# Adéquation arithmétique architecture, problèmes et étude de cas

Arnaud Tisserand

N° d'ordre : 66

# THÈSE

*présentée devant*

**L'ÉCOLE NORMALE SUPÉRIEURE DE LYON**

pour obtenir

**le Titre de Docteur de l'École Normale Supérieure de Lyon**
**spécialité : Informatique**
**au titre de la formation doctorale d'Informatique de Lyon**

par ARNAUD TISSERAND

# ADÉQUATION ARITHMÉTIQUE ARCHITECTURE
## PROBLÈMES ET ÉTUDE DE CAS

Soutenue le 25 septembre 1997

Après avis de : LUIGI DADDA
DANIEL ETIEMBLE

Devant la commission d'examen formée de :

| | |
|---|---|
| LUIGI | DADDA |
| DANIEL | ETIEMBLE |
| JEAN-MICHEL | MULLER (Directeur de thèse) |
| MARC | RENAUDIN |
| YVES | ROBERT |

Aux zglshs et au papa des prostons
A mes amis
A ma famille

# Remerciements

Tout d'abord, je tiens à remercier les membres du jury :

Monsieur Luigi Dadda, Professeur au *Politecnico di Milano*, pour avoir accepté d'être rapporteur sur cette thèse. Son autorité et son expérience en arithmétique des ordinateurs furent des aides précieuses pour la correction du manuscrit.

Monsieur Daniel Etiemble, Professeur à l'Université Paris Sud, qui fut un rapporteur avisé. Sa grande compétence et sa clairvoyance sur les aspects des architectures des machines m'ont permis de bien mettre en valeur certains points de mon travail.

Monsieur Marc Renaudin, Maître de Conférence à Télécom Bretagne, pour son intérêt et son enthousiasme durant nos nombreuses réunions de travail et son aide durant la rédaction de la thèse. Pouvoir travailler avec des chercheurs de grande qualité est l'un des points les plus motivants pour un jeune chercheur. J'espère donc pouvoir continuer de travailler longtemps avec lui.

Monsieur Yves Robert, Professeur à l'Ecole Normale Supérieure de Lyon, pour son aide et sa disponibilité tout au long de ma formation à la recherche et de mes travaux de thèse.

Monsieur Jean-Michel Muller, Chargé de Recherche au CNRS, pour avoir accepté d'encadrer ma thèse. Ces quelques années passées au côté de Jean-Michel furent très enrichissantes professionnellement et personnellement. Il est un chercheur extrêmement brillant et un directeur de thèse formidable, à un point tel qu'il n'est pas possible de souhaiter meilleur encadrement pour une thèse.

Ensuite, je voudrais remercier chaleureusement toutes les personnes rencontrées ces dernières années : Bernard, Thierry, Richard, Olivier, Laure, Sandrine, Patrick, Bruno, Daniel, Olivier, Mathilde, Jean-Philippe, Xavier, Catherine, Martin, Laurent-Stéphane, Jean-Claude, Asger, Frédéric, Jean-Marc, Jean-Marc, Jacques-Olivier, Olivier, Johanne, Loïc, Laurent, Blaise, David, Anne, Bertrand, François, Frédéric, Pascal, Pierre, Philippe, Alain, Valérie, Sylvie, Jocelyne, Marie, Christelle, Eliane, Marie-Jeanne....

Enfin, pour finir, je tiens à remercier du fond du coeur ma famille qui m'a soutenue durant toutes ces années.

# Table des matières

# Liste des figures

# Liste des tables

# Introduction

LES premières générations de processeurs n'étaient capables d'évaluer matériellement que des additions et des multiplications (et encore, ces dernières étaient souvent obtenues par des additions successives), les autres opérations comme la division, la racine carrée et les *fonctions élémentaires* (sinus, cosinus, exponentielle...) étaient alors évaluées par des routines logicielles.

Les importants besoins en calcul scientifique et dans les applications spécialisées ont poussé les constructeurs de machines à intégrer de plus en plus de fonctionnalités arithmétiques directement dans leurs circuits intégrés pour obtenir de plus grandes performances. Les processeurs actuels sont des véritables "dévoreurs de nombres", leurs performances s'expriment en centaines de MFlops (Millions d'instructions flottantes par seconde). Par exemple, un processeur DEC Alpha 21164 est capable de commencer au moins une multiplication flottante et une addition flottante sur 53 bits à chaque cycle d'horloge à une fréquence de quelques centaines de MHz.

Les performances des unités de calcul de ces processeurs témoignent de l'effort de conception investi dans la réalisation de machines pour le calcul à hautes performances. En effet, il est aujourd'hui possible d'effectuer plusieurs millions d'additions, de multiplications, de divisions, de racines carrées, de sinus, de cosinus..., en quelques secondes avec une machine coûtant moins de 10 000 F.

Ces importantes améliorations des performances des unités de calcul (sur des nombres entiers et réels) ont été rendues possibles grâce aux recherches effectuées dans de nombreux domaines, dont les principaux sont :

- La micro-électronique avec la diminution de la taille des gravures (technologies $0.35, 0.25\,\mu$m), l'augmentation des fréquences d'horloge, les optimisations des composants...
- L'architecture des ordinateurs avec des techniques de plus en plus sophistiquées de hiérarchie mémoire (les caches), de parallélisme (pipeline et multiplication des unités fonctionnelles), de prédictions dans les branchements...
- L'arithmétique des ordinateurs avec la mise au point et l'utilisation d'algorithmes de calcul et de systèmes de représentation des nombres de plus en plus performants.

La littérature en arithmétique des ordinateurs montre qu'en changeant notre façon de représenter les nombres et/ou en utilisant des algorithmes spécifiques de calcul, il est possible de réaliser des opérateurs matériels particulièrement efficaces. Dans cette thèse, nous nous sommes attachés à étudier certains liens qui existent entre l'arithmétique des ordinateurs et l'architecture des machines. Dans ce cadre, nous avons étudié les points suivants :

- L'arrondi exact des fonctions élémentaires ;
- Le système semi-logarithmique de représentation des nombres ;
- Le calcul en-ligne sur FPGA ;
- Les opérateurs arithmétiques asynchrones.

### L'Arrondi Exact des Fonctions Elémentaires

Le résultat d'un calcul dans le système virgule flottante (système classique pour la manipulation des nombres réels) n'est généralement pas représentable exactement en machine du fait de la représentation des nombres avec une précision limitée. Les calculs sont donc arrondis (le résultat retourné est l'un des deux nombres représentables en machine qui encadrent le résultat théorique). L'*arrondi exact* d'une opération arithmétique est obtenu en utilisant une technique qui permet de déterminer le résultat de l'opération comme si le calcul était effectué exactement, avec une précision infinie, puis arrondi. Les opérations flottantes de base (addition, soustraction, multiplication et division) sont exactement arrondies dans la plupart des machines actuelles (norme IEEE 754). L'arrondi exact des calculs permet de une meilleure portabilité des programmes ou de concevoir des preuves de certains algorithmes numériques par exemple.

Pour le moment, il n'existe pas de pareille exigence pour les *fonctions élémentaires* (sinus, cosinus, exponentielle, logarithme...). En effet, l'arrondi exact des fonctions élémentaires est un problème que l'on a pensé pratiquement insoluble pendant longtemps. Il consiste à déterminer la précision intermédiaire permettant de toujours pouvoir arrondir exactement les résultats du calcul des fonctions élémentaires. On sait depuis longtemps (1885) que le nombre de chiffres intermédiaires nécessaires à la réalisation de cet arrondi exact est fini, mais les grandes valeurs ($10^{244}$ bits en utilisant un résultat de 1975) trouvées jusqu'à présent laissaient penser que cet arrondi n'était pas pratiquement réalisable.

Nous présentons dans cette thèse, une méthode permettant d'arrondir exactement les fonctions élémentaires sans changer significativement le temps moyen de calcul de ces fonctions.

### Le Système Semi-Logarithmique de Représentation des Nombres

Pour certaines applications, il est possible d'accélérer considérablement les calculs en utilisant des systèmes de représentation des nombres différents du système usuel. Par exemple, les systèmes redondants permettent d'effectuer des additions en temps constant. Ces systèmes sont utilisés dans la plupart des multiplieurs et des diviseurs actuels du fait du gain de vitesse qu'ils représentent.

Dans le système logarithmique, on représente les nombres réels par leur signe et le logarithme de leur valeur absolue écrite en virgule. Ce système permet d'effectuer très rapidement des multiplications et des divisions. Le système logarithmique s'est avéré très performants dans certaines applications en contrôle numérique et en traitement du signal et des images. Toutefois, l'implantation matérielle d'unités de calcul fonctionnant dans le système logarithmique n'est pas possible actuellement pour des nombres de taille supérieure à 24 bits. En effet, les opérations d'addition et de soustraction dans le système logarithmique nécessitent des tables à $2^n$ entrées où $n$ est le nombre de chiffres des nombres.

Nous proposons dans cette thèse un nouveau système de représentation des nombres réels baptisé *système semi-logarithmique*. C'est un système à mi-chemin entre le système virgule flottante et le système logarithmique. Nous présentons des algorithmes qui permettent d'effectuer les opérations de base (addition, soustraction, multiplication et division) dans ce système, ainsi que des algorithmes pour effectuer des conversions depuis et vers le système virgule flottante. En particulier, nous montrons en quoi ce système permet d'effectuer beaucoup plus rapidement des additions et des soustractions que le système logarithmique. Enfin, nous comparons la qualité de la représentation de ce système avec celle des systèmes virgule flottante et logarithmique.

### Arithmétique En-Ligne sur FPGA

L'*arithmétique en-ligne* permet de réaliser des architectures de calcul où les chiffres circulent en série, chiffre après chiffre, les poids forts en tête. L'intérêt de cette arithmétique est de pouvoir calculer toutes les fonctions (en arithmétique série poids faibles en tête, il n'est pas possible de calculer certaines opérations comme la division ou le maximum) et d'obtenir des opérateurs de petite taille avec un nombre d'entrées/sorties plus faible que leur équivalents parallèles.

Nous présentons dans cette thèse la réalisation d'une bibliothèque d'opérateurs en-ligne pour des implantations sur des composants programmables FPGA (*field programmable gate arrays*). Cette association d'une arithmétique permettant de réaliser des opérateurs ayant une petite surface de circuit et très performants et d'un support matériel flexible semble être particulièrement efficace pour des applications de contrôle embarqué (collaboration avec l'équipe aérospatiale du *Centre Suisse d'Electronique et de Microtechnique* de Neuchatel et le département d'automatique de l'*Ecole Polytechnique Fédérale de Lausanne*) ou bien des applications de traitement du signal et des images (collaboration avec l'équipe de McCanny et Woods à la *Queen's University of Belfast* et l'équipe de M. Ercegovac à l'*University of California at Los Angeles*).

### Les Opérateurs Arithmétiques Asynchrones

Les *circuits intégrés asynchrones* semblent être de plus en plus une alternative efficace aux circuits intégrés synchrones pour la réalisation de systèmes de calcul à très hautes performances. En effet, les circuits asynchrones ne posent pas certains problèmes de conception de plus en plus difficiles à résoudre pour les circuits synchrones.

Les *opérateurs arithmétiques asynchrones* permettent de calculer les fonctions arithmétiques (addition, multiplication, division...) avec un délai variable. Dans cette thèse, nous avons développé des opérateurs arithmétiques asynchrones dont le temps moyen de calcul est plus faible que le temps de calcul des équivalents synchrones. En particulier, nous présentons dans ce document l'étude de différents additionneurs asynchrones dont le temps moyen de calcul est en $\mathcal{O}(\sqrt{\log n})$. Ces opérateurs semblent particulièrement efficaces pour des implantations matérielles à faible consommation pour des applications embarquées ou des appareils portables. Dans ce domaine, nous étudions en collaboration avec de l'équipe de M. Renaudin au CNET les différents aspects liés à l'implantation et à l'utilisation de ces opérateurs.

# Arrondi exact des fonctions élémentaires

CHAPITRE

# 1 Arrondi exact des fonctions élémentaires

LES calculs effectués en machine sur les nombres réels sont en général entachés d'erreurs du fait de la précision limitée de la représentation de ces nombres dans l'ordinateur. Par exemple, en base 10, le résultat de la division $\frac{10.0}{3.0}$ ne peut pas s'écrire sur un nombre fini de chiffres. On procède alors à une phase d'*arrondi*. C'est-à-dire que l'on décide de choisir l'un des deux nombres représentables qui encadrent le résultat exact. Différents modes d'arrondi sont possibles, comme l'arrondi par excès ou l'arrondi au plus près.

L'*arrondi exact* d'une opération est obtenu en utilisant des techniques qui permettent de déterminer le résultat comme si le calcul était effectué exactement, avec une précision "infinie", puis arrondi suivant le mode d'arrondi choisi. Les opérations qui répondent à cette spécification sont dites *exactement arrondies*.

La plupart du temps, les nombres réels sont représentés en machine en utilisant la *norme IEEE 754*. Cette norme spécifie le codage des nombres réels au niveau du bit. Elle spécifie aussi le codage de valeurs spéciales comme les dépassements de capacité et les résultats d'opérations indéterminés (comme le résultat de $\sqrt{-1}$, celui de $\frac{0}{0}$ ou celui de $\infty - \infty$). Ceci permet une compatibilité totale lors des échanges de données entre les machines. La norme IEEE 754 spécifie différents modes d'arrondi valides dans tous les formats de la norme (simple précision, double précision, formats étendus...).

Cette spécification conjointe du codage des données (nombres réels et valeurs spéciales) et des modes d'arrondi permet une spécification complète des opérations de base (addition, soustraction, multiplication, division et racine carrée). C'est probablement le point le plus important de la norme IEEE 754. En effet, les opérations ne sont plus dépendantes de leur réalisation pratique car leur comportement est complètement spécifié dans le cadre d'une véritable structure mathématique. On a alors une compatibilité totale des algorithmes d'une machine à l'autre. C'est-à-dire qu'un algorithme (où l'ordre des opérations est déterminé) donnera exactement le même résultat sur différentes machines compatibles avec la norme IEEE 754. Il y a aussi une compatibilité des codes sources correspondants à ces algorithmes, à condition que les compilateurs aient des comportements identiques sur les différentes plateformes de développement. Les contraintes mathématiques ainsi introduites peuvent permettre de réaliser des preuves sur les algorithmes numériques employés.

L'arrondi exact des fonctions arithmétiques (addition, soustraction et multiplication) et des fonctions algébriques (division et racine carrée) est toujours assez facilement réalisable en machine. En effet, pour ces opérations on sait déterminer la précision intermédiaire nécessaire pour arrondir exactement le résultat quelle que soit la précision finale demandée. On a cru pendant très longtemps que cet arrondi exact n'était pas pratiquement réalisable pour les fonctions élémentaires ($\sin, \cos, \exp, \log, \arctan \ldots$). La précision nécessaire lors des calculs intermédiaires pouvait être, pensait-on, si importante qu'elle interdit totalement une mise en œuvre effective de l'arrondi exact de ces fonctions pourtant si utiles en calcul scientifique.

Nous allons montrer dans ce chapitre que l'arrondi exact des fonctions élémentaires est tout à fait réalisable en pratique et que son coût n'est pas forcément si élevé que ce que l'on pouvait penser *a priori*. En effet, les tests que nous avons effectués montrent que les cas qui conduisent à des calculs intermédiaires en très grande précision sont rarissimes. Nous montrerons que dans la quasi totalité des cas, l'arrondi exact des fonctions élémentaires peut être réalisé en machine avec des techniques proches de celles de l'arrondi des opérations de base. Nous verrons aussi que dans les cas les plus extrêmes, cet arrondi est toujours possible et que la détection des cas "pathologiques" associés est très simple.

Dans la première partie de ce chapitre, nous présentons les différents aspects de la norme IEEE 754. Les principales caractéristiques de cette norme nous serviront tout au long de cette thèse. Nous décrirons en particulier l'arrondi des opérations de base afin de pouvoir effectuer des analogies et des comparaisons avec celui des fonctions élémentaires par la suite.

La deuxième partie de ce chapitre est dédiée au problème de l'arrondi exact des fonctions élémentaires. Dans ce cadre, nous présenterons des tests effectués sur certains formats de nombres flottants et une modélisation probabiliste du problème qui permettent de bien appréhender les mécanismes sous-jacents à ce problème. Nous rappellerons des résultats de théorie des nombres qui nous permettront de donner des bornes supérieures du temps d'exécution de nos algorithmes.

La dernière partie propose une ébauche de réflexion sur la mise en œuvre pratique (matérielle et/ou logicielle) de l'arrondi exact des fonctions élémentaires.

## 1.1 La norme IEEE 754

Nous présentons ici les principales caractéristiques de la norme IEEE 754 qui spécifie la représentation des nombres réels en "virgule flottante" , le comportement de certaines opérations arithmétiques et des conversions (conversions entre les différents formats). Cette présentation de la norme IEEE 754 nous servira dans d'autres parties de cette thèse.

Une bonne introduction sur l'arithmétique flottante du point de vue de l'utilisateur se trouve dans [Gol91]. Les bases de l'implantation matérielle des opérateurs flottants sont décrites dans [HP96]. Une étude générale plus détaillée des opérations arithmétiques et de leur réalisation matérielle se trouve dans [Omo94].

### 1.1.1 Codage des nombres flottants dans la norme IEEE 754

La norme IEEE 754, établie en 1985 (cf [AI85]), spécifie la représentation interne des nombres réels en arithmétique flottante en base 2, ainsi que le comportement des opérations portant sur ces nombres (opérations mathématiques et conversions entre les différents formats). La norme IEEE 854 (cf [AI87]), rédigée par le même comité, fixe l'implantation de l'arithmétique flottante en base 2 et en base 10 et autorise de plus grandes libertés au niveau de la définition des formats des nombres. Dans la suite, nous ne nous intéresserons qu'à la norme IEEE 754 car c'est-t-elle qui est implantée sur la plupart des ordinateurs actuels.

Un nombre réel $x$ est codé en arithmétique virgule flottante (en base 2) selon le codage :

$$x = s_x \times M_x \times 2^{e_x}$$

où

- $s_x$ est le *signe* de $x$. Plus précisément, le bit de signe $s$ est égal à 0 si $x \geq 0$ et 1 sinon, on a

alors $s_x = (-1)^s$. Les nombres flottants sont stockés selon le codage (signe, valeur absolue), la valeur absolue $M_x 2^{e_x}$ étant stockée dans les deux champs suivants.

- $e_x$ est l'*exposant* de $x$. Il peut avoir n'importe quelle valeur entière dans l'intervalle $[-e_{max}+1, e_{max}]$. En fait, on ne représente pas directement l'exposant mais l'exposant plus un biais de sorte que le champ exposant représente $e_x + e_{max}$. Ceci permet de représenter simplement les exposants négatifs sans recours à une notation spéciale comme le complément à la base et permet d'effectuer facilement des comparaisons entre deux nombres flottants. En effet, l'ordre des bits dans les champs (signe, exposant, mantisse), voir la Figure 1.1, pour les valeurs positives, coïncide avec l'ordre lexicographique. Ceci permet d'effectuer des comparaisons sur les nombres flottants en faisant comme si il s'agissait d'entiers. Les entrées du comparateur entier sont en complément à 2 ce qui permet d'effectuer directement la comparaison. Le plus petit et le plus grand exposant (biaisé) sont réservés pour le codage de valeurs spéciales (cf Tableau 1.2). Par exemple, en simple précision l'exposant biaisé est dans l'intervalle [1,254], les exposants 0 et 255 sont réservés pour le codage des valeurs spéciales. La valeur de $e_{max}$ et la taille du champ exposant dépendent du type utilisé (cf Tableau 1.1).

- $M_x$ est la *mantisse* de $x$. Le nombre de chiffres de la mantisse étant limité, l'introduction de zéros à gauche de cette mantisse a tendance à diminuer la précision de la représentation. Par exemple, avec un système ayant 8 bits de mantisse, si le résultat exact d'une multiplication est 0.00000010101100, alors la réécriture de ce résultat sur 8 bits dans le format cible entraîne une importante perte de précision car les derniers bits du résultat sont alors "perdus". Même si à chaque opération, l'erreur commise n'est pas très importante devant le résultat de l'opération, il est possible que lors d'une séquence d'opérations le résultat final ne soit plus significatif tant l'accumulation des erreurs est importante. Ainsi, on exige que le premier chiffre de la mantisse soit non nul. On parle alors de nombres *normalisés* ($1 \leq M_x < 2$). En base 2, le premier chiffre de la mantisse est donc obligatoirement égal à 1. En pratique ce bit, appelé *bit implicite*, n'est pas stocké physiquement (sauf dans les formats "étendus"). Le fait que les mantisses doivent être supérieures ou égales à 1 implique que le nombre 0 doit être codé d'une façon spéciale (cf Tableau 1.2). On appelle *fraction* `f` les bits de la mantisse effectivement stockés (sans le bit implicite). La taille du champ correspondant à la mantisse dépend du type utilisé (cf Tableau 1.1). Par exemple, en double précision la mantisse est de 53 bits mais seuls les 52 derniers sont effectivement stockés.

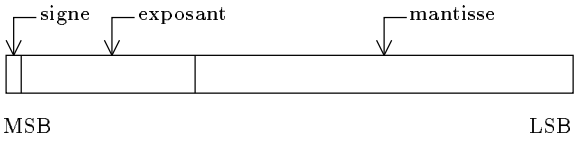



FIG. 1.1 – *Codage normalisé des nombres flottants.*

**Exemple 1.1** *Le nombre flottant simple précision (32 bits) :*

| 1 | 10000001 | 01010000000000000000000 |
|---|---|---|

*représente la valeur (les bits soulignés sont ceux des différents champs du nombre flottant) :*

$$(-1)^{\underline{1}} \times 1.\underline{01010000000000000000000} \times 2^{\underline{10000001}-127} = -1.3125 \times 2^2 = -5.25$$

Le Tableau 1.1 donne la taille des différents champs et la valeur du biais pour chacun des formats définis par la norme IEEE 754.

| format | mantisse (bits) | exposant (bits) | biais | longueur totale (bits) |
|---|---|---|---|---|
| `simple` | 1+23 | 8 | 127 | 32 |
| `simple étendu` | $\geq 32$ | $\geq 11$ | $\geq 1023$ | $\geq 40$ |
| `double` | 1+52 | 11 | 1023 | 64 |
| `double étendu` | $\geq 64$ | $\geq 15$ | $\geq 16383$ | $\geq 80$ |
| `double étendu PC` | 1+64 | 15 | 16383 | 80 |
| `quad` | 112 | 15 | 16383 | 128 |

TAB. 1.1 – *Caractéristiques des formats flottants IEEE 754.*

Du fait du nombre limité de chiffres, tous les réels ne sont pas représentables. Notons $\mathbb{F}$ l'ensemble des nombres réels représentables exactement dans le format utilisé.

En plus des valeurs numériques classiques (les éléments de $\mathbb{F}$), la norme IEEE 754 définit d'autres catégories de valeurs pour gérer "proprement" les résultats d'opérations indéterminés (ex: division par 0), ainsi que les dépassements de capacité. Ces valeurs spéciales sont :

- $\{-\infty, +\infty\}$ Ces valeurs infinies représentent à la fois un dépassement de capacité positif ou négatif et les résultats respectifs des opérations $\frac{a}{0^+}$ et $\frac{a}{0^-}$ où $a \in \mathbb{F}_+^*$.
- $\{0^+, 0^-\}$ L'existence de deux codages différents pour 0 est la conséquence de la définition des deux valeurs $-\infty$ et $+\infty$. On a $0^+ = (-1) \times 0^-$; et $0^+ = 0^-$ est vrai même si les codages de $0^+$ et de $0^-$ sont distincts.
- `NaN` (Not a Number) Cette valeur est utilisée pour coder les résultats indéterminés comme $\sqrt{-1}$ et les résultats qui ne peuvent pas être réduits comme $\frac{0}{0}$ ou $\infty - \infty$.
- **les nombres dénormalisés** Le nombre 0 est codé par l'ensemble des bits du mot machine à 0 (sauf éventuellement le bit de signe). L'exposant non biaisé de 0 est donc $-e_{max}$ soit le plus petit exposant possible. Sur la Figure 1.2, on peut remarquer qu'il y a un saut important entre le plus petit nombre (en valeur absolue) représentable (de manière "normalisée") non nul et 0. On utilise les mots dont l'exposant est $-e_{max}$ sans le "1" implicite pour obtenir un spectre plus régulier au voisinage de 0. Ce sont les nombres *dénormalisés*.

Le Tableau 1.2 donne le codage correspondant à chacune des catégories de valeurs (numériques ou spéciales) représentables dans la norme IEEE 754.

| signe | exposant | mantisse | signification |
|---|---|---|---|
| $\pm$ | $-e_{max} < e_x \leq e_{max}$ | `f` | $\pm 1.\mathtt{f} \times 2^{e_x}$ |
| $\pm$ | $e_x = -e_{max}$ | $\mathtt{f} \neq 0$ | $\pm 0.\mathtt{f} \times 2^{-e_{max}+1}$ |
| $\pm$ | $e_x = -e_{max}$ | $\mathtt{f} = 0$ | $\pm 0$ |
| $\pm$ | $e_x = e_{max} + 1$ | $\mathtt{f} = 0$ | $\pm\infty$ |
| | $e_x = e_{max} + 1$ | $\mathtt{f} \neq 0$ | `NaN` |

TAB. 1.2 – *Valeurs représentables dans les formats IEEE 754.*

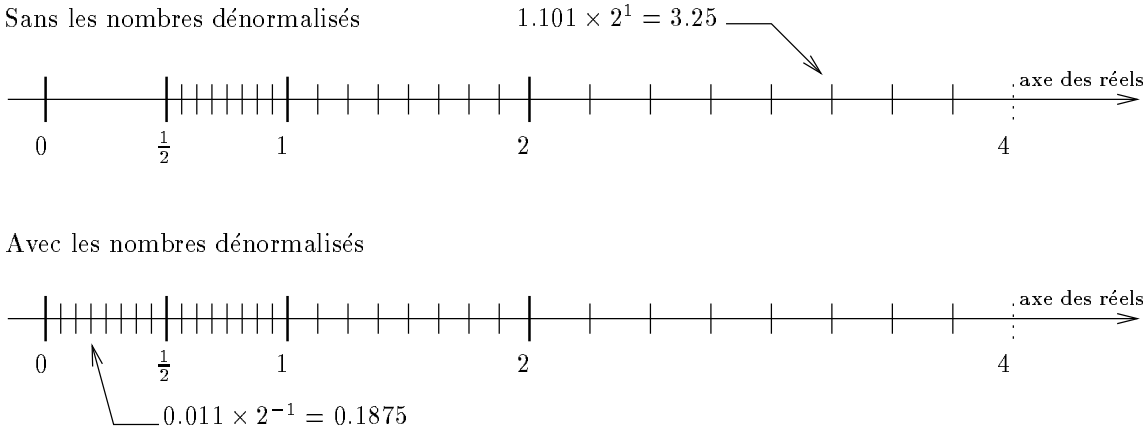



FIG. 1.2 – *Nombres positifs représentables en arithmétique flottante sur 3 bits de mantisse et 2 bits d'exposant avec et sans les nombres dénormalisés.*

La gestion de l'apparition des valeurs spéciales peut se faire de deux façons différentes. Une première solution consiste à configurer l'unité flottante pour que l'exécution d'une opération problématique déclenche une interruption. La gestion se fait alors avec un mécanisme d'exception (à condition que la machine, le système d'exploitation et/ou le langage de programmation utilisés permettent de gérer effectivement ces exceptions). La deuxième solution consiste à ne pas interrompre le calcul même après une opération illégale. Le résultat retourné à l'utilisateur sera alors cohérent avec la spécification IEEE 754 des opérations mises en jeu. Le comportement des opérations est complètement spécifié vis à vis des valeurs spéciales. Par exemple, la valeur spéciale `NaN` se propage lors des calculs. Le Tableau 1.3 présente la spécification complète du résultat de l'addition flottante en fonction des valeurs de ses opérandes. On peut trouver dans [AI85, Dau96] les spécifications correspondantes pour les autres opérations.

| addition | $-\infty$ | $a \in \mathbb{F}^*_-$ | $0^-$ | $0^+$ | $a \in \mathbb{F}^*_+$ | $+\infty$ | `NaN` |
|---|---|---|---|---|---|---|---|
| $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | `NaN` | `NaN` |
| $b \in \mathbb{F}^*_-$ | $-\infty$ | $a+b$ ou $-\infty$ | $b$ | $b$ | $a+b$ | $+\infty$ | `NaN` |
| $0^-$ | $-\infty$ | $a$ | $0^-$ | $0^+$ | $a$ | $+\infty$ | `NaN` |
| $0^+$ | $-\infty$ | $a$ | $0^+$ | $0^+$ | $a$ | $+\infty$ | `NaN` |
| $b \in \mathbb{F}^*_+$ | $-\infty$ | $a+b$ | $b$ | $b$ | $a+b$ ou $+\infty$ | $+\infty$ | `NaN` |
| $+\infty$ | `NaN` | $+\infty$ | $+\infty$ | $+\infty$ | $+\infty$ | $+\infty$ | `NaN` |
| `NaN` | `NaN` | `NaN` | `NaN` | `NaN` | `NaN` | `NaN` | `NaN` |

TAB. 1.3 – *Spécification IEEE 754 de l'addition flottante.*

### 1.1.2 Modes d'arrondi IEEE 754

La norme IEEE 754 donne, en plus des formats et des codages des valeurs numériques et spéciales, une spécification des modes d'arrondi afin de garantir la reproductibilité des calculs. Les opérations ne sont donc plus définies en fonction de caractéristiques physiques de l'architecture de calcul utilisée mais par un ensemble de contraintes mathématiques. C'est l'un des points important de la norme. En effet, pour une précision fixée (un format), un même calcul donne le même résultat

sur toutes les machines compatibles avec la norme. Les contraintes mathématiques ainsi introduites peuvent être utilisées pour la validation et la réalisation de preuves de certains algorithmes numériques.

On appelle *nombre machine* un nombre qui peut être représenté exactement dans le système virgule flottante utilisé. $\mathbb{F}$ est l'ensemble des nombres machine. En général, la somme, le produit ou le quotient de deux nombres machine n'est pas un nombre machine. Le résultat de l'opération doit être arrondi.

La norme définit 4 modes d'arrondi :

- vers $-\infty$ : $\nabla(x)$ est le plus grand nombre machine inférieur ou égal à $x$;
- vers $+\infty$ : $\triangle(x)$ est le plus petit nombre machine supérieur ou égal à $x$;
- vers 0 : $\mathcal{Z}(x)$ vaut $\nabla(x)$ si $x \geq 0$ et $\triangle(x)$ sinon;
- au plus près : $\mathcal{N}(x)$ est le nombre machine le plus proche de $x$. Si $x$ est exactement entre deux nombres machine, la norme impose que l'arrondi soit le nombre machine dont le bit de poids faible de la mantisse est nul.

La norme IEEE 754 impose que le résultat d'une opération arithmétique ($+, -, \times$ ou $\div$) ou de la racine carrée soit l'*arrondi exact* de l'opération. Si $\diamond$ est le mode d'arrondi actif, alors le résultat de l'opération $x \star y$ (où $\star$ est $+, -, \times$ ou $\div$) ou $\sqrt{x}$ doit toujours être $\diamond(x \star y)$ ou $\diamond(\sqrt{x})$. L'unité flottante se comporte alors comme si le calcul était effectué exactement, avec une précision infinie, puis arrondi.

Les opérations qui vérifient cette propriété sont dites *exactement arrondies*. Cette propriété permet de garantir une compatibilité totale. C'est-à-dire, un même programme donnera les mêmes résultats sur toutes les machines compatibles avec la norme (si les compilateurs ne changent pas l'ordre des instructions). De plus, les opérations flottantes étant complètement spécifiées, on dispose d'une véritable structure mathématique qui peut permettre de réaliser des preuves des algorithmes numériques utilisés.

Le choix du mode d'arrondi peut être utilisé pour faire de l'arithmétique d'intervalles. On peut, en effet, calculer $[\nabla(x), \triangle(x)]$ à chaque étape du calcul et ainsi obtenir un encadrement certain du résultat d'une séquence d'opérations [Kul77, KM81].

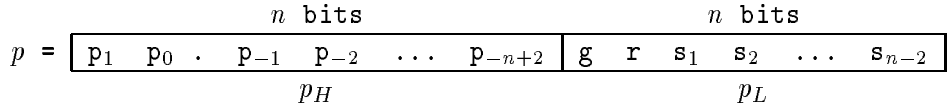
Pour le moment, il n'existe pas de spécification des fonctions élémentaires (sin, cos, exp, log, arctan...) dans la norme IEEE 754 (ou 854). Ceci est dû au fait que pendant longtemps on a cru que l'arrondi exact de ces fonctions n'était pas réalisable à un coût raisonnable. Nous montrerons dans la suite de ce chapitre que l'arrondi exact des fonctions élémentaires est effectivement réalisable, et qu'il est temps de songer à une extension de la norme actuelle à ces fonctions.

### 1.1.3 Réalisation matérielle de l'arrondi exact des fonctions de base

Nous décrivons ici les grandes lignes de l'implantation matérielle de l'arrondi exact des fonctions arithmétiques (addition, multiplication) et évoquerons le cas de la division et de la racine carrée dans le cadre de la norme IEEE 754. Les algorithmes des opérations flottantes et les principes de base de l'arrondi exact de ces fonctions sont détaillés dans [Omo94]. Une bonne introduction sur ce sujet est faite par Goldberg dans [HP96]. Des algorithmes visant des implantations matérielles efficaces sont décrits dans [SBH89] et dans [QTF91]. Enfin, on trouve dans [YZ95] et [PZ95] la description de l'unité flottante du processeur `UltraSPARC` pour la multiplication et celle de la division et de la racine carrée, y compris la réalisation pratique des arrondis de ces fonctions.

### Multiplication

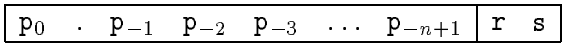
Soient $a$ et $b$ les deux nombres flottants à multiplier. On note $n$ le nombre de bits de la mantisse de $a$ et de $b$. Le produit $p = a \times b$ est donc composé de $2n$ bits et on a $1 \leq p < 4$. On décompose les bits de $p$ en deux mots $(p_H, p_L)$ de $n$ bits chacun suivant le schéma suivant :




De $p_L$, on extrait l'information suivante :

- `g` le bit de garde (*guard bit*) est le bit de poids fort de $p_L$
- `r` le bit d'arrondi (*round bit*) est le deuxième bit de $p_L$ à partir des poids forts
- `s` le bit persistant (*sticky bit*) qui est le `OU` logique des $n-2$ bits de poids faible de $p_L$

La mantisse du résultat devant être normalisée, il faut commencer par rechercher le premier bit non nul du résultat, en modifiant éventuellement l'exposant en conséquence. Le produit $p$ étant dans l'intervalle $[1, 4[$ seul le premier bit de poids fort de $p_H$ peut être non nul. Si ce bit est nul alors l'exposant est correct mais il faut effectuer un décalage à gauche des bits de $p_H$ et injecter `g` comme bit de poids le plus faible dans le nouveau $p_H$. Les bits `r` et `s` restent inchangés. Si le premier bit de $p_H$ est égal à 1 alors il faut incrémenter l'exposant, les bits `r` et `s` sont recalculés par les relations $\texttt{s} \leftarrow \texttt{s} \vee \texttt{r}$ et $\texttt{r} \leftarrow \texttt{g}$. Après cette phase permettant d'assurer la normalisation du résultat, on a un codage sur $n+2$ bits comme suit :




On a donc un nouveau $p_H$ et un nouveau $p_L$ notés respectivement $\hat{p}_H$ et $\hat{p}_L$ ($\hat{p}_L$ étant réduit à deux bits). Le but du bit de garde est de permettre la renormalisation sur $n$ bits de la mantisse du résultat. Ce sont maintenant les valeurs respectives des bits `r` et `s` qui vont nous permettre de calculer l'arrondi exact du résultat.

Par exemple, dans le cas de l'arrondi vers $+\infty$, pour les valeurs positives, il est juste nécessaire de savoir si l'un des bits de $\hat{p}_L$ est égal à 1. Si c'est le cas alors il faut ajouter 1 à la mantisse. Dans le cas contraire (tous valent 0), on a déjà l'arrondi vers le haut du résultat. Dans le cas de l'arrondi au plus près, les choses sont un peu plus complexes. En effet, il faut pouvoir distinguer les valeurs qui se trouvent de part et d'autre du milieu de l'intervalle formé par deux nombres machine consécutifs.

Le Tableau 1.4 résume les différents cas permettant de réaliser l'arrondi exact de la multiplication flottante IEEE 754.

Cette méthode permet d'implanter matériellement l'arrondi exact de la multiplication sur une surface relativement restreinte. En effet, il suffit d'un décaleur d'une position pour des mots de $n$ bits, d'un additionneur sur $n$ bits pour injecter les corrections du Tableau 1.4 et d'un petit arbre de cellules `OU` pour calculer le *sticky* bit. Cependant, cette méthode n'est pas efficace en temps de calcul. Les étapes rajoutées pour arrondir le résultat peuvent causer une augmentation du nombre d'étages du pipeline de l'unité flottante. Dans [SBH89], Santoro, Bewick et Horowitz proposent des algorithmes qui permettent de réaliser l'arrondi exact de la multiplication plus rapidement. Par exemple, ils proposent un algorithme tenant compte de la sommation des produits partiels en

| mode d'arrondi | $p \geq 0$ | $p < 0$ |
|---|---|---|
| $-\infty$ | | $+1$ si `r` $\vee$ `s = 1` |
| $+\infty$ | $+1$ si `r` $\vee$ `s = 1` | |
| 0 | | |
| au plus près | $+1$ si `(r` $\wedge$ `s)` $\vee$ `(`$p_{n-1}$ $\wedge$ `r) = 1` | $+1$ si `(r` $\wedge$ `s)` $\vee$ `(`$p_{n-1}$ $\wedge$ `r) = 1` |

TAB. 1.4 – *Règles permettant d'arrondir exactement la multiplication flottante (IEEE 754). La règle* $+1$ *signifie qu'il faut ajouter 1 à* $\hat{p}_H$ *pour obtenir l'arrondi exact de* $p$*. L'absence de règle indique que* $\hat{p}_H$ *est l'arrondi souhaité.*

notation redondante *carry-save* (chiffres dans $\{0, 1, 2\}$ en base 2) pour calculer le *sticky* bit. Ceci permet d'effectuer rapidement la correction finale (le $+1$ du Tableau 1.4) directement en *carry-save* (sans devoir convertir en notation classique d'abord). Ils proposent aussi différentes stratégies permettant de recouvrir, le plus possible, le calcul de l'arrondi et les autres phases du calcul du produit. Dans [QTF91], Quach, Takagi et Flynn proposent des techniques pour réaliser rapidement l'arrondi exact de la multiplication dans le cas de multiplieurs utilisant, en interne, une notation redondante à chiffres signés. Dans [YZ95], on trouve la description de la multiplication flottante (avec son arrondi) du processeur `UltraSPARC`.

### Addition

L'algorithme d'addition en virgule flottante est un peu plus complexe que celui de la multiplication. En effet, de nombreux cas de figures demandent des corrections plus ou moins complexes. Par exemple, la somme des deux opérandes peut s'écrire exactement sur beaucoup plus ou beaucoup moins de chiffres que $n$. Si, par exemple, l'un des opérandes est petit en valeur absolue devant l'autre (addition du style $1+\epsilon$), l'addition des mantisses alignées peut éventuellement s'écrire exactement sur un grand nombre de chiffres (supérieur à $n$). Dans ce cas, on peut utiliser une technique similaire à celle de la multiplication pour arrondir. Un autre cas qui complique l'algorithme de l'addition flottante est celui de la soustraction de deux quantités proches. Il se produit alors ce que l'on appelle une *cancellation*. C'est-à-dire qu'après la soustraction, les premiers chiffres du résultat sont nuls. Lors de la phase de renormalisation il est possible de devoir ajouter des 0 artificiels dans les poids faibles du résultat. Tous ces cas ne sont pas très complexes à prendre en compte, mais ils compliquent considérablement la logique de contrôle de l'opérateur et nécessitent une part non négligeable du temps de calcul de l'opérateur.

On peut trouver dans [HP96] une bonne introduction sur la réalisation de l'arrondi de l'addition en virgule flottante. L'algorithme d'addition en virgule flottante, ainsi que son arrondi, est complètement décrit dans [Omo94]. Quach, Takagi et Flynn exposent dans [QTF91] un ensemble de remarques et d'algorithmes permettant de réaliser des opérateurs flottants d'addition avec arrondi exact très efficaces.

### Division

On cherche le quotient $q$ de $a/b$ exactement arrondi, $a$ et $b$ étant deux nombres machine de $n$ chiffres de mantisse. Le calcul du quotient peut se faire en utilisant l'une des deux grandes catégories

d'algorithmes classiques de division :

- **Algorithmes itératifs utilisant la multiplication.**
  On a, par exemple, l'itération de Newton[1] appliquée à la division :

$$r^{(i+1)} = r^{(i)} \times (2 - br^{(i)})$$

  qui converge vers $\frac{1}{b}$ si $r^{(0)}$ est bien choisi. Il suffit ensuite de multiplier ce résultat par $a$ pour obtenir le quotient $q$.

  On peut aussi utiliser l'itération de Goldschimdt [AEGP67], qui est :

$$\begin{aligned} a^{(0)} &= a \\ \epsilon^{(0)} &= 1 - b \\ a^{(i+1)} &= a^{(i)} \times \left(1 + \epsilon^{(i)}\right) \\ \epsilon^{(i+1)} &= \left(\epsilon^{(i)}\right)^2 \end{aligned}$$

  Pour laquelle, $a^{(i)}$ converge vers $a/b$ pour peu que $b$ soit compris entre 1/2 et 1.

  Ces itérations étaient surtout utilisées dans les supercalculateurs (IBM360/91 et Cray). Plus récemment, les coprocesseurs arithmétiques Cyrix utilisent une combinaison de l'itération de Newton et d'une table pour la détermination du point de départ de l'itération (cf [Mat89] et [WF91]).

- **Algorithmes basés sur des additions et des décalages.**
  L'algorithme en base 10, que l'on apprend à l'école primaire, est un algorithme fonctionnant par additions et décalages. Ces algorithmes consistent à obtenir une écriture du quotient $0.q_1q_2\ldots q_n$, chiffre après chiffre, en utilisant une itération de la forme ($\beta$ est la base) :

$$\left\{ \begin{array}{rcl} a^{(0)} &=& a \\ a^{(i+1)} &=& \beta a^{(i)} - q_{i+1} b \end{array} \right.$$

  A chaque itération on obtient un nouveau chiffre $q_{i+1}$ du quotient. Si on choisit $\beta = 2$, il faut $n$ itérations pour obtenir $n$ bits du résultat, ce qui est assez lent pour les formats flottants ayant de grandes mantisses. En augmentant la base, en prenant $\beta = 2^k$, on obtient $k$ bits du quotient à chaque itération. Mais chacune des itérations devient de plus en plus complexe au fur et à mesure que $\beta$ augmente. Le choix de la base "optimale" pour une implantation matérielle est très fortement dépendant de paramètres technologiques et de choix algorithmiques. Ceci explique la grande diversité des diviseurs flottants réalisés jusqu'ici.

  C'est par exemple, un algorithme à base d'additions et de décalages du nom de SRT découvert indépendemment par Sweeney, Robertson et Tocher ([Rob58], [Toc58]) qui est utilisé dans les processeurs `Pentium` d'Intel (où $\beta = 4$, cf [Mul95]).

1. La méthode Newton permet de résoudre $f(x) = 0$ en construisant une suite $x^{(i+1)} = x^{(i)} - \frac{f(x^{(i)})}{f'(x^{(i)})}$ où si $x^{(0)}$ est suffisamment proche d'une racine simple $\alpha$ de $f$ alors $x^{(i)}$ converge quadratiquement vers $\alpha$.

On trouve dans [Omo94, EL94, OF95] des informations sur la réalisation matérielle de la division flottante. Suivant la catégorie d'algorithmes utilisée, l'arrondi exact de la division est plus ou moins complexe.

Dans le cas des algorithmes à base d'additions et de décalages, on obtient à chaque itération les chiffres du quotient et le reste partiel. Pour arrondir, il suffit de calculer un bit de garde supplémentaire et de regarder le signe du reste. On trouve dans [PZ95] la description du diviseur flottant du processeur `UltraSPARC` (algorithme SRT en base 8).

Dans le cas des algorithmes itératifs utilisant la multiplication, l'arrondi est beaucoup plus complexe. En effet, l'itération de Newton converge non pas vers le quotient mais vers l'inverse du diviseur $(1/b)$. Ensuite, il faut encore multiplier cet inverse par $a$ pour obtenir le quotient. Il faut donc garantir l'arrondi de l'inverse de $b$ puis celui du produit $a \times \frac{1}{b}$. En pratique, on calcule l'inverse de $b$ avec $2n + \alpha$ bits de précision (où $\alpha$ est une petite constante), puis on calcule le produit $a \times \frac{1}{b}$ et enfin, on arrondi ce dernier résultat sur $n$ bits. On trouve dans [OF96] une excellente étude par la réalisation de l'arrondi exact de la division pour ce genre d'algorithmes.

#### Racine carrée

La racine carrée peut être calculée en effectuant des itérations de fonction semblables à celle de la division. Par exemple, en résolvant $x^2 = a$ avec Newton on a :

$$x^{(i+1)} = \frac{1}{2} \times \left( x^{(i)} + \frac{a}{x^{(i)}} \right)$$

qui converge vers $\sqrt{a}$ si $x^{(0)}$ est bien choisi. En fait, il est souvent préférable de calculer $\frac{1}{\sqrt{a}}$ en résolvant $\frac{1}{x^2} = a$ puis de multiplier le résultat obtenu par $a$. On a donc l'itération :

$$x^{(i+1)} = \frac{x^{(i)}}{2} \times \left( 3 - a(x^{(i)})^2 \right)$$

On peut utiliser les mêmes techniques pour l'arrondi de la racine carrée que celles de la division. Les calculs de ces deux fonctions sont tellement proches au niveau algorithmique, qu'en pratique le calcul de la division et la racine carrée sont effectués sur un seul et unique opérateur flottant (voir par exemple [PZ95]).

### 1.1.4 Avantages de la norme IEEE 754

Pour en finir avec la présentation de la norme IEEE 754, voici les points qui nous semblent importants dans cette norme.

- La totale *spécification des opérations flottantes* assure une réelle structure mathématique sur laquelle il possible de bâtir des preuves des algorithmes numériques employés.
- Les *formats flottants standards* permettent une totale portabilité des données. La spécification du codage de ces formats et des opérations flottantes permet la portabilité des codes. Ceci est probablement l'un des principaux apports de la norme IEEE 754. En effet, avant son apparition, la conception et le portage d'algorithmes numériques demandaient un travail considérable tant il y avait de systèmes flottants distincts et dont les comportements respectifs étaient radicalement différents. C'est ici que l'arrondi exact des fonctions joue son rôle.

- Enfin, la normalisation des données (formats) et des opérations permet la mise en œuvre d'algorithmes et d'architectures de plus en plus performants tout en restant totalement compatibles (coprocesseurs arithmétiques Cyrix par exemple).

## 1.2 Arrondi exact des fonctions élémentaires

L'arrondi exact des fonctions arithmétiques $(+, -, \times)$ et de certaines fonctions algébriques $(\div, \sqrt{})$ est possible car on sait toujours déterminer quel est le nombre machine de $n$ bits de mantisse qui doit être retourné, suivant le mode d'arrondi utilisé, au vu de la mantisse du résultat des calculs intermédiaires sur $n$ bits plus des chiffres supplémentaires.

On va voir que dans le cas des fonctions élémentaires le problème est de pouvoir déterminer quelle est la précision intermédiaire nécessaire pour pourvoir arrondir exactement. Mais avant cela, nous devons étudier comment évaluer ces fonctions élémentaires.

Les travaux présentés ici sur l'arrondi exact des fonctions élémentaires ont été publiés dans [MT96, DMT96, LMT97] et ont été réalisés en collaboration avec J.M. Muller et V. Lefèvre.

### 1.2.1 Evaluation des fonctions élémentaires

L'évaluation d'une fonction élémentaire $f$ (sin, cos, exp, log ou arctan) au point $x$ (l'argument de la fonction) nécessite généralement plusieurs étapes. En effet, les algorithmes permettant de calculer les fonctions élémentaires ne sont valides que dans de petits domaines. Il faut donc commencer par se ramener au domaine de convergence de l'algorithme, puis évaluer la valeur de la fonction en le nouvel argument, en déduire la valeur de la fonction en l'argument original et enfin arrondir le résultat. Les principaux algorithmes des différentes phases relatives à l'évaluation des fonctions élémentaires sont détaillés dans le livre de Jean-Michel Muller [Mul97].

Dans la suite, nous noterons$n$ le nombre de chiffres de la mantisse du résultat final (après arrondi), $m$ le nombre de chiffres de la mantisse du résultat intermédiaire qui sont nécessaires pour arrondir correctement celui-ci. Nous verrons que le problème de l'arrondi exact des fonctions élémentaires consiste à déterminer la valeur de $m$ pour chaque fonction et chaque format de nombre flottant (simple précision, double précision...).

Voici l'ensemble des phases effectuées lors de l'évaluation d'une fonction élémentaire :

① La *réduction d'argument*. Cette première phase consiste à déduire de l'argument $x$ l'argument réduit $x^*$ qui appartient au domaine de convergence de l'algorithme utilisé pour calculer $g(x^*)$, où $f(x)$ peut se déduire simplement de $g(x^*)$ (dans la plupart des cas $f = g$).

② L'*évaluation*. La deuxième phase consiste en l'évaluation effective de $g(x^*)$.

③ L'*arrondi final*. En fait, avant d'effectuer l'arrondi proprement dit, il faut calculer $f(x)$ à partir de $g(x^*)$ (cette opération est en général triviale) et ensuite arrondir le résultat obtenu dans le bon format.

Il existe deux types de réduction d'argument. Le premier, la *réduction additive* , consiste à soustraire suffisamment de fois une constante bien choisie afin de garantir que $x^*$ soit bien dans le domaine de convergence de l'algorithme. On a une réduction de la forme :

$$x^* = x - kC$$

où $k$ est un entier et $C$ une constante. Par exemple, pour certains algorithmes calculant des fonctions trigonométriques on a $C = \frac{\pi}{8}$.

Le deuxième type de réduction est la *réduction multiplicative* qui consiste à diviser l'argument initial par une constante bien choisie suffisamment de fois. On a alors une réduction de la forme :

$$x^* = \frac{x}{C^k}$$

où $k$ est un entier et $C$ une constante. Cette réduction est utilisée, par exemple, dans le cas du calcul de la fonction logarithme où un choix judicieux pour $C$ est la base du système utilisé.

En pratique, la réduction multiplicative ne pose pas de problème majeur de précision. Par contre, dans le cas de la réduction additive, il peut se produire des cas de *cancellation catastrophique* si $x$ est proche d'un multiple entier de $C$. La soustraction flottante de ces deux quantités va donner un résultat dont un grand nombre de bits de poids forts seront nuls. Il y a alors *cancellation catastrophique* si trop de bits de poids faibles ne sont plus significatifs par rapport à l'accumulation des erreurs des précédentes étapes du calcul. La renormalisation va donc introduire des zéros arbitraires dans les poids faibles de l'argument réduit. Il faut donc effectuer la réduction d'argument additive avec une précision intermédiaire plus grande que la précision nécessaire à la représentation de l'argument réduit. Différents algorithmes visant des implantations logicielles sont décrits dans [CW80]. On peut trouver dans [DMMM94] et [DMMM95] un algorithme permettant de réaliser matériellement la réduction d'argument additive avec une bonne précision.

Il existe deux catégories d'algorithmes d'évaluation des fonctions élémentaires. Tout d'abord, les algorithmes par additions et décalages comme CORDIC [Vol59, Wal90] ou BKM [BKM93, BKM94]. Ces algorithmes produisent une suite de valeurs qui converge vers le résultat. Les algorithmes de l'autre catégorie sont basés sur des approximations polynômiales ou rationnelles des fonctions. De nombreux exemples de ces différents types d'algorithmes sont décrits dans [Mul97]. Schulte et Swartzlander ont proposé dans [SS94] des algorithmes basés sur des approximations polynômiales permettant l'évaluation et l'arrondi exact des fonctions élémentaires $2^x$ et $\log_2 x$ en simple précision.

Avec ces algorithmes (additions et décalages ou par approximation), il est toujours possible d'évaluer une fonction élémentaire avec une précision relative meilleure que $2^{-m}$ en base 2 quel que soit $m$. En fait, la fonction à évaluer doit respecter certaines conditions, mais dans le cas des fonctions qui nous intéressent ($\sin, \cos, \exp, \log$ et $\arctan$), ces conditions sont vérifiées.

Il est donc possible d'évaluer une fonction élémentaire usuelle avec autant de chiffres de mantisse que nécessaire. Reste maintenant à étudier le problème de l'arrondi exact du résultat de l'évaluation de ces fonctions.

### 1.2.2 Le dilemme du fabricant de tables

Soient $x$ un nombre machine, et $y = f(x)$, où $f$ est une fonction élémentaire ($\sin, \cos, \exp, \log$ ou $\arctan$). On désire arrondir exactement $f(x)$ à $n$ chiffres suivant l'un des quatre modes d'arrondi IEEE. Nous avons vu dans la section 1.2.1 qu'il était possible de calculer $f(x)$ avec une erreur de $2^{-m}$. Le calcul intermédiaire (avant arrondi) se fait avec $m$ chiffres de mantisse. Dans les cas

suivants, il n'est pas possible d'arrondir exactement $f(x)$.

- pour l'arrondi au plus près :

$$\overbrace{1.\underbrace{xxx\ldots xx}_{n \text{ bits}}1000\ldots00}^{m \text{ bits}}xxx\ldots \quad \text{ou} \quad \overbrace{1.\underbrace{xxx\ldots xx}_{n \text{ bits}}0111\ldots11}^{m \text{ bits}}xxx\ldots$$

- pour l'arrondi vers $+\infty$, $-\infty$ ou $0$ :

$$\overbrace{1.\underbrace{xxx\ldots xx}_{n \text{ bits}}0000\ldots00}^{m \text{ bits}}xxx\ldots \quad \text{ou} \quad \overbrace{1.\underbrace{xxx\ldots xx}_{n \text{ bits}}1111\ldots11}^{m \text{ bits}}xxx\ldots$$

Ce problème est connu sous le nom du *dilemme du fabricant de tables* (*table maker's dilemma* ou TMD). La Figure 1.3 illustre le TMD dans le cas de l'arrondi au plus près. Les $x_k$ sont des nombres machine consécutifs et les $I_k$ sont les intervalles de $\mathbb{R}$ qui ont le même arrondi au plus près : $x_k$. Si $y'$ est la valeur retournée par l'algorithme d'évaluation de $f$, on sait que $y$, la valeur exacte du résultat, est contenue dans l'intervalle centré en $y'$ et de longueur $2^{-m}$. Il peut y avoir un problème si cet intervalle de $y$ chevauche deux intervalles consécutifs $I_k$ et $I_{k+1}$. En effet, si $y'$ n'est pas calculé avec une précision suffisante, on ne sait pas s'il faut arrondir à $x_k$ ou $x_{k+1}$.

