RenPar'19 / SympA'13 / CFSE'7
Toulouse, France, du 9 au 11 septembre 2009

Racines carrées multiplicatives sur FPGA

Florent de Dinechin, Mioara Joldes, Bogdan Pasca, Guillaume Revy

Projet Arénaire, Laboratoire de l'Informatique du Parallélisme (CNRS/INRIA/ENS-Lyon/UCBL)
École Normale Supérieure de Lyon
46, allée d'Italie – 69364 Lyon Cedex 07

Résumé
Les implantations actuelles de la racine carrée dans des bibliothèques d'opérateurs pour FPGA utilisent presque toutes une récurrence à base d'additions. Ce choix est bien adapté à la structure des blocs logiques élémentaires d'un FPGA. Toutefois, il peut être remis en question à présent que la plupart des FPGA haute-performance incluent un grand nombre de blocs multiplieurs et de blocs mémoires. Cet article discute l'implantation d'une racine carrée en virgule flottante en utilisant ces nouvelles ressources, et compare les performances obtenues avec l'approche classique.

Mots-clés : Virgule flottante, FPGA, racine carrée, polynômes

---

1. Introduction

1.1. Extraire des racines carrées flottantes

Il y a deux grandes familles d'algorithmes pour évaluer la racine carrée en matériel.
La première regroupe des récurrences qui donnent un chiffre (souvent un bit) du résultat à chaque itération, chaque itération étant elle-même construite autour d'une addition [5]. Cette famille a été utilisée pour calculer des racines carrées dans les microprocesseurs qui ne disposaient pas encore de multiplieurs. La plupart des implantations pour FPGA disponibles dans les outils ou publiées [9, 8, 4] utilisent aussi cette approche, qui s'est imposée à l'époque où les FPGAs avaient une architecture uniforme à base de cellules reconfigurables à grain très fin. C'est sans doute cette approche qui minimise la complexité en termes d'opérations logiques du calcul de la racine carrée.
La seconde famille utilise des multiplications, et est pertinente pour les processeurs qui possèdent du matériel pour la multiplication. On y trouve les récurrences quadratiques dérivées de l'itération de Newton-Raphson, utilisées dans les processeurs IA32 d'AMD à partir du K5 [13] et dans les processeurs comme Power/PowerPC et Itanium dont l'unité flottante est à base de multiplieur-additionneur fusionné (FMA) [10, 1]. On y trouve aussi des approches à base d'approximation polynomiale par morceaux [6] ou un mélange des deux [12]. Pour les FPGAs, Wang et Leeser, pour le projet VFLOAT [14], utilisent une réduction d'argument à base de tables et de multiplications, suivi d'une approximation polynomiale pour l'argument réduit. Alors que l'approche par récurrence de chiffres cherche à construire un matériel minimal, l'approche multiplicative cherche à utiliser au mieux des ressources données : deux FMA dans Power et Itanium, deux multiplieurs 32 bits dans le ST200 utilisé dans [6].
À présent que tous les FPGA haut-de-gamme intègrent de petits multiplieurs (jusqu'à 2000 multiplieurs typiquement capables de multiplier exactement deux entiers signés de 18 bits), il convient d'explorer la pertinence de cette seconde famille d'algorithmes pour implémenter la racine carrée dans les FPGA [7]. Dans cet article, on étudie essentiellement l'approche polynomiale par morceaux, qui est originale dans ce contexte. C'est une première contribution de cet article. Une version simple précision IEEE-754 (8 bits d'exposant et 23 bits de fraction) est complètement implémentée en plusieurs variantes, et une version double précision est étudiée en détail.
Une autre contribution est de comparer quantitativement ces différentes approches : récurrence de chiffre, approche polynomiale, Newton-Raphson. On verra que, pour la simple précision, l'approche polynomiale est de loin la meilleure des approches multiplicatives publiées jusque là, et offre un compromis ressources/fréquence/latence intéressant comparée à l'approche par récurrence de chiffres. On

verra aussi que, pour la double précision, aucune des approches multiplicatives existantes, y compris celle introduite par cet article, ne présente un avantage clair sur l'approche historique. C'est un résultat inattendu au vu de la puissance de calcul introduite par les multiplieurs des FPGA.

L'objectif à terme de ces travaux est d'implanter les meilleurs algorithmes pour la racine carrée dans le projet FloPoCo[1], un générateur libre de cœurs arithmétiques [2] pour le calcul haute performance sur FPGA. FloPoCo intègre déjà l'algorithme par récurrence de chiffres dans sa version la plus simple (base 2), qui est celui utilisé par la plupart des bibliothèques académiques [8, 4] et des outils de synthèse (Xilinx, Altera). Il offre aussi la version polynomiale, mais uniquement en simple précision.

1.2. Travaux précédents

À notre connaissance, la seule architecture pour la racine carrée à base de multiplieurs actuellement disponible pour FPGA est celle de Wang, Braganza et Leeser dans le projet VFLOAT déjà mentionné [14]. L'utilisation d'outils récents d'approximation polynomiale permet des améliorations significatives par rapport à [14] :

– Qualitativement, nous offrons un choix entre arrondi correct compatible IEEE-754 et un arrondi fidèle bien spécifié (l'opérateur est précis au dernier bit). La racine carrée de VFLOAT est bien moins précise (au moins 2.39 fois d'après [14]).
– Quantitativement, une approche par évaluation polynomiale par morceaux est plus flexible, et permet d'utiliser les ressources du FPGA au plus juste. Elle permet également un compromis entre plus de multiplieurs ou plus de mémoires – nous choisissons d'équilibrer les deux. Au final, elle s'avèrera plus économe, comme le montrera la section 5.

