Thèse de doctorat de l'Université Paris VI

Spécialité Informatique

Présentée par Roselyne Chotin-Avot

Pour obtenir le titre de

Docteur de L'Université Paris VI

ARCHITECTURES MATÉRIELLES POUR L'ARITHMÉTIQUE STOCHASTIQUE DISCRÈTE

Soutenue le 6 juin 2003, devant le jury composé de :

M. Alain Guyot	Rapporteur
M. Jean-Michel Muller	Rapporteur
M. Jean Vignes	Examinateur
M. Jean-Luc Lamotte	Examinateur
M. Eric Martin	Examinateur
M. Alain Greiner	Examinateur
M. Habib Mehrez	Directeur de thèse

Remerciements

Je souhaite avant tout exprimer toute ma reconnaissance au professeur Alain Greiner, directeur du département ASIM du LIP6, pour m'avoir accueillie dans son équipe, pour l'intérêt qu'il a porté à mes travaux, pour ces conseils et pour avoir accepter de participer au jury de cette thèse.

Je remercie également monsieur Jean-Michel Muller, directeur de recherche au CNRS à l'École Normale Supérieure de Lyon, et monsieur Alain Guyot, maître de conférence à l'Institut National Polytechnique de Grenoble, pour l'honneur qu'ils m'ont fait en acceptant d'être les rapporteurs de ce travail et pour le soin qu'ils ont apporté à l'examen de ce manuscrit.

Je tiens également à remercier monsieur Eric Martin, professeur au Laboratoire d'Électronique des Systèmes TEmps Réel de l'université de Bretagne Sud, pour avoir bien voulu être examinateur de ce travail.

Ce travail n'aurait pu aboutir sans une étroite collaboration avec les membres du département ANP du LIP6 et particulièrement messieurs Jean Vignes, Jean-Marie Chesneaux et Jean-Luc Lamotte. Ils ont toujours été disponibles pour répondre à mes questions et ont beaucoup contribué à ce travail, je leur en suis très reconnaissante. Et je remercie plus particulièrement le professeur Jean Vignes et Jean-Luc Lamotte, maître de conférence au LIP6 pour leur participation à ce jury.

Mes remerciements suivants seront pour mon directeur de thèse, monsieur Habib Mehrez, professeur au LIP6, qui a encadré mes recherches pendant ces quatre années.

Pendant cette thèse, j'ai encadré plusieurs stagiaires de maîtrise et de DEA : Téva Laou-

Hap, Antony Pinto, Ludovic Noury et Michaël Rocca. Je leur suis très reconnaissante pour la contribution qu'ils ont apportée à ces travaux.

Enfin merci à vous, tous les thésards et permanents avec lesquels j'ai partagé beaucoup de bon temps lors d'une sortie roller, à la piscine, sur un bateau, autours d'un verre, d'une assiette, d'une casserole... Plus que des collègues de travail, j'ai ici des amis.

Et mes dernières pensées seront pour toi Grégoire, pour ton soutien et ta patience, surtout dans la dernière ligne droite...

Résumé

L'utilisation de l'arithmétique à virgule flottante dans le calcul scientifique pose des problèmes de précision. En effet, les nombres réels n'étant pas tous représentables en virgule flottante, certains vont devoir être approchés. L'arithmétique stochastique discrète permet d'estimer et de contrôler les arrondis de calcul. L'utilisation logicielle de cette arithmétique est très coûteuse en temps de calcul. Le but de cette thèse est donc de proposer une architecture matérielle permettant de réduire ce coût.

Dans un premier temps nous avons réalisé en matériel les fonctionnalités, spécifiques à l'arithmétique stochastique discrète, que sont l'arrondi aléatoire, le calcul du nombre de chiffres significatifs, la détection des zéros informatiques et le contrôle des opérations de comparaison.

Cette arithmétique s'appuyant sur l'arithmétique à virgule flottante, il a fallu dans un deuxième temps développer une unité de calcul sur des nombres à virgule flottante. A cette unité a été ajouté le matériel nécessaire au contrôle et à l'estimation des arrondis de calcul. Ainsi une unité de calcul en virgule flottante effectuant les opérations d'addition, soustraction, multiplication, division, comparaison et conversions, avec estimation et contrôle des arrondis de calcul, a été réalisée jusqu'au dessin des masques physiques.

Enfin nous avons intégré cette unité au sein d'un système sur puce afin de pouvoir l'utiliser en exécutant des programmes réels et de pouvoir ainsi comparer les performances avec le logiciel.

Mots clés

Méthode CESTAC, arithmétique stochastique, arithmétique à virgule flottante, estimation de la précision, contrôle de la précision, système sur puce, accélérateur de calcul, adéquation algorithme architecture VLSI

Abstract

The use of floating point arithmetic in scientific computations is a source of problems of precision. Since all real numbers cannot represented in floating-point format, some will have to be approximated. The discrete stochastic arithmetic permits to control and estimate rounding errors. The software implementation of this arithmetic suffers from computation bottlenecks. The aim of this thesis is to propose a hardware architecture to reduce this cost.

First we implemented in hardware the specific functionalities of discrete stochastic arithmetic which are the random rounding mode, the computation of the number of significant bits, the detection of informatical zeroes and the control of the operations of comparison.

Second, since the discrete stochastic arithmetic is based on floating-point arithmetic, we developed a floating-point unit. The specific hardware, need for the control and the estimation of round-off errors propagation, was added. Thus a floating-point unit computing the operations of addition, subtraction, multiplication, division, comparison and conversions, with estimation and control of the round-off errors, was developed to the physical layout.

Finally we integrated this unit in a system on chip to be able to use it by computing programs and to compare the performances with the software.

Keywords

CESTAC method, stochastic arithmetic , floating-point arithmetic, accuracy estimation, accuracy control, system on chip, accelerator of calculation, adequacy algorithm VLSI architecture

Table des matières

Liste des tableaux

Table des figures

Chapitre

1

Introduction

Sommaire

Avec les avancées technologiques, la puissance de calcul des ordinateurs n'a cessé d'augmenter et aujourd'hui un ordinateur de bureau est capable d'effectuer plusieurs millions d'opérations à la seconde. Du coup, les utilisateurs de ces machines sont amenés à faire de plus en plus d'opérations et à résoudre des problèmes dont le traitement informatique était jusque là irréalisable.

La plupart des calculs se font avec l'arithmétique à virgule flottante qui est la représentation la plus utilisée de l'arithmétique sur les nombres réels. Or tous les nombres réels ne sont pas représentables en virgule flottante puisque la machine a une précision finie. Ils doivent donc être approchés par des nombres qui eux sont représentables. Cet arrondi contribue donc à introduire des erreurs dans les calculs, erreurs qui peuvent aboutir à des résultats totalement faux. Et bien sûr, plus il y a de calculs, plus le résultat final est entaché d'erreurs [8].

Qu'en penser lorsqu'aujourd'hui ces ordinateurs régissent une grande partie de notre vie ? Ils contrôlent les opérations effectuées dans une voiture, un avion, ils sont utilisés pour la modélisation des immeubles dans lesquels nous habitons. Bien sûr les calculs sont vérifiés, effectués sur des machines différentes, mais est-ce suffisant ?

Des méthodes existent pour pallier ce genre de problème en augmentant la précision des calculs ou en estimant l'erreur commise. Toutefois ces solutions ne font que repousser plus loin le problème sans pour autant le résoudre. De plus, ces méthodes sont souvent très coûteuses en temps de calcul comparé à l'utilisation directe de l'arithmétique à virgule flottante de la machine. Le plus souvent l'utilisateur ignore l'existence des problèmes dûs à la représentation des nombres et il fait donc confiance au résultat rendu alors que celui-ci peut être totalement faux.

L'impact que peut avoir un système qui calcule juste ou au moins qui contrôle et indique les erreurs dans les applications numériques est très clair, car alors l'utilisateur saurait dans tous les cas quel degré de confiance il peut accorder à ses résultats.

1.1 Objectifs et cadre de la thèse

L'objectif de cette thèse est donc de réaliser un système de calcul en virgule flottante plus robuste dans le sens où lorsqu'il fournit un résultat, soit il est correct (même là où actuellement l'utilisation de l'arithmétique à virgule flottante est mise en défaut), soit il est accompagné d'une estimation de sa fiabilité.

A l'heure actuelle de nombreux mathématiciens et informaticiens se sont penchés sur ces problèmes de comment réussir à estimer et contrôler la précision des calculs faits par la machine. Quelques solutions que nous passerons en revue voient le jour au niveau logiciel, mais il existe encore très peu de solutions matérielles.

Une des solutions pour estimer l'impact des erreurs d'arrondi dans un programme de calcul est d'utiliser l'arithmétique stochastique discrète. La mise en œuvre logicielle de cette arithmétique est coûteuse en temps de calcul et nécessite quelques modifications du code source afin d'utiliser les fonctionalités mises à disposition pour son utilisation. Le choix d'implanter cette arithmétique a été motivé par la demande du département Algorithmique Numérique et Parallélisme (ANP) du Laboratoire d'Informatique de Paris 6 (LIP6).

L'objectif de cette thèse est donc de concevoir une architecture matérielle mettant en œuvre l'arithmétique stochastique discrète. Dans un premier temps nous étudierons les problèmes liés à l'utilisation de l'arithmétique à virgule flottante et les solutions existantes pour les résoudre. Puis nous passerons en revue les fonctionnalités nécessaires à l'arithmétique stochastique discrète et comment leur implantation en matériel a été réalisée. L'arithmétique stochastique discrète étant basée sur l'arithmétique à virgule flottante, il faudra ensuite réaliser les opérateurs élémentaires de calcul au sein d'une unité normalisée à laquelle s'ajoutera le matériel nécessaire à la mise en œuvre de l'arithmétique stochastique discrète. Une fois cette unité réalisée, elle devra être intégrée dans un système plus complexe afin de permettre son utilisation directe par un programme de calcul.

Ces travaux s'appuient d'une part sur les compétences du département ANP du LIP6 dans le domaine du calcul à haute performance et de la validation numérique. D'autre part nous avons profité des recherches effectuées au sein du département Architecture des Systèmes

Intégrés et Microélectronique (ASIM) du LIP6 au niveau des outils de conception de circuits et systèmes intégrés.

1.2 Plan du développement

La problématique du contrôle de précision s'appuyant sur l'arithmétique à virgule flottante, nous présenterons, dans le chapitre 2, ce qu'est un nombre à virgule flottante, la norme IEEE-754 et ses limites. Dans ce chapitre, nous passerons également en revue les différentes solutions existantes pour le contrôle de la précision, et cela aussi bien au niveau logiciel que matériel.

Il a été choisi d'implanter l'arithmétique stochastique discrète. La théorie qui se cache derrière sera donnée dans le chapitre 3. Nous verrons également dans ce chapitre les différentes implantations de cette arithmétique et plus particulièrement le détail des choix architecturaux pour l'implantation matérielle.

La conception d'une unité de calcul en virgule flottante nécessitant de réaliser en matériel les différents opérateurs qui permettent de faire un calcul en virgule flottante, le chapitre 4 détaillera les choix architecturaux pour la réalisation de ces opérateurs au sein d'une unité de calcul en virgule flottante.

Le chapitre 5 sera consacré à la méthodologie et à l'environnement de conception utilisé pour la réalisation de cette unité. Dans ce chapitre nous présenterons également les résultats et performances des différentes architectures réalisées.

Dans le chapitre 6 nous étudierons l'intégration de notre unité de calcul dans un système sur puce et nous en détaillerons les performances.

Enfin les conclusions et perspectives seront données dans le chapitre 7.

Chapitre

2 Le flottant, ses limites, les solutions

Sommaire

La plupart des applications de calcul scientifiques utilisent les nombres à virgule flottante, qui sont la représentation en machine des nombres réels. Lorsque les nombres à virgule flottante sont apparus dans les processeurs, il n'y avait pas de standard. Chaque concepteur de processeur avait ses propres formats virgule flottante et sa propre manière de les utiliser [71][29][79]. La norme IEEE-754 [4] a été créée dans un soucis d'uniformiser tout cela et de donner la possibilité aux utilisateurs de développer des programmes dont les résultats sont les mêmes quelque soit la machine sur laquelle ils tournent.

Malheureusement, en arithmétique à virgule flottante, tous les réels ne sont pas représentables. Pour coder un réel non représentable, il faut donc l'arrondir vers un nombre à virgule flottante, ce qui introduit des erreurs d'arrondi. Des méthodes ont été développées pour réduire les problèmes dûs à ces erreurs et certaines d'entre elles ont été implantées en matériel.

2.1 La norme IEEE-754

Un réel peut s'écrire de la façon suivante :

$$x = (-1)^s.b^e.m \text{ avec } b \in \mathbb{N},\ \ s \in \{0,1\},\ \ e \in \mathbb{Z},\ \text{ et } m \in [0,b[$$

où b est base, s le signe, e l'exposant et m la mantisse.
La norme IEEE-754 spécifie comment représenter un nombre réel en machine où la base est le binaire. Elle définit :

1. Les différents formats de représentation
2. Les différents modes d'arrondi
3. Les opérations
4. Les nombres spéciaux
5. Les exceptions

2.1.1 Représentation des nombres à virgule flottante

La norme définit quatre formats de représentation des nombres à virgule flottante : simple précision, simple précision étendue, double précision et double précision étendue. Les représentations en format étendu permettent de coder les nombres à virgule flottante sur un nombre de bits plus important et ainsi d'augmenter la précision des calculs intermédiaires. Un nombre à virgule flottante sera représenté par son signe S, son exposant E et sa mantisse M (figure 2.1).

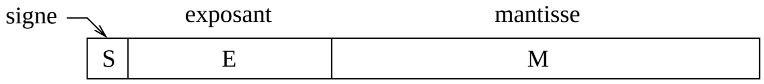


FIG. 2.1 – Représentation d'un nombre à virgule flottante

Le format définit la taille de la mantisse (p_M), de l'exposant (p_E), les valeurs minimale et maximale de l'exposant (E_{min} et E_{max}) et la valeur du biais. Le biais consiste à ajouter une quantité fixe, qui dépend du format, à la valeur de l'exposant et ainsi de travailler avec des exposants toujours positifs. Les valeurs de ces paramètres, en fonction du format, sont données par le tableau 2.1.

Format	Taille	p_M	p_E	E_{max}	E_{min}	Biais
Simple	32	24	8	+127	-126	+127
Simple étendu	≥ 43	≥ 32	≥ 11	$\geq +1023$	≤ -1022	/
Double	64	53	11	+1023	-1022	+1023
Double étendu	≥ 79	≥ 64	≥ 15	$\geq +16383$	≤ -16382	/

TAB. 2.1 – Représentation des nombres à virgule flottante

En base 2, un nombre à virgule flottante s'exprime donc de la manière suivante :

$$X = (-1)^S.2^{E+biais}.\sum_{i=0}^{p_M-1} m_i.2^{-i}$$

où

- S est le signe valant 0 quand $X > 0$ et 1 quand $X < 0$
- E est l'exposant entier compris entre E_{min} et E_{max}

– m_i sont les bits de la mantisse valant 0 ou 1

Avec ce type de représentation, il est possible de représenter un nombre à virgule flottante de plusieurs façons. Par exemple 1.5 peut s'écrire : $2^0.(2^0 + 2^{-1})$ ou $2^1.(2^{-1} + 2^{-2})$ ou encore $2^2.(2^{-2} + 2^{-3})$ etc.

Pour avoir une unique représentation, la notion de nombres *normalisés* et *dénormalisés* a été introduite. Uniquement les nombres *dénormalisés* peuvent avoir un 0 comme premier bit de leur mantisse et dans ce cas leur exposant vaut E_{min}. Un nombre *normalisé* aura donc forcément le bit de poids fort de sa mantisse à 1 (elle sera donc comprise dans l'intervalle $[1, 2[$) et son exposant sera supérieur ou égal à E_{min}. La valeur du premier bit de la mantisse étant forcément connue à partir du moment où la valeur de l'exposant est fixée, il ne sera pas représenté dans le codage machine du nombre (bit implicite). Ainsi 1.5 aura comme unique représentation normalisée $2^0.(2^0 + 2^{-1})$ et sera codé avec S=0, E=127 et M=10..00.

Une implantation conforme à la norme doit obligatoirement supporter au moins le format simple précision.

2.1.2 Les différents modes d'arrondis

Un nombre réel est considéré comme ayant une précision infinie (un nombre infini de bits). Par contre, sa représentation en machine est codée sur un nombre fini de bits. Il va donc falloir ne garder que certains bits pour pouvoir représenter ce nombre. Le mode d'arrondi spécifie comment choisir cette représentation. Un réel en précision infinie est donc encadré par deux nombres à virgule flottante représentables en machine : $F^- \leq X \leq F^+$. La norme définit quatre modes d'arrondi :

– Au plus près : le résultat est arrondi vers le nombre à virgule flottante le plus proche. Si les deux nombres sont à égale distance, le résultat est celui qui a son bit de mantisse le moins significatif à 0. C'est le mode d'arrondi par défaut de la norme
– vers $+\infty$: le résultat est arrondi vers le nombre à virgule flottante supérieur ou égal le plus proche
– vers $-\infty$: le résultat est arrondi vers le nombre à virgule flottante inférieur ou égal le plus proche
– vers zéro : le résultat est arrondi vers le nombre à virgule flottante immédiatement inférieur ou égal s'il est positif, ou immédiatement supérieur ou égal s'il est négatif

2.1.3 Les opérations

Remarque préléminaire : Dans ce paragraphe, nous allons être amenés à parler d'opération sur des entiers. Un entier est ici la représentation en machine des nombres entiers en complément à la base (deux).

Toute implantation de la norme doit fournir les opérations suivantes : addition, soustraction, multiplication, division, racine-carrée, reste de la division flottante, conversion entre les différents formats de nombres à virgule flottante, conversion entre nombre à virgule flottante et entier, conversion entre nombre à virgule flottante et décimal, et comparaison. La réalisation de ces opérations peut se faire aussi bien au niveau matériel que logiciel.

Opérations arithmétiques

L'addition, la soustraction, la multiplication, la division, la racine-carrée et le reste d'opérandes dans le même format doit fournir un résultat conforme à la norme, dans un format au moins aussi grand que celui des opérandes et pour tous les modes d'arrondi supporté par celle-ci.

Conversions

Il doit être possible de convertir un nombre à virgule flottante dans tous les formats de la norme. Une conversion dans un format plus petit que le format d'origine devra se faire dans tous les modes d'arrondi définis par la norme.
La conversion entre tous les formats de nombre à virgule flottante et d'entier doit exister. La conversion vers l'entier doit se faire pour tous les modes d'arrondi supportés par la norme.
La norme demande aussi d'implanter les conversions entre le format binaire (base 2) et le format décimal (base 10).

Comparaison

Il doit être possible de comparer n'importe quels nombres à virgule flottante, même si leurs formats respectifs diffèrent. La comparaison est une opération exacte, il n'y a pas besoin d'arrondir le résultat. Les résultats possibles sont : plus petit, égal, plus grand ou indéfini.

2.1.4 Les nombres spéciaux

Les différents nombres spéciaux définis par la norme sont :
- $+\infty$: S=0, E=1..1, M=0..0
- -∞ : S=1, E=1..1, M=0..0
- NaNq[1] : S=X, E=1..1, M=1X..X
- NaNs[2] : S=X, E=1..1, M=0X..X ($M \neq 0$)
- ± 0 : S=0/1, E=0..0, M=0..0

Remarque : il y a deux représentations pour le zéro (-0 et +0). Le signe est significatif dans certains cas comme par exemple pour la division par 0 dont le résultat est $+\infty$ (resp. $-\infty$) lorsque le diviseur est +0 (resp. -0).

Les opérations arithmétiques avec des nombres infinis sont autorisées et ne produisent pas d'exceptions, sauf dans le cas d'un dépassement de capacité (le nombre résultant d'un calcul est supérieur au plus grand nombre à virgule flottante représentable en machine), d'une division par zéro ou d'un opérande invalide.
Les opérations ayant pour opérande des NaN produisent des résultats qui sont soit des NaN, soit des infinis.
La norme ne spécifie pas comment doivent être traité le signe et la mantisse des NaN. Ceci est laissé à la discrétion du concepteur.

La figure 2.2 représente l'espace des nombres à virgule flottante et où se situent les différents types.

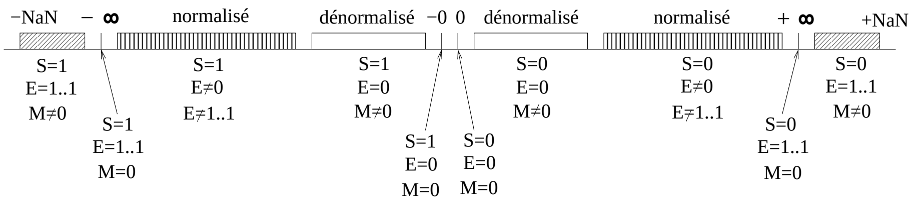


FIG. 2.2 – Les différents types de nombres à virgule flottante

[1] Quiet Not a Number : indique que l'une des opérandes est un NaNs ou un NaNq
[2] Signaling Not a Number : signale le résultat d'une opération impossible

2.1.5 Les exceptions

Il y a cinq types d'exceptions dans la norme qui doivent être signalées par un drapeau lorsqu'elles sont détectées.

Opération invalide

Cette exception est détectée si un opérande est invalide pour l'opération exécutée. Les opérations invalides sont :

1. Une opération sur un NaNs
2. L'addition de deux infinis de signes différents ou la soustraction de deux infinis de mêmes signes
3. La multiplication de zéro par un infini
4. La division de zéro par zéro ou d'un infini par un infini
5. Le reste de la division d'un infini par 0
6. La racine-carrée d'un nombre négatif
7. Les conversions d'infinis ou de NaN

Division par zéro

Cette exception est détectée si le diviseur est zéro et le dividende un nombre fini non nul.

Dépassement de capacité

L'*overflow* est détecté lors d'un dépassement de capacité. C'est-à-dire lorsque le résultat de l'opération effectuée est supérieur au plus grand des nombres à virgule flottante représentables dans le format. Le résultat de l'opération qui a provoqué l'*overflow* dépend de l'arrondi :
- Arrondi au plus près : le résultat est infini
- Arrondi vers $+\infty$: si le résultat est positif c'est $+\infty$ sinon c'est le plus petit nombre à virgule flottante négatif représentable (S=1, E=0, M=1)
- Arrondi vers $-\infty$: si le résultat est négatif c'est $-\infty$ sinon c'est le plus grand nombre à virgule flottante positif représentable (S=0, E=1..10, M=1..1)
- Arrondi vers 0 : le résultat est le plus grand nombre à virgule flottante représentable avec le signe du résultat

Sous-dépassement de capacité

L'*underflow* est détecté lors d'un sous-dépassement de capacité. C'est-à-dire lorsque le résultat de l'opération effectuée est inférieur au plus petit des nombres à virgule flottante représentables dans le format. Deux événements peuvent conduire à un *underflow* :
– La création d'un petit nombre compris entre $\pm 2^{E_{min}}$
– Une grande perte de précision sur des nombres dénormalisés

Résultat inexact

Signale que le résultat d'une opération n'est pas exact lorsqu'il a été arrondi.

2.1.6 La gestion des exceptions

Un gestionnaire d'exception peut être utilisé pour traiter les exceptions précédentes. Dans ce cas, il doit être capable de déterminer :
– Quelle exception a été détectée
– Quelle opération était en train de s'exécuter
– Le format de destination
– En cas d'*overflow*, d'*underflow* et de résultat inexact, le résultat correctement arrondi et non le résultat de l'opération[3].
– En cas d'opération invalide et de division par zéro, la valeur des opérandes de l'opération fautive

2.2 Les limites de la norme

Tous les nombres réels ne sont pas représentables dans l'ensemble des nombres à virgule flottante. Pour coder en machine un réel non représentable, il va donc falloir en faire l'approximation. Tout réel X non représentable dans l'ensemble des nombres à virgule flottante est encadré par deux nombres à virgule flottante : $F^- \leq X \leq F^+$. En fonction du mode d'arrondi de la machine, le réel X est représenté soit par F^-, soit par F^+.

[3]Par exemple lors d'un *overflow*, le résultat rendu est le plus grand nombre à virgule flottante représentable

De plus, l'ensemble des nombres à virgule flottante est un ensemble borné ce qui entraîne des phénomènes de dépassement (resp. sous-dépassement) de capacité lorsqu'une opération fournit un résultat supérieur (resp. inférieur) au plus grand (resp. petit) nombre à virgule flottante représentable dans le format et dans ce cas le résultat est arrondi au plus grand (resp. petit) nombre à virgule flottante représentable.

Le résultat d'une opération virgule flottante est en général entaché d'une erreur. Il va sans dire que plus le nombre d'opérations augmente, plus ces erreurs deviennent importantes. L'utilisateur lui n'est pas informé que le résultat rendu par la machine est erroné, voire complètement faux. C'est pourquoi il est nécessaire d'essayer de contrôler le nombre de chiffres significatifs des calculs.
Nous allons voir au travers d'exemples mathématiques, mais aussi réels[4], les conséquences des erreurs d'arrondi dans les calculs.

2.2.1 1$^{\text{er}}$ exemple

Jean-Michel Muller a proposé dans [56] un exemple de suite qui en arithmétique à virgule flottante converge vers une valeur qui n'est pas celle calculée par l'arithmétique exacte. Cet exemple a été repris par l'équipe de Jean Vignes dans [21]. Soit :

$$x_{n+1} = 111 - \frac{1130}{x_n} + \frac{3000}{x_n.x_{n-1}}$$

La trajectoire dans le plan (x, y) de ce système tend, suivant le point de départ $P_1 = (x_1, y_1)$, vers trois points fixes limites :
- si $P_1 \in (h)$, hyperbole d'équation $y = 11 - \frac{30}{x}$, le point limite est $(6, 6)$
- si $P_1 \notin (h)$ et $P_1 \neq (5, 5)$, le point limite est $(100, 100)$
- si $P_1 = (5, 5)$, le point limite est $(5, 5)$

Lorsque le calcul de la trajectoire, ayant pour point de départ $(2, -4)$ (appartenant à l'hyperbole), est effectué en arithmétique à virgule flottante, quelque soit la précision, le point limite calculé est obtenu rapidement et vaut $(100, 100)$, alors qu'en réalité ce devrait être $(6, 6)$.
En fait, au cours des itérations, la trajectoire quitte la parabole et converge vers le point $(100, 100)$.

[4]http ://www.ima.umn.edu/~arnold/disasters

Le résultat rendu par la machine est donc le point $(100, 100)$ qui n'est pas le résultat exact et l'utilisateur ne sait pas que son résultat est faux.

2.2.2 2^e exemple

Un autre exemple tiré de [21] est de calculer, de façon itérative, la somme $S = \sum_{i=1}^{n} \frac{i}{3}$.

Pour différentes valeurs de n en arithmétique à virgule flottante simple précision, nous obtenons les résultats suivants :

$n = 10^4$	$S = 1.6668333E + 07$	8 chiffres exacts
$n = 10^5$	$S = 1.6666652E + 09$	5 chiffres exacts
$n = 10^6$	$S = 1.6670956E + 11$	3 chiffres exacts
$n = 10^7$	$S = 1.6745195E + 13$	2 chiffres exacts

Plus la valeur de n augmente plus les résultats se dégradent, sans qu'une fois de plus l'utilisateur en soit averti.

2.2.3 3^e exemple

Considérons maintenant la résolution de l'équation du second degré [21] :

$$0.3x^2 + 2.1x + 3.675 = 0$$

Cette équation admet comme solution la racine double $x = -3.5$
Or sur machine en arithmétique à virgule flottante simple précision, la résolution donne des solutions totalement différentes suivant le mode d'arrondi utilisé. Ceci est dû au calcul du discriminant D :

- **Arrondi au plus près** : $D = -3.81470E - 06 < 0$, il y a donc 2 racines complexes conjuguées $Z1 = -3.5 + i * 0.9765625E - 03$ et $Z2 = -3.5 - i * 0.9765625E - 03$
- **Arrondi vers zéro** : $D = 0$, il y a donc une racine double $X = -3.5$
- **Arrondi vers** $+\infty$: $D = 3.81470E - 06 > 0$, il y a donc 2 racines réelles $X1 = -3.500977$ et $X2 = -3.499024$
- **Arrondi vers** $-\infty$: $D = 0$, il y a donc une racine double $X = -3.5$

L'arrondi par défaut d'une machine est en général l'arrondi au plus près, donc le résultat retourné n'est pas celui attendu.

2.2.4 4^{e} exemple : Echec d'un anti-missile Patriot

Le 25 février 1991, pendant la Guerre du Golfe, une batterie d'anti-missiles Patriot américaine, positionnée à Dharam en Arabie Saoudite, échoue dans l'interception d'un missile Scud irakien, qui s'est ensuite écrasé sur une caserne de l'armée américaine causant la mort de 28 soldats et faisant une centaine de blessés.

Un rapport du *General Accounting Office* [32] (organisme qui vérifie l'utilisation des fonds publics américains) explique que l'erreur était due à un calcul imprécis du temps depuis le démarrage du système. L'horloge interne du système calculant le temps en dixièmes de secondes pour obtenir le temps en secondes le logiciel doit multiplier ce temps par $\frac{1}{10}$. $\frac{1}{10}$ n'étant pas exactement représentable en machine, il a donc fallu l'arrondir ce qui a entraîné une petite erreur d'arrondi. $\frac{1}{10}$ en binaire se code 0.00011001100110011.. ce qui représente une erreur de 0.000000095 lorsque $\frac{1}{10}$ est arrondi au 24^{e} bit (taille des registres internes du système). La batterie d'anti-missiles Patriot était en marche depuis une centaine d'heures, ce qui représente 3.6 millions de dixièmes de secondes qui multipliés par l'erreur d'arrondi donnent une erreur de 0.34 s.

Un missile Scud avançant à la vitesse de 1676 m/s, en 0.34 s il parcourt 687 m ce qui est largement suffisant pour sortir du rayon d'action de l'anti-missile Patriot censé se lancer à sa poursuite. En effet le radar de l'anti-missile Patriot a détecté un missile Scud dans son champ d'action. Connaissant la vitesse du Scud, il peut prédire sa position lors du prochain relevé. Ici la position prédite présentait une erreur de 687 m, il n'y avait donc plus de missile Scud dans le champ d'action du radar lors du relevé et le déclenchement de l'anti-missile ne s'est donc pas fait.

Nous sommes ici en présence d'un problème typique lié à la représentation des nombres en machine. Un système fiable aurait dû s'apercevoir qu'au cours du temps les résultats des calculs étaient de moins en moins significatifs et ainsi en tenir compte dans son exécution.

2.3 Les solutions possibles

Le calcul scientifique est basé sur l'exécution d'un algorithme pour résoudre un problème. Cet algorithme s'exécute en général sur un ordinateur et est donc soumis à plusieurs formes d'erreurs. Différentes approches existent pour estimer et améliorer la qualité des résultats. Ces approches sont implantées de façon logicielle et certaines d'entre elles ont vu le jour au niveau matériel.

2.3.1 Les différentes formes d'erreurs

Erreurs dues à la représentation en machine

Une des premières sources d'erreur dans l'exécution d'un algorithme sur une machine est liée à la représentation des réels. En effet l'information étant codée sur un nombre fini de bits, l'ensemble des nombres à virgule flottante est borné et discret, ce qui a pour conséquences :

- tout réel supérieur (resp. inférieur) au plus grand (resp. petit) des nombres à virgule flottante représentables, ne pourra pas être représenté en machine, ce qui implique des dépassements de capacité
- un nombre réel non représentable en machine va être remplacé de façon approximative par un nombre à virgule flottante ce qui implique des erreurs d'arrondi

Erreurs dues à l'imprécision des données

Une autre forme d'erreur provient de l'imprécision sur les données. En effet, un algorithme numérique peut prendre en entrée des données provenant d'un autre calcul numérique, ou de mesures effectuées par des capteurs, mesures qui ne sont pas elles-mêmes complètement fiables, puisque entachées d'inexactitudes liées à la précision de ces capteurs. Ici le résultat fourni par l'ordinateur est non seulement entaché des erreurs dues aux imprécisions des données, mais aussi de la propagation des erreurs d'arrondi.

Erreurs dues à l'algorithme numérique

Lorsque le résultat d'un calcul scientifique est obtenu par un algorithme, plusieurs problèmes peuvent se poser :

- fiabilité des branchements conditionnels : au fur et à mesure des calculs, les résultats ont une partie significative, dont la valeur est certaine, et une partie non significative, due à la propagation des erreurs d'arrondi qui évoluent. Lors des branchments conditionnels où deux nombres à virgule flottante sont comparés, il faut pouvoir s'assurer que la comparaison est faite avec les parties significatives des opérandes, puisque la partie non

significative est entachée d'erreurs et la prendre en compte rendrait la comparaison non fiable
- critère d'arrêt : il dépend d'une quantité ε fixée par l'utilisateur, mais rien ne permet d'affirmer que cette quantité est bien choisie. Elle peut être trop grande et dans ce cas il manque des itérations pour obtenir le résultat correct, ou trop petite et des calculs ont été effectués inutilement
- pas de discrétisation optimal : il minimise l'erreur globale obtenue avec une méthode approchée. Pour l'obtenir, il faut connaître l'erreur due aux erreurs d'arrondi, ce qui est difficile en utilisant l'arithmétique à virgule flottante

2.3.2 Les solutions logicielles

Il existe dans la littérature de nombreuses méthodes logicielles pour pallier les problèmes d'arrondi et d'instabilité numérique. Nous présenterons ici brièvement certaines d'entre elles. Le lecteur désireux d'en savoir davantage pourra se référer à [28] pour plus de détails.