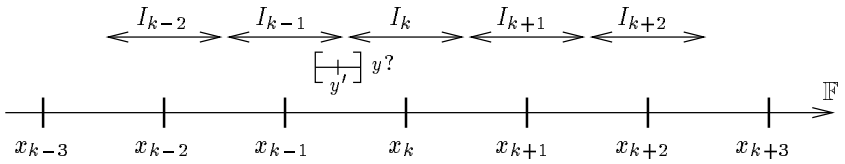



FIG. 1.3 – *Le dilemme du fabricant de tables dans le cas de l'arrondi au plus près.*

Par exemple, en utilisant une arithmétique flottante sur 6 bits, le nombre $\sin(11.1010) = 0.0\boxed{111011}01111110\ldots$ ne peut pas être arrondi au plus près si $m \leq 13$.

Le principal problème de l'arrondi exact des fonctions élémentaires consiste donc a déterminer la précision intermédiaire nécessaire pour chaque fonction élémentaire et pour chaque format de nombre flottant. Nous verrons dans la suite (cf 1.2.5) que la valeur de $m$ est bornée, c'est-à-dire qu'il existe une précision à partir de laquelle on est certain que le dilemme du fabricant de tables ne peut pas se produire.

Mais les bornes que nous fournissent les meilleurs résultats théoriques actuels sont trop importantes (de l'ordre du million de bits en double précision pour l'exponentielle) pour toujours faire les calculs intermédiaires avec une telle précision. Nous allons donc étudier avec différents tests la fréquence et la distribution des cas qui conduisent à de grandes valeurs de $m$. Dans le cadre de cette étude, nous avons réalisé des tests exhaustifs sur les fonctions élémentaires classiques en simple précision ainsi que des tests aléatoires pour les précisions supérieures (cf 1.2.3 et 1.2.4).

Dans les précisions élevées (pour la quadruple précision par exemple), les tests exhaustifs n'étant pas envisageables, il faudra être capable, dans certains cas, d'effectuer les calculs intermédiaires avec une précision très importante. Dans ce but, nous étudierons la faisabilité et le coût de ces calculs

(cf 1.2.6).

Enfin, nous avons réalisé une étude probabiliste du problème. Bien que les hypothèses de cette étude ne soient pas toujours totalement réalistes, elle illustre clairement les résultats obtenus lors des simulations et nous permet de bien appréhender les mécanismes sous-jacents au problème. De plus, cette étude nous permettra de justifier les stratégies utilisables pour réaliser effectivement l'arrondi exact des fonctions élémentaires (cf 1.3).

### 1.2.3 Test exhaustifs sur les fonctions élémentaires en simple précision

Schulte et Swarztlander ont proposé des algorithmes qui permettent d'obtenir les résultats exacts pour les fonctions $1/x$, $\sqrt{x}$, $2^x$ et $\log_2 x$ [SS93, SS94]. Leur étude nous a servi de point de départ pour notre travail. Pour trouver une valeur correcte de $m$ ils ont testé exhaustivement les précédentes fonctions pour des nombres flottants de 16 et 24 bits de mantisse. Le Tableau 1.5 donne les résultats de leurs tests exhaustifs.

| | $n = 16$ | $n = 24$ |
|---|---|---|
| $\log_2 x$ | 35 | 51 |
| $2^x$ | 29 | 48 |

TAB. 1.5 – *Résultats des tests exhaustifs effectués par Schulte et Swarztlander. Valeurs de $m$ permettant d'arrondir exactement les fonctions élémentaires $\log_2 x$ et $2^x$.*

Nous avons effectué des tests exhaustifs sur les nombres flottants en simple précision pour les fonctions $\sin x$, $\cos x$, $e^x$, $\ln x$, $\arctan x$, $\log_2 x$, et $2^x$.

Rappelons les notations : $n$ est la taille de la mantisse du résultat final, $m$ est la taille de la mantisse du résultat intermédiaire et $k = m - n$ est le nombre de bits supplémentaires nécessaires pour arrondir exactement le résultat sur $n$ bits.

Pour les tests exhaustifs, nous avons utilisé une machine à base du processeur Pentium Pro d'Intel pour son format flottant sur 80 bits. Le programme calcule pour chaque combinaison des bits d'entrée la fonction élémentaire testée et compte la plus grande chaîne de 1 ou de 0 consécutifs à partir du chiffre de rang $n+1$. Les calculs intermédiaires sont effectués en double précision étendue (sur 65 bits de mantisse) avec la librairie `math.h`. Cette librairie effectue une réduction d'argument avec un algorithme qui agit comme si elle était effectuée en précision infinie. De plus, la librairie `math.h` évalue les fonctions élémentaires avec une erreur maximale de l'ordre du dernier bit et dont les pertes de précision sont indiquées par un mécanisme de drapeau (cf les spécifications de `matherr` en C) qu'il faut tester après chaque calcul.

Le seul problème qui puisse subsister est celui d'une valeur d'entrée qui conduise à une chaîne de 1 ou de 0 qui s'étende jusqu'au dernier bit de la mantisse sur 65 bits. Mais ce cas ne s'est jamais produit. Dans le pire cas, la chaîne de 1 ou de 0 s'arrête plus de 10 bits avant la fin de la mantisse.

On peut déduire les valeurs de $m$ correspondant aux autres modes d'arrondi à partir de celles trouvées pour l'arrondi au plus près. La recherche des plus grandes chaînes de 1 ou de 0 consécutifs commence au bit de rang $n + 1$ dans le résultat intermédiaire pour le cas de l'arrondi au plus près. Dans le cas des autres modes d'arrondi, la plus grande chaîne de 1 ou de 0 consécutifs ne peut donc pas être plus longue de plus de un bit que celle trouvée dans la recherche du pire cas pour l'arrondi

au plus près.

Le Tableau 1.6 donne la valeur de $m$ nécessaire pour arrondir exactement la fonction $\sin x$ pour des nombres flottants dont la taille de la mantisse varie entre 10 et 24 bits et dont l'exposant varie entre -5 et 5 avec le mode d'arrondi au plus près.

| | exposant | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $n$ | -5 | -4 | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 |
| 10 | 20 | 21 | 18 | 22 | 21 | 18 | 20 | 20 | 21 | 19 | 20 |
| 11 | 24 | 24 | 26 | 24 | 22 | 24 | 20 | 23 | 20 | 21 | 24 |
| 12 | 24 | 26 | 25 | 22 | 22 | 26 | 24 | 26 | 24 | 30 | 24 |
| 13 | 25 | 29 | 30 | 26 | 25 | 26 | 24 | 26 | 24 | 24 | 26 |
| 14 | 27 | 29 | 28 | 28 | 32 | 30 | 28 | 29 | 29 | 27 | 26 |
| 15 | 29 | 29 | 32 | 29 | 29 | 31 | 29 | 29 | 29 | 32 | 28 |
| 16 | 32 | 32 | 36 | 32 | 31 | 30 | 31 | 32 | 33 | 31 | 35 |
| 17 | 34 | 32 | 32 | 37 | 37 | 34 | 33 | 31 | 34 | 37 | 34 |
| 18 | 39 | 36 | 35 | 35 | 39 | 36 | 34 | 38 | 34 | 39 | 39 |
| 19 | 38 | 38 | 36 | 37 | 41 | 36 | 37 | 40 | 38 | 39 | 41 |
| 20 | 40 | 41 | 41 | 42 | 40 | 40 | 39 | 39 | 39 | 40 | 41 |
| 21 | 42 | 41 | 42 | 43 | 44 | 41 | 41 | 41 | 46 | 42 | 49 |
| 22 | 45 | 45 | 43 | 45 | 44 | 42 | 48 | 43 | 45 | 44 | 42 |
| 23 | 46 | 45 | 45 | 46 | 49 | 46 | 49 | 45 | 45 | 46 | 48 |
| 24 | 50 | 52 | 46 | 51 | 46 | 49 | 49 | 48 | 47 | 47 | 47 |

TAB. 1.6 – *Valeurs de $m$ pour la fonction $\sin x$, avec des nombres flottants dont la taille de la mantisse $n$ varie entre 10 et 24, l'exposant varie entre -5 et 5 et avec le mode d'arrondi au plus près.*

Le Tableau 1.7 donne la valeur de $m$ nécessaire en simple précision pour chacune des fonctions élémentaires testées et pour l'arrondi au plus près et pour tous les exposants qui conduisent à des valeurs numériques (l'exponentielle d'un grand nombre est un dépassement de capacité). Le temps de calcul qui a été nécessaire pour effectuer ces tests exhaustifs est de l'ordre de une semaine.

| $f(x)$ | $\sin x$ | $\cos x$ | $e^x$ | $\ln x$ | $\arctan x$ | $\log_2 x$ | $2^x$ |
|---|---|---|---|---|---|---|---|
| $m$ | 52 | 50 | 49 | 53 | 49 | 51 | 48 |

TAB. 1.7 – *Valeurs de $m$ pour la simple précision ($n = 24$).*

On remarque que l'on trouve la même valeur pour le pire cas de $m$ pour les fonctions $2^x$ et $\log_2 x$ que Schulte et Swarztlander. Au vu de ces divers résultats, on est tenté de dire qu'il faut à peu près le double du nombre de bits de la mantisse finale pour effectuer l'arrondi exact. Empiriquement, on a

$$m \simeq 2 \times n$$

et nous verrons que l'approche probabiliste du problème renforce cette intuition.

Nous avons aussi effectué des tests exhaustifs pour obtenir la distribution des longueurs des chaînes de 1 ou de 0 consécutifs. Ces tests ont été effectués avec le logiciel de calcul formel `Maple` pour différentes valeurs de $n$. Les nombres flottants en entrée sont codés exactement (`Maple` fait du calcul exact sur les rationnels), puis les fonctions élémentaires sont évaluées avec une très grande précision (100 chiffres de base 10, car `Maple` calcule en base 10 ), puis ils sont convertis en base 2, et enfin il y a une recherche de la plus grande chaîne de 1 ou de 0 consécutifs. Le Tableau 1.8 donne les résultats de cette recherche de distribution pour des arithmétiques flottantes avec $n$ variant de 1 à 16 bits et pour la fonction $\sin x$. Seul l'exposant 0 a été testé pour chaque valeur de $n$. Le mode d'arrondi choisi est encore le mode d'arrondi au plus près. Chaque case du tableau donne le nombre de cas, parmi les $2^n$ possibles, qui nécessitent $k$ bits supplémentaires pour arrondir $\sin x$ sur $n$ bits.

| | $k$ | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 3 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 5 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 9 | 3 | 1 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 18 | 7 | 3 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 34 | 15 | 13 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 72 | 26 | 13 | 7 | 3 | 6 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 130 | 65 | 31 | 13 | 5 | 6 | 4 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9 | 272 | 119 | 67 | 23 | 16 | 6 | 4 | 3 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10 | 538 | 239 | 114 | 69 | 26 | 20 | 15 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 11 | 1037 | 527 | 215 | 121 | 71 | 50 | 14 | 6 | 2 | 4 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 12 | 2049 | 1030 | 521 | 255 | 127 | 49 | 27 | 21 | 7 | 6 | 3 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 13 | 4144 | 2017 | 1025 | 502 | 255 | 142 | 56 | 26 | 11 | 10 | 3 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 14 | 8247 | 4081 | 2044 | 1012 | 516 | 251 | 106 | 65 | 30 | 23 | 2 | 3 | 1 | 1 | 1 | 1 | 0 | 0 |
| 15 | 16479 | 8114 | 4077 | 2040 | 1033 | 519 | 249 | 127 | 65 | 32 | 13 | 6 | 10 | 1 | 2 | 1 | 0 | 0 |
| 16 | 32850 | 16435 | 8073 | 4091 | 2080 | 999 | 494 | 261 | 136 | 53 | 32 | 21 | 5 | 5 | 0 | 0 | 0 | 1 |

TAB. 1.8 – *Distribution de la taille des chaînes de 1 ou de 0 consécutifs pour la fonction* $\sin x$ *avec arrondi au plus près et pour l'exposant 0.*

On observe dans le Tableau 1.8 une très forte décroissance de la longueur des chaînes de 1 ou de 0 consécutifs lorsque $k$ augmente. Ceci laisse à penser que des très longues chaînes, donc une valeur de $m$ importante, sont extrêmement rares. Nous verrons dans la suite que ces résultats de simulation sont corroborés par la modélisation probabiliste du problème (cf 1.2.7).

### 1.2.4 Tests aléatoires pour les précisions supérieures

Nous avons aussi effectué des tests aléatoires pour des précisions supérieures. Ces tests ont été réalisé avec `Maple` en générant aléatoirement les entrées selon une loi uniforme sur les bits de la mantisse (probabilité $\frac{1}{2}$ pour les 1 et pour les 0). Pour chaque taille de la mantisse et pour chaque fonction, seul l'exposant 0 a été testé. Les calculs intermédiaires ont été effectués avec une précision au moins supérieure à 4 fois à celle de la mantisse finale. Ici encore, le seul problème serait l'existence d'une chaîne de 1 ou de 0 consécutifs qui s'étende jusqu'aux derniers bits. Mais ce cas ne s'est jamais produit.

Le Tableau 1.9 donne la distribution des chaînes qui conduisent au dilemme du fabricant de tables lors du calcul de la fonction $\sin x$ pour des arithmétiques dont la taille de la mantisse varie entre 20 bits et 55 bits. Le mode d'arrondi choisi est l'arrondi au plus près et seul l'exposant 0 est testé.

| | $k$ | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 20 | 270 | 107 | 63 | 31 | 12 | 11 | 2 | 3 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 21 | 235 | 144 | 54 | 34 | 17 | 2 | 4 | 4 | 3 | 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| 22 | 258 | 114 | 58 | 33 | 21 | 5 | 9 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 23 | 244 | 129 | 58 | 43 | 13 | 5 | 3 | 3 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 24 | 248 | 120 | 73 | 37 | 9 | 5 | 3 | 2 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 25 | 259 | 124 | 54 | 36 | 9 | 5 | 8 | 2 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 26 | 242 | 119 | 69 | 38 | 18 | 7 | 4 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 27 | 249 | 139 | 68 | 21 | 12 | 3 | 2 | 3 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 28 | 246 | 127 | 65 | 29 | 19 | 10 | 2 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 29 | 277 | 113 | 60 | 60 | 25 | 13 | 8 | 1 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 30 | 256 | 128 | 57 | 29 | 16 | 6 | 6 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 31 | 266 | 121 | 58 | 38 | 10 | 5 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 32 | 233 | 132 | 67 | 36 | 14 | 9 | 5 | 1 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 33 | 249 | 122 | 67 | 26 | 14 | 10 | 5 | 6 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 34 | 225 | 116 | 67 | 31 | 9 | 13 | 2 | 5 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 35 | 226 | 135 | 65 | 36 | 17 | 11 | 6 | 3 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 36 | 226 | 119 | 72 | 43 | 16 | 11 | 10 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 37 | 248 | 121 | 70 | 37 | 10 | 3 | 5 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 38 | 231 | 131 | 75 | 35 | 14 | 5 | 5 | 2 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 39 | 253 | 120 | 61 | 33 | 17 | 6 | 4 | 5 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 40 | 226 | 142 | 62 | 45 | 12 | 7 | 2 | 3 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 41 | 259 | 99 | 73 | 38 | 10 | 12 | 3 | 3 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 42 | 260 | 124 | 57 | 27 | 19 | 8 | 2 | 1 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 43 | 249 | 125 | 66 | 30 | 15 | 8 | 4 | 0 | 2 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 44 | 242 | 110 | 75 | 44 | 13 | 7 | 5 | 3 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 45 | 248 | 121 | 62 | 38 | 16 | 7 | 5 | 1 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 46 | 251 | 113 | 68 | 31 | 21 | 11 | 2 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 47 | 256 | 124 | 60 | 32 | 12 | 6 | 2 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 48 | 249 | 123 | 62 | 36 | 15 | 10 | 2 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 49 | 259 | 128 | 51 | 32 | 9 | 6 | 4 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 50 | 264 | 114 | 60 | 26 | 20 | 10 | 4 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 51 | 254 | 116 | 60 | 31 | 22 | 7 | 3 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 52 | 259 | 129 | 56 | 33 | 13 | 6 | 1 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 53 | 257 | 113 | 64 | 38 | 18 | 6 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 54 | 261 | 128 | 54 | 29 | 16 | 8 | 0 | 2 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 55 | 234 | 135 | 58 | 39 | 18 | 5 | 5 | 4 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

TAB. 1.9 – *Distribution des chaînes de 1 ou de 0 consécutifs qui provoquent le TMD lors du calcul de la fonction* $\sin x$ *pour différentes tailles de la mantisse (avec 0 comme exposant et l'arrondi au plus près).*

Nous avons effectué des tests aléatoires sur la distribution des chaînes de 1 ou de 0 consécutifs pour chacune des fonctions élémentaires classiques. Les résultats de ces tests sont résumés dans le Tableau 1.10.

Enfin, nous avons réalisé des tests aléatoires pour la fonction $\sin x$ en double et quadruple précision en utilisant `Maple` en cherchant la plus grande valeur de $m$. Les résultats sont :

- en double précision ($n = 53$), après 254 jours de calcul, le pire cas trouvé est $m = 101$;
- en quadruple précision ($n = 112$), après 258 jours de calcul, le pire cas trouvé est $m = 205$.

Là encore, il semble que $m$ soit proche de $2 \times n$. Mais le nombre de cas testés en 258 jours de calcul

| | $k$ | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $f(x)$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| $\sin x$ | 982 | 517 | 253 | 129 | 59 | 30 | 14 | 5 | 6 | 3 | 1 | 1 | 0 | 0 | 0 | 0 |
| $\cos x$ | 1422 | 285 | 148 | 75 | 38 | 16 | 8 | 5 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $e^x$ | 1015 | 499 | 239 | 115 | 73 | 28 | 16 | 5 | 5 | 3 | 1 | 0 | 1 | 0 | 0 | 0 |
| $\ln x$ | 960 | 504 | 280 | 137 | 64 | 24 | 18 | 3 | 6 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| $\arctan x$ | 981 | 506 | 260 | 120 | 66 | 30 | 20 | 6 | 6 | 2 | 0 | 2 | 1 | 0 | 0 | 0 |
| $\log_2 x$ | 1002 | 495 | 266 | 124 | 54 | 29 | 18 | 4 | 6 | 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| $2^x$ | 972 | 524 | 263 | 115 | 61 | 32 | 13 | 9 | 6 | 3 | 1 | 0 | 0 | 1 | 0 | 0 |

TAB. 1.10 – *Distribution des chaînes de 1 ou de 0 consécutifs qui provoquent le TMD lors du calcul de différentes fonctions élémentaires pour des mantisses de 53 bits (double précision), et avec 0 comme exposant pour l'arrondi au plus près. Résultats de tirages aléatoires uniformément répartis.*

(repartis sur plusieurs machines) est extrêmement faible devant le nombre de cas possibles.

### 1.2.5 Bornes supérieures

La question est maintenant de savoir si il existe une valeur maximale pour $m$ ne dépendant que de la précision souhaitée. En 1882, Lindemann a démontré que l'exponentielle d'un nombre algébrique (éventuellement complexe) non nul n'est pas algébrique [Bak75]. Les nombres machine sont rationnels, donc algébriques. Par conséquent, l'exponentielle, le sinus, le cosinus, et l'arctangente d'un nombre machine différent de 0 et le logarithme d'un nombre machine différent de 1 ne peuvent pas comporter dans leur écriture binaire une chaîne infinie de 0 ou de 1. Cette propriété se généralise d'ailleurs à d'autres bases que 2. Le nombre de nombres machine étant fini, il existe une valeur $m$ à partir de laquelle on est certain que le dilemme du fabricant de tables ne peut pas se produire. Le problème reste à trouver un majorant pour $m$.

Le théorème 1.1 montré par Yu. Nesterenko et M. Waldschmidt en 1995 (cf [NW96]) permet de connaître des bornes pour la valeur de $m$ nécessaire pour chaque format de nombre flottant.

Si $p/q$ est un nombre rationnel avec $q > 0$ et $\mathrm{pgcd}(p, q) = 1$, alors on défini $H(p/q) = \max\{|p|, q\}$.

**Théorème 1.1 (Nesterenko et Waldschmidt, 1995)** *Soient $\alpha$ et $\beta$ deux rationnels, avec $\beta \neq 0$. Soient $A$, $B$ et $E$ des réels positifs avec $E \geq e$ vérifiant*

$$A \geq \max\left(H(\alpha), e\right), \quad B \geq H(\beta).$$

*alors*

$$\begin{aligned}&|e^{\beta} - \alpha| \geq \\ &\exp\Big(-211 \times (\log B + \log\log A + 2\log(E|\beta|_+) + 10) \times (\log A + 2E|\beta| + 6\log E) \\ &\times (3.3\log(2) + \log E) \times (\log E)^{-2}\Big)\end{aligned}$$

*où* $|\beta|_+ = \max(1, |\beta|)$.

En appliquant ce théorème, $\alpha$ et $\beta$ étant deux nombres machine, on obtient un majorant de la distance entre un nombre machine et l'exponentielle d'un nombre machine (ou bien un majorant de la distance entre un nombre machine et le logarithme d'un nombre machine). Le Tableau 1.11 donne

un majorant de la précision intermédiaire nécessaire (la valeur de $m$) pour arrondir exactement le résultat du calcul de l'exponentielle sur $n$ bits, et ce pour les principaux formats de nombres flottants définis dans la norme IEEE 754.

| $n$ | intervalles | $m$ |
|---|---|---|
| 24 | $[0, \ln 2]$ | 494 416 |
| | $[0, 10]$ | 3 074 888 |
| 53 | $[0, \ln 2]$ | 1 038 560 |
| | $[0, 10]$ | 5 234 891 |
| 112 | $[0, \ln 2]$ | 2 527 507 |
| | $[0, 10]$ | 10 409 113 |

TAB. 1.11 – *Majorants de m pour le calcul de l'exponentielle dans les principaux formats de nombres flottants IEEE 754.*

Il est donc possible dans le pire cas d'être obligé de calculer des exponentielles et des logarithmes avec une précision intermédiaire dont le nombre de bits est de l'ordre de quelques millions. Il est très vraisemblable que les vraies valeurs de $m$ soient beaucoup plus petites. Les tests exhaustifs effectués pour la simple précision (cf 1.2.3) et les tests aléatoires dans les précisions supérieures (cf 1.2.4) indiquent clairement que les cas nécessitant des calculs intermédiaires avec une précision très importante sont extrêmement rares.

Même si les résultats de théorie de nombres obtenus par Yu. Nesterenko et M. Waldschmidt sont très récents et que l'on peut supposer qu'ils seront améliorés dans le futur, il est nécessaire de pouvoir calculer avec de telles précisions pour garantir l'arrondi exact. De plus, il ne faut pas oublier que pour des précisions élevées, la recherche exhaustive ne sera probablement pas possible prochainement. La question de la réalisibilité de l'arrondi exact dans le pire cas se ramène à l'étude du coût des calculs sur des nombres de plusieurs millions de chiffres.

### 1.2.6 Coût du calcul des fonctions élémentaires en très haute précision

En 1976, Brent a proposé plusieurs algorithmes pour calculer les fonctions élémentaires avec une très haute précision, ainsi qu'une analyse de leur temps d'exécution en fonction de celui de la multiplication [Bre76]. Soit $M(n)$ le nombre d'opérations nécessaires pour effectuer le produit de deux nombres de $n$ chiffres. Brent a démontré le théorème 1.2.

**Théorème 1.2 (Brent, 1976)** *Si les constantes $\pi$ et $\log 2$ sont précalculées, la fonction élémentaire $f(x)$ peut évaluée avec une précision de $n$ bits avec $(K + O(1))M(n)\log_2 n$ opérations, où*

$$K = \begin{cases} 13 & si \quad f(x) = \log x \ ou \ \exp x \\ 34 & si \quad f(x) = \arctan x \ ou \ \sin x \ ou \ \cos x \end{cases}$$

En 1994, D. Zuras a programmé différents algorithmes de multiplication et d'élévation au carré sur une machine HP-9000/270 fonctionnant à 50 MHz [Zur94]. Le meilleur algorithme utilisé permettait de multiplier deux nombres de 1 million de bits en 8 secondes environ.

En 1996, V. Lefèvre dans son stage de DEA ([Lef96]) à réalisé différents algorithmes de multiplications en multiprécision. Les temps de son meilleur algorithme, basé sur une décomposition des nombres de type Toom-Cook, sont résumés dans le Tableau 1.12.

| taille des nombres en kbits | 100 | 200 | 500 | 1 000 | 2 000 | 5 000 | 10 000 |
|---|---|---|---|---|---|---|---|
| Sparc-Station 4 à 110MHz | 0.50 | 1.36 | 5.25 | 14.8 | 40.7 | 154 | 434 |
| Ultra-Sparc 1 à 143 MHz | 0.16 | 0.46 | 1.77 | 4.8 | 13.5 | 51 | 140 |
| Pentium Pro à 150 MHz | 0.27 | 0.74 | 2.73 | 7.8 | 21.6 | 84 | 226 |

TAB. 1.12 – *Temps en secondes obtenus par V. Lefèvre pour la multiplication de deux nombres en multiprécision sur différentes machines.*

En utilisant le meilleur algorithme de multiplication de V. Lefèvre sur une Ultra-Sparc 1 à 143 MHz, on a 4.8 s comme temps de base pour la multiplication de deux nombres de 1 million de bits. A l'aide du théorème 1.2, on peut estimer que le calcul d'une exponentielle sur 1 million de bits serait alors d'environ 20 minutes.

### 1.2.7 Approche probabiliste du problème

Le but de cette section est d'avoir une idée de la fréquence d'apparition des calculs en très haute précision afin de proposer des stratégies particulières pour la mise en œuvre pratique de l'arrondi exact des fonctions élémentaires.

Nous utiliserons le mode d'arrondi au plus proche dans cette étude afin de simplifier les calculs. Des résultats similaires peuvent être obtenus pour les autres modes d'arrondi.

Soit $f$ une fonction élémentaire, $x$ un nombre machine. Soit $y$ la valeur approchée de la mantisse sur $m$ bits de $f(x)$. On désire arrondir exactement $y$ sur $n$ bits. On note $k$ le nombre de bits supplémentaires nécessaires pour pouvoir arrondir $y$ à $n$ bits ($k = m - n$). Enfin, soit $n_e$ le nombre d'exposants différents du type considéré (après la réduction d'argument).

Les $k$ bits de poids faible de $y$ (les bits "supplémentaires" pour l'arrondi) peuvent être considérés comme une séquence aléatoire de 0 et de 1, avec la probabilité $\frac{1}{2}$ pour les 0 comme pour les 1. Ceci a été montré par Feldstein et Goodman sous certaines hypothèses, et après une longue suite de calculs, dans leur travail sur la distribution des chiffres de poids faible de résultats de calculs numériques [FG76]. Cette hypothèse peut sembler douteuse car le phénomène est totalement déterministe, mais pratiquement ce modèle aléatoire des bits de poids faibles (ceux qui nous intéressent) est particulièrement proche des expérimentations effectuées. Nous supposons que les $k$ derniers bits des résultats de l'évaluation de $f(x_1)$ et de $f(x_2)$, quels que soient $x_1$ et $x_2$, sont indépendants. Les valeurs de $y$ qui posent un problème pour l'arrondi au plus près sont (avec $k \geq 1$) :

$$y = y_0.y_1y_2\ldots y_{n-1}\overbrace{0111111\ldots1111}^{\text{k bits}} \qquad \text{ou} \qquad y = y_0.y_1y_2\ldots y_{n-1}\overbrace{1000000\ldots0000}^{\text{k bits}}$$

Le problème est maintenant d'estimer la valeur maximum de $k$. D'après nos hypothèses, la probabilité d'avoir $k \geq k_0$ est $\sum_{i=k_0}^{+\infty} 2^{-i} = 2^{1-k_0}$. Soit $N = n_e \times 2^{n-1}$ le nombre de nombres flottants (du fait de la normalisation, il n'y a que $2^{n-1}$ mantisses différentes pour une valeur de l'exposant).

On cherche la probabilité $\mathcal{P}_{k_0}$ d'avoir au moins une entrée parmis les $N$ nombres machine possibles qui conduit à une valeur de $k$ supérieure ou égale à $k_0$. Le moyen le plus simple de calculer cette probabilité est de passer par les évènements complémentaires. L'événement complémentaire de "au moins une entrée conduit à $k \geq k_0$" est "aucune entrée ne conduit à $k \geq k_0$". La probabilité du dernier événement est la probabilité que "chaque entrée ne produise pas de chaîne telle $k \geq k_0$" à la puissance $N$, car les $N$ entrées sont indépendantes. Enfin, on a que la probabilité que "chaque

entrée ne produise pas de chaîne telle $k \geq k_0$" est égale à $1 - 2^{1-k_0}$. On a donc :

$$\mathcal{P}_{k_0} = 1 - \left(1 - 2^{1-k_0}\right)^N$$

Nous cherchons maintenant la valeur de $k_0$ telle que la probabilité d'obtenir au moins une valeur de $k$ supérieure à $k_0$ (parmis les $N$ nombres flottants différents) soit inférieure ou égale à $\frac{1}{2}$. $\mathcal{P}_{k_0} \leq \frac{1}{2}$ est équivalent à $\frac{1}{2} \leq \left(1 - 2^{1-k_0}\right)^N$. Donc on a :

$$\ln \frac{1}{2} \leq N \times \ln(1 - 2^{1-k_0}) \tag{1.1}$$

On pose $t = 2^{1-k_0}$. Si $t$ est suffisamment petit, on peut approcher $\ln(1 - 2^{1-k_0}) = \ln(1-t)$ par $-t - \frac{t^2}{2}$. L'équation 1.1 devient :

$$\frac{N}{2}t^2 + Nt - \ln 2 \leq 0$$

Cette dernière inégalité est vérifiée pour $t < -1 + \sqrt{1 + \frac{2}{N}\ln 2} \simeq \frac{\ln 2}{N}$. Finalement, on a $\mathcal{P}_{k_0} \leq \frac{1}{2}$ pour $2^{1-k_0} \leq \frac{\ln 2}{N}$, c'est-à-dire :

$$k_0 \geq 1 - \frac{\ln(\ln 2)}{\ln 2} + \log_2 N \simeq 1.529 + \log_2 N$$

Finalement on a :

$$\mathcal{P}_{k_0} \leq \frac{1}{2} \quad \Leftrightarrow \quad k_0 \geq n + \log_2(n_e) + 0.529$$

Cette valeur de $k_0$ indique clairement que si l'on considère un petit nombre d'exposants (ce qui est le cas pour les fonctions élémentaires du fait de la phase de réduction d'argument), la probabilité d'obtenir le dilemme du fabricant de tables est faible si $m$ est un peu supérieur à $2n$. Cette modélisation probabiliste du problème confirme bien les résultats obtenus lors des simulations. Par exemple, dans le Tableau 1.8 on voit bien une décroissance quasi exponentielle (environ $2^{-k}$) de la longueur de la plus grande chaîne de 0 ou de 1 qui conduit au dilemme du fabricant de tables. Bien sûr, cette modélisation n'est qu'une approximation du problème, qui est totalement déterministe, mais elle permet de comprendre intuitivement l'influence des paramètres mis en jeu.

## 1.3 Mise en œuvre de l'arrondi exact des fonctions élémentaires

Nous avons vu précédemment que le problème de l'arrondi exact des fonctions élémentaires est en fait la détermination de la précision nécessaire pour effectuer les calculs intermédiaires (la valeur de $m$). Les différentes campagnes de tests (exhaustifs et aléatoires) nous ont montré qu'en général la précision nécessaire lors des calculs intermédiaires est relativement modérée. La modélisation probabiliste du problème est venue corroborer ces observations.

Les tests exhaustifs nous donnent des bornes sûres de faibles valeurs pour $m$ mais ils ne sont possible que pour des précisions modérées. Nous avons ces bornes pour les fonctions élémentaires classiques en simple précision. V. Lefèvre dans son stage de DEA et maintenant dans sa thèse s'intéresse au cas des fonctions élémentaires en double précision. Il a développé des programmes basés sur des approximations polynômiales, avec des polynômes ayant de tous petits degrés, permettant des calculs extrêmement rapides (environ 6 cycles d'horloge pour le calcul d'une exponentielle, dans

un petit intervalle, sur une machine à base de processeur `UltraSPARC`). A l'heure actuelle, il a testé l'exponentielle dans l'intervalle $[\frac{1}{2}, 1[$ (soit $2^{52}$ cas !) et la valeur correspondante de $m$ dans ce cas est 109 (cf [LMT97]). Les travaux de thèse de V. Lefèvre permettront peut être d'avoir des bornes similaires dans le cadre de la double précision du système flottant pour les fonctions élémentaires classiques. Mais les tests exhaustifs ne pourront probablement pas être effectués pour des précisions supérieures (double étendu ou quadruple précision par exemple).

Les récents résultats de théorie des nombres obtenus par Nesterenko et Waldschmidt seront très probablement améliorés. Mais l'ordre de grandeur des majorants de la taille des nombres à manipuler lors des calculs intermédiaires sera encore probablement trop important pour évaluer systématiquement les fonctions élémentaires avec ces bornes certaines.

Dans les cas les plus extrêmes, il faudra donc recourir à l'évaluation des fonctions élémentaires avec d'importantes précisions (de quelques centaines à peut être quelques millions de chiffres). Nous avons vu qu'il est tout à fait envisageable d'évaluer une fonction telle que l'exponentielle sur un million de bits en temps inférieur à une vingtaine de minutes. Les performances des ordinateurs augmentant sans cesse, ces temps de calcul diminueront de façon substantielle dans le futur. La taille des nombres que nous manipulerons dans le futur augmentera elle aussi, mais peut être pas dans la même mesure.

### 1.3.1 La stratégie de Ziv

Ziv a proposé dans [Ziv91] une stratégie multi-niveaux qui fournit une solution efficace au problème de l'arrondi exact des fonctions élémentaires. Sa stratégie est basée sur des essais successifs avec diverses précisions. Initialement, il choisit une valeur de $m$ un peu plus grande que $n$ ($m = n+20$ par exemple). Il calcule alors la fonction élémentaire avec une erreur de $2^{-m}$. Dans la majorité des cas cette précision sera suffisante pour arrondir exactement. Mais si cela n'est pas suffisant, il suffit de recommencer avec une précision supérieure. La détection des cas qui posent problème est très simple. Il y a une chaîne de 1 ou de 0 consécutifs dont le début dépend du mode d'arrondi considéré et qui s'étend jusqu'au dernier bit de la mantisse. Si la mantisse initiale est calculée avec $m = n + 20$ bits, selon nos hypothèses probabilistes, il n'y a qu'une chance sur $2^{20}$ que le dilemme du fabricant de tables se produise. A chaque étape, si la précision utilisée n'est pas suffisante, il suffit de recommencer avec une précision plus importante. On sait qu'il existe une précision qui permettra de résoudre le dilemme, donc le calcul termine toujours.

On a donc un ensemble de niveaux de calcul offrant des précisions de plus en plus importantes, où à chaque niveau, le calcul permet ou non d'arrondir exactement le résultat. La détection de l'échec ou du succès du niveau consiste en une simple détection d'occurrences particulières de chaînes de bits. Le temps moyen de calcul est très légèrement supérieur à celui du niveau le plus simple car la grande majorité des calculs sont effectués dans ce premier niveau.

### 1.3.2 Une solution multi-niveaux pour l'arrondi exact des fonctions élémentaires

Nous livrons ici quelques réflexions quant à la réalisation pratique de la stratégie de Ziv. Nous allons essayer d'argumenter un découpage en différents niveaux à l'aide de considérations arithmétiques et sur l'impact de l'architecture sur les performances du système flottant.

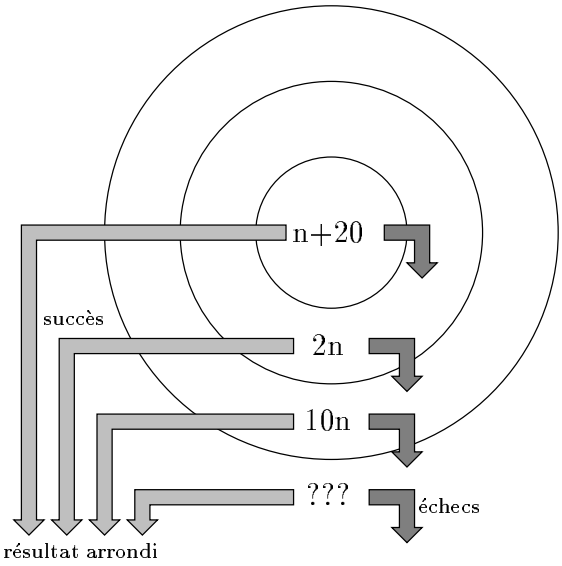



FIG. 1.4 – *La stratégie multi-niveaux de Ziv.*

Le premier niveau se doit d'être le niveau dont les calculs seront les plus rapides. En effet, c'est dans ce niveau que la grande majorité des calculs seront effectués. Il paraît alors assez naturel d'envisager une réalisation totalement matérielle de ce niveau. Le calcul de la précision intermédiaire de ce niveau est fonction de critères plus ou moins contradictoires :

- le temps d'évaluation d'une fonction élémentaire dans ce niveau;
- la fréquence des passages au niveau supérieur;

Le temps d'évaluation d'une fonction élémentaire croît de façon quadratique avec la précision. De plus, une implantation matérielle efficace n'est envisageable que pour des précisions pas trop importantes. Cette contrainte a donc tendance à donner de petites valeurs à $m$.

La fréquence des passages au niveau supérieur diminue rapidement lorsque $m$ augmente. En effet, selon nos hypothèses probabilistes, la probabilité de devoir passer au niveau supérieur si l'on calcule avec $m$ bits est de $2^{1-m}$. Pour limiter le nombre de passages au niveau supérieur, on aurait donc tendance à augmenter $m$.

Le compromis qui doit être réalisé ici dépend d'un très grand nombre de paramètres technologiques et algorithmiques. En effet, la vitesse de ce premier niveau conditionne grandement les performances globales de calcul du système flottant.

On peut toutefois remarquer que la logique nécessaire à la détection des cas qui nécessitent un traitement par le niveau supérieur est relativement simple. En effet, un arbre de cellules `ET` un autre de cellules `OU` sur les bits supplémentaires permettent de décider rapidement si l'arrondi a réussi ou non, et ce quel que soit le mode d'arrondi choisi. En pratique on a un matériel relativement similaire à celui de l'arrondi de la multiplication flottante.

Dans le cas de la simple précision ($n = 24$), on a vu que la plus grande valeur de $m$ trouvée lors des tests exhaustifs était de 54 bits. Les unités arithmétiques flottantes des machines qui ont un type double précision étendue réalisé complètement dans le circuit (les processeurs de compatibles PC) sont de bons candidats pour servir de base au premier niveau de notre stratégie. Il reste quand même à leur ajouter un peu de matériel supplémentaire pour effectuer la détection et un système

pour générer le passage au second niveau.

La réalisation des niveaux supérieurs est moins critique en terme de temps de calcul. Les calculs de ces niveaux ne seront que très peu employés. Toutefois, il convient de réaliser différents niveaux avec des précisions de plus en plus importantes afin de ne pas trop pénaliser le temps de calcul moyen. En effet, il n'existe pas d'algorithme d'évaluation des fonctions élémentaires qui soit performant, en terme de temps de calcul, quelle que soit la précision désirée. Il est donc important de ne pas trop augmenter rapidement la valeur de $m$ pour ne pas trop pénaliser le temps de calcul moyen du système complet.

Les algorithmes mis au point dans ces niveaux ne pourront très vraisemblablement pas être réalisés de façon matérielle. Il n'est pas rentable de concevoir un circuit ou une grosse portion de circuit qui ne sera que très peu utilisée voire jamais, puisqu'il est probable que les bornes théoriques ne seront pas atteintes.

Là encore, le choix de l'algorithme à utiliser pour chaque niveau est fortement dépendant de paramètres de l'implantation (performances des unités arithmétiques flottante et entière, performances de la hiérarchie mémoire...).

Pour des précisions allant jusqu'à quelques milliers de bits, l'*arithmétique en ligne* (voir la troisième partie de cette thèse) est probablement une solution élégante et efficace à ce problème. L'arithmétique en-ligne est une arithmétique série dont les chiffres circulent dans les opérateurs les poids forts en tête. La *E-méthode* proposée par Ercegovac [Erc77, EMT95] permet d'évaluer les fonctions élémentaires chiffre à chiffre tant que la précision suffisante n'est pas atteinte. L'*arrondi au vol* proposé par Ercegovac et Lang [EL92] permet d'arrondir le résultat d'un calcul en-ligne (écrit dans une notation redondante) directement sans devoir convertir auparavant dans un système plus conventionnel. La combinaison de ces deux algorithmes permettrait de pouvoir réaliser le calcul et l'arrondi chiffre par chiffre. Toutefois, le calcul en-ligne n'est plus performant par rapport à d'autres méthodes pour les très grandes précisions.

Dans le cas des précisions encore plus importantes, disons environ à partir de quelques dizaines de milliers de chiffres, les algorithmes comme ceux de Brent deviennent très performants, mais on retrouve alors la notion de niveau de calcul. Un stage de DEA est actuellement en cours au LIP sur la réalisation d'algorithmes pour le calcul des fonctions élémentaires sur quelques millions de bits.

Beaucoup de points restent à explorer pour réaliser effectivement l'arrondi exact des fonctions élémentaires. Comme, par exemple, comment gérer les interruptions ou les exceptions générées par les différents niveaux en cas d'échec. Quels doivent être les interactions avec le système d'exploitation et comment les prendre en compte dans les outils de développement.

### 1.3.3 Vers une extension de la norme IEEE 754 aux fonctions élémentaires

L'arrondi exact des fonctions élémentaires sera prochainement réalisable sur la plupart des machines au moins en double précision. Il est donc temps de songer à une extension de la norme actuelle à ces fonctions. Cette extension tout en préservant les concepts déjà fixés doit proposer des solutions spécifiques aux divers problèmes liés à l'évaluation des fonctions élémentaires.

Plusieurs questions se posent :

- Seul l'arrondi exact doit-il être proposé ? Ou bien un arrondi non forcément exact peut-il être mis en œuvre afin d'éviter les coûteux passages dans les niveaux supérieurs pour les applications demandant de grandes vitesses de calcul ?

- Dans le cas d'un arrondi non forcément exact, doit-on prévoir un mécanisme de drapeau qui indique que le résultat n'a pas pu être arrondi exactement, laissant ainsi le choix du traitement adéquat à l'utilisateur?
- L'arrondi exact des fonctions élémentaires dans toute la plage de représentation des nombres flottants sera peut être très coûteux en temps de calcul. Doit-on définir des petits domaines pour l'arrondi exact?

Un vaste débat sur ce sujet doit naître entre les utilisateurs, les concepteurs de machines et d'outils de développement afin de faire progresser la norme IEEE 754 vers une plus grande portabilité des algorithmes et vers de plus grands moyens pour valider ces algorithmes.

## 1.4 Conclusion

Nous avons montré que l'arrondi exact des fonctions élémentaires est effectivement réalisable. Alors que pendant longtemps on a cru que le coût en temps de calcul de cet arrondi serait prohibitif, il nous paraît évident qu'une stratégie fondée sur différents niveaux de calcul avec des précisions croissantes constitue une solution efficace à ce problème et dont le temps de calcul moyen sera relativement proche du temps de calcul des fonctions élémentaires dans les unités flottantes actuelles.

Les tests exhaustifs effectués en simple précision nous donnent déjà les bornes de la précision nécessaire pour effectuer les calculs intermédiaires. Nous avons vu aussi que dans le cas de la simple précision l'arrondi exact des fonctions élémentaires peut d'ores et déjà être réalisé en utilisant le matériel existant sur certains processeurs. Nous pensons réaliser dans l'avenir une bibliothèque de calcul certifié des fonctions élémentaires pour la simple précision.

Pour les précisions supérieures, des tests exhaustifs dans de petits intervalles et peut être une amélioration des résultats de théorie des nombres permettront peut être d'obtenir des bornes plus raisonnables que le million de bits actuel. Mais le calcul en multiprécision restera probablement le seul moyen d'effectuer l'arrondi exact des fonctions élémentaires quelle que soit la précision désirée. Nous travaillons donc à la réalisation d'un système de calcul de ces fonctions en très haute précision.

# Système semi-logarithmique

CHAPITRE

# 2 Système semi-logarithmique

LE système virgule flottante est aujourd'hui le système le plus utilisé pour représenter les nombres réels en machine. Mais d'autres systèmes de représentation des nombres réels ont été proposés. Parmi tous ces systèmes on peut citer :

- le système virgule fixe;
- le système logarithmique;
- le système SLI d'Olver;
- des systèmes rationnels;
- des modifications du système flottant.

La représentation en virgule fixe n'est principalement employée que pour des applications spécialisées car la "dynamique" des nombres représentables en virgule fixe est beaucoup trop faible pour bon nombre d'applications de calcul scientifique. Les algorithmes utilisés pour les opérations classiques sont plus simples que leurs équivalents dans le système flottant. Par exemple, l'addition de deux réels codés en virgule fixe s'effectue de la même façon que celle de deux entiers. Il n'y a pas de phase d'alignement des mantisses et de renormalisation du résultat qui augmentent la complexité des unités arithmétiques flottantes. Par contre, il est nécessaire de coder sur un grand nombre de bits les nombres réels dans les algorithmes qui manipulent des quantités d'ordres de grandeur différents.

Le système logarithmique [KR71, SA75, ST88, TGJR88, Hen89, ABCW92, Lew93], que nous allons présenter plus en détail dans la section 2.1, a été introduit pour accélérer le calcul des multiplications. Ce système ne peut pas constituer une alternative systématique au système flottant dans la majorité des problèmes. Cependant dans certaines applications, en traitement du signal par exemple, le fait de pouvoir effectuer très rapidement des algorithmes comportant beaucoup de multiplications est un avantage indéniable.

Le système SLI d'Olver [Olv87, Tur91, Tur93] permet de manipuler aussi bien des très grands que des très petit nombres, tout en permettant de manipuler des nombres d'un ordre de grandeur courant avec une bonne précision. Mais les opérations de base (addition, multiplication et division) dans ce système demandent un grand nombre de calculs intermédiaires et/ou de lectures de tables.

Il est possible de manipuler exactement des nombres rationnels sous forme de fractions continues [Vui83, KM85]. Cette représentation donne des algorithmes plus rapides qu'avec une représentation sous forme d'une fraction de deux entiers. Mais les principaux algorithmes de calcul restent plus complexes et plus coûteux que leurs équivalents du système flottant. Une autre solution pour

représenter les nombres rationnels est le *Slash number system* proposé par Matula et Kornerup dans [MK85]. Ils codent dans un même espace les deux composantes d'une fraction rationnelle. Un séparateur supplémentaire indique la taille relative du numérateur et du dénominateur.

Des modifications du système flottant ont été proposées dans le but supprimer les problèmes liés aux dépassements de capacités (*overflow* et *underflow*). On peut trouver dans [Yok92, MI81], par exemple, des systèmes où les tailles des champs de l'exposant et de la mantisse sont variables. On code, en plus des trois champs habituels (signe, exposant et mantisse), un champ supplémentaire qui donne la position de la séparation entre le champ de l'exposant et champ de la mantisse. Cette variabilité permet de représenter des réels ayant une dynamique beaucoup plus importante tout en conservant une bonne précision pour la représentation des nombres d'ordres de grandeur courants. Ces systèmes nécessitent toutefois des algorithmes un peu plus complexes qu'en virgule flottante.

Tous ces systèmes ont été conçus pour diverses raisons comme pour éviter les dépassements de capacité, améliorer la précision ou bien accélérer certains calculs. Malheureusement, il n'existe pas, à l'heure actuelle de système "universel" de représentation des nombres réels permettant d'effectuer très rapidement toutes les opérations avec une excellente précision.

Le système flottant qui offre un bon compromis entre la vitesse des calculs et la représentabilité des nombres est de loin le système le plus utilisé dans les ordinateurs actuels. Mais pour des applications particulières nécessitant beaucoup de multiplications et une faible précision, le système logarithmique peut offrir de meilleures performances en temps de calcul que le système flottant. Toutefois, le système logarithmique pose d'importants problèmes de mise en œuvre matérielle pour des précisions plus importantes du fait de la taille des tables nécessaires pour réaliser certaines opérations (addition).

Nous allons voir que le système que nous proposons, le *système semi-logarithmique*, permet de diminuer ce problème tout en conservant les avantages intrinsèques du système logarithmique. Ce travail a été réalisé en collaboration avec J.M. Muller et A. Scherbyna et a été publié dans [MST95].

Dans la première partie de ce chapitre nous présentons le système logarithmique et les algorithmes utilisés pour le calcul des opérations de base dans ce système. La deuxième partie est consacrée à la représentation des nombres réels dans le système semi-logarithmique. Nous décrivons les algorithmes de calcul des opérations de base (addition, multiplication, division, comparaison et conversion) et nous abordons leur implantation matérielle dans la troisième partie de ce chapitre. La quatrième partie est consacrée à l'étude de l'erreur relative de représentation du système semi-logarithmique et à sa comparaison avec les systèmes flottant et logarithmique. Enfin, nous présentons un exemple d'application du système semi-logarithmique dans le cadre de réseaux de neurones basés sur des ondelettes.

## 2.1 Le système logarithmique

Le système logarithmique, présenté en 1975 par Swartzlander et Alexpoulos dans [SA75], consiste à représenter un nombre réel par son signe et le logarithme de sa valeur absolue écrit en virgule fixe. Ce système permet d'accélérer considérablement les multiplications et les divisions puisqu'il suffit alors de faire la somme ou la différence des logarithmes des deux nombres considérés.

Ce système s'est avéré être particulièrement intéressant dans des applications où la précision

nécessaire n'est pas très importante et où le ratio

$$\frac{\text{nombre de multiplications}}{\text{nombre d'additions}}$$

est grand. On peut citer de nombreux exemples d'utilisation de ce système en traitement du signal et des images. Par exemple, on peut trouver dans [KM91] une méthode de tracé de courbes utilisant le système logarithmique. Différents systèmes de filtrage numérique basés sur des calculs dans le système logarithmique ont été proposés [KR71, HLD70, KPL80]. Certaines transformées comme la FFT peuvent se prêter à des réalisations matérielles rapides dans le système logarithmique [SSCNS83]. On peut aussi citer l'utilisation du système logarithmique dans certaines applications de contrôle numérique [LL86]. Enfin, plus récemment, on peut citer la réalisation d'une unité arithmétique logarithmique extrêmement performante dédiée à certains calculs en traitement du signal [Lew95].

### 2.1.1 Représentation des nombres réels dans le système logarithmique

Dans le système logarithmique en base 2, les nombres sont représentés selon le codage :

$$x = s_x \times 2^{e_x}$$

où

- $s_x$ est le signe de $x$.
- $e_x$ est l'exposant de $x$. A la différence du système flottant, $e_x$ est composé d'une partie entière et d'une partie fractionnaire.

La valeur 0 pose un petit problème de codage car cette valeur n'est pas directement représentable dans le système logarithmique. Deux solutions sont principalement utilisées pour résoudre ce problème. La première est l'utilisation d'un exposant biaisé comme dans le système flottant. La seconde utilise un bit supplémentaire qui code ou non la valeur 0. La seconde solution est utilisée dans des applications où la vitesse des calculs est la caractéristique la plus importante et où la précision nécessaire est relativement faible (voir par exemple [HCC96]).

Il n'existe pas de norme pour la représentation des nombres réels dans le système logarithmique. Jusqu'à présent, chaque nouvelle implantation utilise un codage plus ou moins spécifique à l'application cible.

Nous allons maintenant étudier les algorithmes qui permettent de réaliser les opérations classiques (addition, soustraction, multiplication, division, racine carrée, comparaisons et conversions) dans le système logarithmique.

### 2.1.2 Opérations de base dans le système logarithmique

#### Multiplication et division

Bien évidemment, la multiplication et la division s'obtiennent trivialement en effectuant respectivement la somme et la différence des exposants des deux nombres en question. Les dépassements de capacité vers l'infini ou vers zéro sont facilement détectables.

La multiplication et la division dans le système logarithmique ont à peu près la même complexité que l'addition et la soustraction dans le système flottant.

### Addition et soustraction

Si $a$ et $b$ sont deux nombres du système logarithmique, on obtient leur somme et leur différence en utilisant les relations suivantes :

$$\begin{cases} \log_2(a+b) = \log_2(a) + \log_2(1 + 2^{\log_2(b)-\log_2(a)}) \\ \log_2(a-b) = \log_2(a) + \log_2(1 - 2^{\log_2(b)-\log_2(a)}) \end{cases}$$

où $\log_2(1+2^x)$ et $\log_2(1-2^x)$ sont des fonctions qui sont tabulées.

Le principal inconvénient du système logarithmique réside dans la taille de ces tables. En effet, une implantation directe des opérations d'addition et de soustraction nécessite des tables de $2^n$ entrées pour des nombres dont les exposants ont $n$ bits fractionnaires. Ceci interdit une implantation simple du système logarithmique pour des applications nécessitant des précisions importantes.

Des techniques d'interpolation permettent toutefois de diminuer considérablement la taille de ces tables au prix d'un temps de calcul supérieur (voir par exemple [Tay83, ABCC89, Hen89]). Des unités arithmétiques logarithmiques proposant des précisions et des surfaces de circuit comparables au format simple précision du système flottant ont pu être réalisées grâce à ces techniques d'interpolation [LW91, Lew95, HCC96].

### Comparaisons

La comparaison de deux nombres de même signe se limite à la seule comparaison directe des exposants des deux opérandes. Si l'exposant est biaisé alors la comparaison s'effectue comme la comparaison de deux entiers positifs.

### Conversions système flottant $\leftrightarrow$ système logarithmique

Soient $x$ un nombre réel codé dans le système flottant et $y$ un nombre réel codé dans le système logarithmique, dont les écritures respectives sont :

$$\begin{aligned} x &= s_x \times m_x \times 2^{e_x} \\ y &= s_y \times 2^{e_y} \end{aligned}$$

où $(s_x, s_y) \in \{+1, -1\}^2$, $m_x$ est un réel de l'intervalle $[1, 2[$ codé sur $n$ bits, $e_x$ est un entier sur $k$ bits, et enfin $e_y$ est un réel écrit en virgule fixe avec $k$ bits de partie entière et $n$ bits de partie fractionnaire.

- *système flottant $\rightarrow$ système logarithmique*

$$e_y = e_x + \frac{\lfloor 2^n \log_2 m_x \rfloor}{2^n} \qquad \text{ou} \qquad e_y = e_x + \frac{\lfloor 2^n \log_2 m_x \rceil}{2^n}$$

  où $\lfloor x \rceil$ est l'entier le plus proche de $x$.

- *système logarithmique $\rightarrow$ système flottant*

$$e_x = \lfloor e_y \rfloor \qquad \text{et} \qquad m_x = \frac{\lfloor 2^n 2^{e_y - \lfloor e_y \rfloor} \rfloor}{2^n} \quad \text{ou} \quad m_x = \frac{\lfloor 2^n 2^{e_y - \lfloor e_y \rfloor} \rceil}{2^n}$$

Ici encore suivant le nombre de bits $n$, les fonctions $\log_2 z$ et $2^z$ seront soit complètement tabulées soit calculées à l'aide d'une technique combinant des lectures de tables et des interpolations.

On peut trouver dans [LW91] et [Lai91] une étude complète de la mise en œuvre matérielle de ces conversions pour la réalisation de systèmes de calcul hybrides (systèmes flottant et logarithmique cohabitant dans la même unité).

