)^2\right)^2\right)^2\right)^2$$
5. **Reconstruction**
   $$\exp(x) = \exp(r)^{512}.2^k.(1 + \varepsilon)$$
   avec $|\varepsilon| \leq 2^{-170}$

Nous avons choisi ce schéma d'évaluation, car l'étape de reconstruction utilise l'opérateur multiprécision de mise au carré. Cet opérateur facilite les calculs d'erreurs et est très économique : il coûte environ 0.7 fois le coût d'une multiplication multiprécision.

Nous noterons qu'il existe une alternative à l'étape de mise à la puissance, qui consiste à tabuler les valeurs $2^{\frac{N}{512}}$ pour $N = 1, 2, \ldots, 511$ et d'utiliser la formule $\exp(x) = \exp(r) \times 2^N \times 2^{\frac{M}{512}}$ avec $k = M + N/512$. Nous n'utilisons pas cette méthode car un nombre SCS nécessite beaucoup d'espace mémoire.

5.10.2 Appel des fonctions

D'après les pires cas de Lefèvre nous avons le théorème suivant :

Théorème 7 *(Arrondi correct de l'exponentielle) Soit y la valeur exacte de l'exponentielle d'un nombre virgule flottante double précision x. Soit y^* une approximation de y telle que la distance entre y et y^* soit majorée par ε. Alors si $\varepsilon \leq 2^{-157}$, pour chacun des quatre modes d'arrondi, arrondir y^* est équivalent à arrondir y ;*

Pour arrondir le résultat multiprécision, au minimum 157 bits, nous utilisons les fonctions SCS qui donnent l'arrondi d'un nombre SCS (`scs_get_d`, `scs_get_d_pinf`, `scs_get_d_minf`).

5.10.2.1 En arrondi au plus près

Listing 5.11: *Arrondi de l'exponentielle en multiprécision au plus près*

```
1 double sn_exp(double x){
2   scs_t res_scs;
3   scs_db_number res;
4
5   exp_SC(res_scs, x);
6   scs_get_d(&res.d, res_scs);  res.d = x;
7
8   return res.d;
9 }
```

5.10.2.2 En arrondi vers $+\infty$

Listing 5.12: *Arrondi de l'exponentielle en multiprécision vers $+\infty$*

```
1 double su_exp(double x){
2   scs_t res_scs;
3   scs_db_number res;
4
5   exp_SC(res_scs, x);
6   scs_get_d_pinf(&res.d, res_scs);
7   return res.d;
8 }
```

5.10.2.3 En arrondi vers $-\infty$

Listing 5.13: *Arrondi de l'exponentielle en multiprécision vers $-\infty$*

```
1 double sd_exp(double x){
2   scs_t res_scs;
3   scs_db_number res;
4
5   exp_SC(res_scs, x);
6   scs_get_d_minf(&res.d, res_scs);
7   return res.d;
8 }
```

5.10.3 Programme

La fonction exp_SC va évaluer l'exponentielle de x avec 170 bits de précision et placer le résultat dans res_scs.

Listing 5.14: *Calcul de l'exponentielle en multiprécision*

```
1  void exp_SC(scs_ptr res_scs, double x){
2    scs_t sc1, red;
3    scs_db_number db;
4    int i, k;
5
6
7    /* db.d = x/512  (= 2^9)  */
8
9    db.d = x;
10   db.i[HI_ENDIAN] -= (9 << 20);
11   scs_set_d(sc1, db.d);
12
13
14   DOUBLE2INT(k, (db.d * iln2_o512.d));
15
16   /* 1) Réduction d'argument */
17
18   scs_set(red,     sc_ln2_o512_ptr_1);;
19   scs_set(red_low, sc_ln2_o512_ptr_2);
20   if (k>0){
21     scs_mul_ui(red,      (unsigned int) k);
22     scs_mul_ui(red_low,  (unsigned int) k);
23   }else {
24     scs_mul_ui(red,      (unsigned int)(-k));
25     scs_mul_ui(red_low,  (unsigned int)(-k));
26     red->sign *= -1;
27     red_low->sign *=-1;
28   }
29
30   scs_sub(red, sc1, red);
31   scs_sub(red, red, red_low);
32
33
34   /* 2) Evaluation polynomiale */
35
36   scs_mul(res_scs, constant_poly_ptr[0], red);
37   for(i=1; i< 11; i++){
38     scs_add(res_scs, constant_poly_ptr[i], res_scs);
39     scs_mul(res_scs, red, res_scs);
40   }
41
42   scs_add(res_scs, SCS_ONE, res_scs);
43   scs_mul(res_scs, red, res_scs);
44   scs_add(res_scs, SCS_ONE, res_scs);
45
46   /* 3) Mise a la puissance exp(r)^512  */
47
48   for(i=0; i<9; i++){
49     scs_square(res_scs, res_scs);
50   }
51
52   /* 4) Multiplication par 2^k */
53
54   res_scs->index += (int)(k/30);
55   if ((k%30) > 0)
56     scs_mul_ui(res_scs, (unsigned int) (1<<((k%30))));
```

```
57    else if ((k%30) < 0){
58      res_scs->index --;
59      scs_mul_ui(res_scs, (unsigned int) (1<<((30+(k%30)))));
60    }
61  }
```

◇ *Preuve.*

ligne 9 $db.d = x$

ligne 10 Cette opération divise $db.d$ par $512 = 2^9$. Elle est valide sous condition que $db.d$ et par conséquent que x ne représente par une valeur spéciale (dénormalisé, infini, NaN). Cette condition est vérifiée car les cas spéciaux ont déjà été éliminés au cours de la première étape.

ligne 11 $sc1$ est un nombre multiprécision sur 211 bits qui vérifie :

<38> $sc1 = db.d = \frac{x}{512}$ exactement •

ligne 14 $iln2\_o512.d$ est un flottant double précision tel que : $iln2\_o512.d = \frac{512}{\ln 2}(1 + \varepsilon_{-53})$. Cette ligne place dans k l'entier le plus proche de $db.d \otimes \frac{512}{\ln 2}$. On se sert de la propriété de $DOUBLE2INT$ qui convertit un flottant en entier avec un arrondi au plus près (programme 3.10, page 50).
De plus k vérifie la propriété suivante :

<39> $-1075 \leq |k| \leq 1025$,
$\lfloor \frac{x}{\ln 2} \rfloor \leq k \leq \lceil \frac{x}{\ln 2} \rceil$ •

et k s'écrit sur au plus 11 bits.

ligne 18, 19 Par construction on a :

$$red + red\_low = \frac{\ln 2}{512}(1 + \varepsilon_{-450})$$

et red est construit de façon à ce que la multiplication de red par k soit exacte si $|k| \leq 2^{11}$.

ligne 28 A la fin du test sur k nous avons : $red + red\_low = k \otimes \frac{\ln 2}{512}(1 + \varepsilon_{-450})$
avec $|k| < 2^{11}$, donc :

<40> $red + red\_low = k \times \frac{\ln 2}{512}(1 + \varepsilon_{-411})$ •

ligne 30,31 Par les propriétés <38> et <40> nous avons :

<41> $red = \frac{x}{512} \ominus \left(k \times \frac{\ln 2}{512}\right)(1 + \varepsilon_{-411})$ •

De plus nous avons vu au cours de la première partie qu'il pouvait y avoir au plus 58 bits cancellés, lors de cette soustraction.