Un exposé de M. Langhammer à RSSI'08 [7] suggère qu'Altera développe des coeurs arithmétiques centrés sur les multiplieurs. Cet exposé mentionne un opérateur de racine carrée inverse (il s'agit donc sans doute d'une itération de Newton-Raphson). Ses performances seront reprises dans la comparaison de la section 5.

Enfin, ce travail s'est beaucoup inspiré de la racine carrée du projet FLIP [6], une bibliothèque logicielle d'opérateurs flottants pour processeurs sans unité flottante.

1.3. Adéquation des algorithmes à base de multiplieurs aux architectures de FPGA modernes

Voici les caractéristiques des FPGA ciblés qui nous intéressent.

– Les multiplieurs sont capables de multiplication en virgule fixe 18x18 bits signés, ou 17x17 non signée[2], et retournent tous les bits du produit. On peut construire des multiplieurs plus grands en assemblant ces multiplieurs [3], mais les tailles optimales sont très discrètes : $17i \times 17j$ pour i et j entiers. En fait, les multiplieurs sont intégrés au sein de blocs plus complexes incluant également des additionneurs et des registres spécifiques, mais c'est d'une importance secondaire ici – on laisse aux outils de synthèse logique le soin d'exploiter ces ressources.
– Les mémoires ont une capacité de 9Kbit (Altera) ou 18Kbit (Xilinx) et sont configurables en termes de tailles d'adresse et tailles de données, par exemple de $2^{16} \times 1$ à $2^9 \times 36$ pour le Virtex-4.
– Un FPGA donné contient grosso modo autant de mémoires que de multiplieurs. Dans un premier temps, nous chercherons donc à équilibrer le nombre de mémoires et le nombre de multiplieurs consommés par un opérateur donné. Toutefois, les ressources disponibles dépendent aussi du reste de l'application, et à terme il serait souhaitable d'offrir différents compromis entre multiplieurs et mémoire.

2. La racine carrée par approximation polynomiale par morceaux

Soit à calculer la racine carrée d'un nombre flottant normal x dans un format IEEE-754 :

$$x = 2^e \times 1, f$$

où e est un exposant entier relatif[3] et f est la partie fractionnaire de la mantisse, écrite en binaire sur w_F bits $f_{-1}f_{-2}\cdots f_{-w_F}$ (les indices des bits dénotent leur poids).

[1] `http://www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/`

[2] Chez Altera ils sont légèrement meilleurs, puisqu'ils sont de plus configurables pour une multiplication 18x18 non signée.

[3] L'exposant est codé par un entier positif de w_E bits auquel il faut soustraire le biais normalisé $2^{w_E-1} - 1$, mais ceci est sans importance ici.

Il y a classiquement deux cas à considérer.

Si e **est pair,** la racine carrée s'écrit

$$\sqrt{x} = 2^{e/2} \times \sqrt{1,f}\,.$$

Si e **est impair,** la racine carrée s'écrit

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

Dans les deux cas, le calcul de l'exposant du résultat s'obtient par un simple décalage de e (et la gestion du biais) et nous n'y reviendrons pas.
Le problème se ramène donc à calculer $\sqrt{z}$ pour $z \in [1,4[$.

2.1. Approximation polynomiale par morceaux

On va couper l'intervalle $[1,4[$ en morceaux et utiliser sur chaque morceau un polynôme dont les coefficients seront lus dans une table. On utilise pour obtenir de tels polynômes la commande `fpminimax` de l'outil Sollya[4].
Une première idée est d'utiliser, pour adresser cette table, les bits de poids fort de z. Toutefois, comme dans l'intervalle $[1,4[$ les bits de poids fort ne prennent jamais la valeur 00xxx, cela laissera un quart de la table inutilisé. Par ailleurs, pour un degré de polynôme donné et une taille de morceau donnée, les polynômes utilisés sur la gauche de l'intervalle $[1,4[$ sont moins précis que ceux utilisés sur la droite (la fonction $\sqrt{z}$ varie plus pour les petits z). Pour résoudre ces deux problèmes, on choisit de distinguer à nouveau les cas d'exposant pair et impair : l'intervalle $[1,2]$ (cas pair) sera coupé en deux fois plus de morceaux (deux fois plus petits) que l'intervalle $[2,4]$.
Voici les détails de l'algorithme ainsi obtenu. Soit k un paramètre entier qui va déterminer le nombre de morceaux (2^k en tout). La table des coefficients des polynômes aura donc 2^k entrées. La détermination de la valeur à donner à ce paramètre sera discutée en 2.2, en fonction des caractéristiques des FPGAs survolées en 1.3.

Si e **est pair,** soit $\tau_{\text{pair}}(x) = \sqrt{1+x}$, pour $x \in [0,1)$. C'est τ_{pair} dont on va calculer une approximation polynomiale par morceaux. L'intervalle $[0,1[$ est découpé en 2^{k-1} intervalles de la forme $[\frac{i}{2^{k-1}}, \frac{i+1}{2^{k-1}}[$ pour i allant de 0 à $2^{k-1}-1$. L'indice de polynôme i est constitué des bits $f_{-1}f_{-2}\cdots f_{-k+1}$ de la mantisse de x. Sur chacun de ces intervalles, on approche $\tau_{\text{pair}}(1 + \frac{i}{2^{k-1}} + y)$ par un polynome de degré d : $p_i(y) = c_{0,i} + c_{1,i}y + \cdots + c_{d,i}y^d$. Le degré d est choisi pour que chacun de ces polynômes donne un résultat précis au moins à w_F bits, avec de la marge pour les arrondis du calcul, c'est-à-dire : $|\tau_{\text{pair}}(1 + \frac{i}{2^{k-1}} + y) - p_i(y)| \leq 2^{-w_F-2}$ pour tout $y \in [0, 1/2^{k-1}[$. Comme le degré d détermine le nombre de multiplieurs employés, il sera également discuté en 2.2.