L'arithmétique d'intervalle

Elle a été introduite en 1962 par Moore [55] et consiste à représenter le résultat par un intervalle $[F^-, F^+]$, F^- représentant le résultat arrondi vers $-\infty$ et F^+ celui arrondi vers $+\infty$. Il est certain que le résultat exact se trouve dans l'intervalle $[F^-, F^+]$. Chaque opération virgule flottante devient alors des opérations sur chacune des bornes de l'intervalle.

A l'heure actuelle, de nombreux outils implantants l'arithmétique d'intervalle existent, parmi lesquels nous pouvons citer :
- les langages-XCS développés à l'université de Karlsruhe par l'équipe d'Ulrich Kulisch et disponibles en Pascal [49], C et C++ [48]
- PROFIL/BIAS [51] qui est une bibliothèque C++ dont une version simplifiée a été implantée dans Matlab 5
- et bien sûr tous les outils mathématiques que sont Maple [16], Mathematica [91] et Matlab [52]

L'arithmétique stochastique discrète

Cette arithmétique a été développée par l'équipe de J. Vignes [23][83]. Son but est d'estimer l'erreur d'arrondi engendrée par un programme de calcul. Pour cela un même programme est exécuté plusieurs fois, c'est-à-dire que chaque opération élémentaire est effectuée plusieurs fois avec un mode d'arrondi différent. A la fin un résultat est donc constitué de plusieurs valeurs différentes, chacune représentant une propagation de l'erreur d'arrondi. Le résultat final du calcul sera la partie commune à ces différentes valeurs.

Le logiciel mettant en œuvre l'arithmétique stochastique discrète, nommé CADNA [18][20] se présente sous la forme d'une bibliothèque disponible en Fortran, ADA, C et C++.

L'implantation matérielle de cette arithmétique étant l'objet de cette thèse, ce point sera développé dans le chapitre 3.

Arithmétique multi-précision

Cette arithmétique a été introduite en 1960 dans [65]. Ici les nombres sont représentés sur une taille plus grande (100, 1000, 10000 chiffres) qui est fixée comme étant plus grande que l'instabilité numérique du programme n'en fera perdre. La taille des nombres utilisés étant plus grande que celle de la machine, les différentes opérations sont faites à l'aide d'algorithmes. L'addition et la soustraction, se font chiffre à chiffre comme appris à l'école. Pour la multiplication, les algorithmes décomposent les nombres en plusieurs parties et effectuent ainsi les opérations sur ces parties plus petites, ce qui est moins coûteux. Pour les autres opérations, plusieurs méthodes sont utilisées comme le développement en série entière sur lequel divers algorithmes peuvent être appliqués pour le calculer rapidement [9][10], ou encore l'algorithme CORDIC (COordinate Rotations DIgital Computer) [88][89].

Cette arithmétique peut être étendue à une arithmétique à précision variable où la taille des données est augmentée en fonction des besoins. Elle utilise l'arithmétique redondante [5] qui permet de calculer d'abord l'information dans les poids forts et ainsi d'avoir en premier l'information pertinente. Cela amène à définir des opérateurs en ligne. Cette arithmétique est souvent combinée avec l'arithmétique d'intervalle et dans ce cas le résultat n'est pas un unique nombre de grande précision, mais un intervalle borné par deux nombres de grande précision.

Là encore de nombreuses implantations logicielles de l'arithmétique multi-précision existent :
- La bibliothèque développée par le groupe GNU [34]
- La bibliothèque MP de Brent écrite en Fortran [11]
- La bibliothèque MPFUN développée pour la NASA par Bailey [6]
- La bibliothèque VPI de Ely qui implante une arithmétique d'intervalle à précision variable [31]

Arithmétique exacte

Le calcul formel s'utilise pour avoir des résultats symboliques ou exacts, mais le problème est que la taille du codage des nombres augmente très vite et donc le temps de calcul devient vite très élevé. Par exemple d'après une étude faite dans [14], les programmes écrits en Maple [16] sont 1000 fois plus lents que leurs équivalents C, et 3000 fois pour Mathematica [91].

2.3.3 Les solutions matérielles

Le principal désavantage de telles méthodes logicielles est le temps nécessaire à l'exécution d'un programme de calcul. Ces méthodes prennent entre 10 à 3000 fois plus de temps que leur équivalent en arithmétique à virgule flottante [14][69][74]. Une implantation directe en matériel permettrait d'accélérer de manière significative ce temps de calcul. Plusieurs études ont déjà été menées dans ce sens.

Produit scalaire exact

Dans le calcul scientifique, beaucoup de calculs se font sur des vecteurs et des matrices. Le calcul du produit scalaire est donc énormément utilisé. Des architectures matérielles pour le calcul exact du produit scalaire ont été développées [38][47][50]. Toutes les opérations sont faites en arithmétique exacte et l'arrondi n'est effectué qu'à la fin. L'architecture de ces opérateurs repose sur un multiplieur/accumulateur performant et de grande taille pour ne pas être obligé d'arrondir les calculs. Ce type d'opérateur permet d'améliorer la précision et d'accélérer les calculs, mais cela ne résoud qu'une partie des problèmes.

Arithmétique d'intervalle

L'architecture matérielle nécessaire pour l'implantation de l'arithmétique d'intervalle est présentée dans [90]. La réalisation de l'addition/soustraction se fait en utilisant deux additionneurs :

$$[a,b]+[c,d]=[(a+c)^-,(b+d)^+] \quad \text{et} \quad [a,b]-[c,d]=[(a-d)^-,(b-c)^+]$$

où X^- est le résultat arrondi vers $-\infty$ et X^+ celui vers $+\infty$.

Pour la division, comme nous pouvons le voir dans le tableau 2.2, seulement deux diviseurs sont nécessaires.

$[a,b] \geq [0,0]$ et $[c,d] > [0,0]$	$[a,b]/[c,d] = [(a/d)^-,(b/c)^+]$
$[a,b] \geq [0,0]$ et $[c,d] < [0,0]$	$[a,b]/[c,d] = [(b/d)^-,(a/c)^+]$
$[a,b] \geq [0,0]$ et $0 \in [c,d]$	$[a,b]/[c,d]$ indéfini
$[a,b] \leq [0,0]$ et $[c,d] > [0,0]$	$[a,b]/[c,d] = [(a/c)^-,(b/d)^+]$
$[a,b] \leq [0,0]$ et $[c,d] < [0,0]$	$[a,b]/[c,d] = [(b/c)^-,(a/d)^+]$
$[a,b] \leq [0,0]$ et $0 \in [c,d]$	$[a,b]/[c,d]$ indéfini
$0 \in [a,b]$ et $[c,d] > [0,0]$	$[a,b]/[c,d] = [(a/c)^-,(b/c)^+]$
$0 \in [a,b]$ et $[c,d] < [0,0]$	$[a,b]/[c,d] = [(b/d)^-,(a/d)^+]$
$0 \in [a,b]$ et $0 \in [c,d]$	$[a,b]/[c,d]$ indéfini

Tab. 2.2 – Division en arithmétique d'intervalle

Par contre, la multiplication est l'opération qui coûte le plus cher. En effet, réaliser une multiplication de deux opérandes représentés par des intervalles revient à effectuer :

$$[a,b]\times[c,d]=[\min((a\times c)^-,(a\times d)^-,(b\times c)^-,(b\times d)^-),\max((a\times c)^+,(a\times d)^+,(b\times c)^+,(b\times d)^+)]$$

Ce calcul peut se réduire à l'utilisation de trois multiplieurs et un comparateur [39]. Un autre multiplieur basé sur le même principe et s'appuyant sur les propriétés de la multiplication d'intervalle développées dans [55], n'utilise que deux multiplieurs et une unité retournant le minimum et le maximum [72].

Arithmétique multi-précision

De nombreux processeurs mettant en œuvre l'arithmétique multi-précision ont vu le jour. La plupart d'entre eux ne fonctionnent qu'avec des entiers. Cascade [15] est basé sur une architecture dédiée pour une gestion rapide en mémoire des entiers en multi-précision. Draft [24] a huit unités arithmétiques et logiques 32 bits pour permettre des calculs pouvant atteindre une précision de 256 bits. Seul Cadac [25] permet d'effectuer des calculs multi-précision avec des nombres à virgule flottante.

Arithmétique mixte

Certains coprocesseurs utilisent à la fois l'arithmétique d'intervalle et l'arithmétique multi-précision [73]. L'utilisateur peut fixer la précision souhaitée. Le coprocesseur détermine la précision du calcul et si elle est mauvaise, il réitère le calcul avec une précision plus importante. Ce coprocesseur peut effectuer des calculs allant jusqu'à 1024 bits de précision. Les intervalles multi-précision sont représentés par deux nombres en multi-précision. Là aussi l'architecture est basée sur un accumulateur de grande taille.

Arithmétique exacte

Quelques architectures dédiées pour le calcul en arithmétique exacte ont vu le jour [26]. Ce coprocesseur superscalaire pipeliné est optimisé pour répondre aux spécificités de l'arithmétique exacte en particulier le calcul sur des nombres de grande taille. Des architectures matérielles d'opérateurs pour l'arithmétique rationnelle ont également été réalisées [37].

2.3.4 Problématique

Le but de cette thèse est de développer un système de calcul en virgule flottante permettant de contrôler et d'estimer les erreurs d'arrondis. Nous avons vu précédemment qu'il existe de nombreuses méthodes logicielles pour contrôler la précision des calculs. D'après [84][64], une méthode efficace est celle développée par le laboratoire LIP6/ANP : l'arithmétique stochastique discrète [83]. C'est cette méthode que nous avons choisie d'implanter en matériel. Une version logicielle existe sous la forme de la bibliothèque CADNA[5] (Control of Accuracy and Debugging for Numerical Applications) [20][18]. Cette bibliothèque si elle est fort utile pour l'analyse des erreurs d'arrondi, entraîne un surcoût impor-

[5] http ://www-anp.lip6.fr/~cadna/

tant au temps d'exécution d'un programme de calcul. Pour des algorithmes numériques dont la durée d'exécution peut être de plusieurs jours, ce surplus est insupportable.

A l'heure actuelle, la bibliothèque CADNA est beaucoup utilisée pour faire des simulations, c'est-à-dire que l'utilisateur écrit son algorithme numérique et le fait fonctionner avec la bibliothèque CADNA. Il détecte ainsi les instabilités numériques et les problèmes liés à la propagation des erreurs d'arrondi. Ensuite il réordonne son programme de manière à les faire éventuellement disparaître. Une fois le programme mis au point, il le fait fonctionner avec l'arithmétique à virgule flottante normale, car l'utilisation de CADNA est trop pénalisante en terme de performances pour une application embarquée. Ceci pose différents problèmes. En effet, la mise au point du programme, afin de faire disparaître les problèmes liés aux erreurs d'arrondi, nécessite un gros investissement en temps (évalué à plusieurs mois). De plus rien ne garantit que malgré toutes les simulations effectuées un cas non prévu d'instabilité numérique ne va pas surgir, et là comment faire si le processeur de calcul n'est pas prévu pour la détecter ?

Nous venons donc de voir que pour une application embarquée, la bibliothèque CADNA est trop coûteuse en temps de calcul. C'est pourquoi une implantation matérielle est envisageable. Notre système de calcul devra donc :

1. améliorer le temps d'exécution d'un programme par rapport à la bibliothèque CADNA
2. permettre la détection des instabilités numériques et prendre les bonnes décisions lorsqu'elles surviennent

Pour le premier point, il est certain qu'une implantation matérielle est plus rapide que son équivalent logiciel. En effet dans une implantation matérielle, le système est entièrement dédié à l'application alors qu'un logiciel fonctionne dans un environnement qui fait en général bien plus de choses que faire tourner l'application. De plus, une des causes du temps d'exécution important de la bibliothèque CADNA est le fait qu'elle nécessite à chaque opération de changer plusieurs fois de mode d'arrondi ce qui d'après [50] prend un nombre de cycles assez importants. Une solution matérielle permettrait de faire cela sans surcoût en temps puisque le changement du mode d'arrondi serait inhérent au système. Enfin, nous verrons que certains calculs dans CADNA prennent beaucoup de temps et qu'il est possible, en matériel, de réduire de façon significative ce temps de calcul.

Enfin pour le deuxième point, le système devra détecter les instabilités numériques à chaque calcul. Lorsqu'il rencontre une instabilité, soit il sait prendre la bonne décision (comme dans le cas d'une comparaison instable), soit il ne sait pas (en général c'est le concepteur de l'algorithme qui sait quoi faire en cas de problème) et il le signale, mais en aucun cas il ne fournit un résultat faux sans en avertir l'utilisateur !

De plus le système devra fournir des indications à l'utilisateur lui permettant s'il le souhaite de mettre au point son programme en fonction des instabilités numériques rencontrées. Ces indications, répertoriées dans un fichier, tracent l'évolution de la précision des opérations.

2.4 Conclusion

Dans ce chapitre nous venons d'introduire les problèmes de précision liés à l'arithmétique à virgule flottante et surtout à la représentation en machine des nombres à virgule flottante. Le but de cette thèse est de concevoir un système matériel capable de prendre en compte ces problèmes. Pour cela nous avons décidé d'implanter en matériel l'arithmétique stochastique discrète qui est une des solutions logicielles permettant de résoudre ces problèmes.

Chapitre

3 L'arithmétique stochastique discrète et ses implantations

Sommaire

Ce chapitre sera consacré à une brève présentation de l'arithmétique stochastique discrète et de ses différentes implantations. Dans un premier temps, nous rappellerons les aspects théoriques liés à la méthode CESTAC (Contrôle et Estimation Stochastique des Arrondis de Calculs) et sa modélisation par l'arithmétique stochastique discrète [83]. Puis nous verrons comment elle a été implantée en logiciel au moyen de la bibliothèque CADNA [20]. Enfin nous détaillerons l'architecture que nous avons développée pour l'implantation matérielle.

3.1 La méthode CESTAC

La méthode CESTAC (Contrôle et Estimation Stochastique des Arrondis de Calculs) a été définie par M. La Porte et J. Vignes en 1974, puis généralisée par ce dernier [64][82][84]. Cette méthode a également fait l'objet de plusieurs brevets internationaux [85][86].

L'idée majeure de cette méthode consiste à exécuter plusieurs fois le même programme de calcul en effectuant différemment les arrondis. Pour un même calcul, différents échantillons d'un résultat R sont alors obtenus. La partie commune à tous ces échantillons représente la partie fiable, l'autre étant la partie non significative, résultant de la propagation des erreurs d'arrondi.

L'arithmétique aléatoire permet d'obtenir les différents échantillons du résultat R.

3.1.1 L'arithmétique aléatoire

Tout résultat R d'une opération arithmétique qui n'est pas représentable en virgule flottante, est encadré par deux nombres à virgule flottante successifs, un par défaut F^- et l'autre par excès F^+, chacun représentant aussi légitimement le résultat exact. Déterminer le mode d'arrondi, c'est déterminer la règle qui, en fonction de R, choisit de le représenter soit par F^-, soit par F^+.

L'arithmétique aléatoire, consiste à retenir aléatoirement F^- ou F^+ avec la même probabilité $\frac{1}{2}$. C'est le mode d'arrondi aléatoire. Ainsi un même programme exécuté N fois, fournira N valeurs différentes du résultat. Ce mode d'arrondi aléatoire nécessite d'utiliser un générateur de nombre aléatoire.

Le but de l'arithmétique aléatoire n'est donc pas d'améliorer la précision du résultat mais seulement d'obtenir différents échantillons du même résultat, chacun obtenu avec différentes propagations d'erreur d'arrondi. Enfin à partir de ces échantillons le nombre exact de chiffres significatifs du résultat peut être estimé.

3.1.2 L'estimation de la précision

Un résultat informatique R apparaît comme une variable aléatoire. La précision de ce résultat dépend de sa moyenne (espérance mathématique) et de l'incertitude sur cette moyenne (variance). Il a été montré dans [17][19] que, sous certaines conditions vérifiées en pratique, cette variable aléatoire est modélisée au premier ordre en 2^{-p} par :

$$R \approx r + \sum_{i=1}^{n} g_i(d).2^{-p}.\alpha_i \tag{3.1}$$

où r est le résultat exact, $g_i(d)$ sont des coefficients dépendant uniquement des données et les α_i sont les quantités perdues lors des arrondis.

En exécutant N fois un programme de calcul avec l'arithmétique aléatoire, N échantillons R_i de la variable aléatoire R sont obtenus. Les α_i sont des variables aléatoires supposées indépendantes, et uniformément distribuées. La distribution commune des α_i est uniforme sur $[-1, +1]$ donc centrée.
Les deux conséquences majeures sont :
- l'espérance mathématique de la variable R est le résultat mathématique exact r,
- la distribution de R est quasi-gaussienne.

L'estimation de l'espérance mathématique d'une variable aléatoire gaussienne à partir d'un échantillon se fait à l'aide du test de Student qui fournit un intervalle de confiance pour l'espérance d'une gaussienne à partir d'un échantillon de cette gaussienne sous une

probabilité donnée.
Sous une probabilité β, le nombre de chiffres significatifs exacts du résultat $\overline{R}$ (espérance mathématique de R), c'est-à-dire le nombre de chiffres décimaux significatifs communs à $\overline{R}$ et à r, est estimé par :

$$C_{\overline{R}} = \log_{10}\left(\frac{\sqrt{N}.\left|\overline{R}\right|}{s.\tau_\beta}\right) \text{ avec } \overline{R} = \frac{1}{N}\sum_{i=1}^{N} R_i \text{ et } s^2 = \frac{1}{N-1}\sum_{i=1}^{N}(R_i - \overline{R})^2 \tag{3.2}$$

où τ_β est la valeur de la distribution de Student à $N-1$ degrés de liberté avec une probabilité β.

Ainsi, l'application de la méthode CESTAC à un programme fournissant un résultat R consiste à :

1. exécuter N fois le programme en utilisant le mode d'arrondi aléatoire pour obtenir N échantillons différents R_i du résultat R,
2. choisir l'espérance mathématique $\overline{R}$ comme résultat informatique,
3. calculer le nombre de chiffres significatifs exacts de $\overline{R}$ en utilisant la formule (3.2).

En pratique, $N = 2$ ou 3, et $\beta = 0.95$.
Pour $N = 2$, $\tau_\beta = 12.706$.
Pour $N = 3$, $\tau_\beta = 4.303$.

3.1.3 La validation et l'efficacité de la méthode CESTAC

Validation

Les équations (3.1) et (3.2) ont été établies sous les deux hypothèses principales suivantes :

Hyp. 1 *Les erreurs d'arrondi α_i se comportent comme des variables aléatoires indépendantes centrées uniformément distribuées.*

Hyp. 2 *L'approximation au premier ordre en 2^{-p} dans la modélisation de R est valide.*

Pour s'assurer la validité de la méthode, il faut donc que ces hypothèses soient respectées. Il a été prouvé dans [22] que l'hypothèse 1 était pratiquement toujours respectée.

Par contre pour s'assurer de la légitimité de l'hypothèse 2, il faut continuellement vérifier que les opérations de multiplication et de division ne produisent pas d'instabilité [19][20], c'est-à-dire que les deux opérandes pour la multiplication et le diviseur pour la division, sont toujours significatifs.

Efficacité

Non seulement la méthode CESTAC fonctionne très bien avec 2 ou 3 exécutions mais chercher à augmenter le nombre de ces exécutions est inutile. En effet, améliorer l'estimation du résultat de 1 chiffre décimal significatif nécessite de diviser par 10 la longueur de l'intervalle de confiance. Cette longueur évolue en $\frac{1}{\sqrt{N}}$ où N est le nombre d'exécutions. Il faudrait donc multiplier par 100 le nombre d'exécutions pour mathématiquement augmenter de 1 le nombre de chiffre significatif exact pour une même probabilité, ce qui mène aux limites du modèle. Augmenter N dans de telles proportions revient à mettre en évidence l'erreur de modèle sans améliorer pour autant la qualité de l'estimation. Le petit nombre d'exécutions nécessaire à une bonne utilisation de la méthode CESTAC fait tout son intérêt pratique.

En première approximation, sous l'hypothèse d'une répartition gaussienne pour R, lorsque $N = 3$, la probabilité de fournir une estimation supérieure (cas optimiste) de plus de 1 chiffre à la précision réelle est de 0.00054 et la probabilité de fournir une estimation inférieure (cas pessimiste) de plus de 1 chiffre à la précision réelle est de 0.29.
En choisissant un intervalle de confiance à 95%, un nombre de chiffres significatifs exacts minimal est garanti avec une très forte probabilité (0.99946) quitte à être souvent pessimiste de 1 chiffre.

3.1.4 La notion de zéro informatique

L'approximation au premier ordre en 2^{-p} peut être invalidée dans certaines opérations. Les résultats non significatifs jouent donc un rôle particulier dans la maîtrise de la méthode CESTAC. Ceci amène à introduire la notion de zéro informatique [81] définie par :

Définition 1 *Un résultat informatique R est un zéro informatique si $R = 0$ en étant significatif ou bien si R est quelconque mais non significatif.*

Concrètement, avec la méthode CESTAC, un résultat R, représenté par N résultats Ri, sera un zéro informatique si l'une des deux conditions suivantes est remplie.

1. $\forall\, i,\ R_i = 0$,

2. $C_{\overline{R}} \leq 0$.

Un zéro informatique est noté $\underline{0}$.
Un zéro informatique est un résultat que l'ordinateur ne peut distinguer du zéro mathématique à cause de la propagation des erreurs d'arrondi.

3.1.5 L'implantation synchrone

Pour s'assurer de la validité de la méthode CESTAC, il faut détecter les résultats intermédiaires non significatifs. Ceci nécessite d'utiliser l'implantation synchrone de la méthode c'est-à-dire qu'il faut disposer de tous les échantillons du résultat d'une opération avant d'en exécuter une autre.
L'estimation de la précision des résultats est absolument nécessaire pour la cohérence des opérations de comparaison. En effet lors d'une comparaison, il ne faut considérer que les chiffres significatifs des opérandes. Avec l'implantation synchrone, une opération de comparaison du type $A \geq B$ est vraie lorsque $\overline{A} \geq \overline{B}$ ou $A - B = \underline{0}$.
Cette implantation synchrone assure la validité de la méthode CESTAC ainsi que la cohérence des opérations de comparaison.

3.2 L'arithmétique stochastique discrète

L'arithmétique stochastique discrète a été définie dans [20][83]. Elle met en œuvre l'implantation synchrone de la méthode CESTAC. Un nombre stochastique possède N composantes, chacune étant un échantillon R_i du résultat R. A chaque nombre stochastique est associé son nombre de chiffres significatifs exacts donné par l'équation (3.2). Les quatre opérations élémentaires sur les nombres stochastiques sont l'addition (notée s^+), la soustraction (s^-), la multiplication (s^*) et la division ($s^/$). Les opérations de comparaison sont définies comme suit :
– égalité ($s^=$) : $X_1 \; s^= \; X_2$ ssi $X_1 \; s^- \; X_2 = \underline{0}$
– inégalité stricte ($s^>$)[1] : $X_1 \; s^> \; X_2$ ssi $\overline{X_1} > \overline{X_2}$ et $X_1 \; s^- \; X_2 \neq \underline{0}$
– inégalité large ($s^\geq$)[1] : $X_1 \; s^\geq \; X_2$ ssi $\overline{X_1} \geq \overline{X_2}$ ou $X_1 \; s^- \; X_2 = \underline{0}$

[1]Les inégalités inverses ($<$ et $\leq$) sont obtenues de la même manière en inversant le sens des inégalités

3.3 Le logiciel CADNA

L'implantation logicielle de la méthode CESTAC synchrone a été réalisée sous la forme d'une bibliothèque nommée CADNA[2] (Control of Accuracy and Debugging for Numerical Applications) [20]. Elle a pour but de pouvoir estimer l'impact des erreurs d'arrondi dans les calculs scientifiques. De plus, CADNA intègre tous les concepts de l'arithmétique stochastique discrète ce qui permet de contrôler les branchements.

CADNA s'applique à des programmes écrits en FORTRAN, C, C++ ou ADA et se présente sous la forme d'une bibliothèque utilisable après la compilation lors de l'édition de liens.

Le programmeur dispose de nouveaux types numériques : les types stochastiques. Tous les opérateurs arithmétiques, ainsi que les fonctions élémentaires concernant les types stochastiques sont redéfinis. Le contrôle des erreurs d'arrondi s'effectue uniquement sur les types stochastiques. En sortie, seuls les chiffres significatifs exacts sont affichés ce qui permet de visualiser très facilement la précision des résultats. CADNA affiche également les zéros informatiques (résultats non significatifs).

CADNA permet un véritable débogage numérique. En effet les instabilités numériques peuvent être détectées en cours d'exécution du programme. Il s'agit d'un débogage dynamique qui porte, non pas sur la validité de l'écriture du programme, mais sur les capacités de l'ordinateur à fournir des résultats corrects en l'exécutant.

Bien sûr, CADNA inclu tous les contrôles nécessaires pour garantir la fiabilité de l'estimation des erreurs d'arrondi fournie par la méthode CESTAC. Ces contrôles, imposés par l'étude théorique, permettent au logiciel de s'autovalider puisqu'il est capable de déterminer à quels moments les conditions de validité de la méthode CESTAC ne sont plus vérifiées et, si tel est le cas, d'en avertir l'utilisateur.

Le débogage numérique et l'autovalidation de CESTAC se traduisent par la détection constante d'instabilités numériques susceptibles de se produire en cours d'exécution d'un

[2] http ://www-anp.lip6.fr/~cadna/

programme. L'utilisateur est averti de ces instabilités par l'intermédiaire d'un fichier de trace géré par la bibliothèque CADNA. A chaque instabilité, une trace numérotée est laissée dans ce fichier sous la forme d'un message.

Par exemple, les deux instabilités fondamentales sont :
- DIVISION INSTABLE, cela signifie que lors d'une division, le dénominateur est un zéro informatique.
- TEST INSTABLE, cela signifie que dans l'évaluation de A $\leq$ B, (A-B) est un zéro informatique et que par conséquent, c'est la branche d'égalité qui sera exécutée. Il faut noté que l'utilisateur est averti de cela car la réponse informatique du test (la branche d'égalité est exécutée) peut être différente de la réponse mathématique puisque ce test s'effectue sur des nombres non significatifs.

Après chaque exécution, il faut consulter le fichier trace et analyser les causes de chaque message laissé dans le fichier. Ceci se fait très simplement à l'aide du débogueur symbolique. Les traces sont générées par une procédure interne à CADNA. En plaçant, sous débogueur symbolique, un stop à l'entrée de cette procédure, le programme s'arrête à chaque fois qu'une trace est écrite dans le fichier. L'historique de l'appel fournit alors la ligne du code source responsable de l'instabilité.

Enfin, CADNA permet de prendre en compte les erreurs de données dans l'estimation de la précision.

3.4 L'arithmétique stochastique discrète en matériel

Nous venons de voir que les fonctions spécifiques à l'arithmétique stochastique discrète sont :
- L'arrondi aléatoire
- Le calcul du nombre de bits significatifs
- La détection des zéros informatiques
- Le contrôle des opérations de comparaison
- Le calcul du résultat avec sa précision associée

Nous avons réalisé en matériel les quatre premiers points, le dernier se faisant de manière logicielle. Nous allons maintenant en détailler l'implantation.

3.4.1 Arrondi aléatoire

Nous avons vu que la méthode CESTAC requerrait de choisir aléatoirement comme mode d'arrondi soit l'arrondi par défaut (vers $-\infty$), soit l'arrondi par excès (vers $+\infty$) avec la même probabilité $\frac{1}{2}$. Pour cela il faut donc un générateur aléatoire de 0 et de 1.

Dans la version actuelle de la bibliothèque CADNA implantée en langage C++, le générateur de nombre aléatoire utilisé est la fonction `rand()`.

Le but est donc de réaliser en matériel un générateur de nombres aléatoires qui soit aussi performant que celui utilisé par CADNA au niveau logiciel. En matériel, les générateurs de nombres aléatoires utilisent des registres à décalage à rebouclages linéaires (*Linear Feedback Shift Register* ou LFSR) et servent pour le test des circuits intégrés [7]. Ils sont issus de la théorie de la division polynomiale sur les corps de Galois. Pour générer notre arrondi aléatoire, nous allons donc utiliser un LFSR. Il a été montré (annexe A) qu'un LFSR de degré 32 était un générateur de nombres aléatoires comparable à celui utilisé dans CADNA.

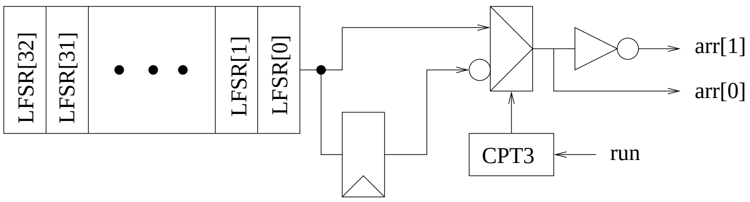


FIG. 3.1 – Arrondi aléatoire

La figure 3.1 présente l'architecture matérielle pour le calcul de l'arrondi aléatoire. Nous avons vu en 3.1.3 que dans la pratique un nombre stochastique était composé de trois échantillons ($N = 3$) d'un résultat R. Chacun de ces échantillons est une composante du nombre stochastique. La sortie du premier registre du LFSR va donc nous permettre de choisir aléatoirement, soit $+\infty$, soit $-\infty$ comme mode d'arrondi pour les opérations sur les deux premières composantes des opérandes stochastiques. Pour les troisièmes composantes, il faut s'assurer que le même mode d'arrondi ne sera pas utilisé pour toutes les composantes, car dans ce cas il n'y a plus de propagation de l'erreur d'arrondi. Ainsi, pour les troisièmes composantes, le mode d'arrondi est l'inverse du précédent. Pour faire cela nous utilisons un compteur qui permet de compter de 1 à 3. Lorsqu'il est à 3, l'arrondi sélectionné est l'inverse de l'arrondi précédent. Ce compteur s'incrémente à chaque fois qu'une nouvelle opération stochastique commence (`run=1`).

3.4.2 Calcul du nombre de bits significatifs (NBS)

Nous avons vu en 3.1.2 que le nombre de chiffres significatifs d'un nombre stochastique s'exprimait par :

$$C_{\overline{R}} = \log_{10} \frac{\sqrt{N}.|\overline{R}|}{s.\tau_\beta} \text{ avec } s^2 = \frac{1}{N-1}\sum_{i=1}^{N}(R_i - \overline{R})^2$$

Il va falloir trouver un autre moyen pour calculer cette quantité, car il serait très pénalisant en matériel d'implanter directement cette fonction. Nous nous plaçons maintenant au niveau matériel et nous ne parlerons alors plus de chiffres significatifs, mais de bits significatifs.

Une approche intuitive

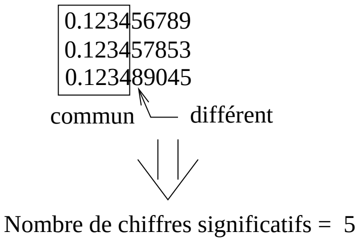


FIG. 3.2 – Approche simpliste

En fait l'information requise par cette fonction est le nombre de chiffres en commun des trois résultats. Une approche simpliste consisterait à comparer chaque chiffre de même rang des résultats et dès qu'ils diffèrent c'est le nombre de chiffres significatifs (figure 3.2).

Mais cette approche ne tient pas compte du fait que deux nombres très proches comme 0.9999999 et 1.0000000 n'ont aucun chiffres en commun alors que leur nombre de chiffres significatifs donné par la méthode CESTAC est 7. Pour remédier à cela, une autre possibilité est de calculer les distances entre les résultats intermédiaires R_i et de chercher la position du premier bit non nul de la distance maximale. Le résultat est le nombre de bits significatifs. L'algorithme suivant décrit cette nouvelle méthode de calcul du nombre de bits significatifs :

- Calcul des distances entières $d_1 = |Bin(R_1) - Bin(R_2)|$, $d_2 = |Bin(R_1) - Bin(R_3)|$, $d_3 = |Bin(R_3) - Bin(R_2)|$, où $Bin(R_i)$ est l'entier associé à la représentation en binaire de R_i
- Recherche de la distance maximale $d = \max(d_1, d_2, d_3)$
- Si l'exposant de la distance d n'est pas nul, cela signifie qu'un des R_i est au moins le double d'un autre, et dans ce cas, le nombre de bits significatifs du résultat est nul. Dans l'autre cas, le nombre de bits significatifs est la position du premier bit non nul de la mantisse de d

Une démonstration de la similitude de la valeur du nombre de bits significatifs fourni par cette méthode et celle donnée par l'équation 3.2 a été établie par Jean-Marie Chesneaux et est présentée dans l'annexe B.

Performances

Nous avons voulu comparer les performances entre les deux méthodes de calcul du nombre de chiffres significatifs. Le tableau 3.1 présente le nombre de cycles nécessaires pour effectuer ce calcul avec l'équation (3.2) et notre méthode de calcul. Ce comparatif est réalisé sur un Pentium III cadencé à 500 MHz en consultant la valeur du compteur de cycles du processeur. Comme de nombreux cycles sont utilisés pour faire autre chose que la comparaison (pour le système d'exploitation par exemple), nous avons fait environ 4.10^8 itérations de calcul pour chacune des méthodes et nous obtenons ainsi le nombre de cycles minimal, maximal et moyen nécessaires pour effectuer le calcul avec chacune des méthodes.

	Min (cycles)	Max (cycles)	Moyenne (cycles)
bibliothèque CADNA	895	73138721	1082
notre méthode	113	919376	126
rapport	7.92	79.5	8.59

TAB. 3.1 – Comparaison entre les deux méthodes

Implantation matérielle

Deux nombres à virgule flottante dont la distance entre les exposants est supérieure ou égale à 2, auront un nombre de bits significatifs nul (voir annexe B). Ainsi les calculs de distance sur les composantes R_1, R_2 et R_3 peuvent s'effectuer en entier, puisque pour les exposants cela permettra de savoir s'ils sont du même ordre de grandeur ou non. Ainsi, comme le montre la figure 3.3, notre méthode de calcul s'implante très facilement en matériel au moyen de :

- trois opérateurs de distances (dont l'architecture est présentée en annexe C) calculant la valeur absolue de la différence entière des résultats deux à deux,
- un OU logique permettant d'avoir la position du premier bit à 1 la plus petite,
- un compteur de zéros en tête qui permet de calculer la position du premier bit à 1 minimale sur les mantisses
- et un peu de logique pour obtenir le nombre de bits significatifs final qui est nul si une des différences des exposants est strictement positive et le nombre calculé précédemment sinon.