<42> $|red| \leq \frac{\ln 2}{1024} \leq 2^{-10}$,
$red = \frac{x}{512} - k \times \frac{\ln 2}{512} + \varepsilon_{-210}$ •

ligne 34-44 Nous effectuons l'évaluation polynomiale dont les coefficients utilisés ont les propriétés suivantes :

<43> $|constant\_poly\_ptr[0] = c_0| \leq 2^{-28}$, $|constant\_poly\_ptr[1] = c_1| \leq 2^{-25}$,
$|constant\_poly\_ptr[2] = c_2| \leq 2^{-21}$, $|constant\_poly\_ptr[3] = c_3| \leq 2^{-18}$,
$|constant\_poly\_ptr[4] = c_4| \leq 2^{-15}$, $|constant\_poly\_ptr[5] = c_5| \leq 2^{-12}$,
$|constant\_poly\_ptr[6] = c_6| \leq 2^{-9}$, $|constant\_poly\_ptr[7] = c_7| \leq 2^{-6}$,
$|constant\_poly\_ptr[8] = c_8| \leq 2^{-4}$, $|constant\_poly\_ptr[9] = c_9| \leq 2^{-2}$,
$|constant\_poly\_ptr[10] = c_{10}| \leq 2^{-1}$ •

Nous avons :
- $P_0 = c_1 \boxplus (red \boxtimes c_0)$ donc $|P_0| \leq 2^{-24}$ et $P_0 = (c_1 + (red \times c_0))(1 + \varepsilon_{-210} + \varepsilon_{-244})$
- $P_1 = c_2 \boxplus (red \boxtimes P_0)$ donc $|P_1| \leq 2^{-20}$ et $P_1 = (c_2 + (red \times P_0))(1 + \varepsilon_{-210} + \varepsilon_{-240})$
- $P_2 = c_3 \boxplus (red \boxtimes P_1)$ donc $|P_2| \leq 2^{-17}$ et $P_2 = (c_3 + (red \times P_1))(1 + \varepsilon_{-210} + \varepsilon_{-236})$
- $P_3 = c_4 \boxplus (red \boxtimes P_2)$ donc $|P_3| \leq 2^{-14}$ et $P_3 = (c_4 + (red \times P_2))(1 + \varepsilon_{-210} + \varepsilon_{-233})$
- $P_4 = c_5 \boxplus (red \boxtimes P_3)$ donc $|P_4| \leq 2^{-11}$ et $P_4 = (c_5 + (red \times P_3))(1 + \varepsilon_{-210} + \varepsilon_{-230})$
- $P_5 = c_6 \boxplus (red \boxtimes P_4)$ donc $|P_5| \leq 2^{-8}$ et $P_5 = (c_6 + (red \times P_4))(1 + \varepsilon_{-210} + \varepsilon_{-228})$
- $P_6 = c_7 \boxplus (red \boxtimes P_5)$ donc $|P_6| \leq 2^{-5}$ et $P_6 = (c_7 + (red \times P_5))(1 + \varepsilon_{-210} + \varepsilon_{-224})$
- $P_7 = c_8 \boxplus (red \boxtimes P_6)$ donc $|P_7| \leq 2^{-3}$ et $P_7 = (c_8 + (red \times P_6))(1 + \varepsilon_{-210} + \varepsilon_{-221})$
- $P_8 = c_9 \boxplus (red \boxtimes P_7)$ donc $|P_8| \leq 2^{-1}$ et $P_8 = (c_9 + (red \times P_7))(1 + \varepsilon_{-210} + \varepsilon_{-219})$
- $P_9 = c_{10} \boxplus (red \boxtimes P_8)$ donc $|P_9| \leq 1 + 2^{-10}$ et $P_9 = (c_{10} + (red \times P_8))(1 + \varepsilon_{-210} + \varepsilon_{-217})$
- $P_{10} = 1 \boxplus (red \boxtimes P_9)$ donc $|P_{10}| \leq 1 + 2^{-9}$ et $P_{10} = (1 + (red \times P_9))(1 + \varepsilon_{-210} + \varepsilon_{-216})$
- $P_{11} = 1 \boxplus (red \boxtimes P_{10})$ donc $|P_{11}| \leq 1 + 2^{-8}$ et $P_{11} = (1 + (red \times P_{10}))(1 + \varepsilon_{-210} + \varepsilon_{-216})$

Donc

$$res\_scs = P_{11} + \varepsilon_{-209}$$

On sait par construction du polynôme que

$$\exp(r) = (1 + r + c_10.r^2 + \cdots + c_0.r^{12}).(1 + \varepsilon_{-179})$$

donc
$|res\_scs| \leq 1 + 2^{-8}$ et
$\exp(x) = 2^k.(exp(r))^{512} = 2^k.(res\_scs.(1 + \varepsilon_{-209}).(1 + \varepsilon_{-179}))^{512}$

ligne 48 Nous élevons le résultat précédent au carré 9 fois, ce qui correspond à une mise à la puissance 512. A chaque itération nous commettons une erreur d'arrondi relative égale à ε_{-207}.
Finalement :
$|res\_scs| \leq 2^3$ et
$\exp(x) = 2^k.\exp(r)^{512}.(1+\varepsilon_{-170})$

ligne 54-60 Par ces lignes nous effectuons une multiplication de res_scs par 2^k. Cette multiplication se fait par un décalage sur l'index de $k/30$, où 30 correspond au nombre de bits utilisés dans un chiffre multiprécision. Ce décalage est une opération exacte. Enfin une multiplication de res_scs par deux à la puissance le reste de la division de k par 30 est effectuée. À l'issue de cette opération nous avons :

$$\exp(x) = (res\_scs).(1+\varepsilon_{-170})$$

□

Nous obtenons une approximation de la fonction exponentielle avec une erreur relative inférieure à 2^{-170}. Donc la deuxième étape basée sur la bibliothèque multiprécision SCS retourne l'arrondi correct de l'exponentielle pour les quatre modes d'arrondi, pour tout argument en double précision.

5.11 Tests

Nous venons de présenter et de prouver l'arrondi correct du code de la fonction exponentielle tel qu'il apparaît dans la bibliothèque *crlibm*(acronyme de *correctly rounded mathematical library*). Cette bibliothèque regroupe la bibliothèque multiprécision *scslib* (voir chapitre 4), ainsi que les fonctions déjà implémentées : première et deuxième partie de l'exponentielle ainsi que les deuxièmes parties du logarithme (en base e, 2, 10), du sinus, du cosinus, de la tangente, de l'arctangente, des fonctions hyperboliques (sur lesquelles nous travaillons actuellement) et de la réduction d'argument trigonométrique.