### Quelques autres opérations

Certaines opérations, comme l'élévation à une puissance entière ou la racine carrée peuvent être réalisées très rapidement dans le système logarithmique. En effet, si $n \in \mathbb{N}^+$ et si $a$ est un réel codé dans le système logarithmique, on a :

$$\begin{aligned} \log_2(a^n) &= n \times \log_2 a \\ \log_2(\sqrt{a}) &= \frac{\log_2 a}{2} \end{aligned}$$

Ces opérations sont particulièrement utiles en traitement du signal pour le filtrage numérique et pour le calcul de certaines transformées comme la FFT par exemple. Ceci explique que, pour certaines de ces applications qui ne nécessitent pas une précision très importante, le système logarithmique offre des implantations matérielles beaucoup plus performantes que le système flottant.

## 2.2 Le système semi-logarithmique

Le système semi-logarithmique est un système de représentation des nombres réels à mi-chemin entre le système flottant et le système logarithmique. Nous verrons dans la suite que l'on a, en fait, une famille de systèmes paramétrée par une valeur que nous noterons $k$. Nous présentons dans cette thèse le système semi-logarithmique en base 2, une extension à des bases différentes ne pose pas de problème particulier.

Le système semi-logarithmique reprend en partie l'idée de base du système logarithmique qui permet une augmentation de la vitesse d'exécution des algorithmes de multiplication et de division, à savoir le fait de coder les nombres par le logarithme en virgule fixe de leur valeur absolue. Mais au lieu de supprimer totalement la mantisse, le système semi-logarithmique conserve une mantisse mais dont l'intervalle de variation (la mantisse est aussi normalisée) est beaucoup plus petit. En fait, les mantisses dans le système semi-logarithmique sont dans un intervalle du type $[1, 1+\epsilon]$. Le paramètre $k$ code le nombre de zéros qui sont les bits de poids forts de la partie fractionnaire de la mantisse. C'est-à-dire que l'on a une écriture sur $n$ bits de la mantisse du style

$$1.\overbrace{\underbrace{0000\ldots000}_{k \text{ zéros}}\texttt{xxxx}\ldots\texttt{xx}}^{n \text{ chiffres}}.$$

Comme dans le système flottant avec le bit implicite, on connaît toujours la valeur des $k+1$ premiers bits de la mantisse. Ces bits ne sont donc pas stockés physiquement. Les $k$ bits "gagnés" sur la mantisse sont utilisés pour stocker la partie fractionnaire de l'exposant réel. Cet "échange" de bits entre la mantisse et l'exposant fait que à nombre de bits effectivement utilisés pour les codages égal, le système flottant et le système semi-logarithmique ont des précisions comparables. Comme dans le système flottant, tous les nombres sont représentables avec une certaine précision dans le système semi-logarithmique.

L'idée de base du système semi-logarithmique est de pouvoir représenter les nombres en utilisant un codage du style $(1+\epsilon) \times 2^e$ où $e$ comporte des chiffres fractionnaires. Les algorithmes de multiplication et de division sont alors particulièrement simples. Par exemple, si on calcule le produit $x \times y = \left((1+\epsilon_x) \times 2^{e_x}\right) \times \left((1+\epsilon_y) \times 2^{e_y}\right)$, on a alors le résultat $(1+\epsilon_x) \times (1+\epsilon_y) \times 2^{e_x+e_y}$

qui peut se réduire à de simples additions si $k$ est bien choisi. En effet, $(1+\epsilon_x)\times(1+\epsilon_y) \simeq 1+\epsilon_x+\epsilon_y$ car $\epsilon_x\epsilon_y$ devient négligeable à partir d'une certaine valeur de $k$ ($\epsilon_x\epsilon_y$ est inférieur à $2^{-n}$ si $k > \frac{n}{2}$). Le temps de calcul de la multiplication dans le système semi-logarithmique est donc comparable à celui de l'addition flottante.

La principale différence entre le système semi-logarithmique et le système logarithmique est le nombre de chiffres fractionnaires de l'exposant. C'est ici l'apport le plus important du système semi-logarithmique par rapport au système logarithmique. En effet, dans ces deux systèmes les algorithmes d'addition et de soustraction utilisent des tables qui ont pour entrée les bits de la partie fractionnaire des exposants. Le fait qu'il y ait moins de bits dans la partie fractionnaire de l'exposant du système semi-logarithmique diminue très significativement la taille de ces tables. Bien évidemment, la mantisse complique un peu les traitements, mais nous verrons qu'ils se ramènent à quelques lectures de petites tables et quelques additions de nombres codés en virgule fixe.

Il est donc possible d'envisager l'implantation matérielle d'unités arithmétiques utilisant le système semi-logarithmique pour des précisions beaucoup plus importantes qu'avec le système logarithmique. De plus, en intégrant des techniques d'interpolation dans nos algorithmes, il est possible de réaliser des unités arithmétiques ayant de petites surfaces de circuit.

Dans la première partie de cette section, nous allons étudier plus en détail la représentation des nombres réels dans le système semi-logarithmique et comparer les différentes caractéristiques de ce système avec les systèmes flottant et logarithmique.

Nous étudierons ensuite les algorithmes permettant le calcul des opérations de base dans le système semi-logarithmique : addition, soustraction, multiplication, division, comparaisons et conversions. Nous aborderons aussi l'implantation matérielle de ces opérations.

Nous terminerons ce chapitre en étudiant l'erreur relative de représentation du système semi-logarithmique et en donnant un exemple d'application de ce système en cours de réalisation.

### 2.2.1 Représentation des nombres réels dans le système semi-logarithmique

Le nombre réel $x$ différent de 0 est représenté dans le système semi-logarithmique de paramètre entier $k$ en utilisant le codage :

$$x = s_x \times m_{k,x} \times 2^{e_{k,x}}$$

où

- $s_x$ est le signe de $x$.
- $m_{k,x}$ est la mantisse de $x$ dont les $k$ premiers bits fractionnaires sont nuls.
- $e_{k,x}$ est l'exposant de $x$. Cet exposant est codé en virgule fixe avec $k$ bits fractionnaires.

Les mantisses dans le système semi-logarithmique sont normalisées tout comme les mantisses du système flottant. Le fait que la mantisse soit normalisée empêche de représenter directement la valeur 0. On utilisera en pratique pour le codage de 0 une valeur "réservée" comme dans le système flottant, ou un bit supplémentaire pour les applications à haute vitesse et à faible précision.

Nous devons maintenant trouver la borne supérieure qui définit les valeurs extrêmes de la mantisse afin de pouvoir complètement spécifier la représentation et les phases de normalisation des différents algorithmes de calcul des opérations de base (la borne inférieure est 1).

L'exposant de $x$ est composé de $k$ bits dans sa partie fractionnaire, $e_{k,x}$ est donc le multiple de $2^{-k}$ qui vérifie :

$$2^{e_{k,x}} \leq |x| < 2^{e_{k,x}+2^{-k}} \tag{2.1}$$

La valeur de $e_{k,x}$ se déduit aisément de $x$ puisqu'elle est une valeur approchée de $\log_2 |x|$. En fait, $e_{k,x}$ est le résultat de $\log_2 |x|$ tronqué au $k^{\text{ième}}$ bit fractionnaire. On a donc

$$e_{k,x} = \frac{\lfloor 2^k \log_2 |x| \rfloor}{2^k}$$

La valeur de la mantisse se déduit directement de $x$ et de $e_{k,x}$. $m_{k,x}$ est le facteur multiplicatif qui permet de corriger l'erreur commise lors du calcul de la valeur approchée de $\log_2 |x|$ en tronquant $e_{k,x}$ au $k^{\text{ième}}$ bit fractionnaire. On a alors :

$$m_{k,x} = \frac{|x|}{2^{e_{k,x}}}$$

De l'équation 2.1 on déduit :

$$1 \leq m_{k,x} = \frac{|x|}{2^{e_{k,x}}} < 2^{\frac{1}{2^k}}$$

Dans les étapes de normalisation de la mantisse lors d'une opération, il faudra comparer le résultat intermédiaire et la valeur $2^{\frac{1}{2^k}}$. Cette valeur est égale à $e^{\frac{\ln 2}{2^k}}$. L'écriture binaire de cette valeur n'est pas particulièrement simple, nous allons donc chercher un majorant de $2^{\frac{1}{2^k}}$ ayant une écriture binaire plus simple. On peut montrer facilement que pour $\alpha \in ]0,1[$ on a :

$$1 + \alpha > 2^{\alpha}$$

Ce qui donne en posant $\alpha = \frac{1}{2^k}$ avec $k \geq 0$

$$2^{\frac{1}{2^k}} \leq 1 + \frac{1}{2^k}$$

Finalement, on a donc comme normalisation de la mantisse :

$$1 \leq m_{k,x} < 1 + \frac{1}{2^k}$$

On a maintenant deux types de contraintes pour la normalisation de la mantisse. La différence entre ces deux encadrements est la simplicité du matériel qui permet d'effectuer la comparaison avec la borne supérieure. En effet, le circuit permettant de comparer la mantisse intermédiaire et $1 + \frac{1}{2^k}$ est plus simple que celui avec $2^{\frac{1}{2^k}}$.

Il existe donc deux formes pour la représentation des nombres réels dans le système semi-logarithmique de paramètre $k$. Ces deux formes sont spécifiées par les définitions suivantes :

**Définition 2.1 (Forme canonique)** *Soit $k$ un entier positif. Tout nombre réel non nul $x$ est représenté dans la forme canonique du système semi-logarithmique de paramètre $k$ par le triplet $(s_x, m_{k,x}, e_{k,x})$ satisfaisant :*

- $s_x = \pm 1$
- $e_{k,x}$ *est un multiple de* $2^{-k}$

- $1 \leq m_{k,x} < 2^{\frac{1}{2^k}}$
- $x = s_x \times m_{k,x} \times 2^{e_{k,x}}$

**Définition 2.2 (Forme Générale)** *Soit $k$ un entier positif. Tout nombre réel non nul $x$ est représenté dans la forme générale du système semi-logarithmique de paramètre $k$ par un triplet $(s_x, m_{k,x}, e_{k,x})$ satisfaisant :*

- $s_x = \pm 1$
- $e_{k,x}$ *est un multiple de* $2^{-k}$
- $1 \leq m_{k,x} < 1 + 2^{-k}$
- $x = s_x \times m_{k,x} \times 2^{e_{k,x}}$

Le nombre réel $x$ est représenté dans le *système semi-logarithmique* de paramètre $k$ avec $n$ bits de mantisse par le signe $s_x$, l'exposant $e_{k,x}$ et de l'écriture en virgule fixe de la mantisse $m_{k,x}$ tronquée au $n^{\text{ième}}$ bit fractionnaire. En pratique, avec $1 \leq m_{k,x} < 1 + 2^{-k}$, $m_{k,x}$ a une écriture binaire de la forme :

$$1.\overbrace{\underbrace{0000\ldots000}_{k \text{ zéros}}\texttt{xxxx}\ldots\texttt{xx}}^{n \text{ chiffres}}$$

Comme les $k+1$ premiers chiffres de $m_{k,x}$ sont parfaitement connus à l'avance, ils ne sont pas effectivement stockés (c'est un peu l'équivalent du "bit implicite" dans le système flottant IEEE 754). De la même façon que dans le système flottant (où les mantisses sont aussi normalisées), une représentation spéciale de la valeur 0 doit être spécifiée.

Afin de simplifier les notations, nous considérerons dans la suite que le paramètre $k$ est implicite. C'est-à-dire, que l'on note "$m_x$" et "$e_x$" à la place de "$m_{k,x}$" et de "$e_{k,x}$".

### 2.2.2 Particularités du système semi-logarithmique

On peut remarquer les points suivants :

- Si $k = 0$, le système semi-logarithmique de paramètre $k$ est réduit au système virgule flottante à $n$ bits de mantisse.
- Si $k \geq n$ alors le système semi-logarithmique de paramètre $k$ est réduit au système logarithmique.
- La forme canonique du système semi-logarithmique est une représentation *non redondante* (c'est-à-dire qu'il y a unicité des représentations). Les comparaisons sont facilement réalisables dans cette forme. En effet, avec un codage constitué des poids forts vers les poids faibles du signe, de l'exposant et de la mantisse, les comparaisons peuvent être effectuées comme si il s'agissait d'entiers.
- La forme générale du système semi-logarithmique est une représentation *redondante*. Par exemple, si $k = 1$ alors la valeur $\sqrt{2}$ a deux écritures possibles : $1.000\ldots0 \times 2^{0.1}$ ($1 \times 2^{\frac{1}{2}} = \sqrt{2}$) et $1.011010100000100111\ldots \times 2^{0.0}$ (la mantisse est alors l'écriture binaire classique de $\sqrt{2}$). Les comparaisons sont plus complexes à réaliser dans la forme générale que dans la

forme canonique, mais nous verrons dans la suite que les algorithmes sont beaucoup plus simples à implanter dans la forme générale que dans la forme canonique car la condition $1 \leq m_{k,x} < 1 + 2^{-k}$ est plus simple à vérifier pratiquement que $1 \leq m_{k,x} < 2^{\frac{1}{2^k}}$.

Toutefois, la conversion de la forme générale vers la forme canonique est facilement réalisable. Supposons que $x = s_x \times m_x \times 2^{e_x}$ est représenté dans la forme générale. Il faut comparer $m_x$ avec $\rho_k = 2^{2^{-k}}$ : si $m_x < \rho_k$ alors le nombre est déjà représenté dans la forme canonique, sinon ($m_x \geq \rho_k$) il faut ajouter $2^{-k}$ à $e_x$ et diviser $m_x$ par $\rho_k$. Le résultat obtenu est alors la représentation de $x$ dans la forme canonique.

Le paramètre $k$ permet de choisir différents compromis entre le système flottant et le système logarithmique. Nous verrons dans la suite pour certaines valeurs de $k$, les algorithmes de calcul des fonctions de base sont particulièrements simples.

Tout comme dans le système flottant, il est possible de calculer l'arrondi d'une valeur pour chacun des modes d'arrondi classiques. On a :

- Arrondi vers 0 :

$$\mathcal{Z}(x) = s_x \times \frac{\left\lfloor 2^n \times \frac{|x|}{2^{\lfloor 2^k \log_2 |x| \rfloor / 2^k}} \right\rfloor}{2^n} \times 2^{\lfloor 2^k \log_2 |x| \rfloor / 2^k}$$

- Arrondi vers $\pm\infty$ :

$$\mathcal{I}(x) = s_x \times \frac{\left\lceil 2^n \times \frac{|x|}{2^{\lfloor 2^k \log_2 |x| \rfloor / 2^k}} \right\rceil}{2^n} \times 2^{\lfloor 2^k \log_2 |x| \rfloor / 2^k}$$

- Arrondi au plus près :

$$\mathcal{N}(x) = s_x \times \frac{\left\lfloor 2^n \times \frac{|x|}{2^{\lfloor 2^k \log_2 |x| \rfloor / 2^k}} \right\rceil}{2^n} \times 2^{\lfloor 2^k \log_2 |x| \rfloor / 2^k}$$

où $\lfloor u \rceil$ est l'entier le plus proche de $u$ (un choix spécial doit être fait si $2^n m_{k,x}$ est un multiple pair de $\frac{1}{2}$).

### 2.2.3 Le système semi-logarithmique comme système logarithmique "à deux bases"

Nous avons présenté dans la section 2.1 le système logarithmique en base 2. Dans ce système, un nombre réel est représenté par le logarithme en base 2 de sa valeur absolue. Un autre choix naturel pour la base d'un système de représentation des nombres réels est la base $e$ (la base des logarithmes népériens). Les bases 2 et $e$ ont des avantages et des inconvénients respectifs. Le principal avantage de la base 2 est que la multiplication d'un nombre écrit en virgule fixe (en base 2) par une *puissance entière* (positive ou négative) de 2 se résume à un simple décalage. Ceci est particulièrement utile pour le calcul des additions et pour les conversions. Le principal avantage de la base $e$ est que lorsque $\epsilon$ est petit on a :

$$\exp(\epsilon) \simeq 1 + \epsilon$$

Pour résumer, on peut dire que si $x$ est entier, $2^x$ est calculé simplement et que lorsque $y$ est un petit réel, $e^y$ est calculé simplement.

Le système semi-logarithmique peut être vu comme un système logarithmique utilisant les deux bases 2 et $e$ en même temps et en tirant partie de leurs avantages respectifs. En effet,

$$1.\overbrace{\underbrace{0000\ldots000}_{k \text{ zéros}}\texttt{xxxx}\ldots\texttt{xx}}^{n \text{ chiffres}} \times 2^{e_x} \quad = \quad (1+\epsilon_x) \times 2^{e_x}$$

où $\epsilon_x$ est très petit (plus petit que $2^{-k}$) et où $e_x$ est un multiple de $2^{-k}$. On a alors

$$x = (1+\epsilon_x) \times 2^{e_x} \simeq e^{\epsilon_x} \times 2^{e_x} = 2^{e_x + \epsilon_x/\ln 2} = 2^{e_x+\epsilon'_x}$$

où $\epsilon'_x = \epsilon_x / \ln 2$.

On peut donc voir le système semi-logarithmique comme un système logarithmique à deux bases, la base 2 pour $e_x$ et la base $e$ pour $\epsilon_x$. Nous verrons l'influence de cette représentation dans les algorithmes de certaines opérations de base.

## 2.3 Opérations dans le système semi-logarithmique

Nous décrivons dans cette section les algorithmes qui permettent d'effectuer les opérations de base dans le système semi-logarithmique et les conversions entre le système flottant et le système semi-logarithmique :

- multiplication
- division
- addition / soustraction
- comparaison
- conversion système flottant $\leftrightarrow$ système semi-logarithmique

Pour chacun des algorithmes, nous étudierons les choix du paramètre $k$ qui conduisent à des implantations matérielles rapides. Nous aborderons l'étude du matériel nécessaire pour la réalisation de ces algorithmes. Enfin, afin d'avoir une idée de l'ordre de grandeur du temps de calcul de ces opérations, nous effectuerons une comparaison entre leur réalisation dans le système semi-logarithmique et leur réalisation dans le système flottant.

### 2.3.1 Multiplication

Nous désirons calculer le produit de $x$ et de $y$ représentés respectivement dans le système semi-logarithmique par $s_x \times m_x \times 2^{e_x}$ et $s_y \times m_y \times 2^{e_y}$. Ici encore le paramètre $k$ est implicite. L'algorithme

2.1 permet de calculer ce produit.

**Algorithme 2.1 : Multiplication dans le système semi-logarithmique**

| | |
|---|---|
| $s \leftarrow s_x \times s_y$ | Le signe du résultat final |
| $e \leftarrow e_x + e_y$ | L'exposant approché |
| $m \leftarrow m_x \times m_y$ | Si $k > n/2$ alors $m_x \times m_y$ se réduit à une addition ($m_x = 1 + \epsilon_1$, $m_y = 1 + \epsilon_2$, avec $\epsilon_1, \epsilon_2 < 2^{-n/2}$, $m_x \times m_y = 1 + \epsilon_1 + \epsilon_2 + \epsilon_1\epsilon_2$, le produit $\epsilon_1\epsilon_2$ est négligeable devant $2^{-n}$) |
| | $\left[m = 1.000\ldots0m_{k-1}m_k m_{k+1}\ldots m_n\right]$ La mantisse $m$ est comprise entre 1 et $1 + 2^{-k+1} + 2^{-2k}$, les chiffres de poids $2^{-k+1}$ et $2^{-k}$ de $m$ ($m_{k-1}$ et $m_k$) peuvent donc être différents de zéro |
| $(\alpha, 2^\alpha) \leftarrow \mathsf{Table}_\times(m_{k-1},\ m_k,\ m_{k+1})$ | On lit les valeurs de $\alpha$ et de $2^\alpha$ dans une petite table à 8 entrées (avec $m_{k-1}$, $m_k$ et $m_{k+1}$ comme bits d'adresse) avec $\alpha = \frac{\lfloor -\log_2(m^*)\times 2^k \rceil}{2^k}$, où $\lfloor u \rceil$ est l'entier le plus proche de $u$ et $m^* = 1.000\ldots0m_{k-1}m_k m_{k+1}$ |
| $\hat{m} \leftarrow m \times 2^\alpha$ | Peut être réduit à une addition si $k > n/2 + 2$ |
| $\hat{e} \leftarrow e - \alpha$ | |
| If $\hat{m} \geq 1$ Then | |
| Return $(s, \hat{m}, \hat{e})$ | |
| Else | |
| Return $(s, \hat{m} \times 2^{2^{-k}}, \hat{e} - 2^{-k})$ | Si $k > n/2 + 2$ le produit $\hat{m} \times 2^{2^{-k}}$ peut être réduit à une addition. Nos simulations indiquent que ce cas est rare |

### Preuve de l'algorithme

Il faut s'assurer que la mantisse du résultat est bien comprise entre 1 et $1+2^{-k}$. De la définition de $\alpha$ :

$$\alpha = \frac{\lfloor -\log_2(m^*) \times 2^k \rceil}{2^k}$$

on déduit aisément :

$$-\log_2 m^* - 2^{-k-1} \leq \alpha \leq -\log_2 m^* + 2^{-k-1}$$

par conséquent :

$$\frac{m}{m^*} \times 2^{-2^{-k-1}} \leq m \times 2^\alpha \leq \frac{m}{m^*} \times 2^{+2^{-k-1}}$$

Le terme $\frac{m}{m^*}$ est égal à $1 + \frac{m-m^*}{m^*}$. La différence $m - m^*$ est plus petite que $2^{-k-1}$ et on a supposé que $m^* \geq 1 + 2^{-k}$ (si ce n'est pas le cas alors les bits $m_{k-1}$ et $m_k$ sont égaux à zéro, et $m$ est la mantisse du résultat). Donc

$$\frac{m}{m^*} \leq 1 + \frac{2^{-k-1}}{1 + 2^{-k}}$$

ce qui donne

$$\begin{aligned} 2^{2^{-k-1}} \times \frac{m}{m^*} &\leq \left(1+2^{-k-1}\right)\left(1+\frac{2^{-k-1}}{1+2^{-k}}\right) \\ &\leq \frac{1}{1+2^{-k}}\left(1+2^{-k-1}\right)\left(1+2^{-k}+2^{-k-1}\right) \\ &\leq \frac{1}{1+2^{-k}}\left(1+2\times 2^{-k}+2^{-2k-1}+2^{-2k-2}\right) \\ &< \frac{1}{1+2^{-k}}\left(1+2\times 2^{-k}+2^{-2k}\right) \\ &= \frac{\left(1+2^{-k}\right)^2}{1+2^{-k}} = 1+2^{-k} \end{aligned}$$

Si $m \times 2^{\alpha} < 1$, alors (dès que $m/m^* \geq 1$):

$$2^{-2^{-k-1}} \leq m \times 2^{\alpha} < 1$$

soit finalement

$$1 < m \times 2^{\alpha} \times 2^{2^{-k}} < 2^{2^{-k}} \leq 1+2^{-k}$$

❑

**Implantation matérielle de la multiplication dans le système semi-logarithmique**

Pour déterminer avec précision le matériel nécessaire, nous avons besoin de définir plus exactement le codage des nombres. La figure 2.1 représente le codage des nombres dans le système semi-logarithmique avec la taille des différents champs en nombre de bits. L'exposant est un réel avec $p$ bits de partie entière et $k$ bits de partie fractionnaire.

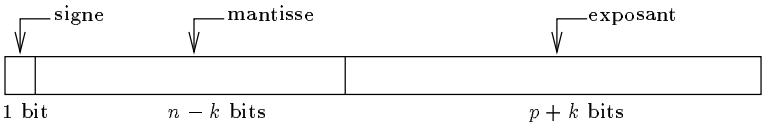



FIG. 2.1 – *Codage des nombres dans le système semi-logarithmique.*

De plus, nous nous plaçons dans le cas d'un système semi-logarithmique où la valeur du paramètre $k$ permet de réaliser les simplifications des multiplications en additions, comme cela a été vu dans l'algorithme (c'est-à-dire que $k > \frac{n}{2}+2$).

Le matériel nécessaire à l'implantation matérielle de l'algorithme de multiplication dans le système semi-logarithmique est relativement simple. Pour le calcul du signe, une seule cellule `XOR` suffit. Le calcul de l'exposant nécessite 2 additionneurs à $p+k$ bits, une table avec 3 bits d'entrée et de valeurs sur $k$ bits (taille $8 \times k$ bits) et un additionneur spécifique à $p+k$ bits qui additionne son entrée avec la constante $-2^{-k}$. Le calcul de la mantisse demande un additionneur à $k$ bits, une table avec 3 bits d'entrée et de valeurs sur $k$ bits (taille $8 \times k$ bits), deux additionneurs à $k+2$ bits et un comparateur à $k+2$ bits.

Nous n'avons pas, pour le moment, de résultat de synthèse logique pour la réalisation de nos algorithmes. Quelques petits problèmes techniques font que les outils de synthèse nécessaires pour cette étude ne nous ont toujours pas été livrés (attente depuis plusieurs mois).

Il n'est donc pas possible actuellement de donner des caractéristiques temporelles précises pour l'implantation de cet algorithme. Mais on peut toutefois estimer que son temps de calcul doit être assez proche du temps de calcul d'une addition flottante étant donnée la similarité des opérations de base mises en jeu.

### 2.3.2 Division

On cherche le quotient de $x$ et de $y$ avec $x = s_x \times m_x \times 2^{e_x}$ et $y = s_y \times m_y \times 2^{e_y}$ représentés dans le système semi-logarithmique de paramètre $k$ en utilisant l'algorithme 2.2.

**Algorithme 2.2 : Division dans le système semi-logarithmique**

$s \leftarrow s_x \times s_y$ — Le signe du résultat final

$e \leftarrow e_x - e_y$ — L'exposant approché

$m \leftarrow \frac{m_x}{m_y}$ — Si $k > n/2$ alors $m_x/m_y$ peut être réduit à une addition ($m_x = 1 + \epsilon_1$, $m_y = 1 + \epsilon_2$, avec $\epsilon_1, \epsilon_2 < 2^{-n/2}$, d'où $m_x/m_y = 1 + \epsilon_1 - \epsilon_2 - \epsilon_1\epsilon_2 + \epsilon_2^2 + \ldots$, et tous les termes sauf $1 + \epsilon_1 - \epsilon_2$ peuvent être négligés). De plus, si $k \leq n/2$, le fait que $m_y$ est très proche de 1 peut être utilisé afin d'accélérer la division en utilisant une méthode de division itérative telle que celle proposée par Goldschmidt dans [AEGP67]

$$\left[ \quad m = \begin{cases} 1.000\ldots00m_{k+1}m_{k+2}\ldots \\ \text{or} \\ 0.111\ldots11m_{k+1}m_{k+2}\ldots \end{cases} \quad \right]$$

De $1 \leq m_x < 1 + 2^{-k}$ et $1 \leq m_y < 1 + 2^{-k}$, on déduit $\frac{1}{1+2^{-k}} < m = \frac{m_x}{m_y} < 1 + 2^{-k}$, ce qui implique $1 - 2^{-k} < m < 1 + 2^{-k}$

If $m_k = 0$ Then

  Return $(s, m, e)$ — Le résultat final

Else

  $(\alpha, 2^\alpha) \leftarrow \mathsf{Table}_{\div}(m_{k+1})$ — Les valeurs de $\alpha$ et de $2^\alpha$ sont tabulées dans une petite table à 2 entrées avec $\alpha = \frac{\lfloor -\log_2(m^* \times 2^k) \rfloor}{2^k}$, où $m^* = 0.111\ldots11m_{k+1}$

  $\hat{m} \leftarrow m \times 2^\alpha$ — Peut être réduit à une addition si $k > n/2 + 2$

  $\hat{e} \leftarrow e - \alpha$

  If $\hat{m} \geq 1$ Then

    Return $(s, \hat{m}, \hat{e})$

  Else

    Return $(s, \hat{m} \times 2^{2^{-k}}, \hat{e} - 2^{-k})$ — Si $k > n/2 + 2$ le produit $\hat{m} \times 2^{2^{-k}}$ se réduit à une addition

La démonstration de cet algorithme utilise le même genre d'argumentation que la démonstration de l'algorithme de multiplication.

Le matériel nécessaire pour la division est relativement similaire à celui de la multiplication. Les tables sont quatre fois plus petites dans l'opérateur de division mais le contrôle est un peu plus complexe. Les performances des opérateurs de multiplication et de division dans le système semi-logarithmique sont donc très proches.

### 2.3.3 Addition et Soustraction

On désire calculer $x \pm y$ où $x = s_x \times m_x \times 2^{e_x}$ et $y = s_y \times m_y \times 2^{e_y}$ sont deux nombres réels représentés dans la forme générale du système semi-logarithmique de paramètre $k$. L'algorithme d'addition (2.3) dans le système semi-logarithmique est proche de celui de l'addition flottante. En effet, il consiste à aligner les mantisses (réécrire les deux opérandes avec le même exposant), additionner les mantisses alignées et renormaliser le résultat.

**Algorithme 2.3 : Addition/Soustraction dans le système semi-logarithmique**

| | |
|---|---|
| If $e_x < e_y$ Then | |
| Exchange$(x, y)$ | |
| $u \leftarrow \lfloor e_x - e_y \rceil$, $v \leftarrow e_x - e_y - u$ | $u$ est un entier, $v$ vérifie $0 \leq \lvert v \rvert \leq 1/2$ |
| $m_y \leftarrow \mathsf{Rshift}(m_y, u)$ | Décalage de $u$ bits vers la droite de $m_y$ |
| $\beta \leftarrow \mathsf{Table}_{\pm,\beta}(v_1, v_2, \ldots, v_{k-1})$ | La valeur $\beta = 2^{-v}$ est tabulée dans une+ table à $(k-1)$ bits d'adresse |
| $m_y^* \leftarrow m_y \times \beta$ | |
| $p \leftarrow s_x \times m_x \pm s_y \times m_y^*$ | Addition $\Rightarrow +$, soustraction $\Rightarrow -$ |
| $s \leftarrow \mathrm{sign}(p)$ | Le signe du résultat final |
| $p \leftarrow \lvert p \rvert$ | |
| If $p \geq 2$ Then | |
| $m \leftarrow \mathsf{Rshift}(p, 1)$ | $m$ est le décalage de 1 bit vers la droite de $p$ |
| $e \leftarrow e_x + 1$ | |
| Else | |
| If $p = 0$ Then | |
| Return $(s_0, m_0, e_0)$ | $(s_0, m_0, e_0)$ est le code spécial choisi pour la valeur 0 |
| $j \leftarrow \#_0(p)$ | $\#_0(p)$ donne le nombre de zéros entre le chiffre de poids $2^0$ et le "1" le plus significatif de $p$ |
| $m \leftarrow \mathsf{Lshift}(p, j)$ | $m$ est le décalage de $j$ bits vers la gauche de $p$ |
| $e \leftarrow e_x - j$ | |
| $(\alpha, 2^\alpha) \leftarrow \mathsf{Table}_{\pm,\alpha}(m_1, \ldots, m_{k+1})$ | Lecture des valeurs $\alpha$ et $2^\alpha$ définies ci-dessous dans une table à $(k+1)$ bits d'entrée (avec $m_1$, $m_2$, ..., $m_{k+1}$ comme bits d'adresse) avec $\alpha = \frac{\lfloor -\log_2(m^*) \times 2^k \rceil}{2^k}$ |
| $\hat{m} \leftarrow m \times 2^\alpha$ | |
| $\hat{e} \leftarrow e - \alpha$ | |
| If $\hat{m} \geq 1$ Then | |
| Return $(s, \hat{m}, \hat{e})$ | Le résultat final |
| Else | |
| Return $(s, \hat{m} \times 2^{2^{-k}}, \hat{e} - 2^{-k})$ | Si $k > n/2 + 2$ le produit $\hat{m} \times 2^{2^{-k}}$ se réduit à une addition |

Si $k > n/2+2$, il n'y a qu'une seule "grande multiplication" dans cet algorithme. Il est possible d'éviter cette multiplication de deux nombres de $n$ bits en modifiant légèrement l'algorithme. Si au lieu de retourner seulement $\alpha$ et $2^{\alpha}$, la table retourne aussi la valeur $2^{-\alpha}$, on peut alors calculer $\hat{m}$ avec $(m - 2^{-\alpha}) \times 2^{\alpha} + 1$. Il est facile de montrer que $m - 2^{-\alpha} < 2^{-k+1}$, donc la multiplication $(m - 2^{-\alpha}) \times 2^{\alpha}$ est une multiplication d'un nombre de $n - k + 1$ bits par un nombre de $n$ bits. Si $k > n/2$, ceci conduit à une réduction significative de la taille du multiplieur utilisé et du temps de calcul. De plus, cette méthode n'augmente pas la quantité de mémoire nécessaire. En effet, seuls $n-k+1$ bits de $2^{-\alpha}$ sont nécessaires (les $k-1$ chiffres les plus significatifs de $2^{-\alpha}$ s'annulent lors de l'addition avec $m$), et seuls les $n - k + 1$ bits les plus significatifs de $2^{\alpha}$ sont nécessaires (l'influence des bits de poids faibles est négligeable).

Contrairement aux algorithmes de multiplication et de division, l'algorithme d'addition et de soustraction demande l'utilisation d'une table de grande taille (avec $2^{k+1}$ valeurs). Toutefois, il faut se souvenir que les algorithmes d'addition et de soustraction dans le système logarithmique sans interpolation demandent des tables avec $2^n$ valeurs où $k << n$.

Si la table avec $2^{k+1}$ éléments ne peut pas réalisée, il est possible d'utiliser des tables avec seulement $2^{\frac{k+1}{2}+1}$ éléments et en décomposant le calcul de $\hat{m}$ en deux temps :

① Soit $j = \frac{k+1}{2}$. Dans une table à $(j+1)$ bits d'adresse $(m_1, m_2, \ldots, m_{j+1})$ on lit les valeurs de $\alpha_1$ et $2^{\alpha_1}$ qui vérifient :

$$\alpha_1 = \frac{\lceil -\log_2(1.m_1 m_2 \ldots m_{j+1}) \times 2^k \rceil}{2^k}$$

et on calcule $m^{(1)} = m \times 2^{\alpha_1}$. On montre facilement que $m^{(1)}$ est entre 1 et $1 + 2^{-j+1}$.

② Dans une table à $(j+1)$ bits d'adresse $(m_j^{(1)}, m_{j+1}^{(1)}, \ldots, m_{k+1}^{(1)})$ on lit les valeurs de $\alpha_2$ et $2^{\alpha_2}$ qui vérifient :

$$\alpha_2 = \frac{\left\lfloor -\log_2\left(1.000\ldots0 m_j^{(1)} m_{j+1}^{(1)} \ldots m_{k+1}^{(1)}\right) \times 2^k \right\rfloor}{2^k}$$

et on calcule $\hat{m} = m^{(1)} \times 2^{\alpha_2}$ et $\hat{e} = e - \alpha_1 - \alpha_2$. Si $\hat{m} \geq 1$ alors $\hat{m}$ est la mantisse du résultat tandis que $\hat{e}$ est son exposant. Si $\hat{m} < 1$, il faut multiplier $\hat{m}$ par $2^{2^{-k}}$ et soustraire $2^{-k}$ à la nouvelle valeur de l'exposant $\hat{e}$. Ceci donne la mantisse et l'exposant du résultat final.

Cette décomposition peut être répétée de nouveau si les tables avec $2^{\frac{k+1}{2}+1}$ entrées sont encore trop grandes.

Les tables sont beaucoup plus importantes pour l'addition et la soustraction que pour la multiplication et la division dans le système semi-logarithmique. On a des tables à $2^{k+1}$ valeurs contre $2^4$ valeurs. Les cellules de base du circuit sont beaucoup plus nombreuses et le contrôle est aussi sensiblement plus complexe que pour la multiplication et la division. On remarque toutefois que la complexité de l'opérateur d'addition/soustraction dans le système semi-logarithmique est du même ordre de grandeur que celle d'un multiplieur flottant.

Le choix d'une bonne valeur pour le paramètre $k$ est en fait assez simple. $k$ ne doit pas être trop grand car plus $k$ est grand plus les tables seront grandes (augmentation exponentielle). Par contre, les algorithmes ne deviennent très simple qu'à partir d'une certaine valeur de $k$. En effet,

dans l'ensemble les algorithmes des opérations de base deviennent particulièrement simples à partir d'une certaine valeur de $k$. Pour $k > \frac{n}{2} + 2$, les multiplications internes des nombres écrits en virgule fixe se réduisent à de simples additions dans tous les algorithmes. En pratique, la meilleure valeur de $k$ est probablement la plus petite qui permettent de réduire les multiplications internes en additions, c'est-à-dire $k = \frac{n}{2} + 3$.

Si nous voyons maintenant le système semi-logarithmique comme un système logarithmique utilisant à la fois les deux bases 2 et $e$, alors on peut écrire $x$ et $y$ sous la forme :

$$\begin{aligned} x &= s_x \times 2^{\tilde{e}_x} \times e^{\epsilon_x} \\ y &= s_y \times 2^{\tilde{e}_y} \times e^{\epsilon_y} \end{aligned}$$

avec $\epsilon_x, \epsilon_y < 2^{-k}$ et où $\tilde{e}_x$ et $\tilde{e}_y$ sont des multiples de $2^{-k}$. L'algorithme d'addition 2.3 (dans lequel on suppose $\tilde{e}_x \geq \tilde{e}_y$) utilise le calcul suivant :

$$\begin{aligned} x + y &= 2^{\tilde{e}_x} e^{\epsilon_x} \left(1 + e^{\epsilon_y - \epsilon_x} 2^{\tilde{e}_y - \tilde{e}_x}\right) \\ &\approx 2^{\tilde{e}_x} e^{\epsilon_x} \left(1 + (1 + \epsilon_y - \epsilon_x) \times 2^{\tilde{e}_y - \tilde{e}_x}\right). \end{aligned}$$

Dans la dernière approximation, l'utilisation d'un système à deux bases est essentiel. En effet, la base 2 permet d'effectuer la multiplication par $2^{\tilde{e}_y - \tilde{e}_x}$ en utilisant un décalage (pour la partie entière de la différence des exposants) et une table (pour la partie fractionnaire). Quant à l'utilisation de la base $e$, elle permet d'effectuer l'approximation $e^{\epsilon_y - \epsilon_x} \simeq 1 + \epsilon_y - \epsilon_x$.

### 2.3.4 Comparaisons

On désire comparer $x = s_x \times m_x \times 2^{e_x}$ et $y = s_y \times m_y \times 2^{e_y}$, deux nombres réels représentés dans la forme générale du système semi-logarithmique de paramètre $k$. On suppose que ces deux nombres sont positifs (si leurs signes sont différents la comparaison est triviale, si ils sont négatifs il suffit de modifier légèrement l'algorithme ci-dessous). On suppose que $e_x \geq e_y$ (dans le cas contraire il suffit d'échanger $x$ et $y$).

La comparaison se ramène donc à l'étude des trois cas suivants :

① Si $e_x - e_y > 2^{-k}$ les valeurs respectives des exposants permettent de conclure directement $x > y$.

② Si $e_x = e_y$, il faut tester les mantisses. Donc $x \geq y$ si et seulement si $m_x \geq m_y$.

③ Si $e_x - e_y = 2^{-k}$, il faut réécrire les mantisses avec le même exposant. La mantisse de $y$, $m_y$, est multipliée par la valeur précalculée $2^{-2^k}$ ce qui donne $m_y^*$. Maintenant, $x \geq y$ si et seulement si $m_x \geq m_y^*$. On remarque que le produit $m_y \times 2^{-2^k}$ se ramène à une addition si $k > n/2$.

### 2.3.5 Conversions

#### Conversion système flottant → système semi-logarithmique

Soit $x$ un nombre positif (la gestion des signes est triviale), représenté dans le système flottant binaire comme $M_x \times 2^{E_x}$ avec :

- $x = M_x \times 2^{E_x}$

- $1 \leq M_x < 2$
- $E_x$ est un entier

On désire convertir $x$ dans la forme générale du système semi-logarithmique de paramètre $k$. C'est-à-dire que l'on cherche $m_x$ et $e_x$ qui vérifient :

- $x = m_x \times 2^{e_x}$
- $1 \leq m_x < 1 + 2^{-k}$
- $e_x$ est un réel avec $k$ bits fractionnaires

On suppose que l'écriture binaire de $M_x$ est de la forme $1.M_1M_2 \ldots M_n$, on défini

$$M_x^* = 1.M_1M_2 \ldots M_{k+1}.$$

La conversion est similaire aux dernières étapes de l'algorithme d'addition :

① Les valeurs $\alpha$ et $2^\alpha$ définies ci-dessous sont lues dans une table à $(k+1)$ bits d'adresse (la même que celle de l'algorithme d'addition avec $M_1$, $M_2$, ..., $M_{k+1}$ comme bits d'adresse).

$$\alpha = \frac{\lfloor -\log_2(M_x^*) \times 2^k \rfloor}{2^k}$$

② On calcule $\hat{m} = M_x \times 2^\alpha$ et $\hat{e} = E_x - \alpha$. Si $\hat{m} \geq 1$ alors $\hat{m}$ est la mantisse du résultat et $\hat{e}$ son exposant. Si $\hat{m} < 1$, il faut multiplier $\hat{m}$ par $2^{2^{-k}}$ et soustraire $2^{-k}$ à $\hat{e}$, ceci donne la mantisse et l'exposant du résultat. Si $k > n/2 + 2$, la dernière multiplication se réduit à une addition.

**Conversion système semi-logarithmique → système flottant**

Soit $x$ un nombre réel représenté dans le système semi-logarithmique de paramètre $k$ par $m_x$ et $e_x$ tels que :

- $x = m_x \times 2^{e_x}$
- $1 \leq m_x < 1 + 2^{-k}$
- $e_x$ est un réel avec $k$ bits fractionnaires

On désire trouver la mantisse $M_x$ $(1 \leq M_x < 2)$ et l'exposant $E_x$ (entier) de $x$ dans le système flottant binaire. Ceci peut se faire en utilisant :

① Soit $e_f$ la partie fractionnaire de $e_x$, on tabule $2^{e_f}$ dans une table à $k$ bits d'adresse (ou avec une combinaison de lecture de tables et d'interpolations).

② On multiplie $m_x$ par $2^{e_f}$ (multiplication d'un nombre de $n-k$ bits par un nombre de $k$ bits), le résultat est noté $M^*$. Soit $E^* = \lfloor e_x \rfloor$.

③ $M^*$ est compris entre 1 et $2^{1-2^{-k}}\left(1 + 2^{-k}\right)$. Si $M^* < 2$ alors $M_x = M^*$ et $E_x = E^*$. Si $M^* \geq 2$ (ce cas est très rare car la borne supérieure de $M^*$ est très proche de 2), alors $M_x$ est obtenu en décalant $M^*$ d'une position à droite et en effectuant $E_x = E^* + 1$.

## 2.4 Précision statique du système semi-logarithmique

Dans cette section, nous allons évaluer l'*erreur relative de représentation maximale* (ERRMax) et l'*erreur relative de représentation moyenne* (ERRMoy) [Cod73] du système semi-logarithmique. Nous effectuerons les calculs dans le cas du mode d'arrondi vers zéro. Le calcul pour les autres modes d'arrondi est similaire. Dans le cas de l'évaluation de l'erreur moyenne, on suppose que la distribution des nombres est la *distribution logarithmique* de Hamming (cf [Ham70]), c'est-à-dire que l'on suppose la densité :

$$P(x) = \frac{1}{x \ln 2} \quad \text{où } 1 \leq x < 2$$

L'erreur relative $\left|\frac{x-\mathcal{Z}(x)}{x}\right|$ permet de bien appréhender la précision avec laquelle les nombres sont représentés (par rapport à leur valeur exacte). A titre d'exemple, nous avons représenté sur la Figure 2.2 l'erreur relative du système semi-logarithmique pour $n = 4$ et pour les différentes valeurs de $k$ (entre 0 et 4). Les courbes représentent l'erreur relative pour chacune des valeurs possibles de la mantisse (l'erreur relative est indépendante de l'exposant lors de multiplications par des multiples de $2^{2^{-k}}$). Sur la Figure 2.2 pour $k = 0$, on retrouve la distribution caractéristique de l'erreur relative du système flottant qui décroit pour les grandes valeurs de la mantisse. Pour $k = 4$, on retrouve la distribution de l'erreur relative du système logarithmique dont l'amplitude est constante.

### 2.4.1 Erreur relative de représentation maximale (ERRMax)

On suppose $x$ compris entre 1 et 2, on a alors :

$$\left|\frac{x-\mathcal{Z}(x)}{x}\right| = \frac{x - \frac{\left\lfloor 2^n \times \frac{x}{2^{\lfloor 2^k \log_2 x \rfloor / 2^k}} \right\rfloor}{2^n} \times 2^{\lfloor 2^k \log_2 x \rfloor / 2^k}}{x}$$

Soit $\Delta_c$ l'intervalle où $\lfloor 2^k \log_2 x \rfloor / 2^k$ est égal à $c$. Dans cet intervalle, $\left|\frac{x-\mathcal{Z}(x)}{x}\right|$ est égal à

$$\left(\frac{2^n x}{2^c} - \left\lfloor \frac{2^n x}{2^c} \right\rfloor\right) \times \frac{2^c}{2^n x} \tag{2.2}$$

On en déduit :

$$\text{ERRMax} = \max_{x\in[1,2]} \left|\frac{x-\mathcal{Z}(x)}{x}\right| \approx \max_{c=0,\ldots,1} \max_{x\in[2^c, 2^{c+1/2^k})} \left(\frac{2^n x}{2^c} - \left\lfloor \frac{2^n x}{2^c} \right\rfloor\right) \times \frac{2^c}{2^n x}$$

soit

$$\text{ERRMax} \approx \max_{c=0,1/2^k,\ldots,1} \max_{x\in[2^c, 2^{c+1/2^k})} \frac{2^c}{2^n x} = 2^{-n}.$$

Le système flottant et le système semi-logarithmique ($k < n$) ont la même valeur de l'erreur relative de représentation maximale. On remarque que le système logarithmique ($k \geq n$) à une plus petite ERRMax qui est $2^{-n}\ln(2)$ (ce que l'on retrouve bien sur la courbe pour $k = 4$ de la Figure 2.2).

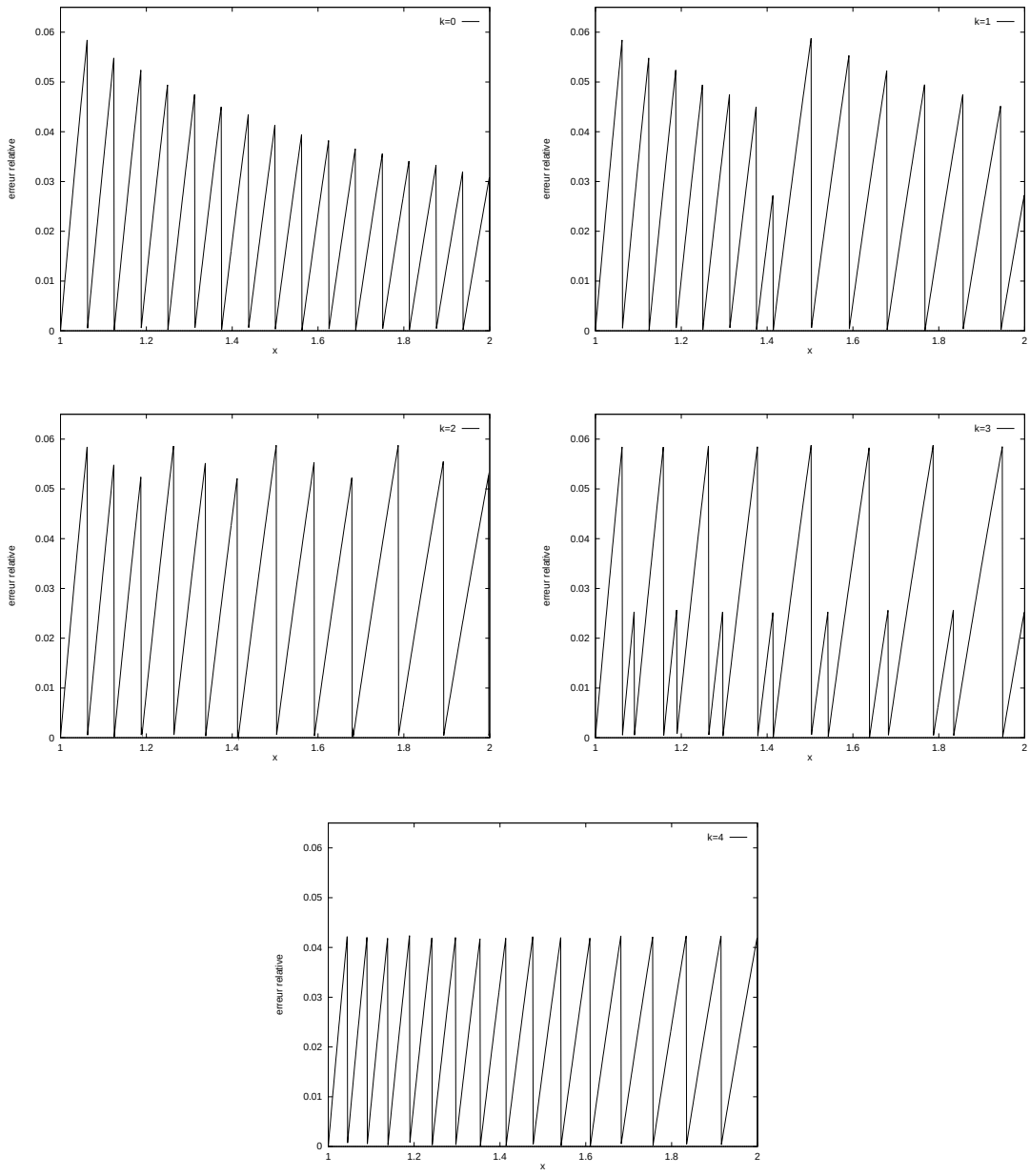



FIG. 2.2 – *Erreur relative du système semi-logarithmique avec $n = 4$ et $k$ variant entre 0 et 4. Les courbes pour $k = 0$ et $k = 4$ représentent respectivement le cas du système flottant et le cas du système logarithmique.*

### 2.4.2 Erreur relative de représentation moyenne (ERRMoy)

On désire évaluer

$$\text{ERRMoy} = \int_1^2 \frac{1}{x \ln 2} \times \left| \frac{x - \mathcal{Z}(x)}{x} \right| dx \tag{2.3}$$

Soit $\Delta_c$ l'intervalle où $\lfloor 2^k \log_2 x \rfloor / 2^k$ est égal à $c$. Dans cet intervalle, $\left| \frac{x-\mathcal{Z}(x)}{x} \right|$ est égal à :

$$\left( \frac{2^n x}{2^c} - \left\lfloor \frac{2^n x}{2^c} \right\rfloor \right) \times \frac{2^c}{2^n x} \tag{2.4}$$

On en déduit [1] :

$$\int_{\Delta_c} \frac{1}{x \ln 2} \times \left| \frac{x - \mathcal{Z}(x)}{x} \right| dx \approx \int_{\Delta_c} \frac{1}{x \ln 2} \times \frac{1}{2} \times \frac{2^c}{2^n x} dx$$

Comme $\Delta_c$ est égal à $\left[2^c, 2^{c+1/2^k}\right)$, on a alors :

$$\int_{\Delta_c} \frac{1}{x \ln 2} \times \left| \frac{x - \mathcal{Z}(x)}{x} \right| dx \approx \frac{2^{c-n-1}}{\ln 2} \left( \frac{1}{2^c} - \frac{1}{2^{c+1/2^k}} \right)$$

Les valeurs extrêmes de $c$ sont 0 (pour $x = 1$) et 1 (pour $x = 2$).
Ce qui donne (en posant $i = c \times 2^k$) :

$$\begin{aligned} \text{ERRMoy} &\approx \sum_{i=0}^{2^k} \frac{2^{i \times 2^{-k} - n - 1}}{\ln 2} \left( \frac{1}{2^{i \times 2^{-k}}} - \frac{1}{2^{(i+1) \times 2^{-k}}} \right) \\ &= \sum_{i=0}^{2^k} \frac{2^{i \times 2^{-k} - n - 1}}{\ln 2} \times \frac{2^{2^{-k}} - 1}{2^{(i+1) \times 2^{-k}}} \end{aligned}$$

Donc

$$\begin{aligned} \text{ERRMoy} &\approx \sum_{i=0}^{2^k} \frac{2^{-2^{-k} - n - 1}}{\ln 2} \times \left( 2^{2^{-k}} - 1 \right) \\ &\approx 2^k \times \frac{2^{-2^{-k} - n - 1}}{\ln 2} \times \left( 2^{2^{-k}} - 1 \right) \\ &\approx 2^{-n-1} \end{aligned}$$

en utilisant $2^{2^{-k}} \approx 1 + 2^{-k} \ln 2$. Cette approximation n'est pas valide pour les petites valeurs de $k$ (pour $k \leq 1$). Pour $k = 0$ (le cas du système flottant), l'erreur relative de représentation moyenne est égale à

$$\int_1^2 \frac{1}{x \ln 2} \times \frac{2^{-n}}{2x} dx = \frac{2^{-n-2}}{\ln 2}.$$

Dans le cas du système logarithmique de base 2, on a $\text{ERRMoy} = 2^{-n-1} \ln(2)$.

Le Tableau 2.4.2 résume les valeurs des erreurs relatives de représentation maximale et moyenne dans différents cas. Cette table indique que le système flottant et le système logarithmique sont un tout petit plus précis que le système semi-logarithmique. Le rapport entre l'erreur relative de représentation moyenne dans le système semi-logarithmique et celle dans le système logarithmique est $1/\ln(2) \approx 1.4$. Ce qui correspond à $\log_2(1/\ln(2)) \approx 1/2$ bit de précision.

1. On remplace $\left( \frac{2^n x}{2^c} - \left\lfloor \frac{2^n x}{2^c} \right\rfloor \right)$ par sa valeur moyenne. Cette approximation est valide si $2^k$ est petit devant $2^n$ (en pratique dès que $k$ est plus petit que $n - 3$).

| système | Arrondi vers 0 | | Arrondi au plus proche | |
|---|---|---|---|---|
| | ERRMax | ERRMoy | ERRMax | ERRMoy |
| flottant | $2^{-n}$ | $0.36 \times 2^{-n}$ | $2^{-n-1}$ | $0.36 \times 2^{-n-1}$ |
| semi-logarithmique ($k \geq 2$) | $2^{-n}$ | $0.5 \times 2^{-n}$ | $2^{-n-1}$ | $0.5 \times 2^{-n-1}$ |
| logarithmique | $0.69 \times 2^{-n}$ | $0.35 \times 2^{-n}$ | $0.69 \times 2^{-n-1}$ | $0.35 \times 2^{-n-1}$ |

TAB. 2.1 – ERRMax *et* ERRMoy *du système semi-logarithmique et des systèmes flottants et logarithmiques pour différentes valeurs de* $k$.

## 2.5 Exemples d'application

Les domaines d'utilisation potentiels du système semi-logarithmique sont les mêmes que ceux du système logarithmique : les algorithmes nécessitant plus de multiplications et de divisions que d'additions et de soustractions (certaines applications de traitement du signal et des images, de filtrage numérique, de calcul de certaines transformées, de contrôle numérique...).