Ainsi le résultat renvoyé par ce bloc est le nombre de bits significatifs (signal NBS) d'un résultat stochastique R représenté par ses trois composantes R_1, R_2 et R_3.

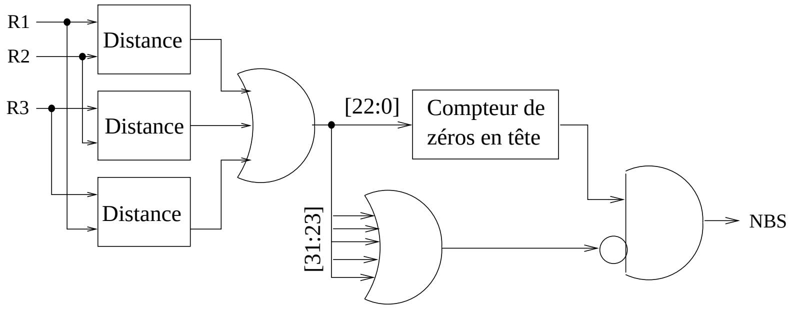


FIG. 3.3 – Calcul du nombre de bits significatifs en matériel

Conclusion

Nous avons établi une nouvelle façon de calculer le nombre de bits significatifs d'un nombre stochastique qui s'implante facilement en matériel.

De plus le temps de calcul par cette nouvelle méthode est nettement inférieur à celui de la méthode CESTAC, ce qui laisse déjà présager les meilleures performances de notre architecture matérielle par rapport à la bibliothèque CADNA telle qu'elle est implantée actuellement.

3.4.3 Détection du zéro informatique

Pour détecter un zéro informatique, nous avons vu en 3.1.4 qu'il fallait vérifier une des conditions suivantes :

1. $\forall\, i,\ R_i = 0$,
2. $C_{\overline{R}} \leq 0$.

La figure 3.4 présente l'implantation matérielle de la détection du zéro informatique. Un OU logique sur les différentes composantes du nombre stochastique (R1, R2 et R3) et sur le nombre de bits significatifs (NBS) permet de tester si ils sont nuls ou non. Ensuite un peu de logique détecte le zéro informatique, c'est-à-dire si les trois composantes ou le nombre de bits significatifs sont nuls.

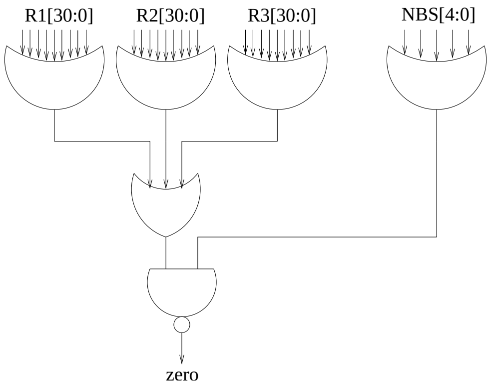


FIG. 3.4 – Détection du zéro informatique

3.4.4 Contrôle des opérations de comparaison

Nous avons vu en 3.2 que pour comparer deux nombres stochastiques, il fallait non seulement vérifier si leur différence était ou non un zéro informatique et en plus comparer leurs espérances mathématiques. Nous venons de voir en 3.4.3 comment s'effectuait la détection des zéros informatiques. Il reste maintenant à définir le matériel nécessaire à la comparaison des espérances mathématiques des deux nombres stochastiques. Dans la pratique, cette espérance mathématique est définie par :

$$\overline{R} = \frac{1}{3}\sum_{i=1}^{3} R_i$$

La comparaison des espérances mathématiques n'est nécessaire que lorsque la différence des opérandes stochastiques n'est pas un zéro informatique. Dans ce cas, les exposants des composantes sont du même ordre de grandeur (annexe B) et pour comparer les espérances mathématiques, il suffit donc de comparer la somme des mantisses des composantes de chaque nombre stochastique. Pour effectuer les deux sommes, nous allons utiliser un additionneur accumulateur, qui additionnera la mantisse de chaque nouvelle composante du nombre stochastique avec la somme des précédentes. A chaque nouveau calcul, l'accumulateur sera initialisé à 0. Puis à l'aide d'un comparateur nous pourrons comparer les sommes ainsi obtenues (figure 3.5).

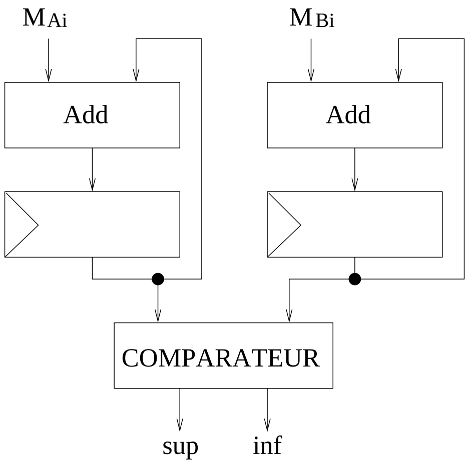


FIG. 3.5 – Contrôle des comparaisons

3.4.5 Calcul du résultat

Le calcul du résultat final avec sa précision associée n'est nécessaire qu'une seule fois lors de son affichage à la fin de l'exécution du programme. Comme il est peu utilisé, il est inutile de le calculer avec un matériel dédié qui nécessiterait l'utilisation d'un multiplieur flottant. Pour cela nous avons décider d'effectuer ce calcul de manière logicielle.

3.4.6 Assemblage du tout

L'ensemble du matériel décrit précédemment est regroupé au sein d'une unité flottante stochastique. Cette unité possède deux parties (figure 3.6) et peut fonctionner soit en mode standard (sans contrôle et estimation des erreurs d'arrondi), soit en mode CESTAC :

- Une unité de calcul flottant IEEE-754 qui effectue les opérations flottantes suivant la norme et dont l'architecture sera présentée dans le chapitre 4. L'opération effectuée est $R = A \ op \ B$.
- Un bloc CESTAC qui réalise toutes les fonctionnalités liées à l'arithmétique stochastique discrète : calcul du nombre de bits significatifs (NBS), détection des zéros informatiques (zero), arrondi aléatoire et contrôle des opérations de comparaison (sup et inf).

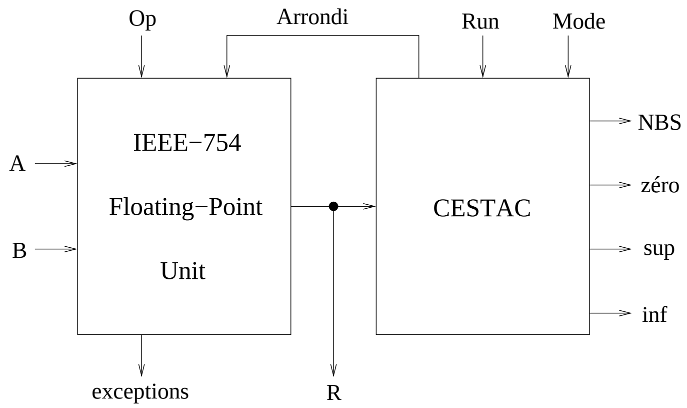


FIG. 3.6 – L'unité flottante stochastique

Le déroulement d'une opération stochastique, en fonction du nombre d'étages de pipeline de l'unité flottante, est présenté par le tableau 3.2. A chaque fois que le calcul sur une nouvelle composante des opérandes commence, les deux composantes entrent dans les accumulateurs permettant d'effectuer le contrôle des opérations de comparaison. A chaque fois qu'un résultat est disponible, il est stocké dans un registre. Lorsque les trois résultats sont disponibles, il reste un cycle pour effectuer toutes les opérations nécessaires à l'arithmétique stochastique discrète, à savoir le calcul du nombre de bits significatifs du résultat, la détection du zéro informatique, et la comparaison des espérances des opérandes pour le contrôle des opérations de comparaison.
En fonction du nombre d'étages de pipeline de l'unité flottante, une opération stochastique peut se dérouler en quatre, cinq ou six cycles.

Cycle	FPU	Bloc CESTAC
1	A_1 *Op* B_1	- A_1 et B_1 dans accumulateurs - stocker R_1
2	A_2 *Op* B_2	- stocker R_2 - A_2 et B_2 dans accumulateurs
3	A_3 *Op* B_3	- stocker R_3 - A_3 et B_3 dans accumulateurs
4		- NBS(R)[3], ZINF(R)[4]et CMP(A,B)[5]

(a) Sans pipeline

Cycle	FPU		CESTAC
	étage 1	étage 2	
1	A_1 *Op* B_1		- A_1 et B_1 dans accumulateurs
2	A_2 *Op* B_2	A_1 *Op* B_1	- stocker R_1 - A_2 et B_2 dans accumulateurs
3	A_3 *Op* B_3	A_2 *Op* B_2	- stocker R_2 - A_3 et B_3 dans accumulateurs
4		A_3 *Op* B_3	- stocker R_3
5			- NBS(R), ZINF(R) et CMP(A,B)

(b) Avec deux étages de pipeline

Cycle	FPU			CESTAC
	étage 1	étage 2	étage 3	
1	A_1 *Op* B_1			- A_1 et B_1 dans accumulateurs
2	A_2 *Op* B_2	A_1 *Op* B_1		- A_2 et B_2 dans accumulateurs
3	A_3 *Op* B_3	A_2 *Op* B_2	A_1 *Op* B_1	- stocker R_1 - A_3 et B_3 dans accumulateurs
4		A_3 *Op* B_3	A_2 *Op* B_2	- stocker R_2
5			A_3 *Op* B_3	- stocker R_3
6				- NBS(R), ZINF(R) et CMP(A,B)

(c) Avec trois étages de pipeline

TAB. 3.2 – Ordonnancement des opérations

[3] NBS(X) : nombre de bits significatifs de X
[4] ZINF(X) : teste si X est un zéro informatique
[5] CMP(A,B) : comparaison des espérances mathématiques de A et B

L'architecture détaillée du bloc CESTAC est présentée figure 3.7.

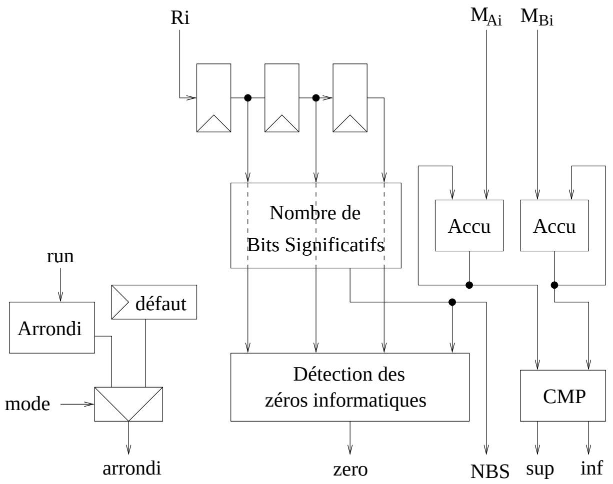


FIG. 3.7 – Bloc CESTAC

3.5 Conclusion

Nous venons de présenter l'arithmétique stochastique discrète et son implantation logicielle sous la forme de la bibliothèque CADNA. Nous avons également détailler l'architecture matérielle développée, au travers d'opérateurs dédiés pour chacune des fonctionnalités essentielles de la méthode à savoir : l'arrondi aléatoire, le calcul du nombre de chiffres significatifs, la détection des zéros informatiques et le contrôle des opérations de comparaison. Enfin nous avons montré comment ces fonctionalités ont été regroupées au sein d'une unité flottante stochastique. Cette unité travaillant sur les résultats d'opérations flottantes, nous allons donc maintenant nous consacrer à l'élaboration d'une unité flottante à la norme IEEE-754.

Chapitre

4

L'unité flottante

Sommaire

Nous avons vu dans le chapitre 3 que l'implantation matérielle de l'arithmétique stochastique discrète prenait en entrée les résultats d'une unité flottante standard. Ce chapitre va donc être dédié à la réalisation matérielle d'une unité flottante compatible avec la norme IEEE-754 qui a été présentée au chapitre 2. Cette unité flottante intègre les opérateurs d'addition, soustraction, multiplication, division, comparaison, conversion entier vers flottant et vice et versa.

4.1 État de l'art

De nombreuses recherches ont déjà été menées sur les unités de calcul en arithmétique à virgule flottante. Selon le type d'applications, les concepteurs de ces unités ont choisi ou non de les rendre compatibles avec la norme IEEE-754 [4]. Par exemple dans [44] les concepteurs ont décidé de ne pas implanter les nombres spéciaux (infinis, NaN), les nombres dénormalisés, ainsi que l'arrondi. Ce choix a été motivé par le fait qu'effectuer ces différents traitements coûtait cher en performances et que cela n'était pas nécessaire aux applications graphiques qu'ils visaient.

Nous avons ensuite les nombreuses études menées par IBM. Toutes les architectures développées se basent sur celle de l'IBM S/360 [3]. A cette époque, il n'y avait pas encore de norme et les ingénieurs d'IBM ont choisi de coder les nombres en base 16. Tous les opérateurs flottants effectuent donc leurs opérations en hexadécimal. Les opérateurs implantés sont l'addition et la multiplication. Les nouvelles générations de processeurs, comme le S/390 [75] se basent toujours sur ce système en base 16, mais effectuent des conversions de l'hexadécimal vers le binaire, et vice et versa, pour être totalement compatibles avec la norme. Certaines architectures comme l'IBM RISC S/6000 [54] ou le PowerPC 603e [45] contiennent un opérateur d'addition/multiplication qui permet d'effectuer l'opération $a + b \times c$ avec un seul arrondi à la fin du calcul. De plus le PowerPC 603e possède un mode non standard qui permet d'augmenter les performances lorsque toutes les spécifications de la norme ne sont pas requises.

Puis nous pouvons citer les nombreux travaux effectués par l'université de Stanford autour du projet SNAP (*Stanford Subnanosecond Arithmetic Processor*) [58]. L'originalité de leur méthode repose sur une architecture à latence variable où le résultat d'une opération flottante est disponible après un certain nombre de cycles qui dépend des valeurs des opérandes. Par contre ce genre de méthode pose de gros problème de synchronisation des données. Cette unité flottante est totalement compatible avec la norme et dispose des opérateurs d'addition, multiplication et division.

Enfin nous avons la famille des Pentium [1] qui implante des unités flottantes respectant la norme. Ils utilisent un format étendu avec des mantisses sur 64 bits afin d'augmenter la précision des calculs. Ils ont également une pile interne de 8 nombres à virgule flottante sur 80 bits afin de stocker des résultats intermédiaires et ainsi d'augmenter les performances.

4.2 Additionneur flottant

4.2.1 Algorithme

Nous présentons ici l'algorithme mettant en œuvre l'addition flottante de la norme IEEE-754 [40][59]. Les notations expA et expB sont utilisées pour les exposants des opérandes A et B.

1. Calcul de l'exposant : l'exposant résultat est celui de l'opérande le plus grand.
2. Alignement : la mantisse du plus petit opérande est décalée à droite de $d = |expA - expB|$ bits afin de l'ajuster avec celle de l'autre opérande.
3. Addition des mantisses : les deux mantisses sont ajoutées (ou soustraites si l'opération à effectuer est une soustraction). Si le résultat est négatif, il est converti en son opposé afin de s'assurer qu'il est toujours positif.
4. Normalisation : la mantisse résultat est normalisée, de telle sorte que son bit de poids fort soit 1 (décalage à gauche du nombre de zéros en tête de mantisse) et l'exposant mis à jour (un décalage d'un bit à gauche entraîne un décrément de un de l'exposant).

5. Arrondi : le résultat est arrondi conformément à la norme IEEE-754 et l'exposant mis à jour en cas d'*overflow* (incrément de un).

6. Signe : le signe du résultat est celui de l'opérande qui a la plus grande valeur absolue.

4.2.2 Architecture matérielle

Les différentes étapes décrites dans l'algorithme précédent ont été réalisées en matériel. Nous allons maintenant en étudier l'architecture (figure 4.1).

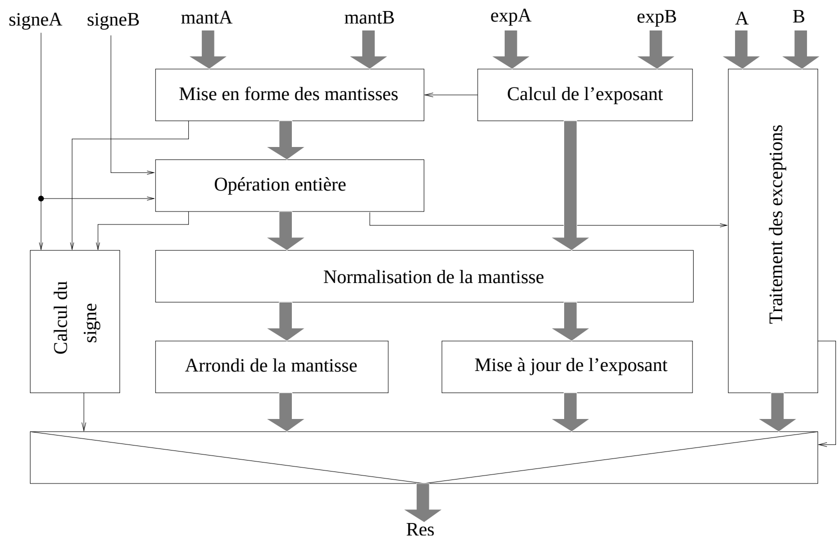


FIG. 4.1 – Architecture générale d'un opérateur flottant

Calcul de l'exposant

Cette étape consiste à calculer la distance entre les exposants des deux opérandes, et à retourner comme exposant résultat le plus grand des deux.
Le calcul de la valeur absolue de la différence se fait à l'aide d'un opérateur de distance (annexe C). Ce bloc permet également la détection des nombres dénormalisés (exposant nul). La figure 4.2 présente l'architecture du bloc. Le signal `ApgB` indique que l'exposant de l'opérande A est supérieur ou égal à celui de l'opérande B et le signal `normA` (resp. `normB`) indique que l'opérande A (resp. B) est normalisé.

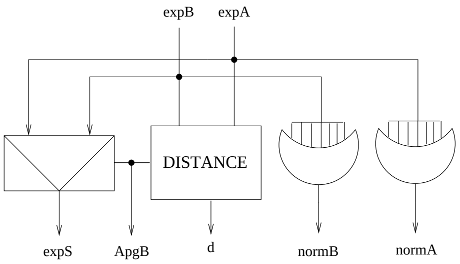


FIG. 4.2 – Calcul de l'exposant

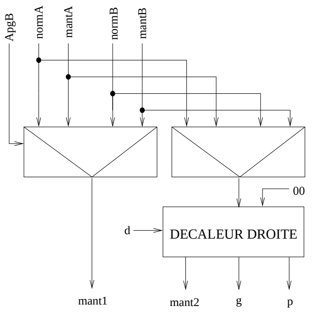


FIG. 4.3 – Mise en forme des mantisses

Mise en forme des mantisses

Ici, le bit implicite est rajouté en poids fort des mantisses (1 si le nombre est normalisé, 0 sinon) et celles-ci sont éventuellement échangées afin d'avoir sur le signal mant1 (resp. mant2), la mantisse de l'opérande qui a l'exposant le plus grand (resp. petit). De plus un bit de garde (g) est rajouté en poids faible de la mantisse de l'opérande qui a le plus petit exposant. Il servira lors du calcul de l'arrondi. Puis cette mantisse est décalée à droite de la valeur d, calculée par le bloc précédent. Enfin le bit persistant (p) vaut 1 si lors du décalage un bit valant 1 a été expulsé. Ce bit servira également lors du calcul de l'arrondi. L'architecture de ce bloc est détaillée dans la figure 4.3.

Opération entière

Ce bloc permet d'additionner (ou de soustraire) les deux mantisses. Pour cela un opérateur de distance est utilisé pour calculer $|mant1 \pm mant2|$ en fonction de l'opération à effectuer. Une soustraction (signal `sous`=1) a lieu lorsque les deux opérandes sont de signes contraires et que l'opération flottante est une addition, ou lorsque les deux opérandes sont de même signe et que l'opération flottante est une soustraction. De plus, si lors d'une soustraction, la deuxième mantisse est plus grande que la première, le signal `neg` est positionné à 1 pour indiquer que le résultat de la soustraction était négatif. Ce signal servira lors du calcul du signe du résultat. L'architecture de ce bloc est présentée par la

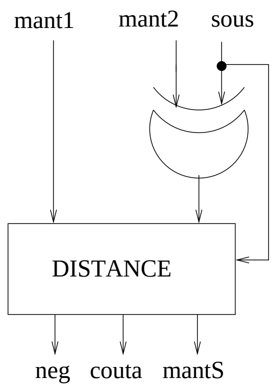


FIG. 4.4 – Addition

figure 4.4.

Normalisation

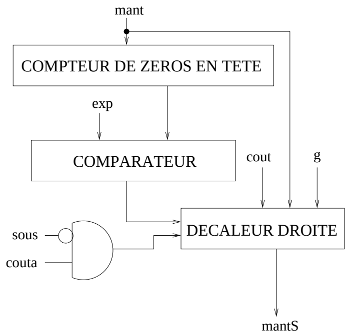


FIG. 4.5 – Normalisation

Une première étape consiste à chercher la position du premier bit à 1 de la mantisse sortant du bloc précédent. Ceci se fait avec un compteur de zéros en tête de mantisse. Ensuite la mantisse est décalée de 1 bit à droite dans le cas de l'addition lorsqu'il y a une retenue sortante, sinon elle est décalée à gauche du minimal entre la position du premier bit à 1 calculée et la valeur de l'exposant sortant du bloc de calcul de l'exposant. Lors du décalage à gauche, le bit de garde est d'abord injecté. La figure 4.5 présente l'architecture de ce bloc.

Arrondi

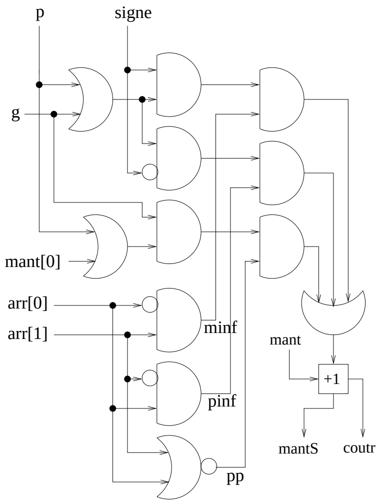


FIG. 4.6 – Arrondi

Ce bloc réalise l'arrondi de l'addition flottante conformément à la norme IEEE-754. En fonction de l'arrondi, du bit de garde, du bit persistant et du signe du résultat, la mantisse est augmentée de un selon le mode d'arrondi :
– vers 0 : pas d'incrément
– vers $-\infty$: incrément si $signe.(p+g)$
– vers $+\infty$: incrément si $\overline{signe}.(p+g)$
– au plus près : incrément si $g.(p+mant_0)$

De plus si arrondir provoque une retenue sortante le signal `coutr` est mis à 1, ce qui permettra d'augmenter de 1 l'exposant lors de sa mise à jour. La figure 4.6 présente l'architecture de ce bloc.

Mise à jour de l'exposant

Selon les décalages effectués, l'exposant devra être augmenté de 2 (décalage d'un bit à droite et retenue sortante lors de l'arrondi), de 1 (décalage d'un bit à droite ou retenue sortante lors de l'arrondi), diminué (décalage à gauche), ou rester inchangé (pas de décalage). Pour gérer l'ensemble de ces possibilités, un multiplexeur permet de choisir la valeur à additionner (ou soustraire) en fonction du décalage (figure 4.7).

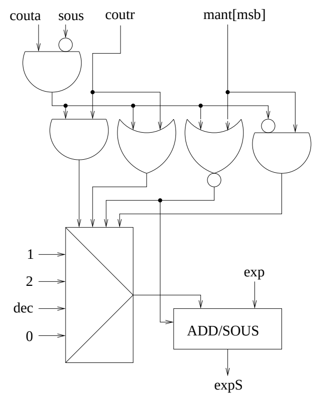


Fig. 4.7 – Mise à jour de l'exposant

Signe

Ce bloc calcule le signe du résultat en fonction du type de l'opération, de la permutation lors de la mise en forme des mantisses, d'un résultat négatif lors de la soustraction des mantisses (`neg=1`) et du signe de l'opérande A. La table de vérité pour le calcul du signe du résultat et le réseau booléen associé sont donnés par la figure 4.8.

sous	perm	neg	signeA	signe
0	X	X	0	0
0	X	X	1	1
1	1	X	0	1
1	1	X	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0

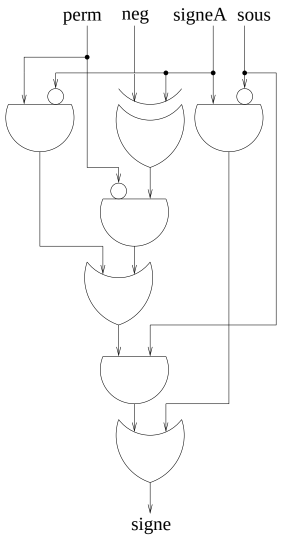


Fig. 4.8 – Signe du résultat

Bloc exception

Ce bloc permet de calculer le résultat de l'addition de deux nombres spéciaux, comme indiqué par le tableau 4.1. La norme IEEE-754 ne précisant pas les valeurs des résultats d'opération avec des NaN, nous avons choisi comme référence l'architecture SPARC V9 [77], car c'est celle qui sert pour la validation de nos architectures.

	0	Nb	$+\infty$	$-\infty$	NaN2q	NaN2s
0	0	Nb	$+\infty$	$-\infty$	NaN2q	NaN2q
Nb	Nb	Nb	$+\infty$	$-\infty$	NaN2q	NaN2q
$+\infty$	$+\infty$	$+\infty$	$+\infty$	NaNs	NaN2q	NaN2q
$-\infty$	$-\infty$	$-\infty$	NaNs	$-\infty$	NaN2q	NaN2q
NaN1q	NaN1q	NaN1q	NaN1q	NaN1q	NaN2q	NaN2q
NaN1s	NaN1q	NaN1q	NaN1q	NaN1q	NaN1q	NaN2q

TAB. 4.1 – Traitement des exceptions

Ce bloc permet également de traiter les *overflows* (exp = 1..10 et décalage à droite) en fonction du mode d'arrondi et conformément à la norme (tableau 4.2).

Arrondi	Signe	Résultat
au plus près	0	$+\infty$
	1	$-\infty$
$-\infty$	0	Plus grand nombre positif[1]
	1	$-\infty$
$+\infty$	0	$+\infty$
	1	Plus petit nombre négatif[2]
vers zéro	0	Plus grand nombre positif
	1	Plus petit nombre négatif

TAB. 4.2 – Résultat de l'addition flottante lors d'un *overflow*

[1] signe=0, exp=1..10, mant=1..1
[2] signe=1, exp=1..10, mant=1..1

Gestion des drapeaux d'exception

Les drapeaux d'exception signalant un résultat inexact, un *overflow* ou une opération invalide sont positionnés. Les autres drapeaux, qui ne sont pas nécessaires à l'addition sont mis à 0 (division par 0 et *underflow*). Le drapeau de résultat inexact est mis à 1 lorsqu'il y a un arrondi ou que le bit évacué lors d'un décalage à droite de la mantisse vaut 1. L'*overflow* est mis à 1 lorsqu'il y a un incrément de l'exposant alors que celui-ci vaut 11..10. Le drapeau d'opération invalide est mis à 1 lorsqu'un opérande est un NaNs ou que les deux opérandes sont des infinis.

4.2.3 Améliorations possibles

Il est possible de réduire la chaîne longue en effectuant l'arrondi en même temps que l'addition des mantisses [67]. Cela nécessite de faire en parallèle $(mantA + mantB)$, $(mantA + mantB + 1)$ et $(mantA + mantB + 2)$ à l'aide d'un additionneur à résultats multiples (annexe E).

L'algorithme de l'addition flottante présente l'énorme désavantage de se dérouler de manière séquentielle, ce qui en matériel aboutit à un chemin critique assez long. Pour diminuer la chaîne critique de l'algorithme, il peut être modifier en tenant compte de certaines remarques [67][40][59][57] :

1. L'aiguillage des mantisses permet de s'assurer de toujours soustraire le plus petit opérande du plus grand. Du coup, la conversion à l'étape 3 est inutile, sauf lorsque les exposants sont égaux. Mais dans ce cas, il n'y a pas d'alignement initial et par conséquent le résultat de l'addition est exact et n'a pas besoin d'être arrondi. La conversion à l'étape 3 et l'arrondi à l'étape 5 sont donc mutuellement exclusifs.

2. Dans le cas d'une addition, il n'y a pas d'annulation des bits de poids forts du résultat. Du coup il n'y a pas besoin de normaliser le résultat. Pour la soustraction, deux cas se présentent :
   - $d > 1$: les mantisses doivent être alignées, et le résultat nécessitera au plus un décalage à gauche d'un bit
   - $d \leq 1$: un décalage d'un bit au plus est nécessaire pour l'alignement et le résultat doit être normalisé

   Du coup l'alignement et la normalisation deviennent mutuellement exclusifs.

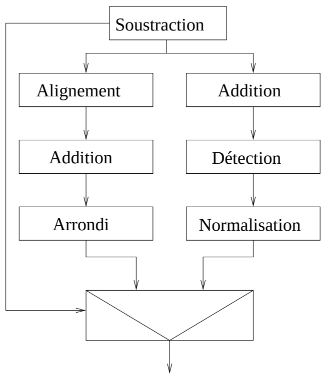


FIG. 4.9 – Algorithme modifié

Cela amène à définir deux branches [67] (figure 4.9) pour la réalisation de l'addition : une lorsque $d > 1$ et une autre lorsque $d \leq 1$. Les deux branches fonctionnent en parallèle, la première réalisant les étapes d'alignement, d'addition des mantisses et d'arrondi, la seconde l'addition des mantisses, la conversion, la détection du premier bit à 1 et la normalisation. Le choix du résultat adéquate dépend du résultat de la différence des exposants.

Une autre amélioration possible est d'effectuer l'étape de détection du premier bit à 1 en parallèle avec l'addition des mantisses [41][68][60][78][13] (annexe D). En effet cette détection est très coûteuse en temps et le fait de la réaliser en parallèle avec l'addition permet de nettement diminuer le temps critique.

4.3 Multiplieur flottant

4.3.1 Algorithme

Nous présentons ici l'algorithme mettant en œuvre la multiplication flottante de la norme IEEE-754 [40].

1. Calcul de l'exposant : l'exposant résultat vaut $expA + expB - biais$.
2. Multiplication : les mantisses des deux opérandes sont multipliées.
3. Arrondi : le résultat est arrondi conformément à la norme IEEE-754 et l'exposant mis à jour en cas d'*overflow* (incrément de un).
4. Signe : le signe du résultat est positif si les signes des deux opérandes sont les mêmes et négatif sinon.

Cet algorithme ne permet pas de traiter les nombres dénormalisés. Pour cela, il faut rajouter une étape de normalisation après la multiplication. Cette normalisation consiste à décaler à gauche (resp. à droite) la mantisse du nombre de bits à 0 en tête (resp. de la valeur absolue de l'exposant) si l'exposant est positif (resp. négatif). L'exposant devra ensuite être mis à jour (un décalage d'un bit à gauche entraîne un décrément de 1 de l'exposant et un décalage d'un bit à droite un incrément).

4.3.2 Architecture matérielle

L'architecture générale du multiplieur flottant, traitant les nombres dénormalisés, est sensiblement la même que celle de l'additionneur, seul diffère le contenu des blocs.

Calcul de l'exposant

Pour calculer l'exposant résultat ($expA + expB - biais$), deux additionneurs entiers sont utilisés : un pour calculer $exp = expA + expB$ et un autre pour calculer $exp - biais$.

Opération entière

Les bits implicites sont rajoutés en poids fort de chacune des mantisses et la multiplication des mantisses est effectuée à l'aide d'un multiplieur de Wallace [27][56]. Ce multiplieur est composé de trois étages :

1. un qui calcule les produits partiels,
2. un autre qui effectue la somme de ces produits partiels à l'aide d'additionneurs élémentaires afin de les réduire à deux termes en utilisant la méthode de Dadda,
3. et un dernier qui réalise la somme de ces deux termes.

Normalisation

Une première étape consiste à chercher la position du premier bit à 1 de la mantisse sortant du bloc précédent. Ceci se fait avec un compteur de zéros en tête de mantisse. Ensuite un décaleur gauche/droite est utilisé pour normaliser la mantisse. Celle-ci est décalée à droite si l'exposant est négatif et à gauche sinon. La valeur du décalage est la valeur absolue de l'exposant lors d'un décalage à droite et le minimal entre la position du premier bit à 1 calculée et la valeur de l'exposant lors d'un décalage à gauche.

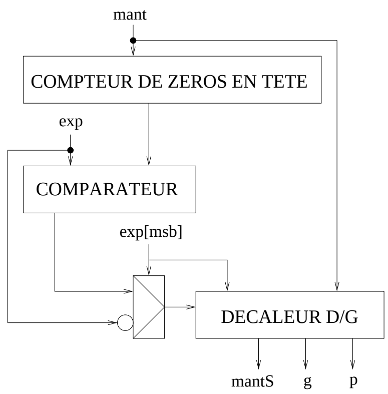


FIG. 4.10 – Normalisation

La mantisse retournée par ce bloc ne contient que les p bits de poids fort du résultat (la multiplication de deux nombres de p bits donne un résultat sur 2p bits). Le premier bit qui n'est pas gardés est tout de même conservé pour le calcul de l'arrondi (bit de garde). De plus un bit persistant est mis à 1 lorsque la partie non gardée n'est pas nulle ou que lors d'un décalage à droite il y a eu évacuation d'un bit à 1. Ce bit persistant servira également au calcul de l'arrondi.
L'architecture de ce bloc est donnée par la figure 4.10.