Si e **est impair,** on doit calculer $\sqrt{2 \times 1,f}$. Soit $\tau_{\text{impair}}(x) = \sqrt{2+x}$ pour $x \in [0,2[$. L'intervalle $[0,2[$ est coupé également en 2^{k-1} intervalles, qui sont à présent de la forme $[\frac{j}{2^{k-2}}, \frac{j+1}{2^{k-2}}[$ pour j allant de 0 à $2^{k-1}-1$. Le lecteur peut vérifier que l'indice de polynôme j est constitué des mêmes bits $f_{-1}f_{-2}\cdots f_{-k+1}$ que dans le cas pair. Sur chacun de ces intervalles on cherche un approximant polynomial q_j de degré d (le même que pour le cas pair, car on veut naturellement partager le matériel d'évaluation polynomiale), avec la même précision : $|\tau_{\text{impair}}(1+\frac{j}{2^{k-2}}+y) - q_j(y)| \leq 2^{-w_F-2}$ pour tout $y \in [0, 1/2^{k-2}[$.

On obtient bien 2^k polynômes dont les coefficients seront stockés dans une ROM à 2^k entrées adressée par $A = e_0 f_{-1} f_{-2} \cdots f_{-k+1}$ (on a ajouté, aux bits déterminant i ou j, le bit e_0 de poids faible de l'exposant qui détermine les cas pair/impair). Reste à construire l'argument réduit y dans les deux cas.
– Dans le cas pair on a $1,f_{-1}\cdots f_{-w_F} = 1 + 0,f_{-1}\cdots f_{-k+1} + 2^{-k+1}0,f_{-k}\cdots f_{-w_F}$.
– Dans le cas impair on calcule la racine carrée de $2 \times 1,f = 1f_{-1},f_{-2}\cdots f_{-w_F} = 1 + f_{-1},f_{-2}\cdots f_{-k+1} + 2^{-k+2}0,f_{-k}\cdots f_{-w_F}$.

Comme on veut construire une architecture d'évaluation polynomiale en virgule fixe qui traite les deux cas, on aligne leurs virgules : notre argument réduit sera $y = 2^{-k+2} \times 0,0f_{-8}\cdots f_{-w_F}$ dans le cas pair et $y = 2^{-k+2} \times 0,f_{-k}\cdots f_{-w_F}0$ dans le cas impair.

[4] `http://sollya.gforge.inria.fr/`

2.2. Coût matériel d'un évaluateur polynomial

Il est clair que plus k est grand (plus les tables sont grosses), plus le degré pourra être petit (moins on aura besoin de multiplieurs), et inversement. Toutefois, il n'est pas si simple de trouver analytiquement l'architecture optimale, à cause de plusieurs effets non linéaires qui se combinent :

- Le plus petit k qui permet d'utiliser complètement un bloc mémoire est 9. Si l'on utilise un bloc mémoire, autant le remplir complètement.
- La consommation de blocs mémoires croît exponentiellement avec $k-9$ et linéairement avec d.
- En fait les coefficients n'ont pas besoin d'être stockés tous sur la même précision. Grossièrement, c_0 sera sur un peu plus de w_F bits, et c_{i+1} sera sur $k-2$ bits de moins que c_i. En effet, considérant $p(y) = \sum_{i=0}^{d} c_i y^i$ et le fait que $y < 2^{-k+2}$, ce choix rend chaque terme aussi précis que c_0, ni plus, ni moins. Cette explication sommaire suffit à se forger une intuition qui est vérifiée sur les polynômes calculés par Sollya.
- La consommation de multiplieurs croît quadratiquement avec $\lfloor w_F/17 \rfloor$ et linéairement avec d. Il y a un important effet d'escalier : une multiplication de 17x17 bits consomme un multiplieur, mais une multiplication de 19x19 bits en consomme 4, ou bien un seul multiplieur et beaucoup de logique autour [3]. La latence suit la même règle.
- Augmenter le degré d'une unité fait typiquement gagner de l'ordre de k bits de précision.

Considérant tout ceci, le plus simple est sans doute, pour quelques valeurs de k, de construire les polynômes, de vérifier qu'ils sont assez précis (en tenant compte de l'erreur d'approximation mais aussi des erreurs d'arrondis) et d'évaluer le coût des architectures correspondantes pour garder la ou les meilleures. Pour cela, terminons notre étude par une discussion de l'évaluation de ces polynômes et des erreurs d'arrondis associées.

2.3. Évaluation des polynômes et analyse d'erreur

On va évaluer le polynôme extrait de la table au moyen du schéma de Horner :

$$p(y) = c_0 + y \times (c_1 + y \times (\cdots + y \times (c_{d-1} + y \times a_d) \cdots)$$

La figure 1 illustre, dans le cas de la simple précision avec $k = 8$, l'alignement des différentes données et valeurs intermédiaires lors de l'évaluation polynomiale.

Une grande qualité du schéma de Horner est que toutes les multiplications sont par y, l'argument réduit, un nombre de $w_F - k + 1$ bits seulement. On aura donc une utilisation optimale des multiplieurs si $w_F - k + 1$ est proche d'un multiple de 17. C'est ce fait qui rend cette approche meilleure que les autres approches multiplicatives pour la simple précision : on arrive à avoir y sur moins de 17 bits. L'autre multiplicande sera de la taille du coefficient a_i correspondant : comme y est petit ($y < 2^{-k+2}$), a_i domine l'addition $a_i + y \times (\cdots)$. On cherche donc à minimiser les tailles des coefficients.

