

Racines carrées multiplicatives sur FPGA

Florent De Dinechin, Mioara Maria Joldes, Bogdan Pasca, Guillaume Revy

► **To cite this version:**

Florent De Dinechin, Mioara Maria Joldes, Bogdan Pasca, Guillaume Revy. Racines carrées multiplicatives sur FPGA. RRLIP2009-19. 10 pages. 2009. <ensl-00388064>

HAL Id: ensl-00388064

<https://hal-ens-lyon.archives-ouvertes.fr/ensl-00388064>

Submitted on 26 May 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Racines carrées multiplicatives sur FPGA

Rapport de Recherche du LIP RR2009-19

Florent de Dinechin, Mioara Joldes, Bogdan Pasca, Guillaume Revy
LIP (CNRS/INRIA/ENS-Lyon/UCBL)
École Normale Supérieure de Lyon — Université de Lyon

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 particulièrement 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 compatible IEEE-754 en utilisant ces nouvelles ressources, et compare les performances obtenues avec l'approche classique.

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 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'itéra-

tion de Newton-Raphson, utilisées dans les processeurs IA32 d'AMD à partir du K5 [12] et dans les processeurs dont l'unité flottante est centrée autour d'un multiplieur-additionneur fusionné (FMA), comme les familles Power/PowerPC et Itanium [10, 1]. On y trouve aussi des approches à base d'approximation polynomiale par morceaux [13, 6] ou un mélange des deux [11]. 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 utilisés dans [6].

À présent que tous les FPGA haut-de-gamme intègrent de petits multiplieurs (jusqu'à 2000 multiplieurs typiquement capables de multiplier 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 considérant 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 FloPoCo¹, un générateur libre de cœurs arithmétiques [2] pour le calcul haute performance sur FPGAs. 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. 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 Leiser dans le projet VFLOAT [13]. Leur algorithme n'est pas une approximation polynomiale par morceaux, mais une réduction d'argument à base de tables suivi d'une approximation polynomiale pour l'argument réduit. En pratique les deux approches se ressemblent beaucoup.

L'utilisation d'outils récents d'approximation polynomiale permet des améliorations significatives par rapport à [13] :

- 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 [13]).
- 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 cœurs arithmétiques centrés sur les multiplieurs. Cet article mentionne un opérateur de racine carrée inverse (il s'agit donc 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<http://www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/>

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 signée², 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, et il paraît raisonnable de chercher à équilibrer le nombre de mémoires et le nombre de multiplieurs consommés par un opérateur donné.

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 relatif³ et f est la partie fractionnaire de la mantisse, écrite en binaire sur w_F bits $f_{-1}f_{-2} \dots f_{-w_F}$ (les indices des bits dénotent leur poids).

Il y a classiquement deux cas à considérer.

Si e est pair, la racine carrée s'écrit

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

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

³L'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

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 Sollya⁴.

Une première idée est d'utiliser, pour adresser cette table, les bits de poids fort de z . Toutefois, comme l'intervalle $[1, 4[$ est de longueur 3, 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} y varie plus). 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 FP-GAs 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 $f_{-1}f_{-2} \dots f_{-k+1}$ de la mantisse de x . Sur chacun de ces intervalles, on approche $\tau_{\text{pair}}(1 + \frac{i}{2^{k-1}} + y)$ par un polynôme de degré d : $p_i(y) = c_{0,i} + c_{1,i}y + \dots + c_{d,i}y^d$. 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 $f_{-1}f_{-2} \dots f_{-k+1}$ 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 $A = e_0f_{-1}f_{-2} \dots f_{-k+1}$ (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 $1, f_{-1} \dots f_{-w_F} = 1 + 0, f_{-1} \dots f_{-k+1} + 2^{-k+1}0, f_{-k} \dots f_{-w_F}$.
- Dans le cas impair on calcule la racine carrée de $2 \times 1, f = 1f_{-1}, f_{-2} \dots f_{-w_F} = 1 + f_{-1}, f_{-2} \dots f_{-k+1} + 2^{-k+2}0, f_{-k} \dots f_{-w_F}$.

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 $y = 2^{-k+2} \times 0, 0f_{-8} \dots f_{-w_F}$ dans le cas pair et $y = 2^{-k+2} \times 0, f_{-k} \dots f_{-w_F}0$ dans le cas impair.

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.

⁴<http://sollya.gforge.inria.fr/>

- 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, sachant que Sollya calculera les coefficients exactement.
- 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 17×17 bits consomme un multiplieur, mais une multiplication de 19×19 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 :

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

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 $a_i + y \times (\dots)$. On cherche donc à minimiser les tailles des coefficients.