Certains modèles de réseaux de neurones utilisant les ondelettes ont des rapports

$$\frac{\text{nombre de multiplications}}{\text{nombre d'additions}}$$

très élevés dans leur calculs. Les ondelettes conduisent à des implantations rapides en traitement du signal et en compression d'images [Dau88]. Il y a de nombreux résultats sur les décompositions en ondelettes dans la littérature, les principaux sont présentés dans [Mal89].

Nous avons commencé avec B. Girau (actuellement au Centre Suisse d'Electronique et de Microtechnique) une collaboration sur la réalisation d'un circuit de calcul spécifique pour les réseaux de neurones utilisant des décompositions en ondelettes. En effet, lors des calculs dans les algorithmes d'apprentissage d'un réseau d'ondelettes, on trouve des calculs du style :

$$O^{(q)}(x_1, \ldots, x_P) = \theta^{(q)} + \prod_{n=1}^{N} \left( w_n^{(q)} \prod_{p=1}^{P} h\left( d_p^{(n)} \left( x_p - t_p^{(n)} \right) \right) \right)$$

où

$$h(x) = -x\mathrm{e}^{-\frac{x^2}{2}}$$

Et encore des formules "particulièrement multiplicatives" comme :

$$\begin{aligned} \frac{\partial o^{(n)}}{\partial \alpha_{(k,l)}^{(n)}} &= \left[ \partial_1 H(Y^{(n)}), \ \ldots \ , \partial_P H(Y^{(n)}) \right] \prod_{i=1}^{k-1} \left( \prod_{j=i+1}^{P} r(\alpha_{(i,j)}^{(n)}, i, j) \right) \\ &\times \prod_{j=k+1}^{l-1} r(\alpha_{(k,j)}^{(n)}, k, j) \times r'(\alpha_{(k,l)}^{(n)}, k, l) \prod_{j=l+1}^{P} r(\alpha_{(k,j)}^{(n)}, k, j) \\ &\times \prod_{i=k+1}^{P} \left( \prod_{j=i+1}^{P} r(\alpha_{(i,j)}^{(n)}, i, j) \right) \end{aligned}$$

## 2.6 Conclusion

Nous avons conçu un nouveau système de représentation des nombres réels à mi chemin entre le système flottant et le système semi-logarithmique. Ce système reprend l'idée de base du système logarithmique : l'utilisation d'exposants réels, ce qui simplifie les opérations de multiplication et d'addition. Mais contrairement au système logarithmique, le système semi-logarithmique utilise une mantisse normalisée. Cette mantisse a un domaine de variation beaucoup plus petit que dans le système flottant ($[1, 1+\epsilon]$ contre $[1, 2[$). Cette mantisse permet de diminuer significativement la taille des tables utilisées dans les opérations d'addition et de soustraction car moins de chiffres fractionnaires sont nécessaires dans l'exposant.

Les algorithmes des opérations de base (addition, multiplication et division) deviennent particulièrement simples pour certaines valeurs du paramètre $k$. En effet, à partir de $k > \frac{n}{2} + 3$ les multiplications internes des nombres écrits en virgule fixe se réduisent à de simples additions.

La réduction de la taille des tables et la simplicité des algorithmes font du système semi-logarithmique un excellent candidat pour remplacer le système logarithmique dans les domaines d'application de ce dernier.

Il reste maintenant à réaliser effectivement les opérateurs correspondant pour obtenir les performances réelles de ce système.

# Arithmétique en-ligne sur FPGA

CHAPITRE

# 3 Arithmétique en-ligne sur FPGA

DE plus en plus de circuits intégrés numériques sont réalisés chaque année et dans de nombreuses applications la vitesse n'est plus la seule caractéristique importante d'une réalisation matérielle. En effet, avec l'essor actuel des systèmes embarqués et des appareils portables, la taille et la puissance consommée des circuits intégrés numériques sont devenus des paramètres de tout premier plan. On a vu ces derniers années la commercialisation de gammes de processeurs avec un ou plusieurs spécimens dédiés aux ordinateurs et appareils portables, comme les PPC603 dans la gamme des PowerPC de Motorola.

L'arithmétique série permet de réaliser des opérateurs de petite taille. De nombreux travaux et réalisations ont vu le jour dans le domaine des opérateurs de calcul série [Sip84, HC87, SD88, Dad89, HC90, IV94, IV95]. L'arithmétique série poids faibles en tête conduit à des solutions particulièrement efficaces pour l'addition et la multiplication. Mais l'arithmétique série poids faibles en tête ne permet pas d'effectuer certaines opérations comme les divisions et les racines carrées. Nous verrons que grâce aux représentations redondantes il est possible de réaliser toutes les opérations usuelles avec une arithmétique série dans laquelle les chiffres circulent poids forts en tête. Cette arithmétique appelée *arithmétique en-ligne* a été proposée par Ercegovac et Trivedi en 1977 [ET77]. Des applications ont été réalisées en arithmétique en-ligne, comme des filtres numériques [BEW89], des systèmes de calculs rapides pour l'algèbre linéaire [EL90, ET91]....

Les FPGA (*field programmable gate arrays*) sont des circuits intégrés dont les fonctionnalités logiques et les connexions internes sont programmables. Ces circuits sont souvent basés sur des topologies régulières de cellules logiques programmables et interconnectables entre elles selon des dispositifs, eux aussi, programmables. Les FPGA permettent de réaliser des systèmes dont les performances sont proches de celles d'un ASIC (*application specific integrated circuit*) tout en offrant un potentiel de programmation proche du logiciel. Certains fabricants de FPGA titrent leur publicité avec des slogans comme : "la puissance du *hard* et la programmation du *soft*".

Les FPGA sont particulièrement utiles dans le domaine du prototypage et des petites séries. Il est possible avec ces circuits de réaliser un système en implantant un certain algorithme, et de faire fonctionner cet algorithme dans une situation réelle d'exploitation. Avant ces circuits, il fallait simuler le comportement des circuits en utilisant des logiciels très gourmands en temps de calcul et pour lesquels la fidélité de la restitution des caractéristiques réelles du circuit testé était limitée.

Nous présentons dans ce chapitre les algorithmes en-ligne usuels (addition, multiplication, division, évaluation de polynômes...), et une bibliothèque écrite en VHDL qui intègre tous ces opérateurs et qui permet de concevoir des algorithmes complets de calcul numérique. Les travaux de ce chapitre ont été publiés dans [EMT95, GT96, Tis96, DMT96, TD97].

La première partie de ce chapitre présente les notions de base de l'arithmétique en-ligne ainsi que les algorithmes permettant de calculer les fonctions usuelles (addition, multiplication, division,

évaluation de polynômes). La deuxième partie présente les différents éléments de la bibliothèque réalisée et la méthodologie de test intégrée dans cette bibliothèque. Enfin, la dernière partie de ce chapitre présente deux exemples d'applications en cours de développement.

## 3.1 Arithmétique en-ligne

Le calcul série est un mode de calcul dans lequel les nombres circulent dans les opérateurs en *série*, chiffre après chiffre. Il y a deux possibilités pour le sens de circulation des chiffres : *poids faibles en tête* (LSDF *least significant digit first*) ou *poids forts en tête* (MSDF *most significant digit first*). Les additions et les multiplications s'effectuent naturellement poids faibles en tête à l'aide des algorithmes usuels. Par contre, il n'est pas possible d'effectuer des divisions, des comparaisons ou d'évaluer les fonctions élémentaires poids faibles en tête dans le système usuel de représentation des nombres.

Ceci est l'un des points critiques du calcul série. Il n'est pas envisageable de changer le sens de circulation des données entre deux opérateurs, tout le bénéfice introduit par le *pipeline* des opérateurs serait ainsi perdu. Il existe des systèmes de calcul série poids faibles en tête basés sur des représentations particulières des réels comme les nombres *p-adiques* (cf [Vui94]). Mais ces systèmes sont peu employés en pratique du fait de la complexité des conversions vers les systèmes plus conventionnels. Nous verrons en 3.1.1 que grâce aux représentations redondantes des nombres, il est possible de réaliser toutes les opérations usuelles poids forts en tête. L'*arithmétique en-ligne* est une arithmétique série dans laquelle les nombres circulent, chiffre après chiffre, poids forts en tête et qui utilise une notation redondante pour représenter les nombres ([Erc84]).

Les principales caractéristiques d'un opérateur en-ligne sont :

- Son *délai* $\delta$ qui est défini comme la différence de l'indice du chiffre entrant et celui du chiffre sortant. Cette valeur ne dépend que de l'algorithme utilisé dans l'opérateur et du système de représentation des nombres choisi. En architecture des ordinateurs cette valeur serait plutôt appelée la *latence* du pipeline associé à l'opérateur[1].

- Son *temps de cycle* ou *période*. C'est le plus grand temps de propagation du signal dans l'opérateur. Cette valeur borne la fréquence maximale d'utilisation de l'opérateur. Cette valeur correspond aux délais électriques dans l'opérateur.

- Sa *surface*. Nous exprimerons la surface d'un opérateur arithmétique en fonction du nombre de chiffres des opérandes. La surface effective d'un opérateur dans le circuit dépend d'un grand nombre de paramètres difficiles à prendre en compte au niveau algorithmique comme le placement des portes logiques et le routage. Dans la suite, la surface sera exprimée de façon asymptotique en fonction du nombre de chiffres des opérandes (comme $\mathcal{O}(n)$ par exemple). De plus, nous donnerons dans certains cas des arguments permettant d'estimer et de comparer la surface effective après les phases de placement et de routage entre différentes solutions.

Il existe d'autres caractéristiques utiles pour la conception de circuits comme la puissance consommée, le nombre d'entrées/sorties, les besoins en signaux de contrôle (horloges, reset...) des opérateurs. Mais ces paramètres sont difficilement estimables avec précision avant la synthèse pour que nous puissions les utiliser efficacement.

1. Dans la suite, nous préciserons *délai en-ligne* ou *délai électrique* si il y a ambiguïté.

Une autre propriété intéressante de l'arithmétique en-ligne est le fait que comme les calculs s'effectuent poids forts en tête, il est possible de contrôler la précision ([DMV94]). Par exemple, comme nous l'avons vu dans le chapitre sur l'arrondi exact des fonctions élémentaires, on peut calculer en-ligne le résultat de l'évaluation d'une fonction jusqu'à ce que le dilemme du fabriquant de tables ne se produise plus. Mais cette propriété est difficilement utilisable en matériel. Nous avons commencé l'étude d'une arithmétique dynamique (à précision variable) en-ligne en utilisant des chiffres codés sur des nombres flottants (chiffres en base $2^{50}$, cf [DMT97]).

La Figure 3.1 donne un exemple de calcul en-ligne et la Figure 3.2 illustre les valeurs du délai et de la période d'un opérateur en-ligne.

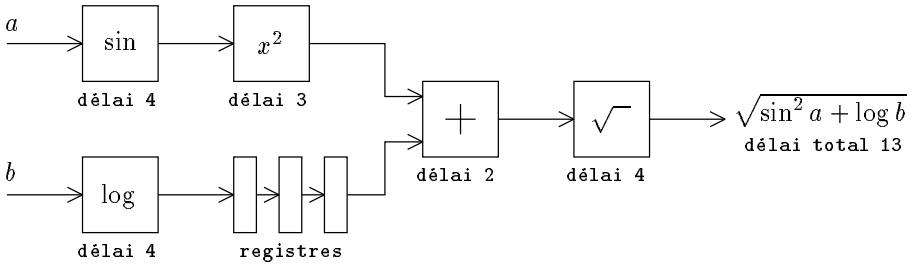



FIG. 3.1 – *Exemple de calcul en-ligne.*

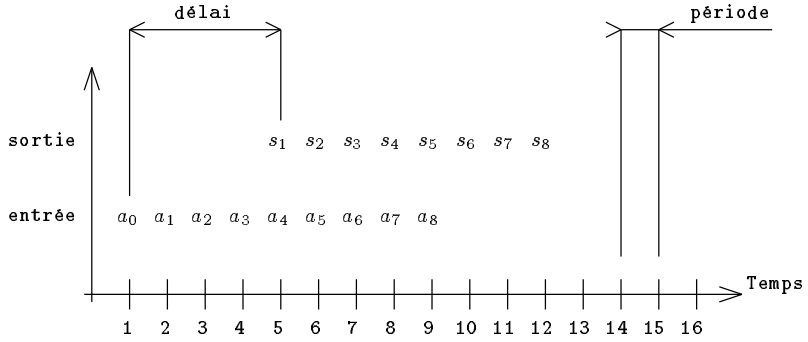



FIG. 3.2 – *Délai et période d'un opérateur en-ligne.*

Les principaux avantages de l'arithmétique en-ligne sont:

- Le parallélisme introduit par le pipeline au niveau du bit, qui permet de paralléliser des suites de calculs intrinsèquement séquentiels
- La petite taille des opérateurs (cf tableau 3.1)
- Le faible nombre de liens de communication
- Le fait que toutes les opérations usuelles soient calculables en-ligne (division, racine carré, sinus, cosinus, logarithme, exponentielle...)
- La maîtrise de la précision

| | parallèle | | série (LSDF) | en-ligne |
|---|---|---|---|---|
| | temps | surface | surface [2] | surface [2] |
| $\pm$ | $\mathcal{O}(1)$ | $\mathcal{O}(n)$ | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| $\times$ | $\mathcal{O}(\log_2 n)$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ |
| $\div$ | $\mathcal{O}(n)$ | $\mathcal{O}(n^2)$ | impossible | $\mathcal{O}(n)$ |
| $\sqrt{}$ | $\mathcal{O}(n)$ | $\mathcal{O}(n^2)$ | impossible | $\mathcal{O}(n)$ |
| $AX+B$ | $\mathcal{O}(\log_2 n)$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ |

TAB. 3.1 – *Complexité en surface et en temps des principaux opérateurs.*

Le Tableau 3.1 donne les tailles des opérateurs usuels dans les arithmétiques parallèle, série (poids faibles en tête) et en-ligne.

### 3.1.1 Représentation redondante des nombres

Dans [Win65], Winograd montre que dans le système usuel de représentation des nombres, il n'existe pas d'algorithme d'addition en temps constant ($\theta(1)$), c'est-à-dire sans propagation de retenue, si on emploie des portes logiques au nombre d'entrées borné.

Avizienis proposa dans [Avi61] des systèmes redondants de représentation des nombres dans lesquels il est possible d'effectuer une addition en temps constant avec des portes au nombre d'entrées borné. On peut trouver dans [Maz93] une présentation de certains systèmes de numération et en particulier celle des systèmes redondants, ainsi qu'une preuve que, sous certaines conditions, on ne peut pas effectuer d'addition en temps constant dans un système non redondant d'écriture des nombres.

Dans le système classique de numération, un nombre réel $x$ est représenté en base $\beta$ par une écriture de la forme :

$$x = \sum_{i=-\infty}^{+\infty} a_i \beta^i$$

où les $a_i$ sont les chiffres de $x$ pris dans l'ensemble $\{0, 1, 2, \ldots, \beta-1\}$, et où il existe $k$ tel que pour $\forall i \geq k$, $a_i = 0$.

En 1961, Avizienis a suggéré pour représenter les nombres d'utiliser les chiffres de l'ensemble $\{-a, -a+1, \ldots, 0, \ldots, a-1, a\}$ au lieu des chiffres de $\{0, 1, 2, \ldots, \beta-1\}$, où $a$ est un entier inférieur où égal à $\beta - 1$. Ces chiffres négatifs sont à l'origine de la désignation de ce système comme système de représentation des nombres à chiffres signés (*signed-digit number system*). On montre aisément que si $2a+1 \geq \beta$ alors tous les nombres sont représentables. Dans ce système, si $2a+1 > \beta$ certains nombres ont plusieurs écritures possibles. Par exemple, le nombre 2345 (dans le système usuel) peut s'écrire en base 10 avec l'ensemble de chiffres $\{-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5\}$ selon les codages 2345, 235(-5) ou 24(-5)(-5). On dit alors que le système est redondant.

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

2. En arithmétique série et en arithmétique en-ligne, le temps est évidemment en $\mathcal{O}(n)$.

gation de retenues. En particulier, l'algorithme 3.1 proposé par Avizienis dans [Avi61] effectue des additions de façon totalement parallèle.

**Algorithme 3.1 : (Avizienis 1961)**

*Entrées :* $x = 0.x_1x_2\ldots x_n \quad et \quad y = 0.y_1y_2\ldots y_n$

*Résultat :* $s = s_0.s_1s_2\ldots s_n$

*Ces nombres sont écrits en base* $\beta$ *avec des chiffres dans l'ensemble* $\{-a,\ldots,0,\ldots a\}$, *et où* $2a \geq \beta + 1$ *et* $a \leq \beta - 1$. *On pose* $w_0 = t_n = 0$

① `Pour` $i$ `variant de 0 à` $n-1$ `effectuer en parallèle :`

$$\begin{cases} t_{i-1} = \begin{cases} 1 & \text{si} \quad x_i + y_i > a - 1 \\ 0 & \text{si} \quad -a+1 \leq x_i + y_i \leq a-1 \\ -1 & \text{si} \quad x_i + y_i < -a+1 \end{cases} & (t_{i-1} \text{ est la retenue intermédiaire}) \\ w_i = x_i + y_i - \beta \times t_{i-1} & (w_i \text{ est la somme intermédiaire}) \end{cases}$$

② `Pour` $i$ `variant de 0 à` $n$ `effectuer en parallèle :`

$$s_i = w_i + t_i$$

Lorsque l'on regarde précisément l'algorithme 3.1 on s'aperçoit que la retenue $t_{i+1}$ ne dépend pas de la retenue $t_i$. Il n'y a donc pas de propagation des retenues. Les additions peuvent donc être effectuées en un temps qui ne dépend pas de la taille des opérandes. Le temps de l'addition dans les systèmes redondants est donc en $\mathcal{O}(1)$.

Dans la suite, afin d'éviter les confusions entre le signe des chiffres et l'opérateur de soustraction, nous noterons les chiffres négatifs avec une barre. Par exemple, le chiffre $-1$ est noté $\bar{1}$.

L'algorithme d'Avizienis présenté ci-dessus n'est pas valide en base 2. En effet, les conditions $2a \geq \beta + 1$ et $a \leq \beta - 1$ ne peuvent pas être satisfaites simultanément pour $\beta = 2$ (on a respectivement $a \geq \frac{3}{2}$ et $a \leq 1$). Il existe des algorithmes permettant de réaliser des additions en temps constant en base 2 en utilisant les notations *carry-save* (chiffres dans $\{0,1,2\}$) ou *borrow-save* (chiffres dans $\{-1,0,1\}$). La représentation *carry-save* est très souvent employée dans les multiplieurs. Nous avons choisi d'utiliser la représentation *borrow-save* du fait de la facilité qu'elle offre pour le traitement des nombres négatifs.

La représentation redondante *borrow-save* a été présentée par A. Guyot, Y. Herreros et J.M. Muller dans [GHM89]. Ils proposent de coder un chiffre $c$ de $\{-1,0,1\}$ sous la forme de deux bits $c^+$ (bit positif) et $c^-$ (bit négatif) tels que $c = c^+ - c^-$.

Exemple de notation *borrow-save* :

$$\begin{aligned} 0.123 &= 0.00011111011111001\ldots \\ &= (0,0).(0,0)(0,0)(0,0)(1,0)(1,0)(1,0)(1,0)(1,0)(0,0)(1,0)(1,0)(1,0)(1,0)(1,0)(0,0)(0,0)(1,0)\ldots \end{aligned}$$

$$= 0.0010000\bar{1}10000\bar{1}010\ldots$$
$$= (0,0).(0,0)(0,0)(1,0)(0,0)(0,0)(0,0)(0,0)(0,1)(1,0)(0,0)(0,0)(0,0)(0,0)(0,1)(0,0)(1,0)(0,0)\ldots$$

Le Tableau 3.2 donne le codage des chiffres dans la représentation *borrow-save*.

| chiffre | représentation $(c^+,c^-)$ |
|---|---|
| $-1$ | $(0,1)$ |
| 0 | $(0,0)$ ou $(1,1)$ |
| 1 | $(1,0)$ |

TAB. 3.2 – *Représentation des chiffres en* borrow-save.

### 3.1.2 Opérateurs en-ligne élémentaires

#### Cellule d'addition en *borrow-save*

Habituellement, on utilise en arithmétique des cellules *full-adder* (FA) pour réaliser des additionneurs. Une cellule FA engendre un bit de somme $s$ et un bit de retenue $r$ par addition (réduction) de 3 bits d'entrée ($a$, $b$ et $c$) avec :

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

Une réalisation possible au niveau logique de la cellule *full-adder* est :

$$\begin{cases} r &= \text{maj}(a,b,c) \\ &= (a \wedge b) \vee (a \wedge c) \vee (b \wedge c) \\ s &= a \oplus b \oplus c \end{cases}$$

Dans la notation *borrow-save*, certains bits peuvent avoir des poids négatifs. Une cellule FA ne peut convenir que pour additionner des bits de même signe. Nous allons donc utiliser une cellule dédiée, appelée PPM (Plus Plus Moins), inspirée de la cellule FA, pour le calcul de chiffres en *borrow-save* :

$$a^{\pm} + b^{\pm} - c^{\mp} = 2r^{\pm} - s^{\mp}$$

La cellule PPM peut effectuer la réduction des 3 bits d'entrée en 2 bits de sortie en utilisant la réalisation logique suivante :

$$\begin{cases} r^{\pm} &= \text{maj}(a^{\pm}, b^{\pm}, \overline{c^{\pm}}) \\ &= (a^{\pm} \wedge b^{\pm}) \vee (a^{\pm} \wedge \overline{c^{\pm}}) \vee (b^{\pm} \wedge \overline{c^{\pm}}) \\ s^{\mp} &= a^{\pm} \oplus b^{\pm} \oplus c^{\pm} \end{cases}$$

La Figure 3.3 présente une architecture logique possible pour les cellules FA et PPM. En pratique, les cellules PPM sont représentées comme des cellules FA mais avec pour chaque entrée et chaque sortie le signe correspondant (voir l'additionneur Figure 3.5).

#### Opposé d'un nombre en *borrow-save*

Du fait du codage particulier $c = c^+ - c^-$, l'opposé d'un nombre représenté en *borrow-save* s'obtient trivialement en permutant les deux bits de chaque chiffre. Seul le routage des fils arrivant dans un opérateur doit être modifié, aucune porte logique n'est nécessaire.

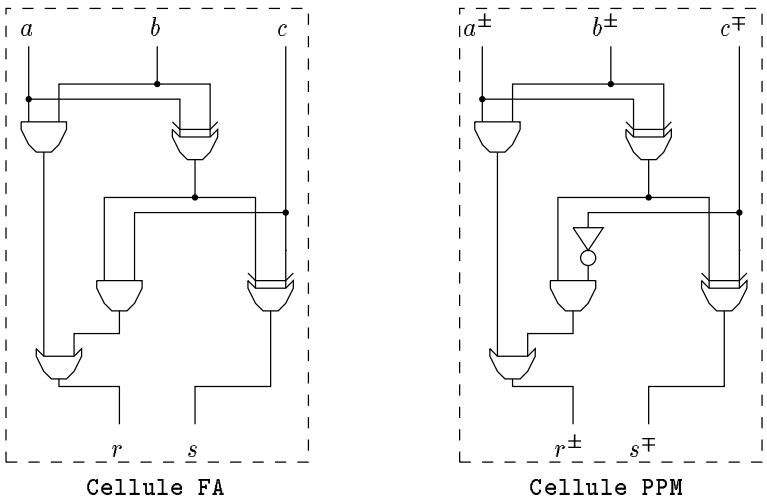



FIG. 3.3 – *Cellules FA et PPM.*

Il existe une deuxième solution pour obtenir l'opposé d'un nombre en *borrow-save*. Il suffit, en effet, de complémenter chacun des deux bits de chaque chiffre. Cette technique utilise deux inverseurs mais peut être utile en sortie d'un opérateur pour augmenter sa sortance (*fan-out*) avec un délai électrique plus petit qu'en utilisant un *buffer* de sortie.

### Produit de deux chiffres en *borrow-save*

Le produit de deux chiffres pris dans $\{-1, 0, 1\}$ est lui aussi dans $\{-1, 0, 1\}$. Ce point est l'un des avantages de la notation *borrow-save*. Dans la notation *carry-save*, le produit de deux chiffres de $\{0, 1, 2\}$ est dans $\{0, 1, 2, 4\}$, cela oblige à gérer des retenues supplémentaires ce qui complique le routage. Dans les bases supérieures à 2, le produit de deux chiffres devient une porte de base complexe et coûteuse en surface. En *borrow-save*, le produit des chiffres $a_i \times b_i$ se traduit en $(a_i^+, a_i^-) \times (b_i^+, b_i^-) = (p_i^+, p_i^-)$ avec

$$\begin{aligned} p_i^+ &= (a_i^+ \vee b_i^+) \wedge (\overline{a_i^+} \vee b_i^-) \\ p_i^- &= (a_i^- \vee b_i^+) \wedge (\overline{a_i^-} \vee b_i^-) \end{aligned}$$

La Figure 3.4 présente une architecture de multiplieur de deux chiffres *borrow-save* particulièrement bien adaptée à une implantation sur FPGA.

### Addition parallèle en *borrow-save*

Soient $a = 0.a_1a_2 \ldots a_n$ et $b = 0.b_1b_2 \ldots b_n$ deux nombres à additionner, et $s = s_0.s_1s_2 \ldots s_n$ leur somme. Ces nombres sont représentés en *borrow-save*. L'algorithme 3.2 permet d'effectuer l'addition parallèle $s = a + b$. La Figure 3.5 représente l'opérateur réalisant l'algorithme 3.2 pour $n = 4$. On remarque en particulier que le signal ne doit au plus traverser que deux cellules PPM, on retrouve

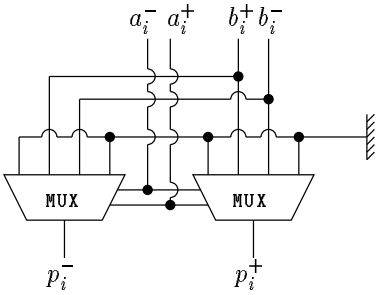



FIG. 3.4 – *Cellule réalisant le produit de 2 chiffres* borrow-save.

bien un temps de calcul en $\mathcal{O}(1)$.

**Algorithme 3.2 : Addition *borrow-save* parallèle**

① `Initialisation :` $c_n^+ = s_n^- = 0$

② `Pour` $i$ `variant de` $n$ `à` $1$ `effectuer en parallèle :`

$$a_i^+ + b_i^+ - a_i^- = 2c_{i-1}^+ - c_i^-$$

③ `Pour` $i$ `variant de` $n$ `à` $1$ `effectuer en parallèle :`

$$c_i^- + b_i^- - c_i^+ = 2s_{i-1}^- - s_i^+$$

*on pose :* $s_0^+ = c_0^+$

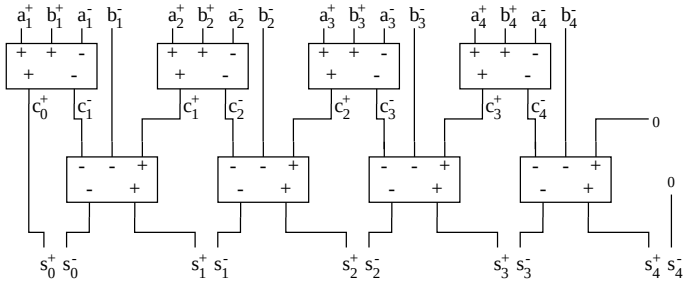



FIG. 3.5 – *Un additionneur* borrow-save *parallèle 4 bits.*

### 3.1.3 Addition en-ligne

Il est possible de traduire l'algorithme d'addition parallèle 3.2 décrit ci-dessus en une version *en-ligne* (cf [Baj93, BDKM94]). On obtient alors l'opérateur décrit Figure 3.6. Le délai de cet opérateur est 2, et il a été démontré dans [BDKM94] que cette valeur est optimale pour la représentation *borrow-save*. Il n'existe pas d'algorithme d'addition en-ligne en base 2 dont le délai soit inférieur à 2. Ceci n'est pas le cas pour des bases supérieures. Par exemple, une traduction poids forts en tête de l'algorithme d'Avizienis en tête conduit à un opérateur de délai 1 pour $\beta > 2$. Ce délai est optimal.

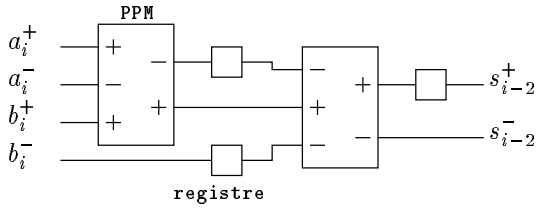



FIG. 3.6 – *Additionneur en-ligne de deux nombres (délai 2).*

L'arithmétique en-ligne présente une caractéristique intéressante pour l'addition de plusieurs nombres. Par exemple, la somme de trois nombres en utilisant l'associativité de l'addition conduit à un opérateur de délai 4. Ce délai n'est pas optimal. On trouve dans [BDKM94] un algorithme qui permet de réaliser un additionneur en-ligne de $k$ nombres avec un délai optimal égal à $\lceil \log_2 k \rceil + 1$ contre $2\lceil \log_2 k \rceil$ avec la solution arborescente associative. La technique consiste à réduire les bits (et non pas les chiffres) de même rang le plus tôt possible. La Figure 3.7 présente un additionneur de 3 nombres en-ligne de délai 3.

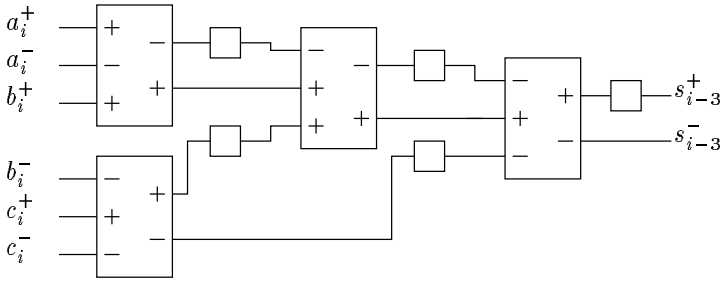


FIG. 3.7 – *Additionneur en-ligne de trois nombres (délai 3).*

On remarque que la taille de l'opérateur d'addition en-ligne ne dépend pas de la taille des opérandes. On trouve dans [Mul94] une caractérisation des fonctions qui sont calculables en-ligne par des opérateurs de taille bornée. Ces fonctions sont les fonctions affines par morceaux avec des coefficients rationnels (fonctions du type $f(x) = ax + b$ et $f(x, y) = ax + by + c$ avec $a, b, c$ dans $\mathbb{Q}$). Ce travail montre aussi que la multiplication, la division, la racine carrée et les fonctions élémentaires ne sont pas calculables en-ligne avec des opérateurs de taille bornée: par exemple, un multiplieur de nombres de $n$ chiffres sera de taille proportionnelle à $n$.

### 3.1.4 Multiplication en-ligne et opérateurs dérivés

On trouve dans [ET77] des algorithmes en-ligne pour effectuer des multiplications et des divisions. Ces algorithmes ont été adaptés au cas de la base 2 pour la représentation *borrow-save* [GHM89, BDKM94]. Nous présentons ici le multiplieur de délai 3 décrit dans [BDKM94]. Ce délai n'est pas optimal, il existe un algorithme de multiplication en-ligne en base 2 de délai 2 (aussi présenté dans [BDKM94]), mais la période de cet opérateur croît avec le nombre de chiffres des opérandes ce qui pose quelques problèmes d'implantation (on ne peut pas décrire un *macro-bloc* qui pourra être réutilisé facilement).

On calcule $p = x \times y$ où $x = 0.x_1x_2 \ldots x_n$, $y = 0.y_1y_2 \ldots y_n$ et $p = 0.p_1p_2 \ldots p_{2n}$ sont représentés

en *borrow-save*. On définit avec $k \leq n$ :

$$\begin{aligned} x^{(k)} &= 0.x_1x_2\ldots x_k \\ y^{(k)} &= 0.y_1y_2\ldots y_k \\ p^{(k)} &= x^{(k)} \times y^{(k)} \end{aligned}$$

On a

$$\left\{ \begin{array}{rcl} x^{(k+1)} &=& x^{(k)} + x_{k+1}2^{-k-1} \\ y^{(k+1)} &=& y^{(k)} + y_{k+1}2^{-k-1} \end{array} \right.$$

Donc

$$p^{(k+1)} = p^{(k)} + y_{k+1}2^{-k-1}x^{(k+1)} + x_{k+1}2^{-k-1}y^{(k)}$$

Si on définit $q^{(k)}$ et $r^{(k)}$ tels que :

$$\begin{aligned} q^{(0)} &= r^{(0)} = 0 \\ q^{(k+1)} &= q^{(k)} + y_{k+1}2^{-k-1}x^{(k+1)} \\ r^{(k+1)} &= r^{(k)} + x_{k+1}2^{-k-1}y^{(k)} \end{aligned}$$

Alors

$$p^{(k)} = q^{(k)} + r^{(k)}$$

L'opérateur correspondant est présenté Figure 3.8 et la partie additionneur final correspondante est présentée Figure 3.9.

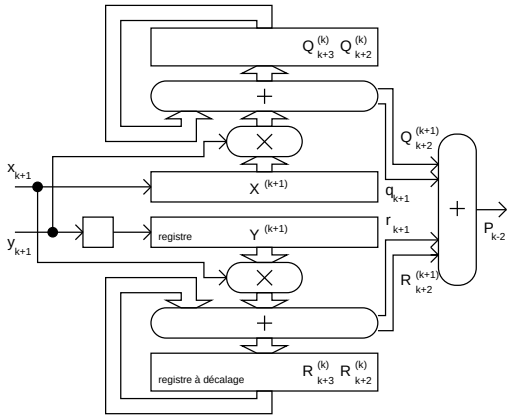



FIG. 3.8 – *Multiplieur en-ligne de délai 3.*

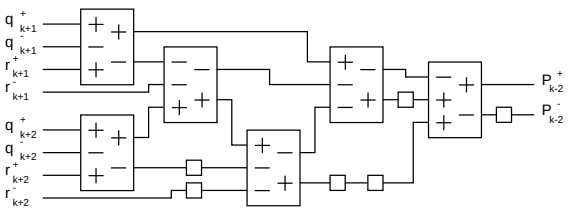


FIG. 3.9 – *Additionneur final du multiplieur en-ligne de délai 3.*

Ce multiplieur peut être modifié comme indiqué dans [BDKM94] pour calculer des carrés (délai 3), des multiplications par une constante (délai 2) et des binômes $ax + b$ (délai 3). Pour le calcul

des binômes, on utilise la technique de réduction des bits vue pour l'addition de plusieurs nombres dans l'additionneur final pour réduire le délai du binômieur. On arrive alors à un opérateur de délai de 3 contre 4 si on avait calculé $(ax) + b$.

Conformément à ce qui a été montré par Muller dans [Mul94], la taille de l'opérateur de multiplication en-ligne dépend de la taille des opérandes. Un résultat similaire a été obtenu par Brent et Kung dans [BK81]. Ils ont, en effet, montré qu'il existe un compromis entre la surface et le temps de calcul d'un multiplieur binaire :

$$\frac{A(n)}{A_0}\left(\frac{T(n)}{T_0}\right)^{2\alpha} \geq n^{1+\alpha}, \forall \alpha \in [0,1]$$

où $A(n)$ et $T(n)$ sont respectivement la surface et le temps nécessaire pour effectuer une multiplication de deux nombres binaires de $n$ chiffres et où $A_0$ et $T_0$ sont des constantes qui dépendent de la technologie.

### 3.1.5 Division et racine carrée en-ligne

De nombreux algorithmes et implantations d'opérateurs en-ligne de division ont été proposés dans la littérature [ET77, Irw78, IO79, EL85, IO87, ET87, LS87, ET89, LE92, LE93c, MRM93, LE93b]. La réalisation d'un opérateur de division en-ligne dépend des ordres de grandeur respectifs du diviseur et du dividende. En effet, certaines contraintes de "normalisation" sur les entrées conduisent à des solutions plus ou moins complexes. De plus, la division est un opérateur gourmand en surface et il est possible d'adapter différentes implantations d'un même algorithme avec des délais et des surfaces variables. Par exemple, il est possible d'obtenir des opérateurs ayant des petits délais en utilisant des techniques de sélection des chiffres du quotient assez complexes et donc volumineuses. Il est donc possible de concevoir différents opérateurs avec des caractéristiques variables selon le compromis délai/surface visé. Pour le moment nous n'utilisons que le diviseur en-ligne présenté dans [MRM93] dont l'algorithme (3.3) est présenté ci-dessous. Le résultat de l'algorithme est $q = a/b$ avec $a < b$ et $\frac{1}{2} \leq b \leq 1$.

**Algorithme 3.3 : Division en-ligne (délai 5)**

`Initialisation :` $a[0] = 0.a_1a_2a_3a_4$, $b[0] = 0.b_1b_2b_3b_4$, $w[0] = a[0]$ `et` $q[0] = 0$

`Pour` $i$ `variant de 1 à` $n$ `effectuer :`

$q_i = \texttt{select}(2w[j-1])$

$w[i] = 2w[i-1] + a_{i+4}2^{-4} + q[i-1]b_{i+4}2^{-4} - q_i b[i-1]$

$b[i] = b[i-1] + b_{j+4}2^{-j-4}$

$q[i] = q[i-1] + q_i 2^{-j}$

*où* $\texttt{select}(x) = \{1 \ si \ x \geq \frac{1}{4}, \quad -1 \ si \ x \leq -\frac{1}{4}, \quad 0 \ sinon\}$

Du fait de la similarité entre les algorithmes de division et de racine carrée, de nombreux opérateurs en-ligne effectuant des racines carrées ont été proposés à partir de diviseurs en-ligne [Erc78, OE82, LE93a, EL94]. Les mêmes problèmes de choix pour le compromis délai/surface se posent.

### 3.1.6 Evaluation de polynômes en-ligne

L'évaluation rapide de polynômes est un enjeu important tant pour le calcul scientifique en général que dans les applications spécialisées. En effet, les fonctions élémentaires (sin, cos, log, exp, tan...) sont de plus en plus intégrées au niveau matériel. On peut citer par exemple les techniques de compression d'images utilisant la transformée en cosinus discret, l'utilisation des transformées en ondelettes en traitement du signal....

Comme nous l'avons vu dans le chapitre sur l'arrondi exact des fonctions élémentaires (page 17), il existe différents types d'algorithmes permettant l'évaluation de ces fonctions (cf [Mul97]). Cette évaluation est souvent basée sur des approximations par des polynômes. En 1885, Weirstrass montra qu'il est possible d'approcher sur un compact une fonction continue d'aussi près que l'on veut par un polynôme. Un grand nombre d'architectures ont été proposées pour évaluer des polynômes ([DM88, MP90, MMY93]). En particulier, le schéma de Horner conduit à des réalisations très régulières et modulaires (cf Figure 3.10). La régularité de l'architecture (particulièrement importante pour la réalisation de circuits intégrés ou FPGA) est la conséquence directe de la régularité du calcul :

$$P(x) = \sum_{i=0}^{d} a_i x^i = a_0 + x(a_1 + x(a_2 + x(\ldots(a_{d-1} + a_d x)\ldots)))$$

où l'on utilise $d$ binômieurs $(ax + b)$ en série pour l'évaluation d'un polynôme de degré $d$.

La modularité provient du fait qu'il suffit de changer les constantes $a_i$ pour changer la fonction à évaluer. En général, l'approximation sur un intervalle avec une certaine précision de deux fonctions distinctes nécessite deux polynômes de degrés différents. En pratique, il suffit de prévoir un nombre de binômieurs correspondant au degré le plus élevé et de court-circuiter les étages inutiles suivant les fonctions.

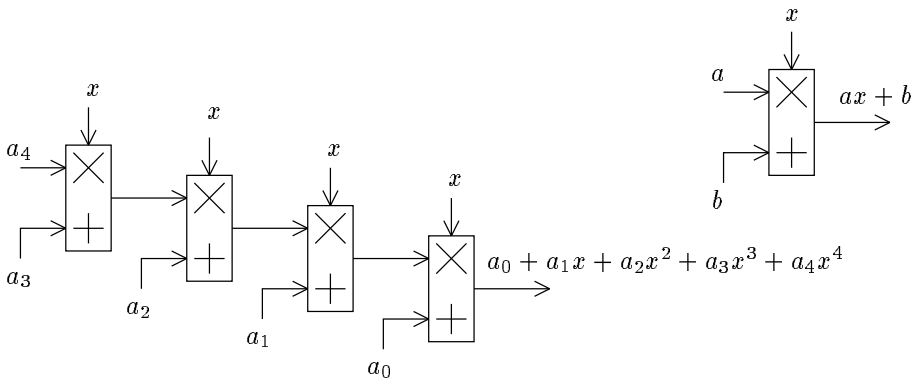



FIG. 3.10 – *Evaluation d'un polynôme de degré 4 avec le schéma de Horner.*

On trouve dans [Baj93, CDHM91] des études de polynômieurs en-ligne basés sur le schéma de Horner. En particulier, dans [Baj93] on trouve une étude de l'implantation d'un polynômieur en-ligne présentant un très faible délai. En utilisant directement le schéma de Horner l'évaluation en-ligne d'un polynôme de degré $d$ conduit à un opérateur de délai $3 \times d$ (le délai d'un binômieur est 3). Dans sa thèse [Baj93], Bajard utilise le fait que pour certaines fonctions comme l'exponentielle présentant un développement de Taylor dont les coefficients décroissent rapidement, il est possible d'utiliser une certaine "renormalisation" des coefficients pour diminuer le délai global. En effet, si

un opérateur calcule la fonction $f(x)$ avec un délai $\delta$, alors ce même opérateur calcule la fonction $2^{-\delta}f(x)$ avec un délai nul. Mais cette méthode à ses limites, car toutes les fonctions n'admettent pas des développements de Taylor permettant de diminuer significativement le délai global.

Un autre problème se pose avec la période d'un tel opérateur. Il n'est pas envisageable de réaliser un opérateur dont la période serait équivalente à la traversée de plusieurs binômieurs. En pratique, un registre doit être inséré entre deux binômieurs consécutifs. Le délai d'un opérateur en-ligne utilisant le schéma de Horner passe alors à $4 \times d$. Différentes autres approches ont été explorées pour diminuer ce délai.

La méthode *divide-and-conquer* (cf Figure 3.11) utilise un arbre de binômieurs [DM88]. Cette méthode permet d'obtenir un délai logarithmique mais elle nécessite des quadrateurs (élévation au carré) et conduit à des circuits jusqu'à deux fois plus grand qu'avec le schéma de Horner. Elle est principalement adaptée à des applications "haute vitesse".

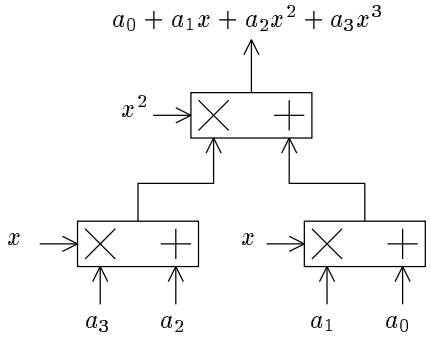



FIG. 3.11 – *Architecture* divide-and-conquer *d'évaluation de polynômes.*

La *E-méthode* proposée par Ercegovac [Erc77] est une méthode, inspirée du schéma de Horner, qui permet d'évaluer un polynôme de degré $d$ avec un délai égal à $d$. Dans [Tis94, EMT95] nous avons étudié une version en-ligne de la *E-méthode*. Nous avons réalisé une implantation de cet algorithme sur une carte DEC-PeRLe1. Cette carte est composée d'une matrice de 16 FPGA XC3090 de Xilinx, de 7 autres XC3090 autour de la matrice pour assurer le contrôle et les communications avec la machine hôte à travers le bus Turbo Channel de DEC. Nous avons alors obtenu un polynômieur fonctionnant à 33 MHz capable d'évaluer des polynômes de degré 16 sur 74 chiffres binaires. Le point le plus important de cette réalisation est qu'il est possible de changer les coefficients du polynôme sans interrompre le pipeline en augmentant simplement le délai (au sens latence) du premier résultat correspondant au nouveau polynôme. Le gain en délai obtenu par rapport au schéma de Horner est dû à l'utilisation d'opérateurs plus complexes qu'un binômieur et à l'utilisation d'un système de sélection des chiffres du résultat inspiré des algorithmes de division. La *E-méthode* conduit en général à des circuits plus grands que leurs équivalents utilisant le schéma de Horner.

La deuxième solution que nous avons étudiée est l'utilisation d'une architecture qui combine un polynômieur basé sur le schéma de Horner et des tables. L'idée proposée par Kla dans [Kla93] repose sur le découpage de l'intervalle d'évaluation en plusieurs petits intervalles sur lesquels la fonction peut être approchée par des polynômes de petits degrés. En pratique, les premiers chiffres de l'opérande $x$ qui arrivent dans l'opérateur servent à choisir à quel intervalle appartient $x$. Ces premiers chiffres indexent une table qui va donner aux différents binômieurs les coefficients correspondant au bon polynôme. Le principe de base de cette technique est représenté Figure 3.12.

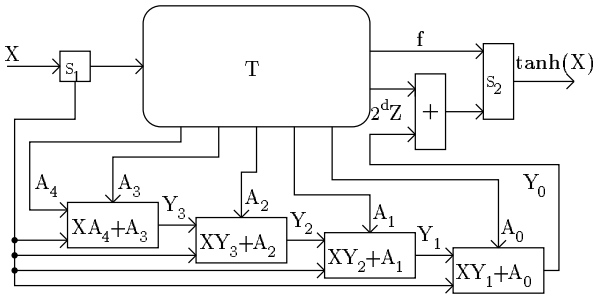



FIG. 3.12 – *Evaluation de polynômes combinant une table et un polynômieur basé sur le schéma de Horner.*

Nous avons utilisé cette technique pour évaluer la fonction tangente hyperbolique dans une application de réseaux de neurones (cf 3.3.1 et [GT96]). L'opérateur réalisé permet d'évaluer la fonction tanh dans l'intervalle $[-4, 4]$ sur des nombres en virgule fixe de 24 bits (8 de partie entière et 16 de partie fractionnaire). Le découpage retenu coupe l'intervalle initial en 16 morceaux et utilise un polynômieur de degré 5. La surface globale de l'opérateur obtenu est d'environ 600 blocs logiques sur un FPGA XC4020 de Xilinx.

Nous travaillons actuellement sur un système permettant de réaliser de façon quasi-automatique le découpage en sous-intervalles en essayant d'optimiser le nombre de chiffres des coefficients pour diminuer la taille des binômieurs.

### 3.1.7 Conversions

La conversion du système binaire usuel vers la représentation *borrow-save* est immédiate. En effet, il suffit de n'utiliser que le bit positif de chaque chiffre. La conversion de la représentation *borrow-save* vers la représentation binaire usuelle se fait en effectuant une "vraie" addition avec propagation de retenue. Si $a$ est écrit en *borrow-save* ($a = \sum_{i=1}^{n} a_i 2^{-i}$ avec $a_i = a_i^+ - a_i^-$) alors la conversion de $a$ en écriture binaire usuelle se fait en effectuant la soustraction

$$\sum_{i=1}^{n} a_i^+ 2^{-i} - \sum_{i=1}^{n} a_i^- 2^{-i} \tag{3.1}$$

Cette soustraction nécessite d'attendre de connaître tous les chiffres avant de commencer la conversion. Le délai introduit par la conversion est donc relativement important. Pour éviter ce problème, Ercegovac et Lang ont proposé une méthode de conversion "au vol" en base 2 (cf [EL87]).

Selon l'application cible, il faut être aussi capable de communiquer avec des dispositifs de nature analogique. La conversion d'un signal analogique vers la représentation *borrow-save* se fait en utilisant des convertisseurs analogiques/numériques (A/D) conventionnels car la conversion de l'écriture binaire usuelle vers la représentation *borrow-save* est immédiate et d'un coût quasi-nul (routage). La conversion d'un nombre écrit en *borrow-save* vers "son" équivalent analogique (conversion D/A) peut se faire sans devoir passer par une écriture binaire usuelle intermédiaire. En effet, on a vu dans l'équation 3.1 que l'écriture *borrow-save* peut se voir comme la soustraction de deux quantités. Il suffit d'utiliser deux convertisseurs numérique/analogique pour obtenir un convertisseur rapide. Le premier avec une référence de tension positive et le second avec une référence de tension égale à l'opposé de celle du premier convertisseur. La soustraction est alors effectuée au niveau analogique. Les convertisseurs numérique/analogique sont relativement petits devant un additionneur (réseaux

R-2R), le fait de doubler le convertisseur D/A n'est donc pas très problématique. Les deux tensions de référence opposées doivent être réalisées avec un miroir de tension, dispositif très bien maîtrisé actuellement.

### 3.1.8 Normalisation

Il est parfois nécessaire de renormaliser un résultat avant d'effectuer la suite des calculs. En effet, la notation redondante résultant de certaines opérations peut introduire des codages qui s'écrivent sur plus de chiffres que ce qui est strictement nécessaire. Par exemple, on peut avoir des écritures parfaitement équivalentes comme $10000\bar{1}$ et 011111. La technique la plus simple est d'utiliser l'algorithme de normalisation au vol proposé par Ercegovac et Land dans [EL87]. Des versions spécifiques de cet algorithme peuvent être adaptées pour certaines plages de valeur des opérandes.

Le principal problème avec la normalisation est de déterminer quand il faut renormaliser un résultat intermédiaire. L'opérateur de normalisation occupe une surface sensiblement supérieure à celle d'un additionneur en-ligne et introduit un délai (électrique) supplémentaire, il n'est donc pas question de renormaliser tous les résultats intermédiaires. Pour le moment nous n'avons pas de méthode optimale qui permette de placer au mieux (ou presque) les opérateurs de renormalisation. Une heuristique qui semble donner de bons résultats est de renormaliser systématiquement les résultats intermédiaires dans les boucles. Ce point nécessite encore du travail pour effectivement maîtriser le délai et la surface des algorithmes en-ligne.

## 3.2 Bibliothèque réalisée

Nous avons réalisé une bibliothèque VHDL d'opérateurs en-ligne. Le but de cette bibliothèque est de fournir une plateforme de développement pour des algorithmes en-ligne sur FPGA. Cette bibliothèque offre :

- Un jeu d'opérateurs complet
- Une technique de contrôle simple et robuste
- Un système de validation rapide intégré

Nous allons reprendre chacun de ces points plus en détails. Mais juste avant, nous allons présenter les technologies cibles de notre bibliothèque.

### 3.2.1 Les FPGA cibles

Notre bibliothèque, même si elle peut être synthétisée sur d'autres technologies que les FPGA, a été conçue en priorité pour deux types de FPGA :

- Les FPGA Actel et plus particulièrement les familles ACT1 et ACT2. Ces FPGA sont basé sur une technologie *anti-fusible* et ne sont donc programmables qu'une seule fois. Mais ces FPGA sont les seuls qui possèdent les califications nécessaires pour les applications aérospatiales (cf section 3.3.2).
- Les FPGA Xilinx (familles 3000 et 4000). Nous avons ciblé ces FPGA du fait de leur reprogrammabilité (technologie SRAM) et de leur granularité assez petite. Les blocs logiques de ces FPGA permettent de réaliser la plupart des opérateurs élémentaires (PPM, produit

de deux chiffres,...) sur un seul bloc logique. De plus, le fait qu'ils intègrent deux registres de 1 bit dans chaque bloc logique conduit à des circuits particulièrement compacts pour des opérateurs *borrow-save*.

Le choix de la granularité des cellules élémentaires dans les opérateurs fait que l'on peut synthétiser la plupart de nos opérateurs sur d'autres cibles technologiques (FPGA Altera...) mais les circuits obtenus risquent d'être plus volumineux et lents.

### 3.2.2 Opérateurs réalisés

A l'heure actuelle, notre bibliothèque VHDL est composée de 3 parties :

- Des *packages*;
- Des *cellules élémentaires*;
- Des *opérateurs complexes*.

#### Les packages

Le package `borrow_save` définit les types de données nécessaires pour manipuler des nombres entiers et réels en virgule fixe dans la représentation *borrow-save*. Il définit les procédures de conversions vers les systèmes plus conventionnels (écriture binaire usuelle, écriture en complément à deux et représentation *carry-save*). Ce package regroupe aussi un ensemble complet de routines pour les interactions avec l'utilisateur (entrées/sorties spécifiques).

Le package `bit_vector_conv` transforme les types de données comme les types *borrow-save* en vecteurs de bits pour pouvoir interfacer nos opérateurs avec les bibliothèques standards (qui utilisent des vecteurs de bits).

Et enfin, le package `math` rassemble différentes routines qui réalisent de façon logicielle les algorithmes des opérateurs en-ligne et différentes routines utilitaires pour la validation.

#### Les cellules élémentaires

Le Tableau 3.3 présente l'ensemble des cellules élémentaires de la bibliothèque. La taille est donnée à titre indicatif et correspond à une implantation sur des FPGA Actel de la famille ACT 1200 (Actel 1240A et 1280A).

#### Les opérateurs complexes

Les opérateurs en-ligne complexes opérationnels sont présentés dans le Tableau 3.4. Ici encore, la taille est donnée à titre indicatif et est le résultat d'une implantation sur des FPGA de la famille Actel 1200.

Chaque opérateur en-ligne est décliné en deux versions : avec et sans bloc de contrôle. Le bloc de contrôle est présenté dans la section 3.2.3. Il permet de réaliser un circuit complet avec un contrôle distribué particulièrement simple à concevoir.

La taille des opérateurs obtenus est variable en fonction du placement et du routage réalisés. Les additionneurs en-ligne de 2 et 3 nombres sont particulièrement optimisés. Dans le cas des multiplieurs par une constante, le nombre et le type des cellules élémentaires utilisées dépend de la valeur de la constante. Il en est de même pour les binômieurs.

| cellule | taille | fonction |
|---|---|---|
| ppm | 2 | cellule `PPM` |
| fa | 2 | cellule `FA` |
| flip_flop | 1 | registre 1 bit |
| flip_flop_wr | 1 | registre 1 bit sans `reset` |
| latch | 1 | un latch |
| mux2 | 1 | multiplexeur à 2 entrées |
| mux4 | 1 | multiplexeur à 4 entrées |
| mul_bs_digit | 2 | produit de 2 chiffres *borrow-save* |
| control_block | 3+délai | bloc de contrôle (cf 3.2.3) |
| fifo | $n$ | `FIFO` 1 bit de taille $n$ |
| bs_fifo | $2 \times n$ | `FIFO` 1 nombre de taille $n$ |
| bs_fifo_ctrl | $3 \times n$ | `FIFO` 1 nombre de taille avec contrôle $n$ |
| opposite | 0 | opposé d'un nombre |

TAB. 3.3 – *Cellules élémentaires de la bibliothèque.*

| cellule | taille | fonction |
|---|---|---|
| add_bs_int | $4 \times n$ | additionneur parallèle entier |
| add_bs_real | $4 \times n$ | additionneur parallèle réel (virgule fixe) |
| add_ol | 7 | additionneur en-ligne de 2 nombres |
| add_ol_ctrl | 12 | additionneur en-ligne de 2 nombres avec bloc de contrôle |
| add_ol_3 | 13 | additionneur en-ligne de 3 nombres |
| add_ol_3_ctrl | 19 | additionneur en-ligne de 3 nombres avec bloc de contrôle |
| add_ol_k | ? | additionneur en-ligne de k nombres |
| add_ol_k_ctrl | ? | additionneur en-ligne de k nombres avec bloc de contrôle |
| mul_cst_ol | ? | multiplieur en-ligne par une constante |
| mul_cst_ol_ctrl | ? | multiplieur en-ligne par une constante avec bloc de contrôle |
| mul_ol | $5n + 6$ | multiplieur en-ligne de 2 nombres |
| mul_ol_ctrl | $5n + 13$ | multiplieur en-ligne de 2 nombres avec bloc de contrôle |
| bin_ol | ? | binômieur en-ligne de 2 nombres |
| bin_ol_ctrl | ? | binômieur en-ligne de 2 nombres avec bloc de contrôle |
| sqr_ol | $3n + 2$ | quadrateur en-ligne ($x^2$) |
| sqr_ol_ctrl | $3n + 9$ | quadrateur en-ligne avec bloc de contrôle |
| div_ol | $6n + 20$ | diviseur en-ligne de 2 nombres |
| div_ol_ctrl | $6n + 28$ | diviseur en-ligne de 2 nombres avec bloc de contrôle |

TAB. 3.4 – *Opérateurs en-ligne de la bibliothèque.*

L'opérateur de division en-ligne introduit d'importants délais électriques et conduit donc à une période beaucoup plus élevée que celle des autres opérateurs. Nous travaillons actuellement une implantation de cet opérateur en utilisant des techniques de *retiming* pour diminuer le temps de cycle sans augmenter le délai. Nous pensons utiliser une modification de cet opérateur de division pour réaliser un opérateur évaluant des racines carrées dans l'avenir.