Arrondi

L'arrondi est réalisé de la même façon que pour l'addition.

Mise à jour de l'exposant

L'exposant résultat est ajusté s'il y a eu un décalage lors de l'étape de normalisation. Pour cela un additionneur/soustracteur entier est utilisé afin d'additionner (resp. soustraire) à l'exposant la valeur du décalage à droite (resp. gauche).

Signe

Le signe se calcule comme étant un ou-exclusif entre les signes des deux opérandes.

Bloc exception

Le traitement de la multiplication de nombres spéciaux ne diffère de celui de l'addition uniquement lorsque les opérandes sont des infinis (tableau 4.3). Pour les NaN, les résultats sont les mêmes que pour l'addition.

	0	Nb	∞
0	0	0	NaNs
Nb	0	Nb	∞
∞	NaNs	∞	∞

TAB. 4.3 – Traitement des exceptions

Gestion des drapeaux d'exception

Les drapeaux d'exception inexacte, *underflow*, *overflow* et invalide sont positionnés. Le drapeau de division par 0, qui n'est pas nécessaire à la multiplication est mis à 0. Le drapeau inexacte est mis à 1 lorsque le bit de garde ou le bit persistant vaut 1 ou qu'il y a un *overflow*. L'*underflow* indique une inexactitude sur un nombre dénormalisé (l'exposant est nul et le bit de garde ou le bit persistant vaut 1). L'*overflow* est mis à 1 lorsque les deux retenues sortantes des deux additionneurs du bloc de calcul de l'exposant valent 1 ou lorsque l'exposant résultat valant $2^n - 2$ (exp=1..10) est augmenté suite à une retenue sortante lors de l'arrondi. Le drapeau de l'exception invalide est mis à 1 lorsqu'un opérande est un NaNs ou que les deux opérandes sont des infinis.

4.4 Division flottante

4.4.1 Algorithme

Nous présentons ici l'algorithme mettant en œuvre la division flottante de la norme IEEE-754 [40].

1. Calcul de l'exposant : l'exposant résultat vaut $expA - expB + biais$.
2. Division : les mantisses des deux opérandes sont divisées.
3. Arrondi : le résultat est arrondi conformément à la norme IEEE-754 et l'exposant mis à jour en cas d'*overflow* (incrément de un).
4. Signe : le signe du résultat est positif si les signes des deux opérandes sont les mêmes et négatif sinon.

Cet algorithme ne permet pas de traiter les nombres dénormalisés. Pour cela, il faut rajouter :

1. avant la division des mantisses, une étape de normalisation du diviseur. Cette étape consiste à normaliser la mantisse d'un diviseur dénormalisé (en décalant à gauche du nombre de zéros en poids fort de la mantisse). Dans ce cas, l'exposant résultat ne sera plus $(expA - expB + biais)$, mais $(expA + decB + biais)$ où $decB$ est le nombre de décalage à effectuer pour normaliser le diviseur. Le traitement des nombres dénormalisés nécessitera donc un important ajout de matériel (un compteur de zéros en tête de mantisse et un décaleur),

2. après la division, une étape de normalisation consistant à décaler à gauche (resp. à droite) la mantisse du nombre de bits à 0 en tête de mantisse (resp. de la valeur absolue de l'exposant) si l'exposant est positif (resp. négatif). L'exposant devra ensuite être mis à jour (un décalage d'un bit à gauche entraîne un décrément de 1 de l'exposant et un décalage d'un bit à droite un incrément)

4.4.2 Architecture

L'architecture générale du diviseur flottant, traitant les nombres dénormalisés, est sensiblement la même que celle de l'additionneur, seul diffère le contenu des blocs.

Calcul de l'exposant

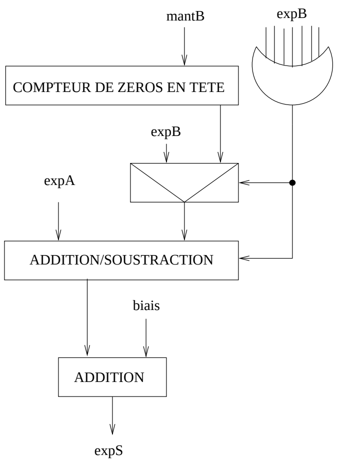


FIG. 4.11 – Calcul de l'exposant

Tout d'abord, un compteur de zéros en tête de mantisse est utilisé pour calculer le nombre de décalages à effectuer sur la mantisse d'un diviseur dénormalisé. Un multiplexeur permet de choisir entre la valeur du décalage et la valeur de l'exposant en fonction du signal indiquant si le diviseur est normalisé ou non. Si le diviseur est normalisé, ce nombre est soustrait de l'exposant du dividende, sinon il est additionné. Ensuite le biais est additionné au résultat obtenu. La figure 4.11 présente l'architecture du bloc.

Mise en forme des mantisses

La mantisse d'un diviseur dénormalisé est décalée à gauche de la valeur calculée par le bloc *Calcul de l'exposant*. Ensuite le bit implicite est ajouté en poids fort des mantisses des opérandes.

Division entière

Ce bloc effectue la division entière des mantisses. Le diviseur entier utilisé est un diviseur redondant qui fournit un résultat en notation redondante [2]. L'algorithme utilisé est le suivant [80] :

$R_0 = A$

Pour i=0 à N-1 Faire

$$\begin{array}{rccl} \text{si} & -B < R_i < B & \text{alors} & Q_i = 0\,;\ R_{i+1} = 2.R_i \\ \text{sinon si} & R_i < 0 & \text{alors} & Q_i = -1\,;\ R_{i+1} = 2.(R_i + B) \\ \text{sinon si} & R_i > 0 & \text{alors} & Q_i = 1\,;\ R_{i+1} = 2.(R_i - B) \end{array}$$

Fin Pour

où A est le dividende, B le diviseur, Q le quotient et R le reste et N la taille du quotient.

Ce diviseur est composé de N tranches constituées d'une cellule de tête et de N cellules de queue [2]. La cellule de tête permet de calculer la valeur d'un bit du quotient Q_i et de sélectionner l'opération à effectuer sur les restes partiels R_i (addition ou soustraction). Les cellules de queue permettent de calculer les restes partiels en fonction de l'opération sélectionnée. Le résultat est ensuite converti pour revenir en une notation en complément à deux à l'aide d'un convertisseur permettant d'effectuer la conversion dès l'apparition des premiers bits du quotient sans avoir à attendre d'avoir terminé la division. De plus si le reste de cette division n'est pas nul, le bit persistant p est positionné à 1 pour indiquer qu'il y a eu une perte d'information lors de la division (cela servira pour le calcul de l'arrondi).

Autres blocs

Toutes les autres étapes (normalisation, arrondi, mise à jour de l'exposant et calcul du signe) sont réalisées de la même façon que pour la multiplication.

Bloc exception

Le traitement de la division de nombres spéciaux ne diffère de celui de l'addition uniquement lorsque les opérandes sont des infinis ou nuls (tableau 4.4). Pour les NaN, les résultats sont les mêmes que pour l'addition.

Gestion des drapeaux d'exception

Les drapeaux d'exception sont gérés de la même façon que pour la multiplication. Le drapeau division par 0 est positionné à 1 lorsque l'opérande B vaut 0.

	0	Nb	∞
0	NaNs	0	0
Nb	∞	Nb	0
∞	∞	∞	NaNs

TAB. 4.4 – Traitement des exceptions

4.5 Autres opérateurs flottants

4.5.1 Comparaison de deux nombres à virgule flottante

La comparaison de deux nombres à virgule flottante consiste à comparer les signes, exposants et mantisses des deux nombres et en fonction du résultat à positionner correctement les drapeaux indicateurs. La table de vérité des signaux `sup`, `inf` et `equ` en fonction des résultats des différentes comparaisons est données par le tableau 4.5.

Signes	Exposants	Mantisses	`sup`	`inf`	`equ`
$\neq$	X	X	$\overline{signeA}$	signeA	0
$=$	$>$	X	1	0	0
$=$	$<$	X	0	1	0
$=$	$=$	$>$	1	0	0
$=$	$=$	$<$	0	1	0
$=$	$=$	$=$	0	0	1

TAB. 4.5 – Comparaison de deux nombres à virgule flottante

Dans l'opération de comparaison, il faut également tenir compte du fait que +0 et -0 sont des nombres égaux, et que lorsqu'un des opérandes est un NaN le résultat est forcément $A > B$ (Sup vaut 1).
L'exception invalide est positionnée à 1, lorsqu'un des opérandes est un NaN.

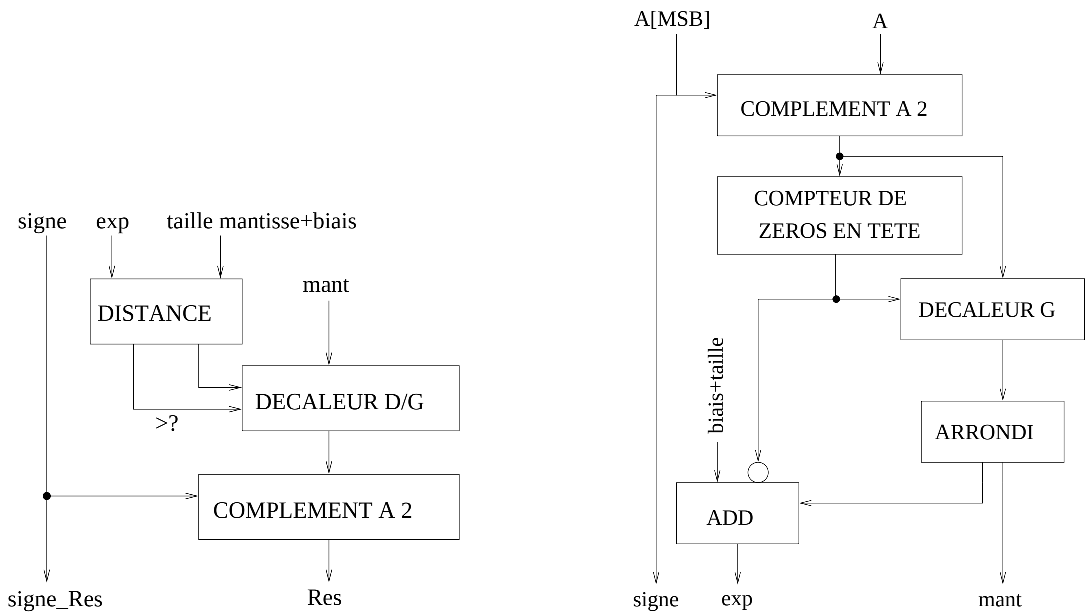
4.5.2 Conversion flottant $\rightarrow$ entier

La conversion d'un nombre à virgule flottante en entier s'effectue de la façon suivante :

1. Calcul du décalage qui vaut $|exposant - (taille\_mantisse + biais)|$

2. La mantisse est décalée de la valeur calculée à droite si ($exposant > taille\_mant + biais$), à gauche sinon
3. Si le nombre à virgule flottante est négatif, le complément à deux du nombre obtenu précédemment est effectué

L'exception inexacte vaut 1 lorsqu'un des bits évacués lors du décalage valait 1 et invalide est mis à 1 quand la valeur du décalage est négative. L'architecture de ce bloc est présentée figure 4.12(a).



(a) Flottant → entier (b) Entier → flottant

FIG. 4.12 – Conversion entier ↔ flottant

4.5.3 Conversion entier → flottant

La conversion d'un entier en nombre à virgule flottante se réalise de la façon suivante :

1. Complément : si l'entier est négatif, le complément à 2 est effectué
2. Normalisation : normalise l'entier (décalage à gauche du nombre de bit à zéros en poids forts) afin que son bit de poids fort vaille 1. La mantisse résultat est formée des bits de poids fort de l'entier normalisé

3. Arrondi : l'arrondi est calculé de la même manière que pour les autres opérateurs
4. Calcul de l'exposant : l'exposant valant $(biais + taille)$ est ajusté en fonction du nombre de décalages effectués lors de la normalisation

Il faut également traiter l'exception inexacte qui vaut 1 lorsque la mantisse a subit un incrément lors de l'arrondi.
L'architecture de ce bloc est présentée figure 4.12(b).

4.6 L'unité flottante standard

L'unité flottante peut intégrer une partie ou l'ensemble des opérateurs décrits précédemment. Pour sa conception, le but est de réutiliser au maximum le matériel entre les différents opérateurs en rajoutant du contrôle. Par exemple un seul décaleur sera utilisé pour effectuer la normalisation de chacun des opérateurs flottants. Seulement la partie de l'opération entière sur les mantisses est propre à chacun des opérateurs et ne pourra donc pas être réutilisée.

Nous allons maintenant décrire l'architecture de chacun des blocs nécessaires à la réalisation de notre unité flottante. Cette architecture dépend des opérations à effectuer. Pour cela, nous partons d'une unité flottante de base, réalisant l'addition, la comparaison et les conversions, à laquelle peuvent être rajoutés la multiplication et la division.

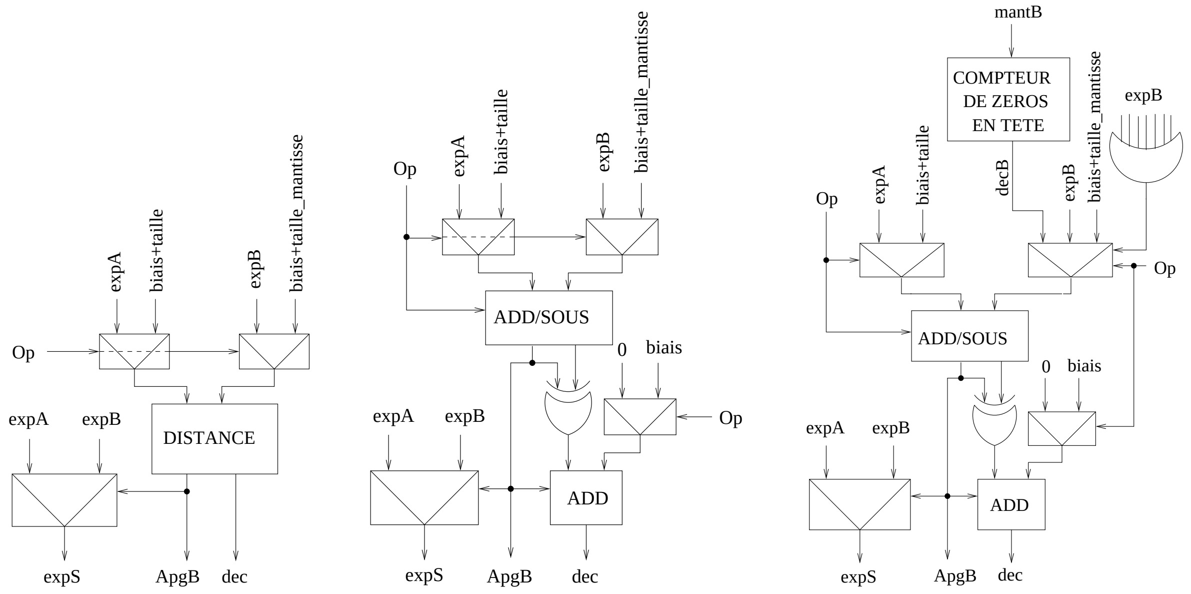
Calcul de l'exposant

Pour l'unité flottante de base, ce bloc est quasiment le même que pour l'additionneur, sauf qu'il faut ajouter des multiplexeurs à l'entrée de l'opérateur de distance afin de pouvoir effectuer le calcul de l'exposant dans le cas des conversions, soit $d = |exposant - (taille\_mantisse + biais)|$ ou $(d = biais + taille)$. Le signal `Op` est fonction de l'opération flottante à effectuer. Le signal `ApgB` indique que l'exposant de l'opérande A est supérieur ou égal à celui de l'opérande B et permet ainsi de choisir le plus grand exposant comme exposant résultat (`expS`) pour l'addition flottante(figure 4.13(a)).

Pour ajouter la multiplication, il faut complètement modifier l'architecture du bloc, car l'opération effectuée sur les exposants n'est plus une soustraction, mais une addition. Un premier additionneur va effectuer $(expA + expB)$ ou $(expA - expB)$ ou $(exp -$

(*taille_mant* + *biais*)), en fonction de l'opération (signal `Op`) puis un deuxième permettra de soustraire le biais au résultat ou de complémenter à 2 le résultat précédent afin d'obtenir la valeur absolue (figure 4.13(b)).

Et enfin pour la division, il faut rajouter le compteur de zéros en tête de mantisse qui permet de calculer le nombre de décalages nécessaire pour normaliser le diviseur. Le résultat est calculé en fonction de ce nombre de décalages (figure 4.13(c)).



(a) (b) (c)

FIG. 4.13 – Calcul de l'exposant

Mise en forme des mantisses

Ce bloc permet de mettre en forme les mantisses pour l'opération entière. Cela revient à rajouter les bits implicites. De plus pour certaines opérations (addition/soustraction et division), il faut également décaler les mantisses. Dans le cas de l'addition/soustraction, il s'agit de décaler à droite la mantisse la plus petite de la valeur calculée par le bloc de calcul de l'exposant (`dec`). Dans le cas de la division, la mantisse du diviseur dénormalisé est décalée à gauche du nombre de zéros en tête de mantisse (signal `decB` fournis par le bloc de calcul de l'exposant). L'architecture de ce bloc traitant à la fois l'addition/soustraction et la division est présentée figure 4.14. Lorsque l'unité flottante n'intègre pas la division, l'architecture reste la même que pour l'addition (figure 4.3).

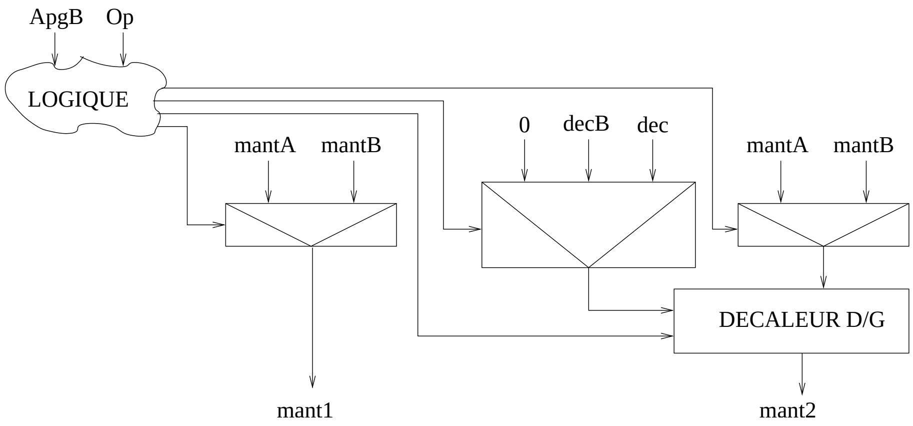


Fig. 4.14 – Mise en forme des mantisses

Opération entière

L'opération entière, correspondant à l'opération flottante souhaitée, est effectuée sur les mantisses. Pour cela, la multiplication et la division sont réalisées en parallèle. Pour l'addition/soustraction et la comparaison, au lieu d'utiliser un additionneur en parallèle du multiplieur et du diviseur, nous utilisons l'additionneur servant à effectuer la conversion du multiplieur de Wallace ou du diviseur *borrow save*. Comme en sortie de l'arbre de Wallace, le résultat de la multiplication de deux nombres sur p bits donne un résultat sur 2p bits, il faut étendre les résultats des autres opérations à 2p bits. Pour la division, le reste est mis en poids faibles afin de pouvoir effectuer la conversion, et pour les autres opérations, il s'agit d'une extension de signe. La figure 4.15(a) présente l'architecture de ce bloc dans une unité flottante réalisant l'addition/soustraction, la multiplication, la comparaison et les conversions. La division se rajoute comme indiqué par la figure 4.15(b).

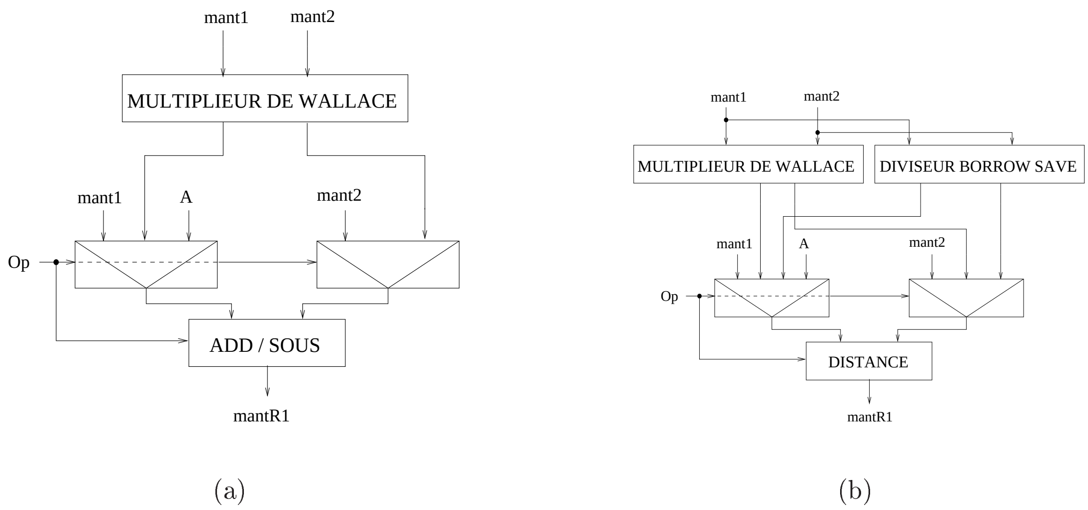


FIG. 4.15 – Opération entière

Normalisation

L'architecture de ce bloc est présentée figure 4.16. Une première étape consiste à chercher la position du premier bit à 1 de la mantisse sortant du bloc précédent. Ceci se fait avec un compteur de zéros en tête de mantisse. Ensuite un décaleur gauche/droite est utilisé pour normaliser la mantisse. Celle-ci est décalée à droite si l'exposant est négatif, ou s'il s'agit d'une conversion flottant → entier, et à gauche sinon. La valeur du décalage est la valeur absolue de l'exposant lors d'un décalage à droite et le minimal entre la position du premier bit à 1 calculée et la valeur de l'exposant lors d'un décalage à gauche.

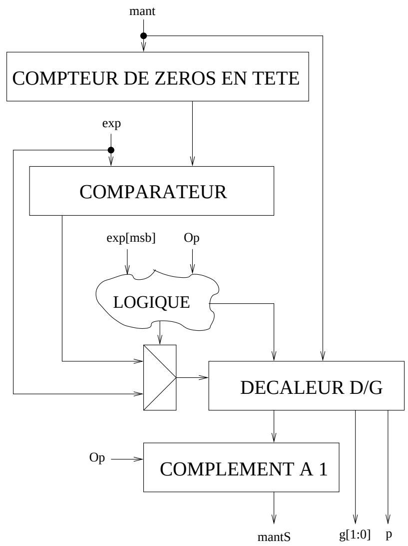


FIG. 4.16 – Normalisation

Ce bloc permet également de positionner le bit de garde qui servira pour le calcul de l'arrondi. Ce bit est le dernier qui n'est pas conservé lors d'un décalage à droite. Il y a également le bit persistant qui vaut 1 si lors d'une multiplication ou d'une division, un des bits abandonnés (bits de poids faibles pour la multiplication ou reste pour la division) vaut 1 ou si lors d'un décalage à droite, un des bits évacués valait 1. De plus, lors d'une conversion flottant →

entier, si le signe du nombre à virgule flottante est négatif, il faut effectuer le complément à un du résultat (l'ajout de 1 pour le complément à deux se fera dans le bloc arrondi).

Arrondi

Ce bloc réalise l'arrondi conformément à la norme IEEE-754. En fonction de l'arrondi, du bit de garde, du bit persistant et du signe du résultat, la mantisse est augmentée de 1 :
- arrondi vers 0 : pas d'incrément
- arrondi vers $-\infty$: incrément si $signe.(p+g)$
- arrondi vers $+\infty$: incrément si $\overline{signe}.(p+g)$
- arrondi au plus près : incrément si $g.(p+mant_0)$

De plus lors d'une conversion flottant → entier, il faut terminer le complément à deux, commencé dans le bloc normalisation, en faisant l'incrément de 1.
Enfin si arrondir provoque une retenue sortante le signal `coutr` est mis à 1, ce qui permettra d'incrémenter de 1 l'exposant lors de sa mise à jour.

Mise à jour de l'exposant

L'exposant résultat est incrémenté ou décrémenté en fonction des décalages effectués. Pour cela un additionneur/soustracteur entier est utilisé afin d'additionner (ou soustraire) à l'exposant la valeur du décalage.

Signe

Le calcul du signe pour l'addition a été vu figure 4.8. Pour la multiplication et la division, le signe du résultat est le ou-exclusif entre les signes des deux opérandes et le signe de l'opérande pour les conversions. Pour calculer le signe du résultat dans le cas d'une unité flottante complète, il suffit donc de rajouter un peu de logique au calcul du signe pour l'addition.

Bloc exception

Nous avons vu que le traitement des nombres spéciaux ne différait que pour les calculs avec des infinis. Il suffira donc de sélectionner les valeurs adéquates en fonction de l'opération effectuée. Ce bloc positionne également l'exception invalide lors d'une opération avec un NaNs ou deux infinis.

Gestion des drapeaux d'exception

Le drapeau d'exception inexacte vaut 1 lorsque le bit de garde ou le bit persistant vaut 1 ou lorsqu'il y a un *overflow*. Le drapeau invalide est positionné à 1 par le bloc exception dans le cas d'une opération invalide. L'*overflow* est mis à 1 lorsque le résultat dépasse le plus grand nombre à virgule flottante représentable. L'*underflow* signifie qu'un résultat dénormalisé est inexact. Et le signal division par 0 est mis à 1 lorsque le diviseur vaut 0.

4.7 Conclusion

Nous venons de présenter l'architecture des différents opérateurs flottants que sont l'addition/soustraction, la multiplication, la division, la comparaison et les conversions entier vers flottant et vice et versa. Ces différents opérateurs ont été intégrés au sein d'une unité de calcul flottant. Nous allons maintenant nous intéresser à la méthodologie de conception de ces différentes architectures ainsi qu'à leurs caractéristiques.

Chapitre

5 Méthodologie de conception et résultats

Sommaire

Pour concevoir les différentes parties de notre unité flottante stochastique, il a été choisi d'utiliser l'environnement de conception GenOptim [43][80]. Nous allons dans un premier temps détailler les fonctionnalités de GenOptim puis nous verrons comment nous les avons utilisées. Enfin nous présenterons les caractéristiques des différents composants que nous avons développés.

5.1 Fonctionnalités de GenOptim

GenOptim a été créé dans le but de faciliter le travail du concepteur de circuit numérique afin qu'il puisse s'affranchir des problèmes d'ordre technologique et se concentrer uniquement sur l'architecture du circuit. Pour cela GenOptim met à disposition du concepteur différentes fonctionnalités.

5.1.1 Cellules virtuelles

La notion de cellule virtuelle a été introduite afin de ne pas être dépendant des cellules d'une bibliothèque. Ainsi le concepteur dispose d'une bibliothèque de cellules virtuelles qu'il utilise pour concevoir ses générateurs. Ensuite lors de l'exécution du générateur cette bibliothèque virtuelle sera projetée sur la bibliothèque réelle.

Le concept de cellule virtuelle permet aussi de créer et d'utiliser des cellules qui pourraient ne pas être disponibles dans certaines bibliothèques réelles. Par exemple supposons que notre bibliothèque cible ne possède pas de multiplexeur. Il suffit de créer une cellule virtuelle "multiplexeur" à base de cellules virtuelle OU et ET et ainsi lors de la projection vers la bibliothèque réelle le multiplexeur sera construit à partir de cellules de base (Figure 5.1).

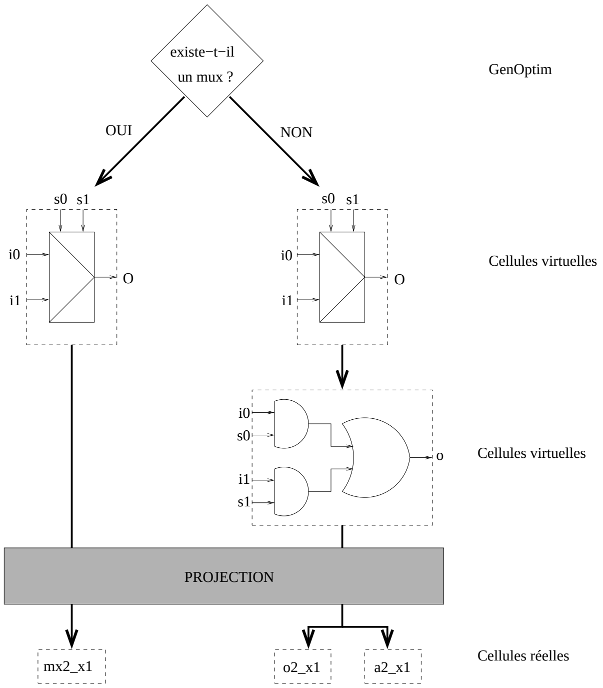


FIG. 5.1 – Multiplexeur virtuel

5.1.2 Portage vers les bibliothèques de cellules

Avec le concept de cellules virtuelles l'écriture du générateur ne nécessite pas de spécifier la bibliothèque de cellules cible. Les générateurs n'utilisent que des cellules virtuelles et ce n'est que lors de l'exécution du générateur que la bibliothèque cible est spécifiée. Ainsi il est possible à partir d'un même générateur d'obtenir le même circuit avec différentes bibliothèques de cellules sans avoir à réécrire le générateur. Cette fonctionnalité permet donc un portage des générateurs écrits vers différentes bibliothèques de cellules sans avoir à en modifier l'architecture interne. Cela permet donc aussi une migration technologique plus rapide.

5.1.3 Portage vers les technologies

La caractérisation d'une bibliothèque de cellules peut se faire pour différentes technologies. Ceci est particulièrement vrai pour les bibliothèques de cellules symboliques telles que celles développées au laboratoire LIP6/ASIM [36]. Pour générer le circuit dans une technologie cible, à chaque cellule est associée ses caractéristiques physiques telles que la surface active, le délai, le facteur de charge etc... Ces caractéristiques sont propres aux cellules et diffèrent selon la technologie cible. Ces informations sont utilisées par GenOptim pour l'optimisation électrique et l'analyse du chemin critique.

5.1.4 Optimisations électriques

Lorsque le générateur est utilisé, il est possible de demander une optimisation électrique afin d'optimiser une chaîne longue. Cette optimisation intervient à plusieurs niveaux :

Cellules amplifiées

Lorsque le générateur est exécuté la bibliothèque de cellules cible est spécifiée. Dans cette bibliothèque, plusieurs cellules réelles correspondent à une même cellule virtuelle, chacune ayant une puissance différente. Lors de la phase d'optimisation électrique, GenOptim va choisir la cellule la plus adaptée pour remplacer la cellule virtuelle.

Insertion d'amplificateurs

Plus un signal attaque de portes, plus son temps de propagation augmente. GenOptim permet d'insérer des amplificateurs sur les signaux attaquant un grand nombre de portes et ainsi de diminuer leurs temps de propagation. Les amplificateurs sont insérés soit en série, soit en parallèle.

Duplication de portes

Lorsqu'un signal en sortie d'une porte attaque un grand nombre de portes, il est possible de dupliquer la porte en amont afin de diminuer le temps de propagation du signal.

5.1.5 Analyse du chemin critique

GenOptim peut fournir une analyse détaillée du chemin critique du circuit généré. Cette analyse tient compte des différentes optimisations électriques éventuellement effectuées et même si ce n'est qu'une estimation de ce chemin, cela donne déjà une bonne idée des points faibles du circuit et permet de revenir à la conception de l'architecture sans passer par les phases de placement, routage et analyse temporelle.

5.1.6 Traducteurs

La dernière fonctionnalité qu'apporte GenOptim est l'existence de nombreux traducteurs qui sont directement intégrer au générateur. Ainsi en changeant uniquement l'option correspondant au format de sortie du générateur, il est possible de générer une vue structurelle du circuit utilisable directement avec Alliance [35], Compass, Cadence, Xilinx...

5.2 Conception dans l'environnement GenOptim

5.2.1 Les fichiers générés

Les générateurs dans l'environnement GenOptim s'écrivent en langage C++. Un générateur permet de créer différents fichiers :
- Une vue comportementale qui décrit le circuit
- Une vue structurelle qui est l'ensemble des cellules de la bibliothèque utilisées et leurs interconnexions
- Un ensemble de vecteurs de test pour la simulation du circuit
- Un rapport détaillé du circuit

5.2.2 Les paramètres d'entrée

Un générateur dispose également d'un ensemble de paramètres en entrée. Ces paramètres sont de deux sortes. Dans un premier temps, les paramètres propres au générateur qui permettent d'orienter la conception du circuit. Cela peut être par exemple la taille des données, l'algorithme utilisé etc... Viennent ensuite les paramètres de GenOptim qui donnent :
- le format de sortie de la vue générée : Alliance, Vhdl, Cadence, Xilinx...
- la bibliothèque de cellules cible avec ses caractéristiques technologiques
- le fichier de construction des cellules virtuelles indisponibles dans la bibliothèque de cellules cible
- le niveau de mise à plat de la hiérarchie du circuit généré
- s'il faut créer ou non le rapport détaillé du circuit
- le délai RC maximal autorisé lors de l'optimisation électrique
- s'il faut faire ou non l'optimisation électrique
- le format du fichier de vecteurs de test : Alliance, Vhdl, Verilog...
- s'il faut générer ou non la vue comportementale si elle existe

– s'il faut supprimer ou non les signaux et instances de cellules non utilisés.