Comme dans le cas de *scslib*, nous avons utilisé les outils de compilation automake/autoconf. Ces outils nous permettent de rendre *crlibm* facilement installable et configurable sur une grande variété de machines et systèmes. Nous avons testé et compilé avec succès la bibliothèque *crlibm* sur les configurations suivantes :

- Pentium III avec Debian GNU/Linux gcc
- UltraSPARC IIi avec SunOS 5.8 et gcc
- Pentium 4 avec Debian GNU/Linux et gcc
- PowerPC G4 avec MacOS 10.2 et gcc
- Itanium avec Debian GNU/Linux et gcc

Pour comparer notre bibliothèque avec les solutions déjà existantes nous avons cherché les bibliothèques mathématiques les plus couramment utilisées. Le choix de bibliothèques fut difficile, tant leur nombre est important. Sans être exhaustifs nous pouvons citer la bibliothèque *fdlibm* de chez Sun disponible sur le site *Netlib* à l'adresse `http ://www.netlib.org`, *libmultim* produite par IBM [2], une bibliothèque spécifique aux Itanium et Pentium IV de chez Intel [40], la bibliothèque mathématique produite par HP pour les systèmes HP-UX sur Itanium [63].

Parmi toutes ces bibliothèques nous avons sélectionnée les bibliothèques suivantes pour établir des comparaisons :

- **libm :**
  Nous désignons par *libm* la bibliothèque mathématique fournie par défaut sur le système. Dans le cas de Linux, il sagit de celle inclue dans $glibc$, celle de Sun sur UltraSparc, et pour MacOS celle développée par Apple.
- **libm_intel :**
  Pour l'Itanium, la bibliothèque par défaut fournie avec $glibc$ est celle développée au sein des laboratoires d'Intel spécialement pour ce type d'architecture. Cette bibliothèque est écrite en assembleur et décrite dans [40].
- **MPFR :**
  MPFR est une bibliothèque multiprécision avec arrondi correct [3]. Elle propose les fonctions élémentaires suivantes : $\arctan$, $\arcsin$, $\arccos$, $\exp 2$, $\exp$, $\exp m1$, $\log$, $\log 2$, $\log 10$, pow, $\sin$, $\cos$, $\tan$, $\sinh$, $\cosh$, $\tanh$, $\mathrm{arcsinh}$, $\mathrm{arccosh}$, $\mathrm{arctanh}$. Parmi les bibliothèques testées, MPFR est la seule à proposer l'arrondi correct pour tous les modes d'arrondi définis par la norme.
- **libultim :**
  C'est le nom de la bibliothèque mathématique du projet *MathLib* de chez IBM [2]. Cette bibliothèque est censée fournir l'arrondi correct, en arrondi au plus près, pour les fonctions $\arctan$, $\arctan 2$, $\arcsin$, $\arccos$, $\exp$, $\log$, pow, $\sin$, $\cos$, $\tan$, $\sqrt{}$.

5.11.1 La mesure du temps de bibliothèques mathématiques

Pour tester la fonction exponentielle de *crlibm*, nous avons effectué des tests sur des valeurs tirées aléatoirement entre -745 et $+744$. Le nombre d'appels consécutifs à la fonction exponentielle est également un des paramètres pris en compte pour les tests.

Pour mesurer le temps précisément nous avons utilisé des routines assembleur sur toutes les architectures, à l'exception du PowerPC. Chaque mesure correspond au temps minimum de 10 appels à la même valeur. Nous minimisons ainsi l'impact des interruptions systèmes.

Le temps moyen correspond à une moyenne sur 10^6 appels consécutifs sur des données tirées aléatoirements. Nous avons également inclu le temps maximum pris par les évaluations sur les 10^6 valeurs aléatoires. Enfin, nous avons également testé les bibliothèques avec arrondi correct (MPFR, *crlibm* et *libtulim*) sur le pire cas en arrondi au plus près :

$$x = 7.541752774995950085206221e - 10$$

Ce cas nécessite de disposer du résultat avec au minimum 112 bits de précision. Par conséquent, le temps mis pour traiter ce cas constitue une borne supérieure.

5.11.2 Commentaires

5.11.2.1 Les bibliothèques spécifiques à une architecture

Sans surprise, la bibliothèque écrite en assembleur pour l'Itanium affiche d'excellentes performances. En effet, les documents constructeur annoncaient une latence de 48 cycles, et nous observons une latence de 131 cycles. Le temps pris pour l'appel de la fonction est donc relativement important par rapport aux 48 cycles d'évaluation. De plus, l'existence d'un chemin d'exécution plus long que les autres conduit à un temps d'exécution maximum de 2767 cycles.

	libm		crlibm	MPFR	libultim
	# erreur	temps			
UltraSparc IIi	$\frac{1}{41}$	1	2.66	83	1.2
		1	312	434	9331
Itanium	$\frac{1}{491}$	1	2.6	153	1.8
		1	49	254	22357
Pentium III	$\frac{1}{587}$	1	1.21	45	0.9
		1	35	86	3443
PowerPC G4	$\frac{1}{4739}$	1	0.91	13.5	0.93
		1	10	29.5	1677

TAB. 5.9: *Temps de l'évaluation de l'exponentielle normalisé rapporté au temps pris par la libm par défaut sur différents systèmes. Pour chaque processeur, la première ligne correspond au temps en moyenne, et la seconde au temps pris dans le pire cas en arrondi au plus près. Nous avons également inclus la fréquence avec laquelle la libm renvoie un mauvais arrondi.*

Architecture	Bibliothèque	libm	crlibm	MPFR	libultim
Pentium III (en cycles)	Tps moyen	462	562	21114	413
	Tps maximum	837	18413	43316	66050
	Tps pire cas	448	15963	38969	1542415
Xeon (en cycles)	Tps moyen	982	1178	24991	942
	Tps maximum	3124	34236	138768	108920
	Tps pire cas	1080	30384	50436	2592660
Itanium (en cycles)	Tps moyen	202	518	30995	371
	Tps maximum	2767	9349	70660	139415
	Tps pire cas	131	6434	33388	2928718
UltraSparc IIi (Unité arbitraire)	Tps moyen	762	2033	63570	950
	Tps maximum	9823	112366	292129	157987
	Tps pire cas	292	91190	126827	2724912
PowerPC G4 (Unité arbitraire)	Tps moyen	2.27	2.07	30.7	2.1
	Tps maximum	3	21	61	116
	Tps pire cas	2	20	59	3354

TAB. 5.10: *Temps de l'évaluation de l'exponentielle pour chaque bibliothèque.*

Il semble que ce chemin soit pris assez souvent puisque le temps d'exécution moyen est supérieur à 1.5 fois celui du temps minimum.