A titre d'exemple, nous présentons ci-dessous le code source VHDL qui correspond à l'additionneur en-ligne de 2 nombres et sur la Figure 3.13 le schéma de la cellule correspondante.

```
use work.borrow_save.all;

entity add_ol is
        port (a,b : in bs_digit ; clk : in bit ; reset : in bit ; s : out bs_digit);
end add_ol;

architecture struct of add_ol is

        component ppm
                port(x, y, z : in bit ; c, s : out bit);
        end component;
        for all : ppm use entity work.ppm(behav);

        component flip_flop
                port(input, clk, reset: in bit ; output : out bit);
        end component;
        for all : flip_flop use entity work.flip_flop(behav);

        signal bi1m, cim, ci1m, ci1p, si1p : bit;

begin

        ppm1 : ppm port map(a.p,b.p,a.m,ci1p,cim);
        ppm2 : ppm port map(bi1m,ci1m,ci1p,s.m,si1p);

        ff1 : flip_flop port map(b.m,clk,reset,bi1m);
        ff2 : flip_flop port map(cim,clk,reset,ci1m);
        ff3 : flip_flop port map(si1p,clk,reset,s.p);

end struct;
```

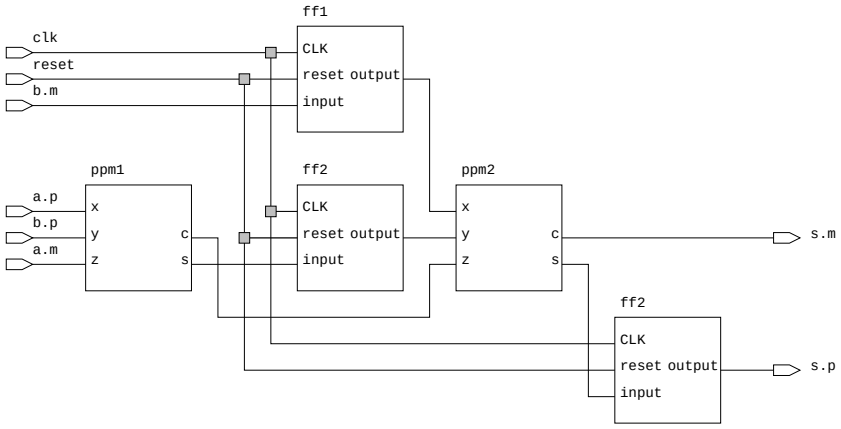



FIG. 3.13 – *Cellule* `add_on_line`.

### 3.2.3 Contrôle des opérateurs en-ligne

Une fois les différents opérateurs décrits, la conception d'un circuit est encore loin d'être achevée. En effet, il faut cadencer tous ces opérateurs et synchroniser les échanges de données dans le circuit. La phase de synchronisation des opérateurs est particulièrement délicate dans le cas d'opérateurs en-ligne car il faut s'assurer que les chiffres arrivent bien au bon moment dans chacun des opérateurs en tenant compte des délais en-ligne des différents opérateurs.

Habituellement, on utilise un contrôleur global qui supervise le flot des signaux entre les différents opérateurs. On parle alors de contrôle centralisé. Le contrôleur est basé sur un automate fini et sur différents compteurs. La conception de contrôleurs est aujourd'hui bien maîtrisée. Toutefois, cette technique n'est pas spécialement bien adaptée à des implantations sur FPGA. En effet, les signaux de contrôle doivent être routés depuis le contrôleur vers chaque opérateur du circuit. Ces signaux de contrôle utilisent alors une part importante des dispositifs de communication du FPGA. De plus, la diversité des signaux de contrôle (délais variables suivant les opérateurs) fait que le routage de ces signaux n'est pas très régulier ce qui est particulièrement problématique dans les FPGA. En effet, la plupart des FPGA n'offrent qu'un nombre très limité de dispositifs de communications arbitraires à grande distance.

Nous avons choisi une autre solution pour effectuer le contrôle des opérateurs en-ligne : le contrôle distribué. En effet, il est possible d'obtenir un contrôle totalement distribué en n'utilisant que des communications locales entre les opérateurs. Nous avons choisi de rajouter à chaque nombre un signal supplémentaire qui code la validité des chiffres qui arrivent. Ceci n'est pas très coûteux en arithmétique en-ligne car le nombre de signaux de communication entre les opérateurs est faible (2 bits par nombre en *borrow-save* seulement). Le signal de contrôle est à "1" si les bits qui arrivent sont significatifs et il est à "0" entre deux nombres consécutifs. En pratique, le signal de contrôle doit être en avance d'un cycle sur les chiffres pour pouvoir effectuer l'initialisation des opérateurs. Un cycle d'initialisation complet est nécessaire pour les opérateurs complexes. Par exemple, dans un multiplieur, il y a deux registres de $2n$ bits à remettre à zéro avant chaque nouveau calcul ($n$ est la taille des nombres).

Le signal de contrôle va donc ce propager dans le circuit de la même façon que les chiffres, avec juste un cycle d'avance. Afin de respecter la cohérence des données, il faut synchroniser le signal de contrôle à la sortie des opérateurs. En effet, dans un opérateur de délai $\delta$, le signal de contrôle doit sortir $\delta$ cycles après être entré. Il suffit donc d'insérer $\delta$ registres 1 bit sur le chemin du signal de contrôle.

Le code source VHDL correspondant au bloc de contrôle est présenté ci-dessous et le schéma

correspondant est donné Figure 3.14.

```
use work.borrow_save.all;

entity control_block is
        generic (delay : integer := 2);
        port (ctrl_in, clk : in bit ; ctrl_out, clear : out bit);
end control_block;

architecture struct of control_block is

        component flip_flop
                port(input, clk, reset: in bit ; output : out bit);
        end component;
        for all : flip_flop use entity work.flip_flop(behav);

        component flip_flop_wr
                port(input, clk : in bit ; output : out bit);
        end component;
        for all : flip_flop_wr use entity work.flip_flop_wr(behav);

        signal i1,i2 : bit;
        signal t : bit_vector(1 to delay);
        signal internal_reset : bit;
begin

        ff_start : flip_flop_wr port map(ctrl_in,clk,i1);

        i2 <= ctrl_in and not(i1);

        clear <= not(i2);

        internal_reset <= not(i2);

        fg : for i in 1 to delay-1 generate
                ff_delay : flip_flop_wr port map(t(i),clk,t(i+1));
        end generate fg;

        ff_delay_end : flip_flop port map(t(delay),clk,internal_reset,ctrl_out);

        t(1) <= i2;

end struct;
```

### 3.2.4 Plateforme de test

Lors de la réalisation d'un algorithme de calcul utilisant plusieurs opérateurs en-ligne, il faut faire attention de bien respecter les délais et les connexions entre les opérateurs. Il faut donc valider les implantations. Ce problème de la validation est déjà important dans les conceptions classiques de circuits intégrés, mais la réalisation d'algorithmes de calcul pose quelques problèmes supplémentaires. En effet, la preuve mathématique de la validité d'un algorithme n'est pas suffisante pour certifier le bon fonctionnement d'une réalisation car bon nombre de détails techniques sont occultés dans la modélisation mathématique. L'exemple typique est la détermination du temps de cycle qui doit prendre en compte le délai électrique du plus grand chemin que le signal peut traverser. Le problème en arithmétique est que la longueur de ce type de chemin peut varier considérablement

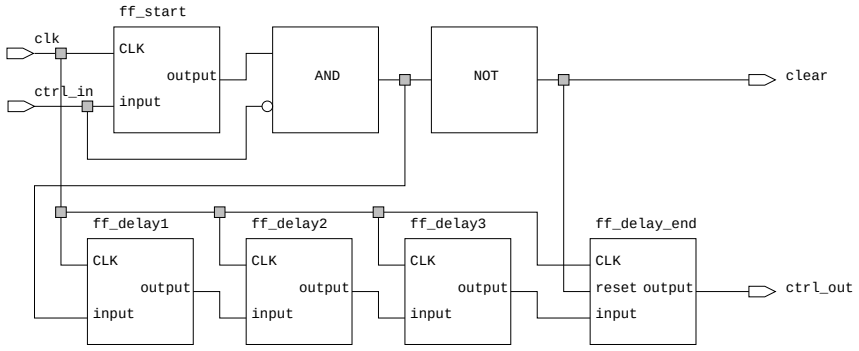



FIG. 3.14 – *Bloc de contrôle (délai 4).*

selon les données (par exemple la différence entre le meilleur cas et le pire cas d'un additionneur séquentiel), et est donc difficilement estimable de façon statique.

La solution classique est l'utilisation de vecteurs de test. On injecte en entrée du circuit un ensemble de valeurs dont on connaît la sortie théorique. A chaque présentation, on compare la valeur en sortie du circuit avec la valeur théorique. Il existe des programmes qui génèrent des vecteurs de test mais il est très difficile de trouver les cas problématiques en arithmétique. En effet, seule une solide connaissance des algorithmes et des représentations utilisées permet de construire les "cas limites".

Nous avons choisi d'effectuer deux types de tests :

- Des tests exhaustifs pour valider les opérateurs en-ligne (sur des tailles raisonnables d'opérandes)
- Des tests aléatoires pour valider un algorithme utilisant tout un ensemble d'opérateurs en-ligne

Le test exhaustif pour des petits nombres est tout à fait réalisable. Par exemple, pour tester un multiplieur en-ligne de deux nombres de 4 chiffres, il faut tester les $2^{4\times2\times2}$ combinaisons possibles. Ce genre de tests ne prend que quelques dizaines de minutes en utilisant un simulateur logique (style simulateur VHDL).

Mais on voit bien qu'il n'est pas possible de tester exhaustivement des opérateurs plus grands et encore moins des algorithmes qui utilisent de ces plusieurs opérateurs. Nous avons donc réalisé un système permettant de tester des opérateurs ou des algorithmes complexes à l'aide de vecteurs de test.

Notre système de test est basé sur l'utilisation de la surcharge des opérateurs en VHDL (*overloading*). Il est possible en VHDL de redéfinir des fonctions existantes pour quelles s'appliquent à différents types de données. Par exemple, en surchargeant la fonction "+" pour nos types de données (nombres écrits en *borrow-save* par exemple), il est possible d'exécuter une première fois l'opération $a + b$ avec $a$ et $b$ deux nombres codés en *borrow-save* et une deuxième fois avec $a$ et $b$ codés en virgule flottante. La première exécution sera faite par le simulateur en utilisant le code VHDL correspondant à l'opérateur d'addition en-ligne. La seconde exécution sera faite en utilisant soit l'additionneur flottant (ou entier) soit en utilisant une bibliothèque de calcul certifiée. L'opérateur décrit en VHDL peut être considéré comme valide si la valeur retournée par le simulateur

VHDL et par le processeur flottant sont égales.

Nous avons surchargé les opérations usuelles avec les types de données spécifiques à la représentation *borrow-save* en leur associant la description VHDL des opérateurs en-ligne correspondants. Le test d'un algorithme $A$ devient alors particulièrement simple puisqu'il suffit d'exécuter les opérations suivantes :

① écrire $A$ en VHDL en utilisant des nombres flottants (ou entiers)

② exécuter $A$ et récupérer son résultat $R_A$

③ modifier le programme VHDL de $A$ en $A'$ en utilisant les nombres *borrow-save*

④ exécuter $A'$ et récupérer son résultat $R_{A'}$

⑤ comparer $R_A$ et la conversion de $R_{A'}$ en flottant

## 3.3 Applications

### 3.3.1 Réseaux de neurones MLP sur FPGA

Avec Bernard Girau (LIP), nous avons étudié l'implantation sur FPGA d'un algorithme d'apprentissage de réseaux de neurones. Le modèle de réseaux de neurones utilisé est un perceptron multicouche, l'algorithme d'apprentissage est celui de la descente de gradient. L'utilisation de l'arithmétique en-ligne était motivée par le nombre de connexions dans le réseau de neurones et la nature de certaines opérations comme le calcul de tangentes hyperboliques, irréalisables en calcul série poids faibles en tête.

Un perceptron multicouches ou MLP (MultiLayer Perceptron) est composé de couches successives de neurones. Deux couches adjacentes sont complètement connectées, c'est-à-dire chaque neurone d'une couche est connecté à tous les neurones de la couche suivante. Les applications classiques en connexionisme utilisent des réseaux MLP à quelques couches (de 1 à 4 ou 5). Chaque couche contient une centaine de neurones au maximum. On voit ici la nécessité des communications séries entre les couches. Chaque neurone effectue un calcul du type :

$$s = \tanh\left(\theta + \sum_{i=1}^{n} w_i x_i\right)$$

Le fonctionnement du réseau en apprentissage est décomposé en itérations. Chaque itération consiste en deux phases : la présentation sur les entrées du réseau d'un exemple dont on connaît la valeur de sortie et le calcul de la sortie du réseau durant la première phase ou phase *forward*, la deuxième phase ou phase *backward* consiste à calculer la différence entre la valeur théorique et la réponse du réseau et à remettre à jour des poids du réseau en fonction de l'erreur mesurée.

Nous résumons ci-dessous les principales caractéristiques de la solution proposée dans [GT96].

| | |
|---|---|
| codage des nombres | virgule fixe sur 24 bits (8 bits entiers et 16 bits fractionnaires) |
| cible technologique | 2 FPGA XC 4025 et 1 XC 4020 de Xilinx |
| fréquence du système | 33 MHz |
| phase *forward* | 8 multiplieurs et un additionneur à 9 entrées sur un XC 4025 (environ 700 CLB[3] pour le calcul et 250 pour le contrôle de la mémoire et les communications) |
| phase *backward* | 8 multiplieurs, un additionneur de 8 entrées et un multiplieur par une constante (le coefficient d'apprentissage) sur un XC 4025 (800 CLB pour le calcul et 100 pour le contrôle) |
| calcul des tanh | polynômieur de degré 5 sur un XC 4020 (450 CLB pour le polynômieur et 250 CLB pour la table) |
| performance estimée | environ 5 millions de connexions mises à jour par seconde |

Le parallélisme introduit par le pipeline au niveau du bit et par le traitement de plusieurs entrées (8 entrées d'un neurone) simultanément permet d'obtenir des performances supérieures à celles obtenues par des solutions logicielles. La flexibilité d'une solution FPGA (reprogrammables, comme les Xilinx) permet d'envisager d'utiliser notre architecture pour le prototypage ou pour des applications évolutives.

### 3.3.2 Contrôle numérique sur FPGA

Dans le cadre d'une collaboration avec le Centre Suisse d'Electronique et de Microtechnique (Neuchatel), nous avons réalisé avec Martin Dimmler de l'EPFL un régulateur numérique pour une application de positionnement rapide d'un miroir, pour une application de télécommunications aérospatiales.

L'algorithme utilisé pour contrôler la position du miroir est le régulateur PID (*proportionnal-integral-differential*) qui est l'un des algorithmes les plus employé dans les applications de contrôle numérique. Les équations qui régissent le fonctionnement du régulateur PID sont données ci-dessous :

$$\begin{aligned} u_k &= K\left(e_k + \frac{h}{T_I}x_{i,k} + T_D x_{d,k}\right) \\ \text{où}\quad x_{i,k} &= e_k + x_{i,k-1} \\ x_{d,k} &= \frac{1}{h+\tau}\left(e_k - e_{k-1} + \tau x_{d,k-1}\right) \\ e_k &= s_k - y_k \end{aligned} \tag{3.2}$$

où $e_k$ est l'erreur entre la commande $s_k$ et la sortie mesurée $y_k$, $x_{i,k}$ et $x_{d,k}$ sont des caractéristiques de l'état du régulateur, $u_k$ est la sortie, $h$ la période d'échantillonage et $\tau$, $K$, $T_I$, $T_D$ des constantes utilisées pour le réglage du PID. La Figure 3.15 représente l'ensemble des opérateurs en-ligne correspondant au régulateur PID.

3. Bloc logique dans les FPGA Xilinx (*configurable logic bloc*).

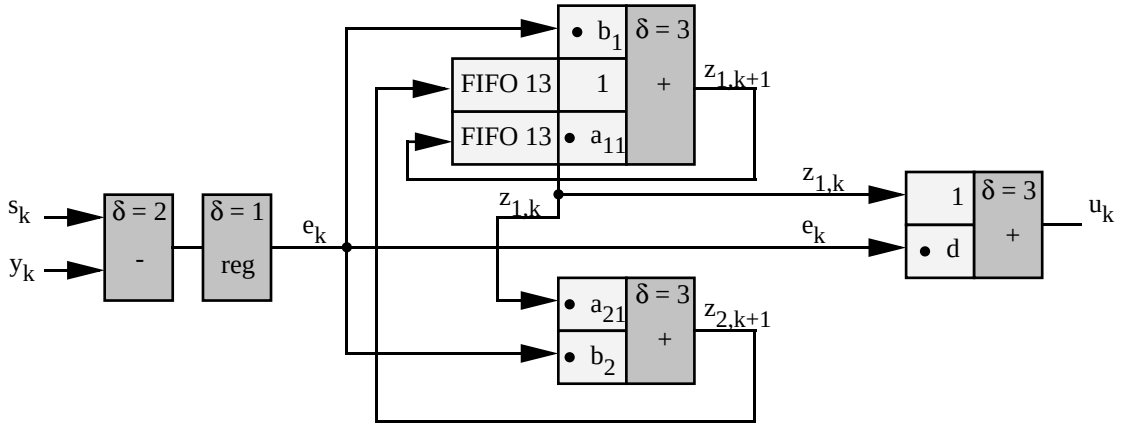



FIG. 3.15 – *Schéma du régulateur PID réalisé.*

Un prototype a été réalisé avec des FPGA Actel 1280A et 1240A. Ce choix des FPGA est important pour cette application car seuls les FPGA Actel (technologie anti-fusible) sont certifiés pour les applications aérospatiales.

Le premier FPGA (Actel 1240A) est utilisé pour la sérialisation des données qui proviennent de convertisseurs A/D parallèles, pour l'interfaçage avec la machine hôte (un compatible PC) et pour le contrôle des entrées/sorties. Le schéma du circuit de ce FPGA est présenté Figure 3.16. Seuls 118 blocs élémentaires du FPGA sur les 684 disponibles sont utilisés. Ce FPGA pourrait devenir complètement inutile avec des convertisseurs A/D et D/A série poids forts en tête. Ces convertisseurs existent bien mais pas pour des applications aérospatiales.

Le second FPGA (Actel 1280A) est utilisé pour l'implantation du régulateur PID. Le schéma correspondant est présenté Figure 3.17. La surface occupée est de 758 cellules sur les 1232 disponibles.

La fréquence de fonctionnement du système est de 10 MHz seulement en raison des convertisseurs A/D utilisés. Les circuits réalisés sur les deux FPGA peuvent supporter des fréquences beaucoup plus élevées.

Le but de cette réalisation était de démontrer le potentiel de l'arithmétique en-ligne sur FPGA. Une comparaison avec une solution équivalente basée sur des DSP est en cours de réalisation. Les premiers résultats sont extrêmement prometteurs. En effet, la solution FPGA est près de 4 fois moins volumineuse (paramètre important pour les applications embarquées) et sa consommation est environ 20 fois moins importante que la solution basée sur les DSP.

Les résultats et des méthodes mis en œuvre dans cette réalisation ont été publiés dans [TD97, Tis96] et soumis au journal *IEEE Transactions on Mechatronics*.

## 3.4 Conclusion

L'arithmétique en-ligne conduit à des réalisations beaucoup moins volumineuses qu'en arithmétique parallèle et qui n'utilisent qu'un nombre très limité d'entrées/sorties. De plus, l'arithmétique série poids faibles en tête ne permet pas de calculer des opérations autres que des additions et des multiplications et elle ne peut pas être utilisée directement dans des applications qui néces-

sitent des conversions analogique/numérique car il n'existe pas de convertisseur A/D poids faibles en tête. Tous ces arguments font de l'arithmétique en-ligne un bon candidat pour la représentation des nombres et le calcul dans les applications embarquées qui nécessitent des calculs complexes.

Nous avons réalisé une bibliothèque VHDL contenant tous les outils et les éléments nécessaires pour concevoir et réaliser des algorithmes en-ligne complexes. L'ensemble des opérateurs est décrit de façon modulaire et hiérarchique, ce qui offre une bonne réutilisabilité des codes sources VHDL. Nous avons simplifié la conception du contrôle global du circuit en utilisant une technique de contrôle distribué particulièrement simple à mettre en œuvre. La reprogrammabilité des FPGA associée au système de test intégré dans la bibliothèque permet de concevoir et de valider rapidement des algorithmes. On minimise ainsi l'un des principaux problèmes en conception de circuits, à savoir le temps de développement.

Les performances obtenues sur nos deux applications de test sont très encourageantes. Pour le moment les cibles technologiques sont relativement limitées du fait de la granularité des cellules élémentaires choisies. Le gain principal obtenu par l'utilisation de l'arithmétique en-ligne est dû au parallélisme au niveau du bit et à l'augmentation du nombre de calculs effectués simultanément du fait de la petite taille des opérateurs. L'un des grands apports de l'arithmétique en-ligne est aussi la diminution du nombre d'entrées/sorties dans les circuits. Les performances en terme de vitesse sur les FPGA, même si elles restent inférieures aux fréquences obtenues sur les meilleurs ASIC, permettent de rivaliser et même de gagner par rapport à certaines réalisations à base de processeurs puissants ou de circuits intégrés standards. La grande régularité des opérateurs, la localité des communications entre les opérateurs et des signaux de contrôle contribuent à la réalisation d'un pipeline très efficace.

Les perspectives de ce travail sont :

- La réalisation d'un système de conception automatisé des opérateurs d'évaluation de polynômes qui utilisent des tables sur les premiers chiffres pour réduire le délai. La bibliothèque sera alors en mesure de proposer des opérateurs pour calculer les fonctions élémentaires.

- Nous étudions actuellement l'utilisation d'une représentation redondante avec une base plus importante pour accélérer les opérations de division et de racine carrée. Le principal problème est ici de trouver un codage des chiffres qui permet des implantations simples et efficaces des opérateurs élémentaires (additions et produit de chiffres, opposé d'un chiffre, normalisation...). L'augmentation de la base aurait un double avantage: diminuer le temps de calcul des divisions et des racines carrées et la simplification du contrôle car les délais des opérateurs en-ligne sont beaucoup plus petits dans les bases supérieures à 2. Par exemple, en base 4, l'addition, la multiplication par une constante, la multiplication normale et l'élévation au carré sont des opérateurs de délai 1. Un stage de fin de diplôme est actuellement en cours sur ce sujet à l'*Ecole Polytechnique Fédérale de Lausanne.*

- Nous abordons aussi la consommation des opérateurs en-ligne pour effectuer des comparaisons avec des solutions conventionnelles. Ce point est particulièrement important mais il est relativement complexe car un grand nombre de paramètres entrent en compte dans cette étude. Cette étude sera l'un des mes axes de recherche durant mon séjour post-doctoral au *Centre Suisse d'Electronique et de Microtechnique* à Neuchatel.

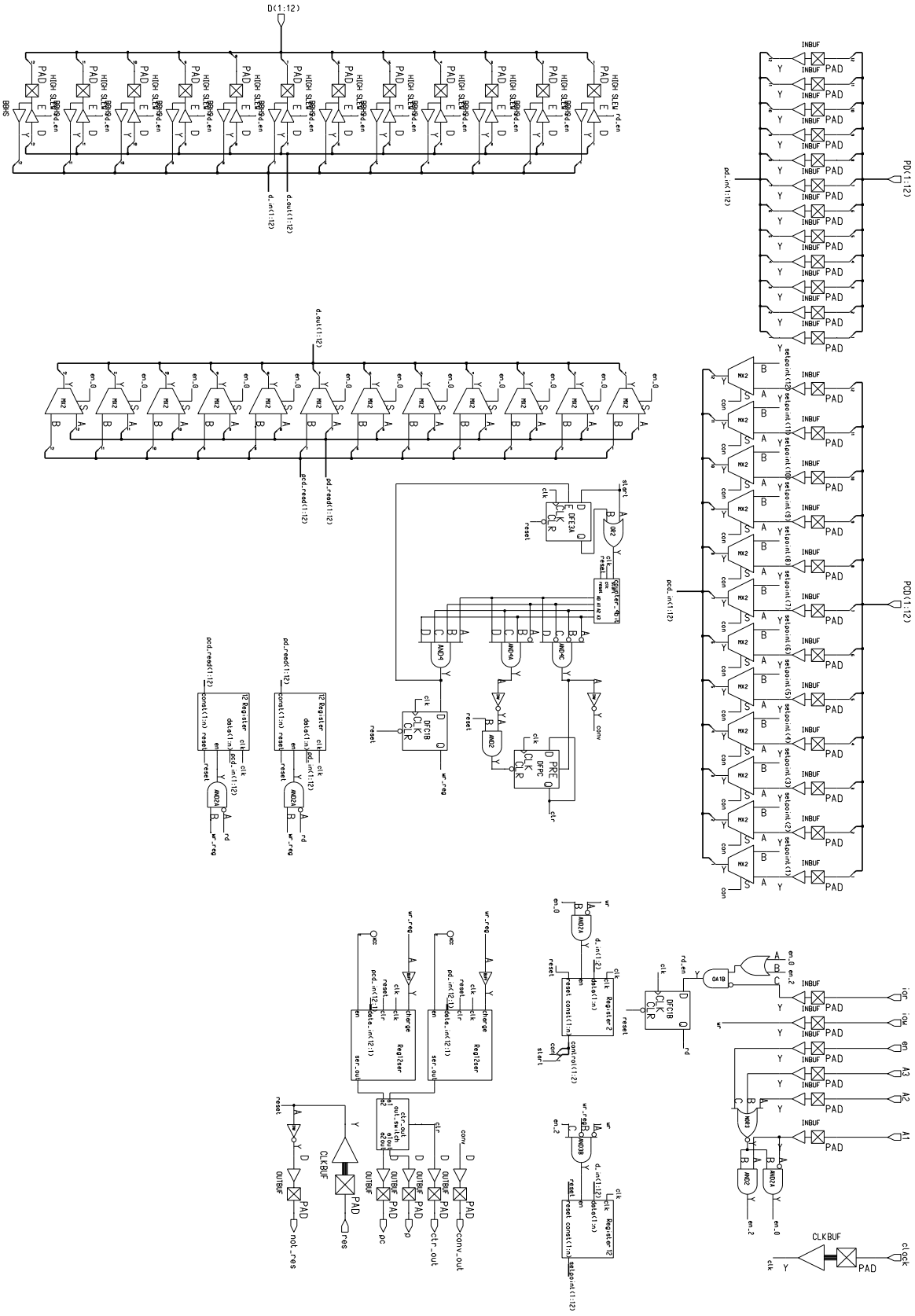



FIG. 3.16 – *Schéma du circuit du premier FPGA (interfaçage PC, E/S).*

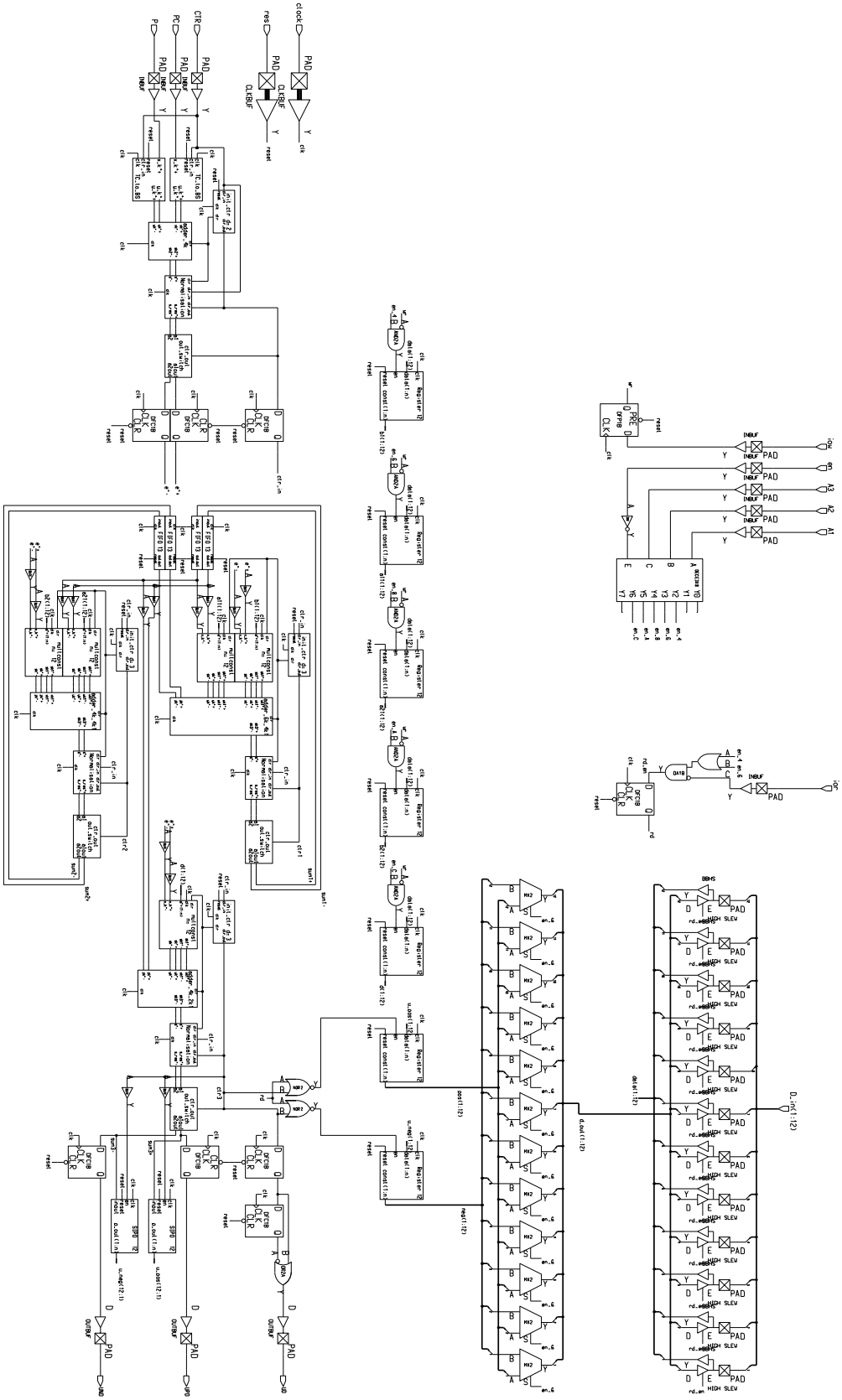


FIG. 3.17 – *Schéma du circuit du second FPGA (régulateur PID).*

# Opérateurs arithmétiques asynchrones

CHAPITRE

# 4 Opérateurs arithmétiques asynchrones

LA plupart des circuits intégrés logiques sont conçus en se basant sur deux hypothèses fondamentales : la discrétisation des signaux électriques et la discrétisation du temps. Les signaux manipulés sont en général binaires. Ceci permet des réalisations électriques simples et performantes. De plus, cette discrétisation binaire des signaux offre un cadre de conception fiable et efficace grâce à l'algèbre de Boole. La discrétisation du temps permet quant à elle de s'affranchir de nombreux problèmes d'implantation comme celui des valeurs transitoires.

Les circuits intégrés analogiques utilisent des signaux électriques qui ont des valeurs continues. Ces circuits permettent de réaliser très rapidement des opérations comme l'addition ou la multiplication de tensions avec un petit nombre de transistors. Toutefois, les applications des circuits intégrés analogiques sont restreintes car la précision des valeurs manipulées, la possibilité de mémorisation et la fiabilité des opérations sont très limitées. Enfin, les circuits intégrés analogiques sont très sensibles aux bruits (bruits dans les alimentations, perturbations électromagnétiques...), ce qui rend leur emploi particulièrement délicat et donc réservé à des applications très particulières.

Si la discrétisation des signaux électriques est difficile à remettre en cause généralement, ceci est plus simple pour la discrétisation du temps. En conséquence, il y a depuis longtemps deux modes de fonctionnement des circuits intégrés : le fonctionnement *synchrone* et le fonctionnement *asynchrone*.

Les circuits synchrones séparent leurs étages de calcul par des registres commandés par une horloge. Il n'y a échange de données entre les étages qu'aux fronts d'horloge. La période de l'horloge est choisie pour garantir le bon fonctionnement du circuit, c'est-à-dire qu'elle doit être plus grande que le temps de propagation du signal dans le pire cas. La conception de circuits synchrones est de plus en plus complexe avec l'augmentation de la fréquence d'horloge. Par exemple, lors de la réalisation du processeur DEC Alpha 21164a à 500 MHz, plus de la moitié des effectifs de l'équipe de conception s'occupait des problèmes relatifs à l'horloge. La dérive des horloges, les pics de consommation aux fronts d'horloge sont parmi les problèmes rencontrés lors de la conception d'un circuit intégré numérique synchrone rapide.

Les circuits asynchrones sont des circuits qui n'utilisent pas d'horloge globale pour cadencer les échanges de données entre leurs différentes parties. Ce sont les cellules elles-mêmes qui assurent localement la synchronisation. C'est pour cela que l'on parle aussi de *circuits localement synchronisés* ou *auto-séquencés* (*self-timed circuits*). Nous présenterons en 4.1 le fonctionnement et les principales caractéristiques de ces circuits.

La principale caractéristique des circuits asynchrones qui nous intéresse en arithmétique est la capacité de ces circuits à calculer en "temps moyen". Le temps de calcul d'un opérateur asynchrone est le temps qu'il faut au signal de sortie pour être l'image fonctionnelle des entrées. Suivant les valeurs des opérandes, ce temps peut varier considérablement. Par exemple, dans le cas d'un

additionneur séquentiel binaire, c'est la longueur de la plus longue chaîne de propagation de la retenue qui va déterminer le temps de calcul. Cette variabilité du temps de calcul exprime en fait le parallélisme dynamique introduit par les données, dans l'exemple de l'additionneur, plusieurs retenues se propagent en même temps à différentes positions. En plus de cette variation que l'on peut qualifier de "fonctionnelle", il existe une variation du temps de calcul liée à l'implantation physique. Par exemple, la convergence électrique dans un circuit intégré peut être plus ou moins rapide en fonction du nombre de transitions des différents signaux (en CMOS, les transistors P et les transistors N ne sont pas identiques et les transitions de "1" vers "0" et de "0" vers "1" n'ont pas les mêmes durées).

C'est cette notion de temps de calcul variable et de calcul en temps moyen dans les opérateurs arithmétiques qui va nous intéresser dans ce chapitre. Nous allons essayer de comprendre quels sont les mécanismes sous-jacent à ce mode de calcul et essayer de minimiser le temps de calcul moyen de quelques opérateurs arithmétiques. Nous effectuerons un parallèle avec la conception d'opérateurs arithmétiques synchrones pour lesquels "il suffit" de minimiser le pire cas.

La première partie de ce chapitre présente le fonctionnement général et les nombreux potentiels des circuits asynchrones. Nous décrivons dans la seconde partie le simulateur que nous avons réalisé pour effectuer les expériences des parties suivantes. La troisième partie est consacrée à l'étude de certains additionneurs asynchrones. Nous détaillerons leur réalisation et les résultats de leur comportement obtenus en simulation. La quatrième partie est l'étude probabiliste du comportement de ces additionneurs asynchrones. Nous aborderons dans la cinquième partie de ce chapitre le cas d'autres opérations arithmétiques comme la multiplication et la division. Enfin, nous donnerons les perspectives de ce travail.

Ce travail a été réalisé en collaboration avec Jean-Michel Muller (LIP), Marc Renaudin (CNET) et Jean-Marc Vincent (LMC-IMAG), et certaines des parties de ce chapitre ont été publiées dans [MTV97].

## 4.1 Opérateurs asynchrones

Nous donnons ici quelques notions élémentaires sur le fonctionnement des circuits asynchrones. On trouve dans [Hau95, Has95, RH94a, RH94b] une étude plus approfondie du fonctionnement de ces circuits, de leurs propriétés et des implantations utilisées en pratique. On trouve dans [WDF+97] une présentation du processeur asynchrone AMULET1 (microprocesseur ARM complètement asynchrone). Dans [JB90], on trouve la description d'un processeur de traitement du signal (DSP) asynchrone. Différentes applications avec circuits asynchrones ont été réalisées, par exemple, on trouve dans [RRPB96] la description d'un réseau de processeurs dans un circuit asynchrone utilisable pour le traitement d'images. Je remercie les auteurs de [RRV97, Ren96] pour leur aide dans la réalisation de cette présentation des opérateurs asynchrones.

### 4.1.1 Le contrôle local

Le principe de base fondamental du fonctionnement des circuits asynchrones est la localisation des signaux de contrôle. Le fonctionnement d'une cellule élémentaire asynchrone est décomposé en différentes étapes de calcul et de communication. Lorsque toutes les informations d'entrée sont présentes, le contrôle local déclenche le début du traitement de ces informations. Une fois le traitement effectué, la cellule émet le résultat vers la cellule suivante. La cellule peut alors signaler

à la cellule précédente que les informations qu'elle lui avait envoyé ont été prises en compte et qu'elle peut maintenant envoyer les informations suivantes. La cellule doit alors attendre elle-même que la cellule suivante lui signale que les informations ont été utilisées, et ainsi de suite. Il y a donc un échange bidirectionnel de signaux. Chaque communication doit être acquittée une fois les informations prises en compte. Les communications sont dites à "poignée de main".

### Les protocoles de communications

Pour assurer le bon fonctionnement des échanges à poignée de main, deux principaux protocoles de communication sont utilisés en général :

- Le protocole "2 phases" (ou NRZ pour Non Retour à Zéro ou encore *half-handshake*) est illustré sur la Figure 4.1a. La première phase est la phase active du récepteur qui détecte la présence de nouvelles données, effectue le traitement et génère un signal d'acquittement. La seconde phase est la phase active de l'émetteur qui détecte le signal d'acquittement et émet les nouvelles données lorsqu'elles sont disponibles.
- Le protocole "4 phases" (ou RZ pour Retour à Zero ou encore *full-handshake*) est illustré sur la Figure 4.1b. La phase 1 est la première phase active du récepteur qui détecte la présence de nouvelles données, effectue le traitement et génère un signal d'acquittement. La phase 2 est la première phase active de l'émetteur qui détecte le signal d'acquittement et émet des données invalides (retour à zéro). La phase 3 est la deuxième phase active du récepteur qui détecte le passage des données dans l'état invalide et place le signal d'acquittement dans l'état initial (retour à zéro). La phase 4 est la deuxième phase active de l'émetteur qui détecte le passage à zéro du signal d'acquittement, il est alors prêt à émettre de nouvelles données.

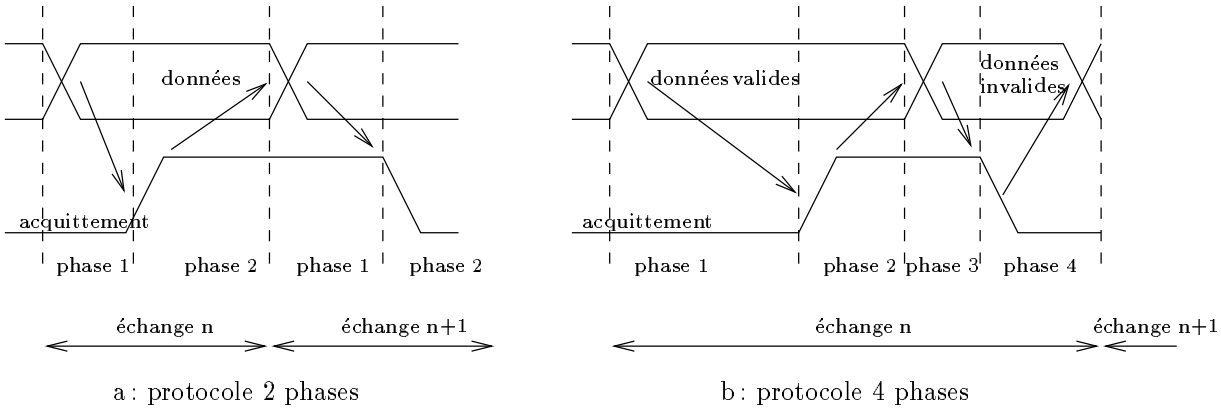



FIG. 4.1 – *Protocoles de communication à 2 phases et à 4 phases.*

Le protocole à 4 phases requiert deux fois plus de transitions que le protocole à 2 phases, mais il est possible de masquer les phases de retour à zéro pendant certains traitements. Le protocole à 2 phases nécessite un matériel plus important car il doit détecter des transitions et pas des niveaux.

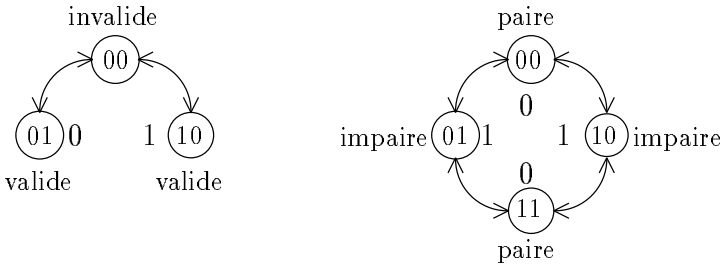
Dans les deux cas, on remarque que chaque changement d'un signal est acquitté par un autre signal. Un changement dans les données est acquitté par le signal d'acquittement et chaque changement du signal d'acquittement est acquitté par un changement des données. C'est ce qui permet d'assurer l'insensibilité aux variations des délais.

### Le codage des données

Les protocoles précédents sont basés sur la détection de la présence de signaux et sur la génération d'un acquittement en fin de traitement. Il faut donc être capable de détecter tous les changements d'un signal. L'utilisation d'un seul fil par bit n'est pas possible car elle ne permet pas de détecter que la nouvelle valeur prend un état identique au précédent. Il faut donc utiliser un codage particulier des données ([Ver88]). Il faut donc utiliser deux fils par bit ce qui augmente la surface des circuits par rapport aux réalisations synchrones. Deux codages sont possibles avec 2 bits pour exprimer seulement 2 valeurs ("0" et "1") : le codage à 3 états et le codage à 4 états.

Dans le codage à 3 états (illustré Figure 4.2a), un fil prend la valeur 1 pour coder la valeur "1" et l'autre fil prend la valeur 1 pour coder la valeur "0". L'état "11" est interdit alors que l'état "00" représente l'état invalide (utile pour le protocole 4 phases). Ce codage garantit que le passage d'un état à l'autre se fait toujours en ne changeant qu'un seul des deux bits, ce qui peut être détecté sans risque d'aléa. Le signal de sortie est généré lorsque l'on détecte que l'un des deux bits est passé à 1. Ce codage des données est particulièrement bien adapté au protocole 4 phases.

Dans le codage à 4 états (illustré Figure 4.2b), chaque valeur "1" ou "0" est codée par deux combinaisons. L'une des combinaisons est considérée de parité paire et l'autre de parité impaire. L'utilisation du code de Gray permet de garantir qu'un seul des deux bits est modifié à chaque changement d'état. L'analyse de la parité permet de détecter la présence d'une nouvelle donnée et de générer le signal de fin de calcul. Ce codage est particulièrement bien adapté au protocole 4 phases, mais l'implantation de ce codage est plus complexe et plus lente que celle du codage 3 états (machine à états + mémoire pour stocker la dernière parité).




a : codage 3 états b : codage 4 états

FIG. 4.2 – *Codage des données à 3 et 4 états.*

De nombreuses solutions ont été proposées pour l'implantation du codage 3 états, en particulier, on trouve dans [Has95, RH94a, RH94b] l'étude et des utilisations de la logique "cascode" différentielle (DCVSL pour *differential cascode voltage switch logic*), implantation logique particulièrement efficace.

On trouve dans [DDH94] une autre technique pour la génération des signaux de fin de calcul. L'idée est de mesurer le courant qui circule dans les portes logiques car dans certaines technologies (c'est le cas en technologie CMOS par exemple) il va diminuer rapidement à la fin du calcul de la porte. Cette technique originale est néanmoins difficile à mettre en œuvre.

### 4.1.2 Les avantages des circuits asynchrones

Les circuits intégrés fonctionnant dans le mode asynchrone présentent les particularités suivantes :

- **Le calcul en temps moyen** : comme nous l'avons vu dans l'introduction, le temps de calcul d'un opérateur asynchrone est le temps qu'il faut au signal de sortie pour être l'image fonctionnelle des entrées. La variabilité de ce temps de calcul traduit le parallélisme dynamique maximal pour chaque valeur des entrées. Par exemple, le temps moyen de calcul d'un additionneur binaire séquentiel de $n$ bits est en $\mathcal{O}(\log_2 n)$ si on suppose la distribution uniforme des entrées (cf [BGN46]). On a vu qu'il est aussi possible d'utiliser certaines spécificités technologiques pour diminuer le temps moyen de calcul (différence de vitesse de transition entre les transistors P et les transistors N). Toutefois, si le temps de calcul minimum et le temps de calcul maximum peuvent en général être facilement déterminés, le temps de calcul moyen peut nécessiter une modélisation statistique très complexe.

- **L'absence de signal d'horloge** : cette caractéristique est l'une des causes de l'intérêt actuel pour les circuits asynchrones. En effet, les problèmes de manipulation des horloges sont de plus en plus critiques avec l'augmentation de la fréquence dans les circuits intégrés synchrones. La conception d'un circuit synchrone demande, pour s'assurer de la correction d'une cellule, de tenir compte des caractéristiques des signaux d'horloge (estimations de délais, retards, métastabilité...), ce qui complique considérablement la spécification de la cellule. En asynchrone, les éléments de synchronisation sont locaux et distribués dans tout le circuit ce qui simplifie la conception car il suffit de s'assurer que les cellules respectent le protocole de communication pour garantir le bon fonctionnement du circuit. Il n'y a plus de paramètre global à prendre en compte, la conception des différentes parties d'un circuit peuvent être réalisées séparément sans aucun risque.

  Un autre avantage lié à la suppression des signaux d'horloge est la meilleure répartition des transitions dans le temps. Dans les circuits synchrones, tous les opérateurs deviennent actifs aux fronts d'horloge ce qui introduit des pics de courant et donc du bruit dans les alimentations. Ce problème est considérablement plus faible en asynchrone.

- **La modularité** : la localisation des signaux de contrôle et l'utilisation d'un protocole de communication unique permettent une conception très simple d'opérateurs complexes en connectant des modules existants. La distribution et la hiérarchisation de la phase de conception est beaucoup plus simple en asynchrone car la seule hypothèse nécessaire pour assurer une bonne interopérabilité avec les autres composants est le respect du protocole de communication. La conception modulaire de circuits intégrés asynchrones est donc beaucoup plus fiable. La migration technologique est aussi grandement facilitée par le fait qu'en asynchrone, le fonctionnement d'un circuit est indépendant de la réalisation effective pourvu que le protocole de communication soit respecté.

- **La faible consommation** : dans certaines technologies, une cellule à laquelle on ne présente pas de données ne consomme pas (c'est le cas en CMOS). L'activité conditionnelle est une caractéristique naturelle des circuits asynchrones. Des techniques similaires ont été utilisées dans les circuits synchrones (arrêt des horloges dans certaines parties du circuit), mais leur conception est très délicate et nécessite un matériel supplémentaire (la manipulation de différentes horloges est extrêmement délicate pour des fréquences élevées).

La réduction de la tension d'alimentation des circuits est souvent utilisée pour limiter la puissance consommée (la puissance consommée varie comme le carré de la tension d'alimentation). Les opérateurs asynchrones sont bien adaptés à cette technique car la réduction de la tension d'alimentation ne provoquera pas de dysfonctionnement (jusqu'à un certain point) mais seulement une augmentation des délais des opérateurs. Le processeur élaboré à Caltech [MBL+89] fonctionne encore à une tension de 0.5 Volt (il a même été alimenté par une pomme de terre équipée de deux électrodes !). Ce principe de la diminution de la tension d'alimentation a été aussi utilisé par Philips pour l'implantation d'un circuit correcteur d'erreurs assez élégant [BBK+94]. Ce circuit schématisé à la Figure 4.3 à une consommation qui dépend directement du débit des données qui arrivent en entrée. Suivant le taux de remplissage de la FIFO d'entrée, la tension d'alimentation est modifiée via le convertisseur de tensions continues. Plus la charge en entrée est importante plus la tension augmente et donc plus le traitement est rapide (et inversement). On a donc une adaptation des performances du circuit aux besoins réels.

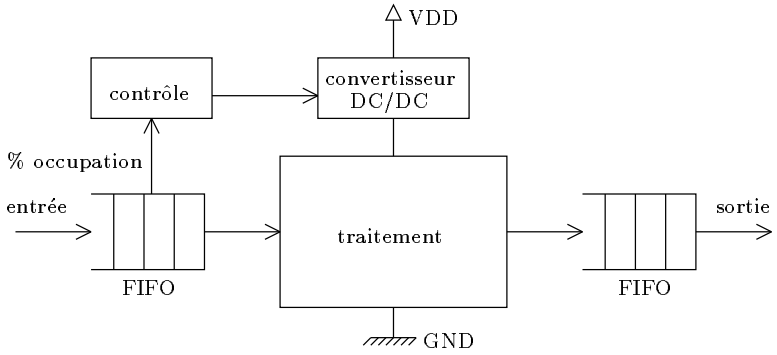



FIG. 4.3 – *Régulation de la tension d'alimentation en fonction de la charge (circuit Philips).*

### 4.1.3 Les types de circuits asynchrones

La discrétisation du temps dans les circuits synchrones permet d'ignorer certains aléas de fonctionnement puisqu'elle interdit de regarder les données lorsqu'elles sont instables. Ces aléas peuvent devenir problématiques dans le cas des circuits asynchrones. Par exemple, le cas des aléas statiques est un cas typique de phénomènes qui ne posent aucun problème en synchrone mais qui ne doivent pas survenir dans les circuits asynchrones. En effet, ces transitoires (*glitches*) pourraient être interprétées comme des événements porteurs d'information par le protocole de communication. Ce type d'aléa est illustré Figure 4.4. Il existe aujourd'hui des techniques qui permettent de réaliser des circuits en garantissant l'absence de ce genre de phénomènes, c'est la conception *hazard free*.

Les circuits asynchrones demandent donc de réaliser toutes les parties du circuit (combinatoires et séquentielles) avec attention et finesse. Cette contrainte existe aussi dans les circuits synchrones mais à un niveau plus haut du circuit, c'est la gestion des signaux d'horloge.

La suppression de la discrétisation du temps entraîne donc un prix à payer. Toutefois, certaines réalisations asynchrones parviennent à relâcher la contrainte de la correction fonctionnelle indépendamment des délais en introduisant des hypothèses temporelles qui conduisent à des mises en œuvre plus simples. Il existe en pratique différentes méthodes de conception de circuits asynchrones qui se caractérisent par leurs hypothèses temporelles. On trouve dans [Has95] une présentation complète des différentes classes de circuits asynchrones.

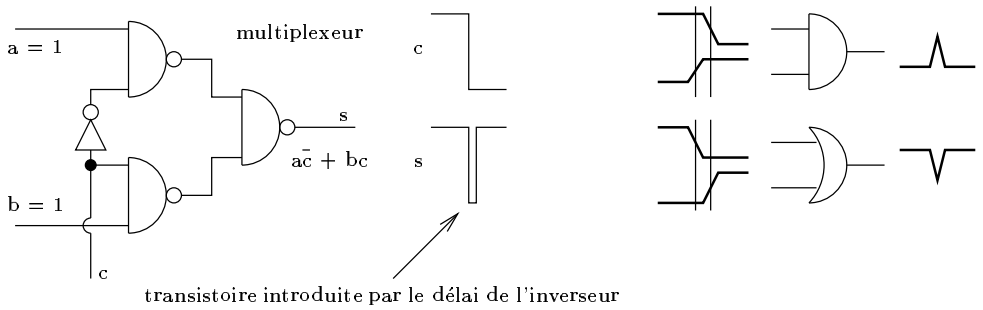



FIG. 4.4 – *Aléas de fonctionnement statiques.*

## 4.2 Simulation des opérateurs arithmétiques asynchrones

Nous verrons dans la suite que la modélisation mathématique des opérateurs arithmétiques asynchrones n'est pas forcément simple. De plus, cette modélisation suppose un certain nombre d'hypothèses qui masquent certains aspects de l'implantation (les hypothèses sont forcément simplificatrices par rapport à la réalité). Nous avons donc réalisé un simulateur pour étudier les opérateurs arithmétiques asynchrones. Ce simulateur utilise en entrée une description de l'opérateur à simuler qui est transformée en un modèle proche d'un modèle flot de données pour la simulation. De plus, notre simulateur intègre différentes fonctionnalités qui permettent de réaliser automatiquement des mesures statistiques pour pouvoir ensuite comparer les résultats de simulations obtenus avec les résultats théoriques donnés par les modèles probabilistes que nous étudierons en 4.4.

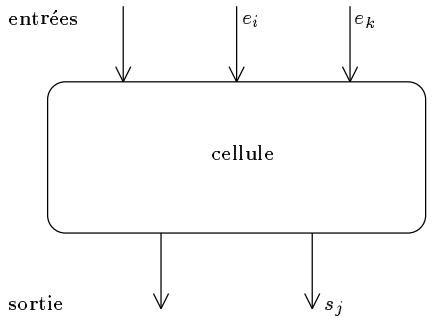
### 4.2.1 Modèle de circuit asynchrone pour la simulation

Le choix d'un modèle pour la simulation est toujours délicat. Il faut que ce modèle soit suffisamment proche de la réalité pour que les résultats soient utilisables et représentatifs du fonctionnement d'un circuit réel, mais il doit être suffisamment simple pour simuler des opérateurs complexes (avec beaucoup de portes) et tester de nombreux échantillons en un temps acceptable. Le modèle que nous avons retenu est proche d'un modèle "flot de données". Il est basé sur les deux entités suivantes :

- Un **signal** est un couple $(v, d)$ où $v$ est la valeur effective du signal (en binaire $v \in \{0, 1\}$) et où $d$ est la date de validité du signal. Nous avons vu qu'il faut être capable de prendre en compte des temps de propagations différents pour les portes mais aussi pour les fils. La date de validité permet de modéliser simplement et fidèlement le fonctionnement du contrôle local des opérateurs asynchrones.