Par ailleurs, dans l'addition $a_i + y \times (\cdots)$, le produit $y \times (\cdots)$ est beaucoup plus précis que a_i. Il doit donc être tronqué, ce qui introduit une erreur d'arrondi au pire égale au poids du dernier bit conservé. Toutefois on remarque que cette erreur est ensuite (sauf au dernier pas de Horner) multipliée par $y < 2^{-k+2}$: la contribution de ces erreurs de troncature à l'erreur totale est donc minime, sauf pour la dernière. Les additions, elles, sont réalisées en virgule fixe, donc exactes. L'accumulation de toutes les erreurs d'arrondi est calculée automatiquement facilement grâce à l'outil Gappa[5] de G. Melquiond [11]. On la combine avec l'erreur d'approximation (calculée, elle, par Sollya) entre le polynôme et la racine carrée exacte, pour obtenir l'erreur totale de ce schéma d'évaluation. On verra cela plus en détail en 4, ainsi que la manière d'obtenir l'arrondi fidèle ou l'arrondi correct.

3. Une architecture simple précision en détail

L'implantation actuelle pour la simple précision utilise $k = 8$. Ceci nous donne un argument réduit y sur 17 bits (les 16 bits de $f_{-8} \cdots f_{-23}$, plus un 0 à droite ou à gauche). Un polynôme de degré 2 peut alors être utilisé.

Un script Sollya détermine que les coefficients de chaque polynôme vérifient, sur chaque intervalle, $1 \leq c_0 < 2$ (on ne stockera pas le bit de poids 0, qui est toujours 1) ; $c_1 < 2^{-1}$; $c_2 < 2^{-3}$. Ceci détermine les

[5] http://lipforge.ens-lyon.fr/www/gappa/

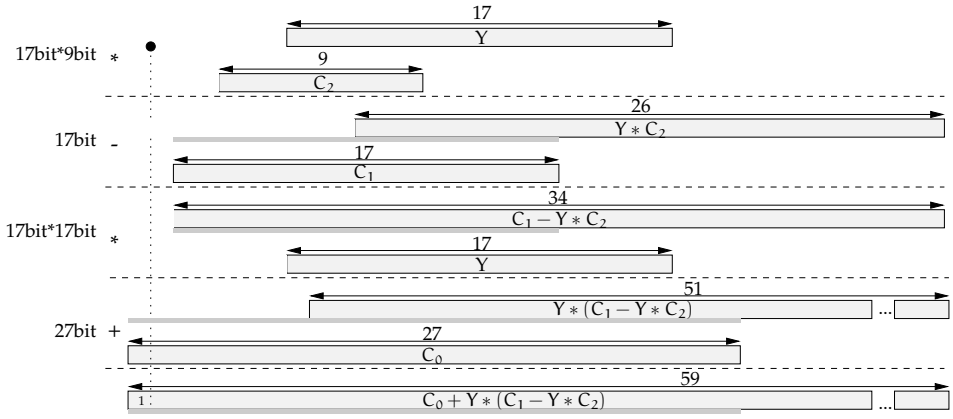

FIG. 1 – Alignement virgule fixe des nombres manipulés par l'évaluation polynomiale.