En ce qui concerne les résultats sur UltraSparc IIi, la libm prise pour référence est celle développée au sein des laboratoires Sun spécifiquement pour ce type de machine. C'est en partie une des raisons qui explique le facteur 2.66 de notre bibliothèque générique par rapport à celle de Sun. Une seconde raison est liée à l'architecture de l'UltraSPARC IIi et au caractère «hermétique» de ses pipelines entiers et flottants. Cette caractéristique joue en notre défaveur à chaque fois que nous manipulons les flottants comme des entiers (ajout d'un ulp, extraction

des exposants, ...).

La bibliothèque mathématique d'Apple affiche de piètres performances, puisqu'elle ne garantit pas l'arrondi correct, et qu'elle est 1.1 fois plus lente que la version que nous avons développé avec *crlibm*.

5.11.2.2 Le surcoût de l'arrondi correct

Parmi les bibliothèques testées seules MPFR, *libultim* et *crlibm* fournissent l'arrondi correct en arrondi au plus près. Les arrondis dirigés, très utiles en arithmétique par intervalles, sont proposés uniquement par les bibliothèques MPFR et *crlibm*.

La bibliothèque *libultim* produit l'arrondi correct de l'exponentielle en moyenne entre 0.9 (sur Pentium III) et 1.8 (sur Itanium) fois le coût de la bibliothèque standard libm (voir tableau 5.9). L'exponentielle que nous avons développée s'exécute en moyenne entre 0.91 (sur PowerPC) et 2.66 (sur UltraSparc IIi) fois le coût de la bibliothèque standard, ce qui reste raisonnable. MPFR fournit l'arrondi correct en moyenne entre 13.5 (sur PowerPC) et 153 (sur Itanium) par rapport à la libm. Ce surcoût lié à l'utilisation permanente de multiprécision n'est pas acceptable pour de nombreuses applications.

La principale différence au niveau des performances entre *libultim*, et notre bibliothèque *crlibm* se situe sur le temps d'évaluation dans le cas d'un pire cas. En effet, pour le pire cas en arrondi au plus près, nous fournissons l'arrondi correct pour un surcoût compris entre 10 (sur PowerPC) et 312 (sur UltraSparc IIi) fois le coût d'une évaluation standard (libm). Sur le même pire cas *libultim* à un surcoût entre 1677 (sur PowerPC) et 22357 (sur Itanium) fois le coût de la libm standard. Nous tirons donc pleinement avantage de la borne connue sur les pires cas pour améliorer les performances de la deuxième étape de l'évaluation.

5.11.2.3 Le prix des grosses tables

Nous avons essayé de tenir compte de la taille des tables de façon à consommer peu de ressources, et à disposer de performances acceptables quel que soit le nombre d'appels consécutifs à la fonction. La conséquence directe de cette attention est que nous sommes en moyenne moins pénalisés que la bibliothèque *libultim* et ses 13 Koctets de tables pour de petits nombres d'appels consécutifs à la même fonction. A l'inverse, cette taille de table plus importante lui est profitable lors de nombreux appels consécutifs. La bibliothèque *libultim* arrive même à être plus rapide qu'une évaluation standard par la *libm*, qui elle ne fournit pas l'arrondi correct.

Cependant lors de la conception de cette bibliothèque, un des objectifs était d'être performant quelles que soient les conditions d'appel. Aussi nous pensons que le contexte dans lequel nous avons fait les tests était favorable aux méthodes utilisant des grosses tables, et que dans un context plus proche de la réalité, les performances de *libultim* serait différentes.

5.11.2.4 Relations entre les deux étapes de *crlibm*

Nous avons également débranché l'appel à la deuxième partie de l'évaluation, l'évaluation multiprécision, et nous avons obtenu en arrondi au plus près, un arrondi faux sur 2097152 justes ($\approx 2^{21}$). Pour comparaison, la bibliothèque mathématique d'Apple donne 1 erreur d'arrondi sur 4739, celle de la glibc sur Pentium III, Xeon donne 1 erreur d'arrondi sur 587 cas, et celle de l'Itanium 1 erreur sur 491. La première étape de l'évaluation de l'exponentielle est donc très précise, voire même trop précise, puisque globalement le surcoût lié à la deuxième

étape sur le temps moyen est quasi nul. La deuxième étape est environ 30 fois plus lente que la première étape et n'est appelée que 1 fois sur 2^{15} (vérifiée d'après des arguments probabilistes, et corroborée par nos tests), le surcoût est de la deuxième étape est :

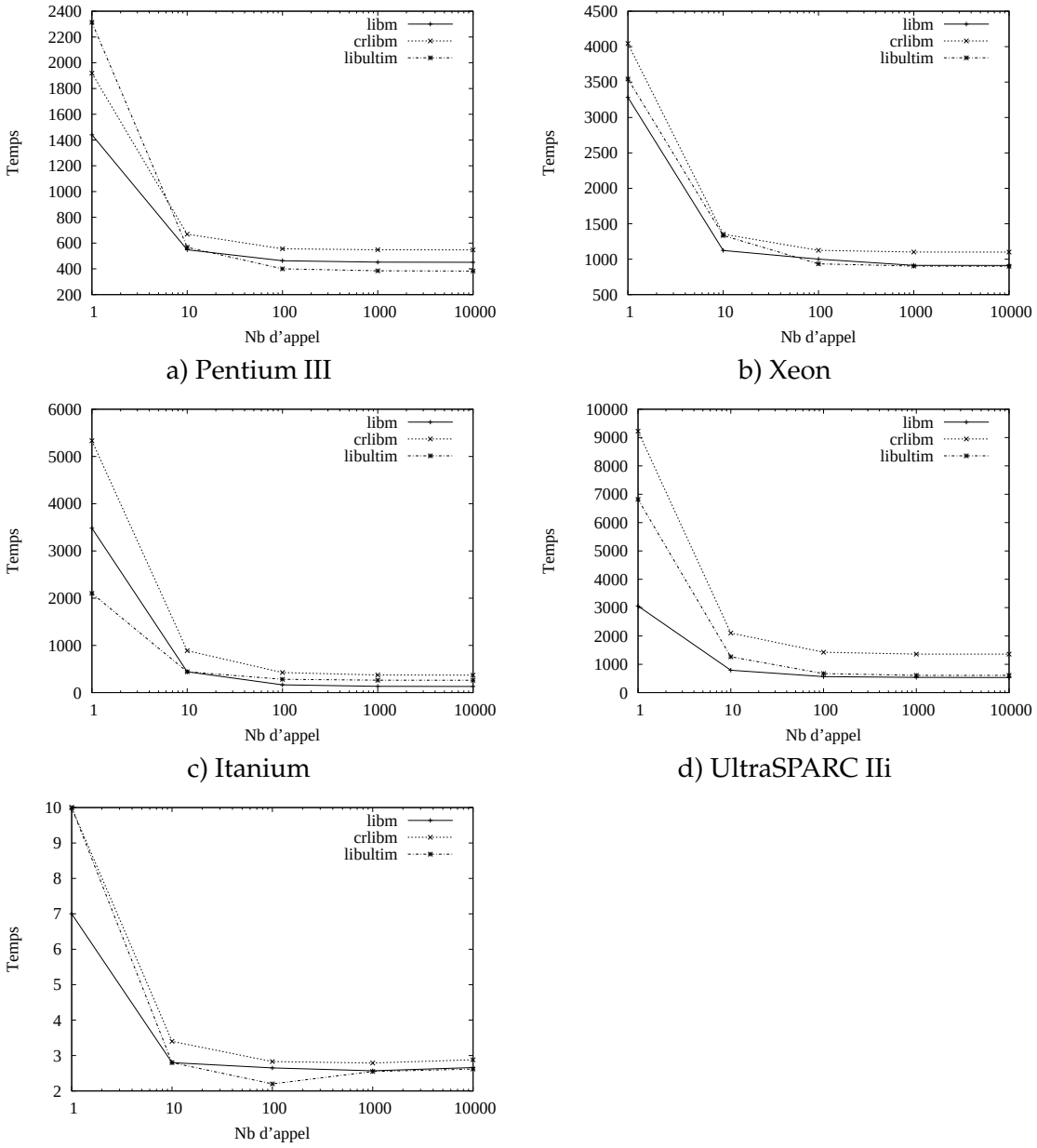
$$\frac{1 \times (2^{15} - 1) + 30 \times 1}{2^{15}} = 1.00088501$$