Par ailleurs, dans l'addition $a_i + y \times (\dots)$, le produit $y \times (\dots)$ 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 troncatures 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 Gappa⁵. 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 $f_{-8} \dots f_{-23}$, plus un 0 à droite ou à gauche).

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 bits de poids fort de leur représentation en virgule fixe⁶. 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

$$\sqrt{1.00010011001001001010100.}$$

⁵<http://lipforge.ens-lyon.fr/www/gappa/>

⁶Ces 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.

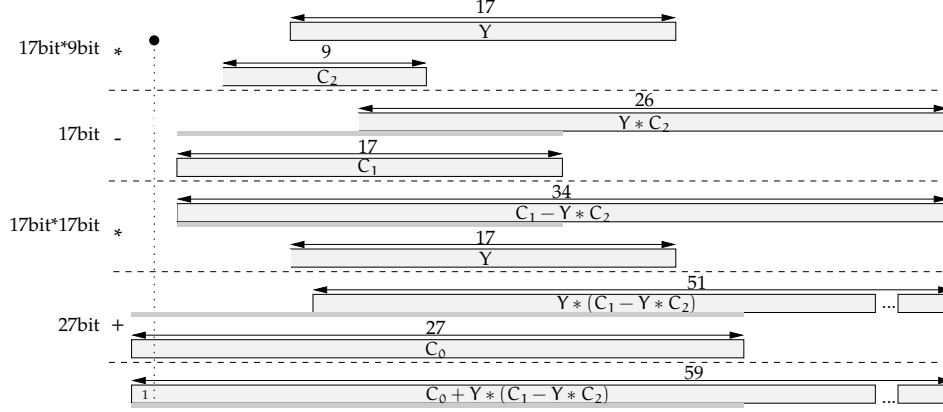


FIG. 1 – Alignement virgule fixe des nombres manipulés par l'évaluation polynomiale

- $e_0 = 0$, nous sommes dans le cas pair
- les premiers 7 bits $f_{-1}...f_{-7} = 0001001$ donnent l'intervalle. L'adresse entrée dans la ROM des coefficients est $A = 00001001$. L'argument réduit est $Y = 01001001001010100$.
- Le polynôme est $p_9(y) = 1.00001000110110001101110101 + 0.11110111011100101y - 0.0001110011y^2$, pour $y \in [0, 1/128]$.

multiplieurs, mais on vérifie au moyen de Gappa que

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

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 utiliser $\lfloor z \rfloor = \lfloor z + 1/2 \rfloor$. Ainsi, cet arrondi final réalise une erreur bornée par

$$\epsilon_{\text{final}} = 2^{-w_F-1}. \quad (3)$$

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

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

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

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 $(c_1 + \dots)$.

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 aux

4.2 Arrondi correct

L'approche utilisée pour obtenir l'arrondi correct est identique à celle présentée dans [6]. Elle consiste à calculer une approximation de \sqrt{x} assez précise, à la tronquer à $w_F + 1$ bits, puis à l'élever au carré. La comparaison avec x permet alors de décider si cette approximation est la bonne, ou s'il faut l'augmenter.

Voici les détails. On a besoin d'une erreur d'approximation légèrement meilleure que dans le cas fidèle :

$$\epsilon_{\text{approx}} = |\tau(y) - p(y)| < 2^{-w_F-3}. \quad (4)$$

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

$$2^{-w_F-1} < \tau(y) - r \leq 0 \quad \text{et donc} \\ |(\tau(y) + 2^{-w_F-2}) - r| < 2^{-w_F-2}. \quad (5)$$