5.2.3 Fonctions de conception

L'environnement GenOptim met à disposition du concepteur un ensemble de macro-fonctions permettant l'établissement des différents fichiers de sortie du générateur :

- Des fonctions d'ouverture et de fermeture des différentes vues
- Des fonctions de déclaration de signaux, de connecteurs
- Des fonctions d'utilisation des cellules virtuelles, des générateurs
- Des fonctions de manipulation de signaux : concaténation, changement de nom...
- Des fonctions de description du comportement
- Des fonctions d'affectation d'entrées/sorties pour la génération des vecteurs de test.

5.2.4 Bibliothèque de générateurs

De plus GenOptim contient une bibliothèque de générateurs mis à disposition du concepteur. Ces générateurs sont aussi bien logiques (opérateurs booléens, multiplexeurs, compteur de zéros en tête, décodeur d'adresse...) qu'arithmétiques entiers (additionneurs, multiplieurs, diviseur...). Ils prennent en général en paramètre la taille de leurs entrées/sorties, le type d'opérateur booléen (or, and...), le type d'algorithme utilisé (par exemple pour un additionneur à anticipation de retenue ou à retenue conservée...), etc...

5.2.5 Flot de conception

La figure 5.2 présente le flot de conception d'un générateur dans l'environnement GenOptim. Le concepteur doit écrire le modèle du générateur. Ce modèle est compilé avec GenOptim pour donner le générateur à proprement parlé. C'est ensuite l'exécution de ce générateur avec les différents paramètres qui va permettre de générer les différents fichiers.

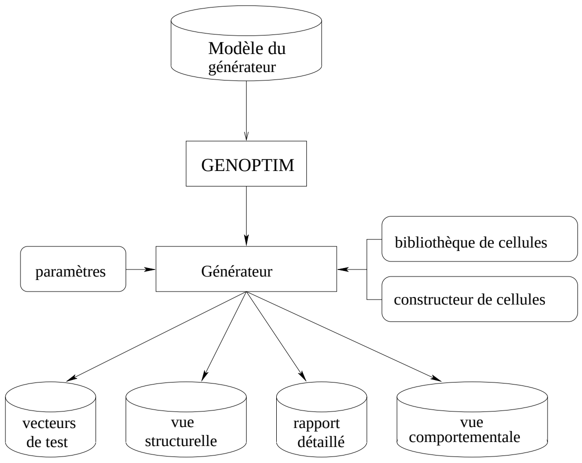


FIG. 5.2 – Conception dans l'environnement GenOptim

5.3 Conception de l'unité flottante stochastique

5.3.1 Les différents générateurs

Nous allons construire un générateur pour chacun des blocs composant l'unité flottante stochastique. Ces générateurs prennent comme paramètre la taille de l'exposant et de la mantisse afin de pouvoir générer une unité flottante aussi bien 32 bits (simple précision) que 64 bits (double précision) voire toute autre combinaison (précision étendue). Chacun de ces générateurs est validé avec son propre jeu de vecteurs de test.

Une fois les différents générateurs de bloc élaborés et validés, nous les utilisons dans le générateur de l'unité flottante stochastique. Ce générateur prend comme paramètres :

- la taille de l'exposant et de la mantisse afin de pouvoir traiter différentes tailles de données
- le nombres d'étages de pipeline souhaités
- certains choix architecturaux peuvent être faits pour améliorer la chaîne longue comme par exemple la détection du premier bit à 1 en parallèle avec l'opération entière sur les mantisses (voir annexe D)
- les opérateurs à intégrer à l'unité, car selon les applications il est possible de n'avoir besoin que d'un additionneur et d'un multiplieur par exemple
- notre unité flottante peut intégrer la gestion du contrôle des arrondis de calcul par l'ajout du bloc CESTAC

5.3.2 Environnement de validation

La validation de notre unité flottante stochastique concerne essentiellement la validation de l'unité flottante. En effet l'implantation matérielle de la méthode CESTAC s'appuyant directement sur le modèle C qui a servi à la validation de notre méthode de calcul, à partir du moment où les opérations effectuées par l'unité flottante seront correctes, l'unité stochastique sera elle aussi validée.
Pour valider notre unité flottante (non stochastique), nous avons utilisé le simulateur VHDL issu de la chaîne de CAO Alliance[35][36]. Ce simulateur prend en entrée une description VHDL comportementale ou structurelle du circuit et un jeu de vecteurs de test. La description VHDL de notre circuit est fournie en utilisant notre générateur d'unité flottante et il faut maintenant fournir un "bon" jeu de vecteurs de test.

La génération du jeu de vecteurs de test

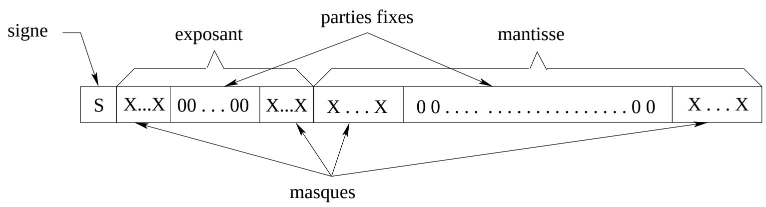


FIG. 5.3 – Vecteur de test

Il est bien sûr impensable de générer un jeu de vecteurs de test exhaustifs pour notre unité flottante, mais il faut néanmoins que les vecteurs soient représentatifs pour s'assurer de sa validité. En général les cas litigieux se retrouvent pour les valeurs extrêmes de l'exposant et/ou de la mantisse (tous les bits valent 1 ou 0) puisqu'ils regroupent à la fois les nombres normalisés, dénormalisés, les infinis, les NaNs et les zéros. Nous avons donc mis en place un système qui permet de traiter tous ces cas. L'exposant et la mantisse sont décomposés chacun en trois parties : une partie fixe dont tous les bits sont à 0 (ou 1) et deux masques dont la valeur varie (figure 5.3). Les valeurs des masques prennent toutes les valeurs entre 0...0 et 1...1.
L'algorithme de génération des vecteurs de test consiste donc à :
- faire varier les bits masqués des deux opérandes
- alterner les parties fixes entre 0...0 et 1...1
- alterner le signe entre 0 et 1

Toutes les combinaisons possibles entre les valeurs des masques, du signe et des parties fixes sont générées et ce pour tous les opérandes de l'opération flottante à tester. Un jeu de vecteurs de test permet de tester toutes ces combinaisons pour une opération flottante et un mode d'arrondi donné. Pour chacun des vecteurs, l'opération est effectuée sur un processeur SPARC respectant la norme IEEE-754 et la simulation de notre unité flottante doit donner le même résultat.

Ce générateur de vecteurs de test prend comme paramètres l'opération flottante à tester, le mode d'arrondi, la taille de l'exposant et de la mantisse, la taille des masques de l'exposant et de la mantisse et le nombre d'étages de pipeline. Il est donc possible grâce à ce générateur de valider dans de bonnes conditions notre unité flottante.

5.4 Caractéristiques des différents composants

Les différents composants de l'unité flottante stochastique ont été générés dans l'environnement GenOptim. La bibliothèque de cellules cibles est Sxlib développée par le département ASIM du LIP6 [36]. Les descriptions structurelles ainsi obtenues ont ensuite été validées, puis placées et routées avec Cadence/Silicon Ensemble. L'analyse des chemins critiques a été effectuée avec Avertec/TAS (figure 5.4).

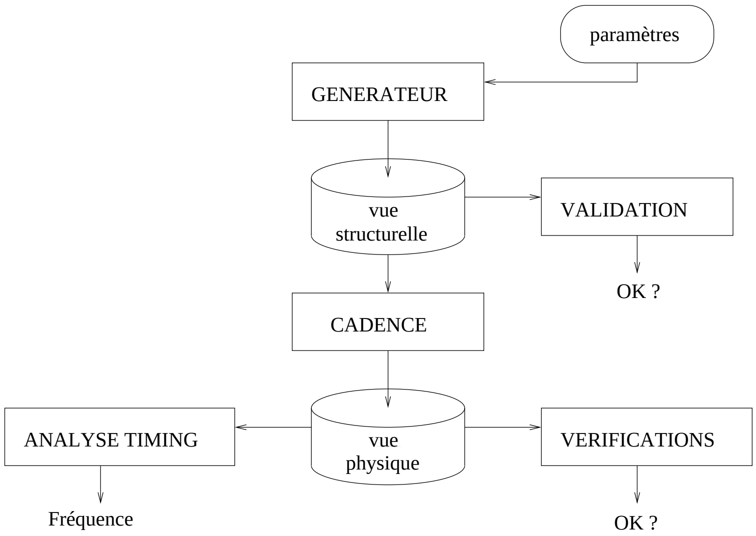


FIG. 5.4 – Flot de conception

Opérateurs	Étages de pipeline	Simple précision					Double précision				
		Nombre de transistors	$0.35\mu m$		$0.25\mu m$		Nombre de transistors	$0.35\mu m$		$0.25\mu m$	
			Surface (mm^2)	Temps (ns)	Surface (mm^2)	Temps (ns)		Surface (mm^2)	Temps (ns)	Surface (mm^2)	Temps (ns)
addv1	1	16149	0.48	35	0.25	8.1	34202	1.02	46	0.52	10
	2	17747	0.52	20	0.27	4.7	36952	1.09	27	0.56	6.6
	3	19709	0.57	13	0.29	3.2	40174	1.17	18	0.6	4.5
addv2	1	19876	0.59	33	0.3	7.6	42154	1.24	42	0.63	10
	2	23850	0.69	18	0.35	4.2	49476	1.42	24	0.72	5.9
	3	28932	0.82	13	0.42	3.1	59058	1.67	18	0.85	4.5
multiplication	1	34368	0.88	36	0.45	7.9	128175	3.21	47	1.64	11
	2	38104	0.97	19	0.49	4.0	134539	3.38	24	1.72	5.7
	3	41242	1.04	15	0.53	3.5	140649	3.52	22	1.8	5.3
division	1	61859	1.84	94	0.94	21	220963	6.59	213	3.36	49
	2	65797	1.94	48	0.99	11	233087	6.89	118	3.52	27
	3	71333	2.08	34	1.06	7.6	241381	7.11	86	3.63	20
comparaison		2548	0.07	6	0.04	1.4	5312	0.15	7.1	0.08	1.6
conversion flottant → entier		5847	0.14	13	0.07	3.2	12476	0.3	17	0.15	4.2
conversion entier → flottant		5727	0.17	15	0.09	3.4	12641	0.38	19	0.19	4.5

TAB. 5.1 – Caractéristiques des opérateurs flottants

5.4.1 Les opérateurs flottants

Le tableau 5.1 présente les caractéristiques en terme de nombre de transistors, de surface et de temps de propagation des différents opérateurs flottants décrits dans le chapitre 4 en format simple et double précision et selon le nombre d'étages de pipeline et la technologie. Dans ce tableau, addv1 est l'additionneur flottant présenté en 4.2.2 et addv2 l'additionneur flottant découpé en deux chemins parallèles vu en 4.2.3. Les opérateurs de comparaison et de conversion étant assez simples, ils seront purement combinatoires (sans pipeline). En effet leur temps de propagation sans étages de pipeline est largement inférieur à celui des autres opérateurs avec trois étages de pipeline.

Synthèse

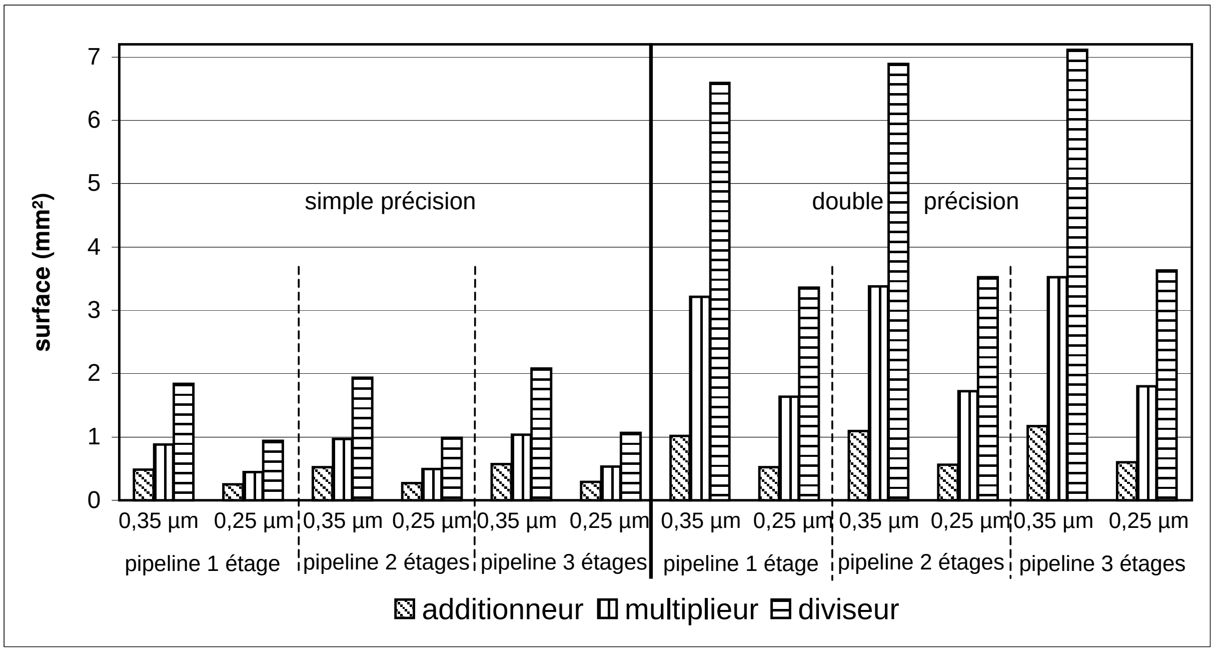
Aux vues des caractéristiques entre l'additionneur de base et celui découpé en deux chemins parallèles, il ne semble pas très intéressant d'implanter l'additionneur découpé. En effet, il ne présente qu'un gain compris entre 0 et 3 ns par rapport à l'additionneur de base, alors que sa surface est nettement plus importante (entre 20 et 40 %). De plus il sera plus difficile d'intégrer ce découpage au sein d'une même unité flottante pour la comparaison, les conversions, la multiplication et la division.

La figure 5.5 présente un comparatif des surfaces (5.5(a)) et des temps de propagation (5.5(b)) de l'additionneur, du multiplieur et du diviseur.

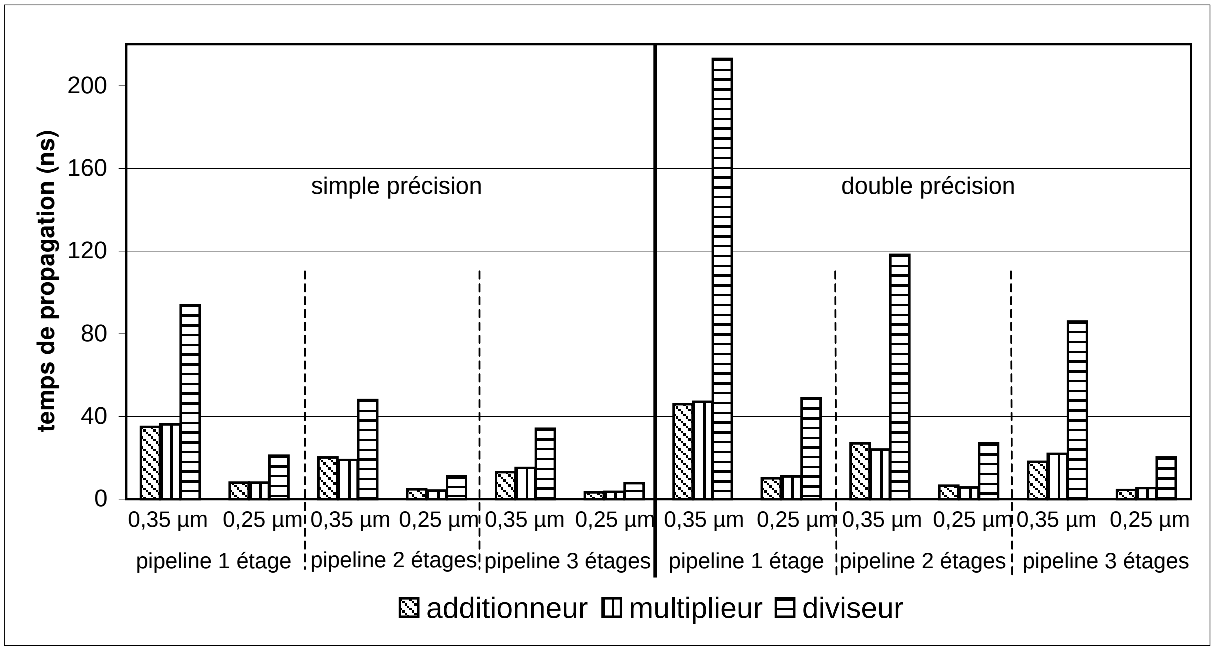
Il semble tout à fait judicieux d'intégrer l'addition/soustraction, la multiplication, la comparaison et les conversions au sein d'une même unité flottante. En effet leurs temps de propagations étant sensiblement identiques, et ce quel que soit le nombre d'étage de pipeline ou la technologie, il ne sera pas pénalisant pour les performances de l'unité flottante de tous les intégrer.

Par contre pour la division, vu que son temps de propagation est nettement supérieur à celui des autres opérateurs, plusieurs stratégies sont possibles :

- Soit l'intégrer dans l'unité flottante avec le même nombre d'étages de pipeline que les autres opérateurs et dans ce cas les performances seront dégradées puisque ce seront celles du diviseur.



(a) En surface



(b) En temps de propagation

FIG. 5.5 – Comparatif des opérateurs

– Soit s'arranger pour que son délai soit le même que les autres opérateurs en lui permettant un nombre plus important d'étages de pipeline.
– Soit de ne pas l'implanter et de la réaliser de manière logicielle.

La deuxième stratégie est plus difficile à mettre en œuvre, car elle nécessite d'avoir des opérations flottantes qui n'ont pas toutes le même nombre de cycles et cela pose des problèmes de synchronisation. En effet dans ce cas une division commencée avant une addition peut se terminer après ce qui cause des problèmes de dépendances sur les données.

Il a donc été décidé de réaliser une unité flottante dont tous les opérateurs ont le même nombre d'étages de pipeline et dont les caractéristiques vont être présentées dans la section suivante.

5.4.2 L'unité flottante standard

Le tableau 5.2 présente les caractéristiques en terme de nombre de transistors, de surface et de temps de propagation des différentes unités flottantes décrites dans le chapitre 4 en format simple et double précision et selon le nombre d'étages de pipeline et la technologie. Nous présentons tout d'abord les caractéristiques de l'unité flottante incluant les opérations d'addition/soustraction, comparaison et conversions. Puis nous y ajoutons la multiplication et ensuite la division.

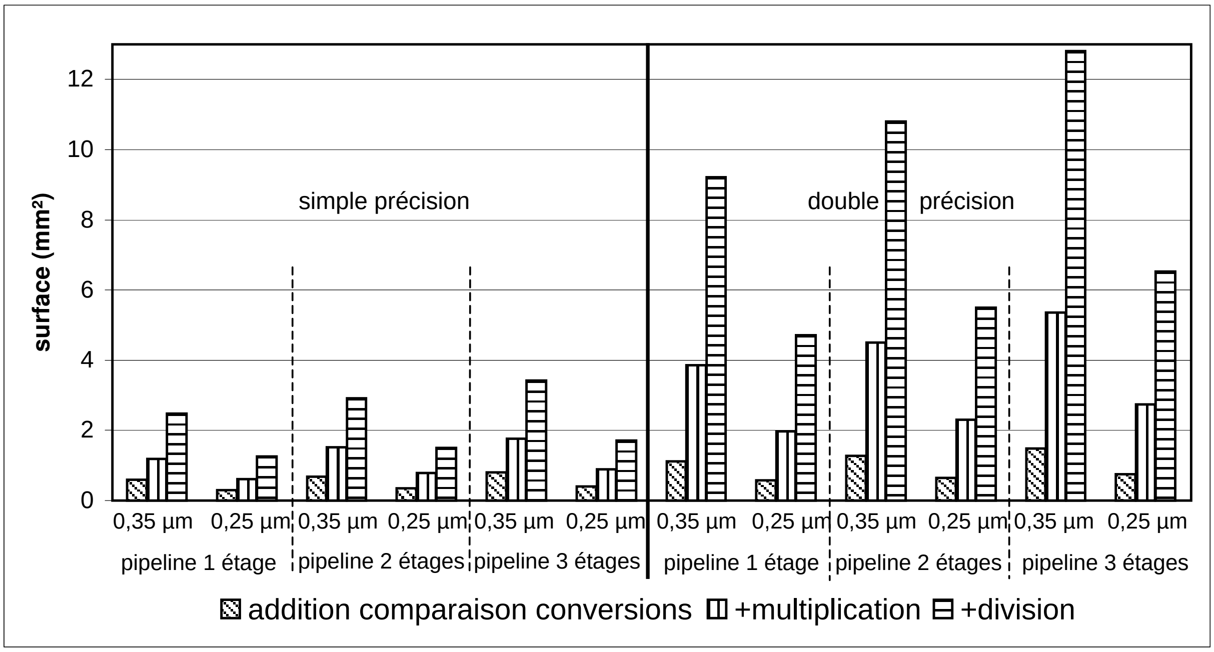
Synthèse

La figure 5.6 présente un comparatif des surfaces (5.6(a)) et temps de propagation (5.6(b)) des différentes unités flottantes.

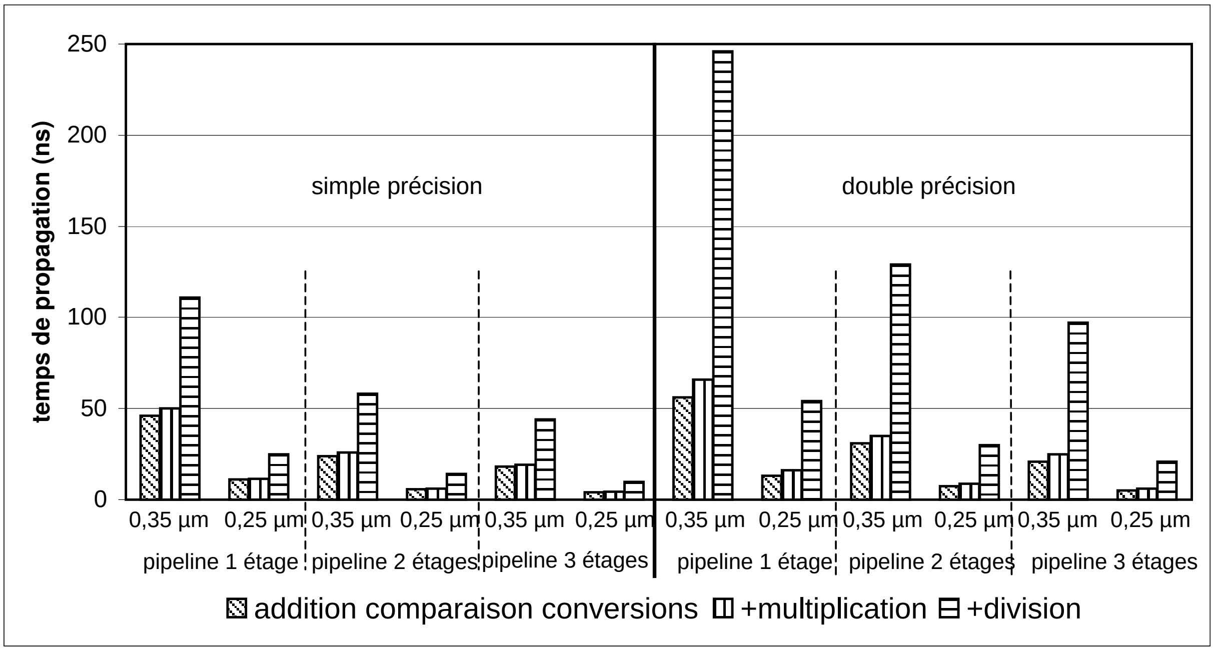
L'ajout du multiplieur au sein de l'unité flottante ne réalisant que l'addition, la comparaison et les conversions, représente une augmentation de sa surface d'environ 98% et du temps de propagation de 13%. Par contre l'ajout du diviseur implique un surcoût en surface d'environ 74% et 163% en temps de propagation par rapport à l'unité flottante comprenant l'addition, la comparaison, les conversions et la multiplication.

Opérateurs inclus	Étages de pipeline	Simple précision					Double précision				
		Nombre de transistors	0.35μm		0.25μm		Nombre de transistors	0.35μm		0.25μm	
			Surface (mm^2)	Temps (ns)	Surface (mm^2)	Temps (ns)		Surface (mm^2)	Temps (ns)	Surface (mm^2)	Temps (ns)
addition comparaison conversions	1	20756	0.59	46	0.3	11	40502	1.12	56	0.57	13
	2	24248	0.68	24	0.35	5.8	46396	1.27	31	0.65	7.5
	3	28798	0.8	18	0.4	4.0	54510	1.48	21	0.75	5.1
+multiplication	1	46052	1.19	50	0.61	11.4	152300	3.85	66	1.97	16
	2	53834	1.51	26	0.78	6.1	178154	4.5	35	2.3	8.6
	3	63916	1.77	19	0.89	4.4	211522	5.35	25	2.74	6.2
+division	1	88858	2.48	111	1.26	25	330780	9.22	246	4.71	54
	2	103782	2.91	58	1.5	14	386428	10.8	129	5.5	30
	3	123286	3.42	44	1.71	9.7	458940	12.8	97	6.53	21

TAB. 5.2 – Caractéristiques de l'unité flottante standard



(a) En surface



(b) En temps de propagation

FIG. 5.6 – Comparatif des unités flottantes

5.4.3 L'unité flottante stochastique

Le tableau 5.3 présente les caractéristiques en terme de nombre de transistors, de surface et de temps de propagation du bloc CESTAC en format simple et double précision et selon la technologie.

Techno (μm)	Précision	Transistors	Surface (mm^2)	Temps(ns)
0.35	simple	26430	0.68	8.3
0.25	simple	26430	0.35	1.9
0.35	double	53832	1.38	11
0.25	double	53832	0.7	2.6

TAB. 5.3 – Caractéristiques du bloc CESTAC

Synthèse

La gestion de l'arithmétique stochastique discrète en matériel n'est pas pénalisante en terme de temps de propagation puisque les temps de propagation de l'unité flottante classique et stochastique sont les mêmes. En effet, le bloc CESTAC ayant un temps de propagation nettement inférieur à celui de l'unité flottante, il n'est pas pris en compte pour l'évaluation du temps de propagation total de l'unité flottante stochastique. Par contre, l'ajout du bloc CESTAC entraîne un surcoût en terme de surface d'environ 112% pour l'unité flottante comprenant l'addition, la comparaison et les conversions, 49% lorsque la multiplication est ajoutée et de 25% avec en plus la division.

Un dessin des masques de l'unité flottante stochastique en simple précision est donné page 84.

5.5 Conclusion

Nous venons dans ce chapitre de présenter notre méthodologie de conception qui se base sur l'élaboration de générateurs dans l'environnement GenOptim, générateurs permettant d'obtenir des vues structurelles qui seront ensuite simulées avec les outils de la chaîne Alliance pour permettre leur validation. Cela a permis de générer des unités flottantes stochastiques qui ont pu être placées, routées et analysées.

Maintenant il nous reste à intégrer cette unité dans un système plus complexe afin de pouvoir exécuter des programmes de calcul dans lesquels le contrôle et l'estimation de la précision pourront être utilisés.


Bloc
CESTAC
Surface : 0.35 mm²
Fréquence : 526 MHz
Techno : 0.25 µm
Unité
flottante
IEEE-754
Surface : 1.26 mm²
Fréquence : 83 MHz
Pipeline : 3 étages
Techno : 0.25 µm
Opérations : addition,
comparaison,
conversions,
multiplication,
division

Chapitre

6

Intégration système

Sommaire

Maintenant que nous avons une unité flottante mettant en œuvre l'arithmétique stochastique discrète, il va falloir l'intégrer dans un système complet afin de pouvoir l'utiliser pour faire tourner des programmes de calcul. Pour cela nous avons choisi de l'intégrer en tant que coprocesseur dans un système sur puce s'articulant autours d'un bus.

6.1 Architecture générale

L'architecture générale du système s'articule autours d'un PI-Bus (*Peripheral Interconnect Bus*) qui a été spécialement conçu pour les systèmes intégrés sur puce [61]. Les différents composants (processeurs, mémoires, coprocesseur, unités de contrôle...) communiquent les uns avec les autres par l'intermédiaire de *wrappers* et selon le protocole VCI (*Virtual Component Interface*) [87]. VCI a été créé dans un souci d'harmoniser les interfaces des composants pouvant être connectés sur un système sur puce. Les *wrappers* servent à faire la traduction entre le protocole VCI et le protocole du bus système (le Pi-Bus dans notre cas). Un *wrapper* peut être maître et/ou esclave selon que le composant qu'il interface est un composant maître ou esclave. L'architecture générale du système pour notre application est présentée par la figure 6.1.

Les différents composants de ce système sont :

- Un processeur RISC, le MIPS R3000, sur lequel seront exécutés les programmes. Il possède ses propres caches de données et d'instructions. Les communications entre le Mips et ses caches se font selon le protocole VCI. Les caches sont connectés à des *wrappers* maîtres permettant de convertir le protocole VCI en un protocole de bus (ici celui du Pi-Bus).
- Une mémoire qui va contenir les séquences d'instructions à exécuter lors du démarrage du système ou du déclenchement d'une exception. La mémoire contient également le programme à exécuter et l'ensemble des données qu'il manipule.
- Un écran permettant l'affichage des résultats.

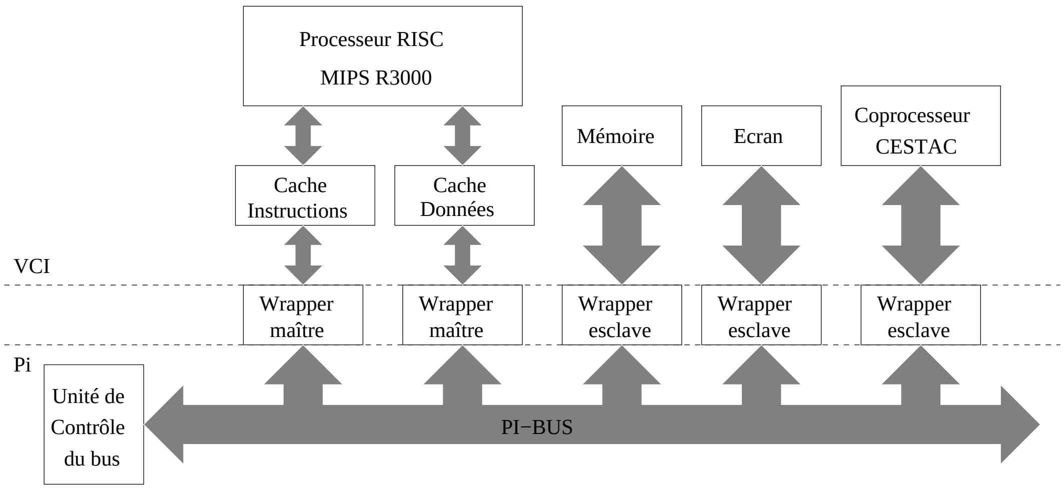


FIG. 6.1 – Système sur puce

– Un coprocesseur CESTAC, intégrant notre unité flottante stochastique, qui va permettre l'exécution des opérations flottantes du programme en utilisant soit l'arithmétique flottante IEEE-754 (mode standard), soit l'arithmétique stochastique discrète (mode CESTAC).

L'ensemble de ces composants, à part le coprocesseur CESTAC, fait partie de la bibliothèque disponible au laboratoire LIP6/ASIM. Le Mips R3000 utilisé dans ce système est fort ancien mais très utilisé au laboratoire LIP6/ASIM [62][30]. Par contre, il n'inclut pas les opérations virgule flottante et c'est pourquoi, nous avons décidé de les intégrer dans un coprocesseur. Nous allons maintenant nous intéresser au coprocesseur CESTAC.

6.2 Le coprocesseur CESTAC

L'architecture interne du coprocesseur CESTAC est donnée par la figure 6.2. Il est composé de trois automates communiquants :

– Un automate VCI permettant de traiter les requêtes VCI destinées au coprocesseur.
– Un automate FIFO servant à mettre les résultats de l'unité flottante dans une FIFO.
– Un automate FPU contrôlant l'unité flottante stochastique décrite dans les chapitres précédents.

Nous allons maintenant étudier comment notre coprocesseur communique avec le système complet au moyen du protocole VCI, ainsi que l'implantation de l'unité flottante stochastique dans ce système.