ce qui correspond a une augmentation du temps en moyenne de 0.09%.

5.12 Conclusion

Nous avons voulu ce chapitre didactique, pour permettre au lecteur d'appréhender le processus d'élaboration d'un programme d'évaluation d'une fonction élémentaire, et d'être à même de le reproduire pour les autres fonctions. A travers l'exemple de l'exponentielle, nous avons expliqué ce qu'impliquait l'arrondi correct sur les choix algorithmiques, le choix des paramètres comme la précision, la taille des tables, et enfin l'ordonnancement des instructions. Nous avons également ajouté la preuve de l'arrondi correct de l'exponentielle en double précision pour les quatre modes d'arrondi.

Nous avons montré qu'il était possible de remplir le contrat que nous nous étions fixé puisque le coût en moyenne de l'arrondi correct par la bibliothèque *crlibm* est entre 0.91 et 2.66 fois le prix d'une évaluation «normale». Cependant, notre première partie de l'évaluation semble être trop précise. Repenser la première partie de l'exponentielle pour la rendre moins précise, mais plus efficace constitue une sérieuse possibilité d'amélioration des performances.



TAB. 5.11: *Temps d'exécution moyen de la fonction exponentielle, en fonction du nombre d'appel consécutif sur des données tirées aléatoirements. Les temps sont données en cycle d'horloge pour l'Itanium, le Pentium III, le Xeon, et dans une unité arbitraire pour l'UltraSparc et le PPC.*

Conclusion

Cette thèse s'est articulée autour des méthodes, outils et preuves pour atteindre l'arrondi correct des fonctions élémentaires en double précision. Nous nous sommes basé sur la norme flottante IEEE-754 pour certifier les algorithmes et implémentations, et sur les caractéristiques des processeurs en 2003 pour optimiser l'évaluation de ces fonctions, et rendre le surcoût lié à l'arrondi correct raisonnable.

Contributions

Normalisation des fonctions élémentaires

Nous avons dans un premier temps mené une réflexion sur la norme IEEE-754, et les fonctions élémentaires. Ce qui nous a conduit à émettre une liste des caractéristiques et contraintes, en matière de précision, propriétés mathématiques, etc, que devraient respecter toutes implémentations de fonctions élémentaires.

Évaluation des fonctions élémentaires

Les contraintes et besoins sont différents selon que l'on considère une évaluation logicielle ou matérielle. Même si l'évaluation logicielle avec arrondi correct est le fil conducteur de ce mémoire, nous avons analysé la problématique des évaluations matérielles, et proposé une méthode pour les petites précisions. Cette méthode est très compétitive vis à vis des méthodes «multipartites» connues comme étant celles présentant les meilleurs compromis entre tailles de table et nombres d'opérations.

L'évaluation logicielle est composée de trois étapes : la réduction d'arguments, l'évaluation polynomiale et la reconstruction. Chacune d'elle représente une problématique différente. La réduction d'arguments, par exemple, si elle est additive, peut conduire à de catastrophiques annulations de précision. Pour faire face à ce problème, il existe deux classes d'algorithmes : ceux destinés à s'exécuter rapidement pour les petits arguments et les autres plus complexes pour les grands arguments. Pour les arguments de taille moyenne, nous avons proposé une réduction d'arguments en «grande base». Nous avons également indiqué la façon de construire des polynômes d'approximation avec de bonnes propriétés.

Optimisations appliquées aux fonctions élémentaires

Une fois l'algorithme défini, il faut l'implémenter efficacement. Cependant ce processus (création de l'algorithme/implémentation) n'est pas séquentiel, mais parallèle. En effet, les diverses optimisations liées à l'évaluation avec arrondi correct des fonctions élémentaires sont

aussi bien algorithmiques, qu'architecturales. Nous avons donc étudié et défini une liste d'optimisations utiles pour l'évaluation des fonctions élémentaires. Nous avons également étudié l'influence de la mémoire sur les performances globales de ces algorithmes et proposé une implémentation efficace des tests d'arrondi correct pour les quatre modes d'arrondi.

Bibliothèque multiprécision *scslib*

La façon qui nous a semblé la plus efficace pour parvenir à une implémentation de l'arrondi correct des fonction élémentaires, était de diviser le processus en deux phases : une évaluation rapide et précise la plupart du temps et une évaluation précise tout le temps. Cette deuxième phase repose sur des opérateurs multiprécision que nous voulions à la fois simples et performants.

Nous avons donc mis en place une bibliothèque de calcul en multiprécision *scslib*, basée sur une représentation des nombres inspirée des travaux de Brent [14]. Cette représentation permet d'effectuer les opérations multiprécision comme l'addition ou la multiplication et d'exprimer un parallélisme au niveau du chiffre. Cette bibliothèque est disponible sur Internet [4] et s'installe facilement sur une grande variété de système grâce aux outils gnu automake/autoconf. Nous avons également créé une liste de scripts destinés à appréhender sous Maple la représentation SCS. Ces scripts, ainsi que la bibliothèque *scslib*, ont constitué un des éléments de base dans le processus de réalisation de la bibliothèque *crlibm*.

Bibliothèque mathématique *crlibm*

Nous avons combiné tous ces éléments pour produire la bibliothèque d'évaluation des fonctions élémentaires avec arrondi correct que nous avons baptisée *crlibm*. Cette bibliothèque regroupe, à l'écriture de cette thèse, la première partie de l'exponentielle et les deuxièmes parties de l'exponentielle, du logarithme en base e, 2, 10, du sinus, du cosinus, de la tangente, de l'arctangente, et des fonctions hyperboliques. À travers l'exemple de l'exponentielle nous décrivons le processus de développement d'une évaluation efficace d'une fonction et de la construction de sa preuve de l'arrondi correct.