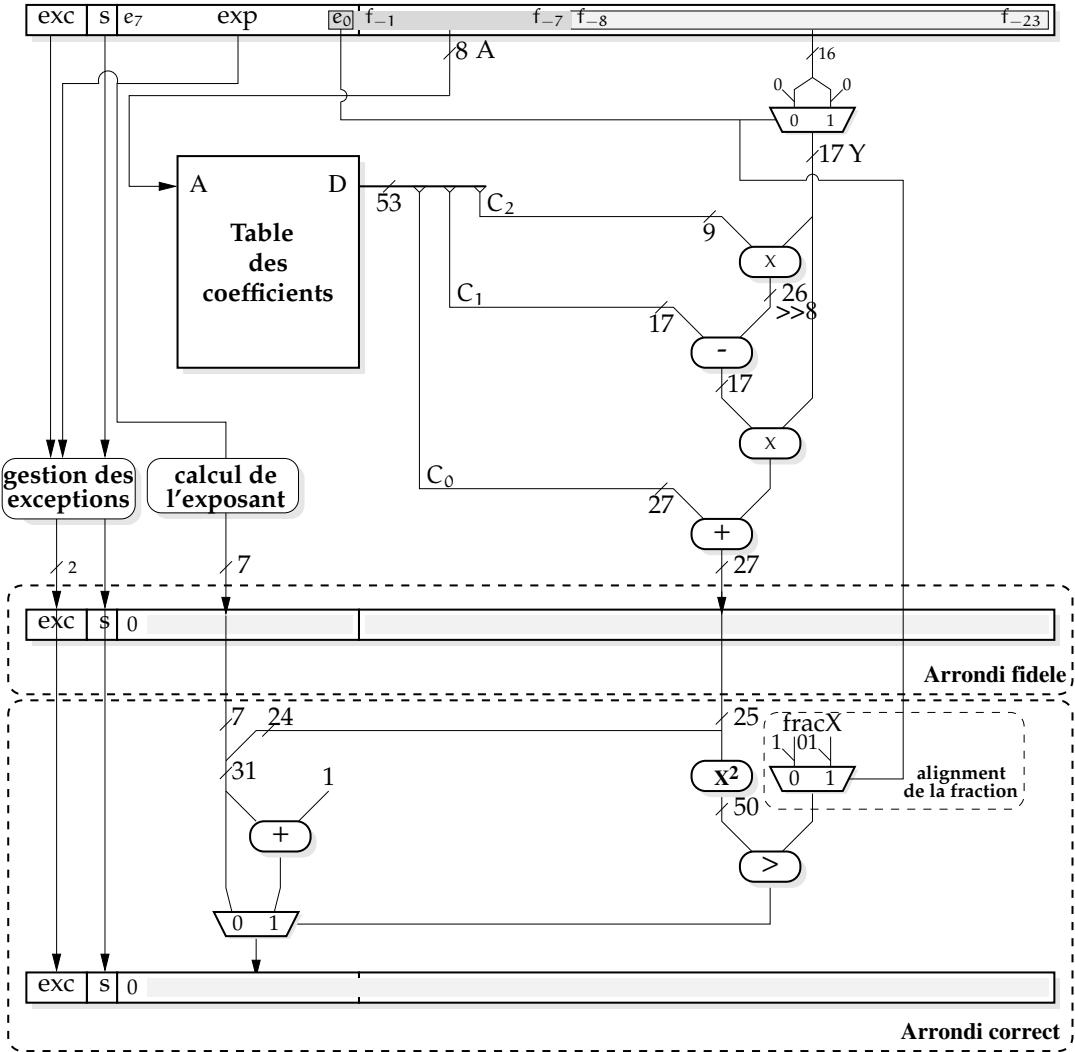


FIG. 2 – Architecture de la racine carrée simple précision

Par inégalité triangulaire, on a

$$\begin{aligned} |(\tau(y) + 2^{-w_F-2}) - r| &\leq \\ \underbrace{|(\tau(y) + 2^{-w_F-2}) - (p(y) + 2^{-w_F-2})|}_{< 2^{-w_F-3} \text{ d'après (4)}} &+ \\ |(p(y) + 2^{-w_F-2}) - r|. &\quad (6) \end{aligned}$$

D'après (4), (5) et (6), pour garantir l'arrondi correct, il suffit que l'erreur d'évaluation vérifie

$$|(p(y) + 2^{-w_F-2}) - r| < 2^{-w_F-3}. \quad (7)$$

En simple précision, on aura $w_F = 23$ et donc $|\tau(y) - p(y)| < 2^{-26}$ et $|(p(y) + 2^{-25}) - r| < 2^{-26}$.

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) : on note \tilde{r} le résultat obtenu

$$\begin{aligned} 0 \leq r - \tilde{r} &< 2^{-w_F-1} \quad \text{et donc} \\ 2^{-w_F-1} &< \tau(y) - \tilde{r} < 2^{-w_F-1}. \end{aligned} \quad (8)$$

De la même manière que dans [6], l'algorithme d'arrondi correct est alors

$$o(\tilde{r}) = \begin{cases} \tilde{r} \text{ tronqué sur } w_F \text{ bits} & \text{si } \tilde{r} \geq \tau(y), \\ \tilde{r} + 2^{-w_F-1} \text{ tronqué sur } w_F \text{ bits} & \text{sinon.} \end{cases} \quad (9)$$

L'implantation de ce test peut donc se faire en élevant \tilde{r} au carré, et en le comparant à $1.f$ ou $2 \times 1.f$ (suivant la parité de l'exposant e). Par contre sur FPGA, contrairement à ce qui est fait dans [6], \tilde{r}^2 peut 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 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 x . Cette optimisation n'a pas encore été réalisée pour des raisons de réutilisation de code. Ce sera testé avant publication.