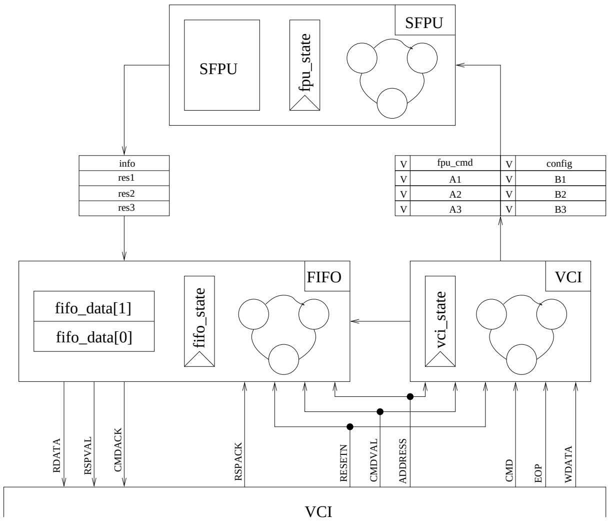
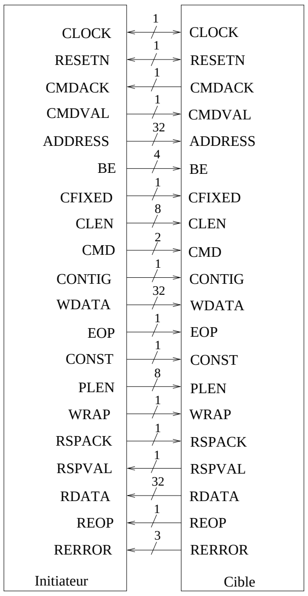


Fig. 6.2 – Le coprocesseur CESTAC

6.2.1 L'interface

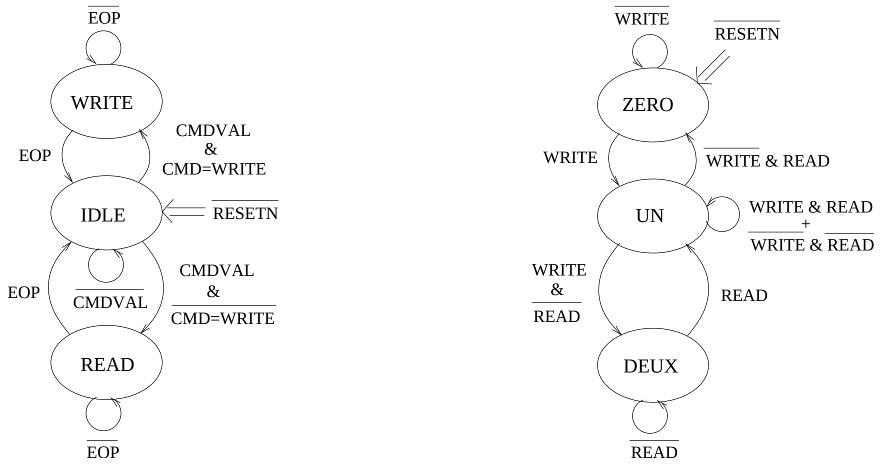
Les connexions VCI se font d'un initiateur vers une cible. Dans notre système l'initiateur est le processeur Mips et la cible notre coprocesseur. La figure 6.3 décrit l'interface VCI entre un initiateur et une cible et donne une brève description des signaux utilisés dans notre architecture.



- CLOCK : horloge du système
- RESETN : redémarrage du système (actif à l'état bas)
- CMDACK : accusé de réception d'une requête par la cible
- CMDVAL : actif lorsque la commande est valide
- ADDRESS : adresse de la donnée à lire ou à écrire
- CMD : type de commande (lecture ou écriture)
- WDATA : donnée émise par l'initiateur
- EOP : fin de paquet des données émises
- RSPACK : accusé de réception d'une requête par l'initiateur
- RSPVAL : la réponse est valide
- RDATA : donnée réponse
- REOP : fin de paquet de la réponse
- RERROR : type d'erreur dans la réponse

Fig. 6.3 – Interface entre l'initiateur et la cible

6.2.2 Le protocole VCI



(a) VCI (b) FIFO

FIG. 6.4 – Les automates de communication

Le mode écriture

Le coprocesseur CESTAC comprend 8 registres pouvant être adressés en écriture par l'automate VCI (figures 6.2 et 6.4(a)) :

- un registre de commande (`fpu_cmd`) qui contient le code de l'opération à effectuer par l'unité flottante CESTAC
- un registre de configuration (`config`) qui permet de positionner le mode de fonctionnement (CESTAC ou normal), le mode d'arrondi (lorsque le mode CESTAC est désactivé)
- les six registres contenant les trois composantes des deux opérandes stochastiques (A1, A2, A3, B1, B2, B3)

Ces différents registres possèdent un bit de validité qui est mis à 1 lorsqu'une donnée est écrite et remis à 0 lorsque cette donnée a été consommée par l'unité flottante.

L'écriture dans un registre se fait par une transaction VCI d'écriture. Le signal ADDRESS contient l'adresse du registre à écrire et le signal WDATA la donnée à y écrire.
Un automate VCI (figure 6.4(a)) permet d'écrire dans ces registres en fonction des signaux VCI. Lorsqu'une commande VCI d'écriture est reçue par l'automate (CMDVAL=1

et CMD=WRITE) celui-ci passe dans l'état d'écriture où la donnée présente sur le signal WDATA est écrite dans le registre adressé par le signal ADDRESS.

Le mode lecture

Le coprocesseur comprend également 4 registres pouvant être adressés en lecture par l'automate FIFO (figures 6.2 et 6.4(b)) :

- un registre d'information (figure 6.5) qui contient des informations sur l'opération qui vient de s'exécuter dans l'unité flottante, à savoir le nombre de bits significatifs du résultat (NBSR), si c'est un zéro informatique (R0), les exceptions qui ont été déclenchées (EXC), les drapeaux de comparaison des nombres stochastiques (inf et sup)
- les trois registres contenant les trois composantes du résultat stochastique (res1, res2, res3)

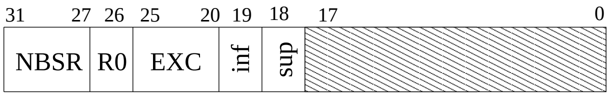


FIG. 6.5 – Le registre INFO

Ces registres sont écrits par l'unité flottante et lu lorsque l'automate VCI est dans l'état de lecture. L'automate passe dans l'état lecture lorsqu'il reçoit une commande VCI de lecture (CMDVAL=1 et CMD≠WRITE).

Les résultats sont écrits dans une FIFO contrôlée par un automate FIFO (figure 6.4(b)) afin d'éviter les signaux de Mealy. Une donnée est écrite dans la FIFO chaque fois qu'il y a une requête de lecture, c'est-à-dire lorsque l'automate VCI est dans l'état READ. Une consommation dans la FIFO a lieu dès lors que l'initiateur est prêt à recevoir une donnée (RSPACK=1). La FIFO a une profondeur de deux cases et donc l'automate qui la contrôle possède trois états :

- ZERO : s'il y a un ordre d'écriture, l'automate passe dans l'état UN et écrit la donnée adressée par le signal ADDRESS dans la première case de la FIFO
- UN :
  - s'il y a un ordre d'écriture, l'automate passe dans l'état DEUX et écrit la donnée adressée par le signal ADDRESS dans la deuxième case de la FIFO

  – s'il y a un ordre de lecture, l'automate repasse dans l'état ZERO
- DEUX : lorsqu'il y a un ordre de lecture, l'automate passe dans l'état UN, et la donnée présente dans la deuxième case de la FIFO passe dans la première

Les signaux générés

Certains signaux VCI doivent être positionnés selon l'état dans lequel se trouve le coprocesseur :
- le signal d'acceptation d'une requête VCI (CMDACK) est activé dès que l'automate VCI n'est pas en attente
- le signal de réponse valide (RSPVAL) est activé lorsque l'automate VCI est dans l'état d'écriture ou lorsque la FIFO des réponses n'est pas vide.
- le signal pour la donnée réponse (RDATA) prend la valeur de la première case de la FIFO.
- les réponses ayant la taille d'un mot de 32 bits, le signal fin de paquet réponse (REOP) est activé dès lors qu'une réponse est émise, c'est-à-dire lorsque l'automate VCI est en mode lecture et que la FIFO des réponses n'est pas vide
- le signal d'erreur (RERROR) est nul tant qu'il n'y a pas d'erreur dans la réponse

Le tableau 6.1 résume les valeurs des signaux CMDACK, RSPVAL et REOP en fonction des états des automates.

	ZERO	UN	DEUX
IDLE	0	0	0
WRITE	1	1	1
READ	1	1	1

(a) signal CMDACK

	ZERO	UN	DEUX
IDLE	0	1	1
WRITE	1	1	1
READ	0	1	1

(b) signal RSPVAL

	ZERO	UN	DEUX
IDLE	0	0	0
WRITE	0	0	0
READ	0	1	1

(c) signal REOP

TAB. 6.1 – Les signaux générés

6.2.3 L'unité flottante stochastique et son automate

L'unité flottante stochastique présentée dans les chapitres précédents est utilisée en simple précision et avec trois étages de pipeline. Un automate (FPU) va permettre d'orienter correctement les entrées et sorties de l'unité, en fonction du mode (standard ou CESTAC) et de l'état d'avancement du calcul. Cet automate est présenté figure 6.6

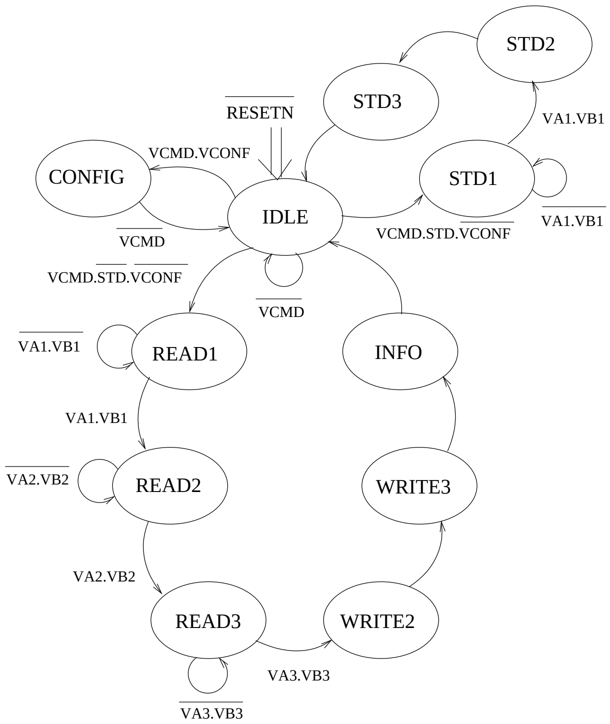


Fig. 6.6 – Automate de l'unité flottante

L'automate possède trois branches, une pour chacun des modes de fonctionnement. L'accès à une branche se fait à partir du moment où une commande est reçue, c'est-à-dire que le bit de validité du registre de commande (VCMD[1]) est à 1. Ensuite le choix de la branche se fait en fonction du mode (STD) ou du bit de validité du registre de configuration (VCONF).

Mode configuration

Si le bit de validité du registre de configuration (VCONF) est activé, alors l'opération demandée est la configuration de l'unité flottante. Cette opération permet de positionner le mode de fonctionnement (standard ou CESTAC) et le mode d'arrondi dans le cas ou le mode est standard. Une fois l'unité flottante configurée, l'automate repasse dans l'état d'attente.

[1]le bit de validité du registre X est noté VX

Mode standard

En mode standard, il faut trois cycles pour effectuer une opération flottante (l'unité flottante a trois étages de pipeline). Au premier cycle (STD1) les deux opérandes sont attendus (VA1=0 ou VB1=0) et une fois obtenues, le calcul commence et l'automate passe dans l'état STD2, puis STD3 et enfin en état d'attente. A la fin du calcul (STD3) le résultat est écrit dans le registre RES1 accessible en lecture par l'automate FIFO. Les éventuelles exceptions détectées sont écrites dans le registre INFO également accessible en lecture par l'automate FIFO.

Mode CESTAC

En mode CESTAC le fonctionnement est différent car une opération se fait sur les trois composantes du nombre stochastique. Le premier état (READ1) permet d'obtenir la première composante des deux opérandes stochastiques. L'automate ne passe dans l'état suivant (READ2) que lorsque les deux premières composantes des opérandes sont disponibles (VA1=1 et VB1=1). Puis l'état READ2 fait la même chose pour la deuxième composante, et l'état READ3 pour la troisième. A ce moment le résultat de l'opération sur les premières composantes est disponible et écrit dans le registre RES1. Puis dans l'état WRITE2 c'est au tour du résultat sur les deuxièmes composantes d'être écrit dans le registre RES2. Et enfin dans l'état WRITE3, le troisième résultat est écrit dans le registre RES3. Enfin les informations obtenues par le bloc CESTAC sont écrites dans le registre INFO lorsque l'automate est dans l'état INFO. La sémantique du registre INFO a été présentée figure 6.5. Dans ce registre, le champ NBSR est le nombre de bits significatifs de R, le champ R0 indique si R est un zéro informatique ou non, le champ EXC indique les exceptions flottantes détectées et les champs sup et inf sont les résultats des comparaisons de nombres stochastiques. Ensuite l'automate repasse dans l'état d'attente.

6.3 Exécution de programmes

Le but est maintenant d'utiliser notre système pour effectuer des calculs en arithmétique à virgule flottante simple précision. Le processeur que nous utilisons ne possédant par d'instruction flottante, il va falloir dans un premier temps les redéfinir afin qu'il puisse les faire exécuter sur notre coprocesseur. Une fois que nous aurons redéfini l'ensemble des opérations flottantes nécessaires, nous pourrons faire tourner des programmes sur notre système.

6.3.1 Redéfinition des opérations

Nous venons de voir que pour effectuer une opération flottante en mode standard, il suffisait d'écrire les valeurs des opérandes dans les registres A1 et B1, ainsi que l'opération à effectuer dans le registre CMD. Ensuite pour récupérer le résultat de l'opération, il faut aller lire le registre RES1.

Une opération flottante va donc se réduire à écrire (resp. lire) les opérandes (resp. résultats) à l'adresse mémoire des registres correspondants.

Voici par exemple le code C d'une opération d'addition en mode standard, ainsi que les adresses en mémoire des différents registres du coprocesseur :

```
flottant plus_std(flottant a, flottant b)
{
   flottant res;

   *fpucmd = FPUCMD_ADD;
   *a1 = a;
   *b1 = b;
   res = *res1;

   return res;
}
```

(a) Addition standard

```
long *cmd    = 0xa2000000;
long *a1     = 0xa2000004;
long *b1     = 0xa2000008;
long *a2     = 0xa200000C;
long *b2     = 0xa2000010;
long *a3     = 0xa2000014;
long *b3     = 0xa2000018;
long *config = 0xa200001C;
long *res1   = 0xa2000000;
long *res2   = 0xa2000004;
long *res3   = 0xa2000008;
long *info   = 0xa200000C;
```

(b) Adresses des registres

Comme le jeu d'instruction du Mips utilisé dans le système ne possède pas d'instructions flottantes, nous ne pouvons pas utiliser le type `float` du C et donc nous sommes obligés de travailler avec nos propres types flottants. Dans le cas des opérations standard le type `flottant` est en fait le type `long` du C car nous travaillons avec les valeurs binaires des flottants que nous manipulons.

En mode CESTAC les opérations se déroulent de la même façon, sauf que les opérandes sont écrites dans A1, B1, A2, B2, A3, B3 et les résultats lus dans RES1, RES2, RES3 et INFO. Pour le mode CESTAC, le type `flottant` utilisé ne sera plus équivalent à `long` mais sera défini par comme étant une structure. Et du coup l'opération d'addition en mode CESTAC s'écrira :

```
typedef struct
{
   long x0;
   long x1;
   long x2;
   long nbs;
} flottant;
```

(a) Type flottant

```
flottant plus_ces(flottant a, flottant b)
{
   flottant res;

   *fpucmd = FPUCMD_ADD;
   *a1 = a.x0;
   *b1 = b.x0;
   *a2 = a.x1;
   *b2 = b.x1;
   *a3 = a.x2;
   *b3 = b.x2;
   res.x0 = *res1;
   res.x1 = *res2;
   res.x2 = *res3;
   res.nbs = ((*info & 0xF8000000) >> 27);

   return res;
}
```

(b) Addition CESTAC

Ainsi nous avons redéfini l'ensemble des opérations flottantes dans chacun des deux modes (standard et CESTAC), à savoir : l'addition, la soustraction, la multiplication, la division, les conversions entier $\leftrightarrow$ flottant, les comparaisons, et les fonctions mathématiques (racine carrée, logarithme etc.).

Une fois l'ensemble de ces opérations défini, nous pouvons commencer à exécuter des programmes.

6.3.2 Jeu de test

Pour évaluer notre système, nous avons utilisé le jeu de test fourni avec la bibliothèque CADNA[2], ainsi que quelques exemples proposés par W. Kahan [46].

Test 1

Cet exemple est dû à S. M. Rump [70]. Il s'agit d'évaluer le polynôme $P(x,y) = 9x^4 - y^4 + 2(y^2)$ pour les couples $(x,y) = (10864, 18817)$ et $(x,y) = (\frac{1}{3}, \frac{2}{3})$.

Test 2

Le but de ce test est de résoudre l'équation du second degré : $0.3x^2 - 2.1x + 3.675 = 0$

Test 3

Cet exemple permet de calculer le déterminant d'une matrice de Hilbert de taille 11. Les coefficients d'une matrice de Hilbert sont définis par : $H_{i,j} = \frac{1}{i+j-1}$

Test 4

Nous voulons trouver les racines du polynôme $f(x) = 1.47x^3 + 1.19x^2 - 1.83x + 0.45$ par la méthode de Newton, c'est-à-dire en utilisant l'algorithme itératif $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ qui s'arrête lorsque $|x_n - x_{n-1}| < 1e^{-12}$ et avec comme valeur initiale $x_0 = 0.5$

Test 5

Il s'agit de résoudre par la méthode de Gauss avec pivot partiel le système $A.X = B$ avec

$$A = \begin{pmatrix} 21 & 130 & 0 & 2.1 \\ 13 & 80 & 4.74e^8 & 752 \\ 0 & -0.4 & 3.9816e^8 & 4.2 \\ 0 & 0 & 1.7 & 9e^{-9} \end{pmatrix} \text{ et } B = \begin{pmatrix} 153.1 \\ 849.74 \\ 7.7816 \\ 2.6e^{-8} \end{pmatrix}$$

[2] http ://www-anp.lip6.fr/cadna/

Test 6

Nous cherchons à résoudre un système linéaire d'ordre 20 par la méthode de Jacobi où le critère d'arrêt est $||X_{n+1} - X_n|| \leq 0.0003$. Les coefficients de A et B sont définis aléatoirement.

Test 7

Ce test, proposé par W. Kahan [46], permet de calculer le rapport des aires de deux triangles dont les longueurs des côtés sont x, y et z pour le premier et x, y et $2z$ pour le second, x, y et z satisfaisant la condition $x \geq y \geq 2z \geq 2(x - y)$. Ce rapport est défini par :

$$r = \sqrt{\frac{(x + y + z)(y + z - x)(x + z - y)(x + y - z)}{(x + y + 2z)(y + 2z - x)(x + 2z - y)(x + y - 2z)}}$$

Ce calcul est effectué pour $x = y = 1.234567e + 06$ et $z = 1.043e - 08$.

Test 8

Ce test a été adapté d'un exemple de Jean-Michel Muller [28] par W. Kahan [46]. Il consiste à calculer :

$$t = L - \frac{(M - N/(L - (M - N/z)/(L - (M - N/y)/z)))}{(L - (M - N/(L - (M - N/y)/z))/(L - (M - N/z)/(L - (M - N/y)z)))}$$

avec :

$$\begin{array}{lll} L = a + b + c & M = a(b + c) + bc & N = abc \\ x = (b + c)/2 & y = (b^2 + c^2)/(b + c) & z = L - (M - N/x)/y \\ a = 3.0e + 08 & b = 6 & c = 5 \end{array}$$

Test 9

Ce dernier test est celui du calcul de l'itération logistique. Il s'agit d'étudier le comportement de la suite suivante en fonction de la valeur de a avec comme premier élément $x_0 = 0.3$ et ce pour 10000 itérations :

$$x_{n+1} = ax_n(1 - x_n) \text{ avec } 0 < a < 4$$

6.4 Résultats et performances

L'ensemble du jeu de test décrit précédemment a été exécuté sur notre système à l'aide du simulateur CASS (Cycle Accurate System Simulator)[42] [63] qui est un simulateur précis au cycle et au bit près et qui prend en entrée des models décrit au niveau RTL. Nous allons maintenant décrire les résultats obtenus en utilisant le mode CESTAC de notre système.

6.4.1 Résultats

Test 1

Lorsque le polynôme P est évalué avec $(x, y) = (10864, 18817)$, le résultat obtenu est un zéro informatique qui signifie que le résultat n'a aucun bits significatifs alors qu'en utilisant l'arithmétique flottante le résultat est +7.08158976e+08. Avec $(x, y) = (\frac{1}{3}, \frac{2}{3})$ le résultat fourni est +8.02469134e-01 avec 23 bits significatifs. Ce résultat est le même en utilisant l'arithmétique flottante. Ici pour un même algorithme, le nombre de bits significatifs du résultat obtenu peut varier de la valeur minimale (0) à une valeur optimale (23).

Test 2

Ici le discriminant calculé par la méthode CESTAC est un zéro informatique et c'est donc le cas du discriminant nul qui va être utilisé pour déterminer la solution. La solution est donc la racine double $x = 3.5$ qui est également la solution mathématique. En utilisant l'arithmétique flottante avec l'arrondi au plus près (mode par défaut), le discriminant est négatif et les solutions sont donc deux racines complexes conjuguées :
`z1=0.3499999E+01+i*0.9765625E-03` et `z2=0.3499999E+01-i*.9765625E-03`.
Le contrôle des opérations de comparaison a donc permis de s'orienter correctement dans le calcul de la solution.

Test 3

La qualité des résultats dans le calcul du pivot se dégrade au fur et à mesure des itérations pour devenir nulle à la 6^{e} itération. Le déterminant de la matrice est alors un zéro informatique. Les résultats fournis sont valides avec leur nombre de bits significatifs associé.

Contrairement à l'arithmétique flottante, la méthode CESTAC permet ici d'estimer la précision des résultats.

Test 4

L'algorithme s'arrête à la 9e itération lorsque $|x_n - x_{n-1}|$ n'est plus significatif. Le résultat a alors 12 bits significatifs. En utilisant l'arithmétique flottante et le critère d'arrêt $|x_n - x_{n-1}| < 1e^{-12}$, la solution est obtenue après 23 itérations, mais sans informations quant à sa précision.

Test 5

Lors de la réduction de la 3e colonne en arithmétique flottante, $A(3,3)$ vaut 4864 et est choisi comme pivot, alors que sa valeur exacte est 0. Avec CESTAC, $A(3,3)$ est un zéro informatique et n'est donc pas choisi comme pivot et le résultat final est correct. Là encore la méthode CESTAC a permis de s'orienter correctement lors des débranchements et ainsi d'obtenir la bonne solution.

Test 6

En utilisant l'arithmétique flottante, le résultat est obtenu à la 678e itération alors qu'à partir de la 33e il n'y a plus d'amélioration du résultat. En revanche avec CESTAC, du fait que le calcul s'arrête dès lors qu'il n'y a plus d'amélioration du résultat (à partir de la 33e itération) cela permet, dans ce cas, de gagner un facteur 20 sur le temps d'exécution.

Test 7

Le résultat obtenu en utilisant la méthode CESTAC est bien le résultat exact $r = 0.5$ avec 23 bits significatifs.

Test 8

Le résultat exact est $t = (b^7 + c^7)/(b^6 + c^6)$, soit 5.74912.... En utilisant l'arithmétique flottante, le résultat obtenu est 3.10^8, de même qu'avec la méthode CESTAC. La différence provient du fait que lors de l'exécution de la méthode, il y a eu une division instable. De ce fait, l'hypothèse de validation de la méthode qui demande de s'assurer que lors d'une division, le diviseur est toujours significatif, n'est pas valide et le résultat final doit être considéré comme faux. La méthode CESTAC n'a donc pas été mise en défaut.

Test 9

Ici les résultats dépendent de la valeur de a :
- Si $a < 3$, la suite converge vers 0.5
- Si $3.0 \leq a \leq 3.57$, la suite est périodique et la période dépend de a
- Si $a > 3.57$ la suite est chaotique, c'est-à-dire qu'un élément ne peut pas être prédit des itérations précédentes

En utilisant la méthode CESTAC, nous pouvons remarquer que plus a se rapproche de 3.57 plus le nombre de bits significatifs du résultat diminue. Par contre, lorsque a est supérieur à 3.57, un zéro informatique apparaît et par conséquent une multiplication instable est détectée, indiquant ici que la suite ne peut pas être calculée.

6.4.2 Performances

Notre système met donc bien en œuvre l'arithmétique stochastique discrète comme le fait la bibliothèque CADNA, reste maintenant à savoir quelles en sont les performances. Pour cela nous avons comparé le nombre de cycles nécessaires à l'exécution des différents programmes avec notre unité flottante stochastique :
- en utilisant l'arithmétique flottante IEEE-754 (IEEE)
- en utilisant la méthode CESTAC logicielle, c'est-à-dire que nous avons porté la bibliothèque CADNA en logiciel sur notre système en utilisant l'unité flottante standard sans contrôle et estimation de la précision en matériel, mais en logiciel (CADNA)
- en utilisant la méthode CESTAC logicielle mais avec calcul du nombre de chiffres significatifs à chaque opération, comme c'est le cas en matériel, et non uniquement pour les opérations de comparaison, les divisions et les multiplications. Ceci permet de connaître le nombre de bits significatifs de chaque résultat (CADNA+)
- en utilisant le matériel spécifique de notre unité flottante stochastique pour effectuer les opérations stochastiques (matériel)

Le tableau 6.2 présente le nombre de cycles nécessaires à l'exécution des différents programmes de test selon le mode d'utilisation ainsi qu'un comparatif en temps d'exécution des différentes implantations logicielles par rapport au matériel.

Test	Temps d'exécution en nombre de cycles				Comparatifs logiciel/matériel		
	IEEE standard	CADNA logiciel	CADNA+ logiciel	CADNA+ matériel	IEEE	CADNA	CADNA+
1	20 898	352 825	412 791	42 432	0.49	8.32	9.73
2	27 039	146 042	173 365	28 218	0.96	5.18	6.14
3	418 968	9 652 594	15 594 459	1 102 328	0.38	8.76	14.1
4	77 792	1 829 161	2 808 039	174 282	0.45	10.5	16.1
5	61 581	622 523	914 372	134 669	0.49	4.62	6.79
6	291 533 982	262 288 214	475 506 026	2 998 673	97.2	87.5	159
7	23 371	175 157	288 578	58 094	0.4	3.02	4.97
8	26 024	407 814	699 617	62 090	0.42	6.57	11.3
9	30 227 253	527 944 035	675 977 221	73 586 880	0.41	7.17	9.19

Tab. 6.2 – Performances du système

La figure 6.7 est une synthèse de ces résultats afin de comparer le temps d'exécution moyen selon le mode d'utilisation. Dans cette synthèse, nous ne prendrons pas en compte le test 6 qui ne montre pas le surcoût en temps dû à la méthode CESTAC a proprement parlé, puisque dans ce cas les bonnes performances sont dues au critère d'arrêt plus optimal lorsque la méthode CESTAC est utilisée.

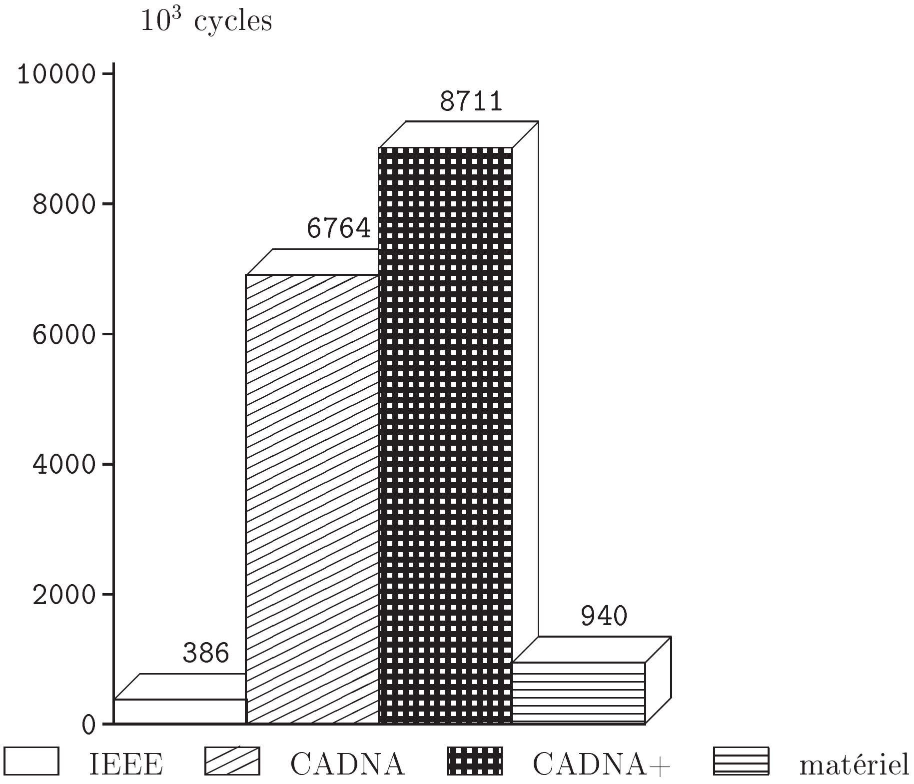


FIG. 6.7 – Temps moyen d'exécution (en nombre de cycles)

6.5 Conclusion

Nous avons conçu un système sur puce intégrant l'unité flottante stochastique. Ce système est capable d'effectuer des opérations flottantes avec estimation et contrôle des erreurs d'arrondi. L'exécution de programme sur ce système donne les mêmes résultats que la bibliothèque CADNA en étant entre 3 et 10.5 fois plus rapide. De plus l'utilisation du mode CESTAC n'est que 1 à 2.6 fois plus lente que l'arithmétique flottante standard.

Chapitre

7

Conclusions et perspectives

Sommaire

Le but de cette thèse était de fournir une implantation matérielle de la méthode CESTAC. Cette implantation devait améliorer les temps d'exécution par rapport à CADNA et permettre la détection rapide des instabilités numériques. Ce chapitre va donc conclure ce travail et ouvrir les perspectives à lui donner.

7.1 Conclusions

7.1.1 Implantation matérielle

Unité flottante stochastique

Dans un premier temps nous avons développé une unité flottante stochastique permettant d'effectuer des opérations aussi bien sur des nombres flottants que sur des nombres stochastiques. Dans le cas d'opération sur des nombres flottants cette unité suit la norme IEEE-754. Pour ce qui est des nombres stochastiques, elle permet en plus l'estimation de la précision du résultat et le contrôle des erreurs d'arrondi. Cette unité a été réalisée sous la forme d'un générateur paramétrable permettant de traiter différents formats de flottants, d'intégrer différents opérateurs flottants, de jouer sur le nombre d'étages de pipeline.

A partir de ce générateur, nous avons pu obtenir, entre autre, une description physique d'une unité flottante stochastique 32 bits à trois niveaux de pipeline réalisant l'addition/soustraction, la multiplication, la division, la comparaison et les conversions entier $\leftrightarrow$ flottant. Cette unité représente une surface de 1.61 mm^2 pour une fréquence de 83 MHz et pour une technologie de $0.25\mu m$.

Système sur puce

Enfin nous avons intégré cette unité flottante dans un système complet afin de pouvoir l'utiliser. Ce système permet d'exécuter des programmes écrits en langage C en utilisant l'arithmétique stochastique discrète.

7.1.2 Temps d'exécution

Notre implantation matériel de la méthode CESTAC dans un système sur puce est entre 3 à 10.5 fois plus rapide que l'utilisation de la bibliothèque CADNA en logiciel sur notre unité flottante en mode standard, c'est-à-dire que l'estimation et le contrôle des erreurs d'arrondi est fait en logiciel sans utiliser le matériel dédié. De plus notre implantation ne représente qu'une perte de temps d'un facteur 1 à 2.6 par rapport à l'utilisation de l'arithmétique flottante standard IEEE-754 de notre unité flottante. Nous pouvons donc dire qu'il est intéressant de porter en matériel la méthode CESTAC.
Bien évidemment, l'idéal serait que l'unité flottante stochastique soit directement intégrée dans le processeur comme unité arithmétique. Pour cela il faudrait faire évoluer le processeur Mips R3000 utilisé vers un processeur Mips R3010 qui possède une unité de calcul en virgule flottante.

7.1.3 Détection des instabilités numériques

La détection des instabilités numériques fait partie intégrante de l'opération stochastique. Ainsi lorsqu'une opération stochastique est effectuée sur notre système, le résultat sera fourni avec et en fonction de l'estimation de la précision. Ainsi l'estimation et le contrôle de la précision se font sur chaque opération stochastique.

7.2 Perspectives

Actuellement le processeur Mips R3000 inclut dans notre système n'implante pas le flottant, ce qui nous oblige à définir un type flottant et toutes les opérations nécessaires pour ce type. Un premier travail serait donc d'enrichir le jeu d'instruction avec les opérations flottantes afin de pouvoir fonctionner directement avec le type flottant plutôt que d'être obligé de passer par un type entier et une conversion en binaire du flottant. Ainsi, lorsqu'une instruction flottante est reçu par le processeur, elle peut être directement envoyée au coprocesseur. Cela permet de faire évoluer le processeur sans pourtant y intégrer directement l'unité flottante.

Une étape suivante serait de mapper l'ensemble du système sur un FPGA (circuit spécifique dont la fonctionnalité est programmée par l'utilisateur), puis d'en évaluer, grandeur réelle, les performances. Et enfin faire éventuellement la même chose sur silicium sous la forme d'un ASIC.

Une autre perspective possible et que nous venons de commencer à mettre en œuvre, est d'intégrer directement notre unité flottante stochastique, sans la division, au processeur et dans ce cas de ne plus passer par un système sur puce. Mais la gestion du flottant au sein d'un processeur est quelque chose d'assez complexe à réaliser car il faut gérer le fait que le traitement des opérations flottantes nécessite un nombre plus important de cycles que les autres opérations. Du coup il faut tenir compte des différentes dépendances qu'il peut exister entre les opérations flottantes et les autres.