Perspectives

Poursuites des travaux sur la bibliothèque crlibm

La bibliothèque *crlibm* est encore loin d'être achevée. Il reste à développer les algorithmes de la première phase pour de nombreuses fonctions et à les implémenter. Les preuves associées, qui sont les garantes du bon comportement de ces implémentations, doivent être construites parallèlement au processus de développement.

Cependant construire de telles preuves manuellement est un processus lent et où il est difficile d'éviter les erreurs humaines. De plus à chaque modification de l'algorithme ou de son implémentation, le programme et la preuve doivent être reconstruites. Il serait intéressant d'étudier et de proposer un outil semblable à un parseur dans le domaine de la compilation, pour l'aide à la construction de preuves de ce type de programmes.

Parallélisme des algorithmes

Les algorithmes d'évaluation des fonctions élémentaires sont principalement composés d'additions et de multiplications flottantes. Toutefois l'architecture des processeurs évolue et de nouveaux opérateurs, bien que ne faisant toujours pas partie de la norme IEEE-754, se démocratisent. Ces nouveaux opérateurs tels que le FMA, l'approximation de l'inverse, ou les opérateurs multimedia peuvent conduire à d'importantes améliorations en terme de performance ou de précision. En outre, les processeurs sont capables d'exécuter de plus en plus d'instructions en parallèle, et une répartition équitable des tâches entre unités entières et unités flottantes dans ces algorithmes reste à étudier.

Polynômes

Nous avons décrit la façon d'obtenir de bonnes approximations polynomiales pour l'approximation des fonctions élémentaires. Cependant il existe d'autres critères, par exemple la présence de motifs particuliers dans les coefficients du polynôme lors de la deuxième phase de l'évaluation, qui sont susceptibles d'en améliorer les performances.

Les outils et méthodes pour le développement d'une bibliothèque d'évaluation de fonctions élémentaires avec arrondi correct pour la double précision sont pratiquement achevés. Nos travaux dans ce domaine permettent d'envisager la distribution prochaine d'une bibliothèque portable avec un surcoût lié à l'arrondi correct raisonnable. Elle pourra alors constituer l'impulsion qui incitera des constructeurs tels qu'Intel ou Sun Microsystem à investir dans le développement de bibliothèques mathématiques avec arrondi correct spécifiques et optimisées pour certaine architecture. L'objectif final, qui est de combiner qualité du résultat et efficacité des implémentations sera alors complètement rempli.

Bibliographie

[1] GMP, the GNU Multi-Precision library.
URL : *http ://swox.com/gmp/*

[2] IBM accurate portable mathematical library.
URL : *http ://oss.software.ibm.com/mathlib/*

[3] MPFR, the Multiprecision Precision Floating-Point Reliable library.
URL : *http ://www.loria.fr/equipes/polka/software.html*

[4] SCS, Software Carry-Save multiprecision mibrary.
URL : *http ://www.ens-lyon.fr/LIP/Arenaire/News/SCSLib/*

[5] Binding techniques. LIA-ISO/IEC 10967 : Language independent arithmetic. LIA-2 : Elementary numerical functions, 2002.
URL : *http ://std.dkuug.dk/jtc1/sc22/wg11*

[6] C : ISO/IEC 9899 revision of the standardization of the C language, known as C99, 2002.
URL : *http ://std.dkuug.dk/jtc1/sc22/wg14*

[7] Draft IEEE standard for binary floating-point arithmetic, 2002. (draft of the 17th of December, 2002).
URL : *http ://www.validlab.com/754R*

[8] M. ABRAMOWITZ ET I. A. STEGUN. *Handbook of mathematical functions with formulas, graphs and mathematical tables*. Applied Math. Series 55. National Bureau of Standards, Washington, D.C., 1964.

[9] AMD. *AMD Athlon processor x86 code optimization*, august 2001.

[10] K.E. ATKINSON. *An introduction to numerical analysis*. Wiley, 1989.

[11] D. H. BAILEY. A Fortran-90 based multiprecision system. *ACM Transactions on Mathematical Software*, 21(4) :379–387, 1995.

[12] J.C. BAJARD, O. BEAUMONT, J.M. CHESNEAUX, M. DAUMAS, J. ERHEL, D. MICHELUCCI, J.M. MULLER, B. PHILIPPE, N. REVOL, J.L. ROCH, ET J. VIGNES. *Qualité des calculs sur Ordinateur*. Masson, Paris, 1997. Coordinated by M. Daumas and J.M. Muller.

[13] S. BOLDO ET M. DAUMAS. A mechanically validated technique for extending the available precision. Dans *35th Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, California, 2001.
URL : *http ://www.ens-lyon.fr/ daumas/SoftArith/BolDau01b.pdf*

[14] R. P. BRENT. A FORTRAN multiple-precision arithmetic package. *ACM Transactions on Mathematical Software*, 4(1) :57–70, 1978.

[15] K. BRIGGS. The doubledouble library, 1998.
URL : *http ://members.lycos.co.uk/keithmbriggs/doubledouble.html*

[16] N. BRISEBARRE ET J.-M. MULLER. Finding the truncated polynomial that is closest to a function. Technical Report RR-2003-21, LIP, École Normale Supérieure de Lyon, april 2003.

[17] W. CODY ET W. WAITE. *Software Manual for the Elementary Functions*. Prentice-Hall, Englewood Cliffs, NJ, 1980.

[18] W. J. CODY. Implementation and testing of function software. Dans P. C. Messina et A. Murli, editors, *Problems and Methodologies in Mathematical Software Production*, Lecture Notes in Computer Science 142. Springer-Verlag, Berlin, 1982.

[19] W. J. CODY, J. T. COONEN, D. M. GAY, K. HANSON, D. HOUGH, W. KAHAN, R. KARPINSKI, J. PALMER, F. N. RIS, ET D. STEVENSON. A proposed radix-and-word-length-independent standard for floating-point arithmetic. *IEEE MICRO*, 4(4) :86–100, aug 1984.

[20] G. COLON-BONET, T. FISCHER, T. GRUTKOWSKI, S.D. NAFFZIGER, R. RIEDLINGER, ET T.J. SULLIVAN. The implementation of the Itanium 2 microprocessor. *IEEE Journal of Solid-State Circuits*, 37(11) :1448–1460, November 2002.

[21] M. CORNEA, J. HARRISON, ET P.T.P. TANG. *Scientific computing on Itanium-based systems*. Intel press, 2002.

[22] A. CUYT, P. KUTERNA, B. VERDONK, ET J. VERVLOET. Arithmos : a reliable integrated computational environment., 2001.
URL : *http ://win-www.uia.ac.be/u/cant/arithmos/index.html*

[23] D. DAS SARMA ET D.W. MATULA. Faithful bipartite ROM reciprocal tables. Dans S. Knowles et W.H. McAllister, editors, *12th IEEE Symposium on Computer Arithmetic*, pages 17–28, Bath, UK, 1995. IEEE Computer Society Press.