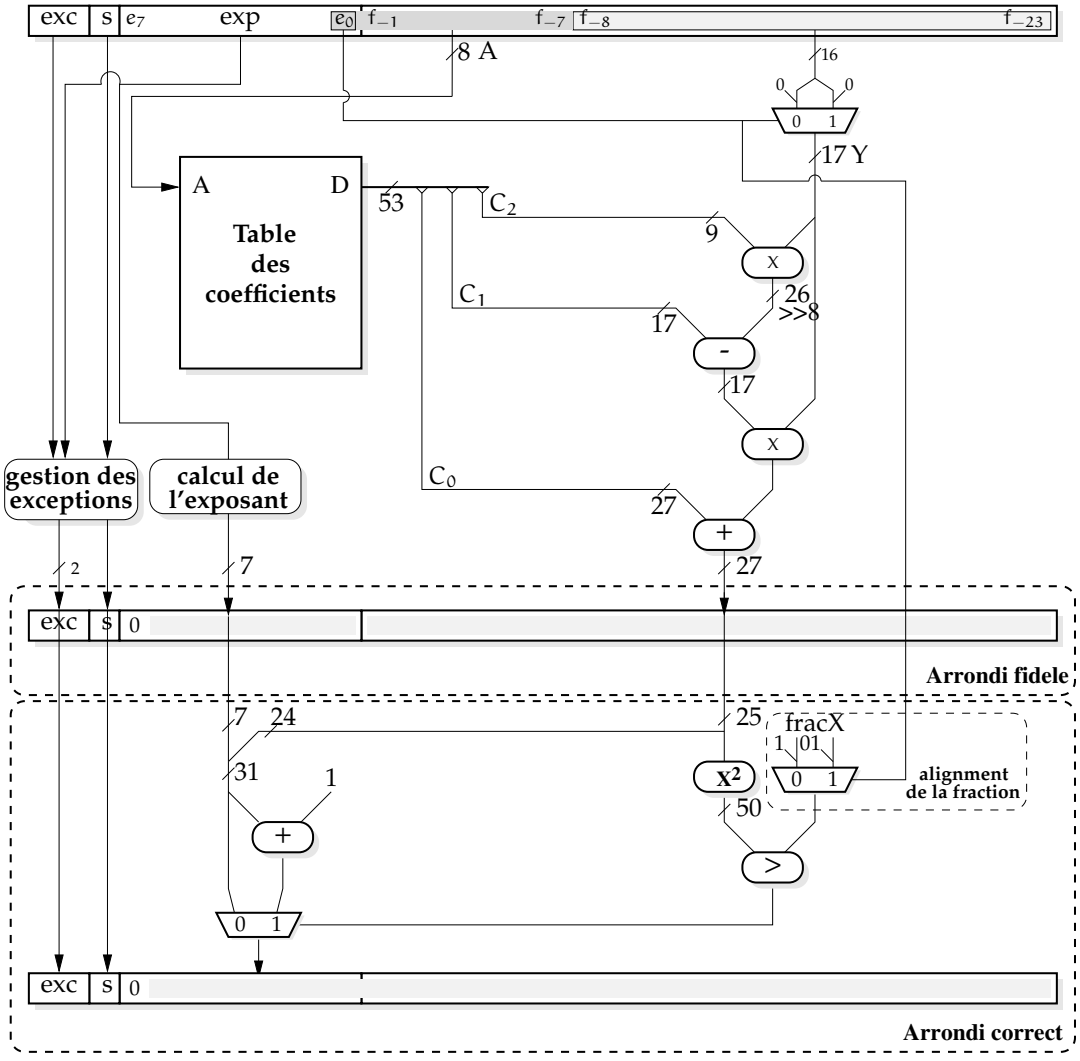

FIG. 2 – Architecture de la racine carrée simple précision.

bits de poids fort de leur représentation en virgule fixe[6]. Les bits de poids faible ont les poids respectifs 27, 18 et 12. Donc, une entrée de la table des polynômes contient $26+17+9 = 52$ bits, ce qui nous donne une taille mémoire de $2^8 \times 52$.
L'architecture correspondante est décrite par la figure 2.
Voici un exemple détaillé du fonctionnement de l'algorithme pour calculer

$$\sqrt{1,00010011001001001010100}.$$

– $e_0 = 0$, nous sommes dans le cas pair
– les 7 premiers bits $f_{-1} \cdots f_{-7} = 0001001$ donnent l'intervalle. L'adresse entrée dans la ROM des coefficients est $A = 00001001$. L'argument réduit est $Y = 01001001001010100$.
– Le polynôme est $p_9(y) = 1,000010001101100011011101 01 + 0,1111011011100101y - 0,0001110011y^2$, pour $y \in [0, 1/128)$.

4. Arrondi fidèle et arrondi correct

4.1. Arrondi fidèle

Comme à la section 2, on note $\tau(y)$ la valeur exacte de la racine carrée sur un des intervalles considérés et $p(y)$ son polynôme d'approximation. L'erreur d'approximation correspondante est bornée finement par Sollya, et on fait en sorte d'obtenir

$$\epsilon_{approx} = |\tau(y) - p(y)| < 2^{-w_F-2}. \tag{1}$$

Soit r la valeur calculée par le pipeline avant l'arrondi final. r est représenté sur $w_F + g$ bits, qui est le max de la taille de c_0 et de la taille à laquelle on a tronqué $y(c_1 + \cdots)$.
L'erreur d'arrondi (différence entre r et $p(y)$) peut être rendue aussi petite que l'on veut : à l'extrême, si l'on ne fait aucune troncature des résultats intermédiaires, elle sera nulle. En pratique, on tronque les résultats intermédiaires pour réduire les entrées des multiplieurs, mais on vérifie au moyen de Gappa que

$$\epsilon_{trunc} = |r - p(y)| < 2^{-w_F-2}. \tag{2}$$

Il reste encore à arrondir r à une valeur sur w_F bits. Cela peut se faire par une simple troncature de r à w_F bits, à condition d'avoir augmenté au préalable chaque coefficient c_0 de la valeur 2^{-w_F-1}, ce qui ne coûte rien. En effet cela revient à utliser $\lfloor z \rceil = \lfloor z + 1/2 \rfloor$. Ainsi, cet arrondi final réalise une erreur bornée par

$$\epsilon_{final} =< 2^{-w_F-1}. \tag{3}$$

La somme de ces trois erreurs est inférieure à 2^{-w_F}, ce qui garantit l'arrondi fidèle.

4.2. Arrondi correct

L'approche utilisée pour obtenir l'arrondi correct est identique à celle présentée dans [6]. Elle utilise l'observation suivante : on sait obtenir une approximation fidèle $\widetilde{r}$ de $\sqrt{x}$ sur $w_F + 1$ bits. Une telle valeur est soit un flottant, soit le milieu entre deux flottants consécutifs. On vérifie facilement qu'on peut en déduire l'arrondi correct sur w_F bits par l'algorithme suivant :

$$\circ(\widetilde{r}) = \begin{cases} \widetilde{r} \text{ tronqué sur } w_F \text{ bits} & \text{si } \widetilde{r} \geq \texttt{sqrtx}, \\ \widetilde{r} + 2^{-w_F-1} \text{ tronqué sur } w_F \text{ bits} & \text{sinon.} \end{cases} \tag{4}$$

Naturellement on implémente le test "si $\widetilde{r} \geq$ sqrtx" en élevant $\widetilde{r}$ au carré et en le comparant à x. Sur FPGA, contrairement à ce qui est fait dans [6], $\widetilde{r}^2$ peut facilement être calculé exactement. Pour ce faire, on utilise l'opérateur de mise au carré de [3], qui permet de calculer le carré au moyen de 3 multiplieurs au lieu de 4 en exploitant la symétrie des produits partiels dans un carré. On pourrait en fait encore

[6] Ces valeurs des poids forts des coefficients sont sans doute une propriété de la fonction $\sqrt{x}$, indépendamment de la précision w_F, de k et du degré d, mais cela reste à vérifier pour de grands degrés.