- Une **cellule**, illustrée sur la figure 4.5, est composée d'un nombre quelconque de ports d'entrée et de ports de sortie. A chaque port correspond un signal. A chaque sortie est associée la table des transitions et la valeur des délais pour chacune des combinaisons des entrées. Les sorties peuvent être des fonctions de n'importe quel sous-ensemble des entrées.

Le circuit est modélisé par un graphe dont les sommets sont les cellules qui correspondent aux portes logiques ou aux opérateurs du circuits. Les arcs du graphe sont les signaux. Le calcul d'un opérateur va se traduire par des flots dans le graphe. Ce principe de modélisation d'un opérateur par un graphe peut être utilisé récursivement. C'est-à-dire que l'on peut utiliser un autre graphe pour décrire le comportement d'une partie du circuit complet. Cette fonctionnalité n'est pas permise pour




| $e_i$ | $e_k$ | $s_j$ | $\Delta$ |
|---|---|---|---|
| 0 | 0 | 0 | $0.5ns$ |
| 0 | 1 | 1 | $1ns$ |
| 1 | 0 | 1 | $1ns$ |
| 1 | 1 | 1 | $1ns$ |

FIG. 4.5 – *Modèle de cellule de notre simulateur.*

le moment, mais elle nous permettra à terme de pouvoir décrire rapidement des circuit complexes en réutilisant des opérateurs existants.

La description de l'opérateur se fait dans un langage proche du langage VHDL. Nous avons choisi ce langage pour être en mesure de réutiliser certaines parties de nos descriptions dans des outils de synthèse. La description est faite en se basant sur une petite bibliothèque décrivant certaines cellules de base (portes logiques simples, cellule *full-adder*...).

```
use async.all;

entity add_seq is
        port ( a    : in signal_vector(1 to n);
               b    : in signal_vector(1 to n);
               c_in : in signal;
               s    : out signal_vector(1 to n);
               r    : out signal_vector(0 to n);
end add_ol;

architecture struct of add_seq is

        component fa
                port(x, y, z : in signal ; c, s : out signal);
        end component;
        for all : fa use entity fa(struct);

begin

        r(0) <= c_in;

        block_add_seq : for i in 1 to n generate

                fa_cell : fa port map (a(i),b(i),r(i-1),r(i),s(i));

        end generate block_add_seq;

end struct;
```

La date de validité du signal peut être utilisée pour prendre en compte des variations technologiques. Par exemple, on peut perturber ces dates lors du calcul pour observer le comportement du circuit face à certaines perturbations.

### 4.2.2 Fonctionnement du simulateur

Le fonctionnement du simulateur est assez simple. Au début, certaines cellules sont marquées comme "déclenchables" (ce sont les cellules qui ont des entrées valides, les opérandes et la retenue entrante dans le cas des additionneurs). La liste des cellules déclenchables est parcourue pour voir si il existe des cellules qui ont certaines de leurs sorties qui peuvent être calculées. C'est-à-dire que l'on regarde pour chaque cellule et pour chaque sortie de la liste si les entrées correspondantes sont présentes (test de validité du signal). Lorsque toutes les entrées correspondant à une sortie sont présentes on calcule la valeur de la sortie en utilisant la table de transition. La date de validité du signal de sortie est la date d'arrivée du dernier signal reçu en entrée (le maximum des dates) auquel on ajoute le délai qui correspond aux valeurs des entrées. Lorsque toutes les sorties d'une cellule sont connues, la cellule passe dans l'état "déclenché" et le nombre de cellules restant à déclencher est décrémenté. La fin de la simulation peut se produire de deux manières différentes. Soit toutes les cellules ont été déclenchées, auquel cas l'opérateur à terminé son calcul. Le temps de calcul de l'opérateur est alors la date de validité du dernier signal déclenché. Soit à cause d'une erreur dans la description du circuit, toutes les cellules n'ont pas été déclenchées et la liste des cellules déclenchables ne change pas après deux parcours successifs des sorties des cellules de cette liste.

Le simulateur a été réalisé en langage C++, le graphe est décrit sous forme de listes d'adjacence du fait du faible nombre de liens de communications entre les cellules (la matrice d'adjacence équivalente est souvent creuse). Différentes options de traitement sont proposées :

- Calcul pour une valeur particulière des entrées. Pour le moment, l'affichage de l'état de l'opérateur se fait en mode texte mais nous travaillons à une version graphique qui devrait être opérationnelle pour la soutenance de la thèse.

- Test exhaustif du circuit. On génère successivement toutes les combinaisons possibles des entrées que l'on injecte dans le circuit. A chaque présentation des entrées, on compare la valeur du résultat (après conversion) avec la valeur théorique. Il est possible de tester exhaustivement des circuits de taille raisonnable assez rapidement.

- Calcul des caractéristiques statistiques du temps de calcul de l'opérateur. Il est possible d'effectuer des mesures du temps de calcul en injectant des données en entrées. Ces mesures peuvent être effectuées pour des distributions exhaustives ou aléatoires.

### 4.2.3 Caractéristiques mesurées

Les caractéristiques obtenues lors d'une simulation sont :

– La distribution du temps de fin du calcul de l'opérateur. On obtient des tableaux de la forme :

| $t$ | $p(t)$ |
|---|---|
| 1 | 0.01 |
| 2 | 0.20 |
| 3 | 0.53 |
| 4 | 0.21 |
| ⋮ | ⋮ |
| n | 0.02 |

où $t$ est le temps de calcul et $p(t)$ est la proportion des simulations ayant fini leur calcul au temps $t$. Ici, on s'est placé dans le cadre d'un opérateur dont toutes les fonctions ont le même délai égal à 1. En pratique, les cellules peuvent avoir des délais quelconques, le tableau est alors donné avec un pas égal au plus petit commun multiple de tous les délais.

- Le temps de calcul moyen de l'opérateur :

$$\mathbb{E}(T) = \frac{\sum_{t=1}^{n} t \times p(t)}{\sum_{t=1}^{n} p(t)}$$

- L'écart-type du temps de calcul de l'opérateur :

$$\sigma(T) = \sqrt{\mathrm{Var}(T)}$$

où

$$\mathrm{Var}(T) = \frac{\sum_{t=1}^{n} (t - \mathbb{E}(T))^2 \times p(t)}{\sum_{t=1}^{n} p(t)}$$

est la variance de $T$.

- Le temps moyen de fin du calcul de certaines sorties de certaines cellules (liste de signaux à tracer). Nous verrons que dans le cas des additionneurs, les signaux à "tracer" sont les signaux correspondant aux bits de somme et aux bits de retenue.

Les mesures sont effectuées en calculant les intervalles de confiance à 5 % sur les résultats. C'est-à-dire que l'on donne une valeur telle que, avec la probabilité de 0.95, la valeur exacte $X$ est dans l'intervalle $]\tilde{X} - \epsilon, \tilde{X} + \epsilon[$ avec $\epsilon = \frac{2 \times \sqrt{\tilde{X}(1-\tilde{X})}}{\sqrt{n}}$ où $\tilde{X}$ est la valeur mesurée. En pratique, cela se résume à effectuer suffisamment de tests pour que deux fois la valeur de $\epsilon$ mesurée soit plus petite que 5 % de la valeur de la moyenne. Le programme détecte si il y a effectivement décroissance de $\epsilon$ lorsque $n$ augmente.

Les données en entrée peuvent être générées exhaustivement ou par des tirages aléatoires. Pour le moment, nous utilisons une distribution uniforme des entrées. Nous étudierons la validité de cette hypothèse d'uniformité des entrées dans les sections suivantes.

## 4.3 Additionneurs asynchrones

L'addition est une opération très utilisée non seulement en temps qu'opération arithmétique mais aussi comme élément de base dans d'autres opérateurs. On trouve des additionneurs dans les opérateurs de multiplication, de division et de racine carrée. L'addition est aussi une des briques de base pour l'évaluation des fonctions élémentaires (par exemple dans les algorithmes basés sur des additions et des décalages [Mul97]). L'addition est omniprésente dans les processeurs comme pour le calcul des adresses. Il est donc important de concevoir des additionneurs performants.

Très tôt dans l'histoire de l'informatique, les chercheurs ont essayé de mettre au point des algorithmes d'addition plus performants que l'algorithme que l'on apprend à l'école. En 1946, dans leur célèbre article "*Preliminary discussion of the logical design of an electronic computing instrument*" ([BGN46]), Burks, Goldstine et Von Neumann analysent le cas de l'additionneur séquentiel binaire. Ils ont montré en particulier que le temps moyen nécessaire pour effectuer la somme de deux nombres binaires de taille $n$ avec un additionneur séquentiel binaire est en $\mathcal{O}(\log_2 n)$ si on

suppose une distribution uniforme des entrées. Leur résultat est basé sur l'analyse de la longueur moyenne de la plus longue chaîne de propagation de la retenue, et cette longueur moyenne est inférieure ou égale à $\log_2 n$.

On constate ici l'importance de la notion de calcul en temps moyen dans les opérateurs asynchrones car on voit que dans le cas de l'addition, il est possible de concevoir simplement un opérateur très performant. La variabilité du temps de calcul dans l'addition est importante puisque dans le meilleur cas il n'y a aucune propagation de retenue et donc tous les bits de somme sont obtenus rapidement. Le pire cas introduit un temps de calcul important puisqu'il faut que le signal de retenue passe successivement dans les $n$ cellules d'addition. L'addition illustre parfaitement la notion de parallélisme dynamique dans les opérateurs asynchrones. Suivant les valeurs des données, certaines parties du calcul, qui *a priori* pouvaient être interdépendantes, vont pouvoir s'effectuer de façon totalement indépendantes et donc en parallèle.

Dans le cas synchrone, Brent ([Bre70]) et Winograd ([Win65]) ont montré que la complexité temporelle de l'addition binaire est $\mathcal{O}(\log_2 n)$, si on utilise un modèle de porte avec des entrances (*fan-in*) et des sortances (*fan-out*) bornées.

Dans la suite, nous allons nous intéresser aux additionneurs asynchrones suivants :

- l'additionneur séquentiel ou RCA (*ripple carry adder*)
- l'additionneur à retenue bondissante ou CSkA (*carry skip adder*)
- l'additionneur à sélection de la retenue ou CSeA (*carry select adder*)

On peut trouver dans [Kor93, Mul89, Omo94] la description complète de ces additionneurs dans le cas synchrone.

Les nombres sont représentés en base 2 en numération simple de position (l'écriture binaire usuelle). Nous allons étudier le cas des additionneurs de deux nombres de $n$ bits. Les nombres manipulés peuvent être des entiers ou bien des nombres réels codés en virgule fixe. La structure des additionneurs et les résultats sont complètement équivalents dans les deux cas. Cependant, afin de fixer les notations, et en particulier les indices relatifs aux poids forts et aux poids faibles, nous allons nous placer dans le cas de nombres entiers de $n$ bits. C'est-à-dire avec des nombres tels que :

$$a = \sum_{i=0}^{n-1} a_i 2^i$$

Les résultats obtenus ici peuvent être trivialement étendus au cas de la représentation en complément à deux.

Par défaut, nous utilisons des cellules *full-adder* dont les délais pour calculer le bit de somme et le bit de retenue sont égaux à 1 quelles que soient les valeurs des entrées. Nous effectuerons des mesures pour des délais différents et nous stipulerons alors clairement les valeurs des différents délais pour ces simulations.

Nous avons vu que dans les opérateurs asynchrones, il y a toujours propagation des signaux, qu'ils transportent une valeur égale à "1" ou à "0". Avec l'utilisation du codage 3 états vu en 4.1.1,

un signal dont la valeur n'est pas encore connue est dans l'état invalide. Il nous faut donc définir ce que l'on entend par propagation et génération de la retenue. Dans le cas synchrone, la propagation de la retenue signifie la propagation d'un signal électrique à "1". En asynchrone, les choses sont différentes. Les notions de propagation et de génération sont définies relativement à la fonction que réalise une cellule *full-adder*. Nous avons représenté dans le tableau suivant les différents cas possibles pour la valeur de la retenue sortante $c_{i+1}$ en fonction des valeurs des deux bits d'entrée $a_i, b_i$ et en fonction de la valeur de la retenue entrante $c_i$.

| $a_i$ | $b_i$ | $c_{i+1}$ |
|---|---|---|
| 0 | 0 | 0 (génération) |
| 0 | 1 | $c_i$ (propagation) |
| 1 | 0 | $c_i$ (propagation) |
| 1 | 1 | 1 (génération) |

Il y a donc *génération* de la retenue sortante lorsque à la seule vue des deux bits $a_i$ et $b_i$ on peut déterminer la valeur de cette retenue et *propagation* dans le cas contraire.

Les caractéristiques statistiques des additionneurs asynchrones qui vont être mesurées dépendent bien évidemment de la distribution des nombres en entrée. Dans la suite, nous allons supposer une distribution uniforme des bits dans les opérandes. C'est-à-dire que chaque bit à la probabilité $\frac{1}{2}$ d'être à "1" ou à "0". Il est bien clair que la distribution réelle des valeurs des entrées dépend du contexte d'utilisation de l'additionneur. Par exemple, dans le cas des additions utilisées pour le calcul des adresses dans un processeur, la distribution des entrées est très loin d'être uniforme (multiples des tailles des pages, longueur des boucles...). Par contre on vu au premier chapitre, lors de la modélisation probabiliste de l'arrondi exact des fonctions élémentaires, que les mantisses des nombres flottants peuvent être vues comme des chaînes aléatoires de "1" et "0" distribués avec équiprobabilité (cf les travaux de Feldstein et Goodman [FG76]). La distribution uniforme, même si elle n'est pas toujours représentative de la réalité, permet de comparer les différents additionneurs asynchrones entre eux. Elle permet aussi de simplifier les calculs de la modélisation mathématique. Nous utilisons la même distribution des entrées en simulation et dans la modélisation mathématique pour vérifier que ces deux approches représentent les mêmes choses. Nous verrons que la connaissance de certaines particularités d'une distribution réelle peuvent être utilisées pour optimiser certains additionneurs. Toutes nos études seront faites dans le cas où tous les bits des opérandes sont présents au début du calcul.

### 4.3.1 Additionneur séquentiel

L'additionneur séquentiel est l'additionneur le plus simple. La structure d'un additionneur séquentiel binaire est rappelée Figure 4.6. Nous avons vu que Burks, Goldstine et Von Neumann ont montré dans [BGN46] que le temps moyen de calcul de cet additionneur est en $\mathcal{O}(\log_2 n)$. Nous avons effectué différentes simulations sur cet additionneur afin de mesurer les autres caractéristiques : l'écart-type du temps de calcul, la distribution du temps de calcul et la distribution du temps de fin de calcul des bits de somme et de retenue. La correction de l'opérateur a été testée exhaustivement pour des tailles allant jusqu'à 12 bits.

#### Tests exhaustifs sur un additionneur séquentiel 8 bits

Nous avons effectué des tests exhaustifs sur un additionneur 8 bits pour obtenir l'ensemble de ses caractéristiques. La Figure 4.7 présente la distribution du temps de calcul de cet additionneur

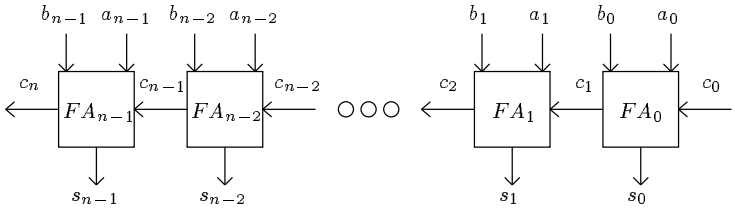



FIG. 4.6 – *Additionneur binaire séquentiel sur n bits.*

(rappel : tous les délais sont égaux à 1).

| $t$ | $p(t)$ |
|---|---|
| 1 | 0.00 |
| 2 | 0.02 |
| 3 | 0.31 |
| 4 | 0.36 |
| 5 | 0.19 |
| 6 | 0.08 |
| 7 | 0.03 |
| 8 | 0.01 |

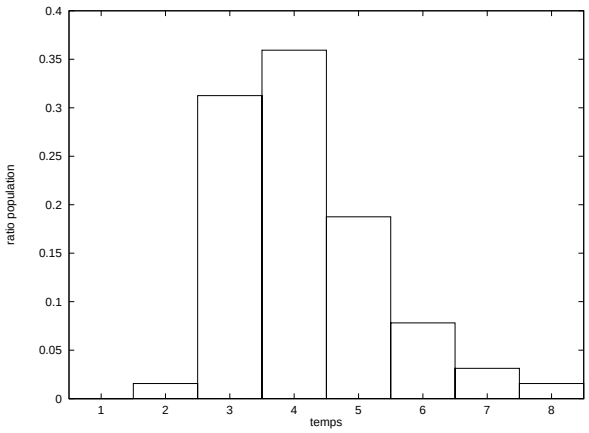



FIG. 4.7 – *Distribution du temps de calcul pour un additionneur séquentiel 8 bits (test exhaustif).*

Pour chaque pas de simulation $t$ (ici les entiers de 1 à 8 car les délais sont égaux à 1), on donne la proportion $p(t)$ de la population dont le calcul s'est terminé au temps $t$. C'est-à-dire le rapport :

$$\frac{\text{nombre de couples des entrées dont l'addition se termine au temps } t}{\text{nombre de couples total}}$$

dans ce cas, le nombre de couples total est $2^{8\times 2}$.

L'histogramme représente la distribution du temps de calcul. Les abscisses représentent le temps $t$ et les ordonnées les pourcentages de la population $p(t)$. La barre d'abscisse $t$ à donc une hauteur de $p(t)$.

Les valeurs mesurées sont (délais unitaires) :

- temps moyen : 4.16
- écart-type : 1.18

Les Figures 4.8 et 4.9 représentent respectivement la distribution du temps moyen d'établissement des bits de somme et celle des bits de retenue. Les histogrammes doivent être lus différemment car les abscisses représentent les numéros de cellules et les ordonnées le temps. Une barre dans l'un de ces deux histogrammes située à l'abscisse $i$ et de hauteur $k$ signifie que la $i^{\text{ème}}$ cellule (en partant des poids faibles) délivre son bit de retenue (ou de somme) en un temps moyen égal à $k$. Les tableaux correspondants donnent pour chaque cellule $i$ le temps moyen d'établissement $t(i)$ du bit de somme (ou de retenue) de cette cellule.

| $i$ | $t(i)$ |
|---|---|
| 1 | 1.00 |
| 2 | 2.00 |
| 3 | 2.50 |
| 4 | 2.75 |
| 5 | 2.88 |
| 6 | 2.94 |
| 7 | 2.97 |
| 8 | 2.98 |

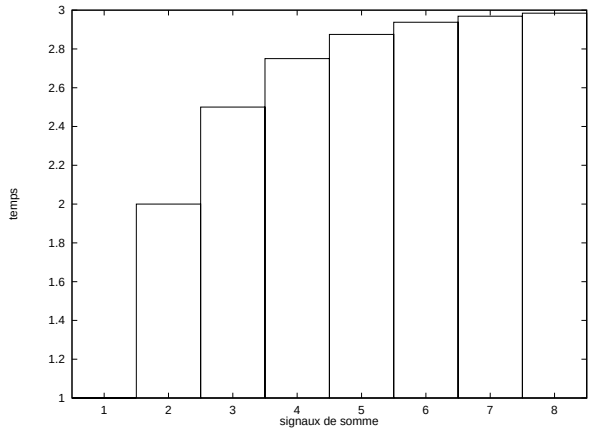



FIG. 4.8 – *Temps moyen d'établissement des bits de somme pour un additionneur séquentiel 8 bits (test exhaustif).*

| $i$ | $t(i)$ |
|---|---|
| 1 | 1.00 |
| 2 | 1.50 |
| 3 | 1.75 |
| 4 | 1.88 |
| 5 | 1.94 |
| 6 | 1.97 |
| 7 | 1.98 |
| 8 | 1.99 |

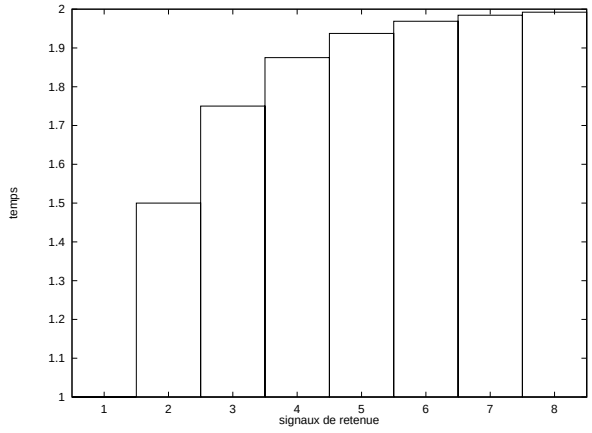



FIG. 4.9 – *Temps moyen d'établissement des bits de retenue pour un additionneur séquentiel 8 bits (test exhaustif).*

### Tests par tirages aléatoires pour un additionneur séquentiel 32 bits

Nous avons simulé et mesuré les performances d'un additionneur séquentiel 32 bits. Les mesures ont été faites sur une distribution uniforme des entrées. La Figure 4.10 présente la distribution du temps de calcul de cet additionneur. Les Figures 4.11 et 4.12 présentent le temps d'établissement moyen des bits de somme et de retenue. Les valeurs mesurées sont (délais unitaires) :

- temps moyen : 6.28
- écart-type : 1.66

Le parallélisme dynamique est parfaitement illustré sur les Figures 4.11 et 4.12. Par exemple, tous les bits de somme sont calculés au plus après 3 unités de temps en moyenne (sauf pour les tout premiers bits). On voit bien sur ces figures que les calculs s'effectuent indépendamment les uns des autres alors qu'ils étaient *a priori* tous interdépendants. Ces valeurs 2 et 3 pour le temps moyen d'établissement des bits de retenue et de somme proviennent simplement de la longueur moyenne

| $t$ | $p(t)$ |
|---|---|
| 1 | 0.00 |
| 2 | 0.00 |
| 3 | 0.00 |
| 4 | 0.09 |
| 5 | 0.26 |
| 6 | 0.27 |
| 7 | 0.17 |
| 8 | 0.09 |
| 9 | 0.05 |
| 10 | 0.02 |
| 11 | 0.01 |

| $t$ | $p(t)$ |
|---|---|
| 12 | 0.01 |
| 13 | 0.00 |
| 14 | 0.00 |
| 15 | 0.00 |
| 16 | 0.00 |
| 17 | 0.00 |
| 18 | 0.00 |
| 19 | 0.00 |
| 20 | 0.00 |
| 21 | 0.00 |
| 22 | 0.00 |

| $t$ | $p(t)$ |
|---|---|
| 23 | 0.00 |
| 24 | 0.00 |
| 25 | 0.00 |
| 26 | 0.00 |
| 27 | 0.00 |
| 28 | 0.00 |
| 29 | 0.00 |
| 30 | 0.00 |
| 31 | 0.00 |
| 32 | 0.00 |

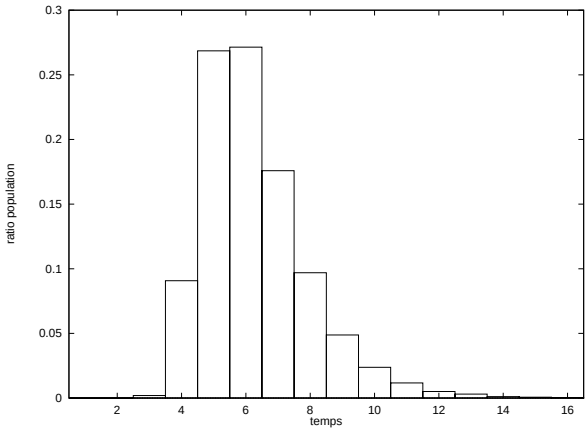



FIG. 4.10 – *Distribution du temps de calcul pour un additionneur séquentiel 32 bits (tests aléatoires).*

| $i$ | $t(i)$ |
|---|---|
| 1 | 1.00 |
| 2 | 2.00 |
| 3 | 2.49 |
| 4 | 2.75 |
| 5 | 2.87 |
| 6 | 2.93 |
| 7 | 2.96 |
| 8 | 2.97 |
| 9 | 2.98 |
| 10 | 2.99 |
| 11 | 2.99 |

| $i$ | $t(i)$ |
|---|---|
| 12 | 2.99 |
| 13 | 2.99 |
| 14 | 2.99 |
| 15 | 2.99 |
| 16 | 2.99 |
| 17 | 2.99 |
| 18 | 2.99 |
| 19 | 2.99 |
| 20 | 3.00 |
| 21 | 2.99 |
| 22 | 3.00 |

| $i$ | $t(i)$ |
|---|---|
| 23 | 3.00 |
| 24 | 3.00 |
| 25 | 2.99 |
| 26 | 2.99 |
| 27 | 2.99 |
| 28 | 3.00 |
| 29 | 3.02 |
| 30 | 3.01 |
| 31 | 3.00 |
| 32 | 3.00 |

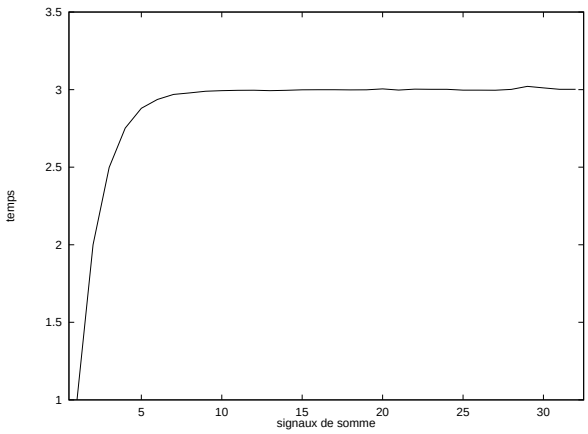



FIG. 4.11 – *Temps moyen d'établissement des bits de somme pour un additionneur séquentiel 32 bits (tests aléatoires).*

des chaînes de retenue (attention, il ne s'agit pas de la longueur moyenne de la plus longue chaîne de retenue qui est $\log_2 n$). En effet, avec une distribution uniforme des entrées, la probabilité qu'il y ait propagation sur $p$ cellules *full-adder* consécutives est $2^{-p}$. La longueur moyenne de ces chaînes est donc :

$$\sum_{i=1}^{n} i \times 2^{-i} = 2 - \frac{1}{2^{n-1}} - \frac{n}{2^n}$$

La longueur moyenne des chaînes de propagation de retenue est donc très proche de 2 car rapidement les deux termes négatifs sont négligeables lorsque $n$ augmente (par exemple, pour $n = 8$ on a 1.961). Il est donc tout à fait logique que les bits de retenues soient tous disponibles (sauf les tous premiers qui sont obtenus plus rapidement) après seulement 2 unités de temps en moyenne. Les bits de somme nécessitant dans 50 % des cas la valeur de la retenue entrante, ils sont tous disponibles (sauf les tout premiers qui sont obtenus plus rapidement) après 3 unités de temps en moyenne.

| $i$ | $t(i)$ |
|---|---|
| 1 | 1.00 |
| 2 | 1.49 |
| 3 | 1.75 |
| 4 | 1.87 |
| 5 | 1.93 |
| 6 | 1.96 |
| 7 | 1.97 |
| 8 | 1.98 |
| 9 | 1.99 |
| 10 | 1.99 |
| 11 | 1.99 |

| $i$ | $t(i)$ |
|---|---|
| 12 | 1.99 |
| 13 | 1.99 |
| 14 | 1.99 |
| 15 | 1.99 |
| 16 | 1.99 |
| 17 | 1.99 |
| 18 | 1.99 |
| 19 | 2.00 |
| 20 | 1.99 |
| 21 | 2.00 |
| 22 | 2.00 |

| $i$ | $t(i)$ |
|---|---|
| 23 | 2.00 |
| 24 | 1.99 |
| 25 | 1.99 |
| 26 | 1.99 |
| 27 | 2.00 |
| 28 | 2.02 |
| 29 | 2.01 |
| 30 | 2.00 |
| 31 | 2.00 |
| 32 | 1.99 |

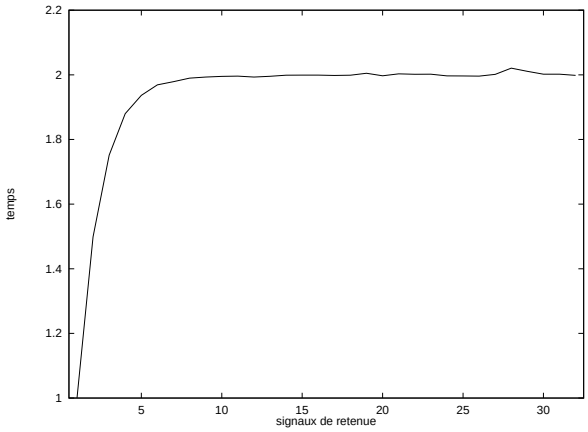



FIG. 4.12 – *Temps moyen d'établissement des bits de retenue pour un additionneur séquentiel 32 bits (tests aléatoires).*

### Temps moyen et écart-type pour différentes tailles de l'additionneur séquentiel

Le Tableau 4.1 donne la moyenne et l'écart-type du temps de calcul pour quelques tailles d'additionneurs séquentiels. La Figure 4.13 représente le temps moyen de calcul (courbe du milieu) en fonction de la taille (comprises entre 8 et 128). Les deux autres courbes de part et d'autre du temps moyen représentent le temps moyen plus ou moins une fois la valeur de l'écart-type.

| taille | temps moyen | écart-type |
|---|---|---|
| 8 | 4.16 | 1.17 |
| 16 | 5.24 | 1.49 |
| 24 | 5.85 | 1.60 |
| 32 | 6.28 | 1.66 |
| 53 | 7.04 | 1.74 |
| 64 | 7.31 | 1.75 |
| 80 | 7.63 | 1.78 |
| 128 | 8.32 | 1.80 |

TAB. 4.1 – *Valeur moyenne et écart-type du temps de calcul de quelques additionneurs séquentiels.*

On constate que le temps moyen mesuré est fait un peu supérieur à $\log_2 n$. Ceci s'explique car le temps moyen de calcul est le temps correspondant à la la longueur moyenne de la plus grande chaîne de propagation de la retenue. Cette est longueur est inférieure à $\log_2 n$ ([BGN46]). En fait, on a donc bien un temps moyen de calcul égal à la longueur moyenne de la plus grand chaîne de propagation de la retenue plus une constante (le nombre de portes traversées est donc inférieur à $\log_2 n + 1$).

### Influence des délais des portes

Nous avons effectué des tests en faisant varier les délais des fonctions qui délivrent les bits de somme et de retenue. Le Tableau 4.2 donne la valeur moyenne et l'écart-type du temps de calcul

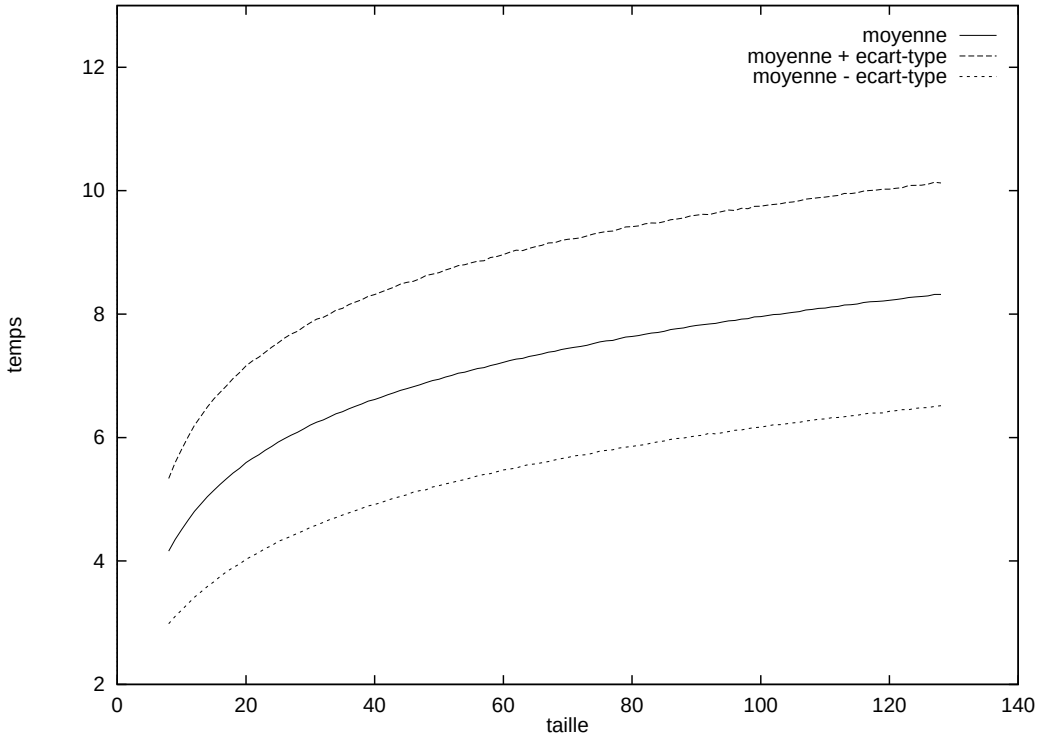



FIG. 4.13 – *Valeur moyenne et écart-type du temps de calcul des additionneurs séquentiels de taille comprises entre 8 et 128.*

d'un additionneur séquentiel de 128 bits dont on fait varier les délais des sorties (somme et retenue) des cellules *full-adder*, la Figure 4.14 est l'illustration graphique de ce tableau.

| $\frac{\text{délai bit de somme}}{\text{délai bit de retenue}}$ | temps moyen | écart-type |
|---|---|---|
| $\frac{10}{1}$ | 1.73 | 0.18 |
| $\frac{2}{1}$ | 4.66 | 0.90 |
| $\frac{1}{1}$ | 8.32 | 1.80 |
| $\frac{1}{2}$ | 7.82 | 1.80 |
| $\frac{1}{10}$ | 7.43 | 1.80 |

TAB. 4.2 – *Influence des délais des sorties des cellules FA sur la moyenne et l'écart-type du temps de calcul pour un additionneur séquentiel 128 bits.*

Bien évidemment, il était prévisible que l'influence du délai des bits de retenue soit beaucoup plus grande que celle du délai des bits de somme. Nous verrons dans la suite que ces variations des délais peuvent nous être utiles pour essayer de voir quels sont les éléments qu'il faut optimiser en priorité dans un opérateur arithmétique asynchrone plus complexe.

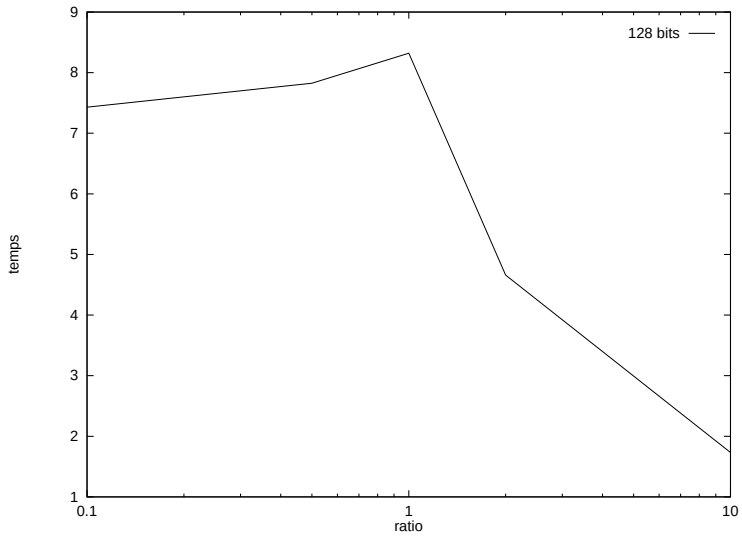



FIG. 4.14 – *Influence des délais des sorties des cellules FA sur la moyenne du temps de calcul pour un additionneur séquentiel 128 bits.*

### 4.3.2 Additionneur à retenue bondissante

L'additionneur à retenue bondissante de deux nombres de $n$ bits (Figure 4.15) est basé sur un découpage en $b$ blocs de $p$ bits avec $n = b \times p$ (voir [Mul89] pour le cas synchrone). L'idée de base dans cet additionneur est de propager directement la retenue entrante dans un bloc à la sortie du bloc si il y a effectivement propagation sur le bloc. Chaque bloc est composé d'un petit additionneur séquentiel de $p$ bits et d'une logique qui permet de détecter si il y a propagation sur chacune des cellules *full-adder* du bloc. La détection de propagation sur une cellule FA se fait en utilisant $a_i \oplus b_i$ ($\oplus$ est le ou exclusif) qui est égal à 1 si il y a propagation (en fait, $a_i \oplus b_i$ est déjà calculé à l'intérieur d'une cellule FA). La logique de détection de propagation sur le bloc est un arbre de cellules ET de chacun des $a_i \oplus b_i$ du bloc (comme $p$ sera en général petit, la logique de détection de propagation sur le bloc sera très rapide).

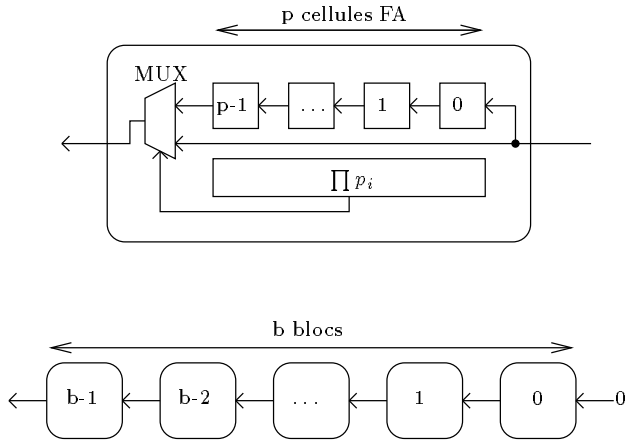



FIG. 4.15 – *Additionneur à retenue bondissante avec b blocs de p bits.*

Il faut trouver un compromis entre la valeur de $b$ et la valeur de $p$. Plus $p$ est grand plus la probabilité qu'il y ait effectivement propagation sur le bloc est petite ($2^{-p}$). Mais plus $p$ est grand plus les bonds de la retenue sont importants, et donc plus le calcul est rapide. En synchrone, on montre facilement que la taille optimale des blocs est $p = \sqrt{n}$ pour un découpage régulier ([LB61]), et qu'il existe des techniques qui donnent de bons résultats pour les découpages en blocs de tailles différentes ([GHM87]).

La taille optimale des blocs d'un additionneur à retenue bondissante asynchrone régulier dépend des vitesses respectives des cellules *full-adder* et de la logique qui permet d'effectuer les sauts (que pour simplifier nous avons noté MUX dans les tableaux).

La correction de notre additionneur à retenue bondissante à été testée exhaustivement pour toutes les combinaisons possibles pour $b$ et $p$ avec $b \times p \leq 12$ bits.

### Tests exhaustifs

Nous avons effectué des tests exhaustifs sur un additionneur à retenue bondissante de taille 8 avec différents découpages pour obtenir l'ensemble de ses caractéristiques (moyenne, écart type et distribution du temps de calcul). Les résultats correspondants sont donnés dans le Tableau 4.3.

`délai FA = 1, délai MUX = 1`

| $p$ | $b$ | moyenne | écart type |
|---|---|---|---|
| 1 | 8 | 5.01 | 1.08 |
| 2 | 4 | 4.53 | 0.71 |
| 4 | 2 | 4.65 | 1.21 |
| 8 | 1 | 4.15 | 1.17 |

`délai FA = 1, délai MUX = 0.5`

| $p$ | $b$ | moyenne | écart type |
|---|---|---|---|
| 1 | 8 | 3.49 | 0.56 |
| 2 | 4 | 3.91 | 0.60 |
| 4 | 2 | 4.33 | 1.09 |
| 8 | 1 | 4.16 | 1.18 |

TAB. 4.3 – *Additionneurs à retenue bondissante de taille 8.*

On remarque que lorsque la logique de détection de propagation sur un bloc (délai MUX) est beaucoup plus rapide qu'une cellule FA (tableau de droite), la taille optimale des blocs est 1. Ce qui est logique étant donné que cela équivaut à avoir réalisé une nouvelle cellule FA dont la sortie de la retenue à été optimisée en vitesse.

A délais égaux (tableau de gauche), le choix de la taille des blocs pour avoir un opérateur qui calcule le plus vite possible est plus difficile à faire. En effet, la plus petite valeur moyenne du temps de calcul est obtenu pour des blocs de taille 8. Cependant ce choix n'est pas le meilleur que l'on puisse faire car il ne tient pas compte de la valeur de l'écart type. En effet, on constate que pour des blocs de taille 2 la distribution est plus "resserrée" que pour des blocs de taille 8 (4.53+0.71=5.24 contre 4.15+1.17=5.32). C'est ici que l'on remarque que la caractérisation complète d'un opérateur arithmétique asynchrone demande de connaître non seulement la valeur moyenne du temps de calcul mais aussi sa distribution. C'est un des points que nous avons essayé de mettre en avant dans ce travail (cf [MTV97]). La distribution du temps de calcul d'un opérateur nous semble être beaucoup plus riche en information que la seule donnée de la valeur moyenne du temps de calcul.

### Tests par tirages aléatoires

Nous avons effectué des tests aléatoires pour un additionneur à retenue bondissante de taille 128 pour différents découpages et pour différents délais des portes (FA et MUX). Les résultats correspondants sont présentés dans le Tableau 4.4 et illustrés Figure 4.16. A titre de comparaison,

| | | FA = 1, MUX = 0.1 | | FA = 1, MUX = 0.5 | | FA = 1, MUX = 1 | |
|---|---|---|---|---|---|---|---|
| $p$ | $b$ | moyenne | écart type | moyenne | écart type | moyenne | écart type |
| 1 | 128 | 2.73 | 0.18 | 5.65 | 0.90 | 9.31 | 1.80 |
| 2 | 64 | 4.24 | 0.09 | 5.24 | 0.46 | 6.70 | 0.93 |
| 3 | 43 | 5.87 | 0.43 | 6.38 | 0.48 | 7.06 | 0.66 |
| 4 | 32 | 6.92 | 0.80 | 7.35 | 0.81 | 7.89 | 0.84 |
| 5 | 26 | 7.59 | 1.10 | 7.99 | 1.10 | 8.50 | 1.10 |
| 6 | 22 | 8.00 | 1.31 | 8.40 | 1.31 | 8.90 | 1.32 |
| 7 | 19 | 8.23 | 1.47 | 8.62 | 1.49 | 9.10 | 1.52 |
| 8 | 16 | 8.30 | 1.61 | 8.66 | 1.64 | 9.11 | 1.69 |
| 16 | 8 | 8.37 | 1.80 | 8.57 | 1.84 | 8.82 | 1.91 |
| 32 | 4 | 8.34 | 1.80 | 8.43 | 1.82 | 8.55 | 1.86 |
| 64 | 2 | 8.32 | 1.80 | 8.36 | 1.80 | 8.40 | 1.82 |
| 128 | 1 | 8.32 | 1.80 | 8.32 | 1.80 | 8.32 | 1.80 |

TAB. 4.4 – *Additionneurs à retenue bondissante de taille 128.*

nous rappelons que la valeur moyenne et l'écart type du temps de calcul d'un additionneur séquentiel de taille 128 sont respectivement 8.32 et 1.80.

Les courbes Figure 4.17 représentent le temps moyen d'établissement des bits de retenue pour différentes tailles de blocs dans un additionneur à retenue bondissante de nombres de 128 bits (tous les délais des portes sont égaux à 1). Ces courbes illustrent bien le temps gain de temps obtenu par le saut direct d'un bloc où il y a propagation.

Le Tableau 4.18 présente la valeur moyenne et l'écart type du temps de calcul de différents additionneurs à retenue bondissante pour différents découpages. La taille optimale des blocs est 2 contre $\sqrt{n}$ dans le cas synchrone.

Nous présentons Figure 4.19 une comparaison entre la distribution du temps de calcul d'additionneurs à retenue bondissante et celle des additionneurs séquentiels de tailles 8, 16, 32, 64 et 128 (tous les délais sont égaux à 1).

Au vu de tous les résultats présentés dans cette section, on peut dire que l'additionneur à retenue bondissante est un bon candidat pour l'implantation d'un additionneur asynchrone performant. En effet, son temps moyen de calcul et sa distribution sont meilleurs que ceux d'un additionneur séquentiel. De plus, nous verrons dans la prochaine section, qu'il présente quasiment les mêmes caractéristiques temporelles que des additionneurs beaucoup plus volumineux.

Une amélioration sensible de l'additionneur à retenue bondissante serait de prendre en compte plusieurs niveaux de saut de blocs. C'est à dire d'utiliser une architecture qui détecte la propagation sur plusieurs blocs consécutifs et de sauter tous ces blocs si il y a lieu. La logique utilisée pour la détection de la propagation sur plusieurs blocs consécutifs est particulièrement simple puisqu'elle consiste en un simple "ET" des informations de propagation sur les blocs. Toutefois, le nombre de niveaux à utiliser ne doit pas être trop important pour différentes raisons. Premièrement, la longueur moyenne de la plus longue chaîne de la propagation de la retenue étant inférieure à $\log_2 n$, le nombre de blocs à sauter d'un coup est de l'ordre de $\frac{\log_2 n}{p}$. Deuxièmement, la logique asynchrone qui permet de choisir entre la valeur de la retenue entrante et celles des retenues entrantes des blocs précédents (lors des sauts de plusieurs blocs consécutifs) devient de plus en plus complexe lorsque le nombre de niveaux augmente. Pratiquement, nous pensons que le "bon" nombre de niveaux est 2 ou 3 pour des additionneurs de tailles "raisonnables" (inférieures à 64).

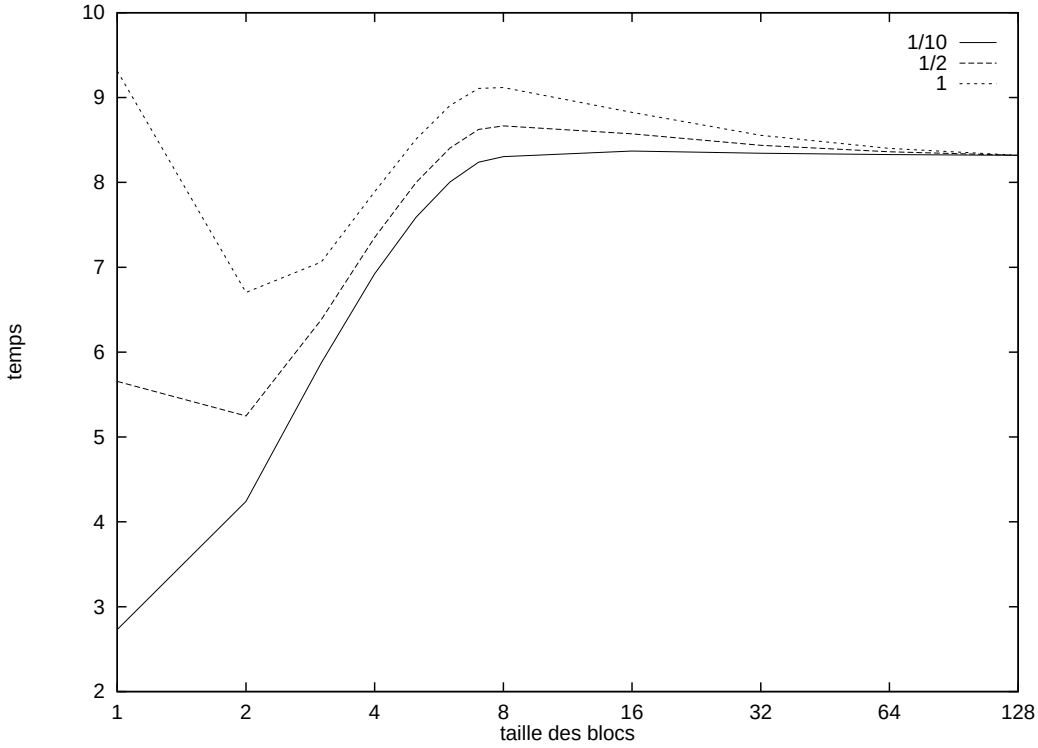



FIG. 4.16 – *Temps moyen de calcul pour différents délais des portes dans des additionneurs à retenue bondissante de taille 128 pour différents délais relatifs des portes.*

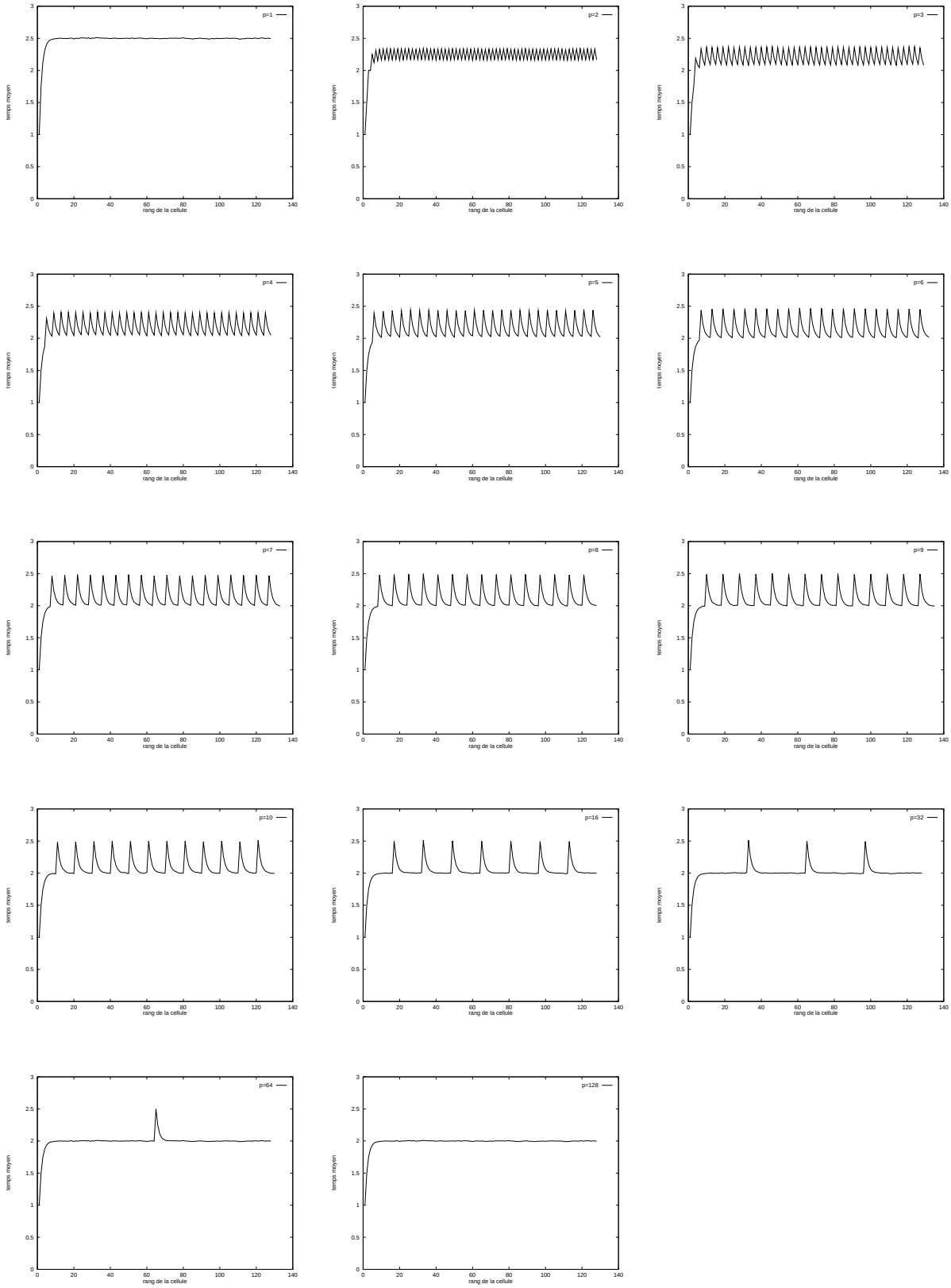



FIG. 4.17 – *Temps moyen d'établissement des bits de retenue pour différents additionneurs à retenue bondissante de taille 128.*

| $p$ | $b$ | moyenne | écart type |
|---|---|---|---|
| 1 | 8 | 5.01 | 1.08 |
| 2 | 4 | 4.53 | 0.71 |
| 4 | 2 | 4.65 | 1.22 |
| 8 | 1 | 4.16 | 1.17 |
| 1 | 16 | 6.19 | 1.48 |
| 2 | 8 | 5.16 | 0.79 |
| 4 | 4 | 5.86 | 1.22 |
| 8 | 2 | 5.66 | 1.62 |
| 16 | 1 | 5.24 | 1.49 |
| 1 | 32 | 7.26 | 1.66 |
| 2 | 16 | 5.68 | 0.85 |
| 4 | 8 | 6.70 | 1.06 |
| 8 | 4 | 6.91 | 1.76 |
| 16 | 2 | 6.53 | 1.74 |
| 32 | 1 | 6.28 | 1.66 |
| 1 | 64 | 8.30 | 1.75 |
| 2 | 32 | 6.19 | 0.91 |
| 4 | 16 | 7.34 | 0.95 |
| 8 | 8 | 8.06 | 1.75 |
| 16 | 4 | 7.71 | 1.85 |
| 32 | 2 | 7.45 | 1.79 |
| 64 | 1 | 7.31 | 1.75 |
| 1 | 128 | 9.31 | 1.80 |
| 2 | 64 | 6.70 | 0.93 |
| 4 | 32 | 7.89 | 0.84 |
| 8 | 16 | 9.11 | 1.69 |
| 16 | 8 | 8.82 | 1.91 |
| 32 | 4 | 8.55 | 1.86 |
| 64 | 2 | 8.40 | 1.82 |
| 128 | 1 | 8.32 | 1.80 |

FIG. 4.18 – *Valeur moyenne et écart type du temps de calcul de quelques additionneurs à retenue bondissante (délais égaux à 1).*

Nous travaillons actuellement en collaboration avec Marc Renaudin du CNET à la conception d'un additionneur à retenue bondissante à plusieurs niveaux.