[24] M. DAUMAS. Expansions : lightweight multiple precision arithmetic. Dans *Architecture and Arithmetic Support for Multimedia*, page 14, Dagstuhl, Germany, 1998.
URL : *ftp ://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR98/RR98-39.ps.Z*

[25] M. DAUMAS ET D. W. MATULA. Validated roundings of dot products by sticky accumulation. *IEEE Transactions on Computers*, 46(5) :623–629, 1997.

[26] M. DAUMAS, C. MAZENC, X. MERRHEIM, ET J. M. MULLER. Modular range reduction : A new algorithm for fast and accurate computation of the elementary functions. *Journal of Universal Computer Science*, 1(3) :162–175, Mar 1995.

[27] F. DE DINECHIN ET A. TISSERAND. Some improvements on multipartite table methods. Dans Neil Burgess et Luigi Ciminiera, editors, *15th IEEE Symposium on Computer Arithmetic*, pages 128–135, Vail, Colorado, june 2001.

[28] D. DEFOUR. Cache-optimised methods for the evaluation of elementary functions. Technical Report RR2002-38, LIP, École Normale Supérieure de Lyon, October 2002. Available at ftp ://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR2002/RR2002-38.ps.gz.

[29] D. DEFOUR ET F. DE DINECHIN. Software carry-save for fast multiple-precision algorithms. Dans *35th International Congress of Mathematical Software.*, pages 29–40, Beijing , China, August 2002.

[30] D. DEFOUR ET F. DE DINECHIN. Software carry-save : A case study for instruction-level parallelism. Dans *7th conference on parallel computing technologies*, Nizhny-Novgorod, september 2003.

[31] D. DEFOUR, F. DE DINECHIN, ET J.M. MULLER. A new scheme for table-based evaluation of functions. Dans *36th Conference on signals, systems and computers conference*, pages 1608–1613, Asilomar, USA, November 2002.

[32] D. DEFOUR, G. HANROT, V. LEFÈVRE, J.-M. MULLER, N. REVOL, ET P. ZIMMERMANN. Proposal for a standardization of mathematical function implementation in floating-point arithmetic. *Numerical Algorithms*, 2003.

[33] D. DEFOUR, P. KORNERUP, J.M. MULLER, ET N. REVOL. A new range reduction. Dans *35th Asilomar conference on signals, systems and computers.*, pages 1656–1661, Asilomar, USA, November 2001.

[34] T. J. DEKKER. A floating point technique for extending the available precision. *Numerische Mathematik*, 18(3) :224–242, 1971.

[35] K. DOWD ET C. SEVERANCE. *High performance computing*. O'Reilly, 2 edition, 1998.

[36] B. P. FLANNERY, W. H. PRESS, S. A. TEUKOLSKY, ET W. T. VETTERLING. *Numerical recipes in C*. Cambridge University Press, 2 edition, 1992.

[37] S. GAL ET B. BACHELIS. An accurate elementary mathematical library for the IEEE floating point standard. *ACM Transactions on Mathematical Software*, 17(1) :26–45, mar 1991.

[38] D. GOLDBERG. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1) :5–47, Mar 1991.

[39] B. GREER, J. HARRISON, G. HENRY, W. LI, ET P. TANG. Scientific computing on the Itanium processor. Dans *SC 2001*, Denver, USA, November 2001.

[40] J. HARRISON, T. KUBASKA, S. STORY, ET P.T.P. TANG. The computation of transcendental functions on the IA-64 architecture. *Intel Technology Journal*, Q4, 1999.

[41] H. HASSLER ET N. TAKAGI. Function evaluation by table look-up and addition. Dans S. Knowles et W.H. McAllister, editors, *12th IEEE Symposium on Computer Arithmetic*, pages 10–16, Bath, UK, 1995. IEEE Computer Society Press.

[42] J. L. HENNESSY ET D. A. PATTERSON. *Computer Architecture : A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, 1990.

[43] Y. HIDA, X. LI, ET D. H. BAILEY. Algorithms for quad-double precision floating-point arithmetic. Dans Neil Burgess et Luigi Ciminiera, editors, *15th IEEE Symposium on Computer Arithmetic*, pages 155–162, Vail, Colorado, Jun 2001.

[44] N. J. HIGHAM. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, 1996.

[45] F. R. HOOTS ET R. L. ROEHRICH. Models for propagation of NORAD element sets. Technical Report 3, SPACETRACK REPORT, 1980.
URL : *http ://www.celestrak.com/NORAD/documentation/spacetrk.pdf*

[46] T. HOREL ET G. LAUTERBACH. UltraSPARC-III : Designing third-generation 64-bit performance. *IEEE Micro*, 19(3) :73–85, may 1999.

[47] IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Standard, Std 754-1985,* New York, 1985.

[48] IEEE standard for radix independent floating-point arithmetic. *ANSI/IEEE Standard, Std 854-1987, New York*, 1987.

[49] B. JACOB ET T. MUDGE. Virtual memory in contemporary microprocessors. *IEEE Micro*, pages 60–75, July 1998.

[50] B. JACOB ET T. MUDGE. Virtual memory : Issues of implementation. *Computer*, 31(6) :33–43, June 1998.

[51] W. KAHAN. Minimizing q*m-n. 1983.
URL : *http ://http.cs.berkeley.edu/∼wkahan/*

[52] W. KAHAN. Computer system support for scientific and engineering computation. 1988.
URL : *http ://www.validlab.com/fp-1988/lectures/*

[53] W. KAHAN. Lecture notes on the status of IEEE-754. 1996.
URL : *http ://http.cs.berkeley.edu/∼wkahan/ieee754status/ieee754.ps*

[54] W. KAHAN. What can you learn about floating-point arithmetic in one hour ? 1996.
URL : *http ://http.cs.berkeley.edu/∼wkahan/ieee754status*

[55] A. KARATSUBA ET Y. OFMAN. Multiplication of multidigit numbers on automata. *Doklady Akademii Nauk SSSR*, 145(2) :293–294, 1962.

[56] R. KARPINSKY. PARANOIA : A floating-point benchmark. *BYTE*, 10(2), 1985.

[57] D. KNUTH. *The Art of Computer Programming*, volume 2, "Seminumerical Algorithms". Addison Wesley, Reading, MA, third edition edition, 1998.

[58] I. KOREN. *Computer arithmetic algorithms*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[59] V. LEFÈVRE. *Moyens arithmétiques pour un calcul fiable*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, 2000.

[60] V. LEFÈVRE ET J. M. MULLER. Worst cases for correct rounding of the elementary functions in double precision. Dans *Proc. 15th IEEE Symposium on Computer Arithmetic*, Vail, USA, 2001. IEEE Computer Society Press.