économiser un de ces trois multiplieurs qui n'est utilisé que pour calculer les bits de poids forts du carré. Ces derniers sont déjà connus puisqu'ils seront égaux à ceux de x. Cette optimisation n'a pas encore été réalisée pour des raisons de réutilisation de code.
Voici à présent les détails. On a besoin d'une erreur d'approximation légèrement meilleure que dans le cas fidèle :

$$\epsilon_{approx} = |\tau(y) - p(y)| < 2^{-w_F-3}. \tag{5}$$

On calcule une valeur r qui approche $\tau(y)$ par au-dessus : il suffit alors d'avoir

$$-2^{-w_F-1} < \tau(y) - r \leq 0 \quad \text{et donc} \quad |(\tau(y) + 2^{-w_F-2}) - r| < 2^{-w_F-2}. \tag{6}$$

Par inégalité triangulaire, on a

$$|(\tau(y) + 2^{-w_F-2}) - r| \leq \underbrace{|(\tau(y) + 2^{-w_F-2}) - (p(y) + 2^{-w_F-2})|}_{<2^{-w_F-3} \text{ d'après (5)}} + |(p(y) + 2^{-w_F-2}) - r|. \tag{7}$$

D'après (5), (6) et (7), il suffit que l'erreur d'évaluation vérifie

$$|(p(y) + 2^{-w_F-2}) - r| < 2^{-w_F-3}. \tag{8}$$

En simple précision, on aura $w_F = 23$ et donc

$$|\tau(y) - p(y)| < 2^{-26} \quad \text{et} \quad |(p(y) + 2^{-25}) - r| < 2^{-26}.$$

On calcule donc une valeur r, qui vérifie $|(p(y) + 2^{-25}) - r| < 2^{-26}$ sur chaque intervalle. On tronque ensuite r sur $w_F + 1$ bits (ici 24 bits), pour obtenir une valeur $\widetilde{r}$:

$$0 \leq r - \widetilde{r} < 2^{-w_F-1} \quad \text{et donc} \quad -2^{-w_F-1} < \tau(y) - \widetilde{r} < 2^{-w_F-1}. \tag{9}$$

Donc $\widetilde{r}$ est bien une approximation fidèle sur $w_F + 1$ bits.

5. Resultats, comparaisons et discussion

Les tables 1 et 2 résument les résultats obtenus pour différents opérateurs de racine carrée. Voici quelques informations supplémentaires sur les données présentées dans ces tables.

- Une précision de 0.5 ulp *(unit in the last place)* correspond à l'arrondi correct, une précision de 1 ulp correspond à l'arrondi fidèle, et une précision plus grande qu'un ulp ne devrait plus être considérée comme acceptable à l'heure actuelle.
- Les chiffres d'Altera tirés de [7] sont, selon leur auteur, préliminaires. Ils concernent une fonction différente mais proche ($1/\sqrt{x}$), et une architecture cible différente (mais dont les multiplieurs sont comparables). Nous pensons toutefois qu'ils ont leur place ici.
- Pour la double précision, les données sont seulement estimées pour l'approximation polynomiale par morceaux. Les polynômes ont été obtenus et l'architecture en virgule fixe construite sur papier. On a $k = 12$ et $d = 4$, soit 2048 polynomes de degré 4. Les coefficients sont sur 12, 23, 34, 45, 56 bits, soit au total 170 bits/polynôme. Ces polynômes seront stockés dans 20 mémoires configurées en 2048x9 bits. L'argument réduit est sur 43 bits. À partir de ces données, les différents multiplieurs et additionneurs ont été synthéthisés séparément dans FloPoCo. Cela donne une estimation assez grossière des ressources consommées. Elle n'est pas très encourageante à aller plus loin.
- Les lignes VFLOAT sont copiées de [14], qui donne des résultats pour Virtex-II. La fréquence est extrapolée pour le Virtex-4.
- Tous les autres résultats sont obtenus sur Virtex-4 xc4vlx15-12 en utilisant ISE 10.1.

Voici à présent quelques commentaires sur ces résultats.

5.1. Les récurrences de chiffre

Une première remarque est que les approches par récurrence de chiffre sont dans un mouchoir de poche. L'implantation de FloPoCo construit un modèle du délai d'une ou plusieurs itérations successives groupées, et utilise ce modèle pour construire automatiquement un pipeline minimal pour une fréquence

outil	précision	latence	fréquence	*slices*	DSP	BRAM
FloPoCo @ 350MHz (SRT)	0.5 ulp	26 cycles	353 MHz	412	0	0
FloPoCo @ 200MHz (SRT)	0.5 ulp	12 cycles	219 MHz	328	0	0
CoreGen (SRT)	0.5 ulp	28 cycles	353 MHz	464	0	0
FPLibrary (SRT)	0.5 ulp	15 cycles	219 MHz	345	0	0
VFLOAT	> 2 ulp	9 cycles	>300 MHz	351	9	3
FloPoCo_Poly_Fidèle	1 ulp	5 cycles	339 MHz	79	2	2
FloPoCo_Poly_Correct	0.5 ulp	12 cycles	237 MHz	241	5	2
Altera ($1/\sqrt{x}$) [7]	?	19 cycles	?	350 ALM	11	?