### Estimation du temps moyen de calcul d'un additionneur à retenue bondissante

Nous donnons ici une estimation du temps moyen de calcul d'un additionneur à retenue bondissante. Les notations sont : $n$ est le nombre de chiffres des opérandes et $p$ est la taille des blocs. On fixe le délai d'une cellule FA à 1. Le temps moyen de calcul de cet additionneur est :

$$T(p) = k_1 p + k_2 \frac{\log_2 n}{p}$$

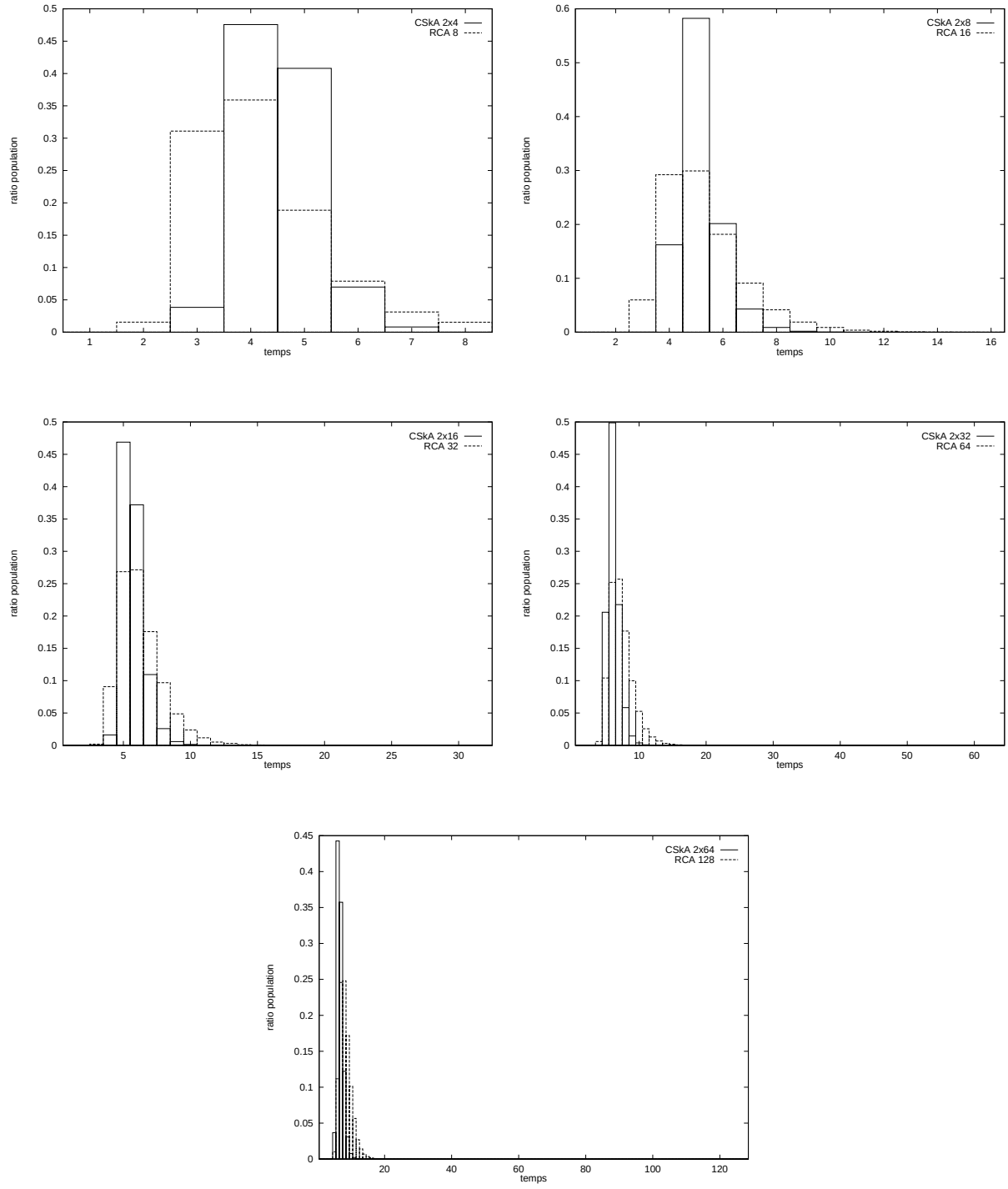



FIG. 4.19 – *Comparaison de la distribution du temps de calcul d'additionneurs à retenue bondissante (CSkA) et d'additionneurs séquentiels (RCA).*

où $k_2$ est le délai de la cellule de décision des sauts relativement à celui d'une cellule FA. Le terme $k_2\frac{\log_2 n}{p}$ est donc le temps nécessaire pour sauter les blocs qui sont dans la plus longue chaîne de propagation de la retenue en moyenne. Le terme $k_1 p$ est la conséquence du découpage d'une chaîne de taille $\log_2 n$ par des blocs de taille $p$. En effet, le nombre de blocs sautés en moyenne n'est probablement pas un multiple de $p$ et donc pour majorer le temps moyen nous supposons qu'il y a un bloc dans lequel presque toutes les cellules FA propagent la retenue d'où le coefficient $k_1$ qui est un peu inférieur à 1.

Le minimum du temps moyen de calcul est obtenu pour $p_{min} = \sqrt{\frac{k_2}{k_1}\log_2 n}$, et on a donc :

$$T(p_{min}) = 2\sqrt{k_1 k_2 \log_2 n}$$

### 4.3.3 Additionneur à sélection de retenue

Nous présentons dans cette section un additionneur à sélection de retenue (CSeA pour *carry select adder*) asynchrone. Cet additionneur à un temps moyen de calcul en $O(\sqrt{\log_2 n})$ tout comme l'additionneur à retenue bondissante mais avec une surface plus importante. La Figure 4.20 présente l'architecture de cet additionneur. L'idée de base est d'utiliser des blocs avec deux additionneurs séquentiels de $p$ bits pour anticiper les valeurs des retenues entrantes. L'un des deux additionneurs séquentiels calculant avec une retenue entrante égale à 0 et l'autre avec une retenue entrante égale à 1. Il suffit ensuite d'un simple système de multiplexeurs pour sélectionner les sorties (bits de somme et la retenue) qui correspondent à la bonne retenue entrante. Nous avons ajouté un petit perfectionnement rendu possible par l'utilisation de la logique asynchrone. Si les retenues sortantes des deux additionneurs séquentiels sont différentes, c'est qu'il y a génération d'une retenue dans le bloc (génération à 1 ou à 0) et la valeur générée est donc transmise au bloc suivant dès que possible (portes XOR et MUX).

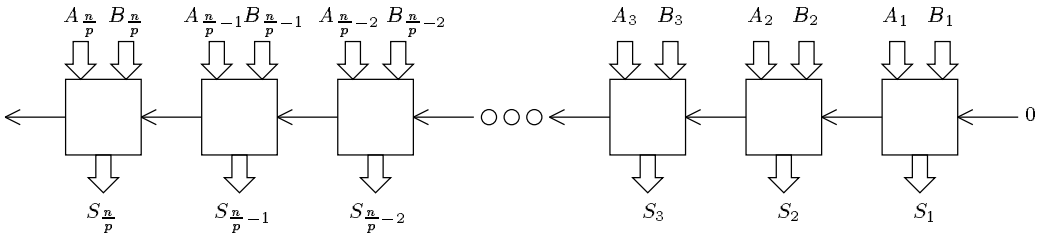



FIG. 4.20 – *L'architecture globale de l'additionneur à sélection de retenue.*

En appliquant récursivement la stratégie de découpage en blocs en obtient en synchrone un temps de calcul de $O(\log_2 n)$ (avec toutefois des portes ayant une sortance (*fan-out*) non bornée). En utilisant une technique similaire à celle présentée dans la section précédente pour l'additionneur à retenue bondissante on montre que le temps moyen de calcul d'un additionneur à sélection de retenue est en $\mathcal{O}(\sqrt{\log_2 n})$ (l'ensemble des résultats relatifs à cet additionneur ont été publiés dans [MTV97]).

#### Tests aléatoires

Nous présentons Figure 4.22 une comparaison entre le temps moyen de calcul d'additionneurs à sélection de retenue et celui des additionneurs séquentiels de même taille. Tous les délais sont

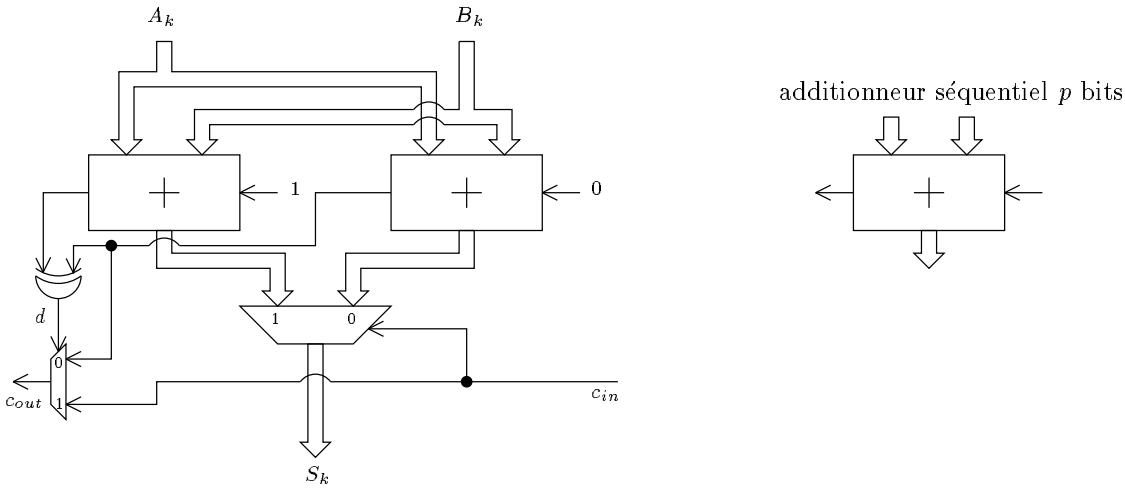



FIG. 4.21 – *L'architecture d'un bloc de l'additionneur à sélection de retenue.*

égaux à 1.

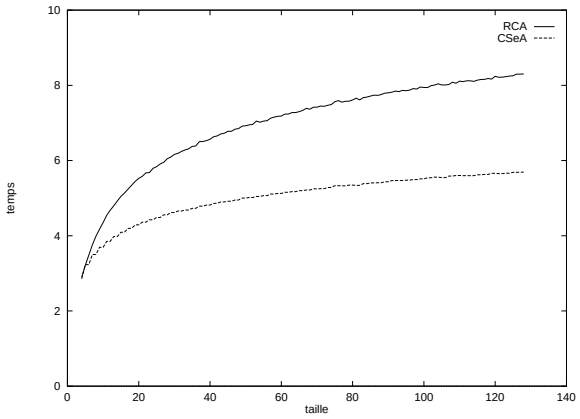
Le Tableau 4.5 donne le temps moyen de calcul pour différents additionneurs à sélection de retenue en comparant avec le temps moyen de calcul des additionneurs séquentiels de même taille (tous les délais sont égaux à 1).

| tailles | CSeA | | | | RCA |
|---|---|---|---|---|---|
| | *taille des blocs* | | | | |
| | 2 | 3 | 4 | 5 | |
| 16 | 4.09 | 4.35 | 4.48 | 4.90 | 5.24 |
| 32 | 4.66 | 4.80 | 5.10 | 5.44 | 6.28 |
| 64 | 5.17 | 5.19 | 5.60 | 5.97 | 7.31 |
| 128 | 5.69 | 5.49 | 5.95 | 6.48 | 8.32 |

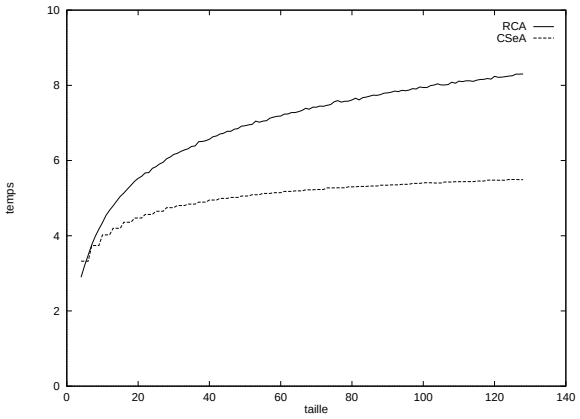
TAB. 4.5 – *Temps moyen de calcul de quelques additionneurs à sélection de retenue.*

La Figure 4.23 présente une comparaison entre la distribution du temps de calcul d'un additionneur à sélection de retenue de taille 128 et d'un additionneur séquentiel de même taille pour différents découpages.

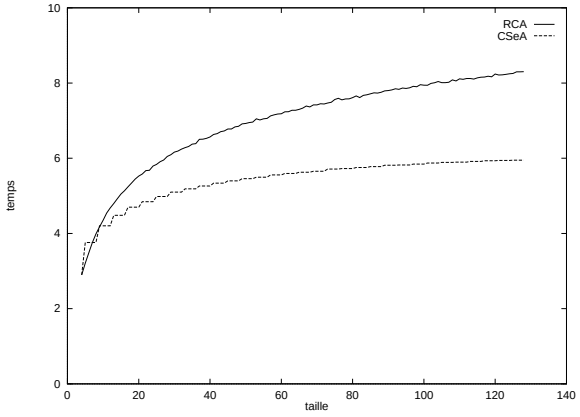
Nous avons effectué des comparaisons entre l'additionneur à sélection de retenue et l'additionneur à retenue bondissante. Les caractéristiques temporelles (valeur moyenne, écart type et distribution du temps de calcul) sont équivalentes avec des délais unitaires pour toutes les portes. Il nous faut attendre d'avoir plus avancé la réalisation pratique de ces additionneurs pour effectuer des comparaisons plus précises au niveau du comportement temporel. Toutefois, la taille d'un additionneur à sélection de retenue est environ deux fois plus grande que celle d'un additionneur à retenue bondissante. Il est donc tout à fait raisonnable de penser que l'additionneur à sélection de retenue sera moins performant qu'un additionneur à retenue bondissante à plusieurs niveaux.



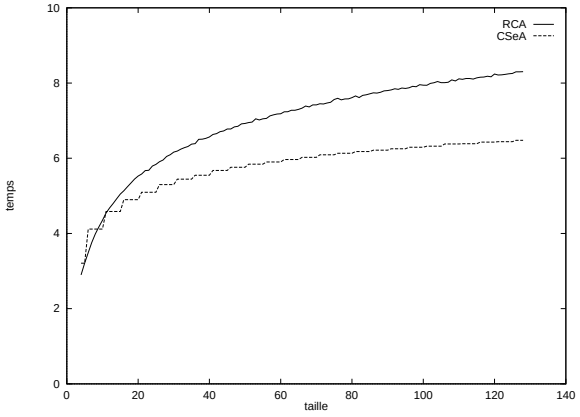

Taille de bloc = 2



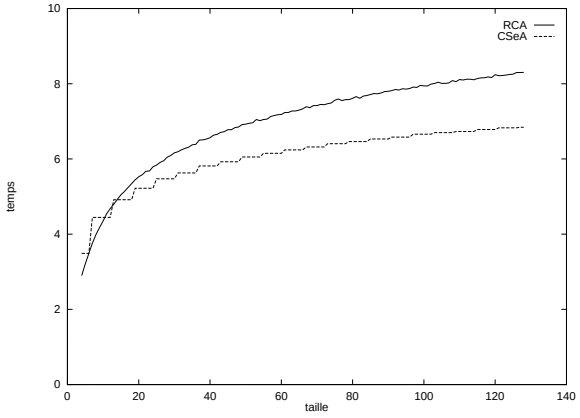

Taille de bloc = 3



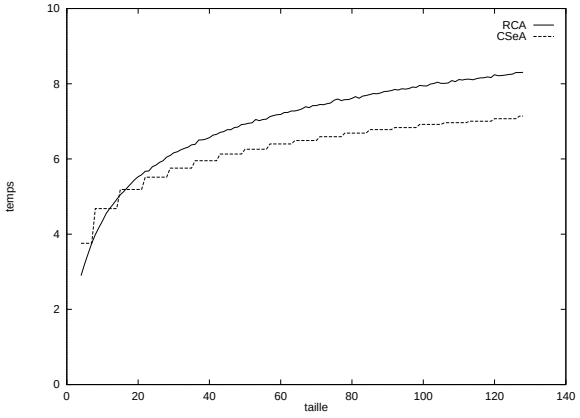

Taille de bloc = 4




Taille de bloc = 5




Taille de bloc = 6




Taille de bloc = 7

FIG. 4.22 – *Comparaison entre le temps de calcul moyen d'additionneurs à sélection de retenue et des additionneurs séquentiels de même taille.*

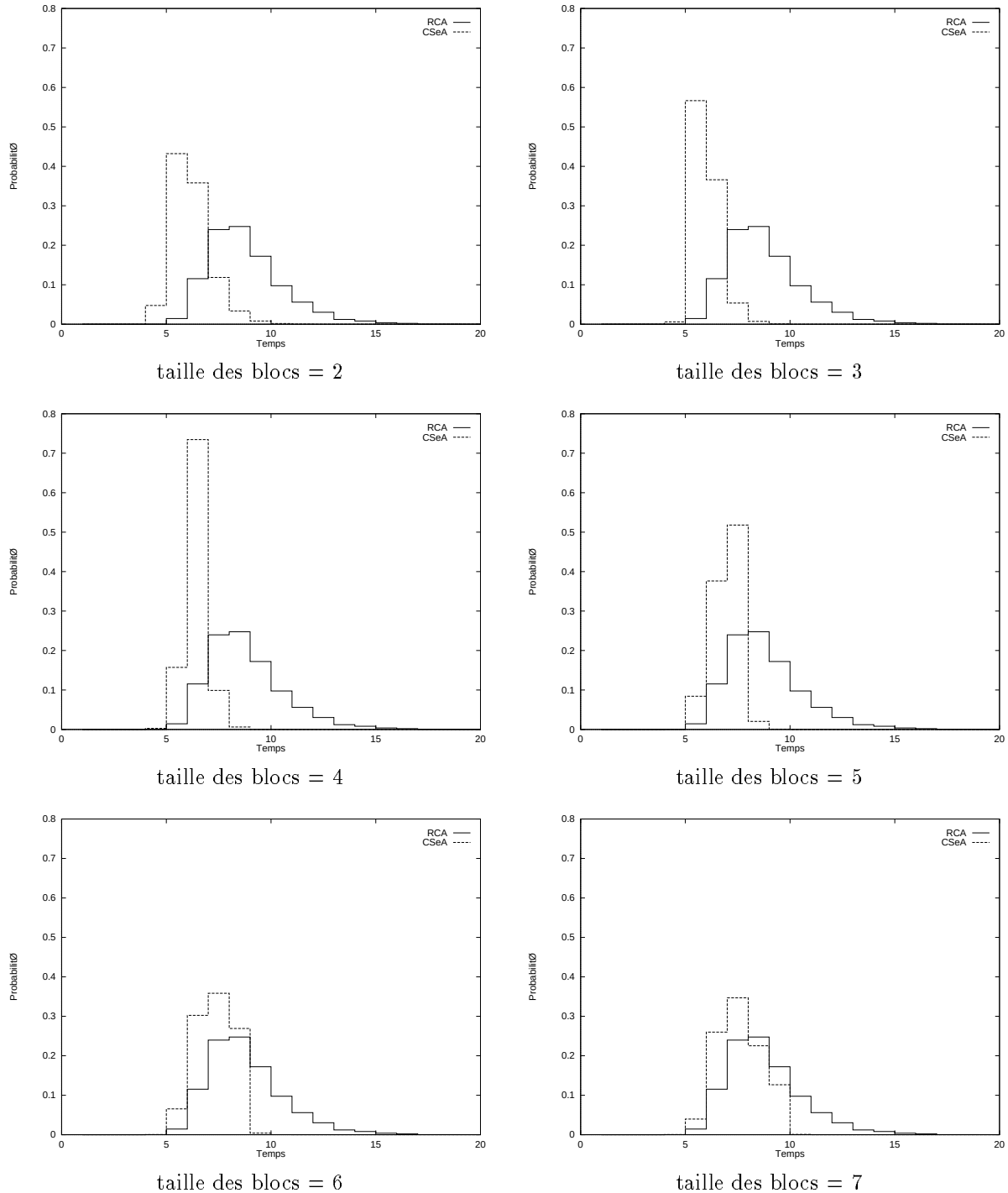



FIG. 4.23 – *Distribution du temps de calcul d'un additionneur à sélection de retenue de taille 128 pour différents découpages avec comparaison de l'additionneur séquentiel de même taille.*

## 4.4 Modélisation probabiliste des additionneurs asynchrones

### 4.4.1 Introduction

Nous présentons dans cette section les premiers résultats que nous avons obtenus sur la modélisation probabiliste des additionneurs asynchrones. Nous nous sommes intéressé au cas des additionneurs réguliers (modèle pour les additionneurs séquentiels, à retenue bondissante, à sélection de retenue). On trouve dans [GO96] une étude des additionneurs à évaluation anticipée de la retenue (*carry look ahead*).

Nous avons commencé notre étude par l'utilisation de fonctions génératrices de probabilité pour essayer d'obtenir les expressions des différentes caractéristiques des additionneurs asynchrones (valeur moyenne, écart type et distribution du temps de calcul). Nous verrons que cette modélisation ne permet pas de trouver (pour le moment) une expression simple de ces caractéristiques. Cette modélisation nous permet toutefois de quantifier numériquement ces caractéristiques.

Pour obtenir numériquement les caractéristiques des additionneurs asynchrones de façon plus rapide nous avons abordé une deuxième modélisation basée sur les chaînes de Markov.

Dans cette section, nous notons $\mathbb{P}$ la probabilité, $\mathbb{E}$ l'espérance et Var la variance.

### 4.4.2 Modèle de base

Les opérateurs asynchrones sont habituellement modélisés par des graphes de cellules (ou tâches) communiquantes. Les sommets du graphe sont les éléments de calcul (portes logiques de base ou complexes, éléments de mémorisation...). Dans le cas des additionneurs séquentiels binaires ce sont les cellules *full-adder*. Les arcs du graphe sont les signaux que se transmettent les cellules adjacentes. Les signaux peuvent être vus comme des couples $(v, d)$ où $v$ est la valeur numérique ou logique transmise et où $d$ est la date à partir de laquelle cette valeur est valide. Cette modélisation des signaux est due à la nature même des techniques de communication dans les circuits asynchrones. Par exemple, dans le codage *double rail* les deux bits permettent de coder les deux valeurs valides 0 et 1, mais aussi un état particulier qui indique que le signal n'est pas encore valide.

Le fonctionnement d'une cellule est relativement simple. Chaque sortie d'une cellule délivre au plus tôt la valeur de la fonction correspondante pour les valeurs présentes sur les entrées de la cellule. Dès qu'une configuration des valeurs des entrées permet de déduire la valeur de la sortie, le signal de sortie est validé avec une date égale à la date de la dernière entrée arrivée plus le délai correspondant à la fonction (et aussi aux valeurs de entrées).

Certaines cellules vont donc pouvoir délivrer leurs signaux de sortie avant même que toutes les contraintes de précédence potentielles entre les opérations soient vérifiées car dans certains cas, la contrainte de précédence disparaît pour certaines configurations particulières des entrées (les cas de génération de retenue pour l'addition). Le graphe va donc se transformer en un ensemble de sous-graphes qui calculent de façon indépendante les uns par rapport aux autres, même si *a priori* des contraintes de précédence potentielles pouvaient les lier. C'est le *parallélisme dynamique* des opérateurs asynchrones.

Cette modélisation des opérateurs asynchrones par un graphe de cellules qui communiquent avec des signaux qui transportent à la fois la valeur transmise et sa date de validité permet une grande modularité dans la modélisation. En effet, les cellules peuvent être aussi bien des portes logiques de base pour décrire une porte complexe que des macro-cellules (additionneurs, mémoires, ...) pour décrire des systèmes plus complexes.

Dans le cas des opérateurs qui nous intéressent (les additionneurs réguliers basés sur des découpages en blocs), le graphe correspondant est très simple (voir la Figure 4.24).

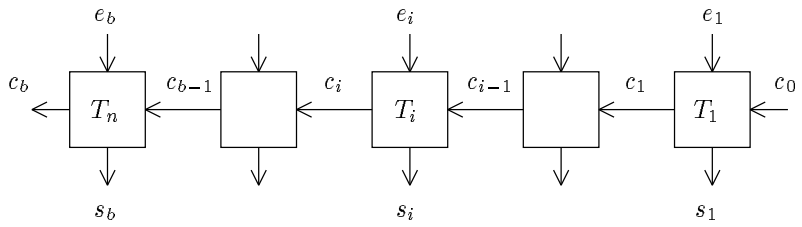



FIG. 4.24 – *Modèle de base du découpage en blocs d'un additionneur asynchrone.*

Les entrées de la cellule ou tâche $T_i$ sont de deux types. La portion $e_i$ des arguments de l'opérateur arithmétique qui correspondent à la cellule (les deux bits $a_i$ et $b_i$ dans le cas d'un additionneur séquentiel binaire) et la retenue de la cellule précédente $c_i$. L'entrée $e_i$ peut très bien être un ensemble plus important de chiffres des arguments comme dans le cas des additionneurs ayant des blocs de taille supérieure ou égale à 2. La sortie $s_i$ de la cellule est l'ensemble des bits de la somme correspondant aux entrées.

Dans les additionneurs asynchrones, ce qui fait le parallélisme dynamique c'est la détermination possible dans certains cas de la valeur de retenue sortante à partir des seules valeurs des entrées. On a représenté ci-dessous les différentes configurations possibles dans le cas d'un additionneur séquentiel binaire.

| $a_i$ | $b_i$ | $c_{i+1}$ |
|---|---|---|
| 0 | 0 | 0 (génération) |
| 0 | 1 | $c_i$ (propagation) |
| 1 | 0 | $c_i$ (propagation) |
| 1 | 1 | 1 (génération) |

Chaque signal est modélisé par une variable aléatoire $X$ correspondant à la retenue entre deux cellules adjacentes dans le graphe. Cette variable représente le fait qu'il y ait propagation ou génération.

$$X = \begin{cases} 1 & \text{si il y a propagation} \\ 0 & \text{si il y a génération} \end{cases}$$

Pour déterminer le temps de calcul de l'opérateur composé de $n$ cellules, nous modélisons les retenues par les variables $X_1$, $X_2$, $\ldots X_n$ associées aux retenues respectives $c_1$, $c_2$, $\ldots$, $c_n$. Les hypothèses probabilistes sur les $X_i$ sont :

- les $X_i$ sont indépendantes : ce qui est justifié si on suppose que les entrées des cellules sont indépendantes.
- les $X_i$ sont de même loi : les cellules sont de même taille et de même structure, et les entrées ont la même distribution dans chaque cellule.

Soit $\mu_i$ la mesure de probabilité associée à $X_i$. On note $p = \mathbb{P}(X_i = 1) = \mu_i(1)$, $0 \leq p \leq 1$, la probabilité d'avoir propagation entre la cellule $i$ et la cellule $i+1$ (dans le cas de l'additionneur séquentiel avec des entrées de loi uniforme on a $p = \frac{1}{2}$). Pour simplifier les notations on note $q = 1 - p$.

Soit $\Omega$ l'espace des suites de $\{0,1\}^{\mathbb{N}}$. $\Omega$ est muni de la mesure de probabilité produit $\otimes\mu_i$. Une configuration de taille $n$ est une suite finie de $n$ bits considérés comme les $n$ premiers bits d'une suite $s$ de $\Omega$ :

$$(X_1, X_2, \ldots, X_n)$$

Le temps de calcul pour chaque combinaison des entrées est le nombre de propagations de retenues dans l'additionneur plus une fois le délai d'une cellule. On suppose que les cellules ont des délais unitaires quelles que soient les valeurs des entrées. On définit donc la variable aléatoire $M_n$ :

$$\begin{array}{rccl} M_n : & \Omega & \longrightarrow & \{0,1,\ldots,n\} \\ & s & \longmapsto & \text{le nombre maximum de } X_i \text{ à“1” consécutifs dans les } n \text{ premiers bits de } s \end{array}$$

où $s = \{s_1, s_2, \ldots, s_n, \ldots\} \in \Omega$

Notre but est la détermination de la loi de distribution de probabilité de $M_n$. En effet, de la loi de $M_n$, on déduit valeur moyenne du temps de calcul, sa variance et sa distribution.

La variable aléatoire $M_n$ nous renseigne sur le temps nécessaire pour arriver dans l'état final de l'opérateur mais pas sur son comportement temporel. Nous définissons la variable aléatoire $\tau_k$ :

$$\begin{array}{rccl} \tau_k : & \Omega & \longrightarrow & \mathbb{N} \cup \{+\infty\} \\ & s & \longmapsto & \inf\{n \geq k \mid s_n = s_{n-1} = \ldots = s_{n-k+1} = 1\} \end{array}$$

Par convention, $\tau_k = +\infty$ lorsque $s$ ne contient aucun bloc de “1” de taille $k$ (lorsque les $X_i$ sont indépendant et identiquement distribués et $p > 0$ on a alors $\mathbb{P}(\tau_k = +\infty) = 0$).

$\tau_k$ est le premier instant d'apparition d'une suite de $X_i$ consécutifs, égaux à 1 et de taille $k$ pour une configuration donnée. Dans une configuration de taille $n$, la valeur de $\tau_k$ est forcément supérieure à $k$ et inférieure ou égale à $n$. Pour $n < k$, la première suite de longueur $k$ dans une configuration de taille $n$ ne peut pas encore exister, on a donc $+\infty$ comme valeur de $\tau_k$ dans ce cas.

Bien évidemment, les variables aléatoires $M_n$ et $\tau_k$ sont étroitement liées. En effet, si dans une suite de taille $n$ il y a au moins $k$ $X_i$ consécutifs égaux à 1 alors l'instant d'apparition de cette suite est au plus égale à $n$. On a donc l'égalité entre les événements :

$$(M_n \geq k) = (\tau_k \leq n)$$

On va donc rechercher aussi la loi de $\tau_k$ pour caractériser le comportement temporel de nos additionneurs.

Les configurations qui mènent à $\tau_k$ sont les suites de $\Omega$ qui se terminent par $k$ “1” consécutifs. L'ensemble des configurations de taille $n$ de $\Omega$ qui mènent à $\tau_k$ est (où $\epsilon$ est le mot vide) :

$$C_k = \sum_{n=0}^{+\infty} \left( \left( \epsilon + 1 + 11 + 111 + \ldots + \underbrace{111\ldots1}_{k-1} \right) 0 \right)^n \underbrace{1111\ldots1}_{k} \tag{4.1}$$

Nous allons maintenant étudier différentes techniques pour déterminer les principales caractéristiques statistiques du temps de calcul moyen de nos opérateurs.

### 4.4.3 Modélisation par les fonctions génératrices

On trouve dans [GKP94] une technique qui permet d'obtenir la fonction génératrice d'une variable aléatoire à partir de la définition en terme de langage de l'ensemble des configurations correspondant à cette variable. Cette technique consiste à remplacer dans l'expression qui définit

les configurations correspondantes à la variable étudiée les symboles du langage associé par leur probabilité de réalisation.

On obtient la fonction génératrice $\phi_k(x)$ de $\tau_k$ en utilisant cette technique à partir de la définition de $C_k$ (équation 4.1). On a donc :

$$\begin{aligned}
\phi_k(x) &= \sum_{n=0}^{+\infty} \Big( \Big(1 + px + (px)^2 + \ldots + (px)^{k-1}\Big) qx \Big)^n (px)^k \\
&= (px)^k \sum_{n=0}^{+\infty} \Big( qx \frac{1-(px)^k}{1-px} \Big)^n \\
&= (px)^k \frac{1}{1 - qx\frac{1-(px)^k}{1-px}} \\
&= \frac{(1-px)(px)^k}{1 - x + qp^k x^{k+1}}
\end{aligned}$$

Notre but est de déterminer la distribution de probabilité de la variable aléatoire $M_n$, c'est-à-dire de déterminer les quantités $p_{n,k} = \mathbb{P}(M_n = k)$. On note $\mathcal{G}_n(x)$ la fonction génératrice associée. On a donc

$$\mathcal{G}_n(x) = \mathbb{E}\, x^{M_n} = \sum_{k=0}^{n} \mathbb{P}(M_n = k) x^k = \sum_{k=0}^{n} p_{n,k} x^k$$

Soit $\mathcal{H}(x,y)$ la fonction génératrice associée à $\{\mathcal{G}_n(x)\}_{n\in\mathbb{N}^*}$. En pratique, la fonction génératrice $\mathcal{G}_n$ permet de calculer $p_{n,k}$ pour $n$ fixé et la fonction génératrice $\mathcal{H}$ permet de faire varier $n$.

$$\begin{aligned}
\mathcal{H}(x,y) &= \sum_{n=1}^{+\infty} \mathcal{G}_n(x) y^n \\
&= \sum_{n=1}^{+\infty} \sum_{k=0}^{n} p_{n,k} x^k y^n
\end{aligned}$$

Or on sait que les événements $(M_n \geq k)$ et $(\tau_k \leq n)$ sont identiques, on peut donc écrire :

$$\underbrace{\mathbb{P}(M_n \geq k)}_{\sum_{j=k}^{n} p_{n,j}} = \underbrace{\mathbb{P}(\tau_k \leq n)}_{\sum_{i=1}^{n} \mathbb{P}(\tau_k = i)}$$

A partir des définitions de $\mathcal{H}(x,y)$, de $\mathcal{G}_n(x)$, et de l'égalité précédente on déduit :

$$\sum_{n=1}^{+\infty} \sum_{k=0}^{n} \mathbb{P}(M_n \geq k) x^k y^n = \sum_{k=0}^{n} \sum_{n=1}^{+\infty} \mathbb{P}(\tau_k \leq n) y^n x^k \tag{4.2}$$

On montre que

$$\sum_{n=1}^{+\infty} \mathbb{P}(\tau_k \leq n) y^n = \frac{\phi_k(x)}{1-y} \tag{4.3}$$

avec

$$\begin{aligned}
\frac{A(x)}{1-x} &= (\sum_{k=0}^{+\infty} a_k x^k)(1 + x^2 + x^3 + \ldots) \\
&= a_0 x^0 + (a_0 + a_1) x^1 + \ldots + (a_0 + a_1 + \ldots + a_n) x^n + \ldots \\
&= \sum_{k=0}^{+\infty} (\sum_{j=0}^{k} a_j) x^k
\end{aligned}$$

Or

$$\mathbb{P}(\tau_k \leq n) = \sum_{k=1}^{n} \mathbb{P}(\tau_k = i)$$

On a donc :

$$\begin{aligned}
\sum_{k=0}^{n} \mathbb{P}(M_n \geq k) x^k &= \sum_{k=0}^{n} \sum_{j=k}^{n} p_{n,j} x^k \\
&= \sum_{k=0}^{n} \Big( \underbrace{\sum_{j=0}^{n} p_{n,j}}_{=1} x^k - \sum_{j=0}^{k-1} p_{n,j} x^k \Big) \\
&= \sum_{k=0}^{n} \Big( x^k - \sum_{j=0}^{k} p_{n,j} x^k + p_{n,k} x^k \Big) \\
&= \frac{1 - x^{n+1}}{1-x} - \underbrace{\sum_{k=0}^{n} \sum_{j=0}^{k} p_{n,j} x^k}_{\frac{\mathcal{G}_n(x)}{1-x} - \frac{x^{n+1}}{1-x}} + \mathcal{G}_n(x) \\
&= \frac{1 - x\mathcal{G}_n(x)}{1-x}
\end{aligned}$$

Donc

$$\begin{aligned}
\sum_{n=1}^{+\infty} \Big( \sum_{k=0}^{n} \mathbb{P}(M_n \geq k) x^k \Big) &= \sum_{n=1}^{+\infty} \frac{1 - x\mathcal{G}_n(x)}{1-x} y^n \\
&= \frac{1}{1-x} \Big( \sum_{n=1}^{+\infty} y^n - x \underbrace{\sum_{n=1}^{+\infty} \mathcal{G}_n(x) y^n}_{\mathcal{H}(x,y)} \Big) \\
&= \frac{1}{1-x} \Big( \frac{y}{1-y} - x\mathcal{H}(x,y) \Big) \\
&= \frac{y}{(1-x)(1-y)} - \frac{x}{1-x} \mathcal{H}(x,y)
\end{aligned}$$

A partir des équations 4.2 et 4.3, on déduit :

$$\sum_{k=0}^{+\infty} \phi_k(y) x^k = \frac{y}{1-x} - \frac{x(1-y)}{1-x} \mathcal{H}(x,y)$$

Soit finalement

$$\mathcal{H}(x,y) = \frac{y}{x(1-y)} - \frac{1-x}{x(1-y)} \sum_{k=0}^{+\infty} \phi_k(x) x^k$$

On vérifie bien que :

$$p_{n,0} = p_{n,n} = 1$$

et

$$\mathcal{H}(1, y) = \frac{y}{1 - y}$$

De la fonction génératrice $\mathcal{H}(x, y)$, on déduit la moyenne, la variance et la distribution de $M_n$. En effet,

$$\mathcal{H}(x, y) = \sum_{n=1}^{+\infty} \sum_{k=0}^{n} p_{n,k} x^k y^n$$

Donc

$$\frac{\partial \mathcal{H}(x, y)}{\partial x} = \sum_{n=1}^{+\infty} \sum_{k=0}^{n} p_{n,k} k x^{k-1} y^n$$

$$\left.\frac{\partial \mathcal{H}(x, y)}{\partial x}\right|_{x=1} = \sum_{n=1}^{+\infty} \Big( \underbrace{\sum_{k=0}^{n} k p_{n,k}}_{\mathbb{E}M_n} \Big) y^n$$

C'est-à-dire que les $n$ premiers termes du développement en série de $\frac{\partial \mathcal{H}(x,y)}{\partial x}$ en $x = 1$ nous donnent les valeurs moyennes de la plus longue chaîne de propagation de la retenue pour des additionneurs de taille 1 à $n$. Malheureusement, nous n'avons pas encore réussit à trouver une expression simple de ces valeurs.

$$\left.\frac{\partial^2 \mathcal{H}(x, y)}{\partial x^2}\right|_{x=1} = \sum_{n=1}^{+\infty} \Big( \underbrace{\underbrace{\sum_{k=0}^{n} k^2 p_{n,k}}_{\mathbb{E}M_n{}^2} - \underbrace{\sum_{k=0}^{n} k p_{n,k}}_{\mathbb{E}M_n}}_{\mathrm{Var}\, M_n + \mathbb{E}M_n^2 - (\mathbb{E}M_n)^2} \Big) y^n$$

De la même façon que pour la valeur moyenne, la variance de la longueur de la plus longue chaîne de propagation de la retenue se déduit du développement en série de $\frac{\partial^2 \mathcal{H}(x,y)}{\partial x^2}$ en $x = 1$. On peut aussi déduire les valeurs des coefficients $p_{n,k}$ en utilisant l'expression :

$$\frac{\partial^{n+k} \mathcal{H}(x, y)}{n!\, k!\, \partial y^n\, \partial x^k} = p_{n,k}$$

Cette modélisation ne permet malheureusement pas de trouver, pour le moment, une expression explicite simple pour les valeurs moyennes, les écarts types et les distributions (les $p_{n,k}$) des additionneurs asynchrones réguliers. Nous travaillons actuellement sur l'utilisation de résultats liés aux nombres de Fibonacci généralisés (cf [SF96]) pour essayer de trouver les expressions du comportement asymptotique des différentes caractéristiques des additionneurs asynchrones réguliers.

Cette modélisation à base de fonctions génératrices de probabilité peut être utilisée pour trouver les valeurs numériques des caractéristiques des additionneurs asynchrones. Toutefois, le temps de calcul nécessaire est très important et nous avons donc choisi de développer une autre modélisation à base de chaînes de Markov qui permet d'obtenir beaucoup plus rapidement la distribution du temps de calcul de nos additionneurs.

### 4.4.4 Modélisation par les chaînes de Markov

Une autre possibilité pour modéliser les opérateurs asynchrones est l'utilisation de chaînes de Markov. Plus exactement, on peut avec une chaîne de Markov relativement simple modéliser la propagation des retenues à travers les différentes cellules de l'additionneur. La Figure 4.25 représente le graphe de transition de l'automate associé à la chaîne de Markov qui modélise la plus longue chaîne de propagation de la retenue. Les sommets représentent les valeurs de la longueur de la chaîne de propagation de la retenue, et les arcs représentent la probabilité de passer d'un état à l'autre.

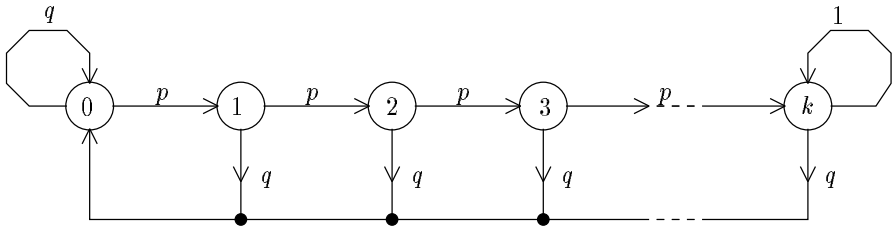



FIG. 4.25 – *Chaîne de Markov représentant un additionneur asynchrone*

On sait que

$$\mathbb{P}(\tau_k \leq n) = \mathbb{P}[\text{automate dans l'état } k \text{ à l'instant } n]$$

On peut calculer les différentes configurations $\Pi_i$ de l'automate avec la relation

$$\Pi_{n+1} = \Pi_n \times A_k$$

où $A_k$ est la matrice (de taille $k$) de transition associée au graphe

$$A_k = \begin{pmatrix} q & p & 0 & \dots & 0 \\ \vdots & 0 & p & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ q & 0 & \dots & 0 & p \\ 0 & \dots & \dots & 0 & 1 \end{pmatrix}$$

et $\Pi_0 = (1, 0, 0, \dots, 0)$

On a donc $\Pi_n = \Pi_0 \times A_k^n$ et $\mathbb{P}(\tau_k \leq n)$ est le dernier coefficient de $\Pi_n$.

$$\begin{aligned} \mathbb{P}(\tau_k = n) &= \mathbb{P}(\tau_k \leq n) - \mathbb{P}(\tau_k \leq n-1) \\ &= \Pi_0 \times A_k^{n-1} \times (A_k - I) \end{aligned}$$

On a donc ici une méthode qui permet de déterminer rapidement les coefficients $p_{n,k}$.

Les premiers tests que nous avons effectué donnent des valeurs équivalentes à celles trouvées dans nos simulations. Par exemple, nous donnons ici les résultats de la comparaison de la modélisation à base de chaînes de Markov avec ceux de la simulation d'un additionneur séquentiel de deux nombres de 16 bits (distribution uniforme des entrées avec $p = \frac{1}{2}$).

| | moyenne | écart type |
|---|---|---|
| modélisation | 5.34 | 1.51 |
| simulation | 5.24 | 1.49 |
| erreur relative | 2 % | 1 % |

## 4.5 Quelques autres opérateurs arithmétiques asynchrones

Nous présentons dans cette section les premiers résultats de l'étude d'opérateurs arithmétiques asynchrones autres que l'addition. Dans un avenir proche, nous souhaitons étudier l'ensemble des opérations classiques en arithmétique des ordinateurs (multiplication, division, racine carrée et les fonctions élémentaires). Pour le moment, nous avons étudié le multiplieur de Braun et nous avons quelques idées sur le diviseur de Guild. Cette section apporte beaucoup plus de questions qu'elle ne donne de réponses, mais elle dégage un certain nombre de problèmes à résoudre pour comprendre un peu mieux les mécanismes sous-jacents dans les opérateurs arithmétiques complexes.

### 4.5.1 Multiplieur de Braun

Le multiplieur de Braun (cf [Mul89]) est une structure cellulaire très simple et très régulière qui est la traduction parallèle directe de l'algorithme scolaire de multiplication en base 2. Un multiplieur de Braun de deux nombres de taille 4 est présenté sur la Figure 4.26.

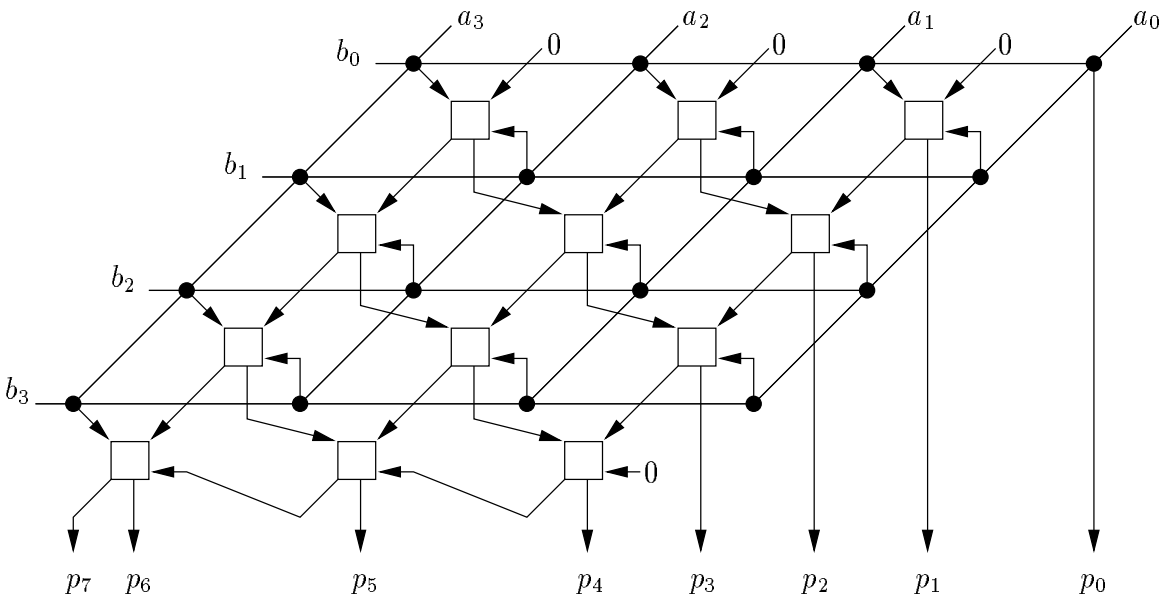



FIG. 4.26 – *Multiplieur de Braun* $4 \times 4$.

Les points noirs représentent les cellules "ET" qui calculent les produits partiels $a_i b_j$, et les boites des cellules *full-adder*. Pour simplifier le problème, nous supposons que tous les produits partiels sont calculés en parallèle et nous ne nous intéressons qu'à la partie réduction (somme) des produits partiels. Le temps de calcul dans le pire cas a une durée égale au temps de traversée de $n + p - 2$ cellules FA. Les délais des cellules FA sont tous égaux à 1.

Le Tableau 4.6 et la Figure 4.27 donnent la distribution du temps de calcul d'un multiplieur de Braun carré de nombres de 8 bits.

La Figure 4.28 représente le temps moyen de calcul de multiplieurs de Braun de taille $n \times n$ pour $n$ variant de 4 à 64.

La Figure 4.29 représente la distribution du temps de calcul de différents multiplieurs de Braun (16, 32 et 64 bits). Les valeurs moyennes et les écarts types mesurés correspondants sont donnés dans le Tableau 4.7.

| $t$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $p(t)$ | 0.00 | 0.00 | 0.01 | 0.01 | 0.03 | 0.04 | 0.10 | 0.20 | 0.28 | 0.14 | 0.11 | 0.05 | 0.02 | 0.01 | 0.00 |

TAB. 4.6 – *Distribution du temps de calcul d'un multiplieur de Braun de taille* $8 \times 8$.

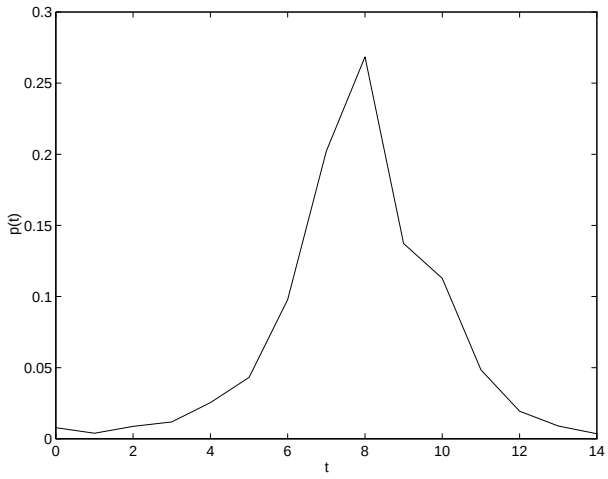



FIG. 4.27 – *Distribution du temps de calcul d'un multiplieur de Braun de taille* $8 \times 8$.

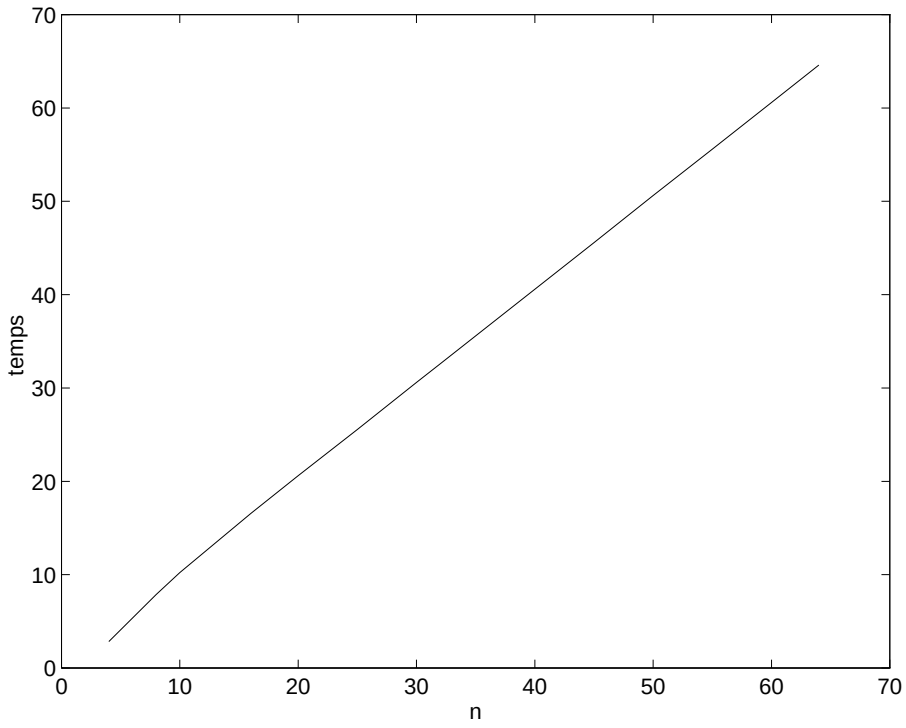



FIG. 4.28 – *Temps moyen de calcul d'un multiplieur de Braun* $n \times n$.

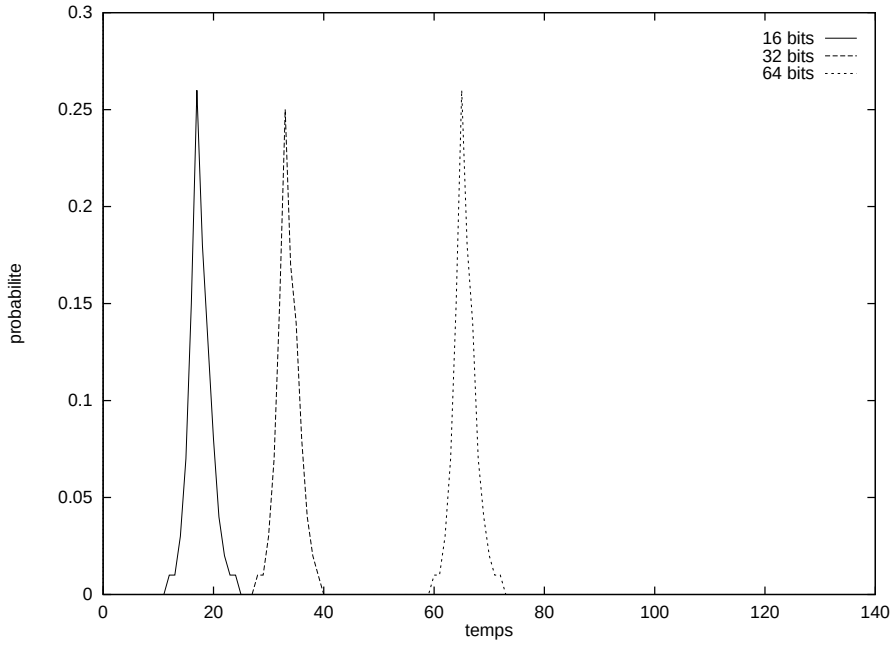



FIG. 4.29 – *Distribution du temps de calcul de différents multiplieurs de Braun* $n \times n$ $(n = 16, 32, 64)$.

| taille | moyenne | écart type |
|---|---|---|
| 16 | 17.6 | 2.04 |
| 32 | 33.5 | 1.96 |
| 64 | 65.6 | 2.03 |

TAB. 4.7 – *Valeur moyenne et écart type du temps de calcul de différents multiplieurs de Braun.*

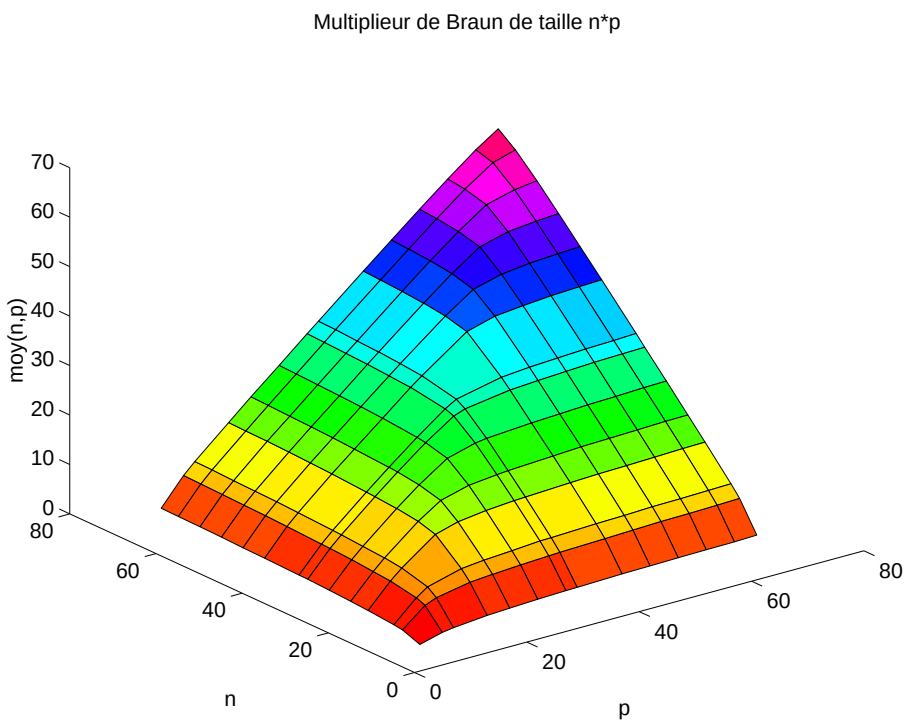



FIG. 4.30 – *Distribution du temps de calcul de différents multiplieurs de Braun $n \times p$ pour différentes valeurs de $n$ et de $p$.*

Tous nos résultats indiquent un comportement du multiplieur de Braun en asynchrone assez étonnant. La valeur moyenne du temps de calcul d'un multiplieur de Braun $n \times n$ est environ égale à $n$. Sa distribution du temps de calcul est très resserrée autour de sa moyenne, la valeur de l'écart type du temps de calcul semble indépendante de la taille des opérandes (on a tout le temps une valeur proche de 2).

Nous avons étudié l'influence de tailles différentes pour les opérandes (le multiplieur devient "rectangulaire"). La surface présentée Figure 4.30 donne le temps moyen de calcul d'un multiplieur de Braun de taille $n \times p$ en faisant varier $n$ et $p$. Pour le moment nous n'avons pas trouvé de relation simple qui nous permette de déterminer la valeur du temps moyen de calcul à la seule vue des tailles $n$ et $p$.

La structure du multiplieur de Braun a donc tendance à fortement "resynchroniser" d'elle même le temps de calcul autour de la moyenne. Le phénomène est très complexe à analyser car il y a des chaînes de propagation de retenues dans toutes les directions dans le multiplieur de Braun. Pour le moment, nous ne réussissons pas à expliquer cette distribution très resserrée du temps de calcul.