[61] V. LEFÈVRE, J.M. MULLER, ET A. TISSERAND. Towards correctly rounded transcendentals. *IEEE Transactions on Computers*, 47(11) :1235–1243, nov 1998.

[62] R.-C. LI. Always Chebyshev interpolation in elementary function computations. Technical report, University of Kentucky, february 2002.
URL : *http ://www.cs.uky.edu/∼rcli/papers/bestvscheb.pdf*

[63] R.-C. LI, P. MARKSTEIN, J. P. OKADA, ET J. W. THOMAS. The libm library and floating-point arithmetic for HP-UX on Itanium. Technical report, Hewlett-Packard company, april 2001.

[64] A. A. LIDDICOAT ET M.J. FLYNN. High-performance floating point divide. Dans *Euromicro Symposium on Digital System Design*, pages 354–361, Warsaw, Poland, september 2001.

[65] P. MARKSTEIN. *IA-64 and Elementary Functions : Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000. ISBN : 0130183482.

[66] C. MOREAU-FINOT. *Preuves et algorithmes utilisant l'arithmétique normalisée IEEE*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, 2001.

[67] J.M. MULLER. *Elementary Functions, Algorithms and Implementation*. Birkhauser, Boston, 1997.

[68] J.M. MULLER. A few results on table-based methods. *Reliable Computing*, 5(3) :279–288, 1999.

[69] K. C. NG. Argument reduction for huge arguments : Good to the last bit (can be obtained by sending an e-mail to the author : kwok.ng@eng.sun.com). Technical report, SunPro, 1992.

[70] B. PARHAMI. *Computer Arithmetic : Algorithms and Hardware Designs*. Oxford University Press, New-York, 2000.

[71] Y. Patt, D. Grunwald, et K. Skadron, editors. *Proceedings of the 29th annual international symposium on computer architecture*. IEEE Computer Society, 2002.

[72] M. PAYNE ET R. HANEK. Radian reduction for trigonometric functions. *SIGNUM Newsletter*, 18 :19–24, 1983.

[73] J.A. PIÑEIRO, J.D. BRUGUERA, ET J.-M. MULLER. Faithful powering computation using table look-up and a fused accumulation tree. Dans Neil Burgess et Luigi Ciminiera, editors, *15th IEEE Symposium on Computer Arithmetic*, pages 40–47, Vail, Colorado, Jun 2001.

[74] E. REMEZ. Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation. *C.R. Académie des Sciences, Paris*, 198, 1934.

[75] A. SEZNEC ET F. LLOANSI. étude des architectures des microproceseurs MIPS R10000, Ultrasparc et Pentium Pro. Technical Report 1024, IRISA Rennes, France, may 1996.

[76] H. SHARANGPANI ET A. ARORA. Itanium processor microarchitecture. *IEEE Micro*, 20(5) :14–43, september 2000.

[77] D. STEHLÉ, V. LEFÈVRE, ET P. ZIMMERMANN. Worst cases and lattice reduction. Technical Report 4586, INRIA, october 2002.
URL : *http ://www.inria.fr/rrrt/rr-4586.html*

[78] P. H. STERBENZ. *Floating point computation*. Prentice-Hall, Englewood Cliffs, NJ, 1974.

[79] J.E. STINE ET M.J. SCHULTE. The symmetric table addition method for accurate function approximation. *Journal of VLSI Signal Processing*, 21(2) :167–177, 1999.

[80] P. T. P. TANG. Table-driven implementation of the logarithm function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 16(4) :378–400, dec 1990.

[81] P. T. P. TANG. Table lookup algorithms for elementary functions and their error analysis. Dans P. Kornerup et D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 232–236, Grenoble, France, June 1991. IEEE Computer Society Press, Los Alamitos, CA.

[82] W.F. WONG ET E. GOTO. Fast evaluation of the elementary functions in single precision. *IEEE Transactions on Computers*, 44(3) :453–457, mar 1995.

[83] A. ZIV. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, 17(3) :410–423, sep 1991.

Index

Résumé

Le codage et le comportement de l'arithmétique à virgule flottante disponible dans les ordinateurs sont spécifiés par la norme IEEE-754. Cette norme impose au système de rendre comme résultat de l'une des quatre opérations ($+$, $\times$, $/$, $\sqrt{}$), l'arrondi du résultat exact. Cette propriété que l'on appelle «arrondi correct», permet de garantir la qualité du résultat. Elle permet également la construction de preuves d'algorithmes, quelles que soient les machines sur lesquelles l'opération est executée. Toutefois cette norme présente des limites, puisque les fonctions élémentaires (sinus, cosinus, exponentielle...) en sont absentes. Cette abscence est liée au «dilemme du fabricant de tables» : il est, contrairement aux opérations de base, difficile de connaître la précision nécessaire pour garantir l'arrondi correct des fonctions élémentaires. Cependant, si l'on fixe le format de représentation, il est alors possible par une recherche exhaustive de déterminer cette borne ; ce fut le travail de thèse de Lefèvre pour la double précision.

L'objectif de ce mémoire est d'exploiter les bornes associées à chaque fonction, pour certifier l'arrondi correct des fonctions élémentaires en double précision pour les quatre modes d'arrondi. À cet effet, nous avons implémenté les évaluations en deux étapes : l'une rapide et juste la plupart du temps, basée sur les propriétés de l'arithmétique IEEE double précision, et l'autre juste tout le temps, composé d'opérateurs multiprécision. Pour cette deuxième phase, nous avons développé une bibliothèque d'opérateurs multiprécision optimisés pour les précisions données par ces bornes et les caractéristiques des processeurs en 2003.

Mots-clés : arithmétique des ordinateurs, norme IEEE-754, fonctions élémentaires, arrondi correct, réduction d'arguments, multiprécision.

Abstract

The representation formats and behaviors of floating point arithmetics available in computers are defined by the IEEE-754 standard. This standard imposes the system to return as a result of one of the four basic operations ($+$, $\times$, $/$, $\sqrt{}$), the rounding of the exact result. This property is called «correct rounding»,this warranties the quality of the result. It enables construction of proof that this particular algorithms can be manipulated independently of the machine. However, due to the «table makers dilemma», elementary functions (sine, cosine, exponential...) are absent in the IEEE-754 standard. Contrary to basic operations, it is difficult to discover the necessary accuracy required to guarantee correct rounding for elementary functions. However if the representation format is set, it is possible that an exhaustive search will help determine this bound : it was Lefevre's work for the double precision.

The objectives of this thesis is to exploit these bounds for each functions and rounding modes, to certify correct rounding in double precision. Thanks to this bound we have defined an evaluation within 2 steps : a quick phase which is based on the property of the IEEE standard that often proves satisfactory and an accurate step based on multiprecision operations which is precise all the time. For the second step we have designed a multiprecision library which was optimized in order to acquire precision corresponding to the bound, and the caracteristics of processors in 2003.

Keywords : computer arithmetic, IEEE-754 standard, elementary functions, correct rounding, range reduction, multiprecision.