UNIVERSITE SAAD DAHLEB DE BLIDA

Faculté des Sciences de l'Ingénieur

Département d'électronique

THÈSE DE DOCTORAT

En Électronique

Spécialité : Microélectronique

IMPLANTATION MATÉRIELLE DU CALCUL INTENSIF EN VIRGULE FLOTTANTE DE LA DIVISION ET DES FONCTIONS ÉLÉMENTAIRES

Par

Mohamed ANANE

Devant le jury composé de :

Abderrezak GUESSOUM, Professeur, Université de Blida	Président
Mouloud KOUDIL, Professeur, ESI	Examinateur
Abdelhamid MERAGHNI, Professeur, ENS Kouba	Examinateur
Nadjia BENBLIDIA, M.C (A), Université de Blida	Examinatrice
Hamid BESSALAH, Maître de Recherche, CDTA	Promoteur
Hassen SALHI, M.C (A), Université de Blida	Co-Promoteur

Blida, 4 mai 2011

DÉDICACES

À mon défunt père
À ma mère
À ma femme et mes enfants
À toute ma famille
À mes amis

REMERCIEMENTS

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

Monsieur Abderrezak GUESSOUM, Professeur à l'université SAAD DAHLEB de Blida, pour avoir accepté de présider mon jury.

Monsieur Mouloud KOUDIL, Professeur et directeur de l'École nationale Superieure d'Informatique (ESI) pour avoir accepté d'examiner ce modeste travail malgré ses diverses préoccupations.

Monsieur Abdelhamid MERAGHNI, Professeur et directeur de l'école Normale Supérieure de Kouba, qui fut mon co-directeur de thèse de magister, je le remercie une deuxième fois.

Madame Nadjia BENBLIDIA Maître de conférences à l'université SAAD DAHLEB de Blida, pour avoir accepté d'être dans ce jury.

Monsieur Hamid BESSALAH, Maître de Recherche au CDTA, mon directeur de thèse qui n'a épargné aucun effort pour m'aider dans ce travail.

Monsieur Hassen SALHI, mon Co-directeur de thèse, Maître de conférences à l'université SAAD DAHLEB de Blida, pour sa gentillesse et sa disponibilité.

Je ne remercie jamais autant ma collègue et ma femme Nadjia pour ses précieux aides et encouragements dans tous mes travaux de recherche. Je remercie aussi mon ancien étudiant ISSAD qui est devenu par la suite un collègue et un ami. Un grand merci va aussi à toute l'équipe A3SP du CDTA bencherif, alim, sofiane, khadidja, et le dernier recru kerdjidj pour toute la convivialité et l'esprit de travail que nous avons réussi à créer dans cette équipe.

Mes remerciements vont aussi à mes deux compagnons nocturnes Boualem Socrate et Lyes Van Gogh pour toutes les nuits de travail passées au CDTA, pour le club des chercheurs que nous avons initié, pour toutes les discussions philosophiques qui venaient détendre l'atmosphère de travail.

Je tiens à remercier aussi mon ami et mon frère hamid Bellemou pour ses encouragements et pour toutes les années passées au CDTA.

Un grand merci va à tous mes collègues et amis enseignants de l'ESI et chercheurs du CDTA et plus particulièrement ceux des laboratoires microélectronique et architectures des systèmes dont j'ai fait partie.

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

RÉSUME

Les besoins en calculs ont été à l'origine de nombreux progrès de l'informatique. Si de nos jours, cette science a beaucoup d'autres applications, le calcul reste un de ses éléments de bases. Dans cette thèse, on s'intéresse particulièrement à une classe d'algorithmes orientés matériel pour augmenter les performances des calculs. Ces derniers ne doivent pas uniquement être rapides, mais requérir aussi peu de ressources matérielles pour fournir un résultat avec un niveau de précision acceptable.

L'objectif de cette thèse est de concevoir des architectures matérielles pour le calcul de la division et un certain nombre de fonctions élémentaires telles que (sinus, cosinus, exponentielle, logarithme, etc…) en double précision de la norme IEEE-754 puis de les implémenter sur circuits FPGA de Xilinx de la famille Virtex-2 avec des métriques de performances : calcul rapide, surface réduite et une précision de calcul de 1 ulp *(one unit in last place).*

Ces architectures sont dédiées a être utilisées comme cœur ou IP (*Intellectual Property*) pour l'accélération des calculs dans les applications DSP, multimédia et des applications à calcul intensif

الملخص :

الهدف من هذه الرسالة هو تصميم هندسيات لحساب القسمة وعددا من الدوال الأساسية، مثل (الجيب والجيب التمام ، الدالة الأوسية ، اللوغاريتم ، الخ...) في الدقة المضاعفة للنظام IEEE - 754 ثم زرعها في دارة FPGA من عائلة Virtex-2 للمصمم Xilinx مع التركيز على رفع الفعاليات: سرعة أداء الحساب ، الدقة القصوى والمساحة المحتلة أصغر ما يمكن .

بعض الدوال الجبرية مفيدة في الدوائر المتكاملة الحديثة مثل أنظمة على رقاقة للاتصالات السلكية واللاسلكية وتطبيقات الوسائط المتعددة. حتى عندما لا يتم استخدامها بشكل متكرر ، حساب وقتهم يمكن أن يؤثر تأثيرا كبيرا على الفعاليات العامة.

TABLE DES MATIERES

LISTE DES FIGURES

LISTE DES TABLEAUX

The Fast drives out the Slow even if the Fast is wrong

W. Kahan

INTRODUCTION

Les besoins en calculs ont été à l'origine de nombreux progrès de l'informatique. Si de nos jours, cette science a beaucoup d'autres applications, le calcul reste un de ses éléments de base. Les opérations arithmétiques sont parmi les instructions élémentaires de calcul dans les microprocesseurs, les processeurs de traitement de signaux numériques et dans les accélérateurs graphiques etc. L'addition est l'opération arithmétique la plus fréquente dans les applications gourmandes en calcul numérique, la multiplication suit de près, puis la division et les autres fonctions élémentaires. Augmenter les performances d'une application donnée revient souvent à augmenter les performances d'exécution de ces opérations arithmétiques et fonctions élémentaires. Si les performances de l'addition et de la multiplication ont fait l'objet de plusieurs améliorations, le gap, en terme de performances, qui sépare ces deux opérations avec la division et les fonctions élémentaires reste très important. Souvent les performances de ces dernières représentent un goulot d'étranglement dans les applications à calcul intensif.

Par ailleurs, les avancées de ces dernières années en microélectronique et notamment en technologie VLSI a rendu possible et intéressante l'implémentation matérielle d'algorithmes de calcul puissants, alors qu'il y'a quelques années ils étaient non accessibles.

Ces contraintes de performances conjuguées à la grande disponibilité du silicium, pour les circuits VLSI et les circuits FPGA, ont motivé le développement d'algorithmes orientés matériel pour venir à bout des performances requises par les applications numériquement intensives où le recours à la division et aux fonctions élémentaires est plus fréquent. Les méthodes matérielles de calcul de ces opérations sont diverses et dépendent de la quantité de matériel que l'on est prêt à leur consacrer. Souvent, les algorithmes arithmétiques permettant d'obtenir de bonnes performances sont complexes et leurs mises en œuvre sur matériel est quelquefois sujet de compromis entre la vitesse d'exécution et la surface occupée. Ce compromis devient plus sévère quand les supports d'implémentation sont des FPGAs, qui sont des circuits à ressources limitées.

Au cours de ces dernières années, les FPGA sont devenus l'option favorite pour la mise en œuvre matérielle des systèmes numériques, car ils permettent d'adapter le matériel à une application donnée. Ces circuits disposent de ressources matérielles qui ne cessent d'augmenter en quantité et en qualité. La souplesse sans précédent de la technologie FPGA est en fait une approche idéale pour les applications complexes de conception de systèmes

embarqués, ainsi que pour les applications de traitement de signal et de calcul scientifique à hautes performances

Autrefois, le niveau d'abstraction dans la conception d'un circuit intégré était le transistor. Avec la complexité des systèmes à intégrer qui ne cesse d'augmenter, alliée aux développements spectaculaires qu'a connu la technologie de fabrication des circuits VLSI, le niveau d'abstraction dans la conception a évolué tour à tour au niveau cellule standard (où le concepteur n'utilise pas le transistor comme élément de base mais une cellule composée de plusieurs transistors), vient après le bloc comme élément de base (qui est lui-même constitué de plusieurs cellules).

Les méthodes de conception ainsi que les logiciels n'ont cessé d'évoluer, dans le but de minimiser le coût de développement qui prend de plus en plus une part importante dans le coût global d'une application, pour arriver aujourd'hui à utiliser des cœurs ou des blocs IP (*Intellectual Property*). Ces IP sont des descriptions de fonctionnalités diverses qui peuvent être implémentés sur différentes cibles. Leur conception est devenue une industrie à part entière. Toutefois, seuls des blocs avec des fonctionnalités figées et peu variées sont disponibles actuellement.

La motivation, commune à tous les travaux présentés ici, fut ainsi de développer des architectures dédiées aux calculs de la division et de quelques fonctions élémentaires en double précision de la virgule flottante. Ces architectures doivent être compactes et optimisées à plusieurs niveaux, algorithmique, architecturale et le niveau le plus bas d'implémentation (niveau slice).

Il faut rappeler que les progrès réalisés dans le domaine du calcul intensif scientifique ne sont pas seulement dus à l'intégration! Si l'on voit de jour en jour croître la capacité mémoire et la puissance de calcul des ordinateurs, bon nombre de mathématiciens appliqués se rappellent le jour où la mise en œuvre de tel ou tel nouvel algorithme a permis de multiplier par 10, parfois par 100, la rapidité d'un calcul précis. La recherche se poursuit, non seulement pour inventer de nouvelles méthodes, mais aussi pour améliorer la fiabilité de celles existantes ou encore mieux les adapter aux nouveaux types de matériels et de problèmes actuels. C'est dans ce contexte que s'est tracé nos premiers pas dans ce travail. Où un grand intérêt est porté aux algorithmes de calcul des fonctions considérées dans cette thèse, dans l'optique de porter des améliorations algorithmiques et de les adapter au support d'implémentation qui est le FPGA.

Cette thèse aborde principalement le calcul matériel, en double précision de la norme IEEE-754, de deux types de fonctions :

- Le premier concerne la division qui fera l'objet du chapitre 2.
- Le second est relatif aux fonctions élémentaires telles que (l'exponentielle (e^x), le logarithme $ln(x)$, l'inverse $(1/x)$, la racine carrée $\sqrt{x}$, le sinus $sin(x)$ et le cosinus $cos(x)$ qui fera l'objet du chapitre 3.

Lorsqu'il s'agit d'implémenter une opération dans un matériel, les exigences sont fortes. En particulier, il faut que l'implémentation utilise le moins de ressources matérielles possibles afin de limiter les coûts et le temps d'exécution. Ceci est particulièrement vrai dans un contexte industriel où il faut être compétitif à moindre coût.

Il est clair que minimiser tous ces paramètres n'est pas chose aisée. Généralement, un gain en temps d'exécution se paie par un surcoût matériel. Réciproquement, un gain matériel conduit à un temps d'exécution plus long. On parle alors de compromis à trouver entre le temps d'exécution et la surface nécessaire à l'implémentation.

A défaut de minimiser toutes ces quantités, nous cherchons à obtenir le meilleur compromis temps surface dans un environnement de contraintes données.

En améliorant le compromis temps-surface, on obtient à surface équivalente un meilleur temps d'exécution. De même, à des temps d'exécution égaux, on obtient une diminution de la surface.

Cette thèse est constituée de trois chapitres. Dans le premier chapitre, on présente le contexte dans lequel se placent les travaux décrits dans cette thèse. On définit le calcul intensif et ses besoins en termes de calcul d'opérations arithmétiques élémentaires ainsi que la norme IEEE-754 qui régit la représentation des nombres dans toutes les architectures développées dans le cadre de cette thèse. On présente aussi les circuits FPGA qui sont les supports d'implémentation de nos architectures, le langage VHDL qui est un langage de description matériel utilisé dans notre travail et enfin l'outil de conception ISE de Xilinx.

Le chapitre 2 est consacré à la division SRT, qui est un algorithme itératif pour le calcul de la division dont les performances temporelles dépendent de plusieurs paramètres. Une étude approfondie de l'algorithme SRT a été effectué pour mettre en évidence ces paramètres. Dans ce chapitre, l'implémentation de plusieurs variantes d'architectures pour les trois bases (4, 8 et 16) avec différents facteurs de redondance sont présentées. L'influence de ces paramètres sur l'exécution de la division SRT est discuté et des solutions pour augmenter ces performances sont proposées.

Le chapitre 3 présente l'implémentation d'une méthode de calcul des fonctions élémentaires. L'objectif est alors de réaliser une architecture qui peut être utilisée comme un IP pour le calcul des fonctions élémentaires. Cette architecture doit être performante, dédiée aux calculs intensifs et n'utilisant que les ressources de base des circuits FPGA pour pouvoir être implémentée dans une large gamme de circuits FPGA. En plus d'être performante celle-ci doit être reconfigurable pour pouvoir calculer un certain nombre de fonctions avec un même niveau de précision ($\mathbf{1}\ \boldsymbol{ulp}$). Un autre effort est consenti dans le but de standardisation dans la mesure où les ressources matérielles consommées par l'architecture ne varient pas en passant du calcul d'une fonction à l'autre afin de faciliter la reconfiguration.

Les résultats d'implémentation ainsi que les performances des architectures sont discutés et comparées à des travaux similaires antérieurs. Et on termine par une conclusion et quelques perspectives.

CHAPITRE 1

Contexte

1.1. Calcul intensif

Le calcul intensif est défini comme étant « l'Ensemble des techniques et des moyens destinés à traiter des applications complexes en faisant appel à des ordinateurs spécialisés dans le traitement rapide de gros volumes de données numériques [1]. En clair c'est la puissance nécessaire à l'exécution d'une application complexe bien définie. Cette puissance est la quantité de calcul (nombre d'opérations élémentaires) que l'on peut exécuter par unité de temps. En effet, une application peut devoir répondre à des contraintes de temps ou de ressources. Une quantité de calcul relativement modeste devant s'exécuter en un temps réduit peut demander une grande puissance de calcul.

Une application sera considérée intensive si son code doit être fortement optimisé pour utiliser au mieux les ressources de calcul disponibles et respecter les contraintes de temps. Quand on pense au calcul intensif, on se réfère souvent à une certaine catégorie d'applications. Un des points communs de ces applications à calcul intensif est qu'elles s'intéressent à la simulation. Ces applications, on les retrouve dans les domaines comme la modélisation du climat [2], la simulation d'explosions nucléaires, l'aéronautique, la physique des particules, la biologie moléculaire ou encore l'astronomie. Ce type d'application demande en général une énorme quantité de calcul et les contraintes de temps d'exécution implicites sont liées à l'échelle de temps des activités humaines. Dans cette catégorie d'application, on utilise souvent des supercalculateurs, des grilles de calcul, etc.

D'un autre côté, on assiste à une montée en puissance d'une nouvelle catégorie d'applications dans le domaine de traitement du signal ou de l'image, tels que les

télécommunications (radiotéléphonie de 3^{éme} et 4^{éme} génération) ou le multimédia. Ces applications, mêmes si elles sont relativement moins demandeuses en quantité de calcul doivent souvent répondre à des contraintes de temps réel qui les rendent gourmandes en puissance de calcul. De plus, elles sont souvent embarquées. Des contraintes de taille et de consommation énergétique se superposent alors aux contraintes de temps pour nécessiter une forte optimisation de leurs architectures.

Les systèmes sur silicium, ou SoC (System on Chip) [3], sont des architectures hautement intégrées regroupant sur une même puce au moins un processeur programmable, de la mémoire et des unités de traitement accélératrices câblées. Ces puces intègrent aussi les interfaces aux matériels périphériques ou au monde extérieur. Ces systèmes sont particulièrement adaptés aux applications intensives embarquées qui comportent une partie de traitement intensif du signal ou de l'image.

Ces SoC sont composés de modules virtuels réutilisables, tels que des cœurs de processeurs, des unités de traitement du signal (DSP), du matériel de contrôle de protocoles, des blocs analogiques, des unités matérielles dédiées ou des bus intégrés sur la puce. Certaines de ces unités peuvent être parallèles, en particulier celles qui sont dédiées au traitement de signal intensif.

1.2. Format des nombres

Avant d'aborder toute solution matérielle pour une quelconque application, il faut d'abord spécifier le type de représentation des nombres que l'on va adopter. Car le choix d'un système de représentation numérique a des répercussions sur la complexité des algorithmes, de l'exécution des opérations arithmétiques et donc sur les coûts et les performances des circuits qui mettent en œuvre ces algorithmes.

Plusieurs représentations de nombres existent, cependant chacune d'entre elles est plus adaptée à un domaine d'applications qu'à un autre. Par exemple pour représenter un nombre dont la valeur varie de 2^{-12} à 2^{12}, il faut au moins 24 bits dans un système de représentation en virgule fixe. Dans ce système plusieurs représentations sont possibles [4], celles-ci vont de la représentation d'entiers non négatifs à la représentation en complément à 2. Ces représentations ont été développées afin de faciliter l'exécution sur matériel des opérations de bases (+, -, × ÷). Le système de représentation en virgule fixe reste suffisant pour une certaine classe d'applications en traitement de signal où les données sont représentées par un faible nombre de bits. Par ailleurs, il existe d'autres représentations telles que la représentation redondante (pour l'arithmétique en ligne) [5] qui est préconisée

pour l'implémentation d'algorithme fortement récursif, la représentation RNS [6] pour la cryptographie, etc. Alors que la notation en virgule flottante est la représentation la plus implémentée dans les processeurs actuels.

La notation en virgule flottante est utilisée pour le calcul scientifique, si nous voulons utiliser ce système de représentation, nous avons alors besoin de définir plusieurs aspects. Ceux-ci incluent la taille de la mantisse, l'emplacement du point de fractionnement dans celle-ci, la taille de l'exposant, etc. De ce fait, plusieurs formats peuvent exister, d'ailleurs certains ont été de facto les normes utilisées par les grands constructeurs de processeurs et certains ont été mis au point par des organismes tels que *l'Institute of Electrical and Electronic Engineers* (IEEE) dans le souci d'uniformisation des représentations. La norme IEEE a été développée afin de soutenir la portabilité entre ordinateurs de différents fabricants ainsi que les différents langages de programmation. Elle met l'accent sur des questions telles que l'arrondi correct des nombres pour obtenir des réponses fiables.

L'organisation IEEE a adopté deux normes : une première norme IEEE-754 [7], créée en 1985 puis révisée en 2008 [8], fixe le fonctionnement des calculateurs scientifiques en base 2 et une deuxième norme IEEE-854 [9] créée en 1987 qui est venue par la suite pour fixer le fonctionnement des calculateurs dans une base quelconque. Toutefois, les auteurs de ces normes expliquent que seules les bases 2 et 10 présentent un intérêt.

Aujourd'hui, le système de représentation des nombres à base de la norme IEEE-754 est le plus largement implémenté dans les processeurs actuels et se sera le standard adopté pour le travail qui est accompli dans le cadre de cette thèse. Par conséquent, il sera présenté plus en détail.

Cette norme définit entre autres, les points suivants :

- La façon dont sont représentés les nombres en machine,
- La précision devant être fournie par certaines opérations,
- Les propriétés devant être respectées pour tout ce qui relève des conversions,
- Les exceptions pouvant être rencontrées,
- Les arrondis disponibles.

1.2.1. La norme IEEE-754

Les nombres dans ce format de représentation sont caractérisés par trois champs, une mantisse $\boldsymbol{m_x}$, un exposant $\boldsymbol{e}$ et un signe $\boldsymbol{s}$.

$$x = (-1)^s \times m_x \times 2^e$$

Où :

– L'**exposant** *e* est basé sur une représentation biaisée pour des raisons d'efficacité de l'implémentation matérielle. Cela signifie que la représentation de l'exposant *e* correspond à la valeur *e* plus un biais entier. Ce biais est égal à 127 pour la simple précision et 1023 pour la double précision. Par exemple, l'exposant 0 est représenté par la valeur du biais. Cette représentation permet dans le cas des flottants positifs, de traiter les nombres flottants comme des entiers pour les comparaisons. La valeur de l'exposant évolue alors entre e_{min} et e_{max}.

– La **mantisse** m_x est un nombre en virgule fixe composé de n bits, qui est de 24 bits pour la simple précision et de 53 bits pour la double précision. La mantisse est toujours normalisée à $1 \leq m_x < 2$. Ainsi, sa partie entière vaut toujours 1, et on peut donc l'écrire sous la forme :

$$m_x = 1.f_x$$

Avec f_x la partie fractionnaire de cette mantisse normalisée, aussi appelée fraction. Le "1*"* de la partie entière n'a dès lors plus besoin d'être stocké dans la représentation : on parle alors de "1*"* implicite.

Pour pouvoir représenter les nombres positifs et négatifs, on adjoint à la représentation un bit de signe *s*, ce qui donne ainsi :

$$x = (-1)^s \times 1.f_x \times 2^e = (-1)^s \times (1+\sum_{i=0}^{n} f_i \times 2^{-i})$$

Enfin, on notera w_f la taille en bits de la fraction, et w_e celle de l'exposant. Un nombre *x* sera ainsi représenté en matériel comme la concaténation de son bit de signe *s*, de son exposant *e* et de sa fraction f_x, comme indiqué sur la figure 1.1 Cela correspond donc à un vecteur de $(w_f + w_e + 1)$ bits.

Figure 1.1 : Représentation en virgule flottante d'un nombre x

La norme impose au minimum deux formats : la simple précision codée sur 32 bits et la double précision codée sur 64 bits. Elle définit également une précision étendue pour ces deux formats.

Le nombre de bits utilisés pour la simple et la double précision étendue est élargi, avec pour chacun un nombre de bit minimum pour les champs exposant et mantisse, le choix

exact du nombre de bits utilisés étant laissé aux constructeurs. Le tableau 1.1 résume la longueur des différents champs pour chacun de ces formats.

Tableau 1.1 : Caractéristiques des représentations en virgule flottante de la norme IEEE-754

Format	Taille (bits) Mantisse	Taille (bits) Exposant	Biais	Nombres représentés (strictement positifs)	
Simple	1 + 23	8	127	Max : 3.4×10^{38}	Min :1.17×10^{-38}
Simple étendu	≥ 32	≥ 11	≥ 1023	Max : 1.7×10^{308}	Min : 2.2×10^{-308}
Double	1 + 52	11	1023	Max : 1.8×10^{308}	Min : 4.9×10^{-324}
Double étendu	≥ 64	≥ 15	≥16383		
Double étendu (PC)	1 + 64	15	16383	Max :1.2×10^{4932}	Min :2.2×10^{-308}

À partir de ce tableau, on voit bien que la plus grande valeur représentable dans le format double précision est : $(2 - 2^{-53}) \times 2^{1023} \approx 1.797693\ldots \times 10^{308}$. Alors que la plus petite valeur positive est : $2^{-1022} \approx 2.22507385\ldots \times 10^{-308}$.

1.2.2. Les modes d'arrondi

Un nombre machine est un nombre qui peut être représenté exactement dans le système à virgule flottante. En général la somme, le produit et le quotient de deux nombres machine n'est pas forcément un nombre machine et le résultat d'une telle opération doit être arrondi.

La norme IEEE-754 impose la présence de quatre modes d'arrondi : l'arrondi vers zéro, l'arrondi vers $+\infty$, l'arrondi vers $-\infty$ et l'arrondi au plus près.

Soit x, le résultat d'une opération (c.-à-d. la valeur à arrondir), soit x^-, x^+ les deux nombres machine entourant x, soit $x^- \leq x \leq x^+$.

- ***Arrondi vers*** $+\infty$: $\Delta(x)$ c'est l'arrondi vers le plus petit nombre machine plus grand ou égal à x. (c.-à-d. x^+)
- ***Arrondi vers*** $-\infty$: $\nabla(x)$ c'est l'arrondi vers le plus grand nombre machine plus petit ou égal à x. (c.-à-d. x^-)
- ***Arrondi vers zéro*** : Z(x) c'est l'arrondi vers $-\infty$ quand $x > 0$ ou l'arrondi vers $+\infty$ quand $x < 0$.
- ***Arrondi au plus près*** : N(x) c'est l'arrondi vers le nombre machine le plus proche de x.

Ces quatre modes d'arrondi sont illustrés dans l'exemple de la figure 1.2

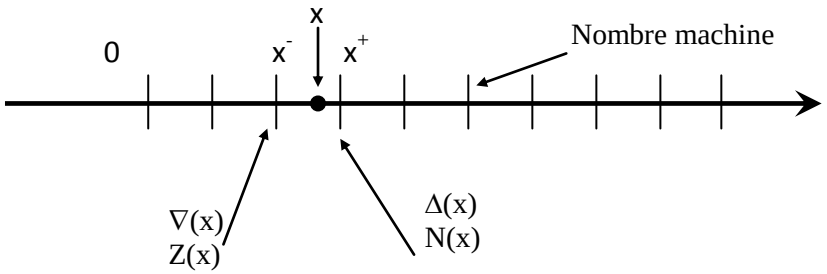


Figure 1.2 : Modes d'arrondis disponibles dans la norme IEEE-754

1.2.3. Nombres spéciaux et exceptions

En plus des nombres qui peuvent être représentés par le format du tableau 1.1. La norme IEEE-754 définit certains nombres appelés nombres spéciaux. Ces nombres sont introduits dans la norme pour la bonne gestion des résultats d'opérations indéterminés (ex : division par zéro), ainsi que les dépassements de capacité. Ces valeurs spéciales sont :

- $\{-\infty, +\infty\}$: Ces valeurs infinies représentent à la fois un dépassement de capacité négatif ou positif et les résultats respectifs des opérations x/-0 et x/+0 où x est un nombre représentable dans la norme.
- $\{\mathbf{+0, -0}\}$: Comme les nombres dans la norme IEEE-754 sont normalisés alors, le bit de poids fort de la mantisse est toujours non nul et égal à un. Ainsi en double précision, le plus petit nombre normalisé positif est $x = 1.0 \times 2^{-1022} \approx 2.225 \times 10^{-308}$. Soit y le nombre normalisé suivant x : $y = (1+2^{-52}) \times 2^{-1022}$. Si on calcule (y-x), la valeur exacte 2^{-1074} n'est pas représentable par un flottant normalisé et est donc arrondi à zéro (en mode d'arrondi au plus prés). Il s'agit donc d'un débordement vers zéro (*underow*). On remarque aussi que ce zéro possède un signe d'où l'existence de deux codages différents pour 0 : $\{+0$ et $-0\}$. C'est aussi la conséquence de la définition de deux valeurs $-\infty$ et $+\infty$ puisque $-\infty/x = -0$ et $+\infty/x = +0$,
- **NaN** (Not-a-Number): Cette valeur est utilisée pour coder les résultats indéterminés comme la racine carrée d'un nombre négatif et les résultats ne peuvent pas être réduits comme 0/0 ou $\infty - \infty$.

1.2.4. La Fonction ulp

La fonction ulp a été introduite pour exprimer la distance entre deux nombres représentables consécutivement au voisinage d'un réel x.

Si x est exactement représentable dans un format à virgule flottante et n'est pas une puissance entière de la base B, le terme ulp (x) (*unit in the last place*) désigne la magnitude du dernier chiffre de la mantisse x.

Si, $x = \pm x_0.x_1x_2 \cdots x_{n-1} \times B^{Ex}$ alors ulp $(x) = B^{Ex-n+1}$.

La fonction ulp est utilisée pour exprimer la précision de la représentation d'un réel x sur machine.

Si x est un réel, représenté par le nombre machine x' (x' normalisé) alors $x' = x + \varepsilon$

- $|\varepsilon| \leq ½$ ulp(x) pour l'arrondi au plus prés
- $|\varepsilon| \leq$ ulp(x) pour les autres modes d'arrondi

Pour davantage de précision, les notions développées dans cette partie sont détaillées dans un ouvrage très intéressant [10].

1.3. Les circuits FPGA

Pour valider les algorithmes et architectures que nous aurons à développer tout au long de cette thèse, nous avons besoin de choisir une cible technologique pour nos implémentations matérielles. Celle-ci peut être un ASIC (*Application Specific Integrated Circuits*), ou un FPGA (*Field-Programmable Gate Array*). Bien que les FPGA sont moins performants que les circuits ASIC néanmoins; ils permettent un bon compromis coût/flexibilité intermédiaire entre le circuit ASIC, très performant, mais très coûteux à concevoir et peu flexible, et le microprocesseur, peu coûteux et très flexible mais moins performant. Les FPGA sont des composants du commerce produits en grande série, donc immédiatement disponibles et relativement peu coûteux. De plus, ils disposent de ressources suffisantes même pour des architectures complexes. Toutes ces raisons en font des FPGAs de bons candidats pour la mise en pratique de nos travaux.

1.3.1. Description générale et domaines d'application

Les FPGA sont des composants électroniques qui comportent un grand nombre de fonctions logiques de base (ET, OU, etc.) que l'utilisateur peut combiner entre elles en fonction des besoins de l'application. Depuis une bonne vingtaine d'années, les FPGA sont couramment utilisés sur les cartes électroniques pour assurer des fonctions traditionnelles de logique câblée, de prototypage et de test de circuits intégrés [11]. La technologie a beaucoup évolué ces derniers temps et actuellement, les FPGA sont devenus de véritables processeurs numériques des signaux, qui viennent concurrencer les composants DSP et

ASIC. Les ASIC ont longtemps été la technologie la mieux adaptée pour réaliser des applications nécessitant des performances élevées dans un encombrement réduit. Cependant ces derniers souffrent des coûts exorbitants de la conception et de l'investissement initial en production. Pour qu'un composant ASIC ait un prix abordable, il faut qu'il soit fabriqué en grande série, de façon à amortir ses investissements initiaux. Du coup, cette technologie est inabordable dans beaucoup de projets. Les FPGA sont par contre beaucoup plus accessibles. Il s'agit en effet de composants standards fabriqués en masse et les coûts initiaux sont répartis sur l'ensemble des utilisateurs. Comme tout composant standard, ils peuvent être achetés en petites quantités. Du fait qu'il s'agit de composants reprogrammables, les FPGA ne présentent pas le risque que l'on a avec les ASIC (où la programmation est figée dans le silicium) et il est plus facile de faire des modifications et des mises à niveaux.

Au cours des dernières années, les FPGA sont devenus l'option favorite pour la mise en œuvre matérielle des systèmes numériques [12], [13], car ils permettent d'adapter le matériel à une application donnée. Ces circuits disposent de ressources matérielles qui ne cessent d'augmenter en quantité et en qualité. La souplesse sans précédent de la technologie FPGA est en fait une approche idéale pour les applications complexes de conception de systèmes embarqués [14], [15], ainsi que pour les applications de traitement de signal [16] et de calcul scientifique à hautes performances [17], où les FPGAs suscitent un engouement de plus en plus important, comme en témoigne le projet de développement d'un super calculateur à base de circuits FPGA [18] par le " *FPGA High Performance Computing Alliance* (FHPCA)*"* en ÉCOSSE [19]. Par le biais de ce projet, l'Écosse veut se placer parmi les leaders mondiaux des nouvelles technologies de calcul.

En plus l'option qui permet de reprogrammer ces circuits nous laisse imaginer dans un avenir proche la réalisation de systèmes monocarte ou à plusieurs cartes d'une même plate-forme pour répondre aux besoins de tous les éléments d'un système, ou même à tous les besoins d'un système entier.

D'une façon générale, un FPGA peut être vu comme une matrice de Blocs Logiques Configurables CLB (*Configurable Logic Block*), où non seulement la logique, mais aussi la connexion est programmable par l'utilisateur. Les spécifications conceptuelles des blocs CLB varient d'un constructeur à un autre et même au sein du même constructeur, d'un circuit à un autre. Un CLB peut être simple comme juste une table LUT (*Look-Up Table*) à quatre entrées ou aussi complexe qu'une unité arithmétique et logique (ALU) à 4 entrées [20]. Il est d'usage de définir la granularité de la logique reconfigurable comme étant la

taille de la plus petite unité fonctionnelle qui peut être traitée par les outils de programmation. Les architectures ayant une granularité plus fine ont tendance à être mieux adaptées pour la manipulation des données au niveau bit et, en général, pour les circuits combinatoires. En revanche, les architectures avec une grosse granularité sont mieux adaptées à des niveaux de manipulation de données plus élevés, par exemple, pour développer des circuits au niveau transfert de registre RTL (*Register Transfer Level*). Le niveau de granularité a un impact considérable sur le temps de configuration. En effet, un dispositif à faible granularité a besoin de plus de points de configuration, produisant ainsi un fichier de configuration (*bitstream*) bien plus grand, dont le routage supplémentaire a un coût inévitable sur la surface occupée et la consommation de puissance. D'autre part, les performances d'une architecture à gros grains tendent à diminuer lors de traitements avec des calculs plus petits que ses granularités.

Les énormes progrès technologiques, de ces dernières années, ont eu un grand impact sur l'industrie des circuits FPGA. Les dispositifs les plus récents fonctionnent avec des horloges internes qui peuvent aller jusqu'à 600 MHz avec une complexité de plus de 10 millions de portes sur une seule puce (circuit FPGA de la famille Virtex-6 de Xilinx) en utilisant une technologie VLSI de 40nm avec une alimentation de 1 volt [21]. Ces améliorations n'ont pas contribué seulement à l'augmentation du nombre de portes logiques, mais aussi dans l'ajout de nombreux nouveaux blocs, des multiplieurs intégrés, des blocs mémoire BRAM, des tranches DSP qui permettent aux utilisateurs de mettre en œuvre un traitement du signal plus complexe, et ce, à des vitesses plus soutenues ou même des microprocesseurs intégrés au sein de la même puce.

Si certains modèles de FPGA sont programmés de manière irréversible grâce à des anti-fusibles comme ceux conçus par la firme Actel, par contre ceux proposés par les firmes Xilinx et Altera sont reconfigurables. Cette technologie permet ainsi de reconfigurer le FPGA autant de fois que nécessaire, soit pour d'éventuelles corrections dans l'architecture ou bien pour les besoins des applications reconfigurables [22].

Pour valider nos algorithmes et architectures, notre choix d'intégration a été porté sur les circuits FPGA de Xilinx, pour son leadership mondial dans le domaine des circuits FPGA, ainsi que pour la disponibilité de l'outil de conception.

La firme Xilinx offre une large gamme de FPGAs dédiés à diverses applications. Les plus onéreux sont les FPGA de la famille des Virtex (Virtex II/pro, Virtex-4, Virtex-5 et Virtex-6). Ces derniers sont très performants grâce à la présence d'un processeur PowerPC 405 *on-chip*, c'est-à-dire directement inclus au sein du FPGA. Ils peuvent dès lors contenir

des systèmes d'exploitation embarqués comme Linux et travailler avec la logique implémentée dans le FPGA.

1.3.2. Métriques de performances

On mesure la qualité d'un circuit à sa vitesse, à la quantité de matériel qu'il occupe et parfois à sa consommation électrique.

- La surface d'un circuit ASIC est la taille en millimètre carré du rectangle de silicium qui le contiendra. Pour un circuit FPGA, la surface occupée est évaluée en termes d'équivalent-portes ou plus exactement en quantité de ressources consommées pour l'implémentation d'une architecture donnée. Les opérateurs arithmétiques, réalisés dans le cadre de ce travail, ont vocation d'être des sous-circuits dans des applications plus conséquentes, d'où la minimisation de la surface est un paramètre important dans le processus d'implémentation de ces architectures. Pour une application complète, la situation est légèrement différente: dans la mesure, où on choisit un circuit FPGA et notre architecture rentre dans le FPGA, ou ne rentre pas. En général, on mesure la surface occupée dans un FPGA par la quantité de ressources utilisées du circuit FPGA. Ces ressources sont principalement des CLB en plus de modules plus élaborés tels que de la mémoire sous forme de blocs (BRAM), de petits multiplieurs $(18 \times 18\, bits)$ ou des tranches DSP. Il est à noter que le routage est aussi une ressource qui, souvent, n'entre pas dans l'estimation de la surface occupée, néanmoins le défaut du routage, lors de l'implémentation d'une architecture, conduit souvent à aller vers un circuit FPGA plus important malgré que les ressources du FPGA sélectionné (en terme de CLB etc …) sont suffisantes.

- La fréquence d'horloge et la quantité d'instructions par seconde sont des critères communs pour mesurer la vitesse, mais elles ne sont pas toujours suffisantes comme mesures de performances d'un circuit. Par exemple, dans un appareil photo numérique, l'utilisateur se soucie plus de la rapidité de son démarrage ou de traitement d'images, plutôt que de la fréquence d'horloge interne ou de la vitesse d'exécution des instructions internes du processeur. Les métriques de performances les plus utiles sont la latence et le débit.

La latence (temps de réponse) mesure le temps entre le début et la fin d'une tâche: dans l'exemple de la caméra, il pourrait être le temps de traitement d'une image, par exemple, 0,25 seconde (caméra A) et 0,20 seconde (caméra B). Le débit est défini comme la quantité des tâches accomplies par seconde, par exemple, pour les caméras A et B, 4 et 5 images par seconde, respectivement. Remarquez que le débit peut être supérieur à l'inverse de la latence grâce à la possibilité du traitement simultané ou du pipelining, par exemple, la caméra A peut traiter 8 images par seconde, au lieu de 4, en captant une nouvelle image tandis que l'image précédente est en traitement. Ce qui nous conduit à dire que le débit est une métrique très utile pour la mesure et la comparaison de performances entre deux circuits.

Les architectures développées dans ce travail n'ont pas vocation à être intégrées dans des applications à faible puissance, d'où le paramètre de consommation de puissance n'est pas pris en compte. Néanmoins, la firme Xilinx propose une panoplie de circuits FPGA dédiés à la conception faible puissance [23].

1.3.3. Architecture des FPGAs de Xilinx

L'architecture de base des circuits FPGA de Xilinx se compose d'un réseau d'éléments logiques de base CLB complété par un réseau maillé d'interconnexions programmables, avec un grand nombre d'entrées/sorties IOB (*Input Output Block*), en plus des blocs mémoires BRAM et des multiplieurs 18×18bits dont disposent tous les circuits (Virtex-2 et plus), d'autres ressources plus complexes et dédiées, comme les tanches DSP et les processeurs embarqués, sont disponibles dans les circuits les plus récents. La structure interne d'un circuit FPGA est illustrée sur la figure 1.3.

La sélection des fonctions logiques, la configuration des interconnexions et la définition des entrées/sorties sont réalisées en téléchargeant un fichier binaire (*BitStream*) de configuration à l'intérieur du FPGA.

Les interconnexions, dans un circuit FPGA, ont un rôle majeur dans le fonctionnement d'un FPGA en raison du besoin de voies de communication rapides et efficaces entre les différents blocs logiques qui sont organisés en matrice ligne et colonne. Les circuits FPGA de Xilinx disposent de trois type de connections : lignes pour connections direct, qui sont destinées au routage de composants voisins tels que (*Carry chain* circuit de la retenue), des lignes de longueur moyenne qui sont dédiées aux routages des CLB et enfin les longues lignes qui sont utilisées pour les signaux globaux.