Enfin nous pourrions améliorer notre unité flottante afin de permettre aux différentes opérations flottantes de se réaliser en un nombre de cycles différent. Ceci est notament très utile pour la division, car nous avons bien vu que cet opérateur à un temps de propagation nettement supérieur aux autres opérateurs. Pour cela il faudrait mettre en place un système de dérivation qui permettrait à la division de s'exécuter en un nombre de cycles plus important que les autres opérations. Puis il faudra intégrer cette nouvelle unité dans le processeur.

Annexe

A

Les générateurs de nombres aléatoires

Sommaire

Nous avons vu en 3.4.1 qu'un moyen pour générer des nombres aléatoires en matériel, était d'utiliser un LFSR (Linear Feedback Shift Register). Le but de cette annexe est de montrer que ce type de générateur est en effet un bon générateur de nombres aléatoires pour la méthode CESTAC, à savoir pour l'obtention du mode d'arrondi.

A.1 Registres à décalage à rebouclages linéaires (LFSR)

Dans [66] il est proposé d'utiliser un LFSR pour la génération de bits aléatoires. Les LFSR sont basés sur une théorie mathématique assez complexe [53]. Ils peuvent être représentés par des polynômes dont les coefficients sont 0 ou 1. Certains de ces polynômes appelés polynômes primitifs permettent de générer des nombres aléatoires [33]. Une table des polynômes primitifs donne en fonction du degré du polynôme les valeurs des coefficients. Par exemple un polynôme primitif de degré 8 est : $x^8 + x^4 + x^3 + x^2 + 1$. A ce polynôme correspond en matériel un LFSR présenté par la figure A.1. L'arrondi aléatoire est donné par le registre 0 du LFSR.

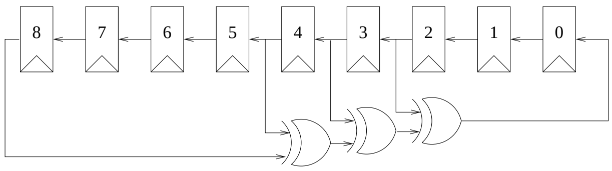


FIG. A.1 – LFSR de degré 8

Il reste maintenant à trouver quel degré de LFSR donne un bon générateur d'arrondi aléatoire pour la méthode CESTAC.

A.2 Test des générateurs de nombres aléatoires

Nous allons maintenant essayer de trouver un LFSR dont la séquence de bits aléatoires soit aussi aléatoire que celle du générateur de nombres aléatoires de la bibliothèque CADNA, à savoir la fonction `rand()` du langage C++.

Les statistiques fournissent de nombreux tests pour vérifier le caractère aléatoire d'une séquence binaire. Juan Soto propose dans [76] un ensemble de seize tests statistiques. Plus un générateur passe de test, plus il est efficace, mais rien ne dit qu'un générateur qui a passé les 16 tests n'échouera pas sur un autre. Les 16 tests proposés sont :

1. Test de distribution uniforme
2. Test de distribution uniforme dans un bloc
3. Test des séquences
4. Test de la plus longue séquence de 1 dans un bloc
5. Test du rang d'une matrice binaire
6. Test de la transformée de Fourier discrète
7. Test de non recouvrement de motif
8. Test de recouvrement de motif
9. Test statistique universel de Maurer
10. Test de compression de Lempel-Ziv
11. Test de complexité linéaire
12. Test des séries
13. Test de l'entropie approximative
14. Test des sommes cumulées
15. Test de la marche aléatoire
16. Test de la marche aléatoire variable

Tests de distribution uniforme (tests 1 et 2)

Ces tests permettent de détecter si la fréquence d'apparition d'un bit à 0 est la même que celle d'un bit à 1. Le test 1 vérifie cette condition pour toute la séquence, alors que le 2^{e} la vérifie pour des sous-séquences. La documentation [76] recommande de prendre une séquence d'au moins 100 bits et des sous-séquences d'au moins 10 bits.

Tests des séquences (test 3 et 4)

Ces tests permettent de compter le nombre d'apparitions d'une suite ininterrompue de bits identiques dans une séquence de bits. Ces tests déterminent si l'oscillation entre les 0 et les 1 est trop rapide ou trop lente. Le test 3 vérifie cette condition pour des séquence de 128, 6272 et 75.10^4 bits et le test 4 pour des sous-séquences de 8, 128 et 10^4 bits.

Test du rang de la matrice binaire (test 5)

Ce test a pour but de trouver des dépendances linéaires dans des sous-séquences de taille fixe de la séquence testée. Pour cela il calcule le rang des matrices formées à partir de sous-séquences disjointes. Les 38 matrices ainsi formées sont de taille 32 et la séquence doit comporter au moins 38912 bits.

Test de la transformée de Fourier discrète (test 6)

Ce test permet de trouver la hauteur du pic de fréquence de la transformée de Fourier discrète de la séquence de bits et ainsi d'y trouver une périodicité. La séquence doit avoir un minimum de 1000 bits.

Tests de recouvrement (tests 7 et 8)

Le but de ces tests est de trouver le nombre d'occurrences d'un motif spécifique dans une séquence. Pour cela ils utilisent des fenêtre de taille 9 et recherchent le motif 000000001 dans les séquences. Le test 7 fait ses recherches sur des fenêtres qui ne se recouvrent pas, alors que celles du test 8 se recouvrent. La séquence doit être d'au moins 2^{20} bits.

Test statistique universel de Maurer (test 9)

Ce test permet de calculer le nombre de bits entre deux mêmes sous-séquences. Plus ce nombre est petit moins la séquence est aléatoire. La séquence doit être d'au moins 904960 bits et les sous-séquences de 7 bits.

Test de compression (test 10)

Ce test a pour but de calculer le nombre de sous-séquences distinctes, qu'une séquence aléatoire doit avoir en nombre assez important. La séquence doit avoir au moins 10^6 bits.

Test de complexité linéaire (test 11)

Le but de ce test est de déterminer si la séquence est assez complexe pour être considérée comme aléatoire. Pour chaque sous-séquence de taille 1000, il cherche le plus petit LFSR capable de générer la sous-séquence, ce qui donne la complexité linéaire de cette sous-séquence. Plus le LFSR est petit, moins la sous-séquence est aléatoire. La taille de la séquence doit être d'au moins 10^6 bits.

Tests des séries et d'entropie (tests 12 et 13)

Le but de ces tests est de trouver la fréquence d'apparition de toutes les sous-séquences possibles, afin de déterminer si elles sont équiprobables. Le test de l'entropie (test 13) compare les fréquences d'une sous-séquence de taille m et d'une autre de taille $m+1$. La taille de la séquence doit être d'au moins n bits ($n \geq 100$), et celle de la sous-séquence de m ($m < \log_2 n$).

Test des sommes cumulées et marche aléatoire (tests 14, 15 et 16)

Le test 14 détermine la somme de la plus grande marche aléatoire sachant que -1 (resp. +1) est ajouté lorsqu'un 0 (resp. 1) est rencontré. Cette somme doit être à peu près égale à 0 (test 14). Le test 15 détermine le nombre de cycles différents pour partir d'un état (une valeur de la somme) et y revenir et le test 16 détermine le nombre de fois qu'un état est visité. La séquence doit être d'au moins 100 bits pour le test 14 et 10^6 bits pour les deux autres.

A.3 Résultats

Nous avons appliqué ces tests pour différentes tailles de LFSR, ainsi que pour le générateur de nombres aléatoires de CADNA. Le tableau A.1 présente les résultats de ces tests. `rand` est la fonction `rand` du langage C++ qui est le générateur de CADNA et `lfsrn` est un LFSR de degré n. Lorsqu'un S apparaît dans une colonne du tableau, cela signifie que le générateur a passé ce test avec succès. La dernière colonne donne le nombre de tests réussis sur un total de seize. Le but de ces tests est de trouver un LFSR dont le caractère aléatoire soit du même ordre que celui de la fonction `rand()` du langage C, c'est-à-dire qui passe autant de tests.

Générateur	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	TOT
`rand`	S	S	S	S	S		S		S	S	S	S	S	S			12
`lfsr5`																	0
`lfsr15`	S			S	S												3
`lfsr25`	S	S	S	S	S			S	S	S			S	S			10
`lfsr32`	S	S	S	S	S			S	S	S	S	S	S	S			12

TAB. A.1 – Résultats des tests statistiques

A.4 Conclusion

Au vu des résultats précédents, pour avoir un générateur de nombres aléatoires aussi bon que celui implanté dans la bibliothèque CADNA en C, il suffit d'utiliser un LFSR de degré 32. C'est donc celui-ci qui sera utilisé pour l'implantation matériel de l'arrondi aléatoire.

Annexe

B

Calcul du nombre de chiffres significatifs

Sommaire

Au chapitre 3, nous avons introduit une nouvelle façon de calculer le nombre de bits significatifs d'un nombre stochastique. Cette annexe présente la démonstration de la légitimité de cette méthode, elle est due à Jean-Marie Chesneaux.

B.1 Introduction

La méthode CESTAC fournit pour chaque résultat R d'un calcul un triplet (R_1, R_2, R_3) représentatif de ce résultat. La valeur théorique du nombre de chiffres significatifs exacts de R est donnée par la formule

$$C_{\overline{R}} = \log_n \left(\frac{\sqrt{3}.\left|\overline{R}\right|}{4,303s} \right).$$

avec $n = 10$ ou 2, $\overline{R} = \frac{1}{3}\sum_{i=1}^{3} R_i$ et $s^2 = \frac{1}{2}\sum_{i=1}^{3}\left(R_i - \overline{R}\right)^2$.

D'autre part, la valeur théorique du nombre de chiffres significatifs exacts entre deux réels a et b est donnée par la formule

$$C_{a,b} = \log_n \left(\frac{|a+b|}{2\,|a-b|} \right).$$

L'objectif est de comparer $C_{\overline{R}}$ avec $C_{opt} = \min\left(C_{R_1,R_2}, C_{R_1,R_3}, C_{R_3,R_2}\right)$.

Pour la suite, nous prendrons $n = 2$.

B.2 Résultats préliminaires

Remarque 1 *Dans le cas où une formule renverrait une valeur négative, la valeur finale retenue est* 0.

Remarque 2 C_{opt} *et* $C_{a,b}$ *renvoient la même valeur pour* R *et* $-R$.

Remarque 3

$$
\begin{aligned}
18 \times s^2 &= (2R_1 - R_2 - R_3)^2 + (2R_2 - R_1 - R_3)^2 + (2R_3 - R_1 - R_2)^2 \\
&= (R_3 - R_1 + R_2 - R_1)^2 + (R_2 - R_1 + R_2 - R_3)^2 + (R_3 - R_2 + R_3 - R_1)^2 \\
&= 2\,(R_3 - R_1)^2 + 2\,(R_2 - R_1)^2 + 2\,(R_3 - R_2)^2 \\
&\quad +2\,(R_3 - R_1)\,(R_2 - R_1) + 2\,(R_2 - R_1)\,(R_2 - R_3) + 2\,(R_3 - R_1)\,(R_3 - R_2) \\
&= 4\,(R_3 - R_1)^2 + 2\,(R_2 - R_1)^2 + 2\,(R_3 - R_2)^2 + 2\,(R_2 - R_1)\,(R_2 - R_3)
\end{aligned}
$$

Proposition 1 *Si les R_i ne sont pas de même signe et si $R \neq (0,0,0)$, alors*

$$C_{\overline{R}} = C_{opt} = 0.$$

D'après la remarque 2, nous pouvons supposer sans perdre de généralité que $R_1 \leq 0$ et $0 \leq R_2 < R_3$.

Alors $|R_1 + R_3| \leq |R_1 - R_3|$ et donc $\dfrac{|R_1 + R_3|}{2\,|R_1 - R_3|} \leq 1/2$ ce qui entraîne, d'après la remarque 1, $C_{opt} = C_{R_1,R_3} = 0$.

D'autre part,

$$s^2 \geq \frac{1}{18}\,(2R_1 - R_3 - R_2)^2 \geq \frac{1}{18}\,(R_3 + R_2 - R_1)^2\,.$$

D'après les hypothèses, $|R_1 + R_2 + R_3| \leq (R_3 + R_2 - R_1)$

En conséquences :

$$\frac{|\overline{R}|}{s} \leq \frac{\sqrt{18}}{3}\frac{|R_1 + R_2 + R_3|}{|R_3 + R_2 - R_1|} \leq \sqrt{2}.$$

Et finalement, $C_{\overline{R}} \leq \log_2\left(\dfrac{\sqrt{2}\sqrt{3}}{4.303}\right) \leq 0.$ ●

Il reste donc à étudier, grâce à la remarque 2 et à la proposition 1, le cas où les R_i sont tous positifs.

Supposons que $0 < R_1 \leq R_2 \leq R_3$.

Proposition 2 *$C_{opt} = C_{R_1,R_3}$. En d'autres termes, le minimum est obtenu pour les deux plus éloignés.*

Comme $C_{a,b} = +\infty$ si $a = b$, le résultat est trivial si $R_1 = R_2 = R_3$.

Supposons que $R_1 \neq R_3$, alors C_{R_1,R_3} existe et est fini.

Si $R_1 = R_2$ ou $R_2 = R_3$, le résultat est également trivial.

Si $R_1 \neq R_2$, alors :

$$\begin{aligned} R_1 (R_2 - R_3) \leq R_1 (R_3 - R_2) \quad &\Rightarrow \quad R_1 (R_2 - R_3) + R_3 R_2 - R_1^2 \leq R_1 (R_3 - R_2) + R_3 R_2 - R_1^2 \\ &\Rightarrow \quad (R_3 + R_1)(R_2 - R_1) \leq (R_3 - R_1)(R_2 + R_1) \\ &\Rightarrow \quad \frac{R_3 + R_1}{R_3 - R_1} \leq \frac{R_2 + R_1}{R_2 - R_1} \end{aligned} .$$

Finalement,

$$C_{R_1,R_3} \leq C_{R_1,R_2}.$$

En partant de $R_3 (R_1 - R_2) \leq R_3 (R_2 - R_1)$, on montre de la même façon que

$$C_{R_1,R_3} \leq C_{R_3,R_2}. \bullet$$

B.3 Résultat principal

Nous supposons toujours que $0 < R_1 \leq R_2 \leq R_3$.

Théorème 1

$$C_{opt} - 1,13 \leq C_{\overline{R}} \leq C_{opt} + 1,2.$$

D'après la proposition 2, $C_{opt} = C_{R_3,R_1}$.

D'après la remarque 3,

$$\begin{aligned}
\frac{\sqrt{3}\times\overline{R}}{4,303\times s} &\leq \frac{\sqrt{6}(R_1+R_2+R_3)}{4,303\times\sqrt{\left((2R_1-R_2-R_3)^2+(2R_2-R_1-R_3)^2+(2R_3-R_1-R_2)^2\right)}}\\
&\leq \frac{2\sqrt{6}(R_1+R_3)}{4,303\times(R_2+R_3-2R_1)}\\
&\leq \frac{2\sqrt{2}(R_1+R_3)}{4,303\times(R_3-R_1)}
\end{aligned}$$

d'où

$$C_{\overline{R}} \leq C_{R_3,R_1} + 1,2.$$

Nous avons, toujours d'après la remarque 3,

$$\begin{aligned}
\frac{\sqrt{3}\times\overline{R}}{4,303\times s} &\geq \frac{\sqrt{6}(R_1+R_3)}{4,303\times\sqrt{4\left(R_3-R_1\right)^2+2\left(R_2-R_1\right)^2+2\left(R_3-R_2\right)^2}}\\
&\geq \frac{(R_1+R_3)}{4,303\times(R_3-R_1)}
\end{aligned}$$

d'où

$$C_{\overline{R}} \geq C_{R_3,R_1} - 1,13. \bullet$$

B.4 Considérations pratiques

Sur ordinateur, C_{opt} se calcule beaucoup plus rapidement que $C_{\overline{R}}$ pour un résultat très comparable comme le montre le théorème 1.

En effet, supposons que $R_2 \geq R_1$ et soit $\lambda_{2,1}$ la position du premier bit significatif de $R_2 - R_1$ par rapport à R_1.

Soit e_1 (resp. e_2) l'exposant de R_1 (resp. R_2) dans la représentation IEEE et p le nombre de bits de la mantisse, nous avons :

$$R_2 - R_1 = 2^{e_1-\lambda_{1,2}} \times \alpha \text{ avec } \alpha \in [1,2[.$$

Nous décidons que la valeur maximale de $C_{\overline{R}}$, C_{opt} ou $\lambda_{1,2}$ est p, donc si $R_1 = R_2 = R_3$, la valeur obtenue est p.

Définition 2 *La nouvelle formule pour calculer $C_{\overline{R}}$ est :*

$$\Lambda = min(\lambda_{2,1}, \lambda_{3,1}, \lambda_{3,2}).$$

Nous obtenons les résultats suivants :

Proposition 3 *Si $R_1 \leq 0$ et $0 < R_2$, alors $\lambda_{2,1} = 0$*

$R_2 - R_1 \geq -R_1 \geq 0 \Rightarrow e_1 - \lambda_{2,1} \geq e_1 \Rightarrow \lambda_{2,1} \leq 0.$

Donc, $\lambda_{2,1} = 0$. ●

Proposition 4 *Si $0 < R_1 \leq R_2 \leq R_3$, alors $\Lambda = \lambda_{3,1}$*

$R_3 - R_1 \geq R2 - R_1 \Rightarrow e_1 - \lambda_{3,1} \geq e_1 - \lambda_{2,1} \Rightarrow \lambda_{2,1} \geq \lambda_{3,1}.$

$$\left.\begin{array}{rcl} R_3 - R_1 \geq R_3 - R_2 & \Rightarrow & e_1 - \lambda_{3,1} \geq e_2 - \lambda_{3,2} \\ & \text{et} & \\ R_2 \geq R_1 & \Rightarrow & e_2 \geq e_1 \end{array}\right\} \Rightarrow e_1 - \lambda_{3,1} \geq e_1 - \lambda_{3,2} \Rightarrow \lambda_{3,2} \geq \lambda_{3,1}.$$

Finalement, $\Lambda = \lambda_{3,1}$. ●

Proposition 5 *Si $R_2 \geq 2.R_1 > 0$, alors $\lambda_{2,1} = C_{R_1,R_2} = 0$*

Si $R_2 \geq 2.R_1$ alors $e_1 \geq e_1 + 1$, $R_2 - R_1 \geq R_2/2$ et $e_1 - \lambda_{2,1} \geq e_2 - 1 \geq e_1 \Rightarrow \lambda_{2,1} \leq 0$ et donc $\lambda_{2,1} = 0$.

C'est-à-dire :

$$\frac{R_2 + R_1}{2.(R_2 - R_1)} \leq \frac{3.R_2}{4.R_2} \leq 1 \Rightarrow C_{R_1,R_2} = 0.$$

Finalement,

$$R_2 \geq 2.R_1 \Rightarrow \lambda_{2,1} = C_{R_1,R_2} = 0. \;●$$

Proposition 6 *Si $2.R_1 > R_2 \geq R_1 > 0$, alors $\lambda_{2,1} - 1 \leq C_{R_1,R_2} \leq \lambda_{2,1} + 1$.*

Supposons que $2.R_1 > R_2 \geq R_1$, alors

$$\frac{R_2 + R_1}{2.(R_2 - R_1)} \leq \frac{R_2}{2^{e_1 - \lambda_{2,1}}}.$$

Mais $2.R_1 > R_2 \Rightarrow 2^{e_1+1} \geq R_2$.

Alors, $C_{R_1,R_2} \leq \lambda_{2,1} + 1$.

Nous avons aussi

$$\frac{R_2 + R_1}{2.(R_2 - R_1)} \geq \frac{R_1}{2^{e_1 - \lambda_{2,1} + 1}} \geq 2^{\lambda_{2,1} - 1}$$

Finalement,

$$\lambda_{2,1} - 1 \leq C_{R_1,R_2} \leq \lambda_{2,1} + 1 \; \bullet$$

Le théorème suivant prouve la validité de l'utilisation de Λ à la place de $C_{\overline{R}}$.

Théorème 2 *Quelque soit* $R = (R_1, R_2, R_3)$*, alors* $\Lambda - 2,13 \leq C_{\overline{R}} \leq \Lambda + 2,2$.

Si R_1, R_2 et R_3 ne sont pas de même signe, grâce aux propositions 1 et 3, $C_{\overline{R}} = \Lambda = 0$ et la double inégalité est vérifiée.

Si R_1, R_2 et R_3 sont de même signe, nous pouvons supposer que $0 < R_1 \leq R_2 \leq R_3$. Grâce aux propositions 2 et 4, $C_{opt} = C_{R_3,R_1}$ et $\Lambda = \lambda_{3,1}$.

D'après les propositions 5 et 6, $\Lambda - 1 \leq C_{opt} \leq \Lambda + 1$.

Le résultat final est obtenu en appliquant le théorème 2 $\bullet$

En pratique, le calcul de Λ est de loin le plus efficace.

Annexe

C

Calcul de la distance

Sommaire

Nous avons souvent besoin d'un opérateur qui calcule la valeur absolue de la différence (que nous appellerons distance), notamment lors de l'addition flottante puisque c'est comme cela qu'est calculée la valeur du décalage à faire pour aligner les mantisses, ou encore lors du calcul du nombre de bits significatifs où les différences entre les trois résultats intermédiaires doivent être calculées.

C.1 Première approche

Une première solution pour calculer cette distance ($D = |A - B|$) est d'utiliser deux additionneurs, un qui calcule $A - B$ et l'autre $B - A$. Ensuite en fonction de la valeur de la retenue sortante le résultat positif est choisi (figure C.1).

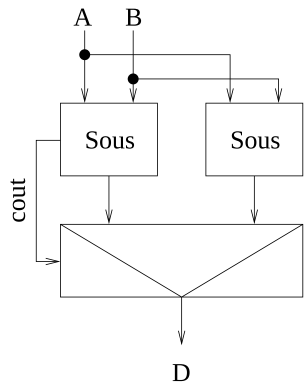


FIG. C.1 – Calcul de la distance

C.2 Autre approche

Cette première architecture est très coûteuse en matériel puisqu'elle nécessite deux soustracteurs. Une autre approche a été suggérée par Alain Guyot dans son cours sur l'arithmétique flottante[1]. Cette approche repose sur la constatation que $A - B = A + \overline{B} + 1$ et $B - A = \overline{A + \overline{B}}$. Ainsi le calcul de $A + \overline{B}$ est fait avec un unique additionneur. L'additionneur fournit comme résultats $A + \overline{B} + 1$ et $\overline{A + \overline{B}}$ et un multiplexeur permet de choisir le résultat positif en fonction de la retenue sortante (figure C.2).

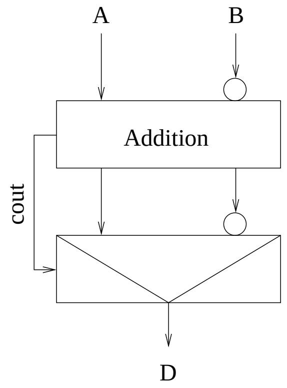


FIG. C.2 – Nouvelle architecture

L'additionneur utilisé pour ce calcul n'est pas un additionneur classique puisqu'il fournit deux résultats ($R = A + \overline{B} + 1$ et $S = A + \overline{B}$). Nous allons maintenant en détailler l'architecture.

C.3 Architecture de l'additionneur

L'architecture de cet additionneur est basée sur celle d'un additionneur à retenue anticipée [12]. Il utilise un assemblage de cellules :

- PiGi qui prennent en entrée les signaux A_i et $\overline{B}_i$ et calculent $P_{i,i} = A_i \; xor \; \overline{B}_i$ et $G_{i,i} = A_i \; and \; \overline{B}_i$ avec $0 \leq i < n$ où n est la taille
- PG qui prennent en entrée les signaux $P_{i,j}$, $G_{i,j}$, $P_{j-1,k}$ et $G_{j-1,k}$ et calculent $P_{i,k} = P_{i,j} \; and \; P_{j-1,k}$ et $G_{i,k} = G_{j-1,k} \; or \; (G_{i,j} \; and \; P_{i,j})$ avec $0 \leq k < j \leq i < n$

Cet assemblage est composé de trois parties :

1. Une première partie qui permet de calculer les signaux de propagation et de génération ($P_{i,i}$ et $G_{i,i}$) avec des cellules PiGi
2. Une deuxième partie calculant les retenues intermédiaires à l'aide de cellules PG

[1] http ://tima-cmp.imag.fr/~guyot/Cours/Arithmetique/flottant

3. Une dernière partie qui calculent les deux résultats de l'addition

$$R_i = P_{i,i}\ xor\ (G_{i,0}\ or\ P_{i,0}) \text{ et } S_i = P_{i,i}\ xor\ G_{i,0}$$

4. Enfin un multiplexeur qui permet de choisir le résultat final entre R et S en fonction de la retenue sortante

La figure C.3 présente l'architecture d'un tel additionneur 8 bits.

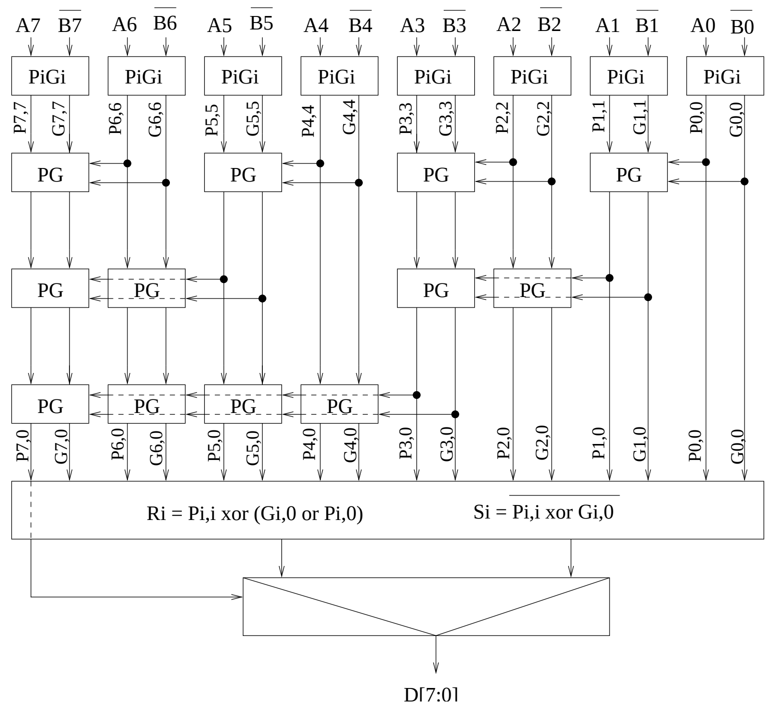


FIG. C.3 – Additionneur 8 bits

C.4 Résultats

Nous allons maintenant présenter à l'aide du tableau C.1 quelques résultats comparatifs entre les deux approches.

La deuxième approche (version 2) est nettement moins volumineuse que la première (version 1) puisqu'elle présente un gain en surface allant de 33% à 40% ce qui est tout à fait normal puisque la deuxième approche utilise un seul additionneur alors que la première en utilisait deux.

Pour ce qui est de la chaîne critique, il y a une perte de 4 à 16% pour les petites tailles

		Surface (mm^2)			Temps critique (ns)		
taille	techno	version 1	version 2	gain	version 1	version 2	gain
8	0.35μ	0.029	0.019	34%	2.7	2.8	-4%
11	0.35μ	0.039	0.025	36%	2.7	2.9	-7%
32	0.35μ	0.14	0.086	39%	4.5	4.2	6%
64	0.35μ	0.302	0.181	40%	6.5	5.7	12%
8	0.25μ	0.015	0.01	33%	0.6	0.7	-16%
11	0.25μ	0.02	0.013	35%	0.6	0.7	-16%
32	0.25μ	0.071	0.044	38%	1.1	1.1	0%
64	0.25μ	0.154	0.093	40%	1.7	1.5	12%

TAB. C.1 – Performances du bloc distance

(8 ou 11 bits), mais un gain de 0 à 12% pour les plus grandes tailles (16 ou 32 bits). Ces résultats sont à peu près les mêmes quelle que soit la technologie utilisée.

C.5 Conclusion

Nous avons réalisé un générateur qui calcule la valeur absolue de la différence de deux nombres.
Cet opérateur peut-être utilisé dans l'addition flottante lors du calcul de l'exposant. Dans ce cas il aura une taille de 8 bits (resp. 11) pour la simple (resp. double) précision. En technologie $0.25\mu m$ et en simple précision, il aura une surface de $0.01mm^2$ et une chaîne critique de $0.7ns$.

Annexe

D

Détection du premier bit à 1

Sommaire

Dans l'addition flottante, lors de la normalisation de la mantisse, il est nécessaire de connaître le nombre de bits à 0 en tête de mantisse, car cela donne le nombre de décalages à effectuer pour normaliser la mantisse. Cette opération se fait à partir du résultat grâce à un compteur de zéros en tête. Mais cela nécessite de connaître le résultat de l'addition des mantisses pour pouvoir faire la détection du premier bit à 1. Il est malgré tout possible d'effectuer cette détection en parallèle avec l'addition des mantisses, ce qui permet de réduire sensiblement le chemin critique (figure D.1).

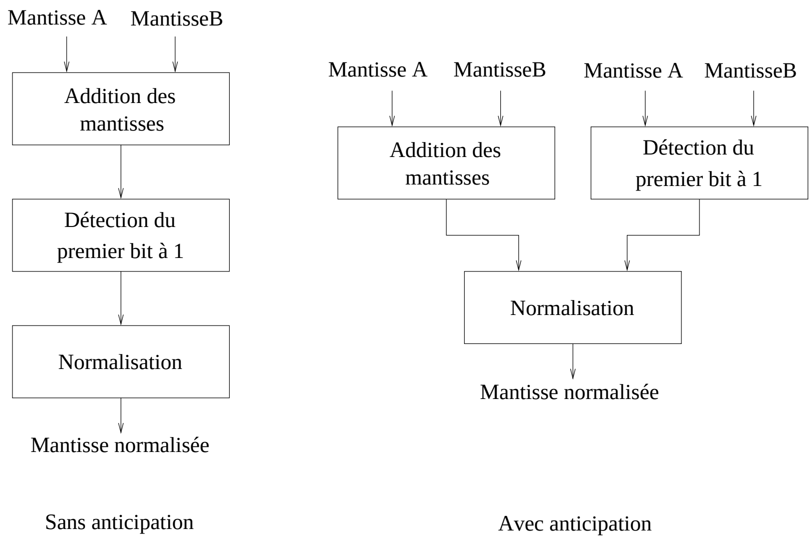


FIG. D.1 – Détection du premier bit à 1

L'architecture proposée, pour la détection du premier bit à 1, a été décrite dans [13].

D.1 Architecture de l'opérateur

L'architecture générale de l'opérateur de détection du premier bit à 1 est présentée par la figure D.2. L'opérateur est constitué de quatre parties :

- Un préencodeur qui met en forme les signaux nécessaires à l'encodeur et à l'arbre de détection
- Un encodeur qui calcule la position du premier bit à 1 avec une erreur possible d'un bit
- Un arbre de détection qui détecte cette erreur
- Un additionneur qui donne la position du premier bit à 1 corrigée en fonction de l'erreur détectée

L'architecture des différents blocs sera décrite dans les sections suivantes.

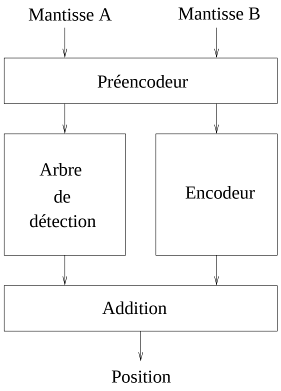


Fig. D.2 – Architecture générale

D.1.1 Le préencodeur

Il réalise la soustraction[1] des deux mantisses sans la propagation de retenue (*borrow save*). La position du premier bit à 1 est fonction de ce résultat et résumée par le tableau D.1 lorsque le résultat est positif. Le cas négatif est similaire, il n'y a qu'à échanger le rôle des 1 par $\overline{1}$.

[1]Une normalisation ne peut intervenir que lors d'une soustraction

Chaîne	Position du premier bit à 1	
$0^k11(x)$	premier 1	k+1
$0^k10(x \geq 0)$	premier 1	k+1
$0^k10^j(x < 0)$	premier 0 après le 1	k+2*
$0^k1\overline{1}^j1(x)$	dernier $\overline{1}$	k+j+1
$0^k1\overline{1}^j0(x \geq 0)$	dernier $\overline{1}$	k+j+1
$0^k1\overline{1}^j0^t(x < 0)$	premier 0 après le dernier $\overline{1}$	k+j+2*

* correction nécessaire

TAB. D.1 – Position du premier bit à 1

Soient A et B les deux opérandes (les mantisses) et $R = A - B$, on note :
- $e_i = 1 \; si \; a_i = b_i \; (r_i = 0)$
- $g_i = 1 \; si \; a_i > b_i \; (r_i = 1)$
- $s_i = 1 \; si \; a_i < b_i \; (r_i = \overline{1})$

La sous-chaîne à détecter est donnée par $g_i\overline{s}_{i+1} + s_i\overline{s}_{i+1}$ pour le cas positif et de la même façon par $s_i\overline{g}_{i+1} + g_i\overline{g}_{i+1}$ pour le cas négatif. La chaîne fournie à l'encodeur est donc :

$$f_i = e_{i-1}(g_i\overline{s}_{i+1} + s_i\overline{g}_{i+1}) + \overline{e}_{i-1}(s_i\overline{s}_{i+1} + g_i\overline{g}_{i+1})$$

Le préencodeur fournit également les signaux nécessaires à la détection de la correction. Il y a correction en présence des chaînes $0^k10^j\overline{1}(x)$ et $0^k1\overline{1}^j0^t\overline{1}(x)$ pour le cas positif et des chaînes $0^k\overline{1}0^j1(x)$ et $0^k\overline{1}1^j0^t1(x)$ pour le cas négatif. On note : $p_i = (g_i+s_i)\overline{s}_{i+1}$, $n_i = e_{i-1}s_i$ et $z_i = \overline{p_i + n_i}$ pour le cas positif et $n_i = (g_i + s_i)\overline{g}_{i+1}$, $p_i = e_{i-1}g_i$ et $z_i = \overline{p_i + n_i}$ pour le cas négatif. Ce sont les signaux P, N et Z qui sont passés à l'arbre de détection pour chacun des cas positif et négatif.