La modélisation probabiliste d'un multiplieur est extrêmement difficile car une hypothèse fortement simplificatrice dans le cas de l'addition n'est plus présente dans le cas de la multiplication : l'indépendance des entrées des cellules FA. En effet, les entrées des cellules FA sont très fortement dépendantes les unes des autres. Par exemple, tous les produits partiels $a_i b_j$ pour $i$ ou $j$ fixé sont dépendants les uns des autres (ils ont un facteur commun). De plus, les entrées des FA dans une diagonale dépendent des sorties des FA de la diagonale située juste au-dessus. Pour le moment, nous n'avons par de modèle simple et efficace pour la multiplication.

Nous travaillons actuellement en collaboration avec Marc Renaudin sur la réalisation de multiplieurs rapides asynchrones basés sur des arbres (Wallace, OS *tree*, ZM *tree*..., cf [RH94b]).

La multiplication de deux nombres de $n$ bits peut être vue comme $n$ additions de nombres de $n$ bits. L'asynchronisme permet-il alors de concevoir un multiplieur dont le temps moyen de calcul soit inférieur à $\mathcal{O}(\log_2 n)$ ? Une réponse positive à cette question avec une architecture simple serait sans aucun doute un argument important en faveur des circuits asynchrones.

### 4.5.2 Diviseur de Guild

Le diviseur de Guild (illustré Figure 4.31) est l'implantation la plus simple de la division non restaurante en base 2 dans un réseau cellulaire [Mul89]. Ce diviseur fournit le quotient dans une écriture signée, avec des chiffres qui valent 1 ou -1.

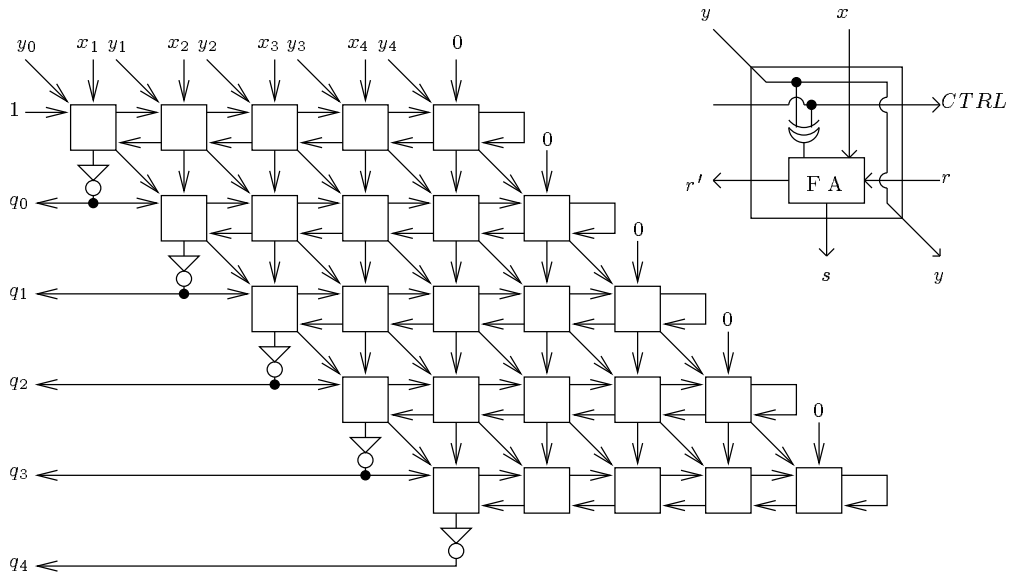



FIG. 4.31 – *Diviseur de Guild 5 bits.*

Ce diviseur n'est pas employé en synchrone car le temps calcul dans le pire cas est quadratique en la taille des opérandes (il faut passer dans toutes les cellules). Nous avons commencé l'étude de ce diviseur car en asynchrone son temps moyen de calcul nous semble être très raisonnable comparativement à des diviseurs plus complexes.

Le temps de calcul du diviseur de Guild en asynchrone est fortement dépendant du temps nécessaire à chaque ligne de cellules FA pour fournir leur chiffre du quotient. En asynchrone ce temps sera en général très court car il dépend seulement de la longueur de la chaîne de propagation de la retenue qui arrive sur la cellule FA la plus à gauche. Nous avons vu dans le cas de l'addition (chaque ligne du diviseur de Guild est un additionneur) avec une distribution uniforme des entrées que la longueur moyenne de la chaîne de propagation de retenue (pas la plus longue) est d'une taille proche de 2. Dans le cas du diviseur, les entrées des lignes ne sont pas uniformément distribuées. Mais peut être la longueur moyenne n'est pas très importante, auquel cas le temps moyen de calcul du diviseur de Guild sera en $\mathcal{O}(n)$ avec une structure très régulière et très simple? Nous travaillons actuellement sur cet opérateur.

## 4.6 Conclusion et perspectives

Nous avons vu que les circuits asynchrones offrent un cadre de réalisation pour les opérateurs arithmétiques intéressant à de nombreux points de vue. Le calcul en temps moyen est probablement la caractéristique la plus importante de ces opérateurs en asynchrone.

Pour l'addition, nous avons montré qu'en utilisant des additionneurs relativement simples (additionneur à retenue bondissante et à sélection de retenue) on peut obtenir des opérateurs dont le temps moyen de calcul est en $\mathcal{O}(\sqrt{\log_2 n})$. Les meilleurs additionneurs en synchrone ont un temps de calcul seulement en $\mathcal{O}(\log_2 n)$.

Nous avons montré que la caractérisation du comportement temporel des opérateurs arithmétiques asynchrones nécessite de connaître non seulement la valeur moyenne du temps de calcul mais aussi sa distribution pour choisir au mieux la structure la plus performante.

Nos modélisations probabilistes des additionneurs asynchrones même si elles ne nous permettent pas encore de trouver une expression simple de la valeur moyenne et de la distribution du temps de calcul, elles nous ont permis de valider notre simulateur. En effet, nous avons vu que les résultats donnés par les modélisations sont très proches de ceux trouvés lors des simulations.

Notre façon d'appréhender le fonctionnement des opérateurs arithmétiques asynchrones s'est beaucoup "affinée" durant cette étude commencée depuis peu. En effet, il nous a fallu penser les différents mécanismes de communication des retenues (propagation, génération) de façon vraiment asynchrone (passage d'un état non valide à un état "1" ou "0"). En fait, la façon de "penser" les algorithmes est radicalement différente en asynchrone. Il faut essayer d'utiliser au mieux la variabilité d'un calcul pour minimiser le temps moyen.

La conception d'opérateurs arithmétiques asynchrones performants est vraiment un axe de recherche très intéressant en arithmétique des ordinateurs. Elle nous oblige à minimiser le temps moyen de calcul des algorithmes, ce qui est beaucoup plus complexe que de minimiser le temps dans le pire cas (ce qui n'est déjà pas forcément très simple). Cela nous permet de voir les opérations arithmétiques et les algorithmes correspondants sous un autre point vue.

Les perspectives de ce travail sont nombreuses et de natures diverses. Les points suivants sont les travaux qui sont en cours de réalisation ou que nous pensons aborder prochainement.

- Modélisation par les fonctions génératrices en se ramenant à des problèmes liés aux nombres de Fibonacci généralisés dans les chaînes et les arbres digitaux (cf [SF96]). Nous espérons pouvoir déterminer les expressions asymptotiques de la valeur moyenne, de la variance (écart type) et de la distribution du temps de calcul. Ce résultat est important pour le domaine des opérateurs arithmétiques asynchrones car pour le moment seule la valeur moyenne de quelques additionneurs a été trouvée de façon formelle (additionneur séquentiel et à évaluation anticipée de la retenue *carry look ahead*).

- Etudier le cas d'autres additionneurs avec des découpages non réguliers comme les additionneurs de Brent et Kung ([BK82]). Le principal problème est alors la modélisation probabiliste car les différents signaux entre les "étages" de calcul ne sont plus indépendants les uns des autres.

- Etudier des multiplieurs plus complexes (arbres, recodages utilisant au mieux la variabilité du temps de calcul...). Mais ici le problème est encore la modélisation probabiliste de ces opérateurs. La multiplication est une addition de plusieurs termes mais qui ne sont pas indépendants les uns des autres.

- Nous étudions l'influence de la distribution réelle des entrées sur la valeur moyenne et la distribution du temps de calcul des opérateurs. Nous avons reçu très récemment des données mesurées sur la longueur des chaînes de propagation de la retenue dans les additions d'un microprocesseur ARM. Nous pensons avoir prochainement des informations supplémentaires sur ce sujet.

- Un dernier point important sur l'addition est d'essayer de trouver la valeur optimale du temps moyen de calcul. Nous travaillons actuellement dans cette direction.

- La consommation des circuits intégrés est un critère très important pour la réalisation de systèmes embarqués et d'appareils portables (cf [Fra79] page 29). Les opérateurs arithmétiques asynchrones offrent des temps de calcul très faibles avec des architectures très simples. Par exemple, un simple additionneur séquentiel effectue la somme de deux nombres de $n$ bits en un temps moyen de $\mathcal{O}(\log_2 n)$ avec une architecture considérablement plus simple que les additionneurs synchrones en $\mathcal{O}(\log_2 n)$. Nous souhaitons comparer les versions synchrones et asynchrones de différents opérateurs arithmétiques en terme de puissance consommée.

- Nous pensons aborder le calcul de fonctions plus complexes comme la division, la racine carrée et les fonctions élémentaires en utilisant des anneaux auto-séquencés (cf [Has95]). Cette architecture asynchrone semble particulièrement bien adaptée pour l'évaluation de fonctions arithmétiques en utilisant des algorithmes à base d'additions et de décalages.

# Conclusion

DANS ce travail de thèse nous avons abordé différentes solutions pour améliorer les performances de certaines opérations en arithmétique des ordinateurs. Dans le terme performances nous regroupons différents aspects : la vitesse des calculs avec le système semi-logarithmique et les opérateurs asynchrones, la fiabilité des calculs avec l'arrondi exact des fonctions élémentaires et enfin le coût des calculs (taille des circuits) avec l'arithmétique en-ligne. Nous résumons ci-dessous les différents thèmes que nous avons abordé durant cette thèse :

- **L'arrondi exact des fonctions élémentaires**
  Après avoir analysé le problème, nous avons effectué des tests exhaustifs pour les fonctions élémentaires en simple précision. Les résultats de ces tests permettent d'ores et déjà de réaliser une bibliothèque de calcul certifié des fonctions élémentaires pour les nombres flottants codés en simple précision. Parallèlement, nous avons réalisé une modélisation probabiliste du problème et nous avons vérifié la validité de cette modélisation en effectuant des tests aléatoires pour les précisions supérieures (double et quadruple précisions). Tous les résultats que nous avons obtenus montrent que la précision intermédiaire qui est en général nécessaire pour arrondir exactement les fonctions élémentaires est un peu supérieure au double de la taille des mantisses finales (après arrondi). En nous basant sur la stratégie multi-niveaux de Ziv, nous avons proposé une solution pour la réalisation effective de l'arrondi exact des fonctions élémentaires en machine avec un coût raisonnable. En effet, avec cette stratégie incrémentale, le temps moyen d'évaluation de ces fonctions ne sera presque pas augmenté par l'ajout de la phase d'arrondi.

  Les perspectives de cette étude sont multiples : les tests exhaustifs en double précision, une bibliothèque de calcul en très grande précision et une extension de norme IEEE 754 aux fonctions élémentaires. Avec l'arrivée de Vincent Lefèvre en thèse dans notre équipe, nous pensons pouvoir tester les fonctions élémentaires au moins dans des petits domaines dans le cas de la double précision. Pour les précisions supérieures les tests exhaustifs ne seront probablement jamais possibles. De plus, les bornes théoriques seront probablement améliorées mais jusqu'à quel point ? Nous avons donc commencé avec un stage de DEA la réalisation d'une bibliothèque de calcul certifié des fonctions élémentaires sur plusieurs millions de bits. Cette bibliothèque pourra être utilisée pour réaliser le niveau ultime de la stratégie de Ziv. Enfin, nous pensons qu'il est temps maintenant de penser à une extension de la norme IEEE 754 aux cas des fonctions élémentaires.

- **Le système semi-logarithmique de représentation des nombres**
  Nous avons conçu un nouveau système de représentation des nombres réels situé à mi-chemin entre le système flottant et le système logarithmique. Ce système permet d'effectuer les additions et les soustractions plus rapidement et en utilisant moins d'espace que dans le système logarithmique. Nous avons conçu des algorithmes pour calculer les opérations de base (addition, soustraction, multiplication et division) et effectuer les conversions vers et depuis le système flottant. Notre étude de l'erreur relative du système semi-logarithmique montre qu'il

est comparable en précision aux systèmes flottant et logarithmique.

Nous pensons implanter prochainement nos algorithmes pour réaliser un processeur semi-logarithmique pour pouvoir effectuer des comparaisons réelles avec le système flottant. Dans ce cadre, nous pensons aborder le problème de l'évaluation des fonctions élémentaires dans le système semi-logarithmique. De plus, nous avons initié une collaboration avec Bernard Girau (coopérant au CSEM) sur l'utilisation du système semi-logarithmique dans une application de réseaux de neurones basée sur des ondelettes.

- **L'arithmétique en-ligne sur FPGA**
  Nous avons réalisé une bibliothèque complète d'opérateurs en-ligne adaptée pour des implantations sur FPGA. Nous avons défini un système de conception du contrôle des opérateurs en-ligne basé sur un fonctionnement local et totalement distribué. Cette technique de contrôle permet de réaliser simplement et rapidement des circuits complexes intégrant un grand nombre d'opérateurs en-ligne. Nous avons aussi développé une technique de validation des algorithmes en-ligne qui peut être utilisée facilement par le concepteur pour "debugger" rapidement ses réalisations.

  Les perspectives de ce travail sont ici encore assez nombreuses. En effet, nous poursuivons l'élaboration de notre bibliothèque, en particulier, nous pensons ajouter dans les mois à venir des opérateurs en-ligne pour évaluer les fonctions élémentaires basés sur des techniques de tabulation et d'approximations polynômiales combinées. Nous comptons étudier l'évaluation des fonctions élémentaires en-ligne en utilisant des algorithmes à base d'additions et de décalages. Dans une autre direction, nous pensons étendre notre bibliothèque pour des implantations dans des circuits intégrés conventionnels (CMOS). L'étude de la consommation des opérateurs en-ligne est aussi un point important que nous allons aborder durant mon séjour post-doctoral.

- **Les opérateurs arithmétiques asynchrones**
  Nous avons vu dans la dernière partie de cette thèse que les opérateurs asynchrones offrent d'importants potentiels pour la réalisation d'opérateurs arithmétiques. Après avoir présenté les principaux aspects relatifs aux circuits intégrés asynchrones, nous avons étudié la réalisation et les performances de plusieurs types d'additionneurs asynchrones rapides. Pour cette étude, nous avons mis au point un simulateur qui permet de tester différents algorithmes avec une bonne précision (le modèle utilisé est très proche du fonctionnement réel des circuits asynchrones) et suffisamment rapide pour effectuer de nombreux tests statistiques pour extraire les principales caractéristiques de ces opérateurs. Nous avons étudié en détail trois additionneurs : l'additionneur séquentiel, l'additionneur à retenue bondissante et l'additionneur à sélection de retenue. En particulier, nous avons montré qu'il est possible d'effectuer une addition de deux nombres de $n$ bits en temps moyen $\mathcal{O}(\sqrt{\log_2 n})$ en utilisant des additionneurs de type retenue bondissante ou à sélection de retenue (la complexité de l'addition est $\mathcal{O}(\log_2 n)$). Nous avons modélisé mathématiquement les comportements de ces additionneurs en utilisant différentes techniques : les fonctions génératrices de probabilités et les chaînes de Markov. Bien que ces modélisations ne permettent pas d'obtenir facilement une expression simple des caractéristiques statistiques de nos opérateurs, elles nous fournissent un cadre formel robuste et puissant qui peut être utilisé pour obtenir des données numériques utiles pour la conception effective de ces opérateurs. Nous avons abordé le cas de la multiplication à travers le multiplieur de Braun et celui des autres opérateurs (division et fonctions élémentaires) en utilisant des anneaux auto-séquencés.

Les perspectives de cette partie de la thèse sont nombreuses et variées. D'un point de vue théorique, il reste encore beaucoup de points à étudier pour obtenir un modèle précis et utilisable du fonctionnement des opérateurs arithmétiques asynchrones. Nous pensons poursuivre avec Jean-Marc Vincent (LMC-IMAG) nos travaux sur les différents aspects liés à la modélisation probabiliste de ces opérateurs (fonctions génératrices, chaînes de Markov). Le but ultime de ce travail de modélisation est de réussir à obtenir une expression de la complexité asymptotique du temps de calcul moyen des opérateurs arithmétiques asynchrones. D'un point de vue pratique, nous collaborons actuellement avec l'équipe de Marc Renaudin (CNET) sur la réalisation à terme d'un ensemble d'opérateurs arithmétiques complet (addition, multiplication, division et fonctions élémentaires). Enfin, nous souhaitons continuer avec Jean-Michel Muller (LIP) à étudier les mécanismes internes des opérations arithmétiques en asynchrone pour mettre au point des algorithmes plus performants.

# Bibliographie


[ABCC89] M. G. Arnold, T. A. Bailey, J. R. Cowles, et J. J. Cupal. Redundant logarithmic number systems. Dans Ercegovac et Swartzlander, éditeurs, *Proceedings of the 9th IEEE Symposium on Computer Arithmetic*, pages 144–151, Santa Monica, USA, 1989. IEEE Computer Society Press, Los Alamitos, CA.

[ABCW92] M. G. Arnold, T. A. Bailey, J. R. Cowles, et M. D. Winkel. Applying features of IEEE 754 to sign/logarithm arithmetic. *IEEE Transactions on Computers*, 41(8):1040–1050, Août 1992.

[AEGP67] S. F. Anderson, J. G. Earle, R. E. Goldschmidt, et D. M. Powers. The IBM 360/370 model 91: floating-point execution unit. *IBM Journal of Research and Development*, Janvier 1967. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990.

[AI85] American National Standards Institute et Institute of Electrical and Electronics Engineers. IEEE standard for binary floating point arithmetic. Dans ANSI/IEEE Standard, éditeur, *Std 754-1985*, 1985.

[AI87] American National Standards Institute et Institute of Electrical and Electronics Engineers. IEEE standard for radix independent floating point arithmetic. Dans ANSI/IEEE Standard, éditeur, *Std 854-1987*, 1987.

[Avi61] A. Avizienis. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, 10:389–400, 1961. Reprinted in E.E. Swartzlander, Computer Arithmetic, Vol. 2, IEEE Computer Society Press Tutorial, 1990.

[Baj93] J. C. Bajard. *Evaluation de fonctions dans des systèmes redondants d'écriture des nombres.* Thèse, Ecole Normale Supérieure de Lyon, Université Claude Bernard, 1993.

[Bak75] G. A. Baker. *Essentials of Padé approximants.* Academic Press, New York, 1975.

[BBK+94] K. Van Berkel, R. Burgess, J. Kessels, M. Roncken, F. Schalij, et A. Peeters. Asynchronous circuits for low power: A DCC error corrector. *IEEE Design and Test of Computers*, 11(2):22–32, Juin 1994.

[BDKM94] J.C. Bajard, J. Duprat, S. Kla, et J.M. Muller. Some operators for on-line radix 2 computations. *Journal of Parallel and Distributed Computing*, 22:336–345, 1994.

[BEW89] R. H. Brackert, M. D. Ercegovac, et A. N. Willson. Design of an on-line multiply-add module for recursive digital filters. Dans M. D. Ercegovac et E. Swartzlander, éditeurs, *Proceedings of the 9th IEEE Symposium on Computer Arithmetic*, pages 34–41, Santa Monica, USA, sep 1989. IEEE Computer Society Press, Los Alamitos, CA.

[BGN46] A.W. BURKS, H.H. GOLDSTINE, ET J. VON NEUMANN. Preliminary discusion of the logical design of an electronic instrument. Rapport technique, The Institut of Advanced Study, Princeton, N.J., 1946. (reprinted in Bell and Newell, Computer structures: readings and examples, Computer Science series, Mc Graw-Hill, 1971).

[BK82] R. P. BRENT ET H. T. KUNG. A regular layout for parallel adders. *IEEE Transactions on Computers*, C-31(3):260–264, Mars 1982. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 2, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990.

[BK81] R.P. BRENT ET H.T. KUNG. The area-time complexity of binary multiplication. *Journal of the ACM*, 28(3):521–534, Juillet 81.

[BKM93] J. C. BAJARD, S. KLA, ET J. M. MULLER. BKM: A new hardware algorithm for complex elementary functions. Dans E. E. Swartzlander, M. J. Irwin, et J. Jullien, éditeurs, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, Windsor, Canada, June 1993. IEEE Computer Society Press, Los Alamitos, CA.

[BKM94] J. C. BAJARD, S. KLA, ET J. M. MULLER. BKM: A new hardware algorithm for complex elementary functions. *IEEE Transactions on Computers*, 43(8):955–963, Août 1994.

[Bre70] R.P. BRENT. On the addition of binary numbers. *IEEE Transactions on Computers*, pages 758–759, Août 1970.

[Bre76] R. P. BRENT. Fast multiple precision evaluation of elementary functions. *Journal of the ACM*, 23:242–251, 1976.

[CDHM91] G. CORBAZ, J. DUPRAT, B. HOCHET, ET J.M. MULLER. Implementation of a VLSI polynomial evaluator for real-time applications. Dans *Proceeding of ASAP91*, 1991.

[Cod73] W. J. CODY. Static and dynamic numerical characteristics of floating-point arithmetic. *IEEE Transactions on Computers*, C-22(6):598–601, Juin 1973.

[CW80] W. CODY ET W. WAITE. *Software manual for the elementary functions.* Prentice-Hall Inc, 1980.

[Dad89] L. DADDA. On serial input multipliers for two's complement numbers. *IEEE Transactions on Computers*, 38:1341–1345, 1989.

[Dau88] JOHN G. DAUGMAN. Complete discrete 2-D gabor transforms by neural networks for image analysis and compression. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 36(7):1169–1179, Juillet 1988.

[Dau96] M. DAUMAS. *Contributions à l'arithmétique des ordinateurs: vers une maîtrise de la précision.* Thèse, Ecole Normale Supérieure de Lyon, 1996.

[DDH94] M.E. DEAN, D.L. DILL, ET M. HOROWITZ. Self-timed logic using curent-sensing completion detection (cscd). *Journal of VLSI Signal Processing*, (7):7–16, 1994.

[DM88] J. DUPRAT ET J. M. MULLER. Hardwired polynomial evaluation. *Journal of Parallel and distributed Computing*, (5), 1988. Special Issue on Parallelism in Computer Arithmetic.

[DMMM94] M. DAUMAS, C. MAZENC, X. MERRHEIM, ET J. M. MULLER. Fast and accurate range reduction for computation of the elementary functions. Dans *Proceedings of the 14th IMACS World Congress on Computational and Applied Mathematics*, pages 1196–1198. IMACS, Piscataway, NJ,, 1994.

[DMMM95] M. DAUMAS, C. MAZENC, X. MERRHEIM, ET J. M. MULLER. Modular range reduction: A new algorithm for fast and accurate computation of the elementary functions. *Journal of Universal Computer Science*, 1(3):162–175, Mars 1995.

[DMT96] M. DAUMAS, J.M. MULLER, ET A. TISSERAND. Theoretical support for standardized elementary functions. Dans IEEE-SMC, éditeur, *Symposium on Modelling, Analysis and Simulation.*, volume 2, pages 1133–1138, Juillet 1996. CESA'96 IMACS Multiconference.

[DMT97] M. DAUMAS, J.M. MULLER, ET A. TISSERAND. Very high radix on-line arithmetic for accurate computations. Dans *Proceedings of the 15th IMACS World Congress on Computational and Applied Mathematics*, 1997.

[DMV94] M. DAUMAS, J. M. MULLER, ET J. VUILLEMIN. Implementing on-line arithmetic on pam. Dans *4th International Workshop on Field-Programmable Logic and Applications*, Sept. 1994.

[EL85] M. D. ERCEGOVAC ET T. LANG. A division algorithm with prediction of quotient digits. Dans *Proceedings of the 7th IEEE Symposium on Computer Arithmetic*, Urbana, USA, 1985. IEEE Computer Society Press, Los Alamitos, CA.

[EL87] M. D. ERCEGOVAC ET T. LANG. On-the-fly conversion of redundant into conventional representations. *IEEE Transactions on Computers*, C-36(7), Juillet 1987. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 2, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990.

[EL90] M. D. ERCEGOVAC ET T. LANG. Redundant and on-line CORDIC: Application to matrix triangularization and SVD. *IEEE Transactions on Computers*, 39(6):725–740, Juin 1990.

[EL92] M. D. ERCEGOVAC ET T. LANG. On-the-fly rounding. *IEEE Transactions on Computers*, 41(12):1497–1503, Décembre 1992.

[EL94] M. D. ERCEGOVAC ET T. LANG. *Division and Square Root: Digit-Recurrence Algorithms and Implementations.* Kluwer Academic Publishers, Boston, 1994.

[EMT95] M. D. ERCEGOVAC, J. M. MULLER, ET A. TISSERAND. Fpga implementation of polynomial evaluation algorithms. Dans *SPIE Photonics East'95 Conference Proceedings*, 1995.

[Erc77] M.D. ERCEGOVAC. A general hardware-oriented method for evaluation of functions and computations in a digital computer. *IEEE Transactions on Computers*, C-26(7):667–680, 1977.

[Erc78] M.D. ERCEGOVAC. An on-line square rooting algorithm. Dans *Proceedings of the 4th IEEE Symposium on Computer Arithmetic.* IEEE Computer Society Press, 1978.

[Erc84] M.D. ERCEGOVAC. On-line arithmetic: an overview. Dans SPIE, éditeur, *SPIE, Real Time Signal Processing VII*, pages 86–93, 1984.

[ET77] M.D. ERCEGOVAC ET K.S. TRIVEDI. On-line algorithms for division and multiplication. *IEEE Transactions on Computers*, C-26(7):681–687, 1977.

[ET87] M. D. ERCEGOVAC ET P. K. G. TU. A radix-4 on-line division algorithm. Dans *Proceedings of the 8th IEEE Symposium on Computer Arithmetic.* IEEE Computer Society Press, 1987.

[ET89] M. D. ERCEGOVAC ET P. K. G. TU. Design of on-line division unit. Dans Milos D. Ercegovac et Earl Swartzlander, éditeurs, *Proceedings of the 9th IEEE Symposium on Computer Arithmetic*, pages 42–49. IEEE Computer Society Press, 1989.

[ET91] M. D. ERCEGOVAC ET P. K. G. TU. Application of on-line arithmetic algorithms to the SVD computation: preliminary results. Dans P. Kornerup et D. W. Matula, éditeurs, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 246–255, Grenoble, France, June 1991. IEEE Computer Society Press, Los Alamitos, CA.

[FG76] A. FELDSTEIN ET R. GOODMAN. Convergence estimates for the distribution of trailing digits. *Journal of the ACM*, 23:287–297, 1976.

[Fra79] FRANQUIN. *Lagaffe mérite des baffes*. Numéro 13. Dupuis, 1979.

[GHM87] A. GUYOT, B. HOCHET, ET J. M. MULLER. A way to build efficient carry-skip adders. *IEEE Transactions on Computers*, C-36(10), Octobre 1987.

[GHM89] A. GUYOT, Y. HERREROS, ET J. M. MULLER. Janus, an on-line multiplier/divider for manipulating large numbers. Dans *Proceedings of the 9th IEEE Symposium on Computer Arithmetic*, pages 106–111. IEEE Computer Society Press, 1989.

[GKP94] R.L. GRAHAM, D.E. KUNTH, ET O. PATASHNIK. *Concrete Mathematics: a foundation for computer science*. Addison-Wesley, 1994.

[GO96] A. DE GLORIA ET M. OLIVIERI. Statistical carry lookahead adders. *IEEE Transactions on Computers*, 45(3):340–347, Mars 1996.

[Gol91] D. GOLDBERG. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–47, Mars 1991.

[GT96] B. GIRAU ET A. TISSERAND. On-line arithmetic based reprogrammable hardware implementation of multilayer perceptron back-propagation. Dans IEEE, éditeur, *5th International Conference on Microelectronics for Neural Networks and Fuzzy Systems MICRONEURO 96*, pages 168–175. IEEE Computer Society Press, Février 1996. Lausanne, Switzerland.

[Ham70] R. W. HAMMING. On the distribution of numbers. *Bell Systems Technical Journal*, 49:1609–1625, 1970. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990.

[Has95] B. EL HASSAN. *Architecture VLSI Asynchrone utilisant la logique différentielle à précharge: Application aux opérateurs arithmétiques*. Thèse, Institut National Polytechnique de Grenoble, 1995.

[Hau95] S. HAUCK. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83(1):69–93, Janvier 1995.

[HC87] M. HATAMIAN ET G.L. CASH. Parallel bit-level pipelined VLSI designs for high-speed signal processing. Dans *Proceedings of the IEEE*, volume 75, pages 1192–1202, Septembre 1987.

[HC90] R. HARTLEY ET P. CORBETT. Digit-serial processing techniques. *IEEE Transactions on Circuits and Systems*, 37(6):707–719, Juin 1990.

[HCC96] S.-C. HUANG, L.-G. CHEN, ET T.-H. CHEN. A 32-bit logarithmic number system processor. *Journal of VLSI Signal Processing*, 14(3):311–320, Décembre 1996.

[Hen89] H. HENKEL. Improved addition for the logarithmic number systems. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37:301–303, 1989.

[HLD70] E.L. HALL, D.D. LYNCH, ET S.J. DWYER. Generation of products and quotients using approximate binary logarithms for digital filtering applications. *IEEE Transactions on Computers*, C-19(2):97–105, Janvier 1970.

[HP96] J.L. HENNESSY ET D.A. PATTERSON. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann, 1996. second edition.

[IO79] M.J. IRWIN ET R.M. OWENS. On-line algorithms for the design of pipeline architecture. Dans *Proceedings of the 4th IEEE Symposium on Computer Architecture*. IEEE Computer Society Press, 1979.

[IO87] M.J. IRWIN ET R.M. OWENS. Digit-pipelined arithmetic as illustrated by the paste-up system: a tutorial. *IEEE Computer*, pages 61–73, 1987.

[Irw78] M. J. IRWIN. A pipelined processing unit for on-line division. Dans *5th Symposium on Computer Architecture*, pages 24–30. IEEE Computer Society Press, Los Alamitos, CA, 1978.

[IV94] P. IENNE ET M.A. VIREDAZ. Bit-serial multipliers and squarers. *IEEE Transactions on Computers*, 43(12):1445–1450, Décembre 1994.

[IV95] P. IENNE ET M.A. VIREDAZ. Genes iv: A bit-serial processing element for a multi-model neural-network accelerator. *Journal of VLSI Signal Processing*, 9:257–273, 1995.

[JB90] G.M. JACOBS ET R.W. BRODERSENN. A fully asynchronous digital signal processor using self-timed circuits. *IEEE Journal of Solid-State Circuits*, 25(6):1526–1537, Décembre 1990.

[Kla93] S. KLA. *Calcul parallèle en en-ligne des fonctions arithmétiques*. Thèse, Ecole Normale Supérieure de Lyon, 1993.

[KM81] U. W. KULISCH ET W. L. MIRANKER. *Computer arithmetic in theory and practice*. Academic Press, New York, 1981.

[KM85] P. KORNERUP ET D. W. MATULA. Finite precision lexicographic continued fraction number systems. Dans *Proceedings of the 7th IEEE Symposium on Computer Arithmetic*, Urbana, USA, 1985. IEEE Computer Society Press, Los Alamitos, CA. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 2, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990.

[KM91] T. KUROKAWA ET T. MIZUKOSHI. A fast and simple method for curve drawing: a new approach using logarithmic number systems. *Journal of Information Processing*, 14:144–152, 1991.

[Kor93] I. KOREN. *Computer arithmetic algorithms*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[KPL80] T. KUROKAWA, J.A. PAYNE, ET S.C. LEE. Error analysis of recursive digital filters implemented with logarithmic number systems. *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-28(6):706–715, Décembre 1980.

[KR71] N.G. KINGSBURY ET P.J.W. RAYNER. Digital filtering using logarithmic arithmetic. *Electronic Letters*, 7:56–58, 1971. Reprinted in E.E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990.

[Kul77] U. W. KULISCH. Mathematical foundation of computer arithmetic. *IEEE Transactions on Computers*, C-26(7):610–621, Juillet 1977.

[Lai91] F.-S. LAI. A 10-ns hybrid number system data execution unit for digital signal processing systems. *IEEE Journal of Solid-State Circuits*, 26(4):590–599, Avril 1991.

[LB61] M. LEHMAN ET N. BURLA. Skip techniques for high-speed carry propagation in binary arithmetic units. *IRE Transactions on Electronic Computers*, page 691, Décembre 1961.

[LE92] M. E. LOUIE ET M. D. ERCEGOVAC. Mapping division algorithms to field programmable gate arrays. Dans *Proceedings of the 1992 Asilomar Conference on Signals, Systems and Computers*, 1992.

[LE93a] M. E. LOUIE ET M. D. ERCEGOVAC. A digit-recurrence square root implementation for field programmable gate arrays. Dans *IEEE Workshop on FPGAs for Custom Computing Machines*, Avril 1993.

[LE93b] M. E. LOUIE ET M. D. ERCEGOVAC. On digit-recurrence division implementation for field programmable gate arrays. Dans E. E. Swartzlander, M. J. Irwin, et J. Jullien, éditeurs, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 202–209, Windsor, Canada, Juin 1993. IEEE Computer Society Press, Los Alamitos, CA.

[LE93c] M.E. LOUIE ET M.D. ERCEGOVAC. On digit-recurrence division implementations for field programmable gate arrays. Dans IEEE Comp. Soc. Press, éditeur, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 202–209, Juin 1993.

[Lef96] V. LEFÈVRE. Calcul certifié des fonctions élémentaires. Master's thesis, Ecole Normale Supérieure de Lyon, 1996.

[Lew93] D. M. LEWIS. An accurate lns arithmetic using interleaved memory function interpolator. Dans E. E. Swartzlander, M. J. Irwin, et J. Jullien, éditeurs, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 2–9, Windsor, Canada, June 1993. IEEE Computer Society Press, Los Alamitos, CA.

[Lew95] D.M. LEWIS. 114 MFLOPS logarithmic number system arithmetic unit for dsp applications. *IEEE Journal of Solid-State Circuits*, 30(12):1547–1553, Décembre 1995.

[LL86] R.O. LAMAIRE ET J.H. LANG. Performance of digital linear regulators which use logarithmic arithmetic. *IEEE Transactions on Automatic Control*, AC-31(5):394–400, Mai 1986.

[LMT97] V. LEFÈVRE, J. M. MULLER, ET A. TISSERAND. Towards correctly rounded transcendentals. Dans *Proceedings of the 13th IEEE Symposium on Computer Arithmetic*, Asilomar, USA, 1997. IEEE Computer Society Press, Los Alamitos, CA.

[LS87] H. LIN ET H. J. SIPS. A novel floating-point on-line division algorithm. Dans M. J. Irwin et R. Stefanelli, éditeurs, *Proceedings of the 8th IEEE Symposium on Computer Arithmetic*, Como, Italy, May 1987. IEEE Computer Society Press, Los Alamitos, CA.

[LW91] F. S. LAI ET C. F. E. WU. A hybrid number system processor with geometric and complex arithmetic capabilities. *IEEE Transactions on Computers*, 40(8):952–962, Août 1991.

[Mal89] STEPHANE G. MALLAT. A theory for multiresolution signal decomposition: The wavelet representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(7):674–693, Juillet 1989.

[Mat89] D. MATULA. Highly parallel divide and square root algorithms for a new generation floating point processor. Dans *SCAN-89 Symposium on Computer Arithmetic and Self-Validating Numerical Methods*, Octobre 1989.

[Maz93] C. MAZENC. *Systèmes de représentation des nombres et arithmétiques sur machines parallèles.* Thèse, École Normale Supérieure de Lyon, Décembre 1993.

[MBL+89] ALAIN J. MARTIN, STEVEN M. BURNS, T. K. LEE, DRAZEN BORKOVIC, ET PIETER J. HAZEWINDUS. The design of an asynchronous microprocessor. Dans Charles L. Seitz, éditeur, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 351–373. MIT Press, 1989.

[MI81] S. MATSUI ET M. IRI. An overflow/underflow free floating-point representation of numbers. *Journal of Information Processing*, 4(3):123–133, 1981. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 2, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990.

[MK85] D.W. MATULA ET P. KORNERUP. Finite precision rational arithmetic: Slash number systems. *IEEE Transactions on Computers*, C-34(1):3–18, Janvier 1985.

[MMY93] X. MERRHEIM, J. M. MULLER, ET H. J. YEH. Fast evaluation of polynomials and inverses of polynomials. Dans E. E. Swartzlander, M. J. Irwin, et J. Jullien, éditeurs, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 186–192, Windsor, Canada, June 1993. IEEE Computer Society Press, Los Alamitos, CA.

[MP90] P.C. MATHIAS ET L.M. PATNAIK. Systolic evaluation of polynomial expression. *IEEE Transactions on Computers*, pages 208–220, 1990.

[MRM93] J. MORAN, I. RIOS, ET J. MENESES. Signed digit arithmetic on FPGAs. Dans *International Workshop on FPGA: Logic and Applications*, Septembre 1993. Oxford.

[MST95] J. M. MULLER, O. SCHERBYNA, ET A. TISSERAND. Semi-logarithmic number systems. Dans *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, Bath, UK, 1995. IEEE Computer Society Press, Los Alamitos, CA.

[MT96] J. M. MULLER ET A. TISSERAND. Towards exact rounding of the elementary functions. Dans Alefeld, Frommer, et Lang, éditeurs, *Scientific Computing and Validated Numerics (Proceedings of SCAN'95)*, Wuppertal, Germany, 1996. Akademie Verlag.

[MTV97] J.M. MULLER, A. TISSERAND, ET J.M. VINCENT. Asynchronous sub-logarithmic adders. Dans IEEE, éditeur, *1997 IEEE Pacific Rim Conference on Communication, Computers and Signal Processing (PACRIM97)*, volume 2, pages 515–518, Août 1997. Victoria Canada.

[Mul89] J.M. MULLER. *Arithmetique des ordinateurs.* Masson, 1989.

[Mul94] J. M. MULLER. Some characterizations of functions computable in on-line arithmetic. *IEEE Transactions on Computers*, 43(6), Juin 1994.

[Mul95] J. M. MULLER. Algorithmes de division pour microprocesseurs : illustration à l'aide du "bug" du pentium. *Technique et Science Informatiques*, 14(8), Octobre 1995.

[Mul97] J.M. MULLER. *Elementary Functions, Algorithms and Implementation.* Birkhauser, Boston, 1997.

[NW96] Y. V. NESTERENKO ET M. WALDSCHMIDT. On the approximation of the values of exponential function and logarithm by algebraic numbers (in russian). *Mat. Zapiski*, 2:23–42, 1996.

[OE82] V. G. OKLOBDZIJA ET M. D. ERCEGOVAC. An on-line square root algorithm. *IEEE Transactions on Computers*, C-31:70–75, 1982.

[OF95] S. F. OBERMAN ET M. J. FLYNN. An analysis of division algorithms and implementations. Rapport Technique CSL-TR-95-675, Computer Systems Laboratory, Dept. on Electrical Engineering and Computer Science Stanford University, 1995.

[OF96] S. F. OBERMAN ET M. J. FLYNN. Fast IEEE rounding for division by functional iteration. Rapport Technique CSL-TR-96-700, Computer Systems Laboratory, Dept. on Electrical Engineering and Computer Science Stanford University, Juillet 1996.

[Olv87] F. W. J. OLVER. Como, italy. Dans *Proceedings of the 8th IEEE Symosium on Computer Arithmetic*, Los Alamitos, CA, May 1987. IEEE Computer Society Press, Los Alamitos, CA. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 2, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990.

[Omo94] A. R. OMONDI. *Computer Arithmetic Systems, Algorithms, Architecture and Implementations*. Prentice Hall International Series in Computer Science, Englewood Cliffs, NJ, 1994.

[PZ95] J.A. PRABHU ET G.B. ZYNER. 167 MHz radix-8 divide and square root using overlapped radix-2 stages. Dans S. Knowles et W.H. McAllister, éditeurs, *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, pages 155–162. IEEE Computer Society Press, 1995.

[QTF91] N. QUACH, N. TAKAGI, ET M. FLYNN. On fast IEEE rounding. Rapport Technique CSL-TR-91-459, Computer Systems Laboratory, Dept. on Electrical Engineering and Computer Science Stanford University, Janvier 1991.

[Ren96] M. RENAUDIN. Conférence invitée. Journées Adéquation Algorithme Architecture en Traitement du Signal et des Images, 1996. CNES Toulouse.

[RH94a] M. RENAUDIN ET B. EL HASSAN. The design of fast asynchronous adder structures and their implementation using d.c.v.s. logic. Dans *Proceedings ISCAS*, Mai 1994.

[RH94b] M. RENAUDIN ET B. EL HASSAN. A low power, 100 mhz 12x18+30b multiplier-accumulator operating in asynchronous and synchronous modes. Dans *Proceedings ESSCIRC*, 1994.

[Rob58] J.E. ROBERTSON. A new class of digital division methods. *IRE Transactions on Electronic Computers*, EC-17, 1958.

[RRPB96] F. ROBIN, M. RENAUDIN, G. PRIVAT, ET N. VAN DEN BOSSCHE. A functionally asynchronous array-processor for morphological filtering of greysacle images. *IEE Computers and Digital Techniques*, Juillet 1996. Special Issue on Asynchronous Processors.

[RRV97] M. RENAUDIN, F. ROBIN, ET P. VIVET. Aaaa: Asynchronisme et adéquation algorithme architecture. *à paraître dans TSI*, 1997.

[SA75] E. E. SWARTZLANDER ET A. G. ALEXPOULOS. The sign-logarithm number system. *IEEE Transactions on Computers*, Décembre 1975. Reprinted in E. E. Swartzlander,

*Computer Arithmetic*, Vol. 1, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990.

[SBH89] M.R. SANTORO, G. BEWICK, ET M.A. HOROWITZ. Rounding algorithms for IEEE multipliers. Dans IEEE Computer Society Press, éditeur, *Proceedings of the 9th Symposium on Computer Arithmetic*, pages 176–183, 1989.

[SD88] S. G. SMITH ET P. B. DENYER. Advanced serial-data computation. *Journal of Parallel and Distributed Computing*, 5:228–249, 1988.

[SF96] R. SEDGEWICK ET P. FLAJOLET. *Introduction à l'analyse des algorithmes*. International Thomson Publishing, 1996.

[Sip84] H. J. SIPS. Bit-sequential arithmetic for parallel processors. *IEEE Transactions on Computers*, C-33(1), Janvier 1984.

[SS93] M. SCHULTE ET E. E. SWARTZLANDER. Exact rounding of certain elementary functions. Dans E. E. Swartzlander, M. J. Irwin, et G. Jullien, éditeurs, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 138–145, Windsor, Canada, June 1993. IEEE Computer Society Press, Los Alamitos, CA.

[SS94] M. J. SCHULTE ET E. E. SWARTZLANDER. Hardware designs for exactly rounded elementary functions. *IEEE Transactions on Computers*, 43(8):964–973, Août 1994.

[SSCNS83] E.E. SWARTZLANDER, D.V. SATISH-CHANDRA, H.T. NAGLE, ET S.A. STARKS. Sign/logarithmic arithmetic for FFT implementation. *IEEE Transactions on Computers*, C-32(6):526–534, Juin 1983.

[ST88] T. STOURAITIS ET F. J. TAYLOR. Floating-point to logarithmic encoder error analysis. *IEEE Transactions on Computers*, C-37:858–863, 1988.

[Tay83] F. J. TAYLOR. An overflow-free residue multiplier. *IEEE Transactions on Computers*, C-32:501–504, 1983.

[TD97] A. TISSERAND ET M. DIMMLER. Fpga implementation of real-time digital controllers using on-line arithmetic. Dans Springer, éditeur, *Field Programmable Logic and Applications*, 1997.

[TGJR88] F. J. TAYLOR, R. GILL, J. JOSEPH, ET J. RADKE. A 20 bit logarithmic number system processor. *IEEE Transactions on Computers*, 37(2):190–200, Février 1988.

[Tis94] A. TISSERAND. Arithmétique en-ligne sur l'accélérateur matériel DECPeRLe-1. Rapport de DEA, Ecole Normale Supérieure de Lyon, juin 1994.

[Tis96] A. TISSERAND. FPGA implementation of on-line arithmetic operators for digital control. Dans IFIP, éditeur, *International Workshop on Logic and Architecture Synthesis (IWLAS'96)*, pages 115–122, Décembre 1996. Grenoble, France.

[Toc58] K. D. TOCHER. Techniques of multiplication and division for automatic binary divider. *Quarterly journal of mechanics and applied mathematics*, 11(3):364–384, 1958.

[Tur91] P. R. TURNER. Implementation and analysis of extended sli operations. Dans P. Kornerup et D. W. Matula, éditeurs, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 118–126, Grenoble, France, June 1991. IEEE Computer Society Press, Los Alamitos, CA.

[Tur93] P. R. TURNER. Complex sli arithmetic: representation, algorithms, and analysis. Dans E. E. Swartzlander, M. J. Irwin, et J. Jullien, éditeurs, *Proceedings of the 11th IEEE*

*Symposium on Computer Arithmetic*, pages 18–25, Windsor, Canada, June 1993. IEEE Computer Society Press, Los Alamitos, CA.

[Ver88] T. VERHOEFF. Delay-insensitive codes - an overview. *Distributed Computing*, 3:1–8, 1988.

[Vol59] J. VOLDER. The cordic computing technique. *IRE Transactions on Computers*, 1959.

[Vui83] J.E. VUILLEMIN. Exact real computer arithmetic with continued fractions. *IEEE Transactions on Computers*, 39(8), 1983.

[Vui94] J. VUILLEMIN. On circuits and numbers. *IEEE Transactions on Computers*, 43(8):868–879, Août 1994.

[Wal90] P. J. L. Wallis, éditeur. *Improving Floating-Point Programming*. Wiley, New York, 1990.

[WDF+97] J.V. WOODS, P. DAY, S.B. FURBER, J.D. GARSIDE, N.C. PAVER, ET S. TEMPLE. AMULET1: An asynchronous ARM microprocessor. *IEEE Transactions on Computers*, 46(4):385–397, Avril 1997.

[WF91] D. C. WONG ET M. J. FLYNN. Fast division using accurate quotient approximations. Dans P. Kornerup et D. W. Matula, éditeurs, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 191–201, Grenoble, France, June 1991. IEEE Computer Society Press, Los Alamitos, CA.

[Win65] S. WINOGRAD. On the time required to perform addition. *Journal of the Association for Computing Machinery*, 12(2):277–285, Avril 1965.

[Yok92] H. YOKOO. Overflow/underflow-free floating-point number representations with self-delimiting variable-length exponent fields. *IEEE Transactions on Computers*, 41(8):1033–1039, Août 1992.

[YZ95] R.K. YU ET G.B. ZYNER. 167 MHz radix-4 floating point multiplier. Dans S. Knowles et W.H. McAllister, éditeurs, *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, pages 149–154. IEEE Computer Society Press, 1995.

[Ziv91] A. ZIV. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, 17(3):410–423, Septembre 1991.

[Zur94] D. ZURAS. More on squaring and multiplying large integers. *IEEE Transactions on Computers*, 43(8):899–908, Août 1994.

# Bibliographie personnelle

### Revues internationales

- Semi-logarithmic number systems. En collaboration avec J.M. Muller et A. Scherbyna. A paraître dans IEEE Transactions on Computers.

- Digital Control of Micro-Systems using On-Line Arithmetic. En collaboration avec M. Dimmler, U. Holmberg et R. Longchamp. Soumis à IEEE Transactions on Mechatronics.

### Actes de conférences internationales avec comité de lecture

- FPGA implementation of real-time digital controllers using on-line arithmetic. En collaboration avec M. Dimmler. Dans Springer éditeur, *Field Programmable Logic and Applications*, 1997.

- Asynchronous sub-logarithmic adders. En collaboration avec J.M. Muller et J.M. Vincent. Dans IEEE éditeur, *1997 IEEE Pacific Rim Conference on Communication, Computers and Signal Processing (PACRIM97)*. IEEE Computer Society Press, Août 1997.

- Towards correctly rounded transcendentals. En collaboration avec V. Lefèvre et J.M. Muller. Dans IEEE éditeur, *13th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society Press, Juillet 1997.

- Very high radix on-line arithmetic for accurate computations. En collaboration avec M. Daumas et J.M. Muller. Dans *Proceedings of the 15th IMACS World Congress on Computational and Applied Mathematics*, 1997.

- FPGA implementation of on-line arithmetic operators for digital control. Dans IFIP éditeur, *International Workshop on Logic and Architecture Synthesis (IWLAS'96)*, pages 115–122, Décembre 1996. Grenoble, France.

- Theoretical support for standardized elementary functions. En collaboration avec M. Daumas et J.M. Muller. Dans IEEE-SMC éditeur, *Symposium on Modelling, Analysis and Simulation.*, volume 2, pages 1133–1138, Juillet 1996. CESA'96 IMACS Multiconference.

- On-line arithmetic based reprogrammable hardware implementation of multilayer perceptron back-propagation. En collaboration avec B. Girau. Dans IEEE éditeur, *5th International Conference on Microelectronics for Neural Networks and Fuzzy Systems MICRONEURO 96*, pages 168–175. IEEE, Février 1996. Lausanne, Switzerland.

- FPGA implementation of polynomial evaluation algorithm. En collaboration avec M.D. Ercegovac et J.M. Muller. Dans SPIE éditeur, *Field Programmable Gate Arrays for Fast Board Development and Reconfigurable Computing*, volume 2607, pages 177–188. SPIE, Octobre 1995. Philadelphia, Pennsylvania.

- Towards exact rounding of the elementary functions. En collaboration avec J.M. Muller. Dans Mathematical Research éditeur, *International Symposium On Scientific Computing, Computer Arithmetic and Validated Numerics SCAN 95*, volume 90, pages 59–71, Septembre 1995. Wuppertal, Germany.

- Semi-logarithmic number systems. En collaboration avec J.M. Muller et A. Scherbyna. Dans IEEE éditeur, *12th Symposium on Computer Arithmetic*, pages 201–207. IEEE Computer Society Press, Juillet 1995. Bath, England.

### Actes de conférences nationales avec comité de lecture

- Etude d'un produit scalaire haute précision sur mémoire active programmable. Dans Réseau Doctoral en Architecture de Machines et Systèmes, éditeur, *Journées Jeunes Chercheurs en Architectures de Machines et Systèmes*, pages 211–220. PRC ANM, Décembre 1994. Monastir, Tunisie.

### Rapports de recherche

- Asynchronous sub-logarithmic adders. En collaboration avec J.M. Muller et J.M. Vincent. Rapport de recherche LIP : 97-12, 1997.

- On-line arithmetic based reprogrammable hardware implementation of multilayer perceptron back-propagation. En collaboration avec B. Girau Rapport de recherche LIP : 96-14, 1996.

- FPGA implementation of polynomial evaluation algorithm. En collaboration avec M.D. Ercegovac et J.M. Muller. Rapport de recherche LIP : 95-34, 1995.

- Towards exact rounding of the elementary functions. En collaboration avec J.M. Muller. Rapport de recherche LIP : 95-33, 1995.

- Semi-logarithmic number systems. En collaboration avec J.M. Muller et A. Scherbyna. Rapport de recherche LIP : 95-04, 1995.

- Arithmétique en-ligne sur l'accélérateur matériel DECPeRLe-1. Rapport de DEA, Ecole Normale Supérieure de Lyon, Juin 1994.

# Index

# ADÉQUATION ARITHMÉTIQUE ARCHITECTURE PROBLÈMES ET ÉTUDE DE CAS

ARNAUD TISSERAND

**Mots clés :** arithmétique des ordinateurs, architecture des ordinateurs, arrondi exact, système logarithmique, calcul en-ligne, circuits asynchrones.

**Résumé :** Les machines actuelles offrent de plus en plus de fonctionnalités arithmétiques au niveau matériel. Les générations actuelles de processeurs proposent des opérateurs matériels rapides pour le calcul des divisions, des racines carrées, des sinus, des cosinus, des logarithmes... La littérature du domaine montre qu'en changeant notre façon de représenter les nombres et/ou en utilisant des algorithmes spécifiques de calcul, il est possible de réaliser des opérateurs matériels particulièrement efficaces. Le but de cette thèse est d'étudier et d'illustrer les liens profonds existants entre l'arithmétique et l'architecture des ordinateurs à travers quatre problèmes.

Les *Opérateurs Arithmétiques Asynchrones* permettent de calculer les fonctions arithmétiques (addition, multiplication, division) avec un délai variable. En particulier, nous avons développé un additionneur asynchrone dont le temps moyen de calcul est $O(\sqrt{\log n})$.

L'*Arithmétique En-Ligne* permet de réaliser des architectures où les nombres circulent en série, chiffre par chiffre, les poids forts en tête. L'intérêt de cette arithmétique est de pouvoir calculer toutes les fonctions (en arithmétique série poids faibles en tête, il n'est pas possible de calculer les divisions et les maximums) et d'obtenir des opérateurs de petite taille avec un nombre d'entrées/sorties plus faible que leur équivalents parallèles.

L'*Arrondi Exact des Fonctions Elémentaires* consiste à déterminer la précision intermédiaire permettant de toujours pouvoir arrondir "exactement" les résultats du calcul des fonctions élémentaires (sinus, cosinus, exponentielle...). Nous proposons dans cette thèse une méthode qui permet d'arrondi exactement les fonctions élémentaires assez rapidement.

Le *Système Semi-Logarithmique de Représentation des Nombres* est un système permettant d'effectuer rapidement les calculs de problèmes dont le nombre de multiplications/divisions est grand devant le nombre d'additions/soustractions.

---

# ARITHMETIC ARCHITECTURE ADEQUACY PROBLEMS AND CASE STUDY

ARNAUD TISSERAND

**Keywords :** computer arithmetic, computer architecture, exact rounding, logarithmic number system, on-ligne computation, asynchronous circuits.

**Abstract :** Nowadays, computers offer more and more arithmetic functions wired in hardware. Last generations of processors integrate fast hardware operators to evaluate divisions, square roots, sines, cosines, logarithms... Many references in computer arithmetic stress on the fact that changing number representation and/or using specific algorithms make it possible to build very efficient hardware operators. The goal in this thesis is to investigate and illustrate the fundmental links between computer arithmetic and computer architecture through four real problems.

*Self-Timed Circuits* evaluate the arithmetic functions with variable input-dependent computation time. We have developped low complexity asynchronous adders, including a space efficient adder with average computation time of $O(\sqrt{\log n})$.

With *On-Line Arithmetic*, numbers flow serially, one digit each clock cycle, most significant digit first. All the functions can be implemented with on-line arithmetic whereas common serial arithmetic, least significant digit first, is restricted to addition and multiplication. I described in VHDL a set of on-line operators much smaller that require less I/O pads than their parallel counterpart.

To attain *Exact Rounding of the Elementary Functions* one has to estimate the appropriate internal precision required for each function (sine, cosine...). We detail here a method which allow reasonnably fast exact rounding of the elementary functions.

The *Semi-Logarithmic Number System* is an hybrid notation that allows fast average computation in problems even in the presence of a high number of multiplications and divisions compared to the number of additions and subtractions.