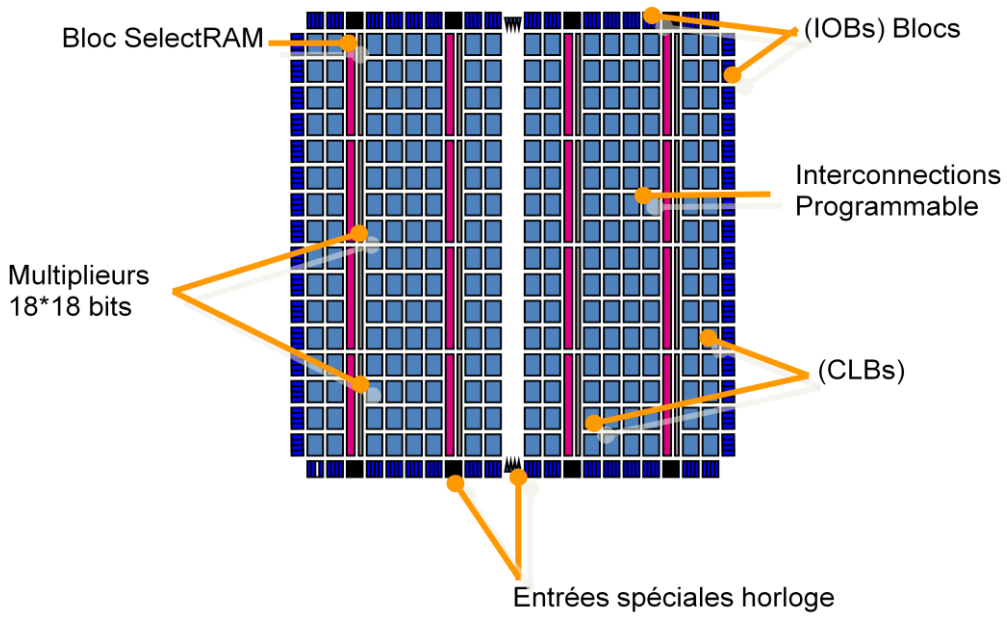


Figure 1.3 : Structure interne d'un FPGA (Virtex-2) [10]

1.3.3.1. Les blocs logiques combinatoires (CLB)

Les blocs logiques configurables (CLB) sont la ressource principale de mise en œuvre de la logique séquentielle ainsi que des circuits combinatoires. Chaque élément CLB est connecté à la matrice d'interconnexion pour accéder au routage général et comprend quatre tranches (*slice*) qui sont interconnectées (figure 1.4) tirée de [21]. Ces slices sont regroupées par paires. Chaque paire est organisée en colonne. SLICEM indique les deux slices dans la colonne de gauche, et SLICEL désigne la paire de slices dans la colonne de droite. Chaque paire dans une colonne a un chemin de retenue *(carry chain)* indépendant; toutefois, seules les slices dans SLICEM ont une chaîne de décalage commune.

Les slices, dans la matrice des CLB, sont identifiés par leur position ligne-colonne (X et Y). Cette identification est souvent nécessaire lors du placement, avec contraintes, d'une logique dans un FPGA.

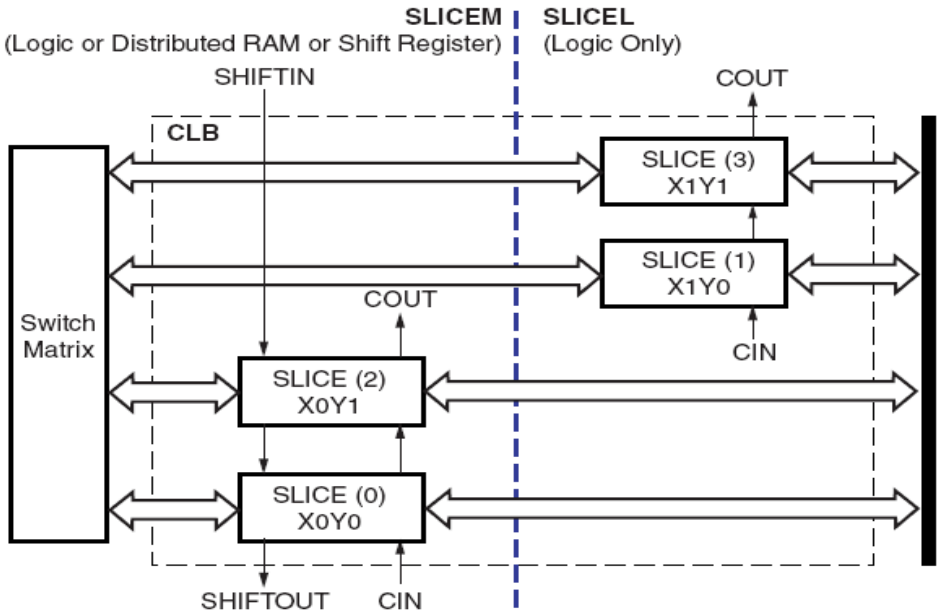


Figure 1.4 : Arrangement des Slices dans un CLB (Virtex-4)

Les éléments communs aux deux slices (SLICEM et SLICEL) sont deux générateurs de fonctions logiques (ou tables *Look-Up Table*), deux éléments de stockage, une fonction de multiplexage et un chemin pour la propagation de la retenue. Ces éléments sont utilisés par les deux slices, SLICEM et SLICEL, pour l'implémentation de la logique, de l'arithmétique et des mémoires ROM. SLICEM prend en charge deux fonctions supplémentaires: stockage de données en utilisant la mémoire distribuée et permet aussi de faire des décalages de données avec des registres de 16 bits. La figure 1.5 illustre l'architecture simplifiée d'un slice [20].

1.3.3.2. Les multiplieurs 18×18 bits

Pour permettre l'implémentation efficace de circuits très calculatoires et massivement parallèles, comme on en trouve en traitement du signal ou en traitement d'images, les générations actuelles de FPGA embarquent toutes de petits multiplieurs entiers. Ces multiplieurs, accessibles par la matrice de routage, permettent donc d'accélérer le calcul de la multiplication, sans toutefois utiliser les ressources CLB du circuit FPGA.
Les familles (Virtex-2 et plus) de Xilinx proposent ainsi des multiplieurs 18×18 bits. Ces multiplieurs partagent les mêmes ressources de routage que les blocs BRAM de 18 Kbits : leur utilisation est donc mutuellement exclusive.

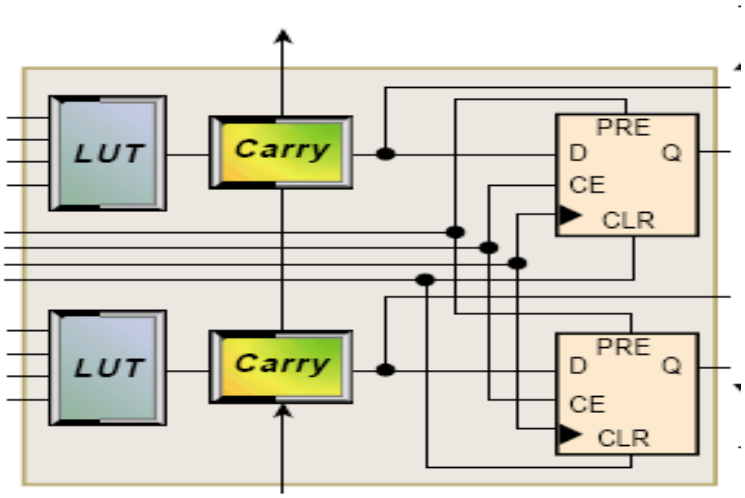


Figure 1.5 : Architecture simplifiée d'un slice

1.3.3.3. Les blocs mémoires BRAM

Les FPGA de Xilinx intègrent plusieurs blocs mémoires BRAM (block select RAM). Ces derniers sont organisés en colonnes le long de la puce du FPGA (Figure 1.3). Le nombre de blocs, qui peut aller de 8 à plus de 1000, dépend de la famille et de la taille du circuit. Chaque bloc peut stocker 18 kbits de données. Lire et écrire sont des opérations synchrones, les deux ports dont dispose ce bloc BRAM sont symétriques et totalement indépendants et ne partagent que les données stockées. Les largeurs des données des deux ports peuvent être configurées de manière indépendante. Chaque port peut être configuré selon plusieurs configurations de 16k×1 bits jusqu'à 512×36 bits. Le contenu de la mémoire peut être défini par la configuration binaire. Ces blocs BRAM peuvent être cascadés pour intégrer des mémoires plus larges.

1.3.3.4. Arbres de distribution d'horloge

Devant le nombre grandissant de cellules logiques et donc de registres embarqués dans les FPGAs, le signal d'horloge contrôlant ces registres doit être capable de supporter une sortance (fanout) très importante tout en assurant une parfaite synchronisation sur tout le circuit. Ce qui est impossible à réaliser en utilisant la matrice de routage. Les FPGA possèdent donc des arbres de routage dédiés à la distribution du signal d'horloge DMC (*Digital Clock Managers).*

On trouve généralement plusieurs arbres de distribution sur un seul FPGA, pour permettre d'avoir différentes horloges sur un même circuit [21].

1.3.3.5. Tranches DSP

Les générations les plus récentes de FPGA proposent même des blocs DSP (*Digital Signal Processing*) dédiés, comme leur nom l'indique, au traitement du signal. Ces blocs sont généralement composés d'un multiplieur et d'un accumulateur MAC (*Multiplier and ACcumulator)* et permettent ainsi de réaliser des filtres de manière très efficace.
On trouve ce genre de blocs DSP dans les FPGAs (Virtex-4, Virtex-5 et Virtex-6) de Xilinx. Par contre, les Virtex-2 n'en possèdent pas.

1.3.3.6. Cœurs de processeurs RISC

Devant l'utilisation croissante des FPGAs pour l'implémentation de systèmes embarqués comme les SoC ou les NoC (pour System-on-Chip et Network-on-Chip respectivement), certains modèles de FPGAs intègrent directement un ou plusieurs cœurs complets de processeurs RISC. Ainsi, les Virtex-2 Pro et Virtex-4, Virtex-5 et Virtex-6 de Xilinx embarquent jusqu'à quatre cœurs de PowerPC 405. Pour les FPGAs qui sont dépourvus de tels processeurs, tels que les Virtex-2, les constructeurs proposent des solutions softcores, comme le MicroBlaze de Xilinx, véritables processeurs implémentés dans la matrice de cellules logiques CLB.

1.4. Programmation ou configuration des FPGAs

Bien que chaque fabricant ait ses propres outils de développement, l'implémentation d'un circuit logique à l'aide d'un FPGA consiste en plusieurs étapes, qui sont pratiquement communes à tous les FPGAs comme le montre la figure 1.6.
Néanmoins, dans cette section, nous allons détailler plus particulièrement la conception et la programmation des circuits FPGA par le flot de ***Foundation ISE*** de Xilinx (figure 1. 7).
Il est important de noter qu'une grande partie est commune avec le flot de conception des circuits ASIC. Cela signifie ainsi que tous les opérateurs présentés dans cette thèse, bien qu'ils ciblent les circuits FPGA, peuvent aussi bien être implémentés sur ASIC, au prix d'un simple reciblage technologique.

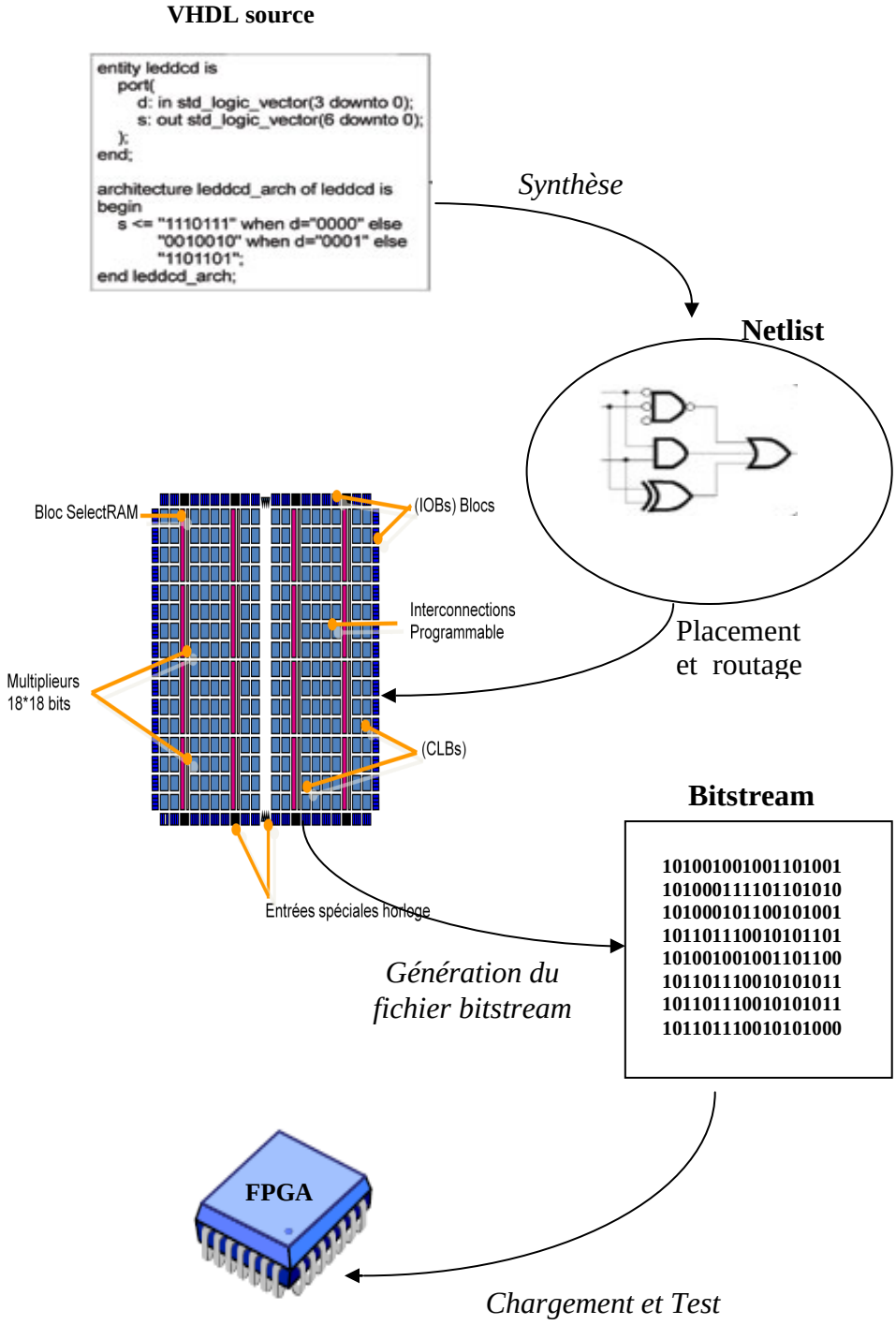


Figure 1.6 : Etapes de programmation d'un FPGA

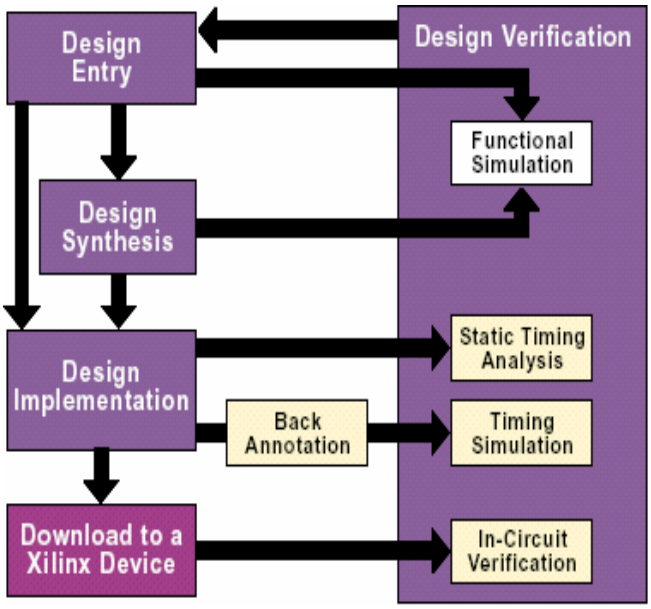


Figure 1.7 : Flot de conception ***Foundation ISE*** [24]

1.4.1. Description du circuit ***(Design entry)***

Le cycle de conception des circuits FPGA commence par une description du circuit. Celle-ci peut être faite à travers un éditeur graphique, textuel ou même les deux. Les circuits présentés dans ce document sont des implémentations directes d'algorithmes. Cependant, si ces algorithmes peuvent facilement être décrits grâce à n'importe quel langage de programmation classique, cela n'est pas du tout le cas de ces circuits. En effet, l'implémentation matérielle d'un algorithme fait appel à des notions temporelles très fortes, comme la concurrence et la synchronisation, qui ne sont pas naturelles dans les langages classiques. Il faut donc pour réaliser un circuit recourir à un langage de description de matériel ou HDL (*Hardware Description Language*), qui sera capable d'exprimer explicitement ces notions temporelles.

Il existe de nombreux HDL, parmi lesquels on trouve VHDL, Verilog, System-C et Handel-C pour les plus connus et utilisés. Si System-C et Handel-C sont plus orientés vers la modélisation de systèmes complexes composés de plusieurs processus concurrents, VHDL et Verilog sont quant à eux bien adaptés à la description d'opérateurs arithmétiques comme ceux présentés dans ce document, ainsi qu'à leur intégration dans des circuits plus importants. Nous avons ainsi choisi pour nos opérateurs d'utiliser le langage VHDL.

VHDL est un langage de description matériel destiné à représenter le comportement ainsi que l'architecture d'un système électronique numérique. Son nom complet est (***VHSIC Hardware Description Language)***. Ce dernier a été commandé par le (**Dod)** Département de la Défense des États-Unis dans le cadre de l'initiative VHSIC (***Very-High-Speed Integrated Circuit)***. Dans un effort de rationalisation, le VHDL reprend la même syntaxe que celle utilisée par le langage Ada (ce dernier étant aussi développé par le département de la défense).

Le VHDL était à l'origine destiné à documenter de manière formelle la structure et le comportement de circuits intégrés. Ce n'est que par la suite que furent développés d'abord des simulateurs logiciels, puis enfin des synthétiseurs pour ce langage. VHDL fut décrit par le standard IEEE-1076 en 1987, puis révisé successivement en 1993 et 2000. Il est désormais l'un des HDL les plus utilisés, tant par l'industrie que par le monde académique.

En VHDL, l'entité de base est le composant (aussi appelé entité). Un composant est représenté par son interface extérieure, qui définit ses ports d'entrée et de sortie, ainsi que son architecture interne. L'architecture d'un composant définit à son tour les sous-composants qui la constituent et les fils qui les relient. Un circuit est alors décrit comme un composant, lui-même constitué d'autres composants, et ainsi de suite jusqu'à arriver aux composants de base.

Cette représentation hiérarchique permet d'ajuster le niveau de détails nécessaires à la description d'un circuit. Pour la simulation d'un système complet, on peut ainsi se contenter d'une description de haut niveau, où les composants sont décrits par leurs comportements, c'est-à-dire leurs réponses aux stimulations extérieures. De l'autre côté du spectre, lorsque l'on souhaite représenter un circuit synthétisable, on peut détailler bien plus la description, en descendant jusqu'aux portes logiques. Cette description structurelle permet ainsi d'avoir un meilleur contrôle sur l'implémentation finale du circuit, et donc sur d'éventuelles optimisations de ce circuit.

1.4.2. Simulation fonctionnelle ***(Functional Simulation)***

La simulation fonctionnelle est la première étape dans la conception d'un circuit. Néanmoins, la compilation de la description VHDL s'impose pour parer à toute erreur dans la syntaxe ou dans la cohérence de la description. Cette opération peut être réalisée lors de la synthèse ou simplement en enregistrant le fichier VHDL de la description du circuit.

Dans cette étape on s'assure que notre circuit fonctionne correctement selon le cahier de charges avant de le configurer sur le FPGA. C'est une simulation initiale qui met en relief

le comportement idéal du circuit sans tenir compte des délais dus au routage entre les différentes cellules qui constituent le circuit et des délais même de ces cellules. Elle permet donc de vérifier uniquement la validité du circuit par rapport à la spécification d'un point de vue fonctionnel et non d'un point de vue temporel. L'outil de simulation permet de déboguer et contrôler chaque variable ou signal dans le circuit. La simulation nécessite toutefois des données d'entrée externes, on parle souvent de vecteurs de test, *testbench* ou stimulus. Le simulateur va ensuite générer des signaux représentant la réponse du circuit aux données d'entrées contenues dans chacun des vecteurs de test par le fonctionnement réel du circuit permettant ainsi de visualiser la forme des signaux et les valeurs attribuées aux variables du circuit. Dans le cas des opérateurs arithmétiques présentés dans ce document, cela revient donc à vérifier la justesse des résultats calculés.

Bien entendu, la simulation fonctionnelle ne garantit pas une validation totale du circuit. Mis à part pour de petits opérateurs (avec une petite taille d'opérandes), il est en effet impossible de réaliser une simulation exhaustive à cause de la lenteur de la simulation logicielle. Il faut donc essayer de choisir des vecteurs de test adaptés à l'opérateur pour couvrir le mieux possible les différents cas de figure.

Dans le flow de Xilinx, on dispose de deux simulateurs, le premier est propre à la société Xilinx qui est le Xilinx ISE simulator. Celui-ci est moins performant que le simulateur ModelSim développé par ModelTech. Pour nos opérateurs, nous avons utilisé l'édition de ModelSim XE.

1.4.3. La synthèse ***(Design synthesis)***

La deuxième étape dans le processus de programmation d'un circuit FPGA est la synthèse. Celle-ci consiste à lire la description VHDL du circuit pour en extraire toute la structure logique. Lorsque des composants de ce circuit sont décrits de manière comportementale, c'est le rôle du synthétiseur d'inférer la logique nécessaire à la réalisation des comportements spécifiés. Cette tâche est d'ailleurs vraisemblablement très difficile d'où l'existence du VHDL synthétisable et du VHDL non synthétisable. D'ailleurs de plus en plus, on assiste à l'émergence d'outil de synthèse très sophistiqué qui réduit d'avantage la part du VHDL non synthétisable.

Un autre rôle du synthétiseur est d'appliquer diverses optimisations logiques au circuit. Il cherche ainsi à minimiser toutes les expressions logiques utilisées par ce circuit.

Enfin, le synthétiseur produit une *netlist*, c'est-à-dire une liste de fils et de portes logiques.

Dans une certaine mesure, les outils de synthèse associés aux langages de description de matériel comme VHDL et Verilog sont extrêmement performants lorsqu'il s'agit de compiler une description de matériel vers une technologie donnée. Néanmoins pour les circuits développés dans ce travail, on a eu recours à certaines techniques de placement, d'utilisations des ressources du CLB au plus bas niveau pour des finalités d'optimisations et de la surface et du délai.

Dans cette étape nous avons utilisé l'outil de synthèse de Xilinx disponible dans le flot ISE 7.1 qui est le XST (Xilinx Synthesis Tool)

1.4.4. Implémentation ***(Design Implementation)***

Après l'étape de synthèse c'est l'implémentation. Celle-ci consiste, en premier lieu, à adapter la *netlist*, générée lors de la synthèse, aux ressources matérielles propres du FPGA ciblé. Cette phase de câblage technologique permet ainsi de passer d'une description structurelle générique à une description spécialisée pour une architecture précise. Les éléments de base ne sont alors plus de simples portes logiques mais les cellules programmables du FPGA. Nous savons qu'un FPGA consiste en des blocs logiques reconfigurables (CLB). Ceux-ci peuvent être encore décomposés en LUTs qui effectuent des opérations logiques. Les CLBs et les LUTs sont entrelacés avec des différentes ressources d'interconnexion, BRAM et Multiplieurs 18×18bits.

Puis vient la phase du placement et du routage, après identification des ressources logiques nécessaires à l'implémentation d'un circuit, la tâche de placement et routage consiste alors à disposer et connecter ces ressources sur la surface du FPGA. Il est possible de donner au placeur/routeur des contraintes spatiales ou temporelles, pour fixer par exemple la position des ports d'entrée/sortie du circuit ou bien une période d'horloge maximale.

À l'issue de cette phase de placement et de routage, le circuit obtenu est tel qu'il sera programmé sur le FPGA. C'est sur ce modèle que sont calculées les estimations de surface et de latence du circuit.

Le Timing Analyser (Analyseur temporel) nous délivre un rapport sur l'analyse temporelle effectuée sur notre architecture. Dans ce rapport sont énumérés tous les chemins critiques avec leurs délais respectifs. A partir de ces données, on peut faire des optimisations sur ces chemins critiques dans le but d'optimiser le délai total de l'architecture et aussi dimensionner l'horloge de notre circuit.

1.4.5. La Simulation temporelle ***(Timming Simulation)***

Après l'implémentation, une autre vérification s'impose pour valider le fonctionnement dynamique de notre circuit. Comme mentionné plus haut, la simulation fonctionnelle ne prend pas en considération les délais des composants et des interconnections. La simulation temporelle est utilisée pour vérifier le fonctionnement du circuit avec les délais des composants et des interconnexions introduits après l'étape d'implémentation. Souvent dans cette étape, on utilise les mêmes vecteurs de test pour vérifier que la fonctionnalité n'a pas été modifiée par l'introduction des délais de propagation et reste conforme au cahier des charges.

1.4.6. La Configuration ou Programmation ***(Download to a Xilinx Device)***

Enfin, la dernière étape avant la programmation effective du FPGA consiste à générer un *BitStream*. Ce *BitStream* est en fait un fichier contenant tous les bits de configuration du FPGA, construit pour correspondre parfaitement au circuit décrit lors du placement/routage.

Une fois le *BitStream* créé, il ne reste plus qu'à le télécharger dans le FPGA grâce à une interface dédiée. Le FPGA restera ainsi configuré tant qu'il sera sous tension, ou bien lorsqu'un autre circuit y sera programmé suivant le même procédé.

Nous avons utilisé pour les travaux présentés ici la version 7.1.i d'ISE, l'environnement de développement fourni par Xilinx. Cette suite logicielle inclut notamment le synthétiseur XST, ainsi que tous les outils évoqués précédemment.

Après cette dernière étape, de programmation du circuit FPGA, une simulation réelle est envisageable pour un test exhaustif. Néanmoins celle-ci ne peut être effectuée seulement avec l'outil ISE, mais par l'utilisation d'une carte de développement pour FPGA, munie de ports d'entrée/sortie rapides.

CHAPITRE 2

La division

2.1 Introduction

L'évaluation rapide et précise de la division et d'autres opérations telles que, la racine carrée et l'inverse est un élément important dans de nombreuses applications à des fins particulières, telles que l'informatique graphique et le calcul scientifique [25], [26] etc. Par ailleurs la division devient de plus en plus une opération importante dans les unités de calcul à virgule flottante, ceci n'est pas pour le recours fréquent à cette opération mais à la latence qu'elle présente dans ces unités de calcul.

Elle est l'opération la plus difficile et la plus longue à exécuter parmi les quatre opérations arithmétiques de base et heureusement qu'elle est la moins employée. On estime que dans la plupart des applications courantes scientifiques ou de gestion, les divisions sont dix fois moins fréquentes que les additions/soustractions ou multiplications [27].

En traitement de signal par exemple, les résultats d'Oberman [28], illustrés dans l'histogramme représenté sur la figure 2.1 révèlent : un recours à l'addition de (25%), à la multiplication de (36%), à la soustraction de (16%) alors que seulement (3%) pour la division et le reste est alloué aux autres opérations moins courantes.

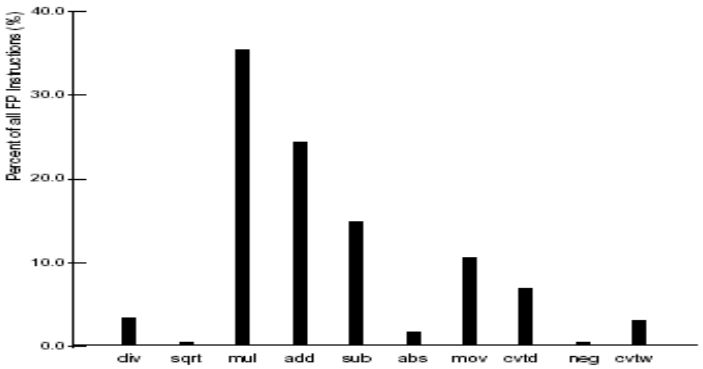


Figure 2.1: Répartition des pourcentages d'utilisation des instructions arithmétiques dans une unité de traitement de signal [28].

Bien que la division soit une instruction relativement sans importance, avec environ 3% de toutes les instructions à virgule flottante, cependant, en termes de latence, celle-ci joue un rôle plus important et elle représente à elle seule 40% du délai totale d'exécution des instructions comme le montre l'histogramme de la figure 2.2.

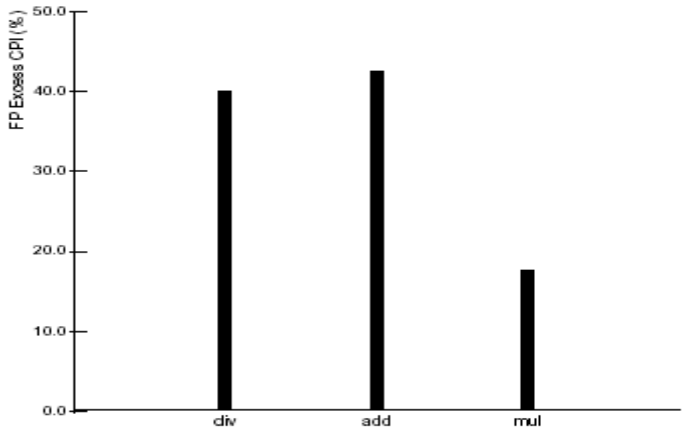


Figure 2.2 : Délai de calcul des opérations arithmétiques
dans une unité de calcul [28].

En termes de nombres de cycles, la gamme de la latence pour l'addition est de 2 à 4 cycles, alors que celle de la multiplication est de 2 à 8 cycles. En revanche, la latence pour la division en double précision dans une FPU (*Floating Point Unit*) où une GPU (*Graphic Processing Unit)*, incrustées dans les cartes graphiques modernes, s'étend de 8 à plus de 60 cycles.

De ce fait, elle occupe presque le même temps de calcul que l'addition dans une unité de calcul (malgré leurs grandes différences de fréquence d'utilisation) qui est de 42% pour l'addition, de 40% pour la division et de 18% pour la multiplication.

Donc la division est une opération de basse fréquence et de latence très élevée dans les systèmes de calcul. A cet effet, il faut la traiter au même ordre que les autres opérations arithmétiques. Pour effectuer cette opération, il y a principalement trois classes d'algorithmes de division :

- Ceux utilisant des additions/soustractions et des décalages.
- Ceux basés sur l'utilisation, tout comme pour la multiplication, des réseaux cellulaires.
- Ceux qui font appel à des méthodes itératives et à l'utilisation des multiplieurs rapides.

Dans ce chapitre, On s'intéresse aux algorithmes dits à ***récurrence de chiffre*** qui représentent la classe de méthodes la plus utilisée pour le calcul de la division. Ces derniers sont des algorithmes itératifs et utilisent l'addition comme opération de base. À chaque itération, on obtient un chiffre du résultat (qui correspond à un ou plusieurs bits), en partant des chiffres ayant le poids le plus fort. Ces algorithmes sont similaires aux méthodes « à la main » pour la division et l'extraction de la racine carrée.

Le calcul de la division ***x/y*** de deux nombres flottants représentés en double précision de la norme IEEE-754 se fait comme suit :

Soit $x = (-1)^{s_x} \times X \times 2^{e_x}$ et $y = (-1)^{s_y} \times Y \times 2^{e_y}$

Où *X* et *Y* sont les mantisses normalisées respectives des deux nombres *x et y* avec :

$$1 \leq X, Y < 2$$

$$\frac{x}{y} = ((-1)^{s_x} \times X \times 2^{e_x}) / ((-1)^{s_y} \times Y \times 2^{e_y}) = X/Y \times (-1)^{s_x - s_y} \times 2^{e_x - e_y}$$

La division de deux nombres flottants se reporte alors à faire la division des deux mantisses, le signe et l'exposant du quotient sont obtenus respectivement en utilisant un simple porte XOR et une soustraction.

Dans ce qui suit, nous allons nous focaliser sur le calcul de la division de deux nombres qui sont supposés être les mantisses de deux nombres flottants. Soit $Q = X/Y$

Les algorithmes étudiés dans ce chapitre sont dits à récurrence de chiffres. Le résultat est représenté en base β et un chiffre de celui-ci est obtenu à chaque itération. De nombreux paramètres sont impliqués dans le calcul de ce chiffre résultat et la qualité d'une solution dépend des contraintes particulières à chaque implémentation matérielle, ce qui rend l'espace de recherche immense.

Dans le cas de la division, X est le dividende et Y le diviseur, Q le quotient et R le reste.

L'opération de division est définie par :

$$X = Y \times Q + R \qquad (2.1)$$

Et $$|R| < |Y| \times ulp, \quad sign(R) = sign(X) \qquad (2.2)$$

On peut avoir deux types de division :

- Division qui produit un quotient Q entier, dans ce cas $l'ulp = 1$
- Division qui produit un quotient fractionnaire : $l'ulp = \beta^{-n}$ (pour un quotient Q sur n chiffres, représenté dans une base β)

Dans ce travail on s'intéresse au deuxième cas, où tous les chiffres du quotient résultat sont fractionnaires. Comme notre opération s'effectue sur des mantisses de deux nombres

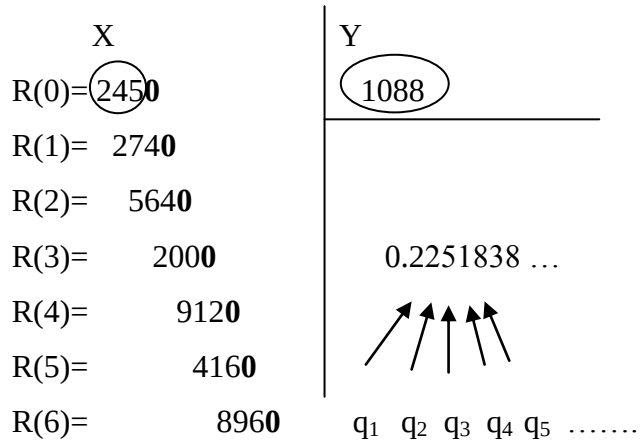
flottants, une autre étape de normalisation, qui rend le diviseur plus petit que le dividende, est nécessaire ($X < Y$ et $1/2 \leq Y < 1$). Cette étape de normalisation est effectuée par de simples décalages.

Dans ce qui suit, nous allons présenter une chronologie d'algorithmes pour le calcul de la division pour arriver en fin à l'algorithme sur lequel nous avons fondé nos travaux.

2.2. Division « à la main »

Dans la division apprise à l'école on travaille en représentation avec une base 10 ($\beta = 10$) et des chiffres pris dans l'ensemble $\{0, 1, \ldots, 9\}$

Exemple 2.1 On effectue la division de X = 245 par Y = 1088



Les 0 en gras résultent d'un décalage dans la base (une multiplication par 10).

On pose le reste partiel initial R(0) = X. On trouve ensuite un entier $q_1 = 2$, tel que $R(1) = 10{\times}R(0) - q_1{\times}Y$, reste positif et soit strictement inférieur à Y. Pour toutes les étapes $0 < j \leq n$, on doit choisir un chiffre q_j aussi grand que possible mais qui respecte la condition : $10{\times}R(j) \geq q_{j+1}{\times}Y$. C'est à dire que l'on veut que $R(j+1) = 10{\times}R(j) - q_{j+1}{\times}Y$ soit toujours compris dans l'intervalle $[0, Y[$.

La sélection du chiffre q_{j+1} demande donc à comparer le reste partiel décalé $10{\times}R(j)$ aux valeurs $Y, 2Y, \ldots$, jusqu'à $9Y$. Ces comparaisons peuvent se faire en parallèle, mais elles sont coûteuses en matériel, puisque chacune demande une soustraction à propagation de la retenue entre le reste partiel et un multiple du diviseur.

C'est d'ailleurs de là que vient la difficulté de la division à la main : « deviner » le bon chiffre du quotient à chaque itération.

2.3. Division restaurante

Il est possible de diminuer le nombre de comparaisons en diminuant la magnitude de la base de numération. En base $\beta = 2$ avec les chiffres $\{0, 1\}$, on a besoin que d'une seule comparaison (donc une seule soustraction) pour la sélection d'un chiffre q_{j+1}. L'algorithme correspondant est appelé, algorithme de division restaurante. À chaque itération, on soustrait le diviseur au reste partiel multiplié par la base (cette multiplication par 2 consiste en fait en un simple décalage, on ajoute alors un zéro). Le résultat devient le nouveau résultat partiel. S'il est positif ou nul, le nouveau chiffre du quotient est 1, sinon le nouveau chiffre devient 0 et on restaure la valeur du reste partiel (en lui ajoutant le diviseur Y). Dans une implémentation matérielle, cette restauration n'est pas effectuée par une véritable addition mais en stockant la valeur $2R(j)$ au lieu de $R(j)$. Ceci permet de remplacer une addition dans le chemin critique par un simple multiplexeur.

La sélection d'un chiffre q_{j+1} de Q se fait à chaque itération en comparant $2R(j)$ à Y. Cette comparaison est réalisée par une soustraction à propagation de la retenue.

$$q_{j+1} = \text{Sel}(\text{R(j)}, \text{Y}) = \begin{cases} 1 & si\ (2R(\text{j}) - \text{Y}) \geq 0 \\ 0 & sinon \end{cases} \qquad (2.3)$$

L'algorithme de la division restaurante est donné comme suit :

Algorithme 1 : Division restaurante.

```
R(0) ← X
pour j = 0 à n faire
        si (2R(j) − Y) ≥ 0 alors
           R(j+1) ← 2R(j) − Y
           q_{j+1} ← 1
        sinon
           R(j+1) ← 2R(j)
           q_{j+1} ← 0
```

Dans cette méthode le délai d'exécution d'une itération est celui de deux opérations de soustraction. L'erreur commise sur le résultat est borné par la valeur d'une unité du dernier chiffre $1ulp$.

2.4. Division non restaurante

Il est possible d'améliorer le délai d'exécution de la division restaurante en intégrant l'étape de restauration dans l'itération suivante. L'algorithme résultant est appelé

algorithme de division non restaurante. Il utilise l'ensemble de chiffres $\{-1, +1\}$ (c'est-à-dire que le quotient Q sera obtenu en représentation redondante) pour effectuer en même temps la mise à jour du reste partiel $R(j)$ et la sélection du chiffre du quotient q_{j+1}.