TAB. 1 – Simple précision : performance des différents opérateurs.

outil	précision	latence	fréquence	*slices*	DSP	BRAM
FloPoCo @ 300MHz(SRT)	0.5 ulp	53 cycles	307 MHz	1740	0	0
FloPoCo @ 200MHz (SRT)	0.5 ulp	40 cycles	206 MHz	1617	0	0
CoreGen (SRT)	0.5 ulp	57 cycles	334 MHz	2061	0	0
FPLibrary (SRT)	0.5 ulp	29 cycles	148 MHz	1352	0	0
VFLOAT	> 2 ulp	17 cycles	>200 MHz	1572	24	116
FloPoCo_Poly_Fidèle	1 ulp	*25 cycles*	*340 MHz*	*2700*	*24*	*20*
Altera ($1/\sqrt{x}$) [7]	?	32 cycles	?	900 ALM	27	?

TAB. 2 – double précision : performance des différents opérateurs. Les nombres en italique sont des estimations.

demandée par l'utilisateur au moyen de l'option `-frequency` [2]. Les itérations font de plus en plus de travail, donc on peut les grouper plus au début du calcul. Le tableau donne des résultats pour une fréquence demandée de 200, 300 ou 350 MHz. CoreGen construit un étage de pipeline par itération, ce qui explique sa consommation supérieure de ressources et de cycles. Par contre, il obtient une meilleure fréquence en double précision (FloPoCo ne parvient pas à dépasser 307MHz), sans doute par une meilleure adaptation de l'algorithme à la structure fine du FPGA.
FPLibrary utilise exactement le même algorithme que FloPoCo et CoreGen, mais groupe les itérations par deux.

5.2. Faut-il utiliser les multiplieurs ?

La question se pose ensuite de la pertinence d'utiliser les multiplieurs pour calculer la racine carrée.
Considérons d'abord la simple précision. Notre approche semble la plus performante de la littérature, et si l'arrondi correct n'est pas critique, elle offre, pour la simple précision, un compromis intéressant : presque pas de logique et une latence très courte, pour une consommation minime de multiplieurs et de mémoires. Un autre avantage, avancé par Langhammer [7], est que l'utilisation de ces multiplieurs permet aussi d'obtenir une performance relativement prédictible, indépendamment de la charge du FPGA et de la difficulté du placement/routage.
On constate toutefois que le surcoût de l'arrondi correct est exorbitant. Non seulement le nombre de multiplieurs double[7] mais en plus, la consommation de logique *slices* explose également pour se rapprocher de celle de la version par récurrence de chiffre. La chûte de fréquence est due à la précision supérieure demandée à `r` : on doit calculer une multiplication 19x17 qui n'est pas synthétisée de façon optimale. Ce problème trouvera certainement une solution, par exemple en prenant $k = 9$. Toujours

[7] Comme déjà mentionné, nous espérons ramener le nombre de multiplieurs à seulement 4 pour la version avec arrondi correct, mais ce n'est pas encore implémenté.

est-il que, à fonctionnalité équivalente (arrondi correct), l'architecture historique à récurrence de chiffres se défend honorablement.
En double précision, l'utilisation des multiplieurs est encore moins convaincante, puisqu'elle consomme plus de ressources que l'architecture historique, y compris les *slices* ! Il ne reste que l'argument d'un pipeline deux fois plus court. L'approche de VFLOAT [14] n'est pas vraiment plus convaincante que la nôtre (sauf son pipeline encore plus court), mais peut-être qu'une optimisation fine de ses chemins de calcul pourrait l'améliorer.

5.3. Comparaison avec Newton-Raphson

Revenons à présent à une discussion sur la pertinence de l'approche polynomiale comparée à l'itération de Newton-Raphson. Cette itération converge vers une racine de $f(y) = 1/y^2 - x$ par la récurrence

$$y_{n+1} = y_n(3 - xy_n^2)/2. \tag{10}$$

La convergence vers $1/\sqrt{x}$ est assurée dès lors que $y_0 \in (0, \sqrt{3}/\sqrt{x})$. La convergence est quadratique (le nombre de bits corrects du résultat double à chaque itération), et on a donc intérêt à lire un y_0 précis à environ k bits dans une table indicée par les k bits de poids fort de la mantisse, comme dans notre approche. Enfin, pour obtenir une approximation $\sqrt{x}$, il reste à multiplier $y_n \approx 1/\sqrt{x}$ par x. On obtient ainsi facilement un arrondi fidèle, et il faut encore un peu de calcul pour obtenir l'arrondi correct, comme dans notre approche.
On peut déjà noter qu'en simple précision, la seule multiplication finale par x consommera 4 multiplieurs 18x18, puisqu'il s'agit d'une multiplication 24x24. Comme notre approche n'utilise que deux multiplieurs en tout, elle est d'ores et déjà meilleure pour la précision simple.
Voyons donc la double précision. On peut utiliser 16 blocs mémoire, configurés en ROM $2^{14} \times 18$ bits, pour partir d'une approximation initiale précise à 14 bits, stockée sur 17 bits. Alors, deux itérations suffisent pour obtenir environ 56 bits corrects.
- Le nombre de multiplieurs dans la première itération est
  - 3 pour xy_0, une multiplication de 53x17 bits
  - et encore 3 pour multiplier le produit précédent (tronqué à 54 bits) par y_0 pour obtenir xy_0^2
  - et encore 3 pour la dernière multiplication par y_0
- y_1 peut être tronqué à 34 bits, et la seconde itération coûte alors
  - 6 multiplieurs pour xy_1
  - encore 6 pour obtenir xy_1^2
  - et encore 6 pour la dernière multiplication par y_1.