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 polynômes 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 synthétisés 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 [13], 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. Nous n'avons pas encore d'explication pour l'avantage considérable de CoreGen sur FloPoCo pour la double. Cela peut être simplement une meilleure exploitation de la structure fine du FPGA, ou un changement d'algorithme. Nous n'expliquons pas plus le léger avantage de FloPoCo pour la simple précision.

FPLibrary utilise exactement le même algorithme que FloPoCo et CoreGen, mais groupe les itérations par deux : cela diminue latence et fréquence, et réduit aussi la consommation de ressources. À très court terme, FloPoCo devrait offrir cette option aussi. En fait, il faut construire un modèle du délai d'une ou plusieurs itérations successives groupées, et utiliser ce modèle pour construire automatiquement un pipeline minimal pour une fréquence demandée par l'utilisateur [2]. Les itérations font de plus en plus de travail, donc on peut sans doute les grouper plus au début du calcul. C'est peut-être ce que fait CoreGen pour la double-précision, les dernières itérations étant, au contraire, coupées en plusieurs étages de pipeline.

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. Il est clair que notre approche est 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 minimale 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

outil	précision	latence	fréquence	<i>slices</i>	DSP	BRAM
FloPoCo (SRT)	0.5 ulp	26 cycles	353 MHz	416	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	<i>slices</i>	DSP	BRAM
FloPoCo (SRT)	0.5 ulp	55 cycles	307 MHz	1755	0	0
CoreGen (SRT)	0.5 ulp	57 cycles	334 MHz	1104	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	<i>25 cycles</i>	<i>340 MHz</i>	<i>2700</i>	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.

correct est exorbitant. Non seulement le nombre de multiplieurs double⁷ mais en plus, la consommation de logique *slices* explose également pour se rapprocher de celle de la version par récurrence de chiffre. La chute de fréquence est due à la précision supérieure demandée à r : on doit calculer une multiplication 19×17 qui n'est pas synthétisée optimalement. Ce problème trouvera certainement une solution, par exemple en prenant $k = 9$. Toujours 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 [13] 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.

⁷Comme 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.

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 solution de $f(y) = 1/y^2 - x$ par la récurrence

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

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.

Cherchons à présent à adapter cette itération à nos FPGA pour la simple précision. Notons d'abord qu'en simple précision, cette seule multiplication finale par x consommera 4 multiplieurs 18×18 , puisqu'il s'agit d'une multiplication 24×24 . Comme notre approche n'utilise que deux multiplieurs en tout, elle est d'ores et déjà meilleure pour la préci-

sion 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 53×17 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 et 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, le vrai 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'accélération. 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 que doit être la virgule flottante sur FPGA.

Remerciements

Merci à Claude-Pierre Jeannerod et à Jean-Michel Muller pour les discussions sur les sujets abordés dans cet article.

Références

- [1] Marius Cornea, John Harrison, and Ping Tak Peter Tang. *Scientific Computing on Itanium-based Systems*. Intel Press, 2002.

- [2] Florent de Dinechin, Cristian Klein, and Bogdan Pasca. Generating high-performance custom floating-point pipelines. Technical Report 2009-16, LIP, École Normale Supérieure de Lyon, 2009.
- [3] Florent de Dinechin and Bogdan Pasca. Large multipliers with less DSP blocks. Technical Report 2009-03, LIP, École Normale Supérieure de Lyon, 2009.
- [4] Jérémie Detrey and Florent 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] Claude-Pierre Jeannerod, Hervé Knochel, Christophe Monat, and Guillaume Revy. Computing floating-point square roots via bivariate polynomial evaluation. Technical Report ensl-00335792, LIP, INRIA/CNRS/UCBL/ENS-Lyon, 2008. LIP research report RR2008-38.
- [7] Martin 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] Peter Markstein. *IA-64 and Elementary Functions : Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000.
- [11] 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, December 2002.
- [12] 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.
- [13] Xiaojun Wang, Sherman Braganza, and Miriam Leiser. Advanced components in the variable precision floating-point library. In *FCCM*, pages 249–258. IEEE Computer Society, 2006.