Dans cette version de division, la première comparaison ne nécessite pas une opération de soustraction mais seulement un test du signe qui est réalisé par une simple porte logique. Ceci résulte en une diminution du délai du chemin critique.

La fonction de sélection demande toujours une comparaison, celle de $R(j)$ avec Y. Cette opération n'est pas coûteuse lorsque le reste partiel est dans une représentation non redondante, puisqu'il suffit de tester son bit de signe. Ce dernier est réalisé par une simple porte logique.

$$q_{j+1} = \text{Sel}(\text{R(j)}, \text{Y}) = \begin{cases} 1 & si\ R(j) \geq 0 \\ -1 & si\ R(j) < 0 \end{cases} \qquad (2.4)$$

Algorithme 2 : Division non restaurante.

R(0) ← X
pour j = 0 à n faire
    si R(j) ≥ 0 alors
        R(j+1) ← 2R(j) − Y
        q_{j+1} ← +1
    sinon
        R(j) ← 2R(j) + Y
        q_{j+1} ← -1

Dans cette méthode le délai du chemin critique diminue, néanmoins le quotient résultat est obtenu dans une représentation redondante où sa conversion nécessite une opération supplémentaire à la fin des calculs. Celle-ci est une soustraction ou une conversion à la volée (*On-the-Fly-Conversion*) [29].

2.5. La division SRT

L'algorithme de la division SRT doit son nom aux initiales de ses trois inventeurs, D.W. Sweeney, J.E. Robertson, et K.D. Tacher, qui semblent l'avoir découvert indépendamment à la même époque, vers la fin des années 50 [30, 31].

La division SRT vient améliorer la vitesse de calcul de l'itération de la division non restaurante, en substituant l'opération de comparaison pour la sélection du chiffre q_{j+1} (qui est en fait une soustraction avec propagation de la retenue sur une longueur qui équivaut à

la taille des opérandes) par une simple lecture de table. Ceci a été rendu possible grâce à l'utilisation de la représentation redondante pour les chiffres du quotient résultat.

Dans ce qui suit, nous allons expliquer les fondements théoriques de la méthode SRT qui ont fait d'elle la méthode la plus implémentée dans les processeurs actuels.
Le résultat de la division SRT est obtenu chiffre après chiffre et à la j^{ème} itération le quotient cumulé est noté par q[j] qui représente les j premiers chiffres de Q.

$$q[j] = \sum_{i=1}^{j} q_i \times \beta^{-i} \qquad (2.5)$$

Le quotient final est $Q = q[n]$, où n est le nombre d'itérations nécessaires pour atteindre la précision de la sortie sur laquelle est lu le résultat.

$$Q = q[n] = \sum_{i=1}^{n} q_i \times \beta^{-n} \qquad (2.6)$$

2.5.1. Représentation du quotient

Le choix de la représentation des chiffres du quotient Q est crucial, dans la mesure où cette représentation a une grande influence sur la manière dont est calculée la division. La représentation la plus utilisée est la représentation redondante [32], où les chiffres q_{j+1} ne sont pas pris seulement dans l'ensemble $\{0, 1, 2, \dots, \beta - 1\}$ mais dans $\{-\beta+1, \dots, -2, -1, 0, 1, 2, \dots, \beta-1\}$. On travaille souvent sur un ensemble symétrique de chiffres consécutifs $\{-a, -a+1 \dots, -2, -1, 0, 1, 2, \dots, a-1, a\}$. Pour être redondante, cette représentation doit satisfaire la relation suivante :

$$\frac{\beta}{2} \leq a \leq \beta - 1 \qquad (2.7)$$

On définit alors le facteur de redondance ρ, qui vaut :

$$\frac{1}{2} \leq \rho = \frac{a}{\beta-1} \leq 1 \qquad (2.8)$$

Dans ce système de représentation, on utilise des chiffres négatifs et pour éviter toutes confusions entre les chiffres négatifs et le signe de la soustraction, on note les chiffres négatifs en les surmontant d'une barre($-5 = \bar{5}$). Un système de représentation redondant est caractérisé par la notation $(\beta - a)$

Exemple 2.1 : (6–3) représente le système de numération redondante de base 6 qui utilise les chiffres de l'ensemble $\{\bar{3}, \bar{2}, \bar{1}, 0, 1, 2, 3\}$, avec un facteur de redondance $\rho = 3/5$.
L'utilisation de la représentation redondante dans la méthode SRT autorise une certaine "erreur" lors de la sélection du chiffre q_{j+1}, qui peut être corrigée lors des prochaines itérations. Ceci est exploité avantageusement pour simplifier la fonction de sélection. Le choix d'un chiffre ne se fait plus par une comparaison exacte, mais en utilisant des

estimations du reste partiel $R(j)$ et du diviseur d. En pratique, la comparaison des estimations peut s'implanter à l'aide d'une table qui sera plus petite et plus rapide que des comparateurs.

L'erreur commise dans le calcul du quotient $Q = X/d$ est de 1 *ulp*, comme on l'a indiqué précédemment dans ce chapitre.

$$|\varepsilon_n| = |\frac{X}{d} - Q| < \beta^{-n}$$

Où ε_n est l'erreur finale du résultat de la division

En plus, il faut qu'à chaque itération le résultat converge vers cette erreur finale ε_n. D'où l'erreur à chaque itération doit être :

$$|\varepsilon_j| = |\frac{X}{d} - q[j]| \leq |\varepsilon_n| + \sum_{i=j+1}^{n} max\ q_i \times \beta^{-i} = |\varepsilon_n| + \frac{a}{\beta-1}\left(\beta^{-j} - \beta^{-n}\right) \quad (2.9)$$

En remplaçant ε_n par sa valeur extrême dans l'expression (2.9), celle-ci devient alors :

$$\left|\varepsilon_j\right| \leq \beta^{-n} + \rho \times \left(\beta^{-j} - \beta^{-n}\right) = \rho \times \beta^{-j} + (1-\rho)\beta^{-n} \quad (2.10)$$

Cette dernière expression n'est pas applicable dans cette forme là, comme mesure de la précision intermédiaire $\left|\varepsilon_j\right|$. Une simplification s'impose, sans toutefois toucher à la précision finale qui est $|\varepsilon_n| \leq \beta^{-n}$. Celle-ci est comme suit :

$$\left|\varepsilon_j\right| = |\frac{X}{d} - q[j]| \leq \rho \times \beta^{-j} \quad (2.11)$$

On remarque aisément que si l'erreur $\left|\varepsilon_j\right|$ verifie l'equation (2.11), elle vérifie forcement l'équation (2.10).

D'où : $$|X - d \times q[j]| \leq \rho \times d \times \beta^{-j} \quad (2.12)$$

L'équation (2.12) montre que le reste de la division est borné. On définit alors un nouveau reste $R(j)$ tel que :

$$R(j) = \beta^{j} \times (X - d.q[j]) \quad (2.13)$$

On remplace (2.12) dans (2.13) et on aura :

$$|R(j)| \leq \rho \times d \quad (2.14)$$

Pour obtenir l'équation de récurrence de la division SRT on calcule :

$$\begin{aligned} R(j+1) - \beta \times R(j) &= \beta^{j+1} \times (X - d \times q[j+1]) - \beta \times \beta^{j} \times (X - d.q[j]) \\ &= \beta^{j+1} \times (q[j] - q[j+1]) \times d \end{aligned} \quad (2.15)$$

De l'équation (2.5), $q[j+1] = q[j] + q_{j+1} \times \beta^{-(j+1)}$, l'équation de récurrence devient alors :

$$R(j+1) = \beta \times R(j) - d \times q_{j+1} \quad (2.16)$$

Les valeurs initiales sont obtenues pour j=0,

$$R[0] = X \; et \; q_0 = 0 \qquad (2.17)$$

Alors que la récurrence est réalisée de telle sorte que $R(j+1)$ est borné (équation 2.14), et le chiffre q_{j+1} du quotient est obtenu par le biais de la fonction de sélection suivante :

$$q_{j+1} = Sel(\beta \times R(j), d)$$

Comme, on l'a indiqué précédemment, l'utilisation de la représentation redondante autorise une certaine erreur dans le choix de q_{j+1}, d'où elle permet à la fonction de sélection d'utiliser seulement $(\widehat{\beta R(j)})$ valeur tronquée de $\beta \times R(j)$ et $(\hat{d})$ valeur tronquée de d.

$$q_{j+1} = Sel(\widehat{\beta R(j)}, \hat{d})$$

La fonction de sélection est d'autant plus facile à réaliser que les tailles en bits de $(\widehat{BR(j)})$ et de $(\hat{d})$ sont faibles. Nous avons vu précédemment, pour les algorithmes de division restaurante et non restaurante, que dans la fonction de sélection d'un chiffre résultat q_{j+1}, on utilise la totalité des chiffres de $R(j)$ et de d. Si l'on utilisait juste une partie de ces deux opérandes pour cette sélection, on pourrait éventuellement simplifier la fonction de sélection : le choix d'un chiffre ne se fera plus par une comparaison exacte, mais en utilisant des estimations du reste partiel et du diviseur. En pratique, la comparaison des estimations peut s'implanter à l'aide d'une table dont le délai sera moins important que celui d'un comparateur.

2.6. Implémentation Matérielle de la division SRT

La division consiste en trois étapes qui sont l'initialisation, l'itération SRT et enfin la terminaison. Le schéma de calcul de la division par la méthode SRT est montré sur la figure 2.3.

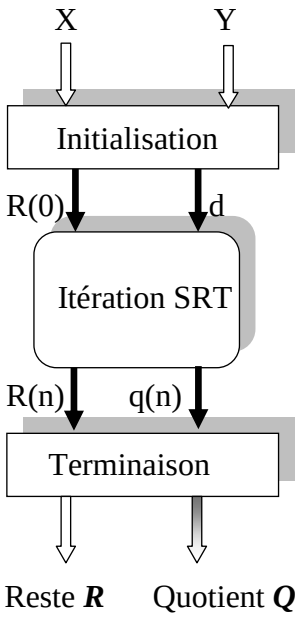


Figure 2.3 : Schéma de calcul de la division

2.6.1. Initialisation

L'étape d'initialisation est nécessaire dans le calcul de la division SRT, celle-ci consiste à satisfaire les conditions de convergence de l'algorithme SRT. Ces conditions sont : le diviseur doit être plus petit que le dividende $X < Y$ et le dividende doit être $1/2 \leq Y < 1$. Comme X et Y sont les mantisses de deux nombres donc $1 \leq X, Y < 2$.
Les conditions de convergences sont satisfaites en divisant X par 4 et Y par 2. D'où les nouveaux opérandes pour les itérations SRT sont alors :

$$R(0) = \frac{X}{4} \; et \; d = \frac{Y}{2}.$$

Cette étape est matériellement réalisée par de simples décalages.

2.6.2. Terminaison

L'étape de terminaison est l'étape finale de la division, après obtention du dernier chiffre q_n du quotient $q[n]$ et du reste $R(n)$. Ce dernier peut être négatif alors que dans la définition de la division, le reste final doit être positif. D'où la nécessité d'avoir une étape de correction qui consiste à ajuster la valeur du quotient comme suit :

$$Q = \begin{cases} q[n] & si \; R(n) \geq 0 \\ q[n] - \beta^{-n} & si \; R(n) < 0 \end{cases}$$

Pour restituer le bon résultat du quotient, vu que nous avons modifié les valeurs de nos opérandes dans l'étape d'initialisation, il faut réaliser la division du quotient résultat par 2, qui n'est rien d'autre qu'un simple décalage.

2.6.3. L'itération SRT

L'itération SRT représente le cœur de la division. Comme l'algorithme SRT produit un chiffre résultat par itération donc le nombre d'itérations est égal à la taille du résultat en chiffres. La taille de ce chiffre, en bits, est d'autant plus grande que la base est importante. Pour venir à bout d'une division de deux nombres sur n bits, avec des chiffres sur k bits, il faut $\lceil n/k \rceil$ itérations. D'où ce nombre est d'autant plus faible que k est grand (c.-à-d. les chiffres q_{j+1} sont représentés dans une grande base de numération β). Ceci nous amène à dire que le nombre d'itérations diminue en augmentant la base β.
Par exemple, la méthode SRT en base 2 produit un chiffre d'un bit par itération. Pour une précision de 53 bits (double précision de la norme IEEE-754), il faut 53 itérations. Alors si on passe à la base $4 = 2^2$, le nombre d'itérations est réduit de moitié et pour la base $16 = 2^4$, le nombre d'itérations est divisé par quatre. Cependant cette réduction ne vient

pas gratuitement, avec l'augmentation de la base, l'itération se complique d'avantage, ceci va être discuté tout au long de ce chapitre.

L'implémentation matérielle de l'équation de récurrence de la division SRT est montrée sur la figure 2.4. En fait, ce schéma reste toujours valable pour les méthodes citées précédemment.

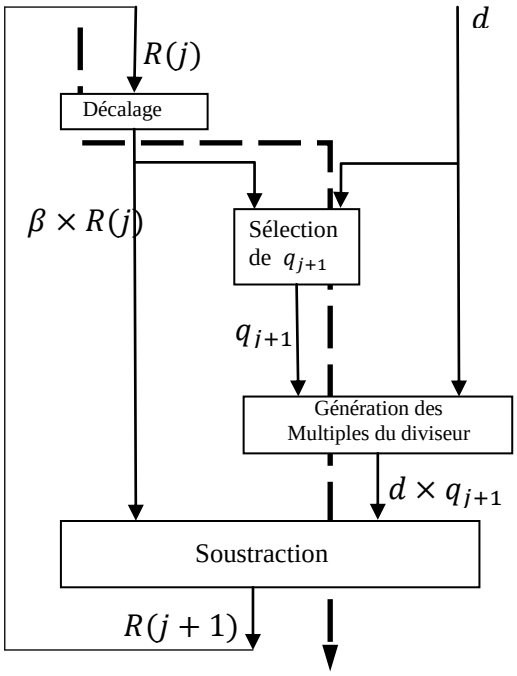


Figure 2.4 : Implémentation matérielle de l'itération SRT

Comme montrée sur la figure 2.4, l'itération SRT consiste en la réalisation des quatre opérations suivantes :

- Etape1 : Calcul de $\beta \times R(j)$.
- Etape 2 : Sélection du chiffre q_{j+1}.
- Etape 3 : Génération du multiple du diviseur $q_{j+1} \times d$.
- Etape 4 : Soustraction de $q_{j+1} \times d$ de $\beta \times R(j)$.

Comme les quatre opérations de l'itération SRT s'exécutent en série, l'optimisation des délais d'exécution de chacune de ces opérations est d'une grande importance dans la mesure qu'elle contribue à la diminution du délai de l'itération et par conséquent du délai total de la division.

- **Etape 1**

Pour la première opération, aucune optimisation n'est nécessaire si la base dans laquelle sont représentés les opérandes est un multiple entier de 2. Cette opération devient

alors un simple décalage à gauche du reste partiel $R(j)$ dont le délai est indépendant de la taille de $R(j)$. Alors que si la base est différente d'un nombre multiple de 2, l'opération de décalage devient alors une multiplication dont le délai dépend de la taille de $R(j)$. C'est pour cela que pour la SRT, on utilise que des bases multiples de 2.

- **Etape 2**

Pour la deuxième opération, qui est la sélection du chiffre q_{j+1} à partir des valeurs de $\beta R(j)$ et d. On a vu que cette opération était très compliquée pour la division à la main (base 10), celle-ci consiste à comparer $\beta R(j)$avec tous les multiples de Y $\{Y, 2Y, \dots, 9Y\}$. Cette sélection est réalisée par neuf comparaisons qui peuvent être réalisées en parallèle ou en série. On a vu aussi que ce nombre de comparaisons a diminué pour la division en base 2 (divisions restaurante et non restaurante). Ceci ne se fait pas sans inconvénient dans la mesure où la vitesse de convergence diminue en diminuant la magnitude de la base.
En résumé le minimum de délai possible pour l'opération de sélection (en base 2) est le délai d'un comparateur qui n'est rien d'autre qu'une soustraction avec une propagation de la retenue égale à la taille des opérandes.

- **Etape 3**

Pour la troisième opération qui est la génération des multiples du diviseur, celle-ci dépend des valeurs de q_{j+1} possibles. Cette opération peut être un simple décalage (pour des q_{j+1} multiples de 2) comme elle peut être une multiplication (pour des q_{j+1} différents d'un multiple de 2). La complexité de cette opération dépend de la valeur de la base β et celle du facteur de redondance ρ.

- **Etape 4**

Pour la quatrième opération qui est une soustraction avec propagation de la retenue, celle-ci ne dépend pas de la valeur de la base, toutefois elle peut faire l'objet d'optimisation de son délai au détriment de l'utilisation d'un matériel supplémentaire. L'utilisation de la représentation CS (carry save) peut diminuer le délai de ce soustracteur, dans le mesure où durant les itérations SRT les valeurs des restes partiels $R(j)$ sont gardés en représentation CS (pas de propagation de la retenue) jusqu'à la dernière itération. Ce procédé diminue certes le délai d'exécution de l'itération SRT, néanmoins il introduit un hardware supplémentaire, $R(j)$ sera désormais représenté par deux valeurs $R_s(j)$ et $R_c(j)$.

Nous avons vu, qu'à traves la description de ces quatre étapes de la réalisation d'une itération SRT, la complexité des étapes 2 et 3 sont étroitement liées à la base β et au facteur de redondance ρ.

2.7. Sélection du chiffre q_{j+1}

Le choix de chaque chiffre q_{j+1} du résultat doit être fait de telle sorte que l'équation 2.14, relative aux bornes du reste partiel $R(j)$, reste vérifiée. La fonction de sélection $q_{j+1} = Sel(\beta R(j), d)$ peut être accélérée en utilisant, à la place des valeurs exactes du reste partiel $\beta R(j)$ et du diviseur d, pour les comparaisons, les troncatures de ces valeurs qui sont notées par ($\widehat{\beta R(j)}$ et $\hat{d}$). Tronquer une valeur est le fait de prendre seulement quelques uns des bits de poids forts de cette valeur, en négligeant les bits de rang inférieur. Le nombre de bits tronqués dépend de la magnitude de l'erreur commise sur la sélection des chiffres q_{j+1}. Celle-ci doit toujours vérifier la condition de convergence de l'équation (2.11).

On définit alors, pour chaque chiffre $k \in \{-a, \ldots, +a\}$ de la représentation utilisée, l'intervalle $[L_k(d), U_k(d)]$ des valeurs de $\beta R(j)$ pour lesquelles le choix du chiffre $q_{j+1} = k$ est correct.

Par définition :

$$L_k(d) \leq \beta R(j) \leq U_k(d) \qquad (2.18)$$

De l'équation 2.14 on a

$$-\rho d \leq R(j+1) = \beta R(j) - kd \leq +\rho d. \qquad (2.19)$$

On en déduit :

$$L_k(d) = (k-\rho)d \quad \text{et} \quad U_k(d) = (k+\rho)d. \qquad (2.20)$$

Pour assurer qu'au moins un chiffre $q_{j+1} = k$ peut être choisie, pour toute valeur de $R(j)$, alors chacune des valeurs de $\beta R(j)$ doit appartenir au moins à un intervalle de sélection $[L_k(d), U_k(d)]$. C'est la condition de continuité, d'où, il est nécessaire à ce que :

$$U_{k-1}(d) \geq L_k(d) \qquad (2.21)$$

Des expressions (2.20) et (2.21) on déduit :

$$(k-1+\rho)d \geq (k-\rho)d$$

Puisque $\rho \geq 1/2$ et $d > 0$, donc

$$U_{k-1}(d) - L_k(d) = (2\rho - 1)d > 0 \qquad (2.22)$$

Ce qui implique l'existence de chevauchements entre les intervalles de sélection consécutifs. Ces chevauchements donnent la possibilité de sélection de deux chiffres voir trois (pour $\rho = 1$, redondance maximale) pour une seule valeur de $R(j)$.

Dû à ce chevauchement, il n'est pas nécessaire de connaitre la valeur exacte du reste $R(j)$ pour pouvoir sélectionner le bon chiffre q_{j+1}.

Sur le diagramme de Robertson, illustré par la figure 2.5, sont présentées les zones de sélections possibles pour un chiffre q_{j+1}. Ce diagramme a comme axes le reste décalé $\beta R(j)$ et le prochain reste $R(j+1)$. Ce diagramme représente les équations de récurrence par des lignes avec le paramètre $q_{j+1} = k$ pour $k = -a, \dots, +a$. On constate que les zones de sélection pour deux chiffres successifs se recouvrent. C'est ce recouvrement qui permet de simplifier l'opération de sélection du chiffre q_{j+1} au lieu de devoir utiliser des comparateurs.

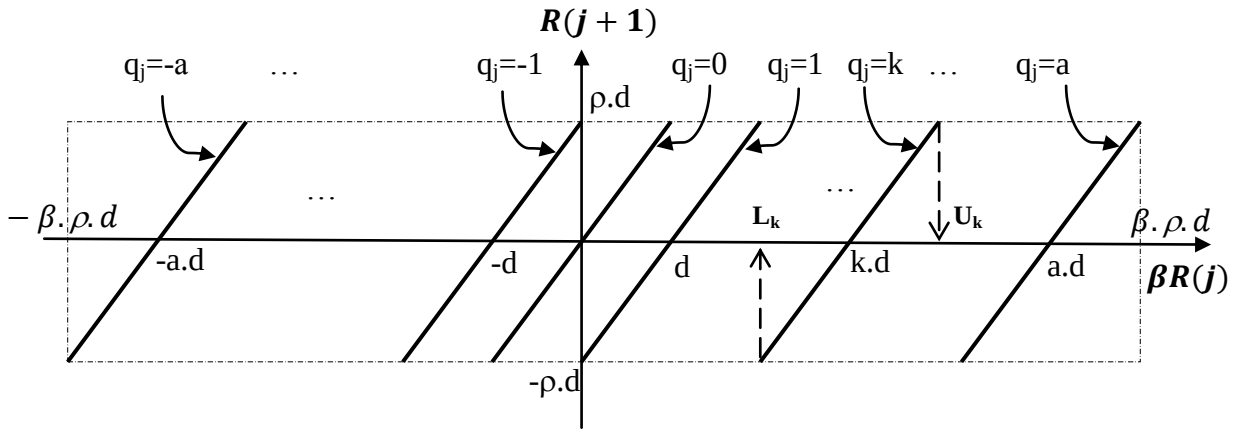


Figure 2.5 : Digramme de Robertson pour la division SRT en base β avec ρ = 1.

Un autre diagramme, qui est utile dans la conception de la fonction de sélection est le tracé de $\beta R(j)$ en fonction de d, appelé le P-D diagramme (représenté sur la figure 2.6). Les bornes de l'intervalle de sélection U_k et L_k sont représentés sur le diagramme comme des droites provenant de l'origine 0, avec des pentes $(k+\rho)$ et $(k-\rho)$, respectivement. Les régions délimitées par ces lignes sont utiles dans l'analyse de la fonction de sélection des chiffres du quotient.

Nous allons maintenant voir les fondements théoriques de cette sélection.

La fonction de sélection de départ est de la forme suivante :

$$q_{j+1} = Sel(\beta R(j), d)$$

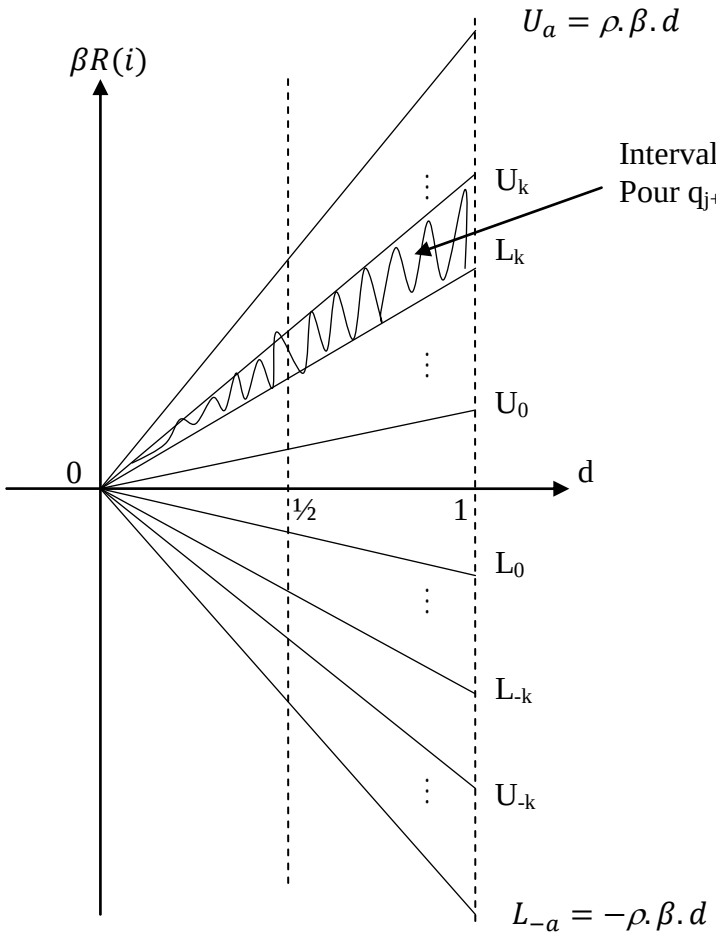


Figure 2.6 : P-D diagramme

Pour une valeur fixe du diviseur d, il est possible de représenter la fonction de sélection par l'ensemble de points $\{s_k(d)\}$, pour $-a \leq k \leq +a$, tel que :

$$q_{j+1} = k \quad si \quad s_k(d) \leq \beta R(j) \leq s_{k+1}(d) - ulp \qquad (2.23)$$

$s_k(d)$ est défini comme étant la valeur minimum de $\beta R(j)$ pour laquelle le chiffre $q_{j+1} = k$ est sélectionné, d'où $s_k(d)$ doit être à l'interieur de l'intervalle de selection.

$$L_k(d) \leq s_k(d) \leq U_k(d).$$

La fonction de sélection doit aussi vérifier la condition de continuité : on doit toujours être capable de choisir un nouveau chiffre du quotient $q_{j+1} = k - 1$ pour $\beta R(j) = s_k(d) - ulp$.

Par conséquent,

$$s_{k-1}(d) = s_k(d) - ulp \leq U_{k-1}(d).$$

Et comme $U_k(d) \geq U_{k-1}(d) + ulp$. On en déduit l'encadrement :

$$L_k(d) \leq s_k(d) \leq U_{k-1}(d) + ulp. \qquad (2.24)$$

Pour plus de simplicité, on utilise souvent l'expression suivante au lieu de l'équation (2.24).

$$L_k(d) \leq s_k(d) \leq U_{k-1}(d) \qquad (2.25)$$

Par conséquent, les valeurs de $s_k(d)$ peuvent être dans la région de chevauchement de deux intervalles de sélection consécutives comme c'est montré sur la figure 2.7.

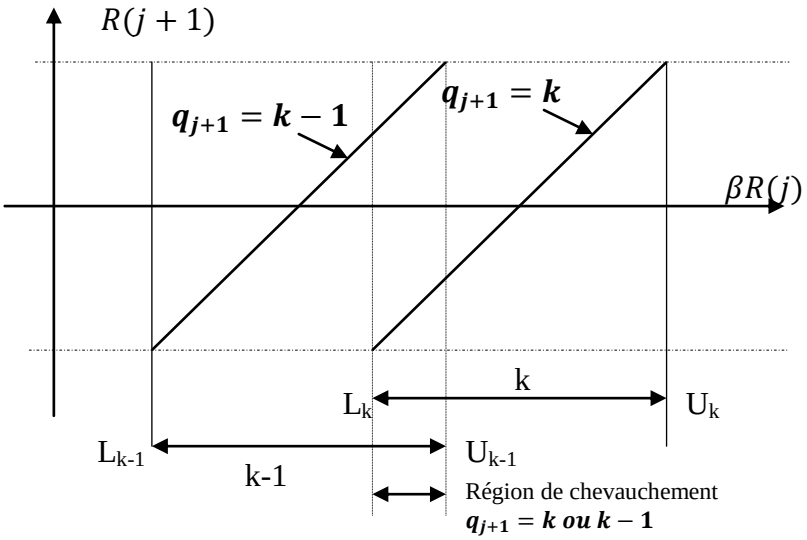


Figure 2.7 : Région de chevauchement entre deux intervalles de sélection consécutifs

L'épaisseur de cette région de chevauchement est donnée par l'expression suivante :

$$U_{k-1} - L_k = (k - 1 + \rho)d - (k - \rho)d = (2\rho - 1)d \qquad (2.26)$$

Celle-ci dépend des valeurs de ρ et de d. On note que ce chevauchement vaut zéro pour des chiffres du quotient non redondant ($\rho = 1/2$). C'est la raison pour laquelle nous utilisons la représentation redondante pour les chiffres du quotient. Par ailleurs, on remarque aussi que l'épaisseur de cette région de convergence est fonction de d. Celle-ci est très mince pour des valeurs de d très faibles. Pour améliorer d'avantage l'épaisseur de cette région de chevauchement, on fait une restriction sur les valeurs du diviseur d de telle sorte que $d \geq 1/2$ (qui représente les valeurs normalisées du diviseur d). En résumé, c'est la représentation redondante plus la restriction sur les valeurs de d qui nous permettent d'avoir une région de chevauchement convenable pour pouvoir simplifier la fonction de sélection. La figure 2.8 montre l'évolution des valeurs de $s_k(d)$ dans la région de chevauchement en fonction de d et de $\beta R(j)$.

Comme les valeurs de $s_k(d)$ dépendent de la valeur de d, il est plus intéressent de mettre les valeurs de $s_k(d)$ indépendantes de d. De nouvelles constantes sont alors utilisées notées par m_k. Ces constantes doivent satisfaire les conditions de l'équation (2.25) :

$$max(L_k) \leq m_k \leq min(U_{k-1}) \quad (2.27)$$

Où le $max(L_k)$ et le $min(U_{k-1})$ peuvent être obtenus pour les valeurs de d allant de 1/2 à 1.

Deux cas se présentent pour les valeurs de m_k selon k est positif ou négatif.

- ***Pour*** $\boldsymbol{k > 0}$

Dans ce cas le maximum de L_k est obtenu pour d=1, alors que le minimum de U_{k-1} est obtenu pour d=1/2 d'où l'expression suivante :

$$(k-\rho).1 \leq m_k \leq (k-1+\rho).\frac{1}{2} \quad (2.28)$$

Ce qui nécessite,

$$(k-\rho) \leq (k-1+\rho)2^{-1} \Rightarrow \rho \geq (k+1)/3 \quad (2.29)$$

- ***Pour*** $\boldsymbol{k \leq 0}$

Dans ce cas le maximum de L_k est obtenu pour $d = 1/2$, alors que le minimum de U_{k-1} est obtenu pour $d = 1$ d'où l'expression suivante :

$$(k-\rho).\frac{1}{2} \leq m_k \leq (k-1+\rho).1 \quad (2.30)$$

Ce qui implique que :

$$\rho \geq (-k+2)/3 \quad (2.31)$$

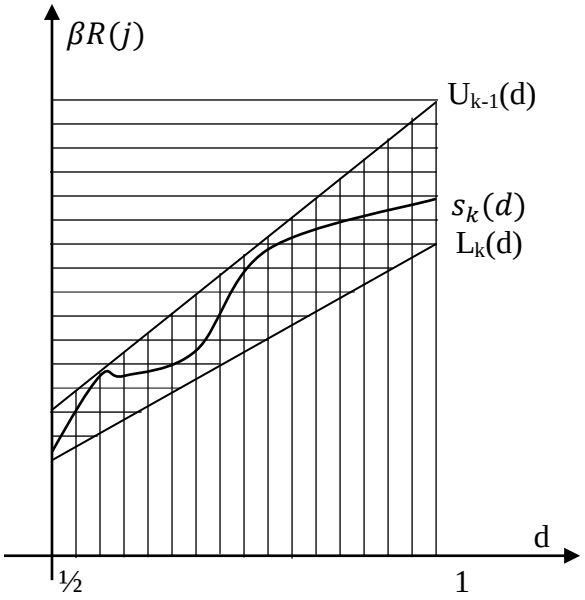


Figure 2.8 : Evolution de $s_k(d)$ dans la région de chevauchement

Comme les chiffres du quotient résultat sont choisis dans l'ensemble $\{-a, \dots, +a\}$ pour un facteur de redondance $\rho = \frac{a}{\beta-1}$. Le pire cas pour l'expression (2.31) est pour $k = -a$, ce qui implique que :

$$\rho \geq (a+2)/3 \qquad \text{et comme} \quad a = (\beta-1)\rho$$

On déduit que

$$\rho \geq 2/(4-\beta) \tag{2.32}$$

Et comme $\rho \leq 1$, nous aurons alors $\beta = 2$ est la seule solution à l'équation (2.32). D'où la base 2 est la seule base pour laquelle la fonction de sélection peut être indépendante du diviseur d.

Ce qui nous conduit à dire que pour les bases $\beta > 2$, il n'est pas possible de trouver une seule constante m_k pour toutes les valeurs de d. Dans ce cas là, on peut subdiviser l'intervalle des valeurs de d en sous intervalles $[d_i, d_{i+1})$ avec :

$$d_0 = 1/2, \qquad d_{i+1} = d_i + 2^{-\delta} \tag{2.33}$$

Où $2^{-\delta}$ représente la largeur de l'intervalle $[d_i, d_{i+1})$, celle-ci est d'autant plus grande que δ est petit. Par ailleurs, pour chaque intervalle $[d_i, d_{i+1})$ la sélection d'un chiffre résultat est représentée par une série de constantes $m_k(i)$ qui, pour tout $d \in [d_i, d_{i+1})$, on a :

$$q_{j+1} = k \quad \text{si} \quad m_k(i) \leq \beta R(j) < m_{k+1}(i) \tag{2.34}$$

En utilisant ce concept, on aurait utilisé seulement δ bits du diviseur d pour constituer la fonction de sélection. Dans chacun des intervalles $[d_i, d_{i+1})$, $m_k(i)$ ne dépend pas de la valeur de d mais de l'intervalle où se situe la valeur de d.

En considérant seulement les δ bits les plus significatifs du diviseur d, l'erreur de troncature commise sur d est notée par ϵ_d où $0 \leq \epsilon_d < ulp(\widehat{d})$ et $ulp(\widehat{d}) = 2^{-\delta}$.

Cette troncature sur les valeurs de d permet d'avoir des constantes $m_k(i)$ avec moins de bits significatifs. Cette reduction dans la taille de $m_k(i)$ rend possible l'exécution de l'opération de comparaison de l'équation (2.34) en utilisant quelques bits les plus significatifs $\beta R(j)$.

Soit $\widehat{\beta R(j)}$ représentant la valeur tronquée de $\beta R(j)$ et $ulp\left(\widehat{\beta R(j)}\right)$ est l'ulp de la valeur tronquée de $\beta R(j)$. L'erreur commise lors de cette troncature est notée ϵ_R.

$$\beta R(j) = \widehat{\beta R(j)} + \epsilon_R$$

L'erreur ϵ_R dépend de la représentation adoptée pour les restes $R(j)$. On va voir par la suite que certaines représentations permettent d'accélérer l'itération SRT.

$0 \leq \epsilon_R < ulp(\widehat{\beta R(j)})$ pour la représentation en complément à 2 des $R(j)$.

$0 \leq \epsilon_R < 2ulp(\widehat{\beta R(j)})$ pour une représentation à retenue conservée CS *(carry-save)* des $R(j)$.

Comme illustré sur la figure 2.9, $\widehat{\beta R(j)}$ est obtenu en gardant seulement les $(\tau + t)$ bits les plus significatifs de $\beta R(j)$. Où τ et t représentent respectivement les τ bits de la partie entière de $\beta R(j)$ et les t premiers bits de la partie fractionnaire de $\beta R(j)$.

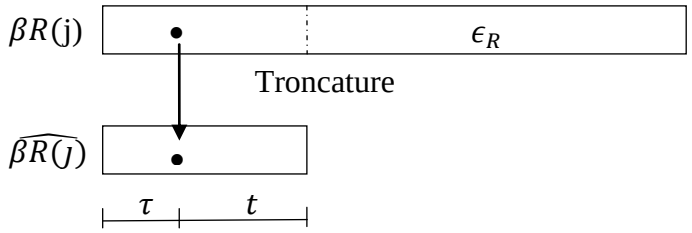


Figure 2.9 : Troncature du reste $\beta R(j)$

Maintenant, notre fonction de sélection est représentée par la constante $m_k(i)$ qui est utilisée dans tout l'intervalle $[d_i, d_{i+1})$, cette constante satisfait les conditions de l'expression (2.27), d'où la nouvelle expression sera :

$$max(L_k(d_i), L_k(d_{i+1})) \leq m_k(i) < min(U_{k-1}(d_i), U_{k-1}(d_{i+1})) \qquad (2.35)$$

Avec les troncatures effectuées sur d et $\beta R(j)$, la fonction $m_k(i)$ devient alors une fonction escalier comme l'illustre la figure 2.10. Celle-ci délimite un rectangle dans lequel un chiffre quotient peut être choisi.

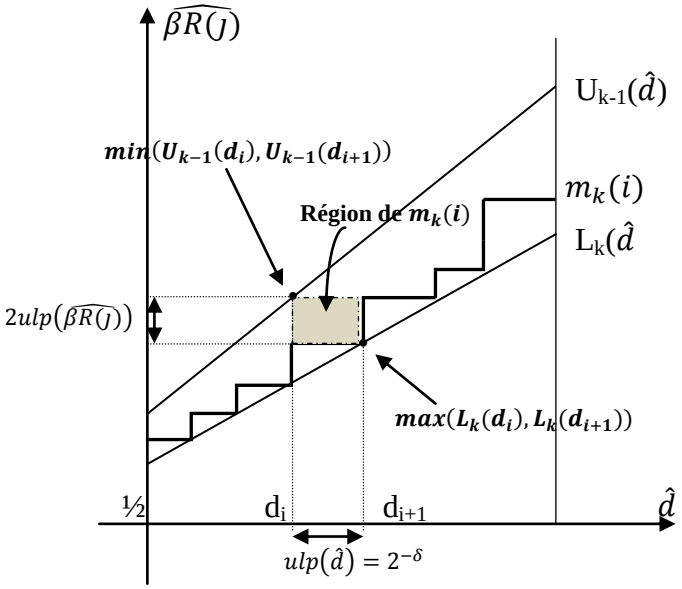


Figure 2.10 : $m_k(i)$ comme fonction du diviseur tronqué

La fonction escalier $m_k(i)$ est maintenant représentée par un ensemble de constantes correspondant aux différentes valeurs tronquées de $\widehat{d}$. Ces constantes sont supposées être représentées à la même précision que $\widehat{\beta R(j)}$.