Au total, nous estimons donc que l'approximation de $1/\sqrt{x}$ coûte 27 multiplieurs. Il s'agit d'une étude très grossière, mais elle colle exactement avec les 27 multiplieurs mentionnés pour l'implémentation d'Altera [7] dans le tableau 2.
Les deux implémentations, approximation polynomiale par morceau et Newton-Raphson, sont donc au coude à coude pour la double précision, avec des consommations de mémoire et de multiplieurs très comparables. Il n'en reste pas moins qu'il paraît préférable, pour la double précision, de s'en tenir à la récurrence de chiffre.

6. Conclusion et travaux futurs

Cet article est une discussion sur la meilleure manière de calculer une racine carrée sur un FPGA moderne. Il présente au passage une architecture originale à base d'évaluation polynomiale par morceaux qui, pour la simple précision, gagne de la latence et des *slices* au prix d'un demi-bit de précision, de deux multiplieurs et de deux blocs mémoire.
Les techniques utilisées resteront pertinentes pour des mantisses un peu plus petites ou un peu plus grandes, mais pas jusqu'à la double précision, pour laquelle l'algorithme par récurrence classique semble encore indétrônable. Il est surprenant de constater combien il est difficile d'offrir une amélioration nette en termes de performances pour la racine carrée par l'utilisation des multiplieurs et des blocs mémoire. Nous ne prétendons pas avoir exploré exhaustivement tous les algorithmes multiplicatifs possibles, mais aucun de ceux qui sont connus ne s'impose de manière décisive.
L'expérience acquise lors du développement de cette racine carrée va être utilisée pour construire un

générateur d'approximations polynomiales générique pour le projet FloPoCo. Ce générateur permettra de mener à bien l'étude quantitative de la pertinence de l'approche polynomiale par morceaux pour des précisions élevées. Il permettra aussi de négocier entre plus de mémoire ou plus de multiplieurs. Surtout, il pourra être utilisé pour de nombreuses autres fonctions élémentaires. Il permettra également de mener la même étude que nous venons de mener, mais pour la division.

Pour en revenir à la racine carrée, avec ces techniques multiplicatives, corriger un résultat déjà précis au bit près pour obtenir l'arrondi correct coûte aussi, voire plus cher que tout le reste du calcul. Un objectif actuel est de réduire ce surcoût.

En tout état de cause, en oubliant l'arrondi correct, une racine carrée à la précision bien maîtrisée est un bloc de base utile dans FloPoCo. En effet, l'objectif de ce projet n'est pas de concevoir les opérateurs comme la racine carrée, qu'on trouve dans les processeurs, mais des opérateurs plus exotiques pour lesquels le FPGA a un vrai potentiel d'accéleration. Prenons l'exemple de la norme d'un vecteur flottant $\sqrt{x^2+y^2+z^2}$. Nous avons montré dans [2] que la somme de carrés pouvait être optimisée de nombreuses manières. Avec en plus une racine carrée fidèle et une analyse d'erreur fine, nous saurons construire un opérateur de norme lui-même fidèle — c'est-à-dire bien plus précis que l'assemblage d'opérateurs flottants de bibliothèques — et ce à coût bien moindre que cet assemblage. C'est ce type d'approche de la virgule flottante qui bénéficie de la flexibilité des FPGA.

Remerciements

Merci à Jean-Michel Muller pour les discussions sur les sujets abordés dans cet article, et à Claude-Pierre Jeannerod pour ses encouragements, son aide et sa relecture approfondie. Ces recherches sont en parties financées par le projet ANR EVAFlo.

Bibliographie

1. M. Cornea, J. Harrison, and P. T. P. Tang. *Scientific Computing on Itanium®-based Systems*. Intel Press, 2002.
2. F. de Dinechin, C. Klein, and B. Pasca. Generating high-performance custom floating-point pipelines. In *Field Programmable Logic and Applications*. IEEE, Aug. 2009.
3. F. de Dinechin and B. Pasca. Large multipliers with fewer DSP blocks. In *Field Programmable Logic and Applications*. IEEE, Aug. 2009.
4. J. Detrey and F. de Dinechin. A tool for unbiased comparison between logarithmic and floating-point arithmetic. *Journal of VLSI Signal Processing*, 49(1) :161–175, 2007.
5. M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2003.
6. C.-P. Jeannerod, H. Knochel, C. Monat, and G. Revy. Faster floating-point square root for integer processors. In *IEEE Symposium on Industrial Embedded Systems (SIES'07)*, 2007.
7. M. Langhammer. Foundation for FPGA acceleration. In *Fourth Annual Reconfigurable Systems Summer Institute*, 2008.
8. B. Lee and N. Burgess. Parameterisable floating-point operators on FPGAs. In *36th Asilomar Conference on Signals, Systems, and Computers*, pages 1064–1068, 2002.
9. Y. Li and W. Chu. Implementation of single precision floating point square root on FPGAs. In *FPGAs for Custom Computing Machines*, pages 56–65. IEEE, 1997.
10. P. Markstein. *IA-64 and Elementary Functions : Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000.
11. G. Melquiond. *De l'arithmétique d'intervalles à la certification de programmes*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, 2006.
12. J. A. Pineiro and J. D. Bruguera. High-speed double-precision computation of reciprocal, division, square root, and inverse square root. *IEEE Transactions on Computers*, 51(12) :1377–1388, Dec. 2002.
13. D. M. Russinoff. A mechanically checked proof of correctness of the AMD K5 floating point square root microcode. *Formal Methods in System Design*, 14(1) :75–125, 1999.
14. X. Wang, S. Braganza, and M. Leeser. Advanced components in the variable precision floating-point library. In *FCCM*, pages 249–258. IEEE Computer Society, 2006.