D.1.2 L'encodeur

Il s'agit d'un simple compteur de zéros en tête qui retourne la position du premier bit à 1 de la chaîne.

D.1.3 L'arbre de détection

Pour réduire la complexité de la détection de la correction, il y a un arbre de détection pour le cas positif et un autre pour le cas négatif. L'arbre de détection doit détecter la chaîne $(pz^k n)$ pour le cas positif et $(nz^k p)$ pour le cas négatif.
Pour détecter la chaîne $(pz^k n)$ on crée un arbre binaire qui prend en entrée les chaînes Z, P et N. Chaque nœud de l'arbre fournit un résultat qui peut prendre 5 valeurs différentes :

- Z : un z a été détecté (z^j)
- D : début de chaîne détectée $(z^j pz^t)$
- F : fin de chaîne détectée $(z^j n(x))$
- Y : toute la chaîne a été détectée $(z^j pz^k n(x))$
- U : les autres cas

La figure D.3 présente le fonctionnement de l'arbre de détection pour le cas positif. Sur le

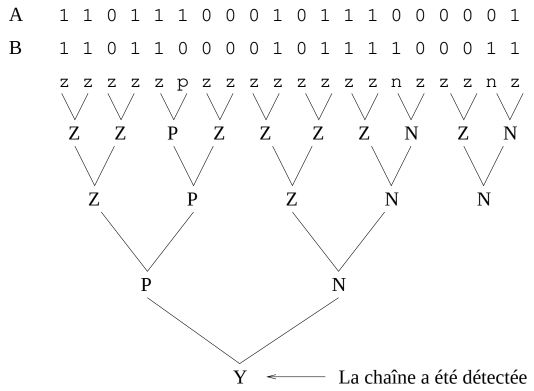


FIG. D.3 – Fonctionnement de l'arbre de détection

même principe on construit l'arbre de détection pour le cas négatif. Le résultat final vaut 1 si l'un des arbres à détecter une correction.

D.1.4 L'additionneur

Il effectue l'addition de la position du premier bit à 1 donnée par l'encodeur et de la correction calculée par l'arbre de détection. Le résultat est la vraie position du premier bit à 1 du résultat de l'addition des mantisses.

D.2 Résultats

La figure D.4 présente un comparatif en termes de surface et de temps critique des deux versions d'additionneur flottant : l'un avec la détection du premier bit à 1 en parallèle et l'autre en série. Ces résultats sont donnés pour différents formats de flottants, un nombre d'étages de pipeline variable et différentes technologies de gravure.

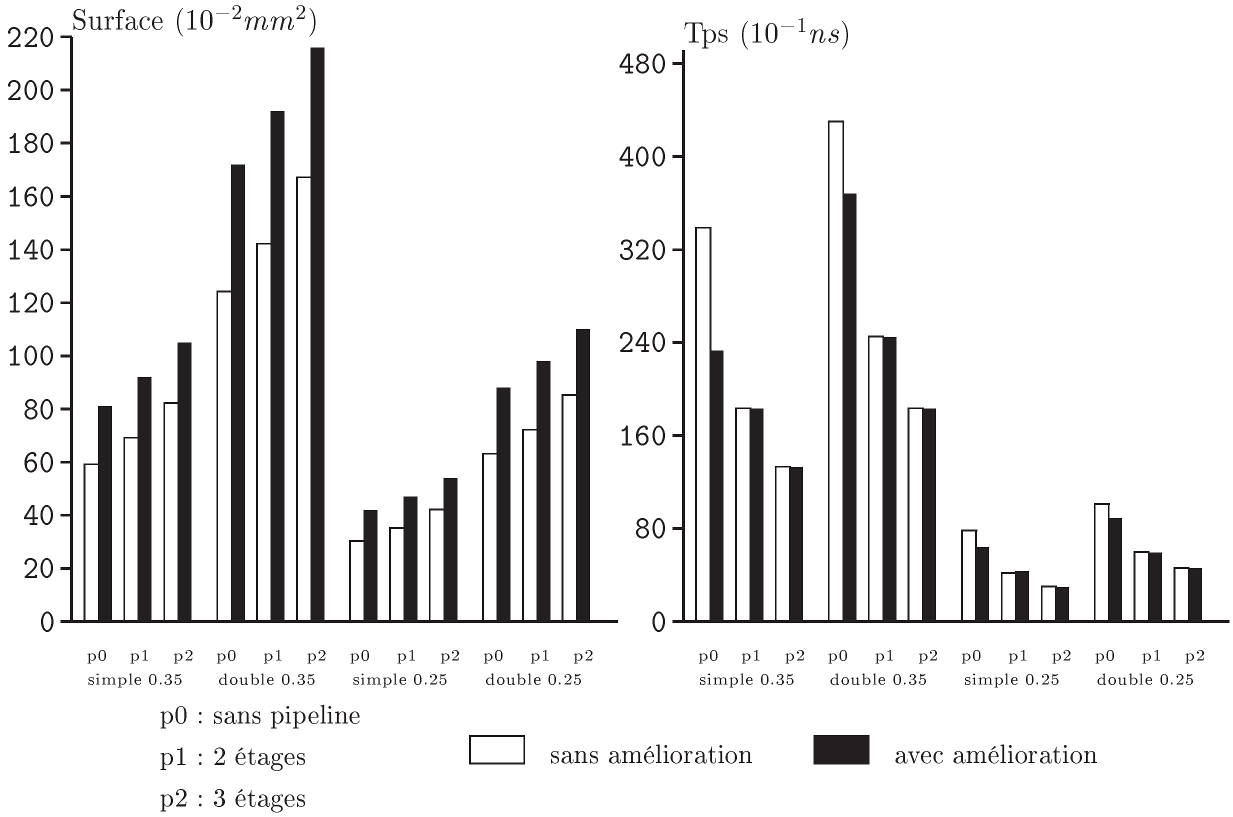


FIG. D.4 – Comparatif de l'additionneur flottant avec détection du premier bit à 1

Le fait d'effectuer la détection du premier bit à 1 en parallèle de l'addition des mantisses représente une augmentation de 10 à 25% de la surface de l'additionneur flottant.

Pour ce qui est du gain au niveau du temps critique, plusieurs constatations peuvent être faites :

- lorsqu'il n'y a pas de pipeline le gain est de 13 à 19%
- avec deux étages de pipeline, il est de 0 à 12%
- avec trois étages de pipeline, il passe de 0 à 6%

D.3 Conclusion

L'utilisation de la détection du premier bit à 1 en parallèle apporte un surcroît de matériel, mais peut améliorer les performances temporelles. Cependant, le gain en temps est conséquent lorsque l'additionneur est totalement combinatoire, mais il devient quasiment inexistant à partir du moment où il y a des étages de pipeline.
Il est donc intéressant de mettre cette amélioration en paramètre du générateur afin de laisser le choix au concepteur de l'utiliser ou non en fonction de l'additionneur qu'il souhaite générer (avec ou sans pipeline).

Annexe

E

Additionneur à résultats multiples

Sommaire

L'arrondi lors de l'addition flottante est une opération très coûteuse en temps puisqu'elle nécessite d'augmenter le résultat de l'addition des mantisses de 1, voire même de deux. Un autre additionneur, en plus de celui utilisé pour l'addition des mantisses, est donc nécessaire pour faire cet incrément. Le but de cette annexe est de présenter l'architecture d'un additionneur dont les résultats sont $A+B$, $A+B+1$ et $A+B+2$.

E.1 Principe de fonctionnement

Pour effectuer les opérations $A+B$ et $A+B+1$ nous allons utiliser la même méthode que pour le calcul de la distance (annexe C) en utilisant un additionneur fournissant deux résultats. La principale différence réside dans le fait que maintenant nous avons besoin d'effectuer $A+B$ et $A+B+1$ au lieu de $\overline{A+\overline{B}}$ et $A+\overline{B}+1$. Pour cela il suffit donc d'utiliser le même additionneur sans inverser l'entrée B et la sortie S et ainsi les résultats obtenus seront bien $A+B$ et $A+B+1$.

Ensuite il reste à effectuer l'opération $A+B+2$. Pour cela nous utilisons une technique présentée dans [67] qui consiste à ajouter un étage de demi-additionneurs (*halfadder*) avant l'additionneur à proprement parlé. Ensuite plusieurs cas se présentent :

- Soit le bit de poids faible du résultat de $A \oplus B$ vaut 0 et dans ce cas, en le forçant à 1, l'opération effectuée donnera comme résultats $A+B+1$ et $A+B+2$. $A+B+1$ aura donc forcément son bit de poids faible à 1 et en le forçant à 0, le résultat obtenu est $A+B$.
- Soit le bit de poids faible du résultat de $A \oplus B$ vaut 1 et dans ce cas, l'opération effectuée donnera comme résultats $A+B$ et $A+B+1$. $A+B+1$ aura forcément son bit de poids faible à 0 et en le forçant à 1, le résultat obtenu est $A+B+2$.

Ainsi en testant la valeur du bit de poids faible du résultat de $A \oplus B$, les trois résultats souhaités ($A+B$, $A+B+1$ et $A+B+2$) peuvent être obtenus.

E.2 Architecture de l'opérateur

L'architecture de l'opérateur est sensiblement la même que pour l'opérateur de distance (figure E.1) où il suffit de rajouter un étage de demi-additionneurs et un peu de logique pour obtenir les résultats souhaités.

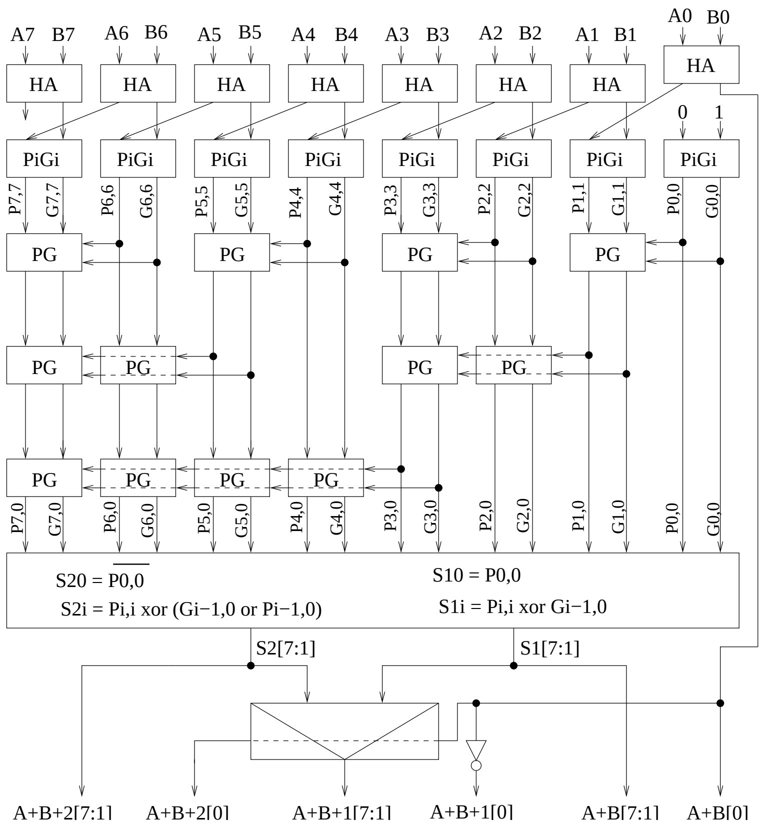
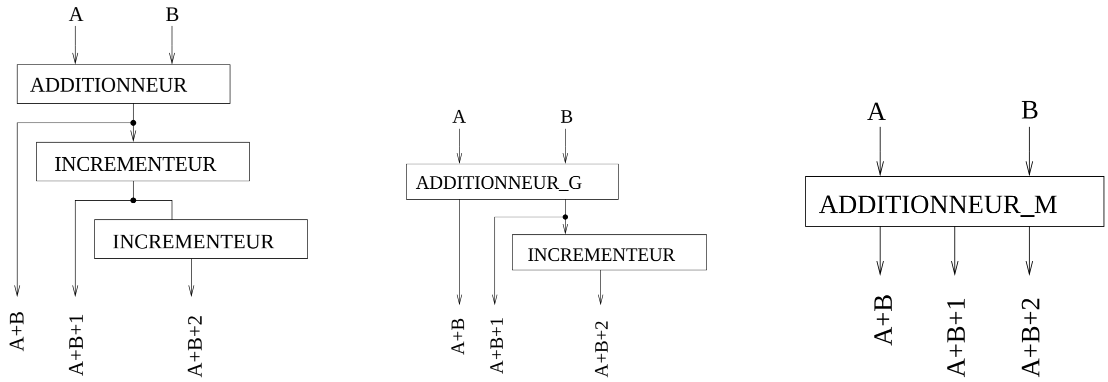


FIG. E.1 – Additionneur à résultats multiples

E.3 Résultats

Nous allons maintenant comparer les caractéristiques en délais et surfaces de différentes architectures permettant de réaliser les opérations $A+B$, $A+B+1$ et $A+B+2$. La première (figure E.2(a)) utilise un additionneur classique pour réaliser l'addition $A+B$ et deux opérateurs d'incrément pour effectuer $A+B+1$ et $A+B+2$. La seconde (figure E.2(b)) utilise l'additionneur d'Alain Guyot pour réaliser $A+B$ et $A+B+1$ et un opérateur d'incrément pour effectuer $A + B + 2$. Enfin la troisième architecture (figure E.2(c)) est l'additionneur à résultats multiples présenté précédemment.



(a) Architecture 1 (b) Architecture 2 (c) Architecture 3

FIG. E.2 – Architectures comparées

Le tableau E.1 présente les caractéristiques des différentes architectures. Les gains sont présentés par rapport à l'architecture précédente, c'est-à-dire que pour l'architecture 2, les comparaisons se font avec l'architecture 1, et pour l'architecture 3 avec la deuxième.

		Surface (mm^2)					Temps critique (ns)				
taille	techno	archi1	archi2	gain	archi3	gain	archi1	archi2	gain	archi3	gain
24	0.35μ	0.072	0.066	8%	0.071	-7%	5.9	4.7	20%	3.6	23%
53	0.35μ	0.175	0.159	9%	0.164	-3%	8	6	25%	4.2	30%
24	0.25μ	0.036	0.034	5%	0.036	-5%	1.4	1.2	14%	0.8	33%
53	0.25μ	0.089	0.081	9%	0.084	-3%	1.9	1.5	21%	1	33%

TAB. E.1 – Caractéristiques des différentes architectures

L'utilisation d'un additionneur à deux résultats représente un gain en temps d'environ 20% et en surface d'environ 7%, par rapport à un additionneur classique suivi d'opérateurs d'incrément. Utiliser un additionneur à trois résultats présente une petite augmentation de surface (environ 5%), par rapport à l'additionneur à deux résultats. Ceci est dû à l'ajout d'un étage de demi-additionneurs et de multiplexeurs. Par contre son utilisation permet un gain de temps d'environ 28%.

E.4 Conclusion

Nous avons réalisé un additionneur à résultats multiples qui permet de calculer en même temps $A + B$, $A + B + 1$ et $A + B + 2$. Cet additionneur peut-être utilisé pour réaliser en même temps l'addition des mantisses et l'opération d'arrondi dans l'addition flottante. Cet opérateur peut-être utilisé avec une taille de 24 bits (resp. 54 bits) dans l'additionneur flottant en simple (resp. double) précision. Dans ce cas, cet additionneur à une surface de $0.036mm^2$ (resp. $0.084mm^2$) en technologie $0.25\mu m$ et un temps critique de $0.8ns$ (resp. $1ns$).

Bibliographie

[1] Pentium Pro Family Developer's Manual. Volume 2 : Programmer's Reference Manual. Intel Corporation, 1996. Chapter 7, Floating-Point Unit.

[2] M. Aberbour, A. Houelle, H. Mehrez, N. Vaucher, and A. Guyot. On Portable Macro-Cell FPU Generators for Division and Square Root Operators Complying with the full IEEE-754 Standard. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 6(1) :114–121, March 1998.

[3] G.M. Amdahl, G.A. Blaauw, and F.P. Brooks Jr. Architecture of the IBM System/360. *IBM Journal of Research and Development*, 8(2) :87–101, 1964.

[4] ANSI/IEEE Std 754-1985. *IEEE Standard for Binary Floating-Point Arithmetic*, 1985.

[5] A. Avizienis. Signed digit number representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, 10 :389–400, 1961.

[6] D.H. Bailey. A Portable High Performance Multiprecision Package. Technical Report RNR-90-022, NASA Ames Research Center, May 1993.

[7] P.H. Bardell, W.H. Mc Anney, and J. Savir. Built-in Test for VLSI : Pseudorandom Techniques. New York, 1987. Wiley-Interscience.

[8] G. Bohlender. What do you need beyond IEEE arithmetic ? In C. Illrich, editor, *Computer Arithmetic and Self-Validating Numerical Methods*, pages 1–32. Academic Press, 1990.

[9] R.P. Brent. The complexity of multiple precision arithmetic. *Complexity of Computational Problem Solving*, 1976.

[10] R.P. Brent. Fast Multiple-Precision Evaluation of Elementary Functions. *ACM*, 23 :242–251, 1976.

[11] R.P. Brent. A fortran multiple-precision arithmetic package. *ACM Trans. Math. Software*, 4 :57–81, 1978.

[12] R.P. Brent and H.T. Kung. A regular layout for parallel adders. *IEEE Trans. on Computers*, C-31 :260–264, March 1982.

[13] J.D. Bruguera and T. Lang. Leading-One Prediction with Concurrent Position Correction. *IEEE Transactions on Computer*, 48(10) :1083–1095, October 1999.

[14] B. Buchberger. Gröbner Bases in MATHEMATICA : Enthusiasm and Frustration. *Programming Environments for High-level Scientific Problem Solving*, pages 80–91, 1991.

[15] T.M. Cater. Cascade : Hardware for High/Variable Precision Arithmetic. *Proceedings of the 9th Symposium on Computer Arithmetic*, pages 184–191, 1989.

[16] B.W. Char, K.O. Geddes, G.H. Gonnet, M.B. Monagan, and S.M. Watt. *MAPLE Reference Manual*. 1988.

[17] J.-M. Chesneaux. Etude théorique et implémentation en ADA de la méthode CESTAC. *Thèse de l'université P. et M. Curie*, 1988.

[18] J.-M. Chesneaux. CADNA, an ADA tool for round-off error analysis and for numerical debugging. *Proceedings Congress on ADA in Aerospace*, 1990.

[19] J.-M. Chesneaux. Study of the computing accuracy by using probabilitic approach. In *Contribution to Computer Arithmetic and Self-Validating Numerical Methods*, pages 19–30. C. Ulrich edition, 1990.

[20] J.-M. Chesneaux. *L'Arithmétique Stochastique et le Logiciel CADNA*. Habilitation à diriger des recherches, Université Pierre et Marie Curie, November 1995.

[21] J.-M. Chesneaux, S. Guilain, and J. Vignes. *La bibliothèque CADNA : présentation et utilisation*, 1996.

[22] J.-M. Chesneaux and J. Vignes. Sur la robustesse de la méthode CESTAC. *C. R. Acad. Scie.*, pages 855–860, 1988. t.307.

[23] J.-M. Chesneaux and J. Vignes. Les fondements de l'arithmétique stochastique. In *C.R. Acad. Sci.*, number 315 in 1, pages 1435–1440. Paris, 1992.

[24] D.M. Chiarulli, W.G. Rudd, and D.A. Buell. DRAFT : A Dynamically Reconfigurable Processor for Ingeger Arithmetic. *Proceedings of the 7th Symposium on Computer Arithmetic*, pages 309–318, 1985.

[25] M.S. Cohen, T.E. Hull, and V.C. Hamarcher. CADAC : A Controlled-Precision Decimal Arithmetic Unit. *IEEE Transactions on Computers*, C-32 :370–377, 1983.

[26] V. Coissard and A. Guyot. OCAPI : a coprocessor for infinite precision arithmetic. *International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN'97)*, 1997.

[27] L. Dadda. Some schemes for parallel multipliers. *Altat Frequenza*, 19 :349–356, March 1965.

[28] M. Daumas and J.-M. Muller. *Qualité des calculs sur ordinateur. Vers des arithmétiques plus fiables ?* 1997.

[29] Digital Equipment Corporation. *DEC VAX Architecture Handbook*. Massachussetts, 1981.

[30] F. Dromard, Y. Body, M.-M. Paget, A. Greiner, P. Bazargan Sabet, and F. Pétrot. Interactive Learning of Processor Architecture. In *5th International Conference on Computer Aided Engineering Education (CAEE'99)*, pages 123–129, Sofia, Bulgaria, September 1999.

[31] J.S. Ely. The VPI Software Package for Variable Precision Interval Arithmetic. *Interval Computation*, 2 :135–153, 1993.

[32] GAO/IMTEC-92-26. Patriot Missile Defense : Software Problem Led to System Failure at Dhahran, Saudi Arabia. 1992.

[33] S.W. Golomb. *Shift Register Sequences.* Laguna Hills, 1982.

[34] T. Granlund. GNU Multiple Precision Arithmetic Library. Technical report, 1996. edition 2.0.

[35] A. Greiner and al ALLIANCE. A complet set of CAD Tools for teaching VLSI Design. *Third EuroChip Workshop*, 1992. http ://www-asim.lip6.fr/alliance.

[36] A. Greiner, F. Pétrot, and F. Wajsbürt. Alliance CAD system portable cell librairies. In *Advanced Training Course Mixed Design of VLSI Circuits*, pages 172–175, Debe, Poland, April 1994.

[37] A. Guyot, Y. Herreros, and J.-M. Muller. JANUS, an on-line multiplier/ivider for manipuling large numbers. In M.J. Irwin and R. Stefanelli, editors, *ARITH-8 : 8th IEEE Symposium on Computer Arithmetic*, pages 106–111, Como, Italy, May 1987. IEEE Computer Society Press.

[38] K. Hafner. Chips for High Precision Arithmetic. *Computer Arithmetic and Self-Validating Numerical Methods*, pages 33–54, 1990.

[39] G. Heindl. An improved algorithm for computing the product of two machine intervals. In Fachbereich Mathematik, editor, *Interner Bericht IAGMPI-9304*, Gesamthochschule Wuppertal, 1993.

[40] J.L. Hennessy, D. Goldberg, and D.A. Patterson. *Computer Architecture : A quantitavtive Approach*. 2nd edition edition, January 1996.

[41] E. Hokenek and R. Montoye. Leading-Zero Anticipator (LZA) in the IBM RISC System/6000 Floating-Point Execution Unit. *IBM Journal of Research and Development*, 34(1) :71–77, January 1990.

[42] D. Hommais and F. Pétrot. Efficient Combinational Loops Handling for Cycle Precise Simulation of System on a Chip. *IEEE Euromicro Conference*, pages 51–54, 1998.

[43] A. Houelle. *GenOptim : un environnement d'aide à la conception de générateurs de circuits portables optimisés en performance et en surface*. PhD thesis, Université Pierre et MArie Curie, 20 June 1997.

[44] N. Ide, M. Hirano, Y. Endo, S. Yoshioka, H. Murakami, A. Kunimatsu, T. Sato, T. Kamei, T. Okada, and M. Suzuoki. 2.44-GFLOPS 300-MHZ Floating-Point Vector-Processing Unit for High-Performance 3-D Graphics Computing. *IEEE Journal of Solid-State Circuits*, 35(7) :1025–1033, July 2000.

[45] R.M. Jessani and C.H. Olson. The floating-point unit of the PowerPC 603e microprocessor. *IBM Journal of Research and Development*, 40(5) :559–566, September 1996.

[46] W. Kahan. *The Improbability of Probabilistic Error Analyses for Numerical Computations*. UCB Statistics Colloquium, evans hall edition, 1996.

[47] J. Kernhof. A CMOS Floating-Point Processing Chip for Verified Exact Vector Arithmetic. *ESSIRC 94*, 1994.

[48] R. Klatte, U. Kulisch, C. Lawo, M. Rausch, and A. Wiethoff. *C_XSC, A C++ Class Library for Extended Scientific Computing*. Berlin/Heidelberg/New York, 1993.

[49] R. Klatte, U. Kulisch, M. Neaga, and Ch. Ullrich. *PASCAL-XSC - Language Reference with Examples*. Berlin, 1992.

[50] A. Knöfel. Hardware Kernel for Scientific/Engineering Computations. *Scientific Computing with Automatic Result Verification*, pages 549–570, 1993.

[51] O. Knuppel. PROFIL/BIAS - A Fast Interval Library. *Computing*, 53 :277–288, 1994.

[52] J. Little and C. Moler. *MATLAB Reference Guide.* Natick, MA, 1993.

[53] E.J. Mc Cluskey. Built-In-Self-Test. *IEEE Design and Test*, 28-37, April 1985.

[54] R. Montoye, E. Hokenek, and S. Runyon. Design of the IBM RISC System/6000 Floating-Point Execution Unit. *IBM Journal of Research and Development*, 34(1) :59–70, January 1990.

[55] R.E. Moore. *Interval arithmetic and automatic error analysis in digital computing.* PhD thesis, Stanford University, 1962.

[56] J.-M. Muller. *Arithmétique des ordinateurs.* 1989.

[57] A. M. Nielsen, D. W. Matula, C. N. Lyu, and G. Even. Pipelined Packet-Forwarding Floating-Point : II. an Adder. In *Proceedings of the 13th Symposium on Computer Arithmetic*, pages 148–155, July 1997. IEEE Computer Society Press.

[58] S. F. Oberman, H. Al-Twaijry, and M. J. Flynn. The SNAP Project : Design of Floating-Point Arithmetic Units. In *Proceedings of the 13th Symposium on Computer Arithmetic*, pages 156–165, July 1997.

[59] S. F. Oberman and M. J. Flynn. A Variable Latency Pipelined Floating-Point Adder. In *Proceedings of Euro-Par'96, Springer LNCS*, volume 1124, pages 182–192, August 1996.

[60] V. Oklobdzija. An Algorithmic and Novel Design of a Leading Zero Detector Circuit : Comparaison with Logic Synthesis. *IEEE Transaction on VLSI Systems*, 2(1) :124–128, March 1994.

[61] OMI 324. *OMI/PI-Bus Specification*, 1994. PI-BUS Rev. 0.3d.

[62] M.-M. Paget, P. Bazargan Sabet, and A. Greiner. An Example of Practice Based Engineering Education : the Design of a Microprogrammed MIPS Processor with the Alliance CAD System. In *Computer Aided Learning In Engineering (CALIE'2001)*, pages 47–50, Tunis, Tunisie, November 2001.

[63] F. Pétrot. Cycle Accurate System Simulation. *Medea-Esprit Conference*, November 1999.

[64] M. Pichat and J. Vignes. *Ingénierie du contrôle de la précision des calculs sur ordinateur.* Technip edition, 1993.

[65] D.A. Pope and M.L. Stein. Multiple precision arithmetic. *Communications of the ACM*, 3 :652–654, 1960.

[66] W. H. Press, S. A. Teukolsk, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C : The Art of Scientific Computing.* Cambridge University Press, 2nd edition, January 1993.

[67] N. T. Quach and M. J. Flynn. An Improved Algorithm for High-Speed Floating-Point Addition. Technical Report CSL-TR-90-442, Computer Systems Laboratory, Stanford University, August 1990.

[68] N. T. Quach and M. J. Flynn. Leading One Prediction - Implementation, Generalization, and Application. Technical Report CSL-TR-91-463, Computer Systems Laboratory, Stanford University, March 1991.

[69] E.K. Reuter and al. Some Experiments Using Interval Arithmetic. *Proceedings of the 4th Symposium on Computer Arithmetic*, pages 75–81, 1978.

[70] S.M. Rump. How reliable are results of computers. *Jahrbuch Uberblicke Mathematik*, pages 163–168, 1983.

[71] R.M. Russel. The CRAY-1 computer system. *Communications of the ACM*, 21(1) :63–72, January 1978.

[72] M.J. Schulte, K.C. Bickerstaff, and E.E. Swartzlander, Jr. Hardware Interval Multipliers. *Journal of Theoretical and Applied Informatics*, 3(2) :73–90, 1996.

[73] M.J. Schulte and E.E. Swartzlander Jr. Hardware Design and Arithmetic Algorithms for a Variable-Precision, Interval Arithmetic Coprocessor. *Proceedings of the 12th Symposium on Computer Arithmetic*, pages 163–171, 1995.

[74] J. Schwartz. Implementing Infinite Precision A rithmetic. *Proceedings of the 9th Symposium on Computer Arithmetic*, pages 10–17, 1989.

[75] E. M. Schwarz, R. M. Smith, and C. A. Krygowski. The S/390 G5 Floating Point Unit Supporting Hex and Binary Architectures. In *Proceedings of the 14th Symposium on Computer Arithmetic*, pages 258–265, 1999. IEEEE Computer Society Press.

[76] J. Soto. Statistical Testing of Random Number Generators. In *22nd National Information Systems Security Conference*, October 99.

[77] SPARC International, Inc., San Jose, California. *The SPARC Architecture Manual version 9*, david l. weaver / tom germond edition. Published by PTR Prentice Hall, Englewood Cliffs, New Jersey 07632.

[78] H. Suzuki, H. Morinaka, H. Makino, Y. Nakase, K. Mashiko, and T. Sumi. Leading-Zero Anticipatory Logic for High-Speed Floating Point Addition. *IEEE journal of Solid-State Circuits*, 31(8) :1157–1164, August 1996.

[79] J.E. Thornton. *Design of a Computer. The Control Data 6600.* Glenview, Illinois, 1970.

[80] N. Vaucher. *Méthodologie de conception d'architectures VLSI génériques appliquée au traitement numérique.* PhD thesis, Université Pierre et Marie Curie, 20 June 1997.

[81] J. Vignes. Zéro mathématique et zéro informatique. In *La vie des Sciences, C.R. Acad. Sci.*, number 1 in 4, pages 1–13. Paris, January 1987.

[82] J. Vignes. Review on stochastic approach to round-off error analysis and its applications. *Math. Comp. Simul.*, 30 :481–491, 1988.

[83] J. Vignes. A stochastic arithmetic for reliable scientific computation. *Math. Comp. Simul.*, 35 :233–261, 1993.

[84] J. Vignes and M. La Porte. Error analysis in computing. In *Information Processing 74*, North-Holland, 1974.

[85] J. Vignes and V. Ung. Arrangement for determining number of exact significant figures in calculated result, 4 January 1983. US Patent 4,367,536.

[86] J. Vignes and V. Ung. Method and apparatus of providing a result of a numerical calculation with the number of exact significant figures, 31 May 1983. US Patent 4,386,413.

[87] Virtual Sockets Interface Alliance. *VSI ALliance Virtual Component Interface Standard*, ocb design working group edition, November 2000.

[88] J.E. Volder. The CORDIC trigonometric computing technique. *IRE Transactions on Electronic Computers*, EC-8 :330–334, September 1959.

[89] J.S. Walther. A unified algorithm for elementary functions. In *Joint Computer Conference Proceedings*, pages 379–385, 1971.

[90] Wolff von Gudenberg. Hardware Support for Interval Arithmetic. In G. Alefeld, A. Frommer, and B. Lang, editors, *Scientific Computing and Validated Numerics*, pages 32–37. Akademie Verlag, 1996. Extra Edition.

[91] S. Wolfram. *Mathematica, a system for doing mathematics by computer.* 1988.

Publications personnelles

Conférences internationales avec comité de lecture et actes

Hardware implementation of a method to control round-off errors
Chotin Roselyne, Mehrez Habib
6th WSEAS International Multiconference on Circuits Systems Communications and Computers (CSCC'2002), Rethymno, Crete, Grèce, July 2002, pp. 157-162

A Floating-Point Unit using stochastic arithmetic compliant with the IEEE-754 standard
Chotin Roselyne, Mehrez Habib
9th IEEE International Conference on Electronics, Circuits and Systems (ICECS'2002), Dubrovnik, Croatie, Septembre 2002, pp. 603-606

Hardware implementation of discrete stochastic arithmetic
Avot-Chotin Roselyne, Mehrez Habib
6th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS'2003), Poznan, Pologne, Avril 2003, pp. 57-64

Use of Redundant Arithmetic on Architecture and Design of a High Performance DCT Macro-bloc Generator
Chotin Roselyne, Dumonteix Yannick, Mehrez Habib
15th Design of Circuits and Integrated Systems Conference (DCIS), Montpellier, France, November 2000, pp. 428-433

Colloques internationaux avec comité de lecture

Hardware implementation of the CESTAC method

Chotin Roselyne, Mehrez Habib

10th GAMM - IMACS International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics (SCAN'2002), Paris, France, septembre 2002

Colloques nationaux avec comité de lecture et actes

Une unité de calcul flottant utilisant l'arithmétique stochastique

Chotin Roselyne, Mehrez Habib

Vèmes Journées Nationales du Réseau Doctoral de Micro-électronique (JNRDM'2002), Grenoble, France, avril 2002, pp. 217-218

Implantation matérielle d'une méthode de contrôle des erreurs d'arrondi de calcul

Chotin Roselyne, Mehrez Habib

Troisième colloque du GDR CAO de circuits et systèmes intégrés, Paris, France, Mai 2002, pp. 63-66

Articles soumis

Hardware implementation of discrete stochastic arithmetic

Avot-Chotin Roselyne, Mehrez Habib

Special Issue of Numerical Algorithms, Kluwer Academic Publishers

On the computation of the CESTAC function

Avot-Chotin Roselyne, Chesneaux Jean-Marie, Lamotte Jean-Luc

5th Conference on Real Numbers and Computers (RNC'5), Lyon, France, Septembre 2003