Le rectangle en pointillé de la figure 2.10 représente l'ensemble de points $(\beta R(j), d)$ qui satisfont :

$$m_k(i) \leq \beta R(j) < m_{k+1}(i) = m_k(i) + ulp \text{ et } d_i \leq d < d_{i+1} = d_i + ulp(\hat{d}) \quad (2.36)$$

Où le chiffre $q_{j+1} = k$ est sélectionné pour l'ensemble des points du rectangle décrit par l'expression (2.36). Ceci en supposant que $\beta R(j)$ est dans une représentation C.S et d est en complément à 2. Ce rectangle a une longueur de $2\, ulp(\widehat{\beta R(j)})$ et une largeur de un $ulp(\hat{d})$. Les extrémums de ce rectangle sont obtenus (pour $k > 0$) en utilisant l'expression (2.35)

$$max(L_k(d_i), L_k(d_{i+1})) = L_k(d_{i+1}) = L_k\left(d_i + ulp(\hat{d})\right) = (k-\rho)\left(d_i + ulp(\hat{d})\right) \quad (2.37)$$

$$min(U_{k-1}(d_i), U_{k-1}(d_{i+1})) = U_{k-1}(d_i) = (k-1+\rho)(d_i) \quad (2.38)$$

D'où la nouvelle expression :

$$(k-\rho)\left(d_i + ulp(\hat{d})\right) \leq m_k(i) < (k-1+\rho)(d_i) \quad (2.39)$$

La fonction de sélection est maintenant une fonction des δ bits les plus significatifs de d, et comme $d \geq 1/2$, nous aurons alors besoin seulement que de $(\delta - 1)$ bits. Comme d est normalisé, son premier bit qui est toujours à 1, en ignorant ce bit d_i peut être exprimé en fonction de i :

$d_i = i.2^{-\delta}$, où i est un entier $2^{\delta-1} \leq i < 2^{\delta}$ et $2^{-\delta} est\ l'ulp(\hat{d})$.

Par ailleurs $m_k(i)$ peut être exprimée comme un entier $A_k(i)$ multiple de $l'ulp(\widehat{\beta R(j)})$ $(A_k(i).2^{-t})$, puisque $m_k(i)$ représentent les frontières de $(\widehat{\beta R(j)})$ dans l'expression (2.36). L'expression (2.39) devient alors :

$$2^{-\delta}(k-\rho)(i+1) \leq A_k(i).2^{-t} < 2^{-\delta}(k-1+\rho).i$$

Et comme $A_k(i)$ est un entier d'où la nouvelle expression :

$$2^{t-\delta}(k-\rho)(i+1) \leq A_k(i) < 2^{t-\delta}(k-1+\rho).i$$

Qui peut devenir

$$2^{t-\delta}(k-\rho)(i+1) \leq A_k(i) \leq 2^{t-\delta}(k-1+\rho).i - 1$$

Et comme $A_k(i)$ est un entier ses bornes doivent être arrondies. La nouvelle expression est alors :

$$\lceil 2^{t-\delta}(k-\rho)(i+1) \rceil \leq A_k(i) \leq \lfloor 2^{t-\delta}(k-1+\rho).i - 1 \rfloor \quad (2.40)$$

⌈ ⌉ : Indique un arrondi vers l'entier le plus grand.

⌊ ⌋: Indique un arrondi vers l'entier le plus petit.

La figure 2.11 illustre les constantes $A_k(i)$ qui seront utilisés comme fonction de sélection en substitution à $m_k(i)$.

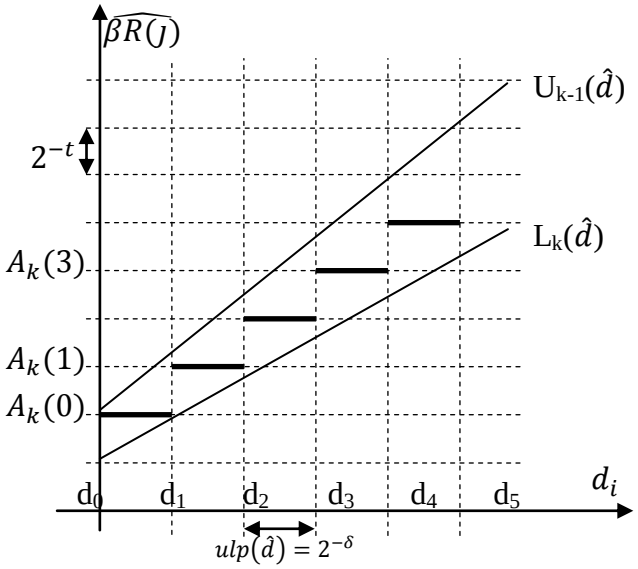


Figure 2.11 : $A_k(i)$ nouvelle fonction de sélection

Le problème maintenant est de trouver ces constantes de sélection $A_k(i)$ pour des valeurs minimales de δ et de t. C'est ces dernières plus celle de τ (qui représente la taille de la partie entière de $\widehat{\beta R(J)}$) qui conditionnent la taille de la mémoire, qui sera utilisée pour la sélection des chiffres q_{j+1} du quotient résultat. Comme illustré sur la figure 2.12, la sélection du chiffre q_{j+1} consiste en une table adressée par les $(\tau + t)$ bits de $\{\widehat{\beta R(J)}\}_t$ plus les $(\delta - 1)$ bits de $\{\hat{d}\}_\delta$. Il est à noter que seulement $(\delta - 1)$ bits de la partie fractionnaire sont utilisés dans l'adressage de la mémoire de sélection, vu que le premier bit fractionnaire de d est toujours à 1 ($d \geq \frac{1}{2}$).

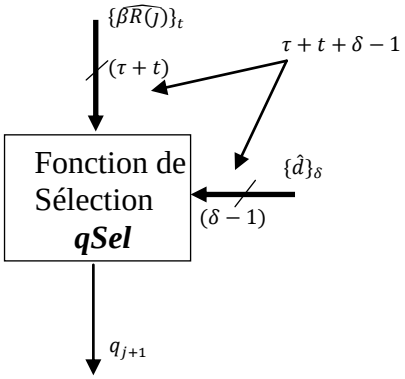


Figure 2.12 : Sélection de q_{j+1} par une table

Malheureusement, il n'existe pas une simple solution pour les valeurs de δ et t, dans la mesure où en réduisant δ, t augmente. Par conséquence l'optimisation doit être faite sur la valeur de $(\delta + t)$, puisque celle-ci a une relation directe avec le nombre de bits nécessaire à la fonction de sélection.

P. KORNERUP dans [33] a donné une formulation analytique pour le calcul de δ et t qui satisfait l'équation (2.40).

$$2^{-\delta} < \frac{\rho - 1/2}{a - \rho} \qquad (2.41)$$

Cette formule est valable pour des chiffres q_{j+1} représentés dans un système de numération redondant $(\beta - a)$ où β représente la base de numération et a est la plus grande valeur que peut prendre le chiffre quotient q_{j+1}.

Pour déterminer t pour une valeur δ donnée ou déterminée au préalable par l'expression (2.41), il a été montré dans [33], qu'en supposant que t_0 est la plus petit valeur qui satisfait l'expression suivante :

$$2^{-t_0} \leq ((\rho - \tfrac{1}{2}) - (a - \rho)2^{-\delta})/2 \qquad (2.42)$$

Et soit $\Delta(\delta, t, i) = \lfloor 2^{t-\delta}(a - 1 + \rho)(i) - 1 \rfloor - \lceil 2^{t-\delta}(a - \rho)(i + 1) \rceil$ (2.43)

Alors la valeur de t qui correspond au nombre de bits sur lequel sera tronquée la partie fractionnaire de $\widehat{\beta R(j)}$ est calculée comme suit :

$$t = \begin{cases} t_0 & si \;\; \Delta(\delta, t_0, 2^{\delta-1}) \geq 1 \\ t_0 & si \;\; \Delta(\delta, t_0, i) = 0 \; pour \; i = 2^{\delta-1}, \ldots, (i_1 - 1) \; et \; \Delta(\delta, t_0, i_1) \geq 1 \\ t_0 + 1 & ailleurs \end{cases} \qquad (2.44)$$

Où i_1 est une valeur comprise entre $2^{\delta-1}$ et 2^{δ}.

La détermination de τ qui représente le nombre de bits de la partie entière de $\widehat{\beta R(j)}$ dépend de la base β utilisée ainsi que de la représentation adoptée pour les restes $R(j)$. Pour un $R(j)$ représenté en CS, τ est calculé comme suit :

$$\tau = 1 + log_2(\beta . \rho . d)$$

Pour une valeur donnée du diviseur $d = i.2^{-\delta}$, il est maintenant possible de calculer les frontières $A_k(i)$, dans l'intervalle de sélection $[L_k, U_{k-1}]$ par l'expression (2.40) telle que la fonction de sélection $q_{j+1} = qSel(\widehat{\beta R(j)}, i)$ soit définie comme suit :

$$\beta R(j) \geq 0: \;\; A_k(i) \leq \widehat{\beta R(j)} < A_{k+1}(i) \Rightarrow q_{j+1} = k \qquad (2.45)$$

$$\beta R(j) < 0: \;\; -A_{k+1}(i) < \widehat{\beta R(j)} \leq -A_k(i) \Rightarrow q_{j+1} = -k \qquad (2.46)$$

En utilisant la symétrie autour de l'axe des $\beta R(j)$, on peut diviser par deux la taille de la table de sélection des chiffres q_{j+1} *(Table qSel).* Le bit à ignorer, lors de l'adressage de cette table, est le bit le plus significatif de $\beta R(j)$. Si celui-ci est à 0, l'adressage se fait alors normalement et le chiffre q_{j+1} du quotient résultat est égal au chiffre lu dans la mémoire $qSel$ (en ajoutant un bit 0, comme signe à ce chiffre). Dans le cas contraire, c.-à-d. $\beta R(j)$ est négatif et son bit le plus significatif est à 1, on calcule alors la valeur positif de $\beta R(j)$ et on adresse avec la mémoire $qSel$, et on ajoute le bit 1 comme signe au chiffre q_{j+1} lu de la table $qSel$. Cette optimisation de la table $qSel$ n'est pas faite gratuitement, puisque celle-ci nécessite un hardware supplémentaire, d'où son utilisation n'est préconisée que quand la taille de la table *qSel* est importante (c.-à-d. pour les grandes bases).

2.8. Architecture de la division SRT

Après ce qui a été dit dans les sections précédentes, l'architecture de l'itération SRT de la figure 2.4 devient alors celle représentée sur la figure 2.13.

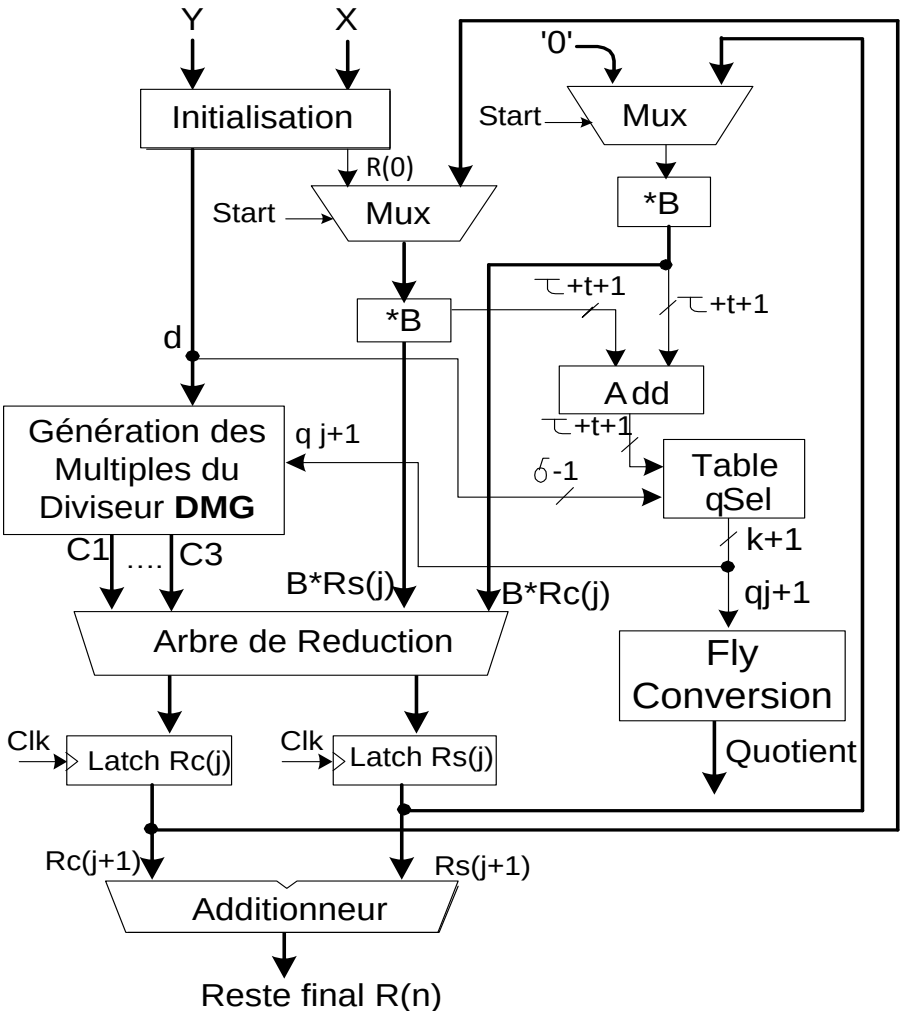


Figure 2.13 : Architecture itérative de la division SRT

La sélection du chiffre q_{j+1} qui était un ensemble de comparaisons est substituée par une lecture de table. On va voir aussi la génération du multiple du diviseur $q_{j+1}\times d$, qui pouvait être un multiplieur, sera substitué par le module DMG, dont la complexité est moindre et la performance est plus importante qu'un multiplieur. Le module DMG génère deux à trois termes au lieu du produit $q_{j+1}\times d$. Ces termes sont admis dans l'arbre de réduction au même titre que les deux composantes du reste $\{\beta.R_s(j)$ et $\beta.R_c(j)\}$ pour obtenir le reste de la (j+1)^{ème} itération $\{R_s(j+1)$ et $R_c(j+1)\}$ dans le format C.S. La représentation C.S dite aussi (à retenue sauvegardée) est utilisée avantageusement dans la mesure où l'addition est réalisée sans propagation de la retenue.

L'optimisation algorithmique et architecturale de cette architecture est expliquée dans les sections suivantes.

2.9. La table *qSel*

La table *qSel* est un élément important dans l'architecture de l'itération SRT. La taille de cette mémoire est donnée par l'expression suivante :

$$Taille_{qSel} = 2^{\delta+\tau+t-1}(1 + log_2\beta) \qquad (2.47)$$

A partir de cette expression, la taille de $qSel$ dépend alors des niveaux de troncature du diviseur d et du reste $R(j)$ ainsi que de la valeur de la base β.

Dans ce qui suit, nous allons étudier l'influence de la base β ainsi que du facteur de redondance ρ sur la taille de la table $qSel$. Pour ce faire, nous allons prendre comme exemple les bases $\beta = 4, 8\ et\ 16$ pour des facteurs de redondances minimale et maximale.

2.9.1. SRT en base 4 ($\beta = 4$)

Deux cas se présentent :

2.9.1.1. SRT en base 4 et ρ minimal

Pour la division SRT en base 4 et un facteur de redondance minimale, les chiffres q_{j+1} sont choisis à partir de l'ensemble $D = \{-2, -1, 0, 1, 2\}$, alors $a = 2\ et\ \rho = 2/3$. Par l'équation (2.41), on peut calculer l'entier minimal δ qui vérifie cette expression.

$$\begin{aligned} 2^{-\delta} &< (\rho - 1/2)/(a - \rho) \\ &< (2/3 - 1/2)/(2 - 2/3) \\ &< (1/6)/(4/3) \\ &< 1/8 \end{aligned}$$

Soit $\delta = 4$ et à partir des expressions (2.42) on peut déterminer t_0.

$$\begin{aligned} 2^{-t_0} &\leq (\rho - 1/2) - (2-\rho)2^{-\delta} \\ &\leq (2/3 - 1/2) - (2 - 2/3)2^{-4} \\ &\leq 1/6 - 1/12 \\ &\leq 1/12 \end{aligned}$$

Alors $t_0 = 4$, et à partir de l'expression (2.44) on peut calculer la valeur de t.

$$t = \begin{cases} t_0 & si \;\; \Delta(4,4,2^3) \geq 1 \\ t_0 & si \;\; \Delta(4,4,i) = 0 \; pour \; i = 2^3, \ldots, i_1 - 1 \; et \; \Delta(4,4,i_1) \geq 1 \\ t_0 + 1 & ailleurs \end{cases} \qquad (2.48)$$

Et selon l'expression (2.43),

$$\begin{aligned} \Delta(4,4,2^3) &= \lfloor 2^{4-4}(2 - 1 + 2/3)2^3 - 1 \rfloor - \lceil 2^{4-4}(2 - 2/3)(2^3 + 1) \rceil \\ &= \lfloor (5/3)2^3 - 1 \rfloor - \lceil (4/3)(2^3 + 1) \rceil \\ &= \lfloor (37/3) \rfloor - \lceil 12 \rceil \\ &= 12 - 12 = 0 \end{aligned}$$

D'où la première supposition de l'expression (2.48) n'est pas vérifiée, on passe à la deuxième.

Pour $i = 2^3 = 8$, $\Delta(4,4,8) = 0$

Pour $i = 2^3 + 1 = 9$, $\Delta(4,4,9) = 0$

Pour $i = 2^3 + 2 = 10$, $\Delta(4,4,10) = 0$

Pour $i = 2^3 + 3 = 11$, $\Delta(4,4,11) = 1$

Donc, c'est la deuxième supposition qui est vérifiée, alors $t = t_0 = 4$.

Il reste à calculer la valeur de τ pour enfin pouvoir estimer la taille de la table $qSel$ pour la base $\beta = 4$ avec $\rho = 2/3$.

La partie entière des restes $\beta R(j)$ représentée par $\tau = 1 + log_2(\beta \rho d)$ qui est inferieur à $(1 + log_2 \beta)$ puisque $\rho \leq 1$ et $d \leq 1$, d'où τ est égale alors à 3. La taille de la mémoire $qSel$ est calculée par l'expression (2.47).

$$Taille_{qSel} = 2^{3+4+4-1}(1 + log_2 4) = 2^{10}(3) bits = 3072 \; bits = 3 \; kbits$$

Une fois la taille de la mémoire est estimée, il reste maintenant à calculer les chiffres résultat q_{j+1} correspondant à chaque adresse (qui est composé de $(\delta - 1)$ $bits$ du diviseur d et de $(\tau + t - 1)$ bits de $\beta R(j)$.

En premier lieu, nous allons calculer les fonctions de sélection $A_k(i)$ ou les frontières correspondant aux valeurs k (que peut prendre un chiffre du quotient résultat q_{j+1}) pour chacune des valeurs $i = 2^{\delta}\hat{d}$.

En utilisant la symétrie autour de l'axe des $\beta R(j)$, on va se restreindre seulement aux valeurs positifs de $\beta \times R(j)$. Les fonctions $A_k(i)$ sont calculées en prenant la borne inférieure de l'expression (2.40).

$$A_k(i) = \left\lceil 2^{t-\delta}(k-\rho)(i+1) \right\rceil = \lceil 2^0(k-2/3)(i+1) \rceil \qquad (2.49)$$

Le tableau 2.1 dresse les valeurs des frontières $A_k(i)$ pour une SRT avec une base $\beta = 4$ et $\rho = 2/3$.

Tableau 2.1: Valeurs de $A_k(i)$ pour SRT avec $\beta = 4\ et\ \rho = 2/3$.

$\boldsymbol{i = 16\hat{d}}$	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
$A_1(i) = 16\hat{m}_1(i)$	*3*	*4*	*4*	*4*	*5*	*5*	*5*	*6*
$A_2(i) = 16\hat{m}_2(i)$	*12*	*14*	*15*	*16*	*18*	*19*	*20*	*22*

Une fois que les frontières $A_k(i)$ sont déterminées, on peut maintenant constituer notre table $qSel$, qui contiendra les chiffres q_{j+1} correspondant à chacune des valeurs de $\beta \times R(j)$. En utilisant la symétrie autour de l'axe des $\beta R(j)$, notre table ne contiendera alors que des chiffres de l'ensemble $\{0, 1, 2\}$. Dans le tableau 2.1 nous avons calculé pour chaque intervalle $[d_i, d_{i+1})$ les valeurs correspondantes des frontières $A_k(i)$. En utilisant l'expression (2.45), on peut déterminer facilement le chiffre q_{j+1} correspondant à chacune des valeurs de $\widehat{\beta R(j)}$. Ce sont ces chiffres qui seront alors stockés dans une table $qSel$ qui sera adressée par les $(\tau + t - 1)$ bits de $\{\widehat{\beta R(j)}\}_t$ plus les $(\delta - 1)$ bits de $\{\hat{d}\}_\delta$.

2.9.1.2. SRT en base 4 et ρ maximal

Pour une SRT en base 4 avec un facteur de redondance maximale, les chiffres q_{j+1} sont choisis à partir de l'ensemble $D = \{-3, -2, -1, 0, 1, 2, 3\}$, alors $a = 3\ et\ \rho = 1$. Par l'équation (2.41), on peut calculer l'entier minimal δ qui vérifie cette expression :

$$\begin{aligned} 2^{-\delta} &< (\rho - 1/2)/(a - \rho) \\ &< (1 - 1/2)/(3 - 1) \\ &< (1/2)/2 \\ &< 1/4 \end{aligned}$$

Donc $\delta = 3$, déterminons $alors\ t_0$:

$$\begin{aligned} 2^{-t_0} &\leq (\rho - 1/2) - (a - \rho)2^{-\delta} \\ &\leq (1 - 1/2) - (3 - 1)2^{-3} \end{aligned}$$

$$\leq 1/2 - 1/4$$
$$\leq 1/4$$

D'où $t_0 = 2$.

De la même manière que pour le cas précédent, en utilisant les expressions (2.43) et (2.44) pour la détermination de t, on trouve $t = 2$.

La partie entière, des restes $\beta R(j)$, représentée par $(\tau = 1 + log_2\beta)$, est égale alors à 3. La taille de la table $qSel$ est calculée par l'expression (2.47).

$$Taille_{qSel} = 2^7(3)bits = 384\ bits$$

En utilisant la symétrie autour de l'axe des $\beta R(j)$, on va se restreindre seulement aux valeurs positives de $\beta R(j)$. Les fonctions $A_k(i)$ sont calculées en prenant la borne inférieure de l'expression (2.40).

$$A_k(i) = \left\lceil 2^{t-\delta}(k-\rho)(i+1) \right\rceil = \lceil 2^0(k-1)(i+1) \rceil \qquad (2.50)$$

Le tableau 2.1 dresse les valeurs des frontières $A_k(i)$ pour une SRT avec une base $\beta = 4$ $et\ \rho = 1$.

Tableau 2.2 Valeurs de $A_k(i)$ pour SRT avec $\beta = 4$ et $\rho = 1$.

$\boldsymbol{i = 8\widehat{d}}$	**4**	**5**	**6**	**7**
$A_1(i) = 8\widehat{m}_1(i)$	0	0	0	0
$A_2(i) = 8\widehat{m}_1(i)$	5	6	7	8
$A_3(i) = 8\widehat{m}_1(i)$	10	12	14	16

2.9.2. SRT en base 8 ($\beta = 8$)

Deux cas se présentent :

2.9.2.1. SRT en base 8 et ρ minimal

Dans ce cas les chiffres q_{j+1} sont choisis à partir de l'ensemble $D = \{-4, -3, -2, -1, 0, 1, 2, 3, 4\}$, alors $a = 4\ et\ \rho = \frac{4}{7}$

$$2^{-\delta} < 1/48 \quad \Rightarrow \delta = 6$$

Et on détermine alors t_0 $\qquad 2^{-t_0} \leq 1/56 \ \Rightarrow t_0 = 6$

Et par le biais de l'expression (2.44), on détermine $t = 6$.

τ est calculé par l'expression suivante : $\tau = 1 + log_2 8 = 4$

La taille de la mémoire $qSel$ est :

$$Taille_{qSel} = 2^{15}(4)bits = 131072\ bits = 128Kbits$$

2.9.2.2. SRT en base 8 et ρ maximal

Dans ce cas les chiffres q_{j+1} sont choisis à partir de l'ensemble $D = \{-7, \dots, 0, \dots, 7\}$, alors $a = 7\ et\ \rho = 1$.

$$2^{-\delta} < 1/12 \quad \Rightarrow \delta = 4$$

Et on détermine alors t_0 $\quad 2^{-t_0} \leq 1/8 \quad \Rightarrow t_0 = 3$

Et par le biais de l'expression (2.44), on détermine $t = t_0 = 3$

La taille de la mémoire $qSel$ est calculée par l'expression (2.47) :

$$Taille_{qSel} = 2^{10}(4)bits = 4096\ bits = 4Kbits$$

2.9.3. SRT en base 16 ($\beta = 16$)

Deux cas se présentent :

2.9.3.1. SRT en base 16 et ρ minimal

Dans ce cas les chiffres q_{j+1} sont choisis à partir de l'ensemble $D = \{-8, \dots, 0, \dots, 8\}$, alors $a = 8\ et\ \rho = 8/15$.

$$2^{-\delta} < 1/224 \quad \Rightarrow \delta = 8$$

Et on détermine alors t_0 $\quad 2^{-t_0} \leq 1/240 \quad \Rightarrow t_0 = 8$

Et par le biais de l'expression (2.44), on détermine $t = t_0 = 8$

τ est calculé par l'expression suivante : $\tau = 1 + log_2 16 = 5$

La taille de la mémoire $qSel$ est calculée par l'expression (2.47) :

$$Taille_{qSel} = 2^{20}(5)bits = 5242880\ bits = 5.12Mbits$$

2.9.3.2. SRT en base 16 et ρ maximal

Dans ce cas les chiffres q_{j+1} sont choisis à partir de l'ensemble $D = \{-15, \dots, 0, \dots, 15\}$, alors $a = 15\ et\ \rho = 1$.

$$2^{-\delta} < 1/28 \quad \Rightarrow \delta = 5$$

Et on détermine alors t_0 $\quad 2^{-t_0} \leq 1/16 \quad \Rightarrow t_0 = 4$

Et par le biais de l'expression (2.44), on détermine $t = t_0 = 4$.

La taille de la mémoire $qSel$ est calculée par l'expression (2.47).

$$Taille_{qSel} = 2^{13}(5)bits = 40960\ bits = 40Kbits$$

Le calcul de la taille de la table $qSel$ n'est pas une chose facile à réaliser en calcul manuel, pour ce faire nous avons réalisé deux procédures en MAPLE [34], pour le calcul des tailles de cette table pour différentes bases, allant de la base 4 à la base 256, respectivement pour des facteurs de redondance minimale et maximale. Ces procédures sont montrées sur la figure 2.14.

```
> table_qSel_rhoMax:=proc(B)
with(MTM):
Digits := 40;
rho:=1;
a:=evalf(B-1);
del:=evalf(log2((a-rho)/(rho-0.5))+1);
delta:=trunc(del);
b:=evalf(log2(1/((rho-0.5)-(a-rho)*2^(-delta))));
tZERO:=trunc(b);
i1:= evalf(2^(delta-1));
i2:= evalf(2^(delta));
t:=0;
      for i from i1 to i2  do
          s:=evalf(2^(tZERO-delta)*(a-1+rho)*i-1);
          q:=evalf(2^(tZERO-delta)*(a-rho)*(i+1));
          DELTA:=floor(s)-ceil(q);
                if DELTA >= 1 then  t:=tZERO;
                end if;
      end do;
    if t=0 then t:=tZERO+1;
    end if;
d:= evalf(1+log2(B));
Tho:= round(d);
T_qSel:= evalf(2^(Tho+t+delta-1)*Tho);
print(Tho,delta,tZERO,t,T_qSel);
end;
```

a. ρ maximal

```
> table_qSel_rhoMin:=proc(B)
with(MTM):
Digits := 40;
a:=evalf(B/2);
rho:=evalf(a/(B-1));
del:=evalf(log2((a-rho)/(rho-0.5))+1);
delta:=trunc(del+0.1);
b:=evalf(log2(1/((rho-0.5)-(a-rho)*2^(-delta))));
tZERO:=trunc(b+1);
i1:= evalf(2^(delta-1));
i2:= evalf(2^(delta));
t:=0;
      for i from i1 to i2  do
          s:=evalf(2^(tZERO-delta)*(a-1+rho)*i-1);
          q:=evalf(2^(tZERO-delta)*(a-rho)*(i+1));
          DELTA:=floor(s)-ceil(q);
                if DELTA >= 1 then  t:=tZERO;
                end if;
          end do;
    if t=0 then t:=tZERO+1;
    end if;
Tho:= evalf(1+log2(B));
T_qSel:= evalf(2^(Tho+t+delta-1)*Tho);
print(Tho,delta,tZERO,t,T_qSel);
end;
```

b. ρ minimal

Figure 2.14 : Procédures en Maple pour le calcul de la taille de la table $qSel$

Les résultats de calcul de ces procédures pour les tailles de la table $qSel$ sont donnés sur le tableau 2.3.

Les résultats du tableau 2.3 montrent que l'augmentation de la base β fait augmenter la taille de la table $qSel$, néanmoins l'utilisation d'un facteur de redondance ρ maximale attenue amplement cette augmentation dans la mesure où pour une base $\beta = 256$, la taille de la table $qSel$ (pour un ρ minimal) est d'environ mille fois plus importante que celle avec un ρ maximal.

Nous avons vu dans cette section qu'à chaque itération la sélection du chiffre q_{j+1} du quotient résultat se fait par une simple lecture de table. Certes la constitution de la table

$qSel$ fait intervenir une grande complexité de calcul, cependant cette complexité est en amont de l'implémentation matérielle. La prochaine étape dans l'itération SRT est l'étape 3 qui consiste à générer les multiples du diviseur $d \times q_{j+1}$.

Tableau 2.3 : Tailles de la table $qSel$

Base β	4	8	16	32	64	128	256
Taille qSel ρ(min) (bits)	3072	$1{,}31072\times10^{5}$	$5{,}2428\times10^{6}$	$5{,}033165\times10^{7}$	$9{,}395241\times10^{8}$	$1{,}717987\times10^{10}$	$3{,}09237\times10^{11}$
Taille qSel ρ (max) (bits)	384	4096	40960	$3{,}93216\times10^{5}$	$3{,}670016\times10^{6}$	$3{,}355443\times10^{7}$	$3{,}01989\times10^{8}$

2.10. Génération des multiples du diviseur

Nous avons noté précédemment, dans ce chapitre, que cette opération dépend des valeurs possibles de q_{j+1}. Celle-ci est un simple décalage si q_{j+1} est un multiple entier de 2, c.-à-d. ($q_{j+1} = 2^k$) et une multiplication dans le cas contraire. Nous avons vu aussi que l'augmentation de la base faisait diminuer le nombre d'itérations. Cette diminution du nombre d'itérations peut éventuellement augmenter les performances d'exécution de la division si celle-ci n'induit pas une augmentation de complexité dans l'exécution de l'itération.

Dans cette section, nous allons étudier l'impact de l'augmentation de la base sur les performances d'exécution de cette opération, et nous allons proposer des solutions en se basant sur des optimisations algorithmiques architecturales conjuguées à une utilisation astucieuse des ressources du circuit FPGA.

L'effet de l'augmentation de la base β sur les performances d'exécution de l'algorithme de la division SRT a fait l'objet de plusieurs travaux de recherche récents [35], [36], [37]. Ces derniers ont souvent reporté que l'utilisation de la base 4 présente les meilleures performances. Ceci n'est pas dû seulement à l'utilisation de la base 4 mais aussi à l'utilisation d'un facteur de redondance ρ minimale. L'utilisation d'un facteur de redondance minimale avec une base $\beta = 4$ permet de représenter les chiffres du quotient q_{j+1} dans l'ensemble $D = \{-2, -1, 0, 1, 2\}$; où la multiplication par q_{j+1} est obtenue par une simple opération de décalage. Quand on augmente la base β, le nombre d'itérations diminue, mais la complexité de l'architecture augmente, à cause de la nécessité de

l'utilisation d'un multiplieur pour effectuer l'opération $(d \times q_{j+1})$. Ce qui ralentit l'exécution de l'itération et augmente la taille de la mémoire $qSEL$. Pour des implémentations sur des circuits FPGAs de la famille Virtex-2 de Xilinx, on peut utiliser les blocs multiplieurs 18×18 bits. Cependant, les travaux reportés dans [35] montrent que cette solution induit des délais considérables. Pour cela, nous avons choisi d'utiliser une autre approche qui se base sur la décomposition des chiffres q_{j+1} du quotient résultat en une somme de deux ou trois termes. Chacun de ces termes peut être égal à 0 ou à un multiple entier de 2, (2^k). Ainsi la multiplication $(d \times q_{j+1})$ est réalisée par de simples décalages et une addition CSA. Ces produits partiels sont admis dans l'arbre de réduction au même titre que $\beta \times R_{sc}(j)$ le reste de la j^{ème} itération en représentation CS. Dans ce travail, on s'intéresse à des bases inférieures ou égales à 16, d'où ces termes sont choisis parmi les valeurs suivantes : $\{0, d, 2d, 4d, 8d, 16d - d, -2d, -4d, -8d, -16d\}$. Nous allons voir par la suite que l'exécution de la multiplication par cette approche peut améliorer d'une manière significative les performances de calcul de la division par la méthode SRT pour les bases 4, 8 et 16. Nous allons donner un exemple ici pour illustrer comment éviter la multiplication dans l'opération $(-d \times q_{j+1})$ par cette approche. D'abord, le chiffre $(-q_{j+1})$ est décomposé en deux ou trois termes, selon la base utilisée. Ensuite les produits partiels PPs sont calculés par de simples décalages. Si tous ces PPs sont positifs, leur réduction en CS est simple et n'exige aucune étape supplémentaire. Néanmoins, si l'un d'eux est négatif, alors sa conversion en complément à 2 devient nécessaire. Cette opération est effectuée sans délai supplémentaire.

Nous donnons ici un exemple pour le chiffre $-q_{j+1} = -3$. Sa décomposition est égale à $(-1) + (-2)$, les PPs sont $(-2d)\ et (-d)$. Les compléments à 2 de ces PPs sont obtenus en inversant tous les bits des termes positifs ($2d$ et d) et en ajoutant un 1 pour chaque produit partiel dans leur position la moins significative. Ceci est schématisé sur la figure 2.15.

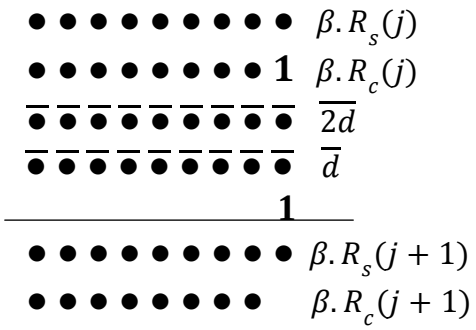


Figure 2.15 : Réduction des PPs négatifs

Dans ce qui suit, nous allons tester cette approche pour différentes bases, avec des facteurs de redondance minimale et maximale et nous allons déduire, par la suite, la base et le facteur de redondance qui donnent les meilleures performances pour le calcul de la division par la méthode SRT.

2.10.1. Division SRT en base 4 avec un facteur de redondance minimale $\rho = 2/3$

En base 4, les chiffres du quotient $(-q_{j+1})$ sont codés en signe valeur absolue sur 3 bits (q_s, q_1, q_0) et sont sélectionnés à partir de $\{-2, -1, 0, 1, 2\}$. Ainsi, les multiples du diviseur seront générés à partir du système $\{-2d, -d, 0, d, 2d\}$. Les chiffres du quotient sont obtenus par une lecture de table. Celle-ci est adressée par la troncature du reste partiel $4.\widehat{R(j)}$ et du diviseur $\hat{d}$ respectivement sur 7 et 3 bits. Le module DGM consiste en (N+1) cellules identiques. Chaque cellule de i reçoit deux bits successifs de d, (d_j, d_{j+1}) ainsi que les trois bits du chiffre quotient $(-q_{j+1})$. Les bits de ce chiffre sont utilisés pour sélectionner un des multiples du diviseur $(-2d, -d, 0, d, 2d)$. Le circuit interne de la j^{éme} cellule et son implémentation sur un slice d'un FPGA de la famille Virtex-2 sont montrés sur la figure 2.16.

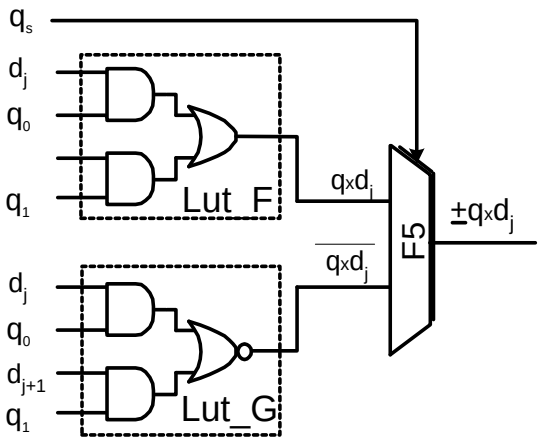


Figure 2.16 : Cellule (j) pour la génération de $(-q_{j+1} \times d)$
et son implémentation dans un Slice

Il est à noter que dans ce cas, nous n'avons pas utilisé notre approche, il n y a pas le besoin vu que les chiffres q_{j+1} sont des multiples entiers de 2.

2.10.2. Division SRT en base 4 avec un facteur de redondance maximale $\rho = 1$

Dans ce cas, la table de sélection $qSEL$ nécessite 7 bits d'adressage (5 bits de $4.R(j)$ et 2 bits de d). Les chiffres du quotient sont sélectionnés à partir du système {-3,..,

0,...,3} et codés en signe et valeur absolue sur 3 bits (q_s, q_1, q_0). Ainsi la taille de la mémoire $qSEL$ diminue d'un facteur d'environ 8 par rapport au cas précédent (Voir le tableau 2.3), néanmoins la complexité de l'implémentation de l'opération $(-q_{j+1} \times d)$ augmente. Celle-ci vient du fait que les chiffres $-q_{j+1}$ ne sont plus que des multiples de 2. Les nouveaux chiffres introduits, comparés aux cas précédents sont $-q_{j+1} = \pm 3$. Le calcul du multiple du diviseur $(-q_{i+1} \times d)$ est accompli par sa décomposition en deux termes (C0 et C1) dont leur addition est égale à $(-q_{j+1} \times d)$. Cette décomposition est illustrée dans le tableau 2.4.

Tableau 2.4 : Décomposition de $(-q_{i+1} \times d = C0 + C1)$ pour la base 4 et un ρ =1

$-q_{j+1}$	C0	C1	$-q_{j+1}$	C0	C1
0	0	0	-	-	-
1	d	0	-1	-d	0
2	0	2d	-2	0	-2d
3	d	2d	-3	-d	-2d

A partir de ce tableau, on constate que le terme C0 peut être égal à 0 ou ±d, par contre C1 est égal à 0 ou ±2d. C0 est alors sélectionné par les bits q_s et q_0 alors que C1 est sélectionné par les bits q_s et q_1.

La cellule de base du générateur des multiples du diviseur (DGM) et son implémentation sur le circuit FPGA Virtex-2 sont montrées sur la figure 2.17. Il est à noter que cette cellule occupe seulement un slice et son délai équivaut à celui d'un slice. La génération des $(-q_{j+1} \times d)$ sur deux termes, résulte en l'addition de quatre termes pour calculer les restes partiels : $R_{cs}(j+1) = 4R_s(j) + 4R_c(j) + C0 + C1$.

Leur réduction nécessite un arbre composé de deux étages d'additionneurs CSA, dont le délai total équivaut au délai d'un slice.

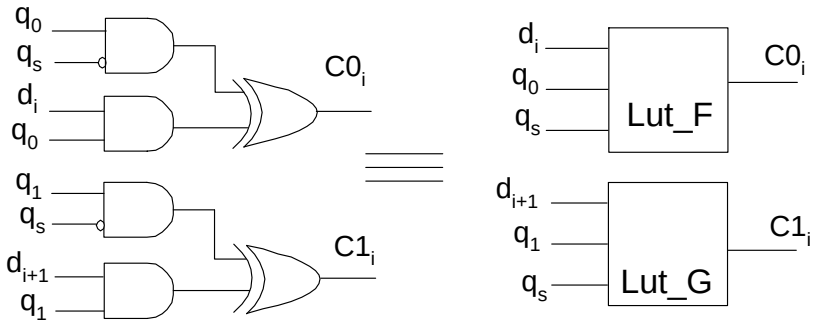


Figure 2.17 : Cellule du générateur (DGM) et son implémentation sur un slice pour $\beta = 4$ et $\rho = 1$.

Ceci a été obtenu grâce aux optimisations de ce délai par l'exploitation de la structure symétrique des slices et leurs ressources internes (2 Luts, muxcy, xorcy) [38]. L'idée de base est d'utiliser les deux luts (F, G) pour implémenter les CSAs du premier étage et les muxcys et xorcys pour les CSAs du deuxième étage. Le circuit interne d'une cellule d'indice (i) et son implémentation sur un slice sont représentés sur la figure 2.18.

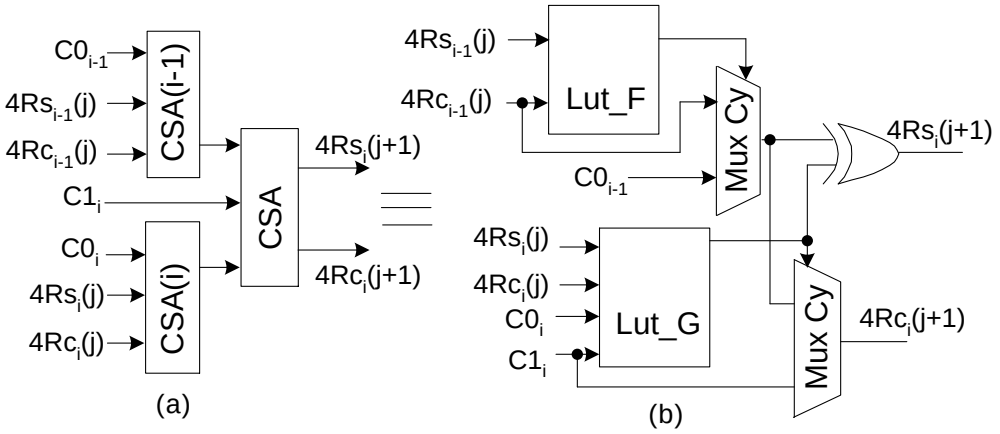


Figure 2.18 : Arbre de réduction pour $\beta = 4$ et $\rho = 1$.
(a) Structure d'une cellule.
(b) Implémentation sur un slice.

2.10.3. Division SRT en base 8 avec un facteur de redondance minimale ρ=4/7

Dans ce cas, on utilise 15 bits pour l'adressage de la table comportant les chiffres du quotient. Ces chiffres sont sélectionnés à partir du système {-4,...,4} et sont codés sur 4 bits (q_s, q_2, q_1, q_0) en signe et valeur absolue. La décomposition de ces chiffres est illustrée sur le tableau 2.5.

Tableau 2.5 : Décomposition de $(-q_{i+1} \times d = \mathrm{C0} + \mathrm{C1})$ pour la base 8 et un ρ =4/7

$-q_{i+1}$	C0	C1	$-q_{i+1}$	C0	C1
0	0	0	-	-	-
1	d	0	-1	-d	0
2	0	2d	-2	0	-2d
3	d	2d	-3	-d	-2d
4	0	4d	-4	0	-4d

Le circuit de génération des multiples de *d*, C0 et C1, est composé de (N+2) cellules identiques. Leur schéma interne est montré sur la figure 2.19. L'arbre de réduction garde la même structure que celui de la section précédente.

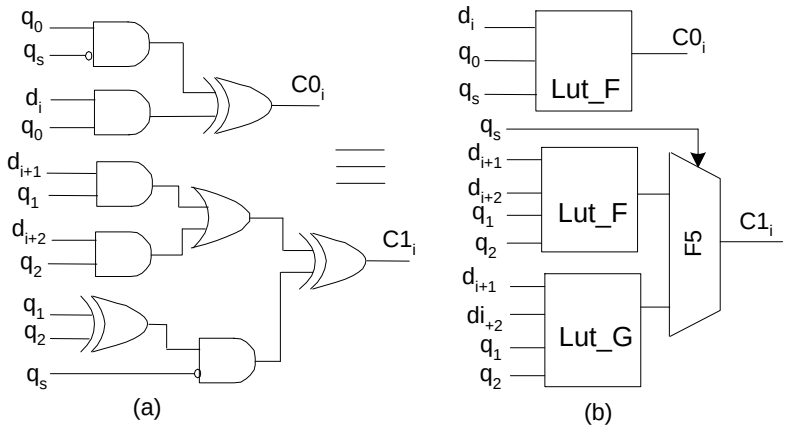


Figure 2.19 : Cellule du générateur (DGM) pour $\beta = 8$ et $\rho = 4/7$.
(a) Structure d'une cellule génération des termes C0 et C1.
(b) Implémentation sur un slice.

2.10.4. Division SRT en base 8 avec un facteur de redondance maximale ρ=1

En base 8 avec un facteur de redondance maximale, la table $qSEL$ utilise 10 bits pour la sélection des chiffres du quotient à partir du système {-7,...,7}. Cet adressage représente 7 bits des restes partiels $8R(j)$ et 3 bits du diviseur d. La décomposition des $(-q_{j+1} \times d)$ est illustrée dans le tableau 2.6.

Tableau 2.6 : Décomposition de $(-q_{i+1} \times d = \mathrm{C0} + \mathrm{C1})$ pour la base 8 et un $\rho = 1$

$-q_{j+1}$	C0	C1	$-q_{j+1}$	C0	C1
0	0	0	-	-	-
1	d	0	-1	-d	0
2	2d	0	-2	-2d	0
3	-d	4d	-3	-d	4d
4	0	4d	-4	0	-4d
5	d	4d	-5	-d	-4d
6	2d	4d	-6	-2d	-4d
7	-d	8d	-7	d	-8d

A partir de ce tableau on constate que le terme C0 correspond à une sélection des multiples 0, $\pm d$ et $\pm 2d$. Tandis que le terme C1 est pour les multiples 0, $\pm 4d$ et $\pm 8d$. Dans les sections précédentes, on a utilisé les bits du chiffre $(-q_{j+1})$ pour sélectionner les multiples de d. Alors que dans ce cas là, les bits de ce chiffre sont insuffisants pour avoir toutes les combinaisons du tableau 2.6. D'où un nouveau codage est nécessaire pour pouvoir

sélectionner les deux termes C0 et C1. Ce codage est montré dans le tableau 2.7. Le chiffre $(-q_{j+1})$ est codé sur deux valeurs de trois bits chacune, r =(r_s, r_1, r_0) et t =(t_s, t_1, t_0). Ce codage des chiffres $(-q_{j+1})$ est stocké dans des tables au même titre que les chiffres $(-q_{j+1})$.

Tableau 2.7 : Codage de $(-q_{j+1})$ en r et t pour la base 8 et un $\rho = 1$

$-q_{j+1}$	r(r_s, r_1, r_0)	t(t_s, t_1, t_0)	$-q_{j+1}$	r(r_s, r_1, r_0)	t(t_s, t_1, t_0)
0 0000	000	000	-	-	-
1 0001	001	000	-1 1001	101	000
2 0010	010	000	-2 1010	110	000
3 0011	101	001	-3 1011	001	101
4 0100	000	001	-4 1100	000	101
5 0101	001	001	-5 1101	101	101
6 0110	010	001	-6 1110	110	101
7 0111	101	010	-7 1111	001	110

La cellule de base qui permet la génération des multiples de d et son implémentation sur le circuit FPGA Virtex-2 sont montrés sur la figure 2.20. Comme la génération de ces multiples engendre seulement deux termes alors l'arbre de réduction garde la même structure que celle de la section 2.10.2.

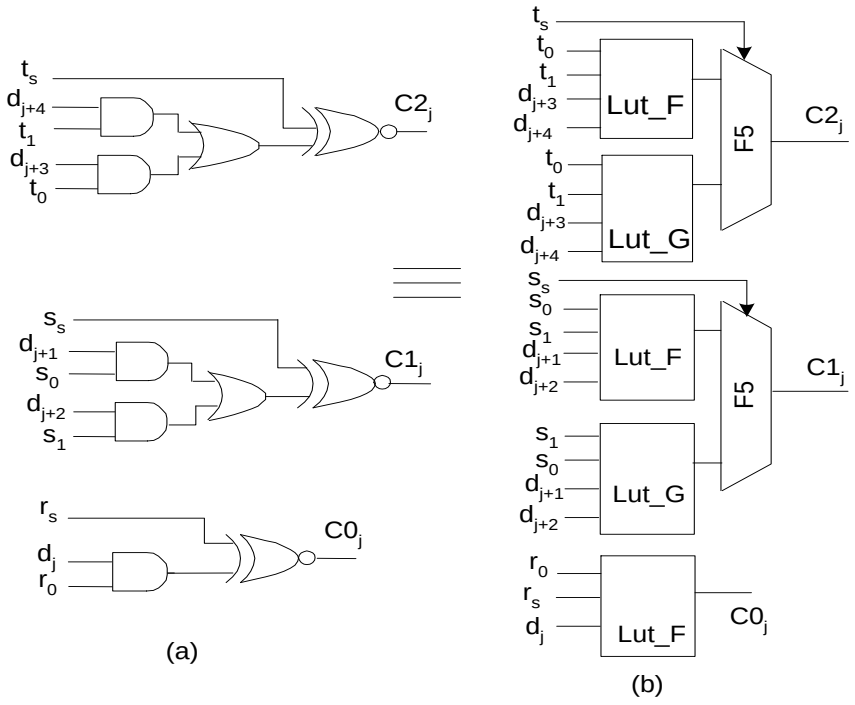


Figure 2.20 : Cellule du générateur (DGM) pour $\beta = 8$ et $\rho = 1$.
(a) Structure d'une cellule de génération des termes C0 et C1
(b) Implémentation sur un slice.

2.10.5. Division SRT en base 16 avec un facteur de redondance maximale ρ=1

En base 16 avec un facteur de redondance maximale, 13 bits sont utilisés pour l'adressage des chiffres du quotient à partir du système {-15,...,15}. Ces chiffres sont codés sur 5 bits. La décomposition de $(-q_{j+1} \times d)$ en trois termes (C0, C1 et C2) est illustrée dans le tableau 2.8.

Tableau 2.8 : Décomposition de $(-q_{j+1} \times d)$ = C0+C1+C2 pour la base 16 et un ρ =1

$-q_{j+1}$	*C0*	*C1*	*C2*	$-q_{j+1}$	*C0*	*C1*	*C2*
0	0	0	0	0	-	-	-
1	d	0	0	-1	-d	0	0
2	0	2d	0	-2	0	-2d	0
3	d	2d	0	-3	-d	-2d	0
4	0	4d	0	-4	0	-4d	0
5	d	4d	0	-5	-d	-4d	0
6	0	-2d	8d	-6	0	2d	-8d
7	-d	0	8d	-7	d	0	-8d
8	0	0	8d	-8	0	0	-8d
9	d	0	8d	-9	-d	0	-8d
10	0	2d	8d	-10	0	-2d	-8d
11	d	2d	8d	-11	-d	-2d	-8d
12	0	4d	8d	-12	0	-4d	-8d
13	d	4d	8d	-13	-d	-4d	-8d
14	0	-2d	16d	-14	0	2d	-16d
15	-d	0	16d	-15	d	0	-16d

De la même façon que dans la section précédente les bits du chiffre $(-q_{j+1})$ sont insuffisants pour sélectionner les multiples de d, d'où le codage de chacun des chiffres $(-q_{j+1})$ en trois valeurs r =(r_s, r_0), s = (s_s, s_1, s_0) et t = (t_s, t_1, t_0) est nécessaire. Ce codage est illustré dans le tableau 2.9.

Les valeurs r, s et t sont stockées dans une table utilisant le même adressage que la table $qSEL$ des chiffres $(-q_{j+1})$. Le module de génération de ces multiples de d consiste en (N+4) cellules.

Tableau 2.9 : Codage de $(-q_{j+1})$ en r, s et t pour la base 16 et un ρ =1

$-\mathbf{q_{j+1}}$	$\mathbf{r=r_s,r_0}$	$\mathbf{s=s_s,s_1,s_0}$	$\mathbf{t=t_s,t_1,t_0}$	$-\mathbf{q_{j+1}}$	$\mathbf{r=r_s,r_0}$	$\mathbf{s=s_s,s_1,s_0}$	$\mathbf{t=t_s,t_1,t_0}$
0	00	000	000	-	-	-	-
1	01	000	000	-1	11	000	000
2	00	001	000	-2	00	101	000
3	01	001	000	-3	11	101	000
4	00	010	000	-4	00	110	000
5	01	010	000	-5	11	110	000
6	00	101	001	-6	00	001	101
7	11	000	001	-7	01	000	101
8	00	000	001	-8	00	000	101
9	01	000	001	-9	11	000	101
10	00	001	001	-10	00	101	101
11	01	001	001	-11	11	101	101
12	00	010	001	-12	00	110	101
13	01	010	001	-13	11	110	101
14	00	101	010	-14	00	001	110
15	11	000	010	-15	01	000	110

Le circuit logique d'une cellule (i) et son implémentation sur le circuit Virtex-2 sont montrés sur la figure 2.21.

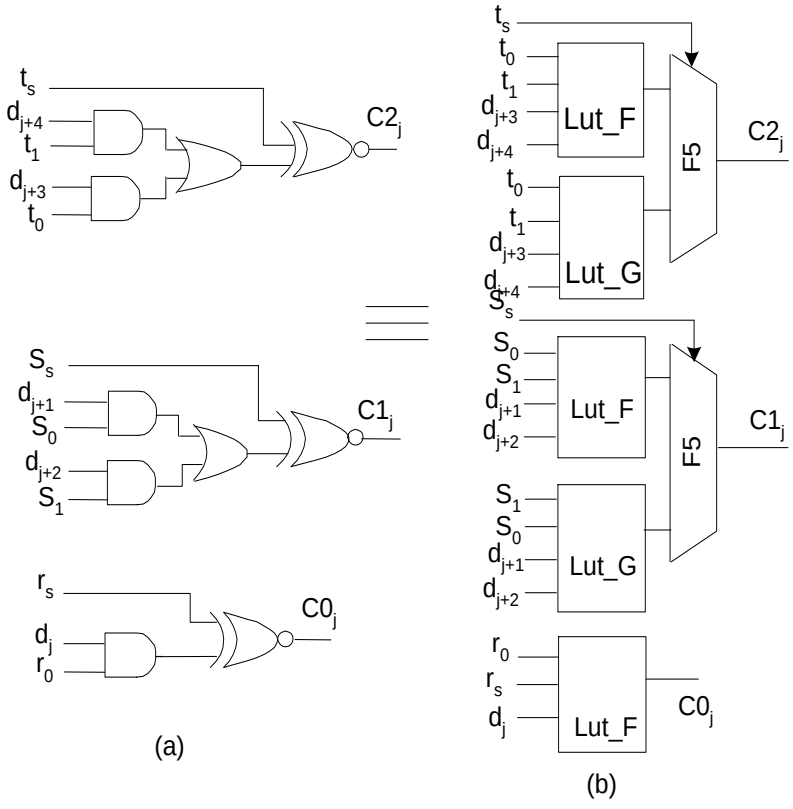


Figure 2.21 : Cellule du générateur (DGM) pour $\beta = 16$ et $\rho = 1$.
(a) Structure d'une cellule de génération des termes C0, C1 et C3
(b) Implémentation sur un slice.

Dans ce cas là, le calcul des restes partiels est exprimé comme suit :

$$R_{cs}(j+1) = 16R_s(j) + 16R_c(j) + C0 + C1 + C2.$$

L'analyse de cette expression, montre que sa réduction nécessite un arbre composé de trois étages d'additionneurs CSA. Pour diminuer le délai de l'arbre de réduction, nous avons introduit le compresseur 4:2 [39]. Grâce à une optimisation dans l'implémentation de ce compresseur, nous avons pu rendre le délai de l'arbre de réduction équivaut au délai de deux slices. Le circuit logique d'une cellule ainsi que son placement sur FPGA sont montrés sur la figure 2.22.

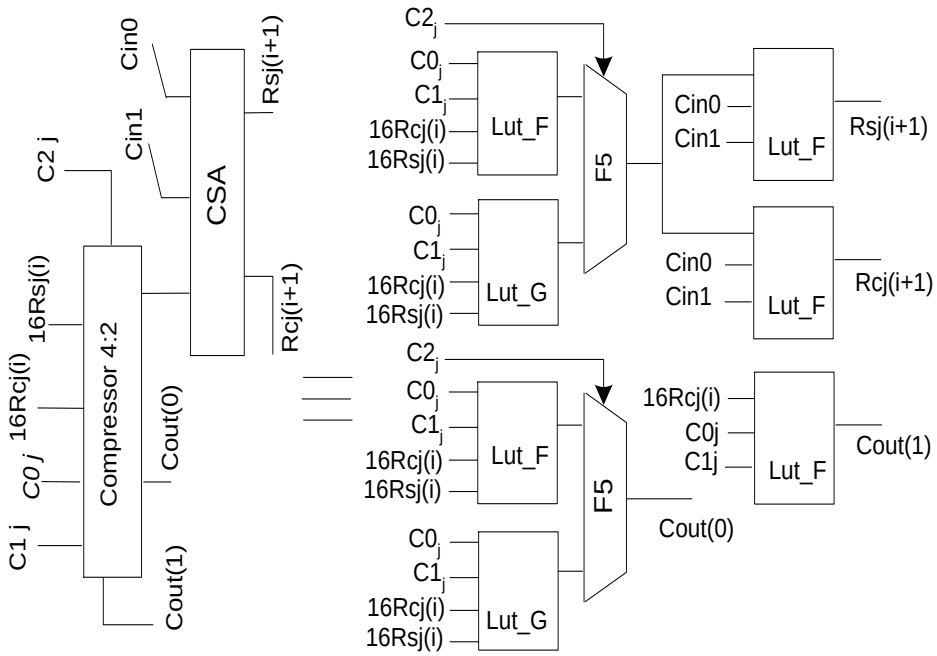


Figure 2.22 : Cellule élémentaire de l'arbre de réduction et son placement sur un Virtex-2 (pour $\beta = 16$ et $\rho = 1$)

2.11. Division SRT à base des blocs multiplieurs 18×18 bits

L'implémentation de la division SRT sur un des circuits FPGAs récents peut être réalisée en utilisant les blocs multiplieurs 18×18 bits pour l'opération $(-q_{j+1} \times d)$ et les chemins dédiés à la retenue *(Carry Chain)* pour l'addition. Cette approche est généralement recommandée pour la réduction du temps de conception. Le principal intérêt de représenter les restes partiels en non redondant réside dans la simplification de la sélection des chiffres du quotient, par l'élimination de l'additionneur chargé de générer les adresses à partir des restes partiels et la diminution du nombre de registres utilisés pour synchroniser le calcul des restes partiels. L'architecture correspondante est représentée sur la figure 2.23.

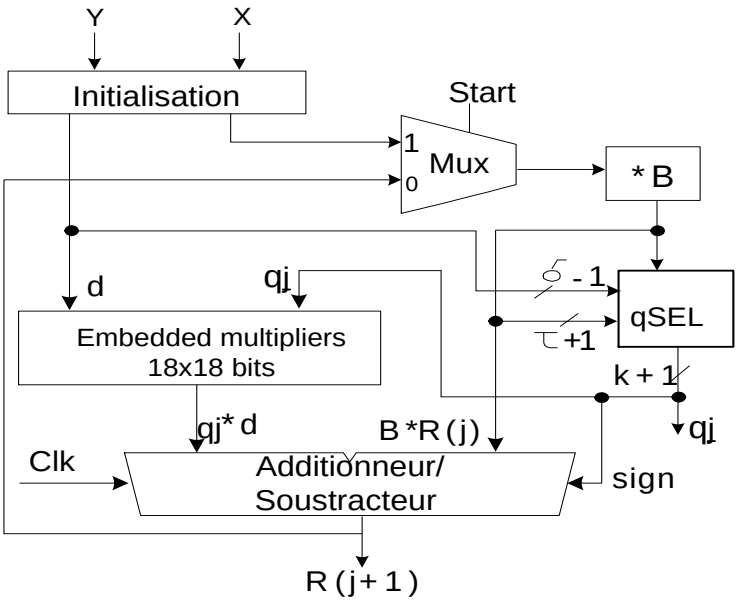


Figure 2.23 : Division SRT à base de blocs multiplieurs 18×18 bits.

2.12. Résultats d'implémentations et Comparaisons

Les architectures que nous avons développées dans ce chapitre ont été conçues dans l'environnement (ISE 7.1i) de Xilinx. Les différents modules constituants ces architectures ont été décrits en code VHDL ou générés par l'outil *CORE GENERATOR* (pour les multiplieurs 18×18 bits). Pour les architectures utilisant notre approche, nous avons introduit certaines contraintes temporelles et de placement, afin de minimiser le délai et le routage.

Dans le but de garantir le fonctionnement correct, ces architectures ont été simulées par l'outil ModelSim XE III 6.0a et synthétisées par XST *(Xilinx Synthesis Tool)*. Pour permettre une bonne comparaison, toutes ces architectures ont été implémentées sur un même circuit FPGA Virtex-2 (XC2V1000-6).

Dans cette section, nous allons faire une comparaison entre toutes les architectures présentées ci-dessus. En premier lieu cette comparaison portera sur la surface totale occupée par chacune de ces architectures en termes de mémoire (qSEL) et de nombre total de slices. Puis, nous allons comparer les performances de notre meilleure architecture à des travaux similaires antérieurs.

Les tailles des mémoires ainsi que les surfaces occupées, en termes de slices, des architectures présentées dans les sections précédentes sont illustrées sur le tableau 2.10.

Tableau 2.10 : Tailles des mémoires et surfaces occupées

base	4		8		16	8*
ρ	2/3	1	4/7	1	1	1
qSel (kbits)	3.1	0.4	131.1	10.3	106.5	10.3
Surface (slices)	608	377	4498	565	2140	343
Nb de multiplieurs18*18 bits	-	-	-	-	-	4

8 Résultats concernant l'architecture utilisant les blocs multiplieurs18*18 bits.*

Il est à noter que la surface (en slices) concerne seulement la logique, alors que la mémoire qSel a été implémentée dans les blocs mémoires SRAM.

A travers ces résultats, nous constatons que le facteur ρ a une grande influence, puisque son augmentation entraîne une réduction considérable des tailles des mémoires pour toutes les bases, alors que la complexité engendrée par l'augmentation du facteur de redondance est circonvenue par notre approche et ne couvre même pas la surface gagnée dans la réduction des surfaces occupées par les mémoires. L'utilisation de notre approche pour la base 8 avec un facteur de redondance maximale représente le meilleur gain en surface par rapport à l'utilisation d'un facteur de redondance minimale.

Les délais d'une itération SRT ainsi que les délais d'exécution de la division pour des opérandes représentés en double précision de la norme IEEE-754 de nos architectures sont données dans le tableau 2.11.

Tableau 2.11 : Performances temporelles

base	4		8		16	8*
ρ	2/3	1	4/7	1	1	1
Délai d'une itération (ns)	12.2	**10.8**	22.5	11.17	19.4	17.8
Délai total (ns)	329.4	291.6	405.6	**211.9**	271.7	303.3

8 Résultats concernant l'architecture utilisant les blocs multiplieurs18*18 bits.*

On remarque que bien que le temps d'exécution d'une itération est le meilleur pour l'architecture avec la base 4 et un facteur de redondance maximale alors que le meilleur délai d'exécution de la division est celui de l'architecture avec la base 8 et un $\rho = 1$. Ceci conclut que le meilleur compromis performances/surface occupée est dans l'utilisation d'une base 8 et un facteur de redondance maximale.

Par ailleurs, on note que les performances des architectures développées dans le cadre de notre approche sont indépendantes de la taille des opérandes.

Malgré la réduction de la surface de (39%) dans l'architecture qui utilise des multiplieurs embarqués (sans prendre en compte la surface des multiplieurs 18×18bits), le délai de l'itération de cette architecture est plus qu'une fois et demi plus grand que notre meilleur architecture. En plus, le délai de l'itération de cette architecture dépend de la taille des opérandes, puisqu'elle utilise un multiplieur et un additionneur dont les tailles sont égales à celle des opérandes.

Concernant l'approche reportée dans [35], celle-ci utilise moins de slices que toutes nos architectures, néanmoins elle est la moins rapide que toutes les architectures avec un ρ =1, et en plus elle utilise des blocs multiplieurs 18×18 bits comme autre ressource matérielle qui n'est pas comptabilisée dans cette comparaison.

A titre de comparaison, nous allons d'abord comparer notre architecture proposée au DIVGEN de Marchand et al., publiés dans [40], les auteurs ont présenté un outil pour générer une unité matérielle pour le calcul de la division. Dans leur travail, ils ont développé des architectures basées sur l'utilisation de la décomposition de q_{j+1} pour contourner la multiplication $q_{j+1} \times d$ et ils ont utilisé un additionneur à propagation de la retenue. Les architectures résultantes ne sont pas optimisées et les délais de leurs itérations dépendent de la taille des opérandes. Le meilleur délai d'une itération, pour une précision de 32 bits (qui est moins que la notre qui est de 53 bits), est de 18,7 ns. Ce résultat est pour la SRT en base égale à 4, ce qui signifie plus d'itérations que notre architecture, ce qui veut dire un temps d'exécution de la division plus important.

Le second travail considéré dans cette comparaison est celui de Lee et Burgess [41]. Dans ce travail, les auteurs ont présenté une unité paramétrée pour le calcul de la division, la multiplication et la racine carrée de deux entiers en virgule fixe. Ils ont utilisé un algorithme SRT avec une base=8. Dans leur algorithme, ils réduisent la taille de la mémoire qSel, en opérant une réduction d'argument. Leur implémentation est optimisée par l'utilisation des blocs multiplieurs 18×18bits pour la multiplication et de l'utilisation du *carry chain* pour les additionneurs/Soustracteurs. L'inconvénient de leurs architectures est que le délai de l'itération dépend de la taille des opérandes. Avec cette architecture, une division en double précision est effectuée en 225 ns qui est plus important que celui de notre architecture qui est de 210,6 ns.

2.13. Conclusion

Dans ce chapitre, nous avons présenté une méthode matérielle pour le calcul de la division en double précision de la norme IEEE-754 basé sur l'algorithme SRT. Nous avons aussi montré que l'utilisation d'un facteur de redondance maximale conjugué à une utilisation judicieuse des ressources internes du circuit FPGA peut mener à des architectures performantes. L'approche que nous avons développée dans ce chapitre est basée sur la décomposition des chiffres du quotient en deux ou trois termes, qui ne sont autre que des multiples entiers de deux. Il en résulte que la multiplication par l'un de ces chiffres est obtenue par de simples décalages. Ensuite ces termes sont admis dans l'arbre de réduction au même titre que le reste intermédiaire. Par ailleurs la représentation CS de ce dernier est avantageusement utilisée ; ce qui nous a permis de réaliser l'itération SRT sans propagation de retenue. Ce qui fait que les délais des architectures développées par notre approche sont indépendants de la taille des opérandes. En plus l'optimisation faite sur l'arbre de réduction a fait que le délai introduit par cet arbre n'excède pas le délai de deux slices.

Une étude comparative pour le calcul de division pour les trois bases (4, 8 et 16) avec divers facteurs de redondance a été effectuée afin de déterminer l'effet et l'influence de ces paramètres sur la division SRT. A travers cette comparaison, nous avons vu que les bases 4 et 8 avec un facteur de redondance maximale présentent le meilleur temps d'exécution d'une itération. Cependant le meilleur temps d'exécution d'une division globale est celui de l'architecture utilisant la base 8 avec un facteur de redondance maximale. Cette architecture présente aussi le meilleur compromis entre le délai d'exécution et la surface occupée.

CHAPITRE 3
CALCUL DES FONCTIONS ELEMENTAIRES

3.1. Introduction

L'exponentielle, les sinus/cosinus et beaucoup d'autres fonctions élémentaires à un seul argument sont très utilisées en traitement de signal, multimédia et calculs scientifiques [42]. Bien que le recours à ces opérations soit moins fréquent que pour l'addition et la multiplication, ces fonctions présentent une grande latence dans beaucoup de processeurs actuels. D'où l'évaluation rapide de ces fonctions élémentaires a toujours été d'une grande importance.

Les fonctions élémentaires sont essentiellement implémentées en logiciel [43], ce qui résulte en des temps d'exécution trop longs qui ne répondent pas aux exigences des applications temps réel et de la haute performance. Ces contraintes de performances conjuguées à la grande disponibilité du silicium, pour les circuits VLSI et les circuits FPGA, ont motivé le développement d'algorithmes orientés matériel pour venir à bout des performances requises par les applications numériquement intensives où le recours à ces fonctions élémentaires est plus fréquent. Les méthodes d'évaluation des fonctions élémentaires sont diverses [44] et dépendent de la quantité de matériel que l'on est prêt à leur consacrer.

Le moyen le plus rapide pour évaluer une fonction est l'utilisation d'une table. Néanmoins, la mémoire requise devient vite prohibitive avec l'augmentation de la précision. Certaines méthodes sont basées sur l'utilisation d'addition et de décalages (c'est le cas de l'algorithme CORDIC [45]). Ce sont des algorithmes itératifs qui donnent lieu à des opérateurs petits, mais qui ont une forte latence à cause de leur convergence linéaire.

Les fonctions élémentaires sont généralement calculées en utilisant des approximations polynomiales [46] qui au lieu d'évaluer la fonction on évalue un polynôme dont le degré croit

avec la précision, par conséquent l'évaluation d'un nombre important d'additions et de multiplications, ce qui se traduit par des temps d'exécution très importants.

Cette dernière décade a vu l'émergence de nouvelles méthodes qui combinent lecture de table et évaluation de polynôme de faible degrés [47, 48, 49]. Celles ci présentent un bon compromis entre les exigences en mémoire et en calculs numériques. Leur utilisation, réduit sensiblement la taille des tables (comparées à la méthode tabulaire) et accélère la vitesse de calcul (comparées aux approximations polynomiales). Ces algorithmes de calcul à l'aide de tables ont été suscités par de nouvelles applications (comme la synthèse de l'image 3D, la réalité virtuelle et les jeux sur ordinateur) gourmandes en débit de calcul et favorisées par le progrès technologique (on peut réaliser des tables de plus en plus grandes). Les premiers algorithmes développés pour ce type de calcul étaient basés sur la lecture de tables et un développement de Taylor d'ordre 1. Ces algorithmes ne nécessitent pas beaucoup de calculs telle que la méthode bipartite, tripartite ou multipartite [50, 51, 52] et la méthode dite approximation linéaire. Avec l'élargissement de ce type de méthodes aux applications qui nécessitent plus de précision, ces algorithmes se sont avérés difficiles (voir impossible dans certains cas) à implémenter sur silicium. Ceci est dû à la grandeur des tables que requièrent ces méthodes. D'autres méthodes ont été développés, celles ci sont toujours basées sur une lecture de table, mais utilisent des polynômes d'ordre deux [53, 54, 55], le principe est de diminuer la taille des tables pour augmenter la masse des calculs. En général, les méthodes à base de tables sont sujettes à de compromis entre la taille des tables et la complexité des calculs.

Les circuits ASICs *(Application Specific Integrated Circuit)* procurent une bonne solution pour l'implémentation de ces méthodes sur matériel, où souvent la surface occupée est sacrifiée au détriment des performances. Cependant, avec l'augmentation de la précision, il devient difficile d'implémenter plusieurs fonctions sur le même chip. Les progrès dans le domaine des circuits programmables FPGAs ont permis de nouvelles options dans l'évaluation matérielle des fonctions élémentaires. Ces circuits ont désormais une capacité telle qu'ils peuvent être utilisés à des tâches d'accélération de calcul en virgule flottante.

Le circuit FPGA offre, en plus des performances temporelles d'un circuit dédié, la flexibilité d'une architecture reconfigurable et d'un faible coût de développement [56]. Cette reconfigurabilité devient de plus en plus indispensable du moment que beaucoup de systèmes calculent plusieurs fonctions mais pas toutes en même temps.

Par ailleurs, tous ces progrès technologiques ont permis aux performances des circuits intégrés numériques de continuer leur progression exponentielle. En parallèle, on constate une

explosion du coût du développement, coût qui devrait continuer à augmenter dans un avenir proche. Il devient alors important de concevoir des unités de calcul qui seront réutilisables dans plusieurs applications et sur plusieurs technologies

Depuis quelques années, l'industrie des circuits intégrés utilise de plus en plus des cœurs et des blocs IP (*Intellectual Property*). Ces cœurs ou blocs sont des descriptions de diverses fonctionnalités qui peuvent être implantées sur différentes cibles. Leur conception est devenue une industrie à part entière. Toutefois, seuls des blocs avec des fonctionnalités figées et peu variées sont disponibles actuellement.

Le but de notre travail dans cette partie de thèse est alors de réaliser une architecture qui peut être utilisée comme un IP pour le calcul des fonctions élémentaires. Cette architecture doit être performante, dédiée aux calculs intensifs et n'utilisant que les ressources de base des circuits FPGA pour pouvoir être implémentée dans une large gamme de circuits FPGA. En plus d'être performante celle-ci doit être reconfigurable pour pouvoir calculer un certain nombre de fonctions avec un même niveau de précision (1 ulp) (*Unit in Last Place*). Pour ce faire, cette architecture doit consommer un même niveau de ressources matériel pour le calcul de toutes les fonctions considérées. Ce travail exige alors une certaine standardisation à tous les niveaux de conception, algorithmique, architecturale et implémentation. Six fonctions sont considérées dans ce travail à savoir l'exponentielle (e^x), le logarithme $ln(x)$, l'inverse $(1/x)$, la racine carrée $\sqrt{x}$, le sinus $sin(x)$ et le cosinus $cos(x)$.

Dans ce travail, nous exploitons les capacités des circuits FPGA pour concevoir une architecture performante et reconfigurable pour le calcul des fonctions élémentaires, en double précision de la norme IEEE-754. La méthode proposée s'applique au calcul de la fonction exponentielle et par reprogrammation du FPGA, les fonctions telles que le logarithme, le sinus, le cosinus, la racine carrée et l'inverse sont calculées. Notre stratégie est d'adapter notre architecture aux ressources du circuit FPGA sélectionné. La première étape est de fixer le degré du polynôme selon la mémoire disponible dans le circuit FPGA. Concernant la partie calcul, l'implémentation de notre méthode est basée sur une architecture pipeline utilisant des FMAs (*Fused Multiply Adder*) optimisés au niveau le plus bas (c.-à-d. niveau slice) des circuits FPGA de la famille Virtex-2 [57], qui sont optimisés pour la haute intégration et les hautes performances.

Différentes optimisations à plusieurs niveaux ont été effectuées pour mener à bien notre travail. En premier lieu on s'est penché sur la réduction du degré du polynôme d'approximation, vu que la complexité des calculs croit exponentiellement avec le degré du polynôme. Un autre facteur qui joue dans la complexité des calculs est la largeur de

l'intervalle d'approximation, différentes techniques ont été utilisées pour unifier cette complexité calculatoire, pour toutes les fonctions considérées, ce qui s'est traduit par des architectures qui ont un même niveau de consommation de ressources matériel. Une bonne connaissance des ressources internes du FPGA ont été plus que nécessaire pour pouvoir opérer des optimisations au niveau le plus bas à savoir au niveau CLB.

Le calcul d'erreur nous a permis d'optimiser la taille des chemins des données (*data paths)*, Ceci a rehaussé les performances et a diminué la surface occupée de nos architectures.

Une autre optimisation a touché la partie mémoire, vu que celle-ci demeure le point culminant dans le choix du circuit FPGA. Comme l'architecture développée dans ce travail est appelée à être utilisée comme un IP ou cœur de coprocesseur pour l'accélération du calcul des fonctions élémentaires, celle-ci ne doit pas consommer toutes les ressources du FPGA. Pour ce faire nous avons choisi d'utiliser que la mémoire disponible dans les blocs BRAM, sans toutefois utiliser celle qui est dans les CLB. Le calcul d'erreur a permis aussi de tronquer les coefficients des polynômes, sans toutefois dépasser la précision permise qui est de 1ulp [58]. Il en résulte des coefficients avec des tailles réduites, ce qui contribue à la diminution de la taille totale des tables des coefficients et par conséquent la mémoire consommée par nos architectures.

3.2. Approximation Polynomiale

L'approximation polynomiale des fonctions est d'une grande utilité pour l'évaluation des fonctions élémentaires. Elle consiste à remplacer une fonction $f(x)$ par un polynôme $P_n(x)$ facile à calculer, puisque n'interviennent que des puissances entières de x, ainsi que des multiplications et des additions/soustractions.

Pour évaluer une fonction élémentaire $f(x)$, il suffit de l'approximer par un polynôme $P_n(x)$ de degré n sur un intervalle fixe $[a, b]$ qui à la forme suivante :

$$P_n(x) = C_0 + C_1.x^1 + \ \dots \dots \dots + \ C_{n-1}.x^{n-1} + C_n.x^n$$

Il y a plusieurs formes de polynômes d'approximations [59]. Celles-ci ont des propriétés différentes et peuvent être classées en deux groupes :

Les approximations qui minimisent l'erreur moyenne telle que Tchebychev et Legendre et les approximations qui minimisent l'erreur maximale telle que Taylor et Minimax.

L'erreur d'approximation e_{appro} est une bonne mesure de l'exactitude de cette approximation et elle est définie comme étant la distance entre $P_n(x)$ et $f(x)$.

$$e_{appro} = \| P_n(x) - f(x) \|$$

Les approximations qui minimisent l'erreur moyenne ne sont pas souvent employées parce que, bien que l'erreur moyenne soit petite, il peut y avoir une grande erreur en un point x_q de l'intervalle $[a, b]$. Par contre les approximations qui minimisent l'erreur maximale sont largement utilisées puisque l'erreur sur tout l'intervalle, demeure inférieure à l'erreur maximale. Celle-ci est définie comme suit :

$$e_{appro(\max)} = max \left\| P_n(x) - f(x) \right\| \quad \text{pour } x \in [a, b]$$

3.2.1. Approximation de Taylor

L'approximation de Taylor est l'une des méthodes les plus utilisées dans le domaine de l'interpolation polynomiale. Son intérêt primordial, réside dans sa simplicité par rapport aux autres méthodes. Il est facile de constater, que l'obtention des coefficients du polynôme de Taylor est assez simple, puisque d'une part leurs formules ne dépendent que des dérivés de la fonction, et d'autre part, elle n'utilise qu'un seul point au voisinage duquel elle sera calculée. On peut remarquer aussi, que lorsqu'on augmente le degré du polynôme, l'expression des coefficients d'ordre inférieur reste la même.

Supposons que $f(x)$ admet des dérivées continues jusqu'à l'ordre $(n + 1)$, sur l'intervalle [a, b] et $x_0 \in$ [a, b]. Nous rappelons alors le résultat suivant :

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{(x-x_0)^2}{2}.f''(x_0) + \cdots \frac{(x-x_0)^n}{(n!)}.f^{(n)}(x_0) + \varepsilon(x)$$

Cette formule représente le polynôme de Taylor, qui approche la fonction $f(x)$ au voisinage de x_0, avec une erreur de $\varepsilon(x)$. Cette dernière formule peut encore s'écrire sous la forme :

$$f(x) = P(x) + \varepsilon(x)$$

où $P(x)$ est le polynôme de Taylor et $\varepsilon(x)$ est l'erreur commise sur $f(x)$, cette erreur est définie comme suit :

$$\varepsilon(x) = \frac{(x-x_0)^{n+1}}{(n+1)!}.f(n+1)(\xi)$$

avec $\xi \in \ ]min(x_0, x),\ max(x_0, x)[$

3.2.2. Approximations Minimax

Il existe une méthode qui offre la meilleure approximation polynomiale possible d'une fonction sur un intervalle donné avec un degré donné. C'est l'approximation qui minimise l'erreur maximale entre la fonction et le polynôme sur un intervalle. Cette approximation, appelé approximation ***minimax***, permet d'obtenir la précision voulue avec un polynôme de

degré plus faible dont l'évaluation demandera donc moins de calculs. Cependant le calcul des coefficients du polynôme d'approximation minimax est une tâche difficile nécessitant une procédure itérative connue sous le nom d'algorithme de Remez [60].

Avec la puissance de calcul des machines actuelles, le calcul des coefficients du polynôme *minimax* de degré modéré est devenu beaucoup plus abordable grâce au logiciel de calcul formel MAPLE. Dès que le degré du polynôme augmente, le calcul de ces coefficients nécessite une très grande puissance de calcul, ce qui rend l'utilisation de ce polynôme non indiquée pour les très grandes précisions. De plus, il n'y a aucun lien entre les coefficients de deux polynômes d'approximations minimax de degrés différents. Cela signifie que quand la précision voulue n'est pas fixée au départ mais choisie par l'utilisateur, il faut un polynôme pour chacune des précisions possibles ce qui implique l'utilisation d'un grand nombre de polynômes, ce qui résulte en une table de taille importante pour stocker les coefficients des différents polynômes. Ce n'est pas le cas des polynômes de Taylor, qui sont constituées d'une unique série entière que l'on tronque plus ou moins loin. Malgré cela l'approximation minimax est un choix incontournable pour les applications où la précision est fixe et le polynôme est de faible degré.

L'approximation Minimax est définie par :

Pour tout $x \in [a,b]$, $max\left|f(x) - P_n^*(x)\right| = min\,(max\,\left\|f(x) - P_n(x)\right\|)$

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

Le polynôme P_n^* minimise l'erreur maximale sur l'intervalle [a, b] sur lequel on cherche à approcher la fonction $f(x)$.

La bibliothèque d'approximation ***numapprox,*** du logiciel MAPLE, offre une fonction *minimax* proposant une approximation du polynôme minimax.

La commande :

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

Celle-ci donne la meilleure approximation rationnelle $Q_{p/q}$ dont le numérateur est de degré p, le dénominateur de degré q, de la fonction $f(x)$ sur l'intervalle $[a,b]$, avec une fonction de poids $w(x)$, où l'erreur est obtenue en err.

L'approximation minimax d'une fonction f sur un intervalle fixe $[a, b]$ est le polynôme de degré n qui réduit au minimum l'erreur maximum sur cet intervalle.
Les arguments nécessaires pour le calcul d'un polynôme d'approximation minimax sont :

- La fonction à approximer.
- L'intervalle de définition de la fonction ou l'intervalle d'approximation.
- L'ordre requis du polynôme d'approximation ou la précision nécessaire.

Le grand avantage dans l'utilisation de l'approximation minimax se situe dans le fait, que pour une précision donnée, on obtient le polynôme de degré le plus faible. Ce qui conduit à un polynôme avec un nombre de termes réduit et par conséquent un temps de calcul réduit.

3.2.3. Approximation par segments

Une autre réduction du degré du polynôme est possible par la réduction de la largeur de l'intervalle d'approximation $[a, b]$ et par conséquent le nombre de termes du polynôme et la complexité calculatoire. Le principe de l'approximation par segments est de découper l'intervalle de calcul $[a, b]$ en plusieurs sous intervalles $[a_i, b_i]$ et au lieu d'utiliser un seul polynôme $P_n(x)$, avec un degré élevé, pour approximer la fonction $f(x)$ sur tout l'intervalle $[a, b]$, on utilisera plusieurs polynômes $P_i(x)$ pour approximer la fonction $f(x)$ dans chacun des intervalles $[a_i, b_i]$. Cette méthode est illustrée sur la figure 3.1.

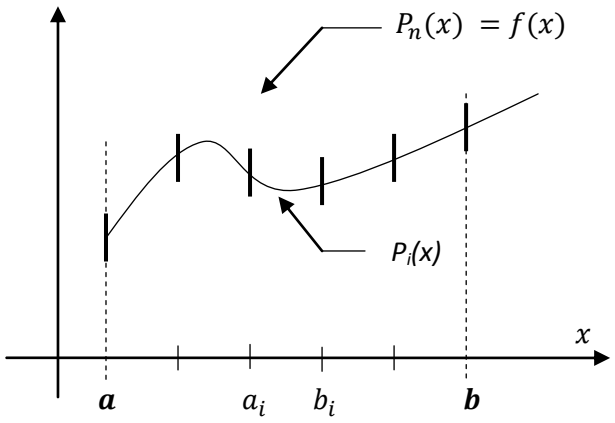


Figure 3.1 : Approximation par segments

Il en résulte que les polynômes $P_i(x)$ sont de degré faible dont la complexité calculatoire est mois importante que celle de $P_n(x)$. Il est à noter que plus on a de sous-intervalles $[a_i, b_i]$, plus le degré des polynômes $P_i(x)$ est faible. Cette méthode est appelée approximation par

segments [61].

Cette technique conduit certes à une réduction de la complexité du hardware et par conséquent réduit le délai de calcul, néanmoins la mémoire requise pour stocker les coefficients des différents polynômes $P_i(x)$ augmente.

3.3. La Réduction d'argument

La réduction d'argument est la première étape dans l'évaluation des fonctions élémentaires. Les performances globales de l'algorithme d'évaluation ainsi que la précision du résultat dépendent en grande partie de la rapidité et de la précision de cette étape.

En effet, les algorithmes utilisés pour évaluer les fonctions élémentaires, et basés sur une évaluation polynomiale, ne sont corrects que si l'argument appartient à un petit intervalle habituellement centré sur zéro.

Comme les fonctions élémentaires ont des domaines de définition infinis, ils doivent être réduits à l'intervalle de convergence de l'algorithme, en utilisant des transformations mathématiques sur l'argument initial. Ce processus s'appelle la *réduction d'argument.*

L'approximation de la fonction se fait sur un intervalle d'entrée $[a, b]$ et la réduction d'argument se calcule pour des valeurs se trouvant à l'extérieur de cet intervalle.

Pour les fonctions élémentaires, il y'a une sélection naturelle des intervalles $[a, b]$ qui simplifie en même temps les étapes de réduction d'argument et d'approximation de la fonction. Ce qui mène à des tables plus petites, pour le stockage des coefficients des polynômes de petits degrés approximant la fonction et à des temps d'exécution plus rapides.

Cette réduction d'argument naturelle est basée sur les propriétés de certaines fonctions élémentaires, qui permettent au domaine de définition de ces fonctions d'être réduit au minimum telles que : ***la périodicité***, ***les théorèmes d'addition*** et ***la symétrie***, etc [62], [63], [64].

- Périodicité

Nous savons que les fonctions trigonométriques sont périodiques et que leur période est de 2π, où $sin\,(x + 2\pi) = sin(x)$, $cos(x + 2\pi) = cos(x)$

Donc le domaine de définition des fonctions trigonométriques est automatiquement réduit à $[0, 2\pi]$.

- Symétrie

Nous utilisons les propriétés de la symétrie de $\boldsymbol{sin(x)}$ pour réduire d'avantage l'intervalle d'approximation. D'abord $\boldsymbol{sin(x)}$ est asymétrique autour du point $\boldsymbol{x = \pi}$ pour l'intervalle $[\mathbf{0}, \mathbf{2}\pi]$. Alors, nous utilisons la relation ci-dessous pour réduire l'intervalle d'approximation à $[\mathbf{0}, \pi]$. $\boldsymbol{sin(x + \pi) = -sin\,(x)}$,

Comme $sin(x)$ est aussi symétrique autour du point $x = \pi/2$ pour l'intervalle $[0, \pi]$, ce qui donne : $sin(\, x + \pi/2) \;=\; sin(\pi/2 - x)$, quand $x \in [0, \pi/2]$,

Ce qui réduit encore l'intervalle d'approximation à $[0, \pi/2]$.

- Formules d'Addition

Une autre propriété fonctionnelle qui permet une extension illimitée de l'intervalle d'approximation est la formule d'addition.

Des exemples typiques de cette formule sont : $\boldsymbol{e^{n+x} \;=\; e^{n}\; e^{x}}$

$$log(mx) \;=\; \log(m) + \log(x) = k \times log2 + \log(x), \quad o\grave{u}\; m = \; 2^{k}.$$

Pour évaluer une fonction élémentaire $f(x)$ pour tout x, il est donc nécessaire de trouver une transformation permettant de déduire $f(x)$ à partir de $g(x^*)$ avec :

- x^* est appelé l' " *argument réduit* " *et* est déduit de x *;*
- x^* appartient au domaine de convergence de l'algorithme utilisé pour calculer g.

En pratique, il y a deux types de réduction d'argument :

3.3.1. Réduction additive

Dans ce cas, le calcul de la fonction pour n'importe quel x se trouvant à l'extérieur de l'intervalle réduit prédéterminé I de taille C avec $I = \; [-C/2, C/2]$ revient à transformer x à :

$$x \; = x^* + k \times C$$

où x^* est l'argument réduit appartenant à I, k est un entier *avec* $\mathrm{k} = \lceil \mathrm{x/C} \rfloor$ (partie entière de *x/C*), et C est une constante dépendant de la fonction à calculer et qui est un multiple de $\pi/4$ pour les fonctions trigonométriques ou de $ln(2)$ pour la fonction exponentielle.

En général, la réduction d'argument additive cause plus de problèmes pour les fonctions trigonométriques que pour la fonction exponentielle, car en pratique, on ne rencontre pas l'exponentielle de très grands nombres qui provoquent des dépassements de capacité.

3.3.2. Réduction multiplicative

Dans ce cas, $x = x^* \times C^k,\ où\ x^*$ est égal à x/C^k avec k un entier et C une constante. x^* appartient à l'intervalle I de taille $1/C$. On a alors $I = [0, \frac{1}{C}]\ et\ k = \lceil x/C \rfloor$.

La valeur de C dépend de la fonction qu'on cherche à calculer.

En pratique, la réduction multiplicative ne pose pas de problème vue que la constante C est choisie comme étant une puissance entière de la base du système. Alors la multiplication ou la division par C^k se traduit par un simple décalage.

On remarque qu'après l'étape de réduction d'argument, on obtient un intervalle $[a, b]$ dans lequel on préconise l'approximation de la fonction considéré $f(x)$. Comme on l'a noté précédemment, la largeur de cet intervalle a un rôle important dans la complexité de l'évaluation du polynôme $P_n(x)$. Cette complexité est d'autant plus importante que la largeur de l'intervalle est grande. Un intérêt particulier est accordé à l'optimisation des intervalles, des différentes fonctions considérées dans ce travail. Cette optimisation réduit la disparité entre ces intervalles, afin d'obtenir un même niveau de complexité pour toute les fonctions. Ce qui se traduira par un même niveau de consommation des ressources matérielles et un même niveau de performance. Ceci est plus qu'indiqué, dans la mesure où on désire réaliser toutes les fonctions avec la même architecture qui sera reconfigurable. Un polynôme de même degré pour toutes les fonctions permet d'avoir le même nombre d'opérations d'additions et de multiplications pour toutes les fonctions. En plus, il faut avoir un même niveau de consommation de mémoire, pour le stockage des coefficients des polynômes, pour atteindre la précision de 1ulp qui sera dans notre cas égale à 2^{-52}.

Dans ce travail, on n'est pas concerné par l'implémentation de la partie réduction d'argument dans notre architecture et on supposera que cette étape est faite, néanmoins une optimisation des intervalles de chacune des fonctions est nécessaire pour pouvoir répondre à l'exigence de d'homogénéisation. D'où un intérêt particulier est accordé aux différentes transformations mathématiques qui donnent ces intervalles d'approximation.

3.4. Réduction d'argument des Fonctions considérées.

Dans cette section nous allons présenter les transformations mathématiques qui nous permettent de réduire les arguments des fonctions élémentaires considérées dans ce travail à savoir l'exponentielle (e^x), le logarithme $ln(x)$, l'inverse $(1/x)$, la racine carrée $\sqrt{x}$, le sinus $sin(x)$ et le cosinus $cos(x)$, en considérant x un nombre normalisé en double précision de la norme IEEE-754 (définie dans le chapitre 1).

3.4.1. L'inverse 1/x

Le calcul de la fonction inverse, définie pour tout x différent de zéro, consiste en inversant sa mantisse ainsi que le signe de son exposant.

$$1/x = 1/(1.f \times 2^e) = 1/0.1f \times 2^{-(e+1)}$$

La réduction d'argument se traduit par le décalage d'un bit à droite de la mantisse et l'ajout d'un 1 à l'exposant.

L'intervalle réduit est alors [0.5, 1[,

$$x^* = 0.1f \times 2^{e+1} \quad et \quad 1/x^* = 1/(0.1f \times 2^{(e+1)}) = 1/0.1f \times 2^{-(e+1)}.$$

On calcule la fonction $1/0.1f$ par approximation minimax sur l'intervalle [0.5, 1[. Puis on reconstruit le résultat final en multipliant par $(-1)^s \times 2^{-e-1}$, où le calcul de $(-e-1)$ n'est autre que le complément à 2 de $(e+1)$.

3.4.2. La racine carrée

La fonction racine carrée est définie pour tout nombre positif appartenant au domaine $[0, X_{max}]$ où X_{max} est le plus grand nombre représentable en double précision [65].

La réduction d'argument pour la fonction racine carrée d'un nombre flottant se fait comme suit : $$\sqrt{x} = \sqrt{1.f \times 2^e} = \sqrt{1.f} \times \sqrt{2^e}$$

Si e est pair alors $\sqrt{x} = \sqrt{0.01f \times 2^{e+2}} = \sqrt{0.01f} \times 2^{(e+2)/2}$ et l'intervalle sera [0.25, 1[.

Si e est impair alors $\sqrt{x} = \sqrt{0.1f \times 2^{e+1}} = \sqrt{0.1f} \times 2^{(e+2)/2}$ et l'intervalle sera [0.5, 1[.

Le calcul de la racine carrée de n'importe quel nombre revient à calculer la racine carrée de sa mantisse, décalée d'un ou de deux bits à droite, puis approximer par minimax sur l'intervalle [0.5, 1[.

On doit d'abord tester l'exposant e : s'il est pair nous prenons $x^* = 0.01f$ et $e' = (e+2)/2$, sinon on prend $x^* = 0.1f$ et $e' = (e+1)/2$

3.4.3. L'exponentielle

Le calcul de la fonction exponentielle d'un nombre (x) en virgule flottante se fait par une première limitation du domaine de définition de la fonction par le fait que le résultat de calcul de cette fonction ne peut pas être supérieur au plus grand nombre représentable en double précision ni inférieur au plus petit nombre représentable en double précision de la norme IEEE-754.

Les arguments qui donnent ces limites sont :

$$A = ln\,(2^{-1023}) = -708.3964185$$
$$B = ln\,(2^{1023}) = 709.0895657$$

Ces deux nombres exprimés en binaire par Maple donnent :

$A = -708 = -1.011000100 \times 2^9$ et $B = 709 = 1.11000101 \times 2^9$

L'exponentielle d'un nombre plus grand que B et un dépassement de capacité alors que celui d'un nombre plus petit que A est arrondi à zéro.

Donc avant tout calcul, on doit tester la donnée d'entrée, qui est en double précision virgule flottante, celle-ci doit vérifier les conditions du calcul de l'exponentielle à savoir être supérieure à A et inférieure à B. Si la valeur d'entrée est à l'extérieur du domaine autorisé, le calcul de la fonction ne se fait pas. Pour cela, on teste d'abord l'exposant s'il est supérieur à 9, dans ce cas, on ne fait pas de calcul (si la valeur de x est négatif alors c'est un arrondi vers zéro sinon le résultat est un dépassement de capacité)

Dans le cas contraire c.-à-d. si le nombre x est à l'intérieur du domaine de calcul permis, on doit dé-normaliser le nombre de façon à avoir une partie entière et une partie fractionnaire.

Pour réduire davantage ce domaine à un intervalle plus petit borné par $[-C/2, C/2]$, on utilise la réduction d'argument additive en prenant $C = ln(2)$, et l'intervalle sera alors $\left[-\frac{ln2}{2}, \frac{ln2}{2}\right]$.

Le calcul se fait d'abord par la dé-normalisation du nombre d'entrée qui se traduit par des décalages à gauche ne dépassant pas 10 bits parce que les nombres supérieurs à $e^{(2^{10})}$ sont non représentables en double précision [66].

Pour calculer l'exponentielle du nombre flottant x on fait la transformation $x = x^* + k \times ln2$ et on utilise les identités suivantes :

$$\boldsymbol{e^{(a+b)} = e^a \times e^b}$$

et $\boldsymbol{e^x = \frac{2^x}{ln2}}$.

La première étape est de calculer l'argument réduit x^* tel que :

$$x^* = x - k \times ln2 \qquad avec\ x^* \in [-\tfrac{ln2}{2}, \tfrac{ln2}{2}].$$

Le calcul de l'argument réduit nécessite d'abord de calculer l'entier k qui se fait comme suit :

$k = \lceil x/ln2 \rfloor$ où $\lceil\ \rfloor$ indique la partie entière la plus proche

Comme la multiplication exige moins de hardware que la division, on multiplie par la constante 1/ln2 qui sera stockée en mémoire.

Le calcul de l'exponentielle devient alors :

$$e^{x} = e^{x^{*}+k\times ln(2)} = e^{x^{*}} \times e^{k\times ln(2)} = e^{x^{*}} \times 2^{k}.$$

L'évaluation de la fonction exponentielle est illustrée sur la figure 3.2

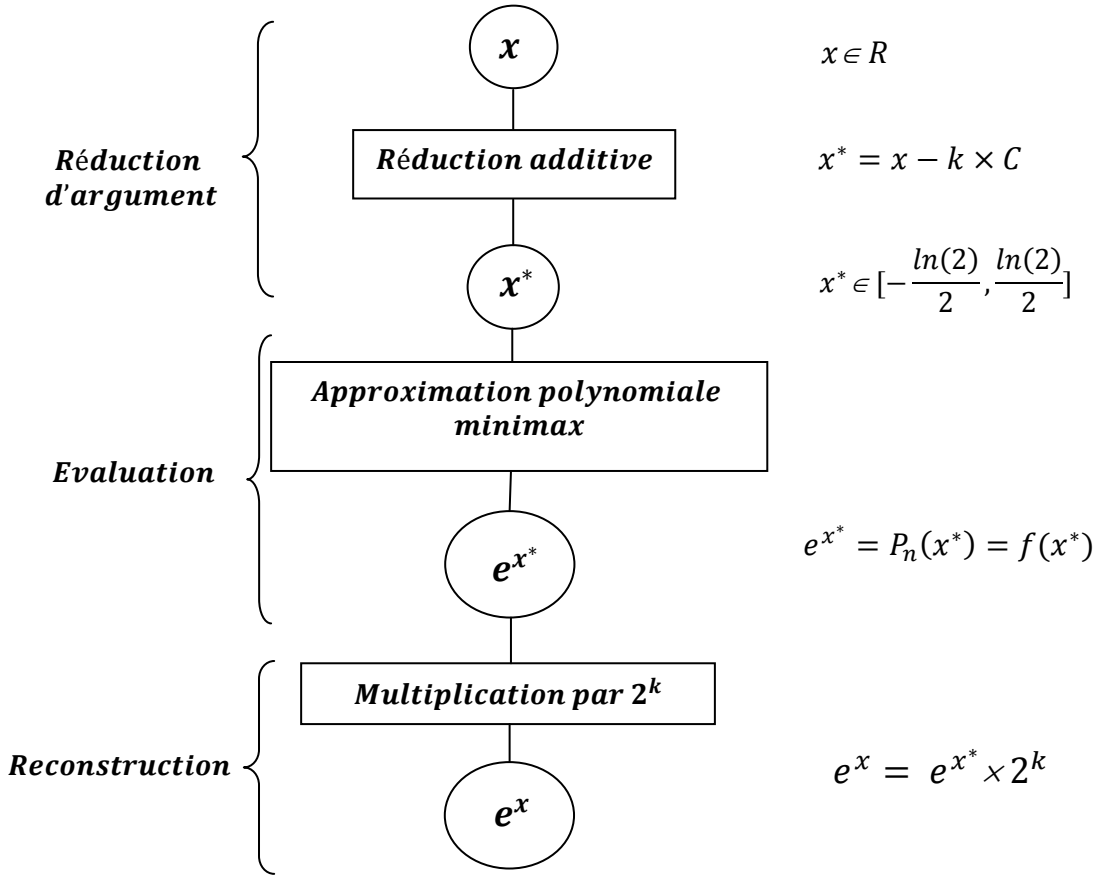


Figure 3.2 : Calcul de la fonction exponentielle avec réduction d'argument

3.4.4. Le logarithme

Le logarithme est défini seulement pour les valeurs de x positif, alors que pour la réduction d'argument, on utilise la propriété additive du logarithme [67].

$$Ln(1.f\times 2^{e}) = ln(0.1f) + ln(2^{e+1}) = ln(0.1f) + (e+1)\times ln(2)$$

Le calcul du logarithme d'un nombre en virgule flottante sera alors la somme du logarithme de la mantisse décalée d'un bit à droite et de l'exposant augmenté de 1 que multiplie par $ln(2)$. L'intervalle de calcul sera alors [0.5, 1[.

3.4.5. Sinus/Cosinus

Ces fonctions sont périodiques et ont des domaines de définition infinis dont le processus se répète tous les 2π. Le résultat de calcul de ces fonctions doit forcément appartenir à l'intervalle $[-1, 1]$. Si le résultat est à l'extérieur de cet intervalle, l'erreur n'est pas vérifiée et les résultats sont complètement faux.

Le calcul de la réduction d'argument se fait sur l'intervalle $[-\pi/4, \pi/4]$ avec une réduction additive et $C = \pi/4$. La transformation utilisée pour exprimer x en fonction de l'argument réduit x^* sera :

$$x = x^* + k \times \pi/4$$

où k est l'entier le plus proche de $x \times (4/\pi)$ et x^* un nombre réel appartenant à $[-\pi/4, \pi/4]$.

$$k = \lceil x \times \pi/4 \rfloor \qquad \text{et} \qquad x^* = x - k \times \pi/4$$

Après la réduction d'argument, calculer $sin(x)$ respectivement $cos(x)$ est équivalent à calculer $sin(x^*)$ ou $cos(x^*)$. L'évaluation de ces dernières est réalisée par une approximation minimax sur l'intervalle [-π/4, π/4].

Selon la valeur de k (N=k modulo 4), le tableau 3.1 donne les résultats de calcul de $sin(x)$ $cos(x)$ fonction des valeurs de $sin(x^*)$ et $cos(x^*)$.

Tableau.3.1 : Calcul des fonctions $sin(x)$ et $cos(x)$

N	$\boldsymbol{sin(x)}$	$\boldsymbol{cos(x)}$
0	$sin(x^*)$	$cos(x^*)$.
1	$cos(x^*)$.	$-sin(x^*)$
2	$-sin(x^*)$	$-cos(x^*)$.
3	$-cos(x^*)$.	$sin(x^*)$

Dans notre travail, on suppose la réduction d'argument est faite pour les fonctions considérées et on se ramène donc à l'approximation minimax des fonctions élémentaires sur les domaines réduits calculés lors de cette section.

3.5. Méthode implémentée

Notre méthode de calcul des fonctions élémentaires est basée sur l'approximation minimax par segments. Celle-ci, consiste à partitionner le domaine de calcul de n'importe quelle fonction en 2^m segments, puis approximer la fonction sur chacun des segments.

Dans cette implémentation hardware, les m bits les plus significatifs de X (x_1) sont utilisés pour adresser la mémoire contenant les coefficients des polynômes. Les (n-m) bits restants de X (x_2) spécifient le point où l'approximation est effectuée.

Le schéma de calcul de cette méthode, pour un polynôme de degré n, est illustré sur la figure 3.3. Cette méthode conduit certes à une diminution de la complexité du hardware ainsi que du temps de calcul par le fait qu'à chaque fois que l'on augmente le nombre d'intervalles, le degré des polynômes P_i diminue. Cependant, cette augmentation dans le nombre des polynômes ne se fait pas sans inconvénient. Celui-ci se matérialise par l'augmentation de la taille totale des tables nécessaire au stockage des coefficients de l'ensemble des polynômes.

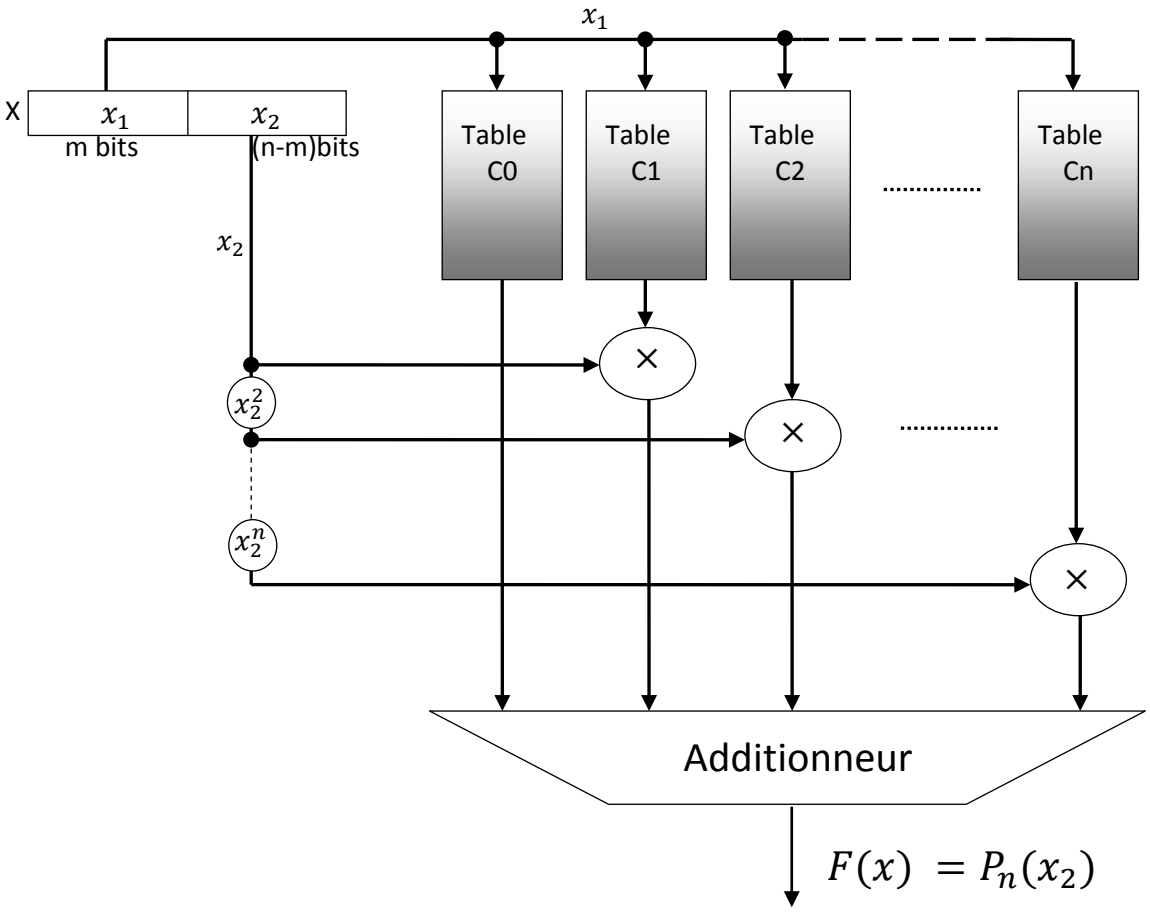


Figure 3.3 : Implémentation matérielle d'une approximation polynômiale par segments

Les tables des coefficients sont en fait adressées par les m bits de x_1. La détermination de ce paramètre est plus que nécessaire avant toute implémentation matérielle.

Comme notre méthode est destinée à être implémentée sur un circuit FPGA dont les ressources en mémoire sont limitées, pour cela on a ciblé une des familles qui est riche en termes de ressources et en particulier riche en mémoire BRAM pour l'implémentation matérielle de la méthode proposée. Il en ressort que la détermination de la valeur de m, qui est

la taille de x_1, est cruciale, dans la mesure où cette valeur représente la taille de l'adresse des mémoires pour le stockage des coefficients.

La première contrainte est de ne pas dépasser la mémoire disponible sur le plus grand circuit FPGA disponible. La deuxième contrainte est la précision des calculs qui influe sur l'ordre du polynôme d'approximation qui ne doit pas excéder 1 ulp, qui est égale à 2^{-52}.

Ce compromis a été résolu grâce à un programme écrit en C pour estimer les valeurs de m et de la taille totale des tables des coefficients pour différents degrés de polynômes.

Ce programme est utilisé juste pour une évaluation grossière de la taille de mémoire exigé par la méthode pour atteindre la précision de 1ulp. Pour ce faire, nous avons alors utilisé dans ce programme les expressions analytiques de l'erreur du polynôme de Taylor. On note par l'occasion que l'erreur du polynôme Minmax est majorée par celle du polynôme de Taylor. Pour cela, une première exécution de ce programme a été faite pour des polynômes P_i de degré 2 et une précision de calcul de 2^{-52}. La valeur de m estimée est de 18, ce qui correspond à une taille totale des tables d'environ de 2^{18}×(55+40+22) = 306708480 bits. Cette taille mémoire est bien supérieure aux ressources disponibles dans le circuit sélectionné. Ce qui nous a obligé à augmenter l'ordre du polynôme à 3 et une nouvelle exécution du programme a résulté en une valeur de m =13 et une taille totale avoisinant 2^{13}×(57+45+32+19) =1253376 bits qui est une valeur implémentable sur le circuit FPGA sélectionné. Ceci nous a permis de fixer l'ordre du polynôme à 3, ce qui signifie que l'évaluation des fonctions considérées se fera par un polynôme de la forme : $P_3(x) = C_0 + C_1 \times x + C_2 \times x^2 + C_3 \times x^3$.

Ce polynôme peut être évalué par le schéma de calcul présenté dans la figure 3.3, néanmoins ce schéma nécessite un nombre d'opération important dont la parallélisation ou l'optimisation est difficile. Ces opérations sont, le calcul de trois multiplications, d'une élévation au carré et d'une autre élévation au cube plus une addition de quatre opérandes. Pour augmenter les performances en termes de rapidité et de précision de calcul, le polynôme peut être évalué selon le schéma de Horner qui permet une réécriture différente du polynôme de façon à réduire le nombre total des multiplications comme suit :

$$P_3(x) = ((C_3 \times x + C_2) \times x + C_1) \times x + C_0.$$

3.5.1. Schéma de HORNER

L'évaluation des fonctions élémentaires se fait souvent par des polynômes qui ne requièrent que deux opérations arithmétiques : l'addition et la multiplication.

Ces polynômes s'écrivent comme suit *:*

$$P_n(x) = C_0 + C_1 x + C_2 x^2 + \ldots\ldots + C_n x^n$$

L'évaluation de ce type de polynômes exige n additions et $n\times(n+1)/2$ multiplications. Cette méthode d'évaluation polynomiale est très lente à cause du calcul des puissances de x qui est relativement coûteux en temps. De plus la précision de calcul du polynôme est faible et cela est dû aux erreurs d'arrondis commises à chaque opération de multiplication.

Pour augmenter les performances en termes de rapidité et de précision de calcul, le schéma de Horner permet une réécriture différente du polynôme de degré n de façon à réduire le nombre total de multiplications qui est la suivante :

$P_n(x) = C_0 + C_1x + C_2x^2 + \ldots + C_{n-1}\,x^{n-1} + C_n\,x^n = C_0 + x\times(C_1 + x\times(C_2 + \ldots + x\times(C_{n-1} + x\times(C_n)) \ldots))$.

Avec l'écriture du polynôme sous la forme Horner, seulement n multiplications sont nécessaires au lieu de $n\times(n+1)/2$. Pour cela, la méthode de Horner est plus rapide et plus précise.

Prenons l'exemple d'un polynôme de degré 3 ;

$$P_3(x) = C_0 + C_1\times x + C_2\times x^2 + C_3\times x^3$$

$$P_3(x) = C_0 + C_1\times x + C_2\times x\times x + C_3\times x\times x\times x$$

Avec l'écriture polynomiale normale, 3 additions et 6 multiplications sont nécessaires.

Le polynôme écrit sous forme de Horner sera :

$$P_3(x) = ((C_3\times x + C_2)\times x + C_1)\times x + C_0$$

Seulement trois multiplications et trois additions sont nécessaires pour l'évaluation du polynôme. Avec la représentation du polynôme sous forme de Horner, le gain en nombre de multiplications est d'autant important que le degré du polynôme soit élevé. De plus la structure de calcul du schéma de Horner est basée sur une seule opération qui est une multiplication suivie d'une addition (FMA) *Fused Multiplier Adder* $(C_{i-1} + x \times C_i)$. Celle-ci s'apprête bien pour une implémentation pipeline ou série comme montré sur la figure 3.4.

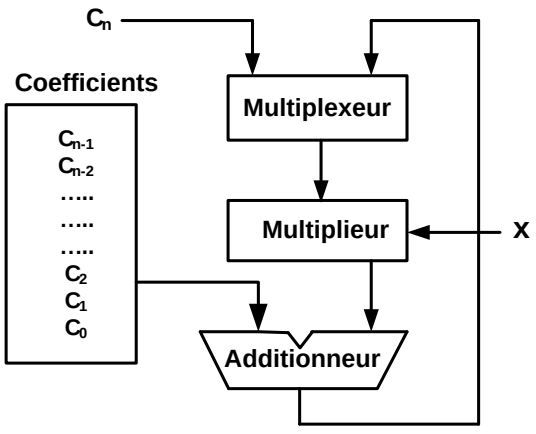


Figure 3.4 : Schéma de Horner pour l'évaluation un polynôme de degré n

3.6. Algorithme de calcul des coefficients

Dans cette section, nous décrivons l'algorithme permettant de déterminer les coefficients C_0, C_1, C_2 et C_3 des polynômes minimax P_i, pour chacun des sous intervalles. Ces coefficients sont calculés par l'outil de calcul formel Maple. Ce dernier permet d'obtenir ces polynômes d'approximation avec des coefficients, sans restriction au niveau de la taille de ces coefficients (en bits). Cependant, le stockage de ces coefficients dans une table exige de les représenter avec une taille de mot finie, et la minimale que possible, afin de minimiser la taille des tables. Il faut donc les arrondir et un tel arrondi des coefficients peut affecter sérieusement la précision de l'approximation. C'est de cette étape que provient la plus grande partie de l'erreur mathématique d'approximation. D'où, il est nécessaire, après arrondi, de recalculer l'erreur d'approximation en utilisant la fonction ***infnorm*** de Maple, pour vérifier si la précision requise est atteinte ou pas.

Pour cela, on a d'abord réalisé un programme en C permettant une estimation grossière des tailles minimales des coefficients arrondis C_0*, C_1*, C_2*et C_3* qui n'affectent pas la précision arrêtée dès le début qui est de 1 *ulp*. Les calculs dans ce programme sont basés sur les expressions de l'erreur de l'interpolation de Taylor. Ce choix est dicté par le fait que l'erreur de l'approximation Minimax n'a pas de formule analytique, ce qui rend sa programmation difficile en C, et que celle-ci est majoré par celle de Taylor.

Ce programme donne respectivement les tailles en bits suivantes 57, 45, 32 et 19 des coefficients C_0*, C_1*, C_2*et C_3*. Ceci représente la première étape dans le processus de détermination et d'optimisation des tailles des coefficients. La seconde étape est d'utiliser une procédure écrite en *Maple*, représentée sur la figure 3.5, qui calcule l'erreur totale commise dans l'approximation d'une fonction élémentaire par un polynôme minimax d'ordre 3 dont les coefficients sont arrondis.

Cette procédure, basée sur quatre paramètres, (p, q, r, t) représentant respectivement les LSB des coefficients C_0*, C_1*, C_2* et C_3*, calcule l'erreur totale commise lors de l'approximation d'une fonction élémentaire par un polynôme minimax d'ordre 3.

Le but de cette procédure est l'optimisation des tailles des coefficients pour avoir les tailles minimales des coefficients pour une précision totale ne dépassent pas 1 *ulp*.

Le principe est de lancer un premier calcul avec les dimensions données par le programme en C (qui représente le point de départ de nos calculs) et vérifier l'amplitude de l'erreur. Si cette dernière est toujours inférieure à 2^{-52}, on diminue la taille de C_0 et on refait une autre exécution, jusqu'à ce qu'on trouve la taille minimale de C_0 qui correspond à l'erreur

d'approximation maximale qui ne dépasse pas l'erreur de calcul recommandé. Une fois la taille de C_0 est optimisée, on passe au coefficient C_1 et on refait le même travail pour le reste des coefficients. Il est à noter que le calcul de l'erreur de l'approximation minimax, pour des tailles de coefficients bien déterminées, est réalisé d'une manière exhaustive (c.-à-d. pour toutes les valeurs possibles de x_2) ce qui représente un volume de calcul énorme. Pour mener à bien ce travail, nous avons utilisé 4 machines (PC Pentium 2.80 GHz, 512 Mo de RAM) pour une durée d'environ deux mois.

```
Minmax:=proc(p,q,r,t)
with(numapprox):
Digits := 60;  errmax:=0;  erround:=0;
MaxC3:=0;    MinC3:=0;
MaxC2:=0;    MinC2:=0;
MaxC1:=0;    MinC1:=0;
MaxC0:=0;    MinC0:=0;
s:=evalf(-ln(2)/2);
for i from 0 to 5678 do
pol[i]:= collect(minimax(exp(s+1/8192*i+x),x=0..1/8192,[3,0],1,'err'),x);
if abs(err) > errmax then    errmax:= abs(err);    end if;
C3[i] := round(2^p*(coeff(pol[i],x,3)))/2^p;
C0[i] := round(2^q*(coeff(pol[i],x,0)))/2^q;
C1[i] := round(2^r*(coeff(pol[i],x,1)))/2^r;
C2[i] := round(2^t*(coeff(pol[i],x,2)))/2^t;
erreur:= infnorm(exp(s+1/8192*i+x)-C0[i]-C1[i]*x-C2[i]*x^2-C3[i]*x^3,x=0..1/8192);
if abs(erreur) > erround then     erround := abs(erreur);     end if;
if C3[i] > MaxC3 then              MaxC3:= C3[i];              end if;
if C2[i] > MaxC2 then              MaxC2:= C2[i];              end if;
if C1[i] > MaxC1 then              MaxC1:= C1[i];              end if;
if C0[i] > MaxC0 then              MaxC0:= C0[i];              end if;
if C3[i] < MinC3 then              MinC3:= C3[i];              end if;
if C2[i] < MinC2 then              MinC2:= C2[i];              end if;
if C1[i] < MinC1 then              MinC1:= C1[i];              end if;
if C0[i] < MinC0 then              MinC0:= C0[i];              end if;
od;
print(errmax,erround,MinC0,MinC1,MinC2,MinC3,MaxC0,MaxC1,MaxC2,MaxC3);
end;
```

Figure 3.5 : Programme Maple pour le calcul des tailles des coefficients

La taille d'un coefficient est en fait égale à son LSB plus son MSB. D'où le calcul des valeurs maximales et minimales de ces coefficients est nécessaire. Ce calcul est inclus dans la même procédure de calcul des coefficients. Avec ces max et ces min, on peut déterminer sur combien de bits seront représentées les parties entières de ces coefficients, qui seront comptabilisées dans les tailles globales de ces coefficients.

Dans le tableau 3.2 sont montrées les tailles d'adressage m, les tailles des coefficients et l'erreur commise pour chaque fonction considérée.

Tableau.3.2 : Tailles des coefficients et erreurs max pour les différentes fonctions.

Fonction	m	Taille de C_0	Taille de C_1	Taille de C_2	Taille de C_3	Max erreur $\times 10^{-16}$
e^x	12	1.55	1.43	0.30	0.19	0.27862008090
$ln(x)$	12	1.54	2.42	1.30	2.19	0.31309458513
$sin(x)$	12	1.53	1.42	0.30	0.20	0.55471384924
$cos(x)$	12	1.53	1.42	0.30	0.20	0.51864464727
$1/x$	13	2.53	3.40	3.30	4.20	0.42997801240
$\sqrt{x}$	13	0.53	0.41	0.30	0.19	0.35048255213

Note. Les tailles des coefficients sont exprimées par des nombres, dont la partie entière représente la taille de la partie entière du coefficient alors que la partie fractionnaire représente la taille de la partie fractionnaire du coefficient.

3.7. Architecture

L'évaluation d'une fonction élémentaire nécessite un polynôme minimax d'ordre 3 et fait intervenir trois fois le calcul de l'opération $(a \times x + b)$ qui est exécutée par un seul module FMA (*Fused Multiplier Adder*). L'évaluation d'une fonction peut être réalisée par une structure séquentielle ou pipeline. Afin d'augmenter les performances de notre architecture en termes de débit de calcul et de temps d'exécution, l'architecture pipeline a été adoptée et l'opération $(a \times x + b)$ est exécutée par un (FMA) sans propagation de la retenue. L'idée est de conserver les résultats issus des FMAs en représentation C.S (Carry-Save). La conversion C.S– Complément à deux est effectuée dans le dernier étage par un additionneur rapide.

L'architecture proposée est illustrée sur la figure 3.6.

Elle comprend trois modules FMA. Chacun de ces modules est constitué d'un multiplieur et d'un additionneur.

3.7.1. Le module FMA *(Fused Multiplier Adder)*

Les trois modules FMAs réalisent le calcul de l'opération $(a \times x + b)$. Comme montré sur la figure 3.6 l'opération que réalise le FMA_1 est $(C_3 \times X_2 + C_2)$, celle du FMA_2 est $(Res\_1 \times X_2 + C_1)$, alors que celle du FMA_3 est $(Res\_2 \times X_2 + C_0)$. La différence, dans ces trois opérations, est que l'opérande « a » est en complément à deux pour le FMA_1 et en C.S pour les FMA_2 et FMA_3. L'implémentation du FMA dans un seul bloc a l'avantage

d'effectuer un seul arrondi final au lieu de deux comparés à l'utilisation d'un multiplieur plus un additionneur séparés en plus des avantages de la réduction du matériel et du temps d'exécution par la non propagation de la retenue.

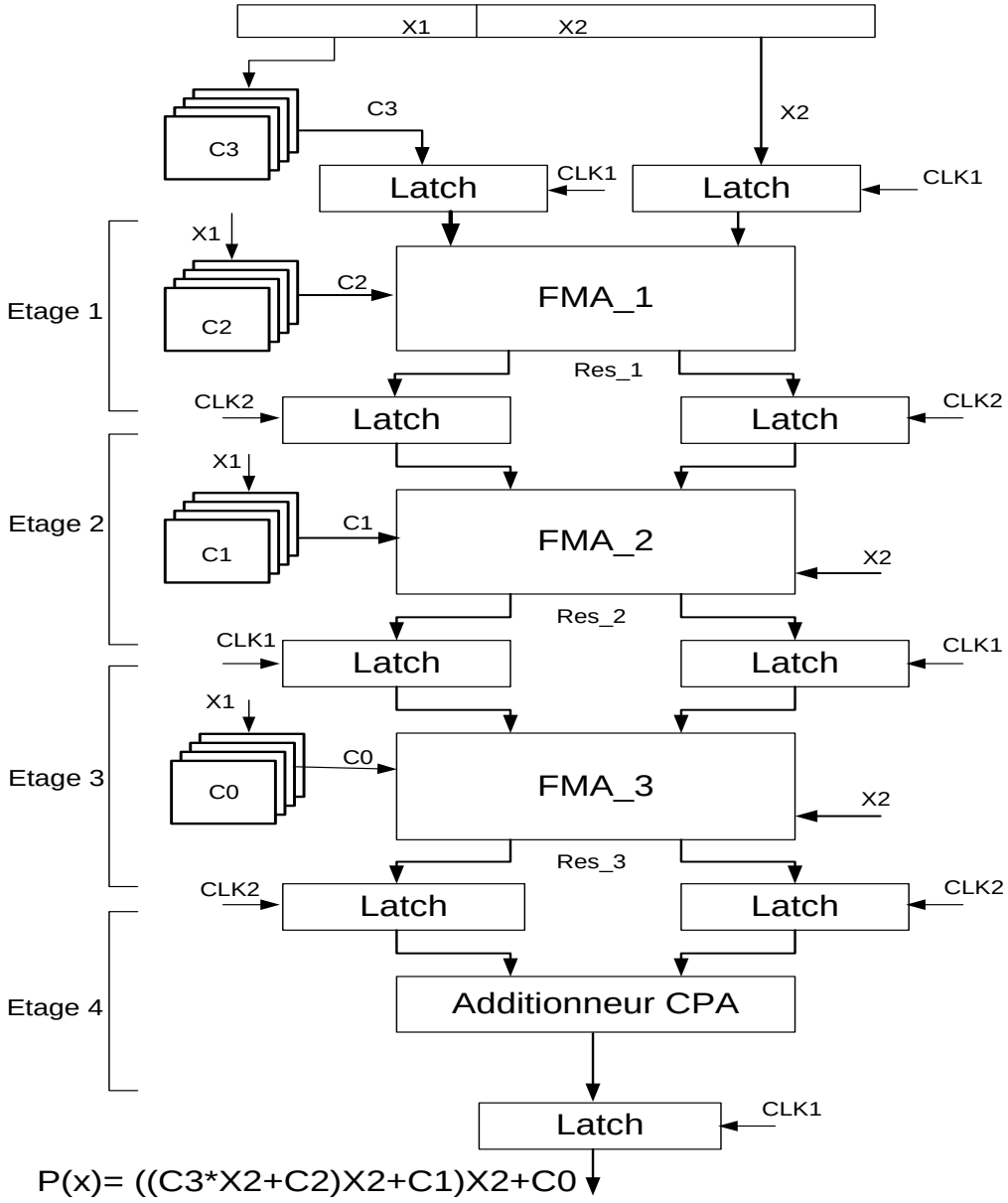


Figure 3.6 : Architecture globale pour l'évaluation des fonctions élémentaires.

En général un multiplieur est constitué de trois parties : la génération des produits partiels, la réduction des produits partiels et enfin l'addition.

Les FMAs de notre architecture ne comportent pas d'addition finale d'où les résultats à leurs sorties sont en représentation C.S. Le terme b est admis dans le bloc réduction des produits partiels au même titre que les produits partiels de $(a \times x)$. Le diagramme bloc d'un FMA est

représenté sur la figure 3.7. Le FMA contient un codeur de Booth modifié [68], un générateur de produits partiels et un module de réduction des produits partiels.

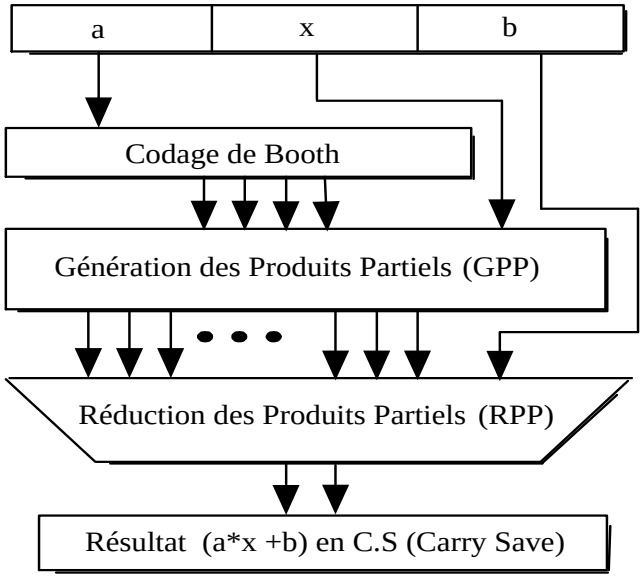


Figure 3.7 : Schéma synoptique du module FMA.

3.7.2. Le codage de Booth

L'idée de Booth est d'augmenter le nombre de zéro dans le multiplicateur en le recodant avec l'ensemble des chiffres $(-1, 0, 1)$. Cette méthode permet de transformer une chaîne de 1 par une écriture avec plus de zéro. Exemple : 00111110 est codé par $0100000\bar{1}0$. L'utilisation de l'algorithme de Booth modifié présente un grand avantage, c'est qu'il permet une réduction d'au moins 50% du nombre des produits partiels à additionner. Ce qui se répercute par une amélioration du délai de réduction des produits partiels, ainsi qu'une réduction substantielle de la surface occupée par le module réduction des produits partiels. Le codage de Booth modifié se fait sur des groupements de trois bits. Chacun de ces groupements est recodé dans le système {-2, -1, 0, 1, 2}, qui est appelé SD4 (*Sign Digit 4*).
Ce codage est montré sur le tableau 3.3. L'intérêt de ce recodage est que les produits de ces digits avec le multiplicande sont réalisés par de simples inversions et décalages. _
Les opérandes concernés par ce codage sont le coefficient C_3 pour le FMA_1 et les résultats Res_1 et Res_2 pour les FMA_2 et FMA_3.

Tableau.3.3 : Codage de Booth modifié

Table de sélection des produits partiels	
Bits du multiplicateur	Sélection
000	+0
001	+multiplicande
010	+multiplicande
011	+2×multiplicande
100	−2×multiplicande
101	−multiplicande
110	−multiplicande
111	−0

Comme illustré sur la figure 3.8, le circuit codage de Booth du FMA_1 est constitué de 10 cellules. Il est à noter que le nombre de cellules dépend de la taille du multiplicateur en bits. Le circuit logique de la j^{ième} cellule qui fait le codage de $C_{3,}$ qui est en complément à 2, à la représentation SD4 (*Sign Digit 4*) est montré sur la figure 3.9.a. Cette cellule est mappée sur deux LUTs comme le montre la figure 3.9.b avec un délai équivalant à celui d'un slice.

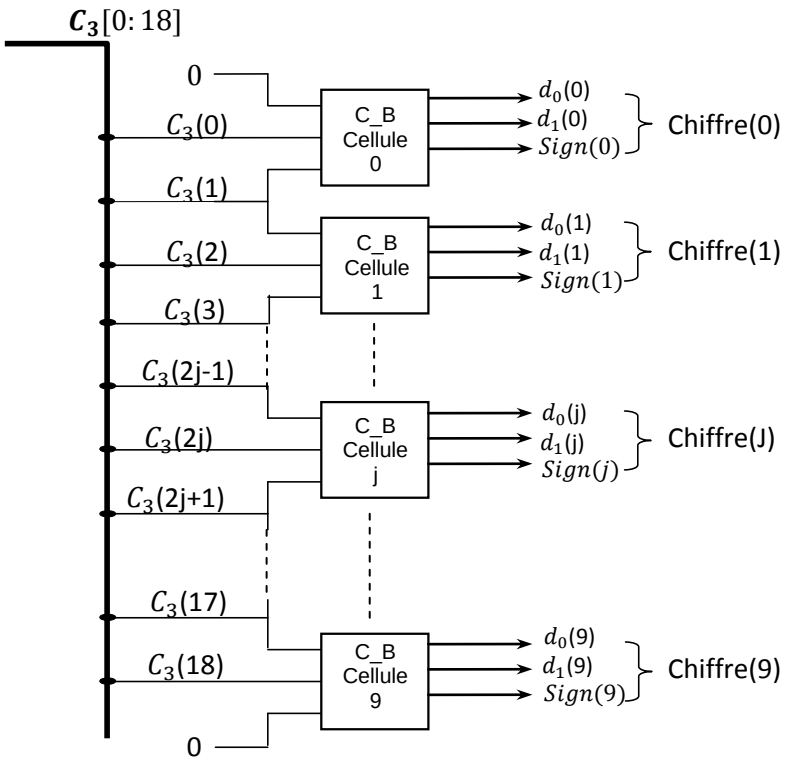


Figure 3.8 : Circuit Codage de Booth du FMA_1.

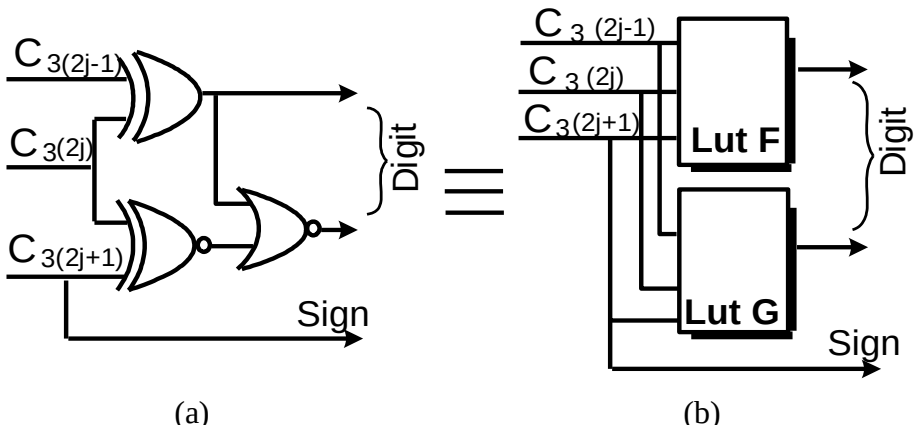


Figure 3.9 : Cellule de Codage C-à-2_SD4

Les deux autres opérandes à coder sont (Res_1 et Res_2) qui sont les résultats de calcul des FMA_1 et FMA_2. Ces deux opérandes sont en C.S et seront recodés dans la représentation SD4 sans propagation de la retenue. Le principe du codage est le même, néanmoins chacune des cellules de codage nécessite 6 bits au lieu de trois pour les opérandes en complément à 2. Le circuit logique de la j^{iéme} cellule est illustré sur la figure 3.10.

Pour éviter la propagation de la retenue, nous calculons tous les Cout en parallèle en exploitant au mieux les possibilités offertes par le circuit Virtex-2 dans l'implémentation d'une fonction de génération à six entrées sur deux *lices* consécutives, cela est représenté sur la figure 3.11. Ainsi le délai total d'un codeur CS-SD4 est équivalent à celui de trois slices.

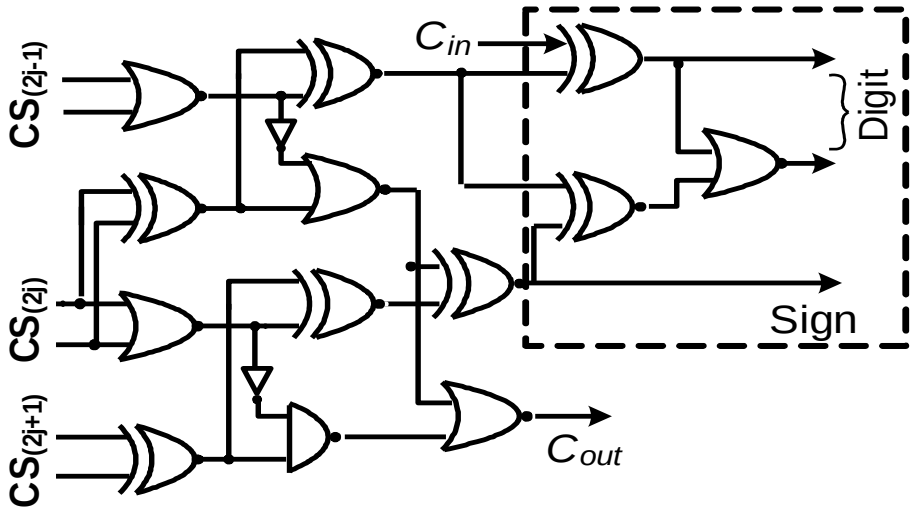


Figure 3.10 : Cellule du codeur CS_SD4

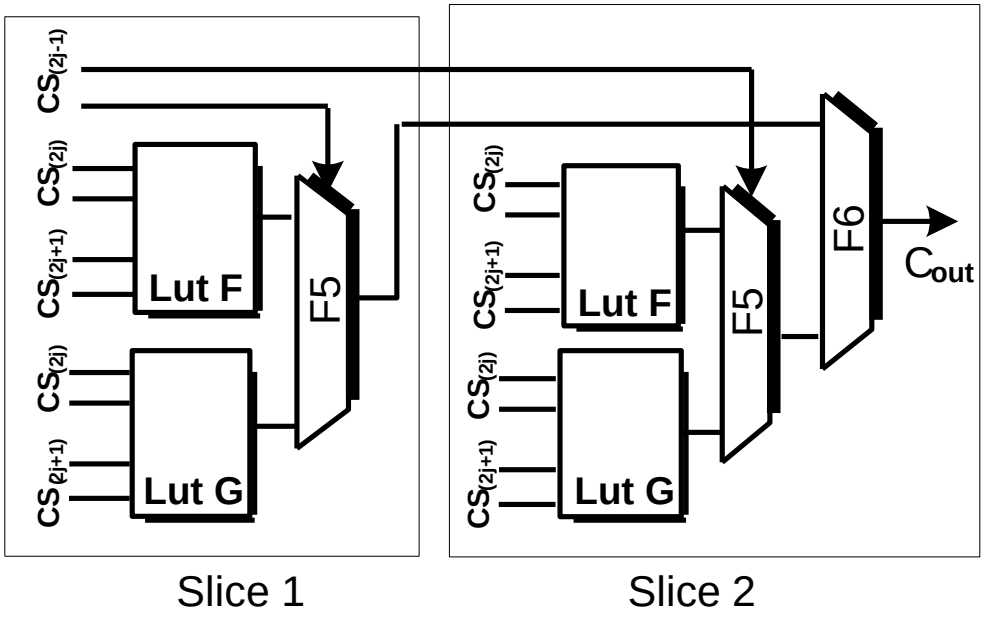


Figure 3.11 : Implémentation du chemin critique du codeur CS-SD4

3.7.3. Génération des produits partiels (GPP)

Chacun des produits partiels est généré par la multiplication d'un chiffre du code Booth (SD4) du multiplicande par le multiplicateur comme montré sur la figure 3.12. Il est à noter que la génération de tous les PPs se fait en parallèle, le schéma logique et l'implémentation sur FPGA d'une cellule «génération des produits partiels PPs » sont illustrés sur la figure 3.13. Cette cellule est implémentée sur un slice et son délai équivaut à la traversée d'un slice.

3.7.4. Réduction des produits partiels (RPP)

Après la génération des PPs, on obtient un *Bitarray* d'une hauteur de $(n+2)/2$, où n est le nombre de bits du multiplicateur. L'opération de réduction consiste alors à réduire ce *Bitarray* en un seul nombre en CS, puis faire une addition pour avoir le résultat final comme illustré sur la figure 3.14. Plusieurs techniques existent pour la réduction de ces PPs [69], [70], [71], celles-ci se concurrent la réduction du *Bitarray* en un minimum de délai. D'une manière générale la réduction se fait sous forme arborescente en plusieurs étapes (étages). Le délai de la réduction dépend alors du nombre d'étapes (étages). Ces techniques utilisent le *Full adder* qui est un compresseur 3:2. Dans notre architecture, nous utilisons une méthode à base des compresseurs 4:2 [39]. Le compresseur 4:2, représenté sur

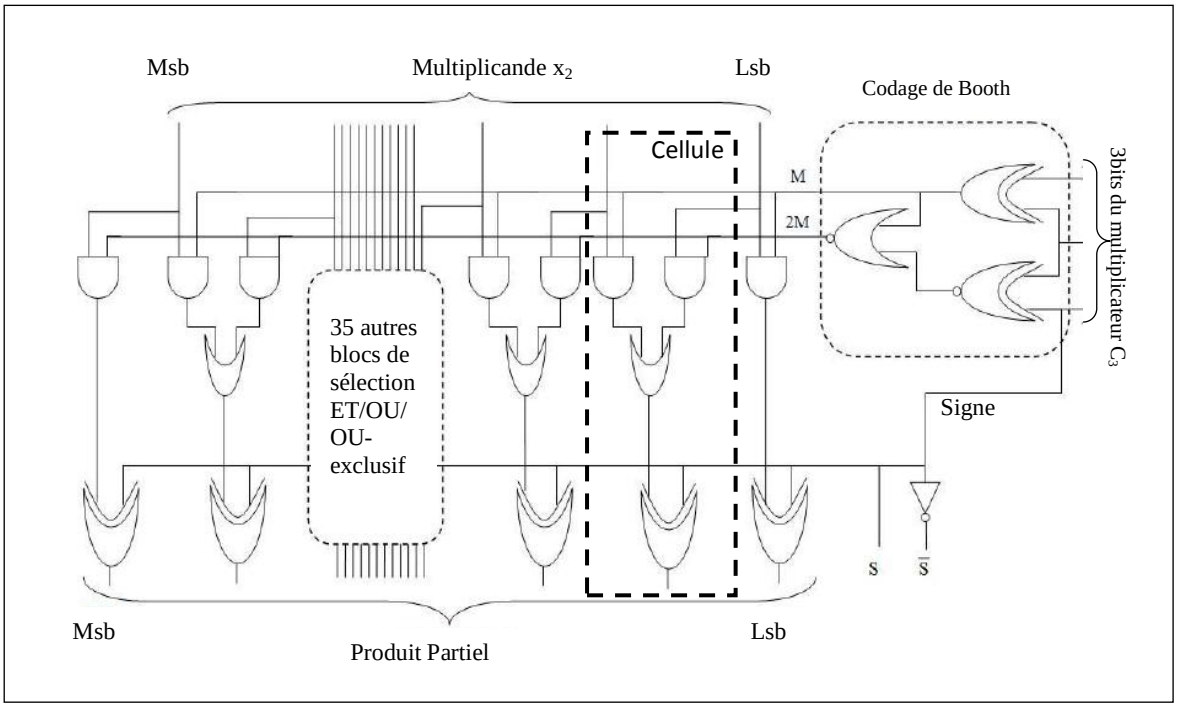


Figure 3.12 : Le circuit logique de la génération d'un PP du FMA_1

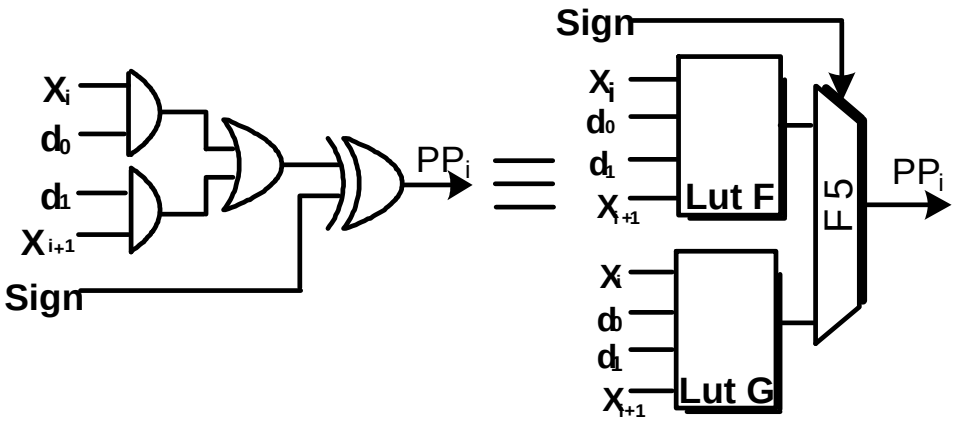


Figure 3.13 : Une Cellule du circuit de génération des PPs

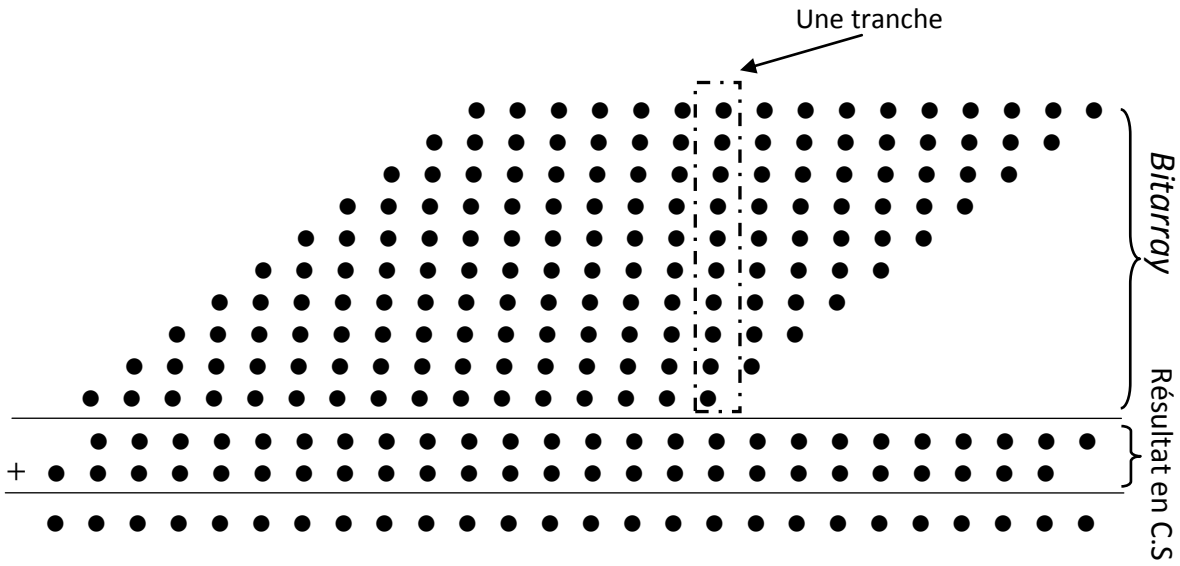


Figure 3.14 : Réduction des PPs

la figure 3.15, est un additionneur à quatre entrées et deux sorties, mais en additionnant quatre bits à '1' on peut avoir $(100)_2$ comme résultat, qui est sur trois bits. Le compresseur 4:2 est en fait constitué de deux *full adder* en cascades. Bien que le compresseur 4:2 présente un taux de compression plus important que celui du *full adder,* son utilisation dans un arbre de réduction permet certes de diminuer le nombre d'étages, cependant le delai totale de la réduction reste tributaire de celui du compresseur 4:2. Pour une implémentation VLSI le délai du compresseur 4:2 est bien plus important que celui du *full adder* et son utilisation au détriment du *full adder* reste à vérifier.

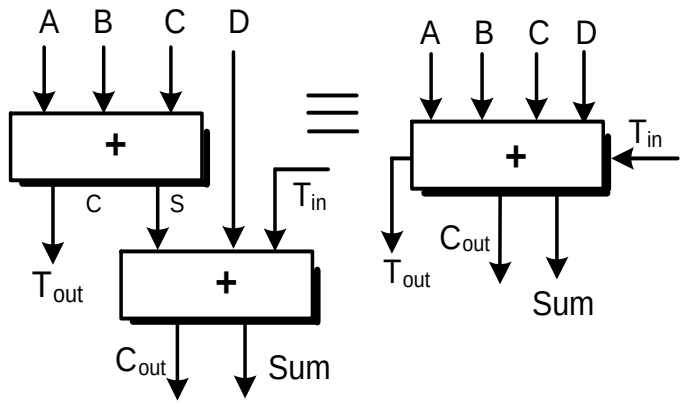
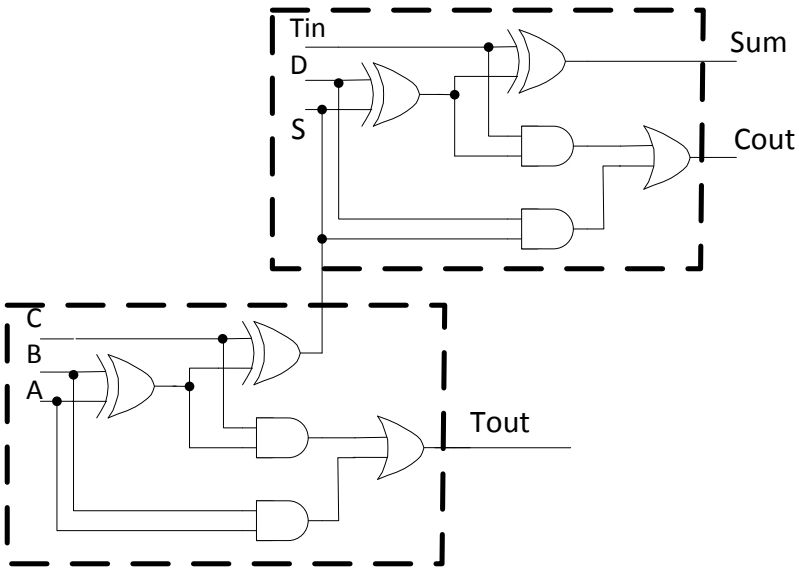


Figure 3.15 : Le Compresseur 4:2.



Sum = Tin⊕D⊕S = Tin⊕D⊕A⊕B⊕C
Cout = (D⊕S).Tin+D.S = (D⊕A⊕B⊕C).Tin+D.(A⊕B⊕C)
Tout = (A⊕B).C+A.B

Figure 3.16 : Circuit logique d'un Compresseur 4:2

Comme notre architecture est dédiée à une implémentation sur circuit FPGA dont l'unité logique de base est le *slice* et non le transistor, ce qui peut être un atout pour l'implémentation FPGA du compresseur 4:2 par rapport au *full adder*. Comme illustré dans la figure 3.16, les sorties Sum, Cout sont des fonctions à 5 entrées alors que Tout est une fonction à 3 entrées. Chacune de ces fonctions peut être implémentée facilement dans un slice (figure 3.17), d'où le délai d'une compression 4:2 n'est autre que celui de la traversée d'un slice. Cette particularité offerte par les circuits FPGA est avantageusement utilisée dans notre architecture dans la mesure où le compresseur 4:2 a un taux de compression plus important que celui du *full adder* alors qu'ils ont le même délai.

Le principe de l'utilisation du compresseur dans l'arbre de réduction des PPs est en premier lieu d'organiser tous les bits du même poids ensemble (dans une tranche) comme illustré dans la figure 3.14, puis faire des réductions jusqu'à avoir seulement deux bits, un bit *Sum* et un bit *Carry*. Ce principe est montré sur la figure 3.18. Il est à noter que le délai d'un étage est le même que celui du compresseur 4:2.

L'utilisation du compresseur dans la réduction des PPs exige un routage inter-traches. Un compresseur se trouvant dans la tranche i à l'étage n génére deux retenues du même poids C_{out1} et C_{out2}, celles-ci sont connectées aux entrées du compresseur se trouvant au même

étage de la $(i+1)^{ième}$ tranche. Dans la figure 3.18 est montrée le principe de réduction d'une tranche de 10 bits (10 PPs), les sortie $C_{out}(i)$ représentent les retenues sortantes de la tranche i vers la tranche *i+1*, alors que les entrées $C_{out}(i-1)$ sont en fait les retenues sortantes de la tranche $(i-1)$. Le délai logique d'une tranche est égal au délai d'un slice fois le nombre d'étages que comporte cette tranche.

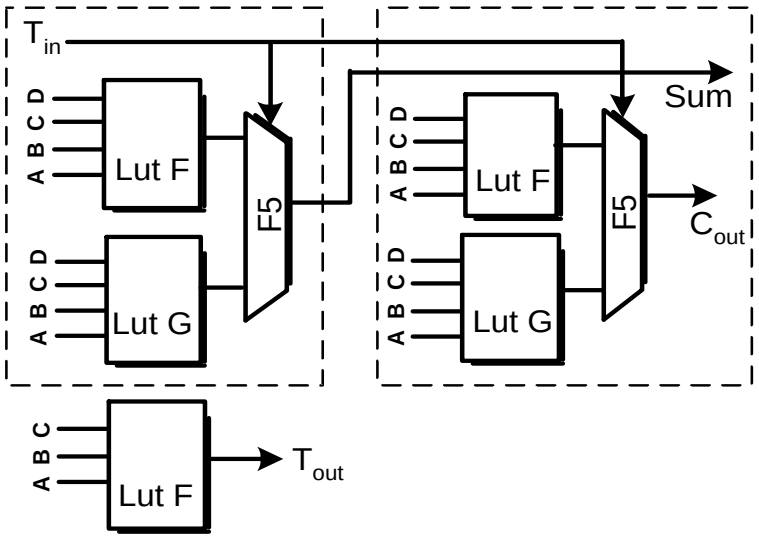


Figure 3.17 : Implémentation du compresseur 4:2

3.8. Calcul d'erreurs

Une bonne évaluation des fonctions élémentaires se ramène pour beaucoup à une bonne gestion des erreurs de calcul dans les algorithmes. Plus précisément, il faudrait pouvoir borner l'erreur commise, qui est définie comme la différence entre le résultat calculé et le résultat idéal. Cette différence peut être absolue, ou relative au résultat idéal. La notion de «résultat idéal» doit également être formalisée : si pour la sortie ce sera clairement la valeur mathématique de la fonction, pour les valeurs intermédiaires du calcul ce sera moins évident. Les erreurs qui font que le résultat calculé s'écarte du résultat idéal sont essentiellement de deux types :

- Erreurs algorithmiques : C'est l'erreur due à l'algorithme utilisé, dans notre cas c'est l'erreur qu'on commet lors de l'approximation d'une fonction par un polynôme.
- Erreurs matérielles : C'est les erreurs dues à l'utilisation des registres à tailles finies pour représenter les valeurs intermédiaires et les résultats. Dans notre cas

c'est les erreurs d'arrondi, apparaissant lors du calcul du polynôme dans chaque opération non exacte.

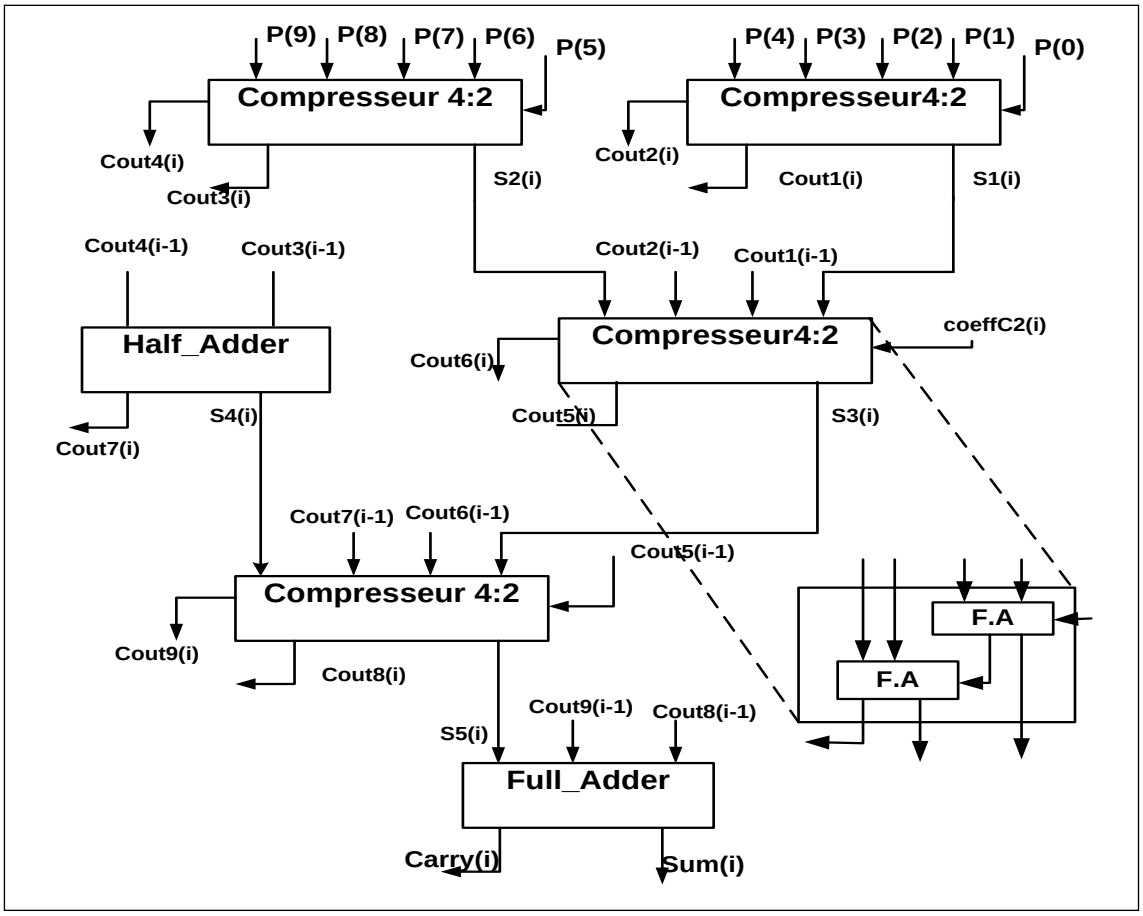


Figure 3. 18 : Architecture du $i^{ième}$ tranche du module réduction des PPs

Nous définissons alors l'erreur totale commise sur le résultat final, de calcul d'une fonction élémentaire, comme étant la somme de l'erreur sur les résultats intermédiaires avant l'arrondi final $\boldsymbol{E_{Inter}}$ et l'erreur due à l'arrondi final $\boldsymbol{E_{Arr\_fin}}$.

$$\boldsymbol{E_{Tot} = E_{Inter} + E_{Arr\_fin} < 2^{-r}}$$

Où r dépend du format de représentation des données d'entrées/sorties de la fonction à calculer ainsi que de la précision désirée.

Pour notre application, les opérandes sont représentées dans la norme IEEE-754 double précision et les calculs se feront avec une précision de $1ulp$ $(r = 52)$.

L'erreur sur les résultats intermédiaires $\boldsymbol{E_{Inter}}$ est la composante de deux erreurs $\boldsymbol{E_{Approx}}$ (l'erreur due à l'approximation polynomiale) et $\boldsymbol{E_{Cal\_inter}}$ (l'erreur due aux tailles finies des résultats intermédiaires).

A la fin de l'évaluation, on effectue un arrondi au plus prés (c'est l'arrondi à la moitié du dernier bit)**.** Ce qui implique :

$$\boldsymbol{E_{Arr\_fin} \leq 2^{-53}}$$

et comme $\boldsymbol{E_{Tot} = E_{Inter} + E_{Arr\_fin} \leq 2^{-52}}$

donc $\boldsymbol{E_{Inter} = E_{Approx} + E_{Cal\_inter} < 2^{-53}}$

Cette erreur est divisée par deux d'où ;

$$\boldsymbol{E_{Approx} < 2^{-54} \quad et \quad E_{Cal\_inter} \leq 2^{-54}}$$

Les programmes Maple (figure 3.5) que nous avons réalisés, pour l'obtention des coefficients, calculent aussi la contribution en erreur due à l'arrondi de ces coefficients. L'erreur $\boldsymbol{E_{Approx}}$ est la composante de deux erreurs : la première est due à l'approximation polynomiale $\boldsymbol{E_{Poly}}$ et la seconde est due à l'arrondi des coefficients $\boldsymbol{E_{Arr\_Coef}}$.

$$\boldsymbol{E_{Approx} = E_{Poly} + E_{Arr\_Coef} < 2^{-54}}$$

Dans le tableau 3.4, sont montrées les résultats des calculs des erreurs $\boldsymbol{E_{Approx}}$ et $\boldsymbol{E_{Poly}}$ pour le pire cas et pour chacune des fonctions considérées.

Il est à noter que ce calcul d'erreurs a été effectué d'une manière exhaustive c.-à-d. pour toutes les valeurs possibles de x de l'intervalle de calcul de chacune des fonctions que nous avons défini dans la section 3.4.

Tableau.3.4 : $\boldsymbol{E_{Approx}}$ et $\boldsymbol{E_{Poly}}$ des fonctions considérées.

Fonction	$\mathbf{E_{Poly}} \times 10^{-17}$	$\mathbf{E_{Approx}} \times 10^{-15}$
e^x	0.0102222534125	0.27862008090
$ln(x)$	0.6935506891277	0.31309458513
$sin(x)$	0.0887740967359	0.55471384924
$cos(x)$	0.1156482302025	0.51864464727
$1/x$	0.3468388359924	0.42997801240
$\sqrt{x}$	0.0541869543923	0.35048255213

L'erreur de calcul intermédiaire $\boldsymbol{E_{Cal\_inter}}$ est l'accumulation des erreurs produites suite aux troncatures des résultats intermédiaires lors du calcul d'une fonction élémentaire.

On note que ces troncatures concernent seulement les résultats issus des trois FMAs (Res_1, Res_2 et Res_3).

$$\boldsymbol{E_{Cal\_inter} \leq 2^{-54}}$$

C'est l'erreur au niveau du résultat final Res_3 c.-à-d. avant l'additionneur CPA. D'où l'erreur maximum permise sur chacune des branches de Res_3 (Carry et Sum) est de 2^{-54}, ce qui veut dire que $E_{Res\_3} = 2^{-54}$.

$$Res\_3 + E_{Res\_3} = x2 \times (Res\_2 + E_{Res\_2}) + C3.$$
$$= x2 \times Res\_2 + C3 + x2 \times E_{Res\_2}$$

D'où $E_{Res\_3} = x2 \times E_{Res\_2}$

De la même manière, on peut remonter et calculer l'erreur de troncature permise au niveau des résultats Res_2 et Res_1 à savoir E_{Res_2} et E_{Res_1}.

$x2$ est représenté sur 52 bits, alors que ses 12 premiers bits sont à zéro.

D'où $\boldsymbol{E_{Res\_2} \leq 2^{-42}}$

Ce qui implique que le résultat intermédiaire Res_2 est tronqué à 2^{-42} (c.-à-d. on ne gardera que 42 bits de sa partie fractionnaire).

De la même façon, on calcule E_{Res_1} à partir de E_{Res_2}.

$$\boldsymbol{E_{Res\_1} \leq 2^{-30}}$$

D'où la partie fractionnaire de $\boldsymbol{E_{Res\_1}}$ ne contiendra que 30 bits.

Ce calcul d'erreur nous a permis de dimensionner les tailles des chemins de données dans notre architecture et de définir les niveaux de troncatures des résultats intermédiaires. Ces opérations de troncature sur les données intermédiaires ont l'avantage de réduire les tailles des chemins des données au minimum possible sans toutefois enfreindre à la précision requise et par voie de conséquence réduire le délai d'exécution ainsi que le hardware nécessaire pour l'implémentation de notre architecture.

3.9. Résultats d'implémentation

L'architecture pipeline présentée dans ce chapitre a été conçue en utilisant l'environnement de conception de Xilinx (ISE 7.1). Tous les blocs ont été décrits en code VHDL. Les tables ont été générées par l'outil CORE GENERATOR. Certaines stratégies ont été utilisées pour optimiser cette architecture et un intérêt particulier a été accordé au compresseur 4:2. Nous avons utilisé la particularité des slices de Virtex-2 (qui sont des fonctions de génération à cinq entrées).

Chacune des sorties du compresseur a été implémentée séparément dans un slice. D'où le délai du compresseur équivaut au délai d'un slice. Ceci réduit sensiblement le délai du compresseur et par conséquent le délai au niveau des blocs de réduction des produits partiels (PPs). Les mêmes optimisations ont touchées les cellules de codage C.à.2-SD4 et CS-SD4. Ainsi le délai pour ces deux cellules est celui de deux slices.

Pour garantir le bon fonctionnement de notre architecture, celle-ci a été simulée en utilisant l'outil ModelSim XE III 6.0a, et synthétisée [72] par l'outil XST (Xilinx Syntesis Tool). Notre architecture a été implémentée dans le circuit XC2V3000 (- 5) fg676.

Les résultats d'implémentation sont donnés dans le tableau 3.5 comme suit :

Tableau 3.5 : Résultats d'implémentation

La superficie totale cumulée :	
Le nombre de Slices	6524 de 14336 (45%)
Le nombre de blocs BRAM	87 de 96 (90%)

Le délai le plus important dans notre architecture est celui de l'étage 3 qui est de (17.372ns). Notre architecture peut opérer avec une fréquence de 57 Mhz et une latence de quatre cycles. Ce qui correspond à un débit de calcul d'environ 57.5 millions d'opérations par second.

3.10. Discussion

De nombreux travaux ont traité de l'implémentation du calcul des fonctions élémentaires sur matériel ; ces travaux se diffèrent par la précision de calcul et le support d'implémentation. Dans cette comparaison, nous considérons les méthodes implémentées sur les circuits FPGA.

Nous avons comparé d'abord notre méthode à l'unité Look-up développée par Mencer et al. [73]. Cette méthode est basée sur le développement de TAYLOR d'ordre 1, utilisé pour approximer la fonction et qui donne moins de précision que le polynôme Minimax que nous utilisons. En plus, la méthode de Mencer concerne seulement la précision de 24 bits alors que celle que nous avons présentée va jusqu'à 53 bits. Ceci est dû à l'exigence importante en mémoire de la méthode de Mencer où les tables ont été mappées sur les CLBs au lieu des BRAMs. La latence de la méthode de Mencer est de quatre pour une

précision de calcul de 24 bits, alors que notre méthode a la même latence pour une précision de calcul de 53 bits.

Nous avons aussi comparé notre méthode à celle proposée par J.A Pineiro et al [74], qui concerne l'implémentation sur circuit FPGA de la fonction puissance dont les cas spécifiques étudiés dans cet article sont X^{-1} et $X^{1/2,}$ qui sont calculées aussi par notre méthode. Ils utilisent un polynôme minimax d'ordre 2 pour calculer les fonctions avec une précision de 23 bits. Leur architecture est basée sur l'optimisation de l'unité élévation au carré d'un nombre et du multiplieur au niveau logique sans tenir compte des ressources internes du support d'implémentation qui est le FPGA telles que le *carry chain* pour l'addition, au lieu d'utiliser un additionneur CLA (Carry Look-Ahead). L'implémentation résultante a exigé un grand nombre de CLBs et un long chemin critique.

Afin de voir l'apport en performance qui est due à l'optimisation de notre opérateur FMA, utilisé pour réaliser l'opération $(a\times b + c)$ dans notre approche, par rapport à un FMA réalisé à base des blocs multiplieurs 18×18 bits et d'un additionneur (*carry chain).* Nous avons alors réalisé une autre architecture avec la deuxième approche où l'on remplace le FMA par un multiplieur (réalisé à base des blocs multiplieurs 18×18 bits) et un additionneur (*carry chain).* Cette dernière approche requiert un temps de cycle de l'ordre de 21.835 ns et un cycle d'horloge de 45 Mhz, qui représente un débit de calcul de 45.7 millions d'opérations par second.

Il est à notre que la méthode proposée peut s'adapter au calcul de toutes les fonctions élémentaires à un seul argument. Ceci procure à son implémentation sur FPGA l'avantage d'être totalement reconfigurable ou partiellement reconfigurable sur un même niveau de matériel.

3.11. Conclusion

Une méthode de calcul rapide des fonctions élémentaires en virgule flottante double précision, dédiée aux circuits FPGAs de Xilinx, a été présentée dans ce chapitre. Cette méthode combine lecture de table et évaluation de polynômes. Les limites des ressources en terme de mémoire des circuits Virtex-2 de Xilinx nous ont obligé à utiliser un polynôme d'approximation minimax de degré trois. Certaines stratégies de conception ont été utilisées pour optimiser le délai d'évaluation d'une fonction. La structure pipeline a été introduite dans notre architecture afin d'augmenter le débit de calcul. Les FMAs sont réalisés sans propagation de la retenue et il en résulte que le délai dans ces modules est

celui de la réduction des produits partiels. Une optimisation particulière a été accordé au compresseur 4:2, vu que de son délai dépend celui du FMA.

L'architecture implémentée est dédiée au calcul de l'exponentielle. Celle-ci peut être adaptée à l'évaluation de toutes les fonctions à un argument par reconfiguration du FPGA.

Par l'utilisation d'un circuit plus important tel que XC2V8000, on peut implémenter le calcul de deux fonctions élémentaires sur le même chip. Par conséquent, nous pouvons calculer deux fonctions élémentaires en parallèle. La reconfiguration, qui est une particularité des circuits FPGA programmables, est aussi une option pour le calcul de plusieurs fonctions élémentaires sans toutefois changer de circuit.

CONCLUSION

Cette thèse s'est articulée autour des méthodes de calcul matériel, en virgule flottante double précision de la norme IEEE-754. Nous voulons, par le biais de ce travail, offrir des solutions matérielles pour l'accélération des calculs dans les applications qui nécessitent un volume de calcul important à exécuter dans un délai de temps réduit. Nous avons dans un premier temps mené une réflexion sur certaines caractéristiques que doivent respecter les architectures que l'on avait à développer dans cette thèse. Ce qui nous a conduit à émettre une liste de caractéristiques et contraintes, en matière de représentation des nombres, de précision de calcul, de performances, du type de support d'implémentation, etc. Ceci pour réponde aux exigences des applications embarquées et à calcul intensif comme on en trouve en traitement de signal, d'images, etc.

La première partie de ce travail a été consacrée à la division où l'algorithme SRT a été choisi. L'étude algorithmique de cette méthode a montré que le nombre d'itérations dépend de la base et du facteur de redondance. Certes en augmentant la base, le nombre d'itérations diminue mais pas forcément le temps d'exécution global. Ceci est dû à l'apparition d'une opération de multiplication dans l'itération SRT. Cette multiplication du diviseur par un chiffre du quotient ($-q_{j+1} \times d$), augmentait la complexité de l'itération SRT et par conséquent le délai d'exécution de l'itération. On a reporté aussi que, dans la littérature, la base 4 est souvent utilisée pour éviter cette complexité supplémentaire. Nous avons alors, en premier lieu mesuré cette complexité et son impact pour diverses bases (4, 8 et 16). Un autre élément qui est souvent omis dans l'étude de l'itération SRT, celui-ci est le facteur de redondance, qui est pris souvent à sa valeur minimale du fait que son utilisation à sa valeur maximale introduisait lui aussi la même complexité qu'en augmentant la magnitude de la base. Cependant, l'utilisation de la valeur maximale du facteur de redondance ρ n'a pas que des inconvénients, notre étude a montré qu'en augmentant ρ la taille de la table de mémorisation des chiffres q_{j+1} diminuait. Notre travail est alors axé sur l'utilisation de cet avantage dans la réduction de la table des coefficients et de trouver une solution pour éviter la multiplication dans l'itération SRT. Pour ce faire, nous avons développé une approche efficace basée sur la décomposition des chiffres du quotient en deux ou trois termes, qui ne sont autres que des multiples entiers de deux. Il en résulte que la multiplication par l'un de ces chiffres est

obtenue par de simples décalages. Cette approche a été testée pour les bases (4, 8 et 16) pour des facteurs de redondances minimales et maximales, où une procédure en langage Maple a été réalisée pour le calcul des tailles des tables. Certes avec cette approche, la multiplication est bannie de l'itération SRT, mais l'addition se complique. Cette dernière était à deux opérandes, elle se retrouve à 3 ou 4 opérandes. Ce problème a été résolu par une solution architecturale, où nous avons utilisé la représentation carry save pour les résultats intermédiaires pour éviter la propagation de la retenue dans l'itération SRT. Une optimisation de l'arbre de réduction au niveau d'abstraction le plus bas d'un circuit FPGA, nous a permis de développer plusieurs architectures, selon notre approche. Il en résulte que la base 8 avec un facteur de redondance maximal représente le meilleur compromis entre la surface occupée et le délai d'exécution de la division. Une étude comparative de notre architecture avec des travaux similaires de la littérature a montré, qu'en plus d'être plus performante, le délai de l'itération de notre architecture est indépendant de la taille des opérandes.

La deuxième partie de notre travail, dans cette thèse, est consacrée aux fonctions élémentaires. Le but était alors de répondre aux exigences des applications fortement calculatoire, par la réalisation d'une architecture qui peut être utilisée comme un IP pour le calcul des fonctions élémentaires. Cette architecture, qui en plus d'être performante, doit être reconfigurable, dédiée aux calculs intensifs et n'utilisant que les ressources de base des circuits FPGA pour pouvoir être implémentée dans une large gamme de circuits FPGA. Six fonctions ont été considérées dans cette partie à savoir l'exponentielle (e^x), le logarithme $ln(x)$, l'inverse $(1/x)$, la racine carrée $\sqrt{x}$, le sinus $sin(x)$ et le cosinus $cos(x)$.

Une méthode de calcul rapide des fonctions élémentaires en virgule flottante double précision, dédiée aux circuits FPGAs de Xilinx, a été présentée dans cette partie. Cette méthode combine lecture de table et évaluation de polynômes. Différentes optimisations à plusieurs niveaux ont été effectuées pour mener à bien ce travail. En premier lieu on s'est penché sur la réduction du degré du polynôme d'approximation, vu que la complexité des calculs croit exponentiellement avec le degré du polynôme. Un autre facteur qui joue dans la complexité des calculs est la largeur de l'intervalle d'approximation. L'utilisation de l'approximation par segments, conjuguée aux optimisations effectuées sur les intervalles de calcul de chacune des fonctions lors de la réduction d'argument, a permis d'égaler cette complexité calculatoire pour toutes les fonctions considérées. Ce qui s'est traduit par des architectures qui ont un même niveau de consommation de ressources

matériels. Le schéma de Horner a été utilisé avantageusement, dans la mesure où d'une part il permet de diminuer le nombre de multiplications dans l'évaluation du polynôme d'approximation et d'autre part il permet de concentrer toute la complexité calculatoire dans une seule opération qui est une multiplication suivie d'une addition FMA *(Fused Multiplier Adder).* Une bonne connaissance des ressources internes du FPGA a été plus que nécessaire pour pouvoir opérer des optimisations au niveau le plus bas à savoir au niveau CLB pour l'optimisation de cette opération, vu que les performances de notre architecture dépendent étroitement de celle du FMA. Le calcul d'erreur nous a permis d'optimiser la taille des chemins des données *(data paths),* Ceci a rehaussé les performances tout en diminuant la surface occupée par nos architectures. Une autre optimisation a touché la partie mémoire, vu que celle-ci demeure le point culminant dans le choix du circuit FPGA. Comme l'architecture développée dans ce travail est appelée à être utilisée comme un IP ou cœur de coprocesseur pour l'accélération du calcul des fonctions élémentaires, celle-ci ne doit pas consommer toutes les ressources du FPGA. Pour ce faire, nous avons alors choisi d'utiliser que la mémoire disponible dans les blocs BRAM. Le calcul d'erreur a permis aussi de tronquer les coefficients des polynômes, sans toutefois dépasser la précision permise qui est de $1ulp$. Il en résulte des coefficients avec des tailles réduites, ce qui contribue à la diminution de la taille totale des tables des coefficients et par conséquent la mémoire consommée par notre architecture.

Certaines stratégies de conception ont été utilisées pour optimiser le délai d'évaluation d'une fonction. La structure pipeline a été introduite dans notre architecture afin d'augmenter le débit de calcul. Les FMAs sont réalisés sans propagation de la retenue et il en résulte que le délai dans ces modules est celui de la réduction des produits partiels. Une optimisation particulière a été accordé au compresseur 4:2, vu que de son délai dépend celui du FMA.

L'architecture résultante est dédiée au calcul de l'exponentielle. Celle-ci peut être adaptée à l'évaluation de toutes les fonctions par reconfiguration du FPGA.

REFERENCES

1. Wikipédia, l'encyclopédie libre, " Calcul intensif", disponible à http://fr.wiktionary.org/wiki/calcul_intensif.

2. J. Nado, G. Renard, M. Tricot, S. Zaki, "FPGA des prévisions météorologiques à plus long terme", disponible à : http://www.diic.fr/jahia/webdav/site/mywebsiteifsic/shared/ingenieur/Ecrits_Ingenieurs/2007_FPGA_previsions_meteo.pdf.

3. P. Morris, "Streamlining SoC Design from ESL to GDSII", Information Quarterly, Volume 5, Number 2, 2006.

4. J.-M. Muller, "Arithmétique des ordinateurs". Masson, 1989.

5. J.-L. Beuchat, A. Tisserand, "Opérateur en-ligne sur FPGA pour l'implantation de quelques fonctions élémentaires", RENPAR'14 / ASF / SYMPA, Hamamet, Tunisie, 10–13 avril 2002.

6. M. Lu, "Arithmetic and Logic in Computer Systems" , A John Wiley & Sons, INC., Publication, 2004.

7. American National Standards Institute and Institute of Electrical and Electronic Engineers, "IEEE Standard for Binary Floating-Point Arithmetic". ANSI/IEEE Standard 754–1985, 1985.

8. IEEE Computer Society. "IEEE Standard for Floating-Point Arithmetic, IEEE Standard 754-2008", August 2008.
Disponible à http://ieeexplore.ieee.org/servlet/opac?punumber=4610933.

9. American National Standards Institute and Institute of Electrical and Electronic Engineers. "IEEE Standard for Radix Independent Floating-Point Arithmetic". ANSI/IEEE Standard 854–1987, 1987.

10. J-M. Muller, N. Brisebarre, F. Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, & S. Torres, "Handbook of Floating-Point Arithmetic", Birkhäuser Boston 2009.

11. G. Snider, P. Kuekes, W. B. Culbertson, R. J. Carter, A.S. Berger & R. Amerson, "The Teramac configurable computer engine", International Workshop on Field Programmable Logic and Applications, pages 44–53. LNCS 975, 1995.

12. D. Stauffer, "Systèmes numériques câblés et micro-programmés", Presses Polytechniques et Universitaires Romandes (PPUR) 1999.

13. N. Veyrat-Charvillon, "Opérateurs arithmétiques matériels pour des applications spécifiques ", thèse de Doctorat de l'Ecole Normale Supérieure de Lyon, Laboratoire d'Informatique et du Parallélisme, Juin 2007.

14. T. Noergaard, “Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers (Embedded Technology)”, Publisher: Newnes (February 24, 2005) 656 pages ISBN-10: 0750677929.

15. P. Patel, “Embedded systems design using FPGA”. 19th International Conference on VLSI Design, 2006, Held jointly with 5th International Conference on Embedded Systems and Design Issue, 3-7 Jan. 2006 Page(s): 1 pp. -Digital Object Identifier 10.1109/VLSID.2006.83.

16. C. K. Koc¸, C. Paar, “Guest Editors’ Introduction to Special Section on Cryptographic Hardware and Embedded Systems”, IEEE Trans. On Computers, Vol. 52, N° 4, April 2003.

17. R. Baxter, et al, “High-Performance Reconfigurable Computing – the View from Edinburgh”, Proc. AHS2007 Conf., Second NASA/ESA Conference on Adaptive Hardware and Systems, Edinburgh, 2007.

18. R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. Mc Cormick, G. Smart, R. Smart, A. Cantle, R. Chamberlain, G. Genest, “Maxwell–a 64 FPGA Supercomputer” EPCC & FHPCA, Edinburgh; Second NASA/ESA Conference on Adaptive Hardware and Systems, 2007. AHS 2007, pp 287-294, 5-8 Aug. 2007.

19. R. Baxter, S. Booth, “The FPGA High-Performance Computing Alliance Parallel Toolkit” Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems Pages: 301-310, 2007, ISBN: 0-7695-2866-X.

20. E. Garcia, “Bien concevoir avec un FPGA”, extrait du numéro 125 d’Electronique, Mai 2002.

21. “Virtex-4 User Guide”, ug070 (v2.2) April 10, 2007. Disponible à http://www.xilinx.com/support/documentation/user_guides/ug070.pdf.

22. “Virtex-6 FPGA Configurable Logic Block: user guide”, disponible à http://www.xilinx.com/support/documentation/user_guides/ug364.pdf.

23. “Virtex-5 LXT FPGAs”, disponible à http://www.xilinx.com/products/virtex5/lxt.htm.

24. “Xilinx ISE 8 Software Manuals and Help”, disponible à http://www.xilinx.com/itp/xilinx8/books/manuals.pdf.

25. M. D. Ercegovac, et T. Lang, “ Digital Arithmetic ”. Morgan Kaufmann, 2003.

26. B. Parhami, “Computer Arithmetic: Algorithms and Hardware Designs”. Oxford University Press, 2000.

27. A. Liddicoat, “High performance arithmetic for division and elementary functions,” Ph.D. dissertation, Stanford University, Stanford, CA, Feb. 2002.

28. F. Oberman, et M.J. Flynn, “Design Issues in Division and Other Floating-Point Operations”. IEEE transactions on computers, vol. 46, No. 2, February 1997.

29. M.D. Ercegovac, et T. Lang, “On the Fly Rounding”. IEEE Transactions on computers, Vol. 41, No. 12, December 1992.

30. J.E. Robertson, “A new class of digital division methods”. IEEE Transactions on Electronic Computers, –EC-7, pp 218–222, September 1958.

31. K.D. Tocher, “Techniques of multiplication and division for automatic binary computers”. Quarterly Journal of Mechanics and Applied Mathematics, 11-part 3, pp.368–384, 1958.

32. H.A.H. Fahmy, “A Redundant Digit Floating Point Systeme”, PhD dissertation, Stanford University June 2003.

33. P. Kornerup, “Digit selection for SRT division and square root”. IEEE Transactions on Computers, vol.54, no.3, pp294–303, mars 2005.

34. M. B. Monagan & al. “Maple Introductory Programming Guide Maple soft”, a division of Waterloo Maple Inc. 1996-2008.

35. J.L. Beauchat et A. Tisserand, “Small Multiplier-Based Multiplication and Division Operators”. Proceeding of 12th Conference on Field Programmable Logic and Applications. pp. 513–522, 2002.

36. X. Wang et E. Nelson, “Tradeoffs of Designing Floating Point Division and Square Root on Virtex FPGAs”, Proceeding of 11th IEEE symposium on Field Programmable to Custom Machines, FCCM’2003.

37. H. Bessalah, M. Anane, M. Issad, & N. Anane, “Division SRT sur Virtex-II, pour des bases B et des facteurs de redondances ρ élevés”, RenPar’17/ SympA’2006/CFSE’5/JC’2006 Canet en Roussillon, France 4 au 6 octobre 2006.

38. Xilinx Inc., “Virtex-2 Platform User Guide”. Chp 3, Par: Design Considerations, 2004.

39. R. A. Weinberger, “4:2 Carry-save Adder module”, *IBM Technical Disclosure, Bull.*, vol. 23, Jan. 1981.

40. R. Michard, A. Tisserand, & N. Veyrat-Charvillon, “Divgen: A divider unit generator”, Proc. of Advanced signal processing algorithms, architectures, and implementations. Vol. 5910, pages 1-12, San Diego, CA, ETATS-UNIS (02/08/2005).

41. B.R. Lee, N. Burgess, “Improved small multiplier based multiplication, squaring and division,” Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), pp 91-100, Napa, CA, 2003.

42. J. Detrey, Florent de Dinechin, “Fonctions élémentaires en virgule flottante pour les accélérateurs reconfigurables” RSTI - TSI. Volume 27 – n° 6/2008, pages 673-698.

43. W. Cody, W. Waite, “Software manual for the elementary Functions”, Prentice Hall, 1980.

44. J-M. Muller, ″Elementary functions, algorithms and implementation″, Brikhauser 1997.

45. Alvaro Vazquez, Julio Villalba & Elisardo Antelo “Computation of Decimal Transcendental Functions Using the CORDIC Algorithm” 19th IEEE International Symposium on Computer Arithmetic Portland, Oregon, USA 08-10 June 2009.

46. C. S. Anderson S. Story, N. Astafiev, “Accurate math functions on the Intel IA-32 architecture: A performance-driven design », 7th Conference on Real Numbers and Computers, Nancy, France, July, 2006, p. 93-105.

47. N. Takagi, “Powering by table look-up and multiplication with operand modification”, IEEE Trans. on Computer, 47(11), pp 1216-1222, 1998.

48. J-.M. Muller, “Calcul des fonctions élémentaires à l’aide des tables”, Sympa' 5 Rennes, Juin 8-11, 1999.

49. J. Detrey and F. de Dinechin, “Table-based polynomials for fast hardware function-evaluation. In Application-specific Systems, Architectures and Processors”, pages 328–333. IEEE, 2005.

50. D. Das Sarma & W. Matula, “Faithful Bipartite Rom Reciprocal Tables” Proc. of the 12th IEEE Symp. On Computer Arithmetic, pp 11- 28, 1995, Bath, UK.

51. J.-M. Muller. “A few results on table-based methods”. Reliable Computing, 5(3) : 279–288,1999.

52. F. de Dinechin and A. Tisserand. “Multipartite table methods”. IEEE Transactions on Computers, 54(3) pp. 319–330, 2005.

53. J.A. Pineiro, S.F. Oberman, J.M. Muller & J.D. Bruguera, “High-Speed Function Approximation using a Minimax Quadratic Interpolator”, IEEE trans. on computer, vol. 54, n°3, March 2005

54. W. F. Wong, E. Goto, “Fast Hardware-Based Algorithms for Elementary Function Computations Using Rectangular Multipliers”, IEEE Transactions on Computers, vol. 43, n°3, 1994, p. 278-294.

55. J. Detrey and F. de Dinechin. “Second order function approximation using a single multiplication on FPGAs. In 14th Intl Conference on Field-Programmable Logic and Applications” (LNCS 3203), pages 221–230. Springer, Août 2004.

56. Steve Kilts “Advanced FPGA Design Architecture, Implementation, and Optimization”, Published by John Wiley & Sons, Inc., Hoboken, New Jersey, 2007.

57. "Virtex™-2 1.5 V Field programmable gate arrays", Advance product specification, DS031-1 (v1.7) October 2001, http://www.xilinx.com/

58. J.-M. Muller. "On the definition of ulp(x)". Research Report RR2005-09, Laboratoire de l'Informatique du Parallélisme (LIP), February 2005.

59. I. Koren. "Computer arithmetic algorithms". Prentice-Hall, 1993.

60. J.-M. Muller. "Elementary Functions, Algorithms and Implementation". Birkhäuser, 2nd edition, 2006.

61. A.S Noetzel, "An Interpolating Memory Unit for Function Evaluation: Analysis and Design", IEEE Transactions on Computers, vol. 38, No 3, March 1989.

62. N. Brisebarre, D. Defour, P. Kornerup, J-M Muller and N. Revel, "A new range reduction algorithm", IEEE Trans. On Computers, Vol. 54, n° 3, March 2005.

63. K. C. Ng. "Argument reduction for huge arguments: good to the last bit". Technical report, SunPro, Mountain View, CA, USA, Juil. 1992.

64. F.J. Jaime, Julio Villalba, Javier Hormigo, Emilio L. Zapata , "Pipelined Architecture for Additive Range Reduction", Journal of Signal Processing Systems ,Volume 53, Issue 1-2 (November 2008), Pages: 103 – 112, ISSN:1939-8018.

65. Stefan Lachowicz, Hans-J. Pfleiderer, "Fast Evaluation of the Square Root and Other Nonlinear Functions in FPGA", delta, pp. 474-477, 4th IEEE International Symposium on Electronic Design, Test and Applications (delta 2008), 2008.

66. Detrey J., de Dinechin F., "A parameterized floating-point exponential function for FPGAs", *dans* G. Brebner, S. Chakraborty, W.-F.Wong (eds), IEEE International Conference on Field-Programmable Technology (FPT'05), pp. 27-34, Singapore, December, 2005.

67. Detrey J., de Dinechin F., "A parameterizable floating-point logarithm operator for FPGAs" , 39th Asilomar Conference on Signals, Systems & Computers, IEEE Signal Processing Society, Pacific Grove, CA, USA, November, 2005, pp. 1186-1190.

68. G.W.BEWICK, "Fast multiplication, algorithms and implementation", PhD thesis, Stanford University, 1994.

69. E.L. Braun, "Digital Computer Design", New York Academic, 1963

70. C.S. Wallace "A Suggestion for a Fast Multiplier", IEEE Transactions on Electronic Computers, vol. 13, 1964, p. 14-17

71. L. Dadda, "On Parallel Digital Multipliers", Alta freq, vol 45, 1976, p. 574-580.

72. Jean-Pierre Deschamps, Gery J. Bioul and Gustavo D. Sutter, "A Synthesis of Arithmetic Circuits: FPGA, ASIC and Embedded Systems", John Wiley & Sons, Inc., publication. March 2006.

73. O. Mencer, Wayne Luk, "Parameterized High Throughput Function Evaluation for FPGAs", The Journal of VLSI Signal Processing, Special Issue on Reconfigurable Computing, Kluwer, March 2004.

74. J.A. Pineiro, J.D. Bruguera, J.M. Muller, "FPGA implementation of a Faithful Polynomial Approximation for Powering Function computation", *Proc. EUROMICRO.* Symposium on Digital System Design, DSD'2001 Warsaw, 2001.

Nom : *ANANE*
Prénom : *Mohamed.*
E-Mail: m_anane@esi.dz

PRODUCTION SCIENTIFIQUE DEPUIS MA PREMIÈRE INSCRIPTION EN 2004/2005

1- Publications dans des revues Internationales

- N.Anane, M.Anane**,** H.Bessalah, M.Issad & K.Messaoudi **HARDWIRED ALGORITHM FOR VARIABLE AND LONG PRECISION MULTIPLICATION ON FPGA**. The Mediterranean Journal of Computers and Networks, Volume 5, No. 4, October 2009. ISSN: 1744-2397

- M.Anane, H.Bessalah, M.Issad, N.Anane & H.Salhi, **"Higher Radix and Redundancy factor for floating point SRT Division"**, Journal IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol, 16, No 6, juin 2008

2- Communications Internationales

- Anane Nadjia, Anane Mohamed, Bessalah Hamid, Issad Mohamed & Messaoudi khadidja, "**RSA Based Encryption/Decryption of Medical Images**", in the 7th International Multi-Conference on Systems, Signals and Devices IEEE-SSD'2010, AMMAN, Jordan June 27-30 2010

- M. Anane, N.Anane, H.Bessalah & M.Issad, "***Reconfigurable Architecture for Elementary Functions Evaluation***", 2009 ACS/IEEE International Conference on Computer Systems and Applications, AICCSA'2009.May 10-13, 2009. Rabat, Morocco.

- N.Anane, M. Anane, H.Bessalah, M.Issad & K.Messaoudi, "***Hardware Algorithm for Variable Precision Multiplication on FPGA***", 2009 ACS/IEEE International Conference on Computer Systems and Applications, AICCSA'2009.May 10-13, 2009. Rabat, Morocco.

- H.Bessalah, M.Anane, K.Messaoudi, M.Issad & N.Anane , "***Left to Right Serial Multiplier for Large Numbers on FPGA",*** I Proceedings of the 2009 IEEE International Conference on Mechatronics. Málaga, Spain, April 2009.

- H. Bessalah, M. Anane, M. Issad, N. Anane & K. Messaoudi, "**Digit Recurrence Divider: Optimization and Verification"**, DTIS'07, IEEE

International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS'07) September 2 - 5, Rabat (Morocco).

- H.Bessalah, M.Anane, M.Issad, N.Anane & K.Messaoudi, **" Division SRT sur Virtex-II, pour des base B et des facteurs de redondances ρ élevés "** Symposium en Architecture des machines SympA'2006, 3 - 6 octobre 2006 Perpignan France.

- H.Bessalah, M.Anane, N.Anane, M.Issad & H.Salhi **"Architecture re-configurable pour le calcul des fonctions élémentaires à haut débit sur Virtex-II "** Symposium en Architecture de machines SympA'2006, 3 - 6 octobre 2006 Perpignan, France.

- H.Bessalah, M.Anane, M.Issad, N.Anane; K.Messaoudi , **" The Redundancy factor effect on SRT division performance"**, International Computer Systems and Information Technology conference ICSIT'05. July 2005 Algiers ALGERIA

- H.Bessalah, M.Anane, N.Anane, M.Issad & H.Salhi "**Re-Configurable Architecture for Elementary Functions Evaluation**", International Computer Systems and Information Technology conference ICSIT'05. July 2005 Algiers ALGERIA.

- M.Issad, M. Anane & H.Bessalah, '**Influence de la base sur les performances de calcul de la division SRT'**. JFAA'2005, 18 - 21 janvier 2005 Dijon (Bourgogne) France.

- M. Anane, H.Bessalah, N. Anane & M.Issad, '**Architecture Pipeline pour le Calcul Rapide des Fonctions Elémentaires sur FPGA**'. JFAA'2005, 18 - 21 janvier 2005 Dijon (Bourgogne) France.

3- Communications nationales

- Anane Mohamed, Anane Nadjia, Bessalah Hamid, Issad Mohamed, Messaoudi khadidja **Software Implementation of the RSA Cryptosystem** First International Congress On Models, Optimization and Security of systems ICMOSS'2010. Ibn Khaldoun University of Tiaret May 29-31, 2010

- Anane Nadjia, Anane Mohamed, Bessalah Hamid, Issad Mohamed, Messaoudi khadidja, "**Medical Images Encryption RSA Based**" , International Conference on Applied Informatics, Centre Universitaire de Bordj Bou-Arréridj Nov 15-17, 2009.

- M.Anane, N.Anane, H.Bessalah & M.Issad ***Architecture Reconfigurable pour le Calcul des Fonctions Élémentaires*** **ESC'09,** Embedded Systems Conference 04-06 Mai 2009 EMP–Alger.

- M.Issad, H.Bessalah, M.Anane, N.Anane & K.Messaoudi, ***An FPGAImplementation of a Generic Modular Adder/Subtracter over GF(M).*** **ESC'09,** Embedded Systems Conference 04-06 Mai 2009 EMP–Alger.

- N.Anane, M.Anane, H.Bessalah, M.Issad et K.Messaoudi, "***Algorithme matériel pour le calcul de la multiplication à précision variable sur circuit FPGA*** ", DAT'08, cercle national de l'armée (Beni Messous) Alger, du 23 au 25 novembre 2008.

- H.Bessalah, K.Messaoudi, M.Anane, M.Issad & N.Anane "***Multiplieur série poids fort en tête pour le calcul en longue précision sur FPGA***", DAT'08, cercle national de l'armée (Beni Messous) Alger, du 23 au 25 novembre 2008.

- H.Bessalah, M.Issad, L.Mahieddine, K.Messaoudi, N.Anane & M.Anane, ***"Implémentation d'un UART sur FPGA pour applications en systèmes embarqués*** ", ICEE'08, 20-21 Octobre 2008, Oran.

- M.Anane, H.Bessalah, N.Anane, **"Reconfigurable High Throughput Function Evaluation for virtex-2 FPGA"**, Conférence Internationale sur la Microélectronique et la Nanotechnologie " ICMNT2006", Tizi–Ouzou, 19-23 Novembre